Contents

1 /

1.1 add_hooks.sh

```
#!/bin/bash

# write pre-commit config options
/bin/cat << END_CONFIG > .pre-commit-config.yaml
repos:
  - repo: https://github.com/psf/black
    rev: 23.3.0
    hooks:
      - id: black
        language_version: python3.9
        args: [--force-exclude, "maillib.py"]
END_CONFIG

pre-commit install
```

1.2 master.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


import argparse
import importlib
import inspect
import json
import logging
import os
import random
import time
from typing import Any, Optional

import yaml

# General logging level
logging.basicConfig(level=logging.INFO, format="%(levelname)s:%(name)s:%(message)s")
logger = logging.getLogger(__name__)

# Tensorflow logging level
#
     https://stackoverflow.com/questions/35911252/disable-tensorflow-debugging-information/42121886#42121886
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "1"  # any {'0', '1', '2', '3'}


################################################################################
## Main Python Functions
################################################################################
####################
## New MarcoPolo Run
####################
def setup_marcopolo_core(args: argparse.Namespace) -> Any:
    """Set up and return the marco polo core

    Dynamic loading of the core is done based on the value of
    "core" in the input.

    Envs are loaded dynamically when "env" is given.
    Note that env is required when the core supports dynamic
    loading and prohibited when it doesn't. This is not checked
    during loading. There will be errors if env is used incorrectly

    Parameters
    ----------
    args : argparse.Namespace
        Object containing all of the global parameters.

    Side-Effects
    ------------
    * The core module named in args is dynamically imported
    * The random number seed is set from a value in args

    Returns
    -------
    MarcoPolo Core
        The core simulation object
    """
    args.vidpath = setup_video_directory(args)

    # Set master_seed
    #  this is the only way to get reproducibility from neat-python
    random.seed(args.master_seed)

    # Dynamically load the Core Manager
    logger.info("Training Core: marco_polo.cores." + args.core)
    tmp_mod = importlib.import_module("marco_polo.cores." + args.core)
```

```
    Core = tmp_mod.Core

    # if an env is requested, load it and use in core
    req_params = inspect.signature(Core.__init__).parameters

    kwargs = {"args": args}

    if "env_factory" in req_params.keys():
        if hasattr(args, "env"):
            # Dynamically load the Env creation function
            logger.info("Training Environment: marco_polo.envs." + args.env)
            env_mod = importlib.import_module("marco_polo.envs." + args.env)
            # function to return class of envs
            env_class_factory = env_mod.get_env_class
            # function to return class of env params
            env_param_class_factory = env_mod.get_env_param_class

            kwargs["env_factory"] = env_class_factory
            kwargs["env_param_factory"] = env_param_class_factory

        else:
            raise Exception("Core requires 'env' to be specified in the .yaml.")

    if "manager_factory" in req_params.keys():
        if hasattr(args, "optimizer"):
            # Dynamically load the Env creation function
            logger.info("Training Optimizer: marco_polo.optimizers." + args.optimizer)
            env_mod = importlib.import_module("marco_polo.optimizers." + args.optimizer)
            # function to return class of envs
            opt_class_factory = env_mod.get_opt_class
            # function to return class of env params
            model_class_factory = env_mod.get_model

            # Initialize marco polo population (env/team pairs) manager
            kwargs["manager_factory"] = opt_class_factory
            kwargs["model_factory"] = model_class_factory

        else:
            raise Exception("Core requires 'optimizer' to be specified in the .yaml.")

    # no env is requested, return core with env
    return Core(**kwargs)


########################
## Restart MarcoPolo Run
########################
def reload_core_from_checkpoint(
    args: argparse.Namespace,
    preload_args: Optional[dict[str, Any]] = None,
    print_args: bool = True,
) -> tuple[Any, int]:
    """Set up and return the marcopolo core from a checkpoint

    Parameters
    ----------
    args: argparse.Namespace
        Object containing all of the global parameters.
    preload_args: dict[str, Any], optional
        Arguments to apply prior to creating the marcopolo core
        object. This would be used to override any of the args
        in the checkpoint args file. An example would be to change
        the number of workers to create in the manager.
        Default is None which means not to override anything
    print_args: bool, default = True
        Whether to print the new args after reading from
        the checkpoint.

    Side-Effects
    ------------
    * The core module named in args is dynamically imported
    * The random number seed is set from a value in args
    * args is overwritten by the checkpoint data

    Returns
    -------
    tuple[MarcoPolo Core, int]
        MarcoPolo Core: The core simulation object
        The int is the starting epoch number
    """
    cp_name = args.start_from
```

```
    tag = ""
    if "logtag" in args.__dict__:
        tag = args.logtag
    start_from = os.path.join(args.log_file, "Checkpoints", cp_name)

    logger.info(f"Starting from {start_from}")

    with open(os.path.join(start_from, "args.json"), mode="r", encoding="utf8") as file:
        args.__dict__ = json.load(file)
        if len(tag) > 0:
            args.logtag = f"[{tag}:{cp_name}]->"
        else:
            args.logtag = f"[{cp_name}]->"

    # update items before load if requested
    if preload_args is not None:
        for key in preload_args.__dict__:
            # Namespace doesn't support direct assignment
            setattr(args, key, preload_args[key])

    # Initialize marco polo population (env/team pairs) manager
    core = setup_marcopolo_core(args)

    #  Immediately load in stored state
    core.reload(folder=start_from)

    if print_args:
        logger.info(f"Args after reload: {core.args}")

    # Tweak to get the reload epoch
    start_epoch = int(start_from.split("-")[-1]) + 1  # Check point at end of epoch

    logger.info(
        f"############ Restarting From Checkpoint At Epoch {start_epoch} ############\n"
    )

    # Run Evolution
    return core, start_epoch


def get_args(config_file: Optional[str] = None) -> argparse.Namespace:
    """
    Parses command line arguments and input script and return the arguments

    Parameters
    ----------
    config_file: str, optional
        Path to config file to arguments from. If both this and
        --config are given, the file specified by --config is used.
    Returns
    -------
    argparse.Namespace
        object containing all the arguments
    """
    # Init argparser
    parser = argparse.ArgumentParser()

    # MarcoPolo parameters
    parser.add_argument("--log_file", default=None)
    parser.add_argument("--config", default=None)
    parser.add_argument("--core", default="ePoet")
    parser.add_argument("--init", default="random")
    parser.add_argument(
        "--visualize_freq",
        type=int,
        default=0,
        help="Frequency to store gifs of agent behavior. Use 0 for no gifs.",
    )
    parser.add_argument("--frame_skip", type=int, default=10)

    parser.add_argument("--eval_jobs", type=int, default=1)
    parser.add_argument("--rollouts_per_eval_job", type=int, default=50)
    parser.add_argument("--num_workers", type=int, default=20)
    parser.add_argument("--poet_epochs", type=int, default=200)
    parser.add_argument("--master_seed", type=int, default=111)

    parser.add_argument("--mc_lower", type=int, default=200)
    parser.add_argument("--mc_upper", type=int, default=340)
    parser.add_argument("--repro_threshold", type=int, default=200)
    parser.add_argument("--num_proposal_envs", type=int, default=8)
    parser.add_argument("--max_admitted_envs", type=int, default=1)
```

```
    parser.add_argument("--max_active_envs", type=int, default=100)
    parser.add_argument("--num_start_envs", type=int, default=1)

    parser.add_argument("--checkpoint_interval", type=int, default=1)
    parser.add_argument("--checkpoint_compression", default=True)
    parser.add_argument("--reproduction_interval", type=int, default=1)

    parser.add_argument("--start_from", default=None)  # Checkpoint folder to start from
    parser.add_argument("--save_env_params", default=True)
    parser.add_argument("--logtag", default="")
    parser.add_argument("--log_pata_ec", default=True)

    # Parse CMDLine args
    args = parser.parse_args()

    ## Load args from yaml if available
    if args.config is not None:
        # args.config overides the function argument
        config_file = args.config

    if config_file:
        args = read_config_from_file(args, config_file)

    # Log input params
    logger.info(f"{args}\n")

    return args


def read_config_from_file(
    args: argparse.Namespace, filename: str
) -> argparse.Namespace:
    """Read config options from file and update args object

    Pararmeters
    -----------
    args: argparse.Namespace
        the config arguments to update
    filename: str
        path of file to read arguments from

    Returns
    -------
    argparse.Namespace
        The updated args object
    """
    with open(filename, mode="r", encoding="utf-8") as file:
        logger.info(f"Loading config from {filename}\n")
        config = yaml.safe_load(file)

        for key, val in config.items():
            vars(args)[key] = val
    return args


def setup_video_directory(args: argparse.Namespace) -> str:
    """Creates Videos directory, if appropriate, and returns the path

    The path is formed from the log_file path
    Whether the directory is created depends on whether videos will
    be created. If args.visualize_freq is zero, it is assumed
    that videos will not be created and the directory will not be
    created.

    Parameters
    ----------
    args: argparse.Namespace
        the arguments for the run

    Returns
    -------
    str
        path to videos directory. Note that this is always returned
        even if the directory is not created.
    """
    # always create the path string, as it is returned
    vidpath = os.path.join(args.log_file, "Videos")
    # only create the actual directory if it will be used
    if args.visualize_freq != 0:
        logger.info(f"Saving video to {vidpath}\n")
        os.makedirs(name=vidpath, exist_ok=True)
```

```
    else:
        logger.info("Not saving video\n")
    return vidpath


################################################################################
## Main CMDLine Function
################################################################################
def main() -> None:
    """
    Main run function. This function parses cmdline arguments and then calls the
    appropriate function.

    Parameters
    ----------
    None

    Side-Effects
    ------------
    None

    Returns
    -------
    None
        All information is output to disk
    """

    # init start time
    sim_start_time = time.time()

    args = get_args()

    # define log folder when launched from VSCODE
    # maintain transparent compatibility with run_MP.sh launch script
    if "VSCODE_LAUNCHED" in os.environ:
        args.log_file = os.path.join(os.environ["OUTPUT_DIR"], os.environ["RUN_NAME"])

    # new run or restart?
    if args.start_from is not None:
        core, start_epoch = reload_core_from_checkpoint(args=args)
        core.evolve_population(start_epoch=start_epoch, sim_start=sim_start_time)
    else:
        core = setup_marcopolo_core(args=args)
        core.evolve_population(start_epoch=0, sim_start=sim_start_time)


################################################################################
## Run
################################################################################
if __name__ == "__main__":
    main()
```

1.3 run_MP.sh

```
#!/bin/bash
# syntax: either
#     run_MP.sh /path/to/folder config.yaml
# or
#     run_MP.sh /path/to/folder checkpoint_name
#
# script checks for a yaml extension an assumes intent based on this.

if [ -z "$1" ]
then
    echo "Missing an experiment directory"
    exit 1
fi

logDir=$1
mkdir -p $logDir
export PYTHONHASHSEED=0
DATETIME=`date +"%Y-%m-%d_%T"`

if [[ $2 == *.yaml ]]
then
  # No checkpoint, start a new run
  config_file=$2
  base_name=$(basename ${config_file})
  cp $config_file $logDir/$base_name

  echo "Starting new MarcoPolo run in $logDir from $config_file"
  python -u master.py \
    --log_file $logDir \
    --config $config_file \
    2>&1 | tee $logDir/run.$DATETIME.log
else
  echo Restarting from checkpoint
  chkpt=$2
  if [ -d $logDir/Checkpoints/$chkpt ]
  then
    # A checkpoint has been provided, load from that.
    echo "Continuing MarcoPolo run in $logDir at checkpoint $chkpt"
    echo ""
    python -u master.py \
      --log_file $logDir \
      --start_from $chkpt \
      2>&1 | tee $logDir/run.$chkpt.$DATETIME.log
  else
    echo "Checkpoint $logDir/Checkpoints/$chkpt does not exist"
  fi
fi
```

1.4 setup_MP.sh

```
#!/bin/bash

# exit when any command fails
set -e

# script for setting up MarcoPolo conda environment
# use: . setup_MP.sh
#       The space is important, as it keeps it in the main shell
# Output is a conda environment called "marcopolo"

# Ubuntu installs first
apt-get install -y cmake python3-opengl

# Mamba install next
#  Check if mamba is installed, only install if it isn't already there
#   https://stackoverflow.com/questions/592620/how-can-i-check-if-a-program-exists-from-a-bash-script
#  Download
#  Batch install
#  Run init because batch install doesn't
#  Fix base environemt BS
#  Remove setup script
MAMBA_PATH=~/mambaforge/condabin
if ! command -v $MAMBA_PATH/mamba &> /dev/null
then
    # Install Mamba and clean up
    rm -rf /opt/conda
    wget -P ~
         "https://github.com/conda-forge/miniforge/releases/latest/download/Mambaforge-$(uname)-$(uname
         -m).sh"
    bash ~/Mambaforge-$(uname)-$(uname -m).sh -b
    $MAMBA_PATH/mamba init
    $MAMBA_PATH/conda config --set auto_activate_base false
    rm ~/Mambaforge-$(uname)-$(uname -m).sh
fi


# Source for shell
#  This is required for create, activate, deactive below
source ~/mambaforge/etc/profile.d/conda.sh

# Create marcopolo env
#  Pin python version
#  Install required packages
#  "-y" installs without asking, same as bash
mamba create -n marcopolo -y python=3.9 gymnasium-box2d imageio matplotlib \
numpy networkx pandas ipykernel h5py pynng gifsicle lbzip2 black \
numba pre-commit pygame scipy tianshou packaging pytorch tensorflow \
-c conda -c conda-forge -c pytorch

# Pip installs
#  Because they aren't available in conda-forge
conda activate marcopolo
pip install --use-pep517 --no-cache-dir neat-python pettingzoo[classic]
./add_hooks.sh
conda deactivate

# reset failure handling
set +e

# should be good to go!
```

1.5 texify.py

```
#!/usr/bin/env python3

import os, re, argparse

latex_header = r'''
\documentclass{article}
\usepackage[margin=1in]{geometry}
\usepackage{textcomp}
\usepackage{listingsutf8}
\usepackage{hyperref}
\usepackage[dvipsnames]{xcolor}
\definecolor{darkgreen}{rgb}{0,0.5,0}
\definecolor{lightblue}{rgb}{0.2,0.5,1}
\hypersetup{colorlinks=true, linkcolor=blue}
\lstset{
    numbers=left,
    upquote=true,
    breaklines=true,
    tabsize=4,
    showstringspaces=false,
    showspaces=false,
    breakatwhitespace=true,
    <SYNTAX_HIGHLIGHTING>
}
\begin{document}
\tableofcontents
\newpage
'''

styles = {
'default': r'''
    basicstyle=\ttfamily\scriptsize,
    keywordstyle=\ttfamily,
    commentstyle=\ttfamily\color{darkgreen},
    stringstyle=\ttfamily\color{blue},
''',
'dark': r'''
    backgroundcolor=\ttfamily\color{black},
    basicstyle=\ttfamily\color{white}\scriptsize,
    keywordstyle=\ttfamily,
    commentstyle=\ttfamily\color{green},
    stringstyle=\ttfamily\color{lightblue},
''' # xterm-mode
}

# Governs syntax highlighting
file_types = {
    '.py': 'Python',
     '.c': 'C',
     '.d': 'C',
     '.m': 'Matlab',
     '.r': 'R',
     '.sh': 'bash',
     '.bash': 'bash',
     '.cpp': 'C++',
     '.cc': 'C++',
     '.pl': 'Perl',
     '.tex': 'TeX',
     '.f': 'Fortran',
     '.for': 'Fortran',
     '.ftn': 'Fortran',
     '.f90': 'Fortran',
     '.f95': 'Fortran',
     '.f03': 'Fortran',
     '.f08': 'Fortran',
     '.csh': 'csh',
     '.ksh': 'ksh',
     '.lisp': 'lisp',
     '.lsp': 'lisp',
     '.cl': 'lisp',
     '.l': 'lisp',
     '.scm': 'lisp',
     '.go': 'Go',
     '.hs': 'Haskell',
     '.lhs': 'Haskell',
     '.bat': 'command.com',
     '.awk': 'Awk',
}
```

```
79
80  def main() -> None:
81      parser = argparse.ArgumentParser(usage='%s [-d DIR] [-i extension ...]\n' % __file__
82          + 'example: %s -d ./src -i foo.m -i makefile .c .d .py\n\n' % __file__,
83          description='Will search under DIR for all source files with the specified file extensions, and
                compile them into a LaTeX file.')
84      parser.add_argument('--dir', '-d', help='root directory under which to search', default='.')
85      parser.add_argument('--include', '-i', action='append', help="Explicitly include a file even if it
                doesn't match the extension list", default=[])
86      parser.add_argument('--style', default='default', choices=styles.keys(), help='Changes syntax
                highlighting, etc.')
87      parser.add_argument('extension', nargs='+', help='Only files with these extensions will be included
                (leading dot optional)')
88      args = parser.parse_args()
89
90      # Permit valid extensions to be input with or without the dot
91      args.extension = [ a if ('.' == a[0]) else '.%s' % a
92          for a in args.extension ]
93
94      # Make relative to base path, escape underscores
95      def format_path(path: str) -> str:
96          if path == args.dir: return '/'
97          assert (path[0:len(args.dir)+1] == args.dir + '/') or (path[0:len(args.dir)+1] == args.dir +
                '\\')
98          return re.sub('_', r'\_', path[len(args.dir)+1:])
99
100     # Print single file
101     def dumpsrc(dirpath: str, fname: str) -> str:
102         path = '%s/%s' % (dirpath, fname)
103         escaped = format_path(path)
104         print(r'\subsection[%s]{%s}' % (os.path.basename(escaped), escaped))
105         ext = os.path.splitext(f)[1]
106         if ext in file_types:
107             s = r'\lstinputlisting[language=%s]{%s}' % (file_types[ext], path)
108         else:
109             s = r'\lstinputlisting{%s}' % path
110         return '%s\n%s\n' % (s, r'\newpage')
111
112     def print_header() -> None:
113         s = latex_header.replace(r'<SYNTAX_HIGHLIGHTING>', styles[args.style].strip(), 1)
114         print(s.strip())
115
116     print_header()
117
118     dirs = {dirpath:fnames for dirpath, _, fnames in os.walk(args.dir)}
119     includes = { os.path.realpath(f):f for f in args.include }
120     for dirpath in sorted(dirs):
121         fnames = dirs[dirpath]
122         src = sorted([f for f in fnames
123             if (os.path.splitext(f)[1] in args.extension) or (os.path.realpath(f) in includes)])
124         if 0 == len(src): continue
125
126         print(r'\section{%s}' % format_path(dirpath))
127         for f in src:
128             print(dumpsrc(dirpath, f))
129
130             # Don't include files twice just because they're explicitly included with -i
131             f = os.path.realpath(f)
132             if f in includes:
133                 del includes[f]
134
135     # Any explicitly included files that weren't already covered (i.e. those outside args.path)
136     if len(includes):
137         print(r'\section{Miscellaneous}')
138         for _,f in includes.items():
139             f = args.dir + '/' + os.path.relpath(f, args.dir)
140             print(dumpsrc(os.path.dirname(f), os.path.basename(f)))
141
142     print(r'\end{document}')
143
144 main()
```

2 marco_polo

2.1 marco_polo/__init__.py

```
1  # Copyright (c) 2023 Mobius Logic, Inc.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #    http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
```

3 marco_polo/algorithms

3.1 marco_polo/algorithms/evolutiontemplate.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


import argparse
from collections import OrderedDict
import json
import logging
import os
from numpy.random import PCG64DXSM, Generator

from marco_polo.optimizers.uber_es.manager import Manager  # for type hinting
from marco_polo.tools.types import Role, PathString  # for type hinting
from marco_polo.tools.wrappers import EnvHistory, TeamHistory  # for type hinting

logger = logging.getLogger(__name__)


################################################################################
## Aux Transfer Funcs
################################################################################


################################################################################
## Optim Func
################################################################################


################################################################################
## Evolve Class
################################################################################
class EvolutionTemplate:
    """
    Template for Evolution Algorithms

    This class implements a sequential-style evolution, similar to a Moran process,
    where environments are checked if they are ready to reproduce, and then they
    produce a single child that replaces the parent environment.

    Attributes
    ----------
    None
    """

    def __init__(
        self, args: argparse.Namespace, manager: Manager, transfer_role: Role
    ) -> None:
        """
        PataECEvolution Initilizer

        Parameters
        ----------
        args : argparse.Namespace
            This stores all of the simulation parameters
        manager : Manager Object
            This class handles the compute and multithreading
        transfer_role : str
            Which role is used to determine the evolution

        Side-Effects
        ------------
        None

        Returns
```

```
        -------
        None
        """

        # self.args = args
        # self.novelty = pata_ec(args, manager, transfer_role)
        # self.manager = manager
        # self.transfer_role = transfer_role
        # self.np_random = Generator(PCG64DXSM(seed=args.master_seed))

        pass

    ########################################
    ## Aux Funcs
    ########################################
    def checkpoint(self, folder: PathString) -> None:
        """
        Save Object Attributes to Disk

        This class has no attributes to save.

        Parameters
        ----------
        folder : PathString
            Path to folder that will contain checkpoint information

        Side-effects
        ------------
        None

        Returns
        -------
        None
        """

        pass

    def reload(self, folder: PathString) -> None:
        """
        Reload Object Attributes

        This class has no attributes to reload.

        Parameters
        ----------
        folder : PathString
            Folder containing checkpoint

        Side-effects
        ------------
        None

        Returns
        -------
        None
        """

        pass

    ########################################
    ## Main Func
    ########################################
    def Evolve(
        self,
        bracket: OrderedDict[EnvHistory, TeamHistory],
        archived_envs: OrderedDict[str, EnvHistory],
        epoch: int,
        repro_threshold: float,
        max_active_envs: int = 8,
        max_children: int = 8,
        max_admitted: int = 1,
    ) -> tuple[
        list[EnvHistory],
        OrderedDict[str, TeamHistory],
        OrderedDict[EnvHistory, TeamHistory],
        int,
        int,
        OrderedDict[str, EnvHistory],
    ]:
        """Try to evolve new environmental niches from old ones.

```

```
        Check if it's time to evolve. If so, get list of candidate optimizers for
        reproduction. Score each optimizer on each env, clip and rank the scores.
        Based on those scores, get a list of children up to max_children, loop through
        the potetntial children evaluating each until we find max_admitted suitable candidates

        Parameters
        ----------
        brackets : OrderedDict[env, agent]
            Ordered dictionary of env, EnvStat pair, viable envs and agents will be derived from this.
        archived_envs : OrderedDict[int, env]
            Ordered dictionary of environments, keyed by ID
        epoch : int
            Current global epoch number
        repro_threshold : float
            Reproduction threashold
        max_active_envs : int, default=8
            Maximum number of environments to keep active, older envs will be archived
        max_children : int, default=8
            Maximum number of mutations to attempt, attempted mutations may not pass mc
        max_admitted : int, default=1
            How many mutations to keep

        Side-Effects
        ------------
        Some
            Updates the id counters and prng states in agents and environments.
            If anything is archived, the novelty archive is updated.
            Calls "_playall()", but only on new environments.

        Returns
        -------
        new_envs : List[env]
            List of newly added environments
        new_teams : OrderedDict[int, agent]
            Ordered dictionary of new team IDs and teams
        bracket : OrderedDict[env, agent]
            Updated environment/agent bracket
        ANNECS : int
            Number of archived environments that were solved
        not_ANNECS : int
            Number of archived environments that were not solved
        to_archive : OrderedDict[int, env]
            Ordered dictionary of newly archived environments
        """

        # init vars for return
        new_envs: list[EnvHistory] = []
        new_teams: OrderedDict[str, TeamHistory] = OrderedDict()
        ANNECS = 0
        not_ANNECS = 0
        to_archive: OrderedDict[str, EnvHistory] = OrderedDict()

        to_remove = list()

        for env, team in bracket.items():
            # check if team is ready for evolution
            if env.stats.transfer_threshold >= repro_threshold:
                # generate new env
                new_env = env.get_mutated_env()

                # fill with some stats
                new_env.stats.created_at = epoch
                # new_env_params.stats.recent_scores.append(child_stats.eval_returns_mean)
                # new_env_params.stats.transfer_threshold = child_stats.eval_returns_mean
                new_env.stats.team = team.copy()

                # this score isn't really valid.
                new_env.stats.best_score = env.stats.recent_scores[
                    -1
                ]  # give it most recent score
                new_env.stats.best_team = new_env.stats.team.copy()

                # update stats
                new_envs.append(new_env)
                new_teams[new_env.stats.team.id] = new_env.stats.team
                ANNECS += 1
                to_archive[env.id] = env

                # update bracket things
                to_remove.append(env)

```

```
238         # add new ones
239         for env in new_envs:
240             bracket[env] = env.stats.team
241 
242         # remove old ones
243         for env in to_remove:
244             del bracket[env]
245 
246         # return info
247         return new_envs, new_teams, bracket, ANNECS, not_ANNECS, to_archive
```

3.2 marco_polo/algorithms/novelty.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


import argparse  # for type hinting
import json
import logging
import os
from collections import OrderedDict

import numpy as np

from marco_polo.tools.iotools import (
    NumpyDecoder,
    NumpyEncoder,
    load_keyed_tuple,
    save_keyed_tuple,
)
from marco_polo.optimizers.uber_es.manager import Manager
from marco_polo.tools.stats import compute_CS_ranks
from marco_polo.tools.types import EnvId, FloatArray, Role, PathString
from marco_polo.tools.wrappers import (
    EnvHistory,
    RunTask,
    TeamHistory,
)  # for type hinting

logger = logging.getLogger(__name__)


################################################################################
## Aux Things
################################################################################
def _cap_score(score: float, lower: float, upper: float) -> float:
    """
    Restrict the range of returned value

    This function ensures that the score of interest is greater than some lower
    bound and less than some upper bound. This is not a normalization, more similar
    to a truncation.

    Parameters
    ----------
    score: float
        Current score to restrict
    lower: float
        Lower bound on restriction
    upper: float
        Upper bound on restriction

    Side-effects
    ------------
    None

    Returns
    -------
    float
        value between lower and upper
    """

    if score < lower:
        score = lower
    elif score > upper:
```

```
        score = upper

    return score


def _euclidean_distance(x: FloatArray, y: FloatArray) -> np.float_:
    """
    Calculate the euclidean distance between 2 vectors

    Originally, this was a complicated function with protections for non-standard
    input format and x/y objects of differing lengths. At this point, we expect
    numpy arrays of equal length, or else the simulation is wrong somewhere else.

    Parameters
    ----------
    x: numpy.array of float
        array of agent scores
    y: numpy.array of float
        array of agent scores

    Side-effects
    ------------
    None

    Returns
    -------
    float
        distance between the vectors
    """

    # print("\n")
    # print(type(x))
    # print(type(y))
    # print(len(x))
    # print(len(y))
    # print("\n")

    # if type(x) is not list:
    #     #print("novelty._euclidean_distance:: inputs not of type list but of type", type(x))
    #     x = np.array([x])
    #     y = np.array([y])

    # n, m = len(x), len(y)

    # if n > m:
    #     # compute distance between shared dimensions
    #     a = np.linalg.norm(y - x[:m])
    #     # impute distance between last point and extra dimensions
    #     b = np.linalg.norm(y[-1] - x[m:])
    # else:
    #     a = np.linalg.norm(x - y[:n])
    #     b = np.linalg.norm(x[-1] - y[n:])
    # # combine as if 2-d euclidean
    # return np.sqrt(a**2 + b**2)

    return np.linalg.norm(x - y)


################################################################################
## PATA_EC Class
################################################################################
class pata_ec:
    """
    Performance of All Transferred Agents Environment Characterization

    Attributes
    ----------
    manager :
        Compute manager object defined under the optimizers directory
    rl : str
        Agent role in the game
    mc_lower : float
        Lower bound on evaluation scores
    mc_upper : float
        Upper bound on evaluation scores
    archived_pata_ec : dict[tuple(int, int), float]
        Dictionary of archived team scores, for reference to avoid recalculating things
    pata_list : list[numpy.array(numpy.float32)]
        Current centered/scaled scores
    k : int
        How many nearest environments to use as a score
```

```
    log_pata_ec : bool
        Should we log raw and scaled scores every epoch
    log_file : string
        Root directory for pata_ec logging
    """

    def __init__(self, args: argparse.Namespace, manager: Manager, role: Role) -> None:
        """
        pata_ec class initializer

        Parameters
        ----------
        args : argparse.Namespace
            Seems to be like a dictionary
            This stores all of the simulation parameters
        manager : Manager Object
            Compute manager object
        role : Role
            The role of the agent to use for novelty calculations

        Side-effects
        ------------
        None

        Returns
        -------
        None
        """

        self.manager = manager
        self.role = role
        self.mc_lower = args.mc_lower
        self.mc_upper = args.mc_upper
        self.archived_pata_ec: dict[tuple[EnvId, EnvId], float] = {}
        self.pata_list: list[FloatArray] = []
        self.k = 5
        self.log_pata_ec = args.log_pata_ec
        self.log_file = os.path.join(args.log_file, "pata_ec")

        # if logging, build output directory
        if self.log_pata_ec:
            os.makedirs(self.log_file, exist_ok=True)

    ########################################
    ## Funcs
    ########################################
    def checkpoint(self, folder: PathString) -> None:
        """
        Store Objects to Disk

        Stores all parameters except the manager object. It checks if the archive
        has objects and stores if it does.

        Parameters
        ----------
        folder : str
            Directory to store objects

        Side-effects
        ------------
        None

        Returns
        -------
        None
            All objects are saved to disk
        """
        # check archive, save if it has things
        if self.archived_pata_ec:
            save_keyed_tuple(
                self.archived_pata_ec, os.path.join(folder, "archived_pata_ec.csv")
            )

        # get all attributes, but remove the 2
        tmp = {
            k: v
            for k, v in self.__dict__.items()
            if k not in {"archived_pata_ec", "manager"}
        }

        # store
```

```
        with open(os.path.join(folder, "pata_ec.json"), mode="w", encoding="utf8") as f:
            json.dump(tmp, f, cls=NumpyEncoder)

    def reload(self, folder: PathString) -> None:
        """
        Reload Objects from Disk

        This function assumes that the args and manager objects are handled elsewhere.
        Selectively reloads archive if it was stored

        Parameters
        ----------
        folder : str
            Directory to reload objects

        Side-Effects
        ------------
        All
            Creates all internal objects

        Returns
        -------
        None
        """
        with open(
            os.path.join(folder, "pata_ec.json"), mode="r", encoding="utf8"
        ) as file:
            # load main stuff
            dct = json.load(file, cls=NumpyDecoder)

            # check if archive exists
            filepath = os.path.join(folder, "archived_pata_ec.csv")
            if os.path.isfile(filepath):
                dct["archived_pata_ec"] = load_keyed_tuple(filepath, float)

            # load into novelty
            for key, val in dct.items():
                if key not in {"manager"}:
                    self.__dict__[key] = val

    def _log_child_pata(
        self,
        filename: str,
        arc_opts: list[EnvHistory],
        act_opts: list[EnvHistory],
        child_opts: list[EnvHistory],
        pataList: list[FloatArray],
    ) -> None:
        """
        Write PATA_EC of Proposed Child Environments

        Parameters
        ----------
        filename : str
            Output file name
        arc_opts : list[env]
            List of archived environments to pair with scores
        act_opts : list[env]
            List of active environments to pair with scores
        child_opts : list[env]
            List of child environments to pair with scores
        pataList : list[iterable[float]]
            List of scores to write out

        Side-Effects
        ------------
        None

        Returns
        -------
        None
            Output written to file
        """
        # constuct a dict with keys given by a tuple of each pair
        tmp_dct: dict[tuple[EnvId, EnvId], float] = {}
        r_keys = [env.id for env in arc_opts + act_opts]
        l_keys = [env.id for env in child_opts]
        for l_key, scores in zip(l_keys, pataList):
            for r_key, value in zip(r_keys, scores):
                tmp_dct[(l_key, r_key)] = value
        filepath = os.path.join(self.log_file, filename)
```

```
322             save_keyed_tuple(tmp_dct, filepath)
323
324         def novelty(
325             self,
326             arc_opts: OrderedDict[str, EnvHistory],
327             opts: list[EnvHistory],
328             opt_list: list[EnvHistory],
329             epoch: int,
330         ) -> dict[EnvId, float]:
331             """
332             Calculate New Novelty
333
334             This calculates PATA_EC based novelty for all envs in `opt_list`.
335             It does not save or store any of this information. It takes all teams
336             from the current/archived population and runs them on the proposed environments.
337
338             Parameters
339             ----------
340             arc_opts : OrderedDict[str, EnvHistory]
341                 Ordered dictionary of archived environments
342             opts : list[EnvHistory]
343                 List of active environments
344             opt_list : list[EnvHistory]
345                 List of proposed environments with or without teams
346             epoch : int
347                 Current simulation time
348
349             Side-Effects
350             ------------
351             None
352
353             Returns
354             -------
355             novelty : dict[EnvId, float]
356                 Dictionary of environment IDs and their novelty scores
357             """
358             # NOTE: We need to be very careful with this function and the update_novelty().
359             #       The env/team pairs must be evaluated in the same order or else
360             #       the distance calculation is wrong.
361
362             # setup objects and counters
363             num_opt = len(arc_opts) + len(opts)
364             tasklist: list[RunTask] = []
365             novelty: dict[EnvId, float] = {}
366             # these 2 for storing output
367             capList: list[FloatArray] = []
368             CNList: list[FloatArray] = []
369
370             # loop over new envs and all teams (associated with archived envs), build task list
371             for env1 in opt_list:
372                 # loop over archive
373                 for env2 in arc_opts.values():
374                     tasklist.append((env1, env2.stats.team))
375
376                 # loop over active population
377                 for env2 in opts:
378                     tasklist.append((env1, env2.stats.team))
379
380             # evaluate all teams on environments
381             stats = self.manager.evaluate(tasks=tasklist)
382
383             # Loop through envs, get team performance, calculate novelty
384             stats_iter = iter(stats)
385             for env1 in opt_list:
386                 # intermediate list
387                 capped_scores = []
388
389                 # loop over teams
390                 for _ in range(num_opt):
391                     # get mean result from run
392                     #  cap score for stability
393                     capped_scores.append(
394                         _cap_score(
395                             score=next(stats_iter)[self.role].eval_returns_mean,
396                             lower=self.mc_lower,
397                             upper=self.mc_upper,
398                         )
399                     )
400
401                 # save capped scores for later
402                 capList.append(np.array(capped_scores))
```

```
403
404             # calculate ranked and centered scores
405             csRank = compute_CS_ranks(np.array(capped_scores))
406             # store for later
407             CNList.append(csRank)
408
409             # calculate Euclidean distance between current env and other envs
410             distances = np.array(
411                 [_euclidean_distance(csRank, c) for c in self.pata_list]
412             )
413
414             # sort, get K closest envs, and calculate the average distance as novelty
415             top_k_indicies = (distances).argsort()[: self.k]
416             novelty[env1.id] = distances[top_k_indicies].mean()
417
418         # logging child pata_ec every epoc for analysis
419         if self.log_pata_ec:
420             # write plain capped scores
421             self._log_child_pata(
422                 filename=f"Epoch_{epoch}_Child_Capped.csv",
423                 arc_opts=list(arc_opts.values()),
424                 act_opts=opts,
425                 child_opts=opt_list,
426                 pataList=capList,
427             )
428
429             # output centered and normalized scores
430             self._log_child_pata(
431                 filename=f"Epoch_{epoch}_Child_Processed.csv",
432                 arc_opts=list(arc_opts.values()),
433                 act_opts=opts,
434                 child_opts=opt_list,
435                 pataList=CNList,
436             )
437
438         # cleanup
439         del capList, CNList
440
441         # return novelty dict
442         return novelty
443
444     def update_archive(
445         self,
446         cur_arc_opts: OrderedDict[str, EnvHistory],
447         cur_opts: OrderedDict[EnvHistory, TeamHistory],
448         new_opts: list[EnvHistory],
449         new_arc_opts: OrderedDict[str, EnvHistory],
450     ) -> None:
451         """
452         Update Archived Novelty Scores
453
454         This function gets called whenever an environment gets archived. It then
455         runs the archived team against all existing environments and updates the
456         archive dictionary with the final scores.
457
458         Parameters
459         ----------
460         cur_arc_opts : OrderedDict[str, EnvHistory]
461             Ordered dictionary of archived environments with teams
462         cur_opts : OrderedDict[EnvHistory, TeamHistory]
463             Ordered dictionary of the currently active environments
464         new_opts : list[EnvHistory]
465             List of new environments that were just added to the current optimizers
466         new_arc_opts : OrderedDict[str, EnvHistory]
467             Ordered dictionary of newly archived environments
468
469         Side-Effects
470         ------------
471         self.archived_pata_ec
472             Update internal archive with new scores
473
474         Returns
475         -------
476         None
477         """
478         # NOTE: Order here doesn't matter as much as the other 2 because we're
479         #       just loading into a dict.
480         #       Do need to make sure the order within this function is consistent,
481         #       otherwise things will get labeled wrong.
482         # do 3 loops to avoid building a super long list of things to loop over.
483
```

```
        # evals to compute
        tasklist = []

        # current archive on new env
        for env1 in new_opts:
            for env2 in cur_arc_opts.values():
                tasklist.append((env1, env2.stats.team))

        # loop over new agents
        for env2 in new_arc_opts.values():
            # current archive
            for env1 in cur_arc_opts.values():
                tasklist.append((env1, env2.stats.team))

            # current active
            for env1 in cur_opts.keys():
                tasklist.append((env1, env2.stats.team))

            # new archive
            for env1 in new_arc_opts.values():
                tasklist.append((env1, env2.stats.team))

        # evaluate all agents on environments
        stats = self.manager.evaluate(tasks=tasklist)

        # loop through and add to archive
        stats_iter = iter(stats)

        # Note: the keys to archived_pata_ec are a tuple: (env1.id, env2.id). This is
        # actually tracking a comparison of env1 to team2. Using env2.id instead of
        # team2.id should give the same result since the team doesn't change for
        # archived cases. Using env2 instead of team2 is kept for historical reasons.
        # current archive on new env
        for env1 in new_opts:
            for env2 in cur_arc_opts.values():
                self.archived_pata_ec[(env1.id, env2.id)] = _cap_score(
                    score=next(stats_iter)[self.role].eval_returns_mean,
                    lower=self.mc_lower,
                    upper=self.mc_upper,
                )

        # loop over new agents
        for env2 in new_arc_opts.values():
            # current archive
            for env1 in cur_arc_opts.values():
                self.archived_pata_ec[(env1.id, env2.id)] = _cap_score(
                    score=next(stats_iter)[self.role].eval_returns_mean,
                    lower=self.mc_lower,
                    upper=self.mc_upper,
                )

            # current active
            for env1 in cur_opts.keys():
                self.archived_pata_ec[(env1.id, env2.id)] = _cap_score(
                    score=next(stats_iter)[self.role].eval_returns_mean,
                    lower=self.mc_lower,
                    upper=self.mc_upper,
                )

            # new archive
            for env1 in new_arc_opts.values():
                self.archived_pata_ec[(env1.id, env2.id)] = _cap_score(
                    score=next(stats_iter)[self.role].eval_returns_mean,
                    lower=self.mc_lower,
                    upper=self.mc_upper,
                )

    def purge_archive(self, to_purge: set[EnvId]) -> None:
        """
        Clear archive of Forgotten Environments

        When an environment gets removed, but wasn't solved, it is forgotten instead
        of logged. In that instance, the archive still has traces of other archived
        agents playing on this environment. This function removes those traces.

        Parameters
        ----------
        to_purge : set[EnvId]
            Set of environment IDs to remove from the archive

        Side-Effects
```

```
        ------------
        Yes
            Removes matching entries from the archive

        Returns
        -------
        None
        """
        # archive keys are (env.id, env_agt.id)
        # we need to remove all matching env.id

        # instantiate list of keys to delete
        delList = []

        # loop through archive, collect keys to remove
        for k in self.archived_pata_ec.keys():
            if k[0] in to_purge:
                delList.append(k)

        # remove matching keys
        for k in delList:
            del self.archived_pata_ec[k]

    def _log_active_pata(
        self, filename: str, optList: list[EnvHistory], pataList: list[FloatArray]
    ) -> None:
        """
        Write PATA_EC of Active Environments

        Parameters
        ----------
        filename : str
            Output file name
        optList : list[EnvHistory]
            List of environments to pair with scores
        pataList : list[FloatArray]
            List of scores to write out

        Side-Effects
        ------------
        None

        Returns
        -------
        None
            Output written to file
        """
        # constuct a dict with keys given by a tuple of each pair
        tmp_dct = {}
        r_keys = [env.id for env in optList]
        l_keys = r_keys[:]
        for l_key, scores in zip(l_keys, pataList):
            for r_key, value in zip(r_keys, scores):
                tmp_dct[(l_key, r_key)] = value
        filepath = os.path.join(self.log_file, filename)
        save_keyed_tuple(tmp_dct, filepath)

    def update_novelty(
        self, arc_opts: OrderedDict[str, EnvHistory], opts: list[EnvHistory], epoch: int
    ) -> None:
        """
        Update Novelty List

        This function updates the internal list of centered and scaled scores for
        the existing envvironments. It calculates new scores for actively evolving
        agents or pulls old scores from the archival dict.

        Parameters
        ----------
        arc_opts : OrderedDict[str, EnvHistory]
            Ordered dictionary of archived optimizers with agents
        opts : list[EnvHistory]
            List of active optimizers with agents
        epoch : int
            Current simulation epoch

        Side-Effects
        ------------
        self.pata_list
            Internal list of normalized scores updated
```

```
        Returns
        -------
        None
            Output written to file
        """

        # NOTE: We need to be very careful with this function and the novelty().
        #       The env/agent pairs must be evaluated in the same order or else
        #       the distance calculation is wrong.

        # setup objects and counters
        all_opt = list(arc_opts.values()) + opts

        tasklist = []
        raw_pata_list: list[FloatArray] = []
        self.pata_list.clear()

        # loop over all envs and all agents, build task list
        # do NOT need to copy the teams here, they are safe
        for env1 in arc_opts.values():
            for env2 in opts:
                tasklist.append((env1, env2.stats.team))

        for env1 in opts:
            for env2 in opts:
                tasklist.append((env1, env2.stats.team))

        # run new tasks
        stats = self.manager.evaluate(tasks=tasklist, epoch=epoch)

        logger.debug(f"Return Stats: {stats}")

        # grab iterator of stats
        stats_iter = iter(stats)

        # loop through everyone, update score vec
        for env1 in all_opt:
            # intermediate list
            capped_scores = []

            # loop over archive first
            for env2 in arc_opts.values():
                # archive key
                capped_scores.append(self.archived_pata_ec[(env1.id, env2.id)])

            # loop over active second
            for _ in opts:
                capped_scores.append(
                    _cap_score(
                        score=next(stats_iter)[self.role].eval_returns_mean,
                        lower=self.mc_lower,
                        upper=self.mc_upper,
                    )
                )

            # track raw scores
            raw_pata_list.append(np.array(capped_scores))
            # compute new centered and scaled ranks, update pata list
            self.pata_list.append(compute_CS_ranks(np.array(capped_scores)))

        # logging pata_ec every epoc for analysis
        if self.log_pata_ec:
            # write plain capped scores
            self._log_active_pata(
                filename=f"Epoch_{epoch}_Active_Capped.csv",
                optList=all_opt,
                pataList=raw_pata_list,
            )

            # output centered and normalized scores
            self._log_active_pata(
                filename=f"Epoch_{epoch}_Active_Processed.csv",
                optList=all_opt,
                pataList=self.pata_list,
            )

        # clean up
        del raw_pata_list, tasklist
```

3.3 marco_polo/algorithms/poet.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


import argparse  # for type hinting
import json
import logging
import os
from collections import OrderedDict
from itertools import chain
from typing import cast, Iterable
from numpy.random import PCG64DXSM, Generator

from marco_polo.algorithms.novelty import pata_ec
from marco_polo.optimizers.uber_es.manager import Manager  # for type hinting
from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder
from marco_polo.tools.types import Role, PathString  # for type hinting
from marco_polo.tools.wrappers import (
    EnvHistory,
    NoneTeam,
    TeamHistory,
    RunTask,
)  # for type hinting

logger = logging.getLogger(__name__)


################################################################################
## Aux Transfer Funcs
################################################################################
def _playall(
    manager: Manager,
    env_list: Iterable[EnvHistory],
    team_list: list[TeamHistory],
    transfer_role: Role,
    epoch: int,
) -> None:
    """
    Combinatorial Environment/Team Transfers

    This function handles team transfer events. Events involve a first play
    of every environment against every team other than its own. From there,
    possible successes recieve 1 optimization step and are played again. If these
    teams pass a transfer threshold, they become the new team for that environment.

    Parameters
    ----------
    manager : Manager
        Manager object to deal with threads
    env_list : Iterable[EnvHistory]
        iterable-object of environments to use
    team_list : list[TeamHistory]
        List of teams to use. This supports None as an team
    transfer_role : Role
        Role of the *agent* in a team that is used to determine
        whether a transfer happens
    epoch : int
        Current simulation time

    Side-Effects
    ------------
    Many
        If a transfer happens, environment statistics, best team,
```

```
        and current team are updated. Teams get copied many times
        when passed around.

    Returns:
    --------
    None
    """
    # log that we're starting transfers
    a_len = len(team_list)
    logger.info(f"Epoch: {epoch} Computing direct transfers: {a_len*(a_len-1)}")

    logger.debug("team_list coming into _playall: " + str(team_list))
    # https://stackoverflow.com/a/57872832
    # setup direct-transfer tasks
    #  this involves pairing each env with all teams OTHER THAN IT'S OWN
    direct_tasks: list[RunTask] = []
    for i, env in enumerate(env_list):
        for team_idx in chain(range(0, i), range(i + 1, a_len)):
            # This does NOT need copied, because we just eval it here
            team = team_list[team_idx]
            direct_tasks.append((env, team))

    logger.debug("Direct transfer tasks: " + str(direct_tasks))
    # run evals
    returns = manager.evaluate(tasks=direct_tasks, epoch=epoch)

    # Check evals for proposal transfers
    logger.debug("Direct transfer scores:")
    proposal_tasks: list[RunTask] = []
    for stats, (env, team) in zip(returns, direct_tasks):
        # Did this team do better than the current team?
        logger.debug(
            f"Attempted Transfer: {env.id} - {stats[transfer_role].eval_returns_mean}"
        )
        if stats[transfer_role].eval_returns_mean > env.stats.transfer_threshold:
            # log successful direct transfer test
            logger.info(
                f"Direct Transfer: Epoch={epoch} - Env={env} - Team={team} -
                    Score={stats[transfer_role].eval_returns_mean}"
            )
            # need to run proposal on direct transfer success
            #  THIS NEEDS TO BE A COPY
            #  We perform an optimization on this team, so it must be independent
            proposal_tasks.append((env, team.copy()))

    # log proposals
    logger.info(f"Epoch: {epoch} Computing proposal transfers: {len(proposal_tasks)}")

    # make sure there are proposals, otherwise skip
    if proposal_tasks:
        # single optimization step for env/team
        _ = manager.optimize_step(tasks=proposal_tasks, epoch=epoch, opt_iter=None)

        # evaluate optimized env/team pair
        returns = manager.evaluate(tasks=proposal_tasks, epoch=epoch)

        # check results of proposals
        for stats, (env, team) in zip(returns, proposal_tasks):
            # grab mean cause I'm tired of typing
            eval_mean = stats[transfer_role].eval_returns_mean

            # Did this team do better than the current team?
            if eval_mean > env.stats.transfer_threshold:
                # log successful direct transfer test
                logger.info(
                    f"Proposal Transfer: Epoch={epoch} - Env={env} - Team={team}"
                )

                # update env stats
                env.stats.recent_scores.clear()
                env.stats.recent_scores.append(eval_mean)
                env.stats.transfer_threshold = eval_mean
                env.stats.iterations_current = 1
                # lifespan best score and team
                if env.stats.best_score < eval_mean:
                    env.stats.best_score = eval_mean
                    env.stats.best_team = team.copy()
                    # this team is the same as "team", but gets a new id because of the copy
                    #  THIS COPY STATEMENTS ISREQUIRED HERE!!!!
                    #  It must be an independent team

```

```
                # update env/team pairing
                #  This does NOT need copied, because "proposal_tasks" is a copy
                env.stats.team = team


def AllActiveBestScore(
    manager: Manager,
    bracket: OrderedDict[EnvHistory, TeamHistory],
    transfer_role: Role,
    epoch: int,
) -> None:
    """
    Play All Agents and Perform Transfers

    Passes all work down to "_playall()".
    Then updates the active env/team bracket.

    Parameters
    ----------
    manager : Manager
        Manager object to deal with threads
    bracket : OrderedDict[EnvHistory, TeamHistory]
        Ordered dictionary of env, team pairs
    transfer_role : Role
        Which role is used to determine the evolution
    epoch : int
        Current simulation time

    Side-Effects
    ------------
    Many
        See "_playall()" for more details, many environment stats are updated.
    """
    # Play all envs against all teams
    _playall(
        manager=manager,
        env_list=bracket.keys(),
        team_list=list(bracket.values()),
        transfer_role=transfer_role,
        epoch=epoch,
    )

    logger.debug("Bracket after _playall:" + str(bracket))

    # update bracket
    for env in bracket:
        logger.debug("Teams in Envs: " + str(env.stats.team))
        bracket[env] = env.stats.team


################################################################################
## Optim Func
################################################################################
def PoetOptLoop(
    tasks: list[RunTask], epoch: int, manager: Manager, verbose: bool = False
) -> None:
    """

    Parameters
    ---------
    tasks : list[RunTask]
        List of RunTask to optimize
    epoch: int
        Main loop counter
    manager : Manager
        Manager object to deal with compute
    verbose : bool, default = False
        Manage logging

    Side-effects
    ------------
    Many
        This function updates the environment statistics based on rollouts.
        Internally, "manager.optimize_chunk()" changes things within environments
        and their corresponding teams.

    Returns
    -------
    None
    """
    # pass to manager to optimize
```

```
    #  returns a list of eval means for each optim step
    results = manager.optimize_chunk(tasks=tasks, epoch=epoch, verbose=verbose)

    # use the results to update environment stats
    for stats, (env, team) in zip(results, tasks):
        # grab eval results
        #  this is a list of mean evaluation values that is optim_iters long
        #  since team is actually several agents, but we're playing a 1-player
        #  game, grab the first key
        eval_mean_list = stats[next(iter(team.agent_group))]
        # store most recent evals
        env.stats.recent_scores.extend(eval_mean_list)
        # transfer threshold is best of N best scores
        env.stats.transfer_threshold = max(env.stats.recent_scores)

        # lifespan best score and team
        #  only checks the team after a whole lest of optimization, so we miss
        #  the teams/scores inbetween. Ergo, there is a chance that
        #  transfer_threshold is transiently better than best_score
        if env.stats.best_score < eval_mean_list[-1]:
            env.stats.best_score = eval_mean_list[-1]
            env.stats.best_team = team.copy()
            # this team is the same as "team", but gets a new id because of the copy
            #  THIS COPY STATEMENTS ISREQUIRED HERE!!!!
            #  It must be an independent team
            # NOTE: Is this missing a total_teams update?
            #       It's a new team copy, we don't every log it?
            #       Or should we be doing this in the `optimize()` function in the manager?


################################################################################
## Evolve Class
################################################################################
class PataECEvolution:
    """
    Classic POET Evolution

    This class performs the classic POET evolution. Internally, it has a separate
    class to calculate novelty, which is then used to test for environment
    reproduction. This class handles environment archiving as well.

    Attributes
    ----------
    args : argparse.Namespace
        This stores all of the simulation parameters
    novelty : Novelty Class Object
        This class handles the novelty calculations
    manager : Manager Object
        This class handles the compute and multithreading
    transfer_role : Role
        Which role is used to determine the evolution
    np_random : Generator
        Numpy random generator
    """

    def __init__(
        self, args: argparse.Namespace, manager: Manager, transfer_role: Role
    ) -> None:
        """
        PataECEvolution Initilizer

        Parameters
        ----------
        args : argparse.Namespace
            This stores all of the simulation parameters
        manager : Manager Object
            This class handles the compute and multithreading
        transfer_role : Role
            Which role is used to determine the evolution

        Side-Effects
        ------------
        Yes
            Sets internal variables

        Returns
        -------
        None
        """

        self.args = args
```

```
        self.novelty = pata_ec(args, manager, transfer_role)
        self.manager = manager
        self.transfer_role = transfer_role
        self.np_random = Generator(PCG64DXSM(seed=args.master_seed))

    ########################################
    ## Aux Funcs
    ########################################
    def checkpoint(self, folder: PathString) -> None:
        """
        Save a checkpoint of the PataECEvolution object

        Parameters
        ----------
        folder: PathString
            Path to folder that will contain checkpoint information

        Side-effects
        ------------
        None

        Returns
        -------
        None
        """
        # Build folder path
        folder = os.path.join(folder, "POET")
        # Make folder
        os.makedirs(folder, exist_ok=True)

        # Dictionary of things to save
        tmp = {
            "transfer_role": self.transfer_role,
            "np_random": self.np_random.bit_generator.state,
        }

        # Save data
        with open(
            os.path.join(folder, "PataECEvolution.json"), mode="w", encoding="utf8"
        ) as f:
            json.dump(tmp, f, cls=NumpyEncoder)

        # Checkpoint novelty
        #  uses built-in functionality
        self.novelty.checkpoint(folder)

    def reload(self, folder: PathString) -> None:
        """
        Reload a checkpointed PataECEvolution object from the supplied folder

        Parameters
        ----------
        folder: PathString
            Folder containing checkpoint

        Side-effects
        ------------
        Yes
            Sets internal variables

        Returns
        -------
        None
        """
        # Build folder path
        folder = os.path.join(folder, "POET")

        # Reload data
        with open(
            os.path.join(folder, "PataECEvolution.json"), mode="r", encoding="utf8"
        ) as f:
            # Read in
            dct = json.load(f, cls=NumpyDecoder)

            # Set attributes
            self.transfer_role = dct["transfer_role"]
            self.np_random.bit_generator.state = dct["np_random"]

        # Create and reload novelty object
        self.novelty = pata_ec(self.args, self.manager, self.transfer_role)
        self.novelty.reload(folder)
```

```
402
403     def _check_env_status(
404         self, bracket: OrderedDict[EnvHistory, TeamHistory], repro_threshold: float
405     ) -> list[EnvHistory]:
406         """
407         Hidden Score Function
408
409         Check the latest scores on envs and return a list of candidates for reproduction.
410
411         Parameters
412         ----------
413         bracket : OrderedDict[EnvHistory, TeamHistory]
414             Ordered dictionary of environment/team pairs
415         repro_threshold : float
416             Minimal score to pass for reproduction
417
418         Side-effects
419         ------------
420         None
421
422         Returns
423         -------
424         list[EnvHistory]
425             List of environments that are score high enough to reproduce
426         """
427         # initialize loop objects
428         repro_candidates = []
429
430         # Loop through all active niches
431         #  This always loops in the same order - orderedDict()
432         for env in bracket.keys():
433             # check if ready to reproduce
434             #  use transfer_threshold, which is best of most recent 5 runs
435             #  just a little more stable than a single run
436             if env.stats.transfer_threshold >= repro_threshold:
437                 repro_candidates.append(env)
438
439         return repro_candidates
440
441     def _get_new_env(
442         self, list_repro: list[EnvHistory], num_offspring: int
443     ) -> tuple[list[tuple[EnvHistory, int, EnvHistory]], list[RunTask]]:
444         """
445         Generate Offspring Environment from Random Parent Environment
446
447         This function uses the list of potential parent environments to generate
448         the mutated children to possibly add to the currently active environments.
449         It picks a parent env from the list, produces a mutated cppn
450         and a new environment parameter set for all new agents.
451
452         Parameters
453         ----------
454         list_repro : list[EnvHistory]
455             List of environments to use as possible parents
456         num_offspring : int
457             How many new children to return.
458
459         Side-Effects
460         ------------
461         Environment Wrapper
462             This process increments the environment ID counter and updates the prng
463             state within the chosen parent environment.
464
465         Return
466         ------
467         children : list[tuple[EnvHistory, int, EnvHistory]]
468             List of Inherited, mutated CppnEnvParams from parent with associated wrapper class,
469             random seed, and parent environment
470         tasks : list[RunTask]
471             List of tuples of environment and team
472         """
473         # Grab list of ID of parent environment
474         env_parent_list = self.np_random.choice(
475             a=list_repro, size=num_offspring, replace=True, shuffle=False
476         )
477
478         # generate seed for child
479         seed_list = self.np_random.integers(low=0, high=2**31 - 1, size=num_offspring)
480
481         # loop and setup return objs
482         tasks = []
```

```
483             children = []
484             for i in range(num_offspring):
485                 # generate child env
486                 child_env = env_parent_list[i].get_mutated_env()
487 
488                 # add to return list
489                 children.append((child_env, seed_list[i], env_parent_list[i]))
490 
491                 # build task list
492                 tasks.append((child_env, env_parent_list[i].stats.team))
493 
494                 # log
495                 logger.info(f"Parent: {env_parent_list[i].id} - Child: {child_env.id}")
496 
497             # return child list and task list
498             return children, tasks
499 
500         def _pass_mcc(self, score: float) -> bool:
501             """
502             Check if score in Minimum Criterion Coevolution (MCC) range.
503 
504             Parameters
505             ----------
506             score : float
507                 Current score to check
508 
509             Side-Effects
510             ------------
511             None
512 
513             Returns
514             -------
515             bool
516                 Whether or not the score passed
517             """
518             # return checks
519             #  1) check that score is above lower
520             #  2) check that score is below upper
521             return (
522                 cast(float, self.args.mc_lower) <= score <= cast(float, self.args.mc_upper)
523             )
524 
525         def _get_child_list(
526             self,
527             archived_envs: OrderedDict[str, EnvHistory],
528             env_list: list[EnvHistory],
529             parent_list: list[EnvHistory],
530             max_children: int,
531             epoch: int,
532         ) -> list[tuple[EnvHistory, int, EnvHistory]]:
533             """
534             Returns a list of viable new environments
535 
536             This function does several things.
537             First, it updates the novelty scores on existing environments.
538             Then, it gets a potential new environment and evaluates it. If that environment
539             passes a score threshold, it is stored as a passing new offspring.
540             Then, it calculates the novelty scores for all passing offspring.
541             Finally, it returns a list of pasisng offspring tuples, sorted by novelty score.
542 
543             This function calls several other internal functions and relies on novelty
544             calculations.
545 
546             Parameters
547             ----------
548             archived_envs : OrderedDict[str, EnvHistory]
549                 Current environment archive
550             env_list : list[EnvHistory]
551                 Current active environments
552             parent_list : list[EnvHistory]
553                 List of environments that are viable for reproduction
554             max_children : int
555                 Maximum number of new environments to test
556             epoch : int
557                 Current simulation time
558 
559             Side-Effects
560             ------------
561             Several
562                 The agent/team id counters are incremented with every copy event.
563                 Evals update the internal state of current environments.
```

```
            Novelty scores get updated.
            The internals of new environments get modified.

        Returns
        -------
        list[tuple[EnvHistory, int, EnvHistory]]
            List of (new env, seed, parent env) tuple
        """
        # update current novelty
        self.novelty.update_novelty(arc_opts=archived_envs, opts=env_list, epoch=epoch)

        # setup return list to hold viable children
        #  These are defined as the potential children than pass MC checks.
        child_list = []
        child_novelties = []
        potential_children = {}
        pass_mcc = 0

        # get list of offspring
        #  children_list is a list of (child, seed, parent)
        children_list, task_list = self._get_new_env(
            list_repro=parent_list, num_offspring=max_children
        )

        # evaluate whole list
        stats = self.manager.evaluate(tasks=task_list, epoch=epoch)

        # check list for mcc
        for i in range(max_children):
            # grab stats
            child_stats = stats[i][self.transfer_role]

            # see if performance is viable
            if self._pass_mcc(score=child_stats.eval_returns_mean):
                # increment counter
                pass_mcc += 1

                # unpack child info
                new_env, seed, env_parent = children_list[i]

                # fill stats
                new_env.stats.created_at = epoch
                new_env.stats.recent_scores.append(child_stats.eval_returns_mean)
                new_env.stats.transfer_threshold = child_stats.eval_returns_mean
                assert env_parent.stats.team is not None
                new_env.stats.team = env_parent.stats.team.copy()

                new_env.stats.best_score = child_stats.eval_returns_mean
                new_env.stats.best_team = new_env.stats.team.copy()
                # this team is the same as "team", but gets a new id because of the copy
                #  THESE COPY STATEMENTS ARE REQUIRED HERE!!!!
                #  The child needs an independent team to own, thus the deep copy

                # add new child to list
                potential_children[new_env] = (new_env, seed, env_parent)

        # log number of passing envs
        logger.info(
            f"Attempted {max_children} reproductions - {pass_mcc} were successful"
        )

        # If there's kids, do stuff
        if potential_children:
            ## Compute the novelty scores:
            novelties = self.novelty.novelty(
                arc_opts=archived_envs,
                opts=env_list,
                opt_list=list(potential_children.keys()),
                epoch=epoch,
            )

            # loop over dictionary of new envs
            for env, vals in potential_children.items():
                # grab children and novelties
                child_novelties.append(novelties[env.id])
                child_list.append(vals)

            # sort child list according to novelty for high to low
            # https://www.geeksforgeeks.org/python-sort-values-first-list-using-second-list/
            child_list = [
                x
```

```
                for _, x in sorted(
                    zip(child_novelties, child_list), key=lambda x: x[0], reverse=True
                )
            ]

        # return child list
        return child_list

    ########################################
    ## Main Func
    ########################################
    def Evolve(
        self,
        bracket: OrderedDict[EnvHistory, TeamHistory],
        archived_envs: OrderedDict[str, EnvHistory],
        epoch: int,
        repro_threshold: float,
        max_active_envs: int = 8,
        max_children: int = 8,
        max_admitted: int = 1,
    ) -> tuple[
        list[EnvHistory],
        OrderedDict[str, TeamHistory],
        OrderedDict[EnvHistory, TeamHistory],
        int,
        int,
        OrderedDict[str, EnvHistory],
    ]:
        """Try to evolve new environmental niches from old ones.

        Check if it's time to evolve. If so, get list of candidate optimizers for
        reproduction. Score each optimizer on each env, clip and rank the scores.
        Based on those scores, get a list of children up to max_children, loop through
        the potetntial children evaluating each until we find max_admitted suitable candidates

        Parameters
        ----------
        brackets : OrderedDict[EnvHistory, TeamHistory]
            Ordered dictionary of env, EnvStat pair, viable envs and teams will be derived from this.
        archived_envs : OrderedDict[str, EnvHistory]
            Ordered dictionary of environments, keyed by ID
        epoch : int
            Current global epoch number
        repro_threshold : float
            Reproduction threashold
        max_active_envs : int, default=8
            Maximum number of environments to keep active, older envs will be archived
        max_children : int, default=8
            Maximum number of mutations to attempt, attempted mutations may not pass mc
        max_admitted : int, default=1
            How many mutations to keep

        Side-Effects
        ------------
        Some
            Updates the id counters and prng states in teams and environments.
            If anything is archived, the novelty archive is updated.
            Calls "_playall()", but only on new environments.

        Returns
        -------
        new_envs : list[EnvHistory]
            List of newly added environments
        new_teams : OrderedDict[str, TeamHistory]
            Ordered dictionary of new team IDs and teams
        bracket : OrderedDict[EnvHistory, TeamHistory]
            Updated environment/team bracket
        ANNECS : int
            Number of archived environments that were solved
        not_ANNECS : int
            Number of archived environments that were not solved
        to_archive : OrderedDict[str, EnvHistory]
            Ordered dictionary of newly archived environments
        """
        # get current list of envs and teams
        env_list = list(bracket.keys())

        # _playall() runs env_i against team_j for i != j
        # we want to run a single env [env_0] against all teams, so
        # we pad the list of teams with a dummy team at j = 0
        team_list_aug = [NoneTeam(None)] + list(bracket.values())
```

```
        cur_env_len = len(env_list)

        # check current envs to see if viable for reproduction
        list_repro = self._check_env_status(
            bracket=bracket, repro_threshold=repro_threshold
        )

        # objects to return
        new_envs: list[EnvHistory] = []
        new_teams: OrderedDict[str, TeamHistory] = OrderedDict()
        ANNECS = 0
        not_ANNECS = 0
        to_archive: OrderedDict[str, EnvHistory] = OrderedDict()

        # early escape if no reproductive envs
        if len(list_repro) == 0:
            logger.info("no suitable niches to reproduce")
            return new_envs, new_teams, bracket, ANNECS, not_ANNECS, to_archive
        else:
            logger.info(f"Epoch:{epoch} - List of niches to reproduce: {list_repro}")

        # Get list of potential children
        # These children have passed initial MCC check, and are ranked by novelty
        child_list = self._get_child_list(
            archived_envs=archived_envs,
            env_list=env_list,
            parent_list=list_repro,
            max_children=max_children,
            epoch=epoch,
        )

        for child in child_list:
            # unpack child object
            #  (new env, seed, parent env)
            new_env, seed, _ = child

            # Perform transfers
            #  include own team, because it didn't do a proposal the first time
            _playall(
                manager=self.manager,
                env_list=[new_env],
                team_list=team_list_aug,
                transfer_role=self.transfer_role,
                epoch=epoch,
            )

            # check MCC a second time
            #  probably to make sure it isn't too easy now?
            if self._pass_mcc(score=new_env.stats.recent_scores[-1]):
                # Pass checks! Add to current population
                new_envs.append(new_env)
                # no copy on these teams - it's fine
                new_teams[new_env.stats.team.id] = new_env.stats.team
                # break loop if finished
                if len(new_envs) >= max_admitted:
                    break

        # add new environments to active bracket
        #  This might put us over the max pop size, we'll fix that next
        for env in new_envs:
            bracket[env] = env.stats.team

        # Check if we've reached max pop size
        #  take care of archiving tasks if yes
        to_purge = set()
        num_remove = len(new_envs) + cur_env_len - max_active_envs
        if num_remove > 0:
            # get oldest keys from bracket
            #  this works cause bracket is an orderedDict
            rem_envs = env_list[0:num_remove]

            # loop over keys, remove, and score
            for env in rem_envs:
                # remove element from current bracket
                del bracket[env]

                # check if solved
                #  "solved" is defined as "was able to reproduce"
                if env.stats.recent_scores[-1] >= repro_threshold:
                    # log that we solved an env and store in archiv
                    ANNECS += 1
```

```
807                     to_archive[env.id] = env
808                 else:
809                     # Not strictly necessary, but we're removing an unsolved env,
810                     #  so log that it was there, but don't keep, in case it comes
811                     #  back later
812                     not_ANNECS += 1
813                     to_purge.add(env.id)
814 
815             # update archive with newly-archived environments
816             self.novelty.update_archive(
817                 cur_arc_opts=archived_envs,
818                 cur_opts=bracket,
819                 new_opts=new_envs,
820                 new_arc_opts=to_archive,
821             )
822 
823             # clean archive of env to forget
824             if not_ANNECS:
825                 self.novelty.purge_archive(to_purge=to_purge)
826 
827         # return info
828         return new_envs, new_teams, bracket, ANNECS, not_ANNECS, to_archive
```

4 marco_polo/base_classes

4.1 marco_polo/base_classes/model.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Base model interface"""

from typing import Any, Union

import numpy as np

from marco_polo.tools.types import ActivationType, FloatArray


class BaseModel:
    """Base interface for a model"""

    def __call__(self, *args: Any, **kwargs: Any) -> Any:
        raise NotImplementedError("Calling BaseModel is not implemented")

    def checkpoint(self, folder: str) -> None:
        """Save model to disk"""
        raise NotImplementedError("checkpoint not implemented for BaseModel")

    def reload(self, folder: str) -> None:
        """Load model from disk"""
        raise NotImplementedError("reload not implemented for BaseModel")

    def get_action(
        self, x: ActivationType, t: float = 0, mean_mode: bool = False
    ) -> Union[np.int_, FloatArray]:
        raise NotImplementedError("get_action not implemented for BaseModel")
```

4.2 marco_polo/base_classes/serialCrew.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Serial Pool for Serial Multiprocessing Manager"""

from marco_polo.tools.extraneous import SingleProcessPool


class SerialCrew:  # pylint: disable=too-few-public-methods
    """Creates a 'crew' of one the does everything directly"""

    def get_crew(self) -> SingleProcessPool:
        """Return a Dummy Pool with only a single process

        Returns
        -------
        SingleProcessPool
            'pool' of one worker
        """
        return SingleProcessPool()
```

5 marco_polo/cores/Connect4

5.1 marco_polo/cores/Connect4/__init__.py

```
1  # Copyright (c) 2023 Mobius Logic, Inc.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #    http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15
16 from .core import Core
```

5.2 marco_polo/cores/Connect4/core.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


import argparse  # for type hinting
import json
import logging
import os
import random
import subprocess
import time

from collections import OrderedDict
from typing import Any, Callable

import numpy as np

from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder
from marco_polo.tools.section_logging import SectionLogger
from marco_polo.tools.wrappers import AgtHistory, EnvHistory, TeamHistory

logger = logging.getLogger(__name__)

from marco_polo.optimizers.uber_es import get_opt_class, get_model

from ...envs.PettingZoo.env import PettingZooEnv, PettingZooEnvParams
from marco_polo.tools.wrappers import MP_AEC_to_Parallel


def get_env() -> Callable[..., Any]:
    return MP_AEC_to_Parallel(PettingZooEnv())


################################################################################
## Core Class
################################################################################
class Core:
    """
    Connect 4 Core

    Simple core for multiplayer Connect 4 from the PettingZoo repo
        (https://github.com/Farama-Foundation/PettingZoo).
    Can be used as a base for other PettingZoo environments and for league play.

    Attributes
    ----------
    args : argparse.Namespace
        Simulation parameters
    """

    def __init__(self, args: argparse.Namespace) -> None:
        # self.env_factory = Env_Factory(Game_Params)  # Returns something that creates niches? This is
            going to be a lot of the code that we we have written at this point.
        # self.model_factory = MLP_Factory(Model_Pramas)     # Returns something that creates models?
        self.args = args
        os.makedirs(os.path.join(args.log_file, "Rollouts"), exist_ok=True)
        os.makedirs(os.path.join(args.log_file, "Videos"), exist_ok=True)

        Manager = get_opt_class(args)
        # The manager expects any incoming args, as well as a list of gym envs
        self.manager = Manager(args)
        self.manager.verbose = False

        self.iterations = 0

        ## Create intial Env Params
        starting_env = EnvHistory(
```

```
            PettingZooEnvParams(), get_env, agents=["player_0", "player_1"]
        )

        self.envs = [starting_env]

        # Here, we're going to use an ordered dictionary to keep track of matchings
        # agent_group: OrderedDict[Role, AgtHistory] = OrderedDict()
        agent_group = OrderedDict()

        for agent in starting_env.agents:
            agent_group[agent] = AgtHistory(
                get_agent(starting_env, agent, args, args.master_seed)
            )

        initial_agents = TeamHistory(agent_group=agent_group)

        self.brackets = OrderedDict()
        self.brackets[starting_env] = initial_agents

    ########################################
    ## poet_algo.PopulationManager.evolve_population() Funcs
    ########################################
    def iteration_update(self, iteration: int) -> None:
        # safe-keeping for Curiosity currently
        pass

    def reproduce(self, iteration: int) -> None:
        pass

    def optimize(self, iteration: int) -> None:
        opt_time = time.time()
        tasks = list(self.brackets.items())
        self.manager.optimize_chunk(tasks, epoch=1)

        # self.manager.evaluate(tasks)

        # logger.info(f"optimize() Full Opt_Chunk Time:{time.time() - opt_time}")

    def transfer(self, iteration: int) -> None:
        pass
        # self.brackets = AllActiveBestScore(self.manager, self.brackets, True, "role1")

    def evolve_population(self, start_epoch: int, sim_start: float) -> None:
        self.optimize(start_epoch)
```

6 marco_polo/cores/Multiplayer

6.1 marco_polo/cores/Multiplayer/__init__.py

```
1  # Copyright (c) 2023 Mobius Logic, Inc.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #    http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15
16 from .core import Core
```

6.2 marco_polo/cores/Multiplayer/core.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Multi-Player  """

import copy
import json
import logging
import os
import random
import time
from collections import OrderedDict
import argparse  # for type hinting
from collections.abc import Callable  # for type hinting
from typing import Any  # for type hinting
import numpy as np

from marco_polo.algorithms.evolutiontemplate import EvolutionTemplate

from marco_polo.cores.ePoet.core import Core as BaseCore
from marco_polo.envs._base.env_params import EnvParams  # for type hinting
from marco_polo.tools.iotools import NumpyDecoder
from marco_polo.tools.section_logging import SectionLogger
from marco_polo.tools.wrappers import AgtHistory, EnvHistory, TeamHistory
from marco_polo.tools.types import Role, PathString

logger = logging.getLogger(__name__)


################################################################################
## Core Class
################################################################################
class Core(BaseCore):
    """
    Example core for Multiplayer Optimization. This core uses a simple sequential
    evolution algorithm to maintain the same number of environments. This is a good
    starting core for multiplayer development as it includes checkpointing and
    reloading.

    Attributes
    ----------
    args : argparse.Namespace
        Seems to be like a dictionary
        This stores all of the simulation parameters
    manager : Manager Object
        Class that handles compute/multi-threading
    evolver : PataECEvolution Object
        This object handles environment evolution
    total_envs : list[env]
        List of all environments in their order of creation
    archived_envs : OrderedDict[int, env]
        Ordered dictionary of envronments that have been archived
    total_teams : OrderedDict[int, env]
        Ordered dictionary of all teams in their order of creation
    brackets : OrderedDict[env, agent]
        Pairing of active agent/environments
    ANNECS : int
        Accumulated number of novel environments created and solved
    not_ANNECS : int
        Accumulated number of novel environments created and NOT solved
    """

    def __init__(
        self,
        args: argparse.Namespace,
        env_factory: Callable[..., Any],
```

```
        env_param_factory: Callable[..., Any],
        manager_factory: Callable[..., Any],
        model_factory: Callable[..., Any],
    ) -> None:
        """
        Core Class Initializer

        This function sets up the simulation starting state. It creates a
        compute manager and an evolution manager. It also creates the first
        agent(s) and environment(s).

        Parameters
        ----------
        args : argparse.Namespace
            Seems to be like a dictionary
            This stores all of the simulation parameters
        env_factory: Callable[..., Any],
            Factory for creating environment classes.
        env_param_factory: Callable[..., Any],
            Factory for creating environment parameter classes.
        manager_factory: Callable[..., Any],
            Factory for creating optimization manager classes.
        model_factory: Callable[..., Any],
            Factory for creating model classes.

        Side-effects
        ------------
        Many
            It builds all of the internal objects to start the simulation

        Returns
        -------
        None
        """

        # store sim args for later
        self.args = args
        self.env_factory = env_factory
        self.env_param_factory = env_param_factory
        self.manager_factory = manager_factory
        self.model_factory = model_factory

        # setup environment and parameter classes
        self.env_class = env_factory(args=args)
        self.env_param_class = env_param_factory(args=args)

        # Setup manager with any incoming args
        Manager = manager_factory(args=args)
        self.manager = Manager(setup_args=args)

        # Setup the environmnet evolution class
        self.role_name = Role("I DONT GET USED")
        self.evolver = EvolutionTemplate(
            args=args, manager=self.manager, transfer_role=self.role_name
        )

        # total env/team trackers
        self.total_teams: OrderedDict[str, TeamHistory] = OrderedDict()
        self.total_envs: list[EnvHistory] = []
        # active environment/team pairing
        self.brackets: OrderedDict[EnvHistory, TeamHistory] = OrderedDict()
        # archive
        self.archived_envs: OrderedDict[str, EnvHistory] = OrderedDict()

        # create initial env/team pairs (number given by args.num_start_envs)
        self._create_initial_env_teams(
            EnvCreator=self.env_class, env_param_class=self.env_param_class
        )

        # stats
        # accumulated number of novel environments created and solved
        self.ANNECS = 0  # pylint: disable=invalid-name
        # accumulated but not solved
        self.not_ANNECS = 0  # pylint: disable=invalid-name

    ########################################
    ## Funcs
    ########################################
    def _create_initial_env_teams(
        self, EnvCreator: Callable[..., Any], env_param_class: Callable[..., EnvParams]
    ) -> None:
```

```
        """
        Creates the initial environments and agents.
        """

        # temp env for creating agents
        tmp_env = EnvCreator()
        tmp_env.augment(params=env_param_class(self.args))

        # create initial team
        initial_agent_set = OrderedDict()
        for agent in tmp_env.agents:
            # copy params so we can tweak for each agent
            tmp_m_params = copy.copy(self.args)

            # grab input and output shape
            tmp_m_params.model_params["input_size"] = np.prod(
                tmp_env.observation_spaces[agent].shape
            )
            tmp_m_params.model_params["output_size"] = np.prod(
                tmp_env.action_spaces[agent].shape
            )

            # create initial agent
            #  args dict is because we made a true copy for different inputs/outputs
            #  original POET stuff used same in/outs for all agents
            initial_agent_set[agent] = AgtHistory(
                self.model_factory(
                    env=tmp_env,
                    role=agent,
                    args=tmp_m_params,
                    seed=self.args.master_seed,
                )
            )
        initial_team = TeamHistory(agent_group=initial_agent_set)

        # pair initial env/team and store
        starting_env = EnvHistory(
            env_param=env_param_class(self.args), env_creator=EnvCreator
        )
        starting_env.stats.team = initial_team
        self.total_teams[initial_team.id] = initial_team
        self.total_envs.append(starting_env)
        self.brackets[starting_env] = initial_team

        # check if we add other/more optimizers
        # Make sure we don't start with more than the maximum number of optimizers
        #  decrement by 1 because we already created 1 optimizer
        nStartOptims = min(self.args.max_active_envs, self.args.num_start_envs)
        nStartOptims -= 1

        # loop over remaing environments/teams to create
        for i in range(nStartOptims):
            # mutate new env
            new_env = starting_env.get_mutated_env()
            # new_env.augment(new_env.env_param)

            # create new team
            new_agent_set = OrderedDict()
            for agent in tmp_env.agents:
                # copy params so we can tweak for each agent
                #  this will create exactly the same agents for all starting envs
                #  Is this what we want? What if we want to evolve agents here too?
                tmp_m_params = copy.copy(self.args)

                # grab input and output shape
                tmp_m_params.model_params["input_size"] = np.prod(
                    tmp_env.observation_spaces[agent].shape
                )
                tmp_m_params.model_params["output_size"] = np.prod(
                    tmp_env.action_spaces[agent].shape
                )

                # create initial agent
                #  args dict is because we made a true copy for different inputs/outputs
                #  original POET stuff used same in/outs for all agents
                new_agent_set[agent] = AgtHistory(
                    self.model_factory(
                        env=tmp_env,
                        role=agent,
                        args=tmp_m_params,
                        seed=self.args.master_seed + i,
```

```
                    )
                )
            new_team = TeamHistory(agent_group=new_agent_set)

            # pair env/team and store
            new_env.stats.team = new_team
            self.total_teams[new_team.id] = new_team
            self.total_envs.append(new_env)
            self.brackets[new_env] = new_team

    def reload(self, folder: PathString) -> None:
        """
        Reload a checkpointed PataECEvolution object from the supplied folder.

        Parameters
        ----------
        folder: PathString
            Folder containing checkpoint.

        Side-effects
        ------------
        Yes
            Sets internal variables

        Returns
        -------
        None

        """

        ## We're going to load everything in as a dict and then let the
        ## manifest enforce order.

        # wrap in section logger for formatting
        with SectionLogger(logger, "Reloading") as section:
            ##########
            # Clear Attributes
            ##########
            # Some of the attributes were populated by initial envs/agents/teams
            # Clear them all here so we can add/rebuild later
            self.total_teams = OrderedDict()
            self.total_envs = []
            self.brackets = OrderedDict()
            # self.archived_envs = OrderedDict() # this is already empty at the beginning

            section.print_raw("Cleared Attributes")

            ##########
            # Reload Agents
            ##########
            # Get agents by directory
            agent_dict = dict()
            a_dir = os.path.join(folder, "Agents")
            # list of directories that should be agent names
            agent_paths = [
                name
                for name in os.listdir(a_dir)
                if os.path.isdir(os.path.join(a_dir, name))
            ]

            # agent/role dictionary
            with open(
                os.path.join(a_dir, "ID_Role.json"), mode="r", encoding="utf8"
            ) as file:
                arDict = json.load(file)

            # blank env for agent params
            agt_env = self.env_class()
            agt_env.augment(params=self.env_param_class(self.args))

            # create default agents, fill with reloaded params
            for agt_id in agent_paths:
                # clean copy of params
                tmp_m_params = copy.copy(self.args)
                # input/output
                inp = np.prod(agt_env.observation_spaces[arDict[agt_id]].shape)
                out = np.prod(agt_env.action_spaces[arDict[agt_id]].shape)
                # augment params
                tmp_m_params.model_params["input_size"] = inp
                tmp_m_params.model_params["output_size"] = out

```

```
                # agent
                tmp_agt = AgtHistory(
                    self.model_factory(
                        env=agt_env,
                        role=Role(arDict[agt_id]),
                        args=tmp_m_params,
                        seed=self.args.master_seed,
                    )
                )
                tmp_agt.reload(os.path.join(a_dir, agt_id))
                agent_dict[tmp_agt.id] = tmp_agt

            section.print_raw(f"Reloaded Agents: {str(agent_dict.keys())}")

            ##########
            # Reload Teams
            ##########
            t_dir = os.path.join(folder, "Teams")
            # list of directories that should be team names
            team_id_list = [
                name
                for name in os.listdir(t_dir)
                if os.path.isfile(os.path.join(t_dir, name))
            ]

            # create default teams, load with reloaded agents
            for team_id in team_id_list:
                tmp_team = TeamHistory(agent_group=None)
                tmp_team.reload(
                    team_file=os.path.join(t_dir, team_id), agent_dict=agent_dict
                )
                self.total_teams[tmp_team.id] = tmp_team

            section.print_raw(f"Reloaded Teams: {str(self.total_teams)}")

            ##########
            # Reload Environmnets
            ##########
            tmp_total_envs = dict()
            e_dir = os.path.join(folder, "Environments")
            # list of directories that should be env names
            env_paths = [
                name
                for name in os.listdir(e_dir)
                if os.path.isdir(os.path.join(e_dir, name))
            ]

            # create default envs, load with reloaded params
            for env_id in env_paths:
                # params
                #  this gets reloaded in the EnvHistory wrapper too - why?
                tmp_params = self.env_param_class(self.args)
                tmp_params.reload(folder=os.path.join(e_dir, env_id))

                # env
                tmp_env = EnvHistory(env_param=tmp_params, env_creator=self.env_class)
                tmp_env.reload(
                    folder=os.path.join(e_dir, env_id), team_dict=self.total_teams
                )
                # save so we can sort them later
                tmp_total_envs[tmp_env.id] = tmp_env

            section.print_raw(f"Reloaded Environments: {str(tmp_total_envs)}")

            ##########
            # Reload Manifest
            ##########
            with open(
                os.path.join(folder, "manifest.json"), mode="r", encoding="utf8"
            ) as f:
                manifest = json.load(f, cls=NumpyDecoder)

            # total env list, in proper order
            self.total_envs = [tmp_total_envs[e] for e in manifest["total_envs"]]

            # archived envs, in proper order
            for env_id in manifest["archived_envs"]:
                self.archived_envs[env_id] = tmp_total_envs[env_id]

            # current bracket, in proper order
            for env_id, team_id in manifest["brackets"].items():
```

```
                self.brackets[tmp_total_envs[env_id]] = self.total_teams[team_id]

            # Counters
            self.ANNECS = manifest["ANNECS"]
            self.not_ANNECS = manifest["not_ANNECS"]
            AgtHistory.id_counter = manifest["AgtHistory.id_counter"]
            TeamHistory.id_counter = manifest["TeamHistory.id_counter"]
            EnvHistory.id_counter = manifest["EnvHistory.id_counter"]

            section.print_raw(
                "Reloaded Manifest: envs, archive, brackets, and counters"
            )

            ##########
            ## Reload Manager
            ##########
            self.manager.reload(folder=folder, cur_opts=self.brackets)

            section.print_raw("Reloaded Manager")

            ##########
            # Reload Evolution
            ##########
            self.evolver.reload(folder=folder)

            section.print_raw("Reloaded Evolver")

            ##########
            # Reset global rng
            ##########
            #  this has to be last!
            rstate = manifest["random"]
            rstate[1] = tuple(rstate[1])
            random.setstate(tuple(rstate))

            section.print_raw("Reloaded Global PRNG")

    def optimize(self, epoch: int) -> None:
        """
        Agent Optimization

        This function calls the specific optimization routine for a batch of parameter
        estimates.

        Parameters
        ----------
        epoch: int
            Current main loop iteration

        Side-Effects
        ------------
        Many
            All side effects occur in "EvolutionTemplate()"

        Returns
        -------
        None
        """

        with SectionLogger(logger, "Optimization", f"Epoch: {epoch}") as section:
            # call out to optimization function
            results = self.manager.optimize_chunk(
                tasks=self.brackets.items(), epoch=epoch
            )

            for stats, (env, team) in zip(results, self.brackets.items()):
                # calculate average of average results
                #  mean of a list of lists
                #  this averages every agents' score into a single number
                #  basically a cheap escape
                eval_mean = np.mean(list(stats.values()))

                # store most recent evals
                env.stats.recent_scores.append(eval_mean)
                # transfer threshold is best of N best scores
                env.stats.transfer_threshold = max(env.stats.recent_scores)

                # lifespan best score and agent
                #  only checks the agent after a whole lest of optimization, so we miss
                #  the agents/scores inbetween. Ergo, there is a chance that
                #  transfer_threshold is transiently better than best_score
```

```
                if env.stats.best_score < eval_mean:
                    env.stats.best_score = eval_mean
                    env.stats.best_team = team.copy()
                    # this team is the same as "team", but gets a new id because of the copy
                    #  THIS COPY STATEMENTS ISREQUIRED HERE!!!!
                    #  It must be an independent agent

                    # add to stats
                    self.total_teams[env.stats.best_team.id] = env.stats.best_team

            section.print_time()

    def transfer(self, epoch: int) -> None:
        """EvolutionTemplate Single-Transfer Function

        This function transfers agents within a team, so that the team is always
        comprised of the best agent.

        Parameters
        ----------
        epoch: int
            Current main loop iteration

        Side-Effects
        ------------
        Many
            Teams switch

        Returns
        -------
        None
        """

        with SectionLogger(logger, "Transfer", f"Epoch: {epoch}") as section:
            # if debug, check current bracket against new bracket (below)
            logger.debug("Bracket before transfer:" + str(self.brackets))

            # run eval to prep transfers
            results = self.manager.evaluate(
                tasks=self.brackets.items(), epoch=epoch, verbose=True
            )

            # loop over results and bracket
            for stats, (env, team) in zip(results, self.brackets.items()):
                # average eval for each agent
                eval_means = {k: v.eval_returns_mean for k, v in stats.items()}
                # get name of best agent and its score
                best_agent_name = max(eval_means, key=eval_means.__getitem__)
                best_mean = eval_means[best_agent_name]

                # loop over whole team, replace other agents if the best one is
                #  "better enough"
                #  would a t-test make sense here?
                #  Gonna use 3x standard deviation of the means
                std_mean_3 = 3 * np.std(list(eval_means.values()))
                for agt in team.keys():
                    # check if best is "better enough"
                    if best_mean > (eval_means[agt] + std_mean_3):
                        # log
                        logger.info(
                            f"Team {str(team)} is replaceing agent {agt} with agent {best_agent_name}"
                        )
                        # replace
                        team[agt] = team[best_agent_name].copy()
                        # NOTE: should we be creating a new team here?
                        #       since the agents are technically changed

            # NOTE: The following is not technically correct.
            #       This is where we have this in poet, but really we should update
            #       the total_teams dict whenever we copy and save a new team. I
            #       think this is protecting us from oversights elsewhere, by being
            #       the final function before we checkpoint.

            # check if we're created new teams
            #  whenever a team transfers or beats the best score of a previous team,
            #  new teams are created to log those events.
            # Here, we add those teams to the total team dict
            # loop over active envs
            for env in self.brackets.keys():
                # loop over possible new teams
                #  we don't need their paired team in the bracket because it is
```

```
565                     #  identical to "stats.team" by definition
566                     for team in [env.stats.team, env.stats.best_team]:
567                         # check if it is novel
568                         if team.id not in self.total_teams:
569                             self.total_teams[team.id] = team
570 
571                 # if debug, check new bracket against original bracket (above)
572                 logger.debug("Bracket after transfer:" + str(self.brackets))
573 
574                 section.print_time()
```

7 marco_polo/cores/ePoet

7.1 marco_polo/cores/ePoet/__init__.py

```
1  # Copyright (c) 2023 Mobius Logic, Inc.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #    http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15
16 from .core import Core
```

7.2 marco_polo/cores/ePoet/core.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Single-Player POET Evolution Core """

import argparse  # for type hinting
import json
import logging
import os
import random
import subprocess
import time
from collections import OrderedDict
from collections.abc import Callable  # for type hinting
from typing import Any, cast  # for type hinting
import numpy as np

from marco_polo.algorithms.poet import AllActiveBestScore, PataECEvolution, PoetOptLoop
from marco_polo.envs._base.env_params import EnvParams  # for type hinting
from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder
from marco_polo.tools.section_logging import SectionLogger
from marco_polo.tools.types import Role, PathString  # for type hinting
from marco_polo.tools.wrappers import AgtHistory, EnvHistory, TeamHistory

logger = logging.getLogger(__name__)


################################################################################
## Core Class
################################################################################
class Core:
    """
    ES Core Manager

    This core sets up multiprocessing using an env created from the passed env factory.
    The algorithm is from the ePOET paper, instantiates a custom NN, and optimizes using
    the evolutionary strategy from openAI/Uber. This is a re-org of the ePOET paper with
    no algorithmic changes.

    This is the most basic (single player) Core, from which other Cores, implementing different
    algorithms, can be created.

    Attributes
    ----------
    args : argparse.Namespace
        Simulation parameters
    manager : Manager Object
        Class that handles compute/multi-threading
    evolver : PataECEvolution Object
        This object handles environment evolution
    total_envs : list[env]
        List of all environments in their order of creation
    archived_envs : OrderedDict[int, env]
        Ordered dictionary of envronments that have been archived
    total_teams : OrderedDict[int, env]
        Ordered dictionary of all teams in their order of creation
    brackets : OrderedDict[env, team]
        Pairing of active environments/teams
    ANNECS : int
        Accumulated number of novel environments created and solved
    not_ANNECS : int
        Accumulated number of novel environments created and NOT solved
    """

    def __init__(
        self,
        args: argparse.Namespace,
```

```
        env_factory: Callable[..., Any],
        env_param_factory: Callable[..., Any],
        manager_factory: Callable[..., Any],
        model_factory: Callable[..., Any],
    ) -> None:
        """
        Core Class Initializer

        This function sets up the simulation starting state. It creates a
        compute manager and an evolution manager. It also creates the first
        teams(s) and environment(s).

        Parameters
        ----------
        args : argparse.Namespace
            Seems to be like a dictionary
            This stores all of the simulation parameters
        env_factory: Callable[..., Any],
            Factory for creating environment classes.
        env_param_factory: Callable[..., Any],
            Factory for creating environment parameter classes.
        manager_factory: Callable[..., Any],
            Factory for creating optimization manager classes.
        model_factory: Callable[..., Any],
            Factory for creating model classes.

        Side-effects
        ------------
        Many
            It builds all of the internal objects to start the simulation

        Returns
        -------
        None
        """

        # store sim args for later
        self.args = args
        self.env_factory = env_factory
        self.env_param_factory = env_param_factory
        self.manager_factory = manager_factory
        self.model_factory = model_factory

        # setup environment and parameter classes
        self.env_class = env_factory(args)
        self.env_param_class = env_param_factory(args)

        # Setup manager with any incoming args
        Manager = manager_factory(args=args)
        self.manager = Manager(setup_args=args)

        # Setup the environment evolution class
        self.role_name = self._get_role_name(self.env_class, self.env_param_class)
        self.evolver = PataECEvolution(
            args=args, manager=self.manager, transfer_role=self.role_name
        )

        # total env/team trackers
        self.total_teams: OrderedDict[str, TeamHistory] = OrderedDict()
        self.total_envs: list[EnvHistory] = []
        # active environment/team pairing
        self.brackets: OrderedDict[EnvHistory, TeamHistory] = OrderedDict()
        # archive
        self.archived_envs: OrderedDict[str, EnvHistory] = OrderedDict()

        # create initial env/team pairs (number given by args.num_start_envs)
        self._create_initial_env_teams(
            EnvCreator=self.env_class, env_param_class=self.env_param_class
        )

        # stats
        # accumulated number of novel environments created and solved
        self.ANNECS = 0  # pylint: disable=invalid-name
        # accumulated but not solved
        self.not_ANNECS = 0  # pylint: disable=invalid-name

    ########################################
    ## Funcs
    ########################################
    def _get_role_name(
        self, EnvCreator: Callable[..., Any], env_param_class: Callable[..., EnvParams]
```

```
    ) -> Role:
        tmp = EnvCreator()
        tmp.augment(params=env_param_class(self.args))
        if hasattr(tmp, "possible_agents"):
            if len(tmp.possible_agents) > 1:
                raise ValueError("Too many players, POET expects a 1 player game.")
            return cast(
                Role, tmp.possible_agents[0]
            )  # mypy can't figure out the type without cast

        raise AttributeError(
            "env.possible_agents not found, please ensure your env implements the inferface properly."
        )

    def _update_model_params(self, env: Any) -> None:
        """Update input and output size based on env class."""
        model_params = self.args.model_params
        if type(env.observation_spaces) is dict:
            a = env.possible_agents[0]
            model_params["input_size"] = np.prod(env.observation_spaces[a].shape)
            model_params["output_size"] = np.prod(env.action_spaces[a].shape)
        else:
            model_params["input_size"] = np.prod(env.observation_spaces.shape)
            model_params["output_size"] = np.prod(env.action_spaces.shape)

    def _create_initial_env_teams(
        self, EnvCreator: Callable[..., Any], env_param_class: Callable[..., EnvParams]
    ) -> None:
        """Create the initial environments and teams."""

        # temporary environment to update model params
        tmp_env = EnvCreator()
        tmp_env.augment(params=env_param_class(self.args))
        self._update_model_params(tmp_env)

        # Make sure we don't start with more than the maximum number of envs
        n_start_envs = min(self.args.max_active_envs, self.args.num_start_envs)

        for i in range(n_start_envs):
            seed = self.args.master_seed + i

            # create env
            env_params = env_param_class(self.args)
            env = EnvCreator()
            env.augment(params=env_params)
            env_history = EnvHistory(env_param=env_params, env_creator=EnvCreator)

            # create team - in this case it has one agent in a dict
            model = self.model_factory(
                env=env, role=self.role_name, args=self.args, seed=seed
            )
            agent = AgtHistory(model)
            agent_group = {self.role_name: agent}
            team = TeamHistory(agent_group=agent_group)

            # pair env/team and store
            env_history.stats.team = team
            self.total_teams[team.id] = team
            self.total_envs.append(env_history)
            self.brackets[env_history] = team

    def checkpoint(self, name: str) -> None:
        """
        Save a checkpoint of the PataECEvolution object

        Parameters
        ----------
        folder: PathString
            Path to folder that will contain checkpoint information

        Side-effects
        ------------
        None

        Returns
        -------
        None

        """
        with SectionLogger(logger, "Checkpointing", f"Epoch: {name}") as section:
            filename = self.args.logtag + "cp-" + name
```

```
            folder = os.path.join(self.args.log_file, "Checkpoints", filename)
            if os.path.isdir(folder):
                section.print_raw(
                    f"Directory already exists at {folder}. Overwriting checkpoint."
                )
            else:
                section.print_raw(f"Saving checkpoint to {folder}")

            os.makedirs(folder, exist_ok=True)

            with open(
                os.path.join(folder, "args.json"), mode="w", encoding="utf8"
            ) as f:
                json.dump(self.args.__dict__, f, cls=NumpyEncoder)

            manifest: dict[str, Any] = {}
            # manifest["args"] = dict(self.args.__dict__)
            logger.debug(f"Bracket Check: {self.brackets}")

            manifest["total_envs"] = [env.id for env in self.total_envs]
            manifest["total_teams"] = list(self.total_teams.keys())
            manifest["archived_envs"] = list(self.archived_envs.keys())
            manifest["brackets"] = {
                env.id: team.id for env, team in self.brackets.items()
            }
            manifest["ANNECS"] = self.ANNECS
            manifest["not_ANNECS"] = self.not_ANNECS
            manifest["AgtHistory.id_counter"] = AgtHistory.id_counter
            manifest["TeamHistory.id_counter"] = TeamHistory.id_counter
            manifest["EnvHistory.id_counter"] = EnvHistory.id_counter
            manifest["random"] = random.getstate()

            file_path = os.path.join(folder, "manifest.json")
            with open(file_path, mode="w", encoding="utf-8") as f:
                json.dump(manifest, f, cls=NumpyEncoder)

            ## Checkpoint Environmnets
            for env in self.total_envs:
                env.checkpoint(folder)

            ## Checkpoint Teams and Agents
            arDict = dict()
            for team in self.total_teams.values():
                # checkpoint team
                team.checkpoint(folder)
                for role, agtHist in team.items():
                    # checkpoint agent
                    agtHist.checkpoint(folder)
                    # add agent id/role to dict
                    arDict[agtHist.id] = role

            # Log agent id/role mapping
            #  This is primarily used in multi-agent reloads
            file_path = os.path.join(folder, "Agents", "ID_Role.json")
            with open(file_path, mode="w", encoding="utf-8") as f:
                json.dump(arDict, f)

            ## Checkpoint Manager
            self.manager.checkpoint(folder=folder, cur_opts=self.brackets.items())

            ## Checkpoint Poet Tools
            self.evolver.checkpoint(folder)

            # Compress checkpoint if requested
            if self.args.checkpoint_compression:
                section.print_raw("Compressing checkpoint")
                self._compress_checkpoint(folder)

    def _compress_checkpoint(self, folder: PathString) -> None:
        """Use lbzip2 to compress checkpoint folder using external tool."""
        # NOTES on json compression, if we ever go that route
        #  it'll probably be faster than this, since it compresses from memory
        # NOTE: https://martinheinz.dev/blog/57
        # NOTE: https://medium.com/@busybus/zipjson-3ed15f8ea85d
        # NOTE: https://janakiev.com/blog/python-json/
        # NOTE:
            https://stackoverflow.com/questions/17742789/running-multiple-bash-commands-with-subprocess

        # NOTE: https://stackoverflow.com/questions/45621476/python-tarfile-slow-than-linux-command
        #       This is why we subprocess instead of using the python tarfile module
        # NOTE: https://anaconda.org/conda-forge/lbzip2
```

```
        ## Compress checkpoint
        # build command
        # -C change to the following directory
        # -c output to this place (sent to stdout here)
        # -f read from this file
        # | lbzip2 pipe stdin to lbzip2
        # -9 lbzip2 level 9
        # -n number of cores to use
        # > output to here
        tar_args = (
            "tar -C "
            + os.path.dirname(p=folder)
            + " --remove-files -cf - "
            + os.path.basename(p=folder)
            + " | lbzip2 -9 -n "
            + str(self.args.num_workers)
            + " > "
            + folder
            + ".tar.bz2"
        )

        # send to shell
        subprocess.run(args=tar_args, shell=True)

    def reload(self, folder: PathString) -> None:
        """
        Reload a checkpointed PataECEvolution object from the supplied folder.

        Parameters
        ----------
        folder: PathString
            Folder containing checkpoint.

        Side-effects
        ------------
        Yes
            Sets internal variables

        Returns
        -------
        None

        """
        ## We're going to load everything in as a dict and then let the
        ## manifest enforce order.

        # wrap in section logger for formatting
        with SectionLogger(logger, "Reloading") as section:
            ##########
            # Clear Attributes
            ##########
            # Some of the attributes were populated by initial envs/agents/teams
            # Clear them all here so we can add/rebuild later
            self.total_teams = OrderedDict()
            self.total_envs = []
            self.brackets = OrderedDict()
            # self.archived_envs = OrderedDict() # this is already empty at the beginning

            section.print_raw("Cleared Attributes")

            ##########
            # Reload Agents
            ##########
            # Get agents by directory
            agent_dict = dict()
            agent_dir = os.path.join(folder, "Agents")
            # list of directories that should be agent names
            agent_paths = [
                name
                for name in os.listdir(agent_dir)
                if os.path.isdir(os.path.join(agent_dir, name))
            ]

            # blank env for agent params
            agt_env = self.env_class()
            agt_env.augment(params=self.env_param_class(self.args))

            # create default agents, fill with reloaded params
            for agent_id in agent_paths:
                # agent
                tmp_agent = AgtHistory(
```

```
                    self.model_factory(
                        env=agt_env,
                        role=self.role_name,
                        args=self.args,
                        seed=self.args.master_seed,
                    )
                )
                tmp_agent.reload(os.path.join(agent_dir, agent_id))
                agent_dict[tmp_agent.id] = tmp_agent

            section.print_raw(f"Reloaded Agents: {str(agent_dict.keys())}")

            ##########
            # Reload Teams
            ##########
            # list of directories that should be team names
            t_dir = os.path.join(folder, "Teams")
            # list of directories that should be team names
            team_id_list = [
                name
                for name in os.listdir(t_dir)
                if os.path.isfile(os.path.join(t_dir, name))
            ]

            # create default teams, load with reloaded agents
            for team_id in team_id_list:
                tmp_team = TeamHistory(agent_group=None)
                tmp_team.reload(
                    team_file=os.path.join(t_dir, team_id), agent_dict=agent_dict
                )
                self.total_teams[tmp_team.id] = tmp_team

            section.print_raw(f"Reloaded Teams: {str(self.total_teams)}")

            ##########
            # Reload Environmnets
            ##########
            tmp_total_envs = dict()
            e_dir = os.path.join(folder, "Environments")
            # list of directories that should be env names
            env_paths = [
                name
                for name in os.listdir(e_dir)
                if os.path.isdir(os.path.join(e_dir, name))
            ]

            # create default envs, load with reloaded params
            for env_id in env_paths:
                # params
                #  this gets reloaded in the EnvHistory wrapper too - why?
                tmp_params = self.env_param_class(self.args)
                tmp_params.reload(folder=os.path.join(e_dir, env_id))

                # env
                tmp_env = EnvHistory(env_param=tmp_params, env_creator=self.env_class)
                tmp_env.reload(
                    folder=os.path.join(e_dir, env_id), team_dict=self.total_teams
                )
                # save so we can sort them later
                tmp_total_envs[tmp_env.id] = tmp_env

            section.print_raw(f"Reloaded Environments: {str(tmp_total_envs)}")

            ##########
            # Reload Manifest
            ##########
            with open(
                os.path.join(folder, "manifest.json"), mode="r", encoding="utf8"
            ) as f:
                manifest = json.load(f, cls=NumpyDecoder)

            # total env list, in proper order
            self.total_envs = [tmp_total_envs[e] for e in manifest["total_envs"]]

            # archived envs, in proper order
            for env_id in manifest["archived_envs"]:
                self.archived_envs[env_id] = tmp_total_envs[env_id]

            # current bracket, in proper order
            for env_id, team_id in manifest["brackets"].items():
                self.brackets[tmp_total_envs[env_id]] = self.total_teams[team_id]
```

```
483
484             # Counters
485             self.ANNECS = manifest["ANNECS"]
486             self.not_ANNECS = manifest["not_ANNECS"]
487             AgtHistory.id_counter = manifest["AgtHistory.id_counter"]
488             TeamHistory.id_counter = manifest["TeamHistory.id_counter"]
489             EnvHistory.id_counter = manifest["EnvHistory.id_counter"]
490
491             section.print_raw(
492                 "Reloaded Manifest: envs, archive, brackets, and counters"
493             )
494
495             ##########
496             ## Reload Manager
497             ##########
498             self.manager.reload(folder=folder, cur_opts=self.brackets)
499
500             section.print_raw("Reloaded Manager")
501
502             ##########
503             # Reload Evolution
504             ##########
505             self.evolver.reload(folder=folder)
506
507             section.print_raw("Reloaded Evolver")
508
509             ##########
510             # Reset global rng
511             ##########
512             #  this has to be last!
513             rstate = manifest["random"]
514             rstate[1] = tuple(rstate[1])
515             random.setstate(tuple(rstate))
516
517             section.print_raw("Reloaded Global PRNG")
518
519     def epoch_update(self, epoch: int) -> None:
520         """
521         Currently does nothing, here for reasons?
522
523         Parameters
524         ----------
525         epoch: int
526             Current simulation time
527
528         Side-Effects
529         ------------
530         None
531
532         Returns
533         -------
534         None
535         """
536         # safe-keeping for Curiosity currently
537
538     def reproduce(self, epoch: int) -> None:
539         """
540         Basic Reproduction Function
541
542         This function handles the current environment/team pairings, tracks the
543         archived environments, total environments/teams, and has some basic statistics.
544         The actual work is passed down to Evolve()
545
546         Parameters
547         ----------
548         epoch: int
549             Current simulation time
550
551         Side-Effects
552         ------------
553         Many
554             This function directly updates internal tracking variables and statistics
555             Evolve() modifies the environments' and teams' internal states
556
557         Returns
558         -------
559         None
560         """
561
562         with SectionLogger(logger, "Evolution", f"Epoch: {epoch}") as section:
563             (
```

```
                new_envs,
                new_teams,
                self.brackets,
                AN,
                no_AN,
                to_archive,
            ) = self.evolver.Evolve(
                bracket=self.brackets,
                archived_envs=self.archived_envs,
                epoch=epoch,
                repro_threshold=self.args.repro_threshold,
                max_active_envs=self.args.max_active_envs,
                max_children=self.args.num_proposal_envs,
                max_admitted=self.args.max_admitted_envs,
            )

            # update internal metrics
            self.total_envs += new_envs
            self.total_teams.update(new_teams)
            self.ANNECS += AN
            self.not_ANNECS += no_AN
            self.archived_envs.update(to_archive)

            # log stats
            section.print_status_banner("Summary")
            for line in self.get_summary():
                section.print_raw(line)

    def get_summary(self) -> list[str]:
        """Return summary of the system

        example usage might be:
        summary = self.get_summary()
        for line in summary:
            print(line)

        Returns
        -------
        list[str]
            list of strings describing the system. Each item is a single line.
        """
        return [
            f"Total Environments: {str(self.total_envs)}",
            f"Archived Environments: {str(self.archived_envs)}",
            f"Current Bracket: {str(self.brackets)}",
            f"ANNECS: {self.ANNECS}",
            f"Not ANNECS: {self.not_ANNECS}",
        ]

    def optimize(self, epoch: int) -> None:
        """
        Team Optimization

        This function calls the specific optimization routine for a batch of parameter
        estimates.

        Parameters
        ----------
        epoch: int
            Current main loop iteration

        Side-Effects
        ------------
        Many
            All side effects occur in "PoetOptLoop()"

        Returns
        -------
        None
        """

        with SectionLogger(logger, "Optimization", f"Epoch: {epoch}") as section:
            # call out to optimization function
            PoetOptLoop(
                tasks=list(self.brackets.items()),
                epoch=epoch,
                manager=self.manager,
                verbose=True,
            )
            # NOTE: Are we missing a total_teams update in here?
            #       We might be safe because the transfers do it.
```

```
645
646             section.print_time()
647
648     def transfer(self, epoch: int) -> None:
649         """
650         Peform Transfer Mechanism
651
652         This work is passed down to "AllActiveBestScore()".
653
654         Parameters
655         ----------
656         epoch: int
657             Current main loop iteration
658
659         Side-Effects
660         ------------
661         Many
662             This function directly updates the env/team pairing bracket
663             "AllActiveBestScore()" modifies environment/team internals
664
665         Returns
666         -------
667         None
668         """
669
670         with SectionLogger(logger, "Transfer", f"Epoch: {epoch}") as section:
671             logger.debug("Bracket before transfer:" + str(self.brackets))
672
673             AllActiveBestScore(
674                 manager=self.manager,
675                 bracket=self.brackets,
676                 transfer_role=self.role_name,
677                 epoch=epoch,
678             )
679
680             logger.debug("Bracket after transfer:" + str(self.brackets))
681
682             # NOTE: The following is not technically correct.
683             #       This is where we put it ages ago, but really we should update
684             #       the total_teams dict whenever we copy and save a new team. I
685             #       think this is protecting us from oversights elsewhere, by being
686             #       the final function before we checkpoint.
687
688             # check if we're created new teams
689             #  whenever a team transfers or beats the best score of a previous team,
690             #  new teams are created to log those events.
691             # Here, we add those teams to the total team dict
692             # loop over active envs
693             for env in self.brackets.keys():
694                 # loop over possible new teams
695                 #  we don't need their paired team in the bracket because it is
696                 #  identical to "stats.team" by definition
697                 for team in [env.stats.team, env.stats.best_team]:
698                     # check if it is novel
699                     if team.id not in self.total_teams:
700                         self.total_teams[team.id] = team
701
702             section.print_time()
703
704     def evolve_population(self, start_epoch: int = 0, sim_start: float = 0.0) -> None:
705         """
706         Main Optimization Loop
707
708         For each step of the epoch evolves the env's, optimizing the functions,
709         and evalutes the transfer potential. All actions are implemented by the Core
710         object, this function simply calls them in the appropriate order.
711
712         Parameters
713         ----------
714         start_epoch : int, default=0
715             Number of start_epoch to start on, important for reloads
716         sim_start : float, default=0.0
717             Start time of the simulation
718
719         Side-Effects
720         ------------
721         None
722
723         Returns
724         -------
725         None
```

```
            Output is written to disk in a directory specific within args.
        """

        # Main Loop
        #  Algorithm 2 in POET paper (https://doi.org/10.48550/arXiv.1901.01753)
        for ep in range(start_epoch, self.args.poet_epochs):
            section = SectionLogger(logger, "Epoch", f"Epoch: {ep}", print_start=False)

            # Environment Evolution Step
            self.epoch_update(epoch=ep)

            # Reproduction Step
            if (ep > 0) and ((ep % self.args.reproduction_interval) == 0):
                self.reproduce(epoch=ep)

            # Optimization Step
            #  this function optimizes for n iterations until the transfer step
            #  this is a change from the original algorithm
            self.optimize(epoch=ep)

            # Transfer Step
            self.transfer(epoch=ep)

            # Checkpoint
            if (ep % self.args.checkpoint_interval) == 0:
                self.checkpoint(name=str(ep))

            # log epoch
            epoch_end = time.time()
            section.print_status_banner("")
            section.print_time()  # for this epoch only
            section.print_raw(f"Simulation Time: {(epoch_end - sim_start):.2f} seconds")
            section.print_end_banner()

        self.report()

    def report(self) -> None:
        report_filename = f"report_{time.time()}.rpt"
        file_path = os.path.join(self.args.log_file, report_filename)
        with open(file_path, mode="w", encoding="utf-8") as file:
            file.writelines("Environments: \n")

            for env in self.total_envs:
                out = f"Env_ID: {env.id}, "

                for k, v in env.stats.__dict__.items():
                    out += f"{k}: {str(v)}, "

                file.writelines(out + "\n")

            file.writelines("\nTeams: \n")

            for team in self.total_teams.values():
                out = f"Team_ID: {team.id}, "
                for role, agent in team.items():
                    theta_hash = hash(agent.get_theta().tobytes())
                    out += f"{role}: {str(agent.id)} Theta Hash: {theta_hash}, "

                file.writelines(out + "\n")
```

8 marco_polo/envs/BipedalWalker

8.1 marco_polo/envs/BipedalWalker/__init__.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


from .env import get_env_class
from .cppn import get_env_param_class
```

8.2 marco_polo/envs/BipedalWalker/bpw_constants.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Constants used by both the BipedalWalker and its renderers"""


SCALE = 30.0  # affects how fast-paced the game is, forces should be adjusted as well

# Output viewport (width and height) in pixels
VIEWPORT_W = 600
VIEWPORT_H = 400

TERRAIN_STEP = 14 / SCALE  # length of a step in scaled units
TERRAIN_LENGTH = 200  # length of terrain course, in steps
TERRAIN_HEIGHT = VIEWPORT_H / SCALE / 4  # base terrain height in scaled units
TERRAIN_STARTPAD = 20  # padding at start of course, in steps

TERRAIN_GRASS = 10  # how long the grass spots are, in steps
```

8.3 marco_polo/envs/BipedalWalker/bpw_renderer.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Default rendering class for bipedal walker"""
import math
from typing import Any, cast, Optional, Union

# This is here to discard the ill-advised 'Hello' message that pygame prints
# by default on import, plus some deprecation warnings.
import contextlib

# redirecting stdout gets rid of pygame spam,
# redirecting stderr gets rid of warning for deprecated package management
# from pygame and/or dependencies
with contextlib.redirect_stdout(None), contextlib.redirect_stderr(None):
    import pygame

import imageio
from Box2D.b2 import circleShape  # type: ignore[import]

import numpy as np

from marco_polo.envs.BipedalWalker.bpw_constants import (
    SCALE,
    TERRAIN_HEIGHT,
    TERRAIN_LENGTH,
    TERRAIN_STEP,
    VIEWPORT_H,
    VIEWPORT_W,
)
from marco_polo.tools.types import RenderFrame


class Viewport:
    """Data for a viewport definintion (x and y box limits)"""

    def __init__(
        self, x_min: float, x_max: float, y_min: float, y_max: float, scale: float
    ) -> None:
        self.x_min = x_min
        self.x_max = x_max
        self.y_min = y_min
        self.y_max = y_max
        self.scale = scale

    @property
    def pixel_x_offset(self) -> float:
        """offset x value, in pixels. i.e. x_min scaled to pixels"""
        return self.x_min * self.scale

    @property
    def pixel_x_min(self) -> float:
        """minimum x value, in pixels."""
        return 0

    @property
    def pixel_x_max(self) -> float:
        """maximum x value, in pixels."""
        return self.scale * self.get_size()[0]

    @property
    def pixel_y_min(self) -> float:
        """minimum y value, in pixels."""
        return 0

    @property
```

```
79      def pixel_y_max(self) -> float:
80          """maximum y value, in pixels."""
81          return self.scale * self.get_size()[1]
82
83      def __repr__(self) -> str:
84          return f"Viewport(xmin={self.x_min}, xmax={self.x_max}, y_min={self.y_min}, y_max={self.y_max},
                scale={self.scale})"
85
86      def get_size(self) -> tuple[float, float]:
87          """Return the size (width and height) of the viewport
88
89          Returns
90          -------
91          tuple[float, float]
92              width, height
93          """
94          return self.x_max - self.x_min, self.y_max - self.y_min
95
96      def get_pixel_size(self) -> tuple[float, float]:
97          """Return the size (width and height) of the viewport, in pixels
98
99          Returns
100         -------
101         tuple[float, float]
102             width, height
103         """
104         return self.pixel_x_max - self.pixel_x_min, self.pixel_y_max - self.pixel_y_min
105
106
107 ColorType = tuple[int, int, int]
108 PointType = tuple[float, float]
109 PolygonType = list[PointType]
110
111
112 def shift_point(point: PointType, delta_x: float, delta_y: float = 0) -> PointType:
113     """Return a point that is shifted by the offsets
114
115     Parameters
116     ----------
117     point: PointType
118         The point to shift
119     delta_x: float
120         Change in X value
121     delta_y: float, default = 0
122         Change in Y value
123
124     Returns
125     -------
126     PointType i.e. (float, float)
127         The shifted point
128     """
129     return point[0] + delta_x, point[1] + delta_y
130
131
132 class BipedalWalkerRenderer:
133     """Default renderer for Bipedal Walker class
134
135     This is a pygame rewrite of the original rendering. It is intended to match
136     the original output as much as possible.
137
138     Attributes
139     ----------
140     self.parent: BipedalWalkerCustom
141         Owner of this renderer
142     self.cloud_poly: list[tuple[PolygonType, float, float]]
143         Clouds to render
144     self.lidar_render: int
145         Step in render (used for deciding when to show lidar)
146     """
147
148     def __init__(self, parent: Any, mode: str = "rgb_array") -> None:
149         """Create the renderer with the given parent
150
151         The parent is the simulation environment. This renderer
152         will pull the scroll position, the terrain, the walker
153         object, and the lidar information from the parent. In
154         addition, it will use the parent's random number generator.
155
156         Parameters
157         ----------
158         parent: BipedalWalkerCustom
```

```
            the environment this class will render
        mode: str, default = "rgb_array"
            The rending mode. If the value is "rgb_array", this will
            return the frame as data. Anything else will not.
        """
        self.parent = parent
        self.render_mode = mode
        if self.render_mode != "rgb_array":
            raise NotImplementedError("only 'rgb_array' is supported as a render mode")

        self.scroll = self.parent.scroll
        self.surface = cast(pygame.Surface, None)  # this fixes a bunch of hassles
        self.cloud_poly: list[tuple[PolygonType, float, float]] = []
        self.lidar_render = 0
        self.scale_facter = SCALE
        self.reset()

    def scale_point(self, point: PointType) -> PointType:
        return point[0] * self.scale_facter, point[1] * self.scale_facter

    def draw_polygon(self, poly: PolygonType, color: ColorType) -> None:
        pygame.draw.polygon(self.surface, color=color, points=poly)

    def draw_lines(
        self, poly: PolygonType, color: ColorType, closed: bool, width: int
    ) -> None:
        pygame.draw.lines(
            self.surface, color=color, closed=closed, points=poly, width=width
        )

    def render_whole_env(
        self, filename: str, generate_terrain: bool = True, generate_clouds: bool = True
    ) -> None:
        """Save image of entire environment at once (without walker)

        Instead of an animated gif, as in the main rendering, this
        creates a single wide image showing all the terrain at once.

        The terrain and clouds are optionally generated if requested.
        This may or may not be needed, depending on how/where this is
        called.

        Parameters
        ----------
        filename: str
            filename to write file to. Will write to the current
            directory unless the name contains another path
        generate_terrain: bool, default = True
            whether to (re-)generate terrain prior to rendering.
            defaults to True
        generate_clouds: bool, default = True
            whether to (re-)generate clouds prior to rendering.
            defaults to True

        Side-effects
        ------------
        A file is written to disk.
        Additionally, if generate_terrain and/or generate_clouds are
        True, this will (re-)generate those items in the environment.
        That may change those values and will advance the random
        sequence.
        """
        # The surface used for this function is a different size than
        # the main surface, so back up the existing one.
        # It will be restored at the end of the function.
        backup_surface = self.surface

        width = round((TERRAIN_STEP * TERRAIN_LENGTH) * SCALE)
        viewport = Viewport(0, width, 0, VIEWPORT_H, SCALE)
        self.surface = pygame.Surface((width, VIEWPORT_H))

        if generate_terrain:
            self.parent.generate_terrain()
        if generate_clouds:
            self._generate_clouds()
        self._draw_sky(viewport)

        # cloud function has hard coded view limits
        cloud_color = (255, 255, 255)
        for poly, _, _ in self.cloud_poly:
            self.draw_polygon(poly, cloud_color)
```

```
240
241         self._draw_terrain(viewport, self.parent.terrain_poly)
242         self._draw_flag(viewport)
243
244         data = self.get_image_data()
245         imageio.imwrite(filename, data)
246         self.surface = backup_surface
247
248     def get_image_data(self) -> RenderFrame:
249         data = pygame.surfarray.array3d(self.surface)
250         data = np.rot90(data, axes=(0, 1))
251         return data
252
253     def _generate_clouds(self) -> None:
254         """Generate clouds to be displayed in the rendering"""
255         self.cloud_poly = []
256         for _ in range(TERRAIN_LENGTH // 20):
257             x_base = self.parent.np_random.uniform(0, TERRAIN_LENGTH) * TERRAIN_STEP
258             y_base = VIEWPORT_H / SCALE * 3 / 4
259             poly: PolygonType = [
260                 self.scale_point(
261                     (
262                         x_base
263                         + 15 * TERRAIN_STEP * math.sin(3.14 * 2 * a / 5)
264                         + self.parent.np_random.uniform(0, 5 * TERRAIN_STEP),
265                         y_base
266                         + 5 * TERRAIN_STEP * math.cos(3.14 * 2 * a / 5)
267                         + self.parent.np_random.uniform(0, 5 * TERRAIN_STEP),
268                     )
269                 )
270                 for a in range(5)
271             ]
272             x_vals = [p[0] for p in poly]
273             x_min: float = min(x_vals)
274             x_max: float = max(x_vals)
275             self.cloud_poly.append((poly, x_min, x_max))
276
277     def reset(self) -> None:
278         """Reset the renderer
279
280         Side-Effects
281         ------------
282         This will reset the lidar viewing sequence and regenerate
283         the clouds.
284         """
285         self.lidar_render = 0
286         # self._set_terrain_number()
287         self._generate_clouds()
288
289     def render(self, *args: Any, **kwargs: Any) -> Union[None, RenderFrame]:
290         """Render the scene"""
291         return self._render(*args, **kwargs)
292
293     def _draw_sky(self, viewport: Viewport) -> None:
294         """Draw the sky background"""
295         width, height = viewport.get_size()
296         x_min, x_max = self.scale_point((0, width))
297         y_min, y_max = self.scale_point((0, height))
298         poly = [(x_min, y_min), (x_max, y_min), (x_max, y_max), (x_min, y_max)]
299         sky_color = (230, 230, 255)
300         self.draw_polygon(poly, sky_color)
301
302     def _draw_clouds(
303         self, viewport: Viewport, clouds: list[tuple[PolygonType, float, float]]
304     ) -> None:
305         """Draw the clouds"""
306         half_scroll = 0.5 * self.scroll
307
308         # the x position needs to be offset by this much for proper tracking
309         x_adjust = half_scroll - viewport.pixel_x_offset
310         cloud_color = (255, 255, 255)
311         for poly, x_min, x_max in clouds:
312             if x_max < half_scroll:
313                 continue
314             if x_min > half_scroll + VIEWPORT_W:
315                 continue
316             new_poly = [shift_point(p, delta_x=x_adjust) for p in poly]
317             self.draw_polygon(new_poly, cloud_color)
318
319     def _draw_terrain(self, viewport: Viewport, terrain: Any) -> None:
320         """Draw the ground terrain"""
```

```
321             for poly, color in terrain:
322                 if poly[1][0] < viewport.pixel_x_offset:
323                     continue
324                 if poly[0][0] > viewport.x_max * SCALE:
325                     continue
326                 new_poly = [shift_point(p, delta_x=-viewport.pixel_x_offset) for p in poly]
327                 self.draw_polygon(new_poly, color)
328 
329         def _draw_lidar(self, viewport: Viewport, lidar: Any) -> None:
330             """Draw the lidar line, if appropriate"""
331             self.lidar_render = (self.lidar_render + 1) % 50
332             i = self.lidar_render
333             if i < 2 * len(lidar):
334                 l = lidar[i] if i < len(lidar) else lidar[len(lidar) - i - 1]
335                 start_pos = self.scale_point(shift_point(l.point1, delta_x=-viewport.x_min))
336                 end_pos = self.scale_point(shift_point(l.point2, delta_x=-viewport.x_min))
337                 self.draw_lines(
338                     [start_pos, end_pos], color=(255, 0, 0), closed=False, width=1
339                 )
340 
341         def _draw_objects(self, viewport: Viewport) -> None:
342             """Draw the objects of the scene"""
343             for obj in self.parent.drawlist:
344                 for fixture in obj.fixtures:
345                     trans = fixture.body.transform
346                     if isinstance(fixture.shape, circleShape):
347                         raise NotImplementedError("circle rendering not supported")
348                     path = [trans * v for v in fixture.shape.vertices]
349                     shifted_path = [
350                         shift_point(point, delta_x=-viewport.x_min) for point in path
351                     ]
352                     scaled_path = [self.scale_point(point) for point in shifted_path]
353                     if len(path) > 2:
354                         self.draw_polygon(scaled_path, obj.color1)
355                     else:
356                         # paths of len two would be a thin line, which would be
357                         # overwritten by the line drawing below, so they are ignored
358                         pass
359                     self.draw_lines(
360                         poly=scaled_path, color=obj.color2, closed=True, width=2
361                     )
362 
363         def _draw_flag(self, viewport: Viewport) -> None:
364             """Draw the flag at the start of the course"""
365             flag_y1 = TERRAIN_HEIGHT * SCALE
366             flag_y2 = flag_y1 + 50
367             flag_x = (TERRAIN_STEP * 3) * SCALE - viewport.pixel_x_offset
368 
369             pole = [(flag_x, flag_y1), (flag_x, flag_y2)]
370             self.draw_lines(poly=pole, color=(0, 0, 0), closed=False, width=2)
371 
372             flag = [(flag_x, flag_y2), (flag_x, flag_y2 - 10), (flag_x + 25, flag_y2 - 5)]
373             self.draw_polygon(poly=flag, color=(230, 51, 0))
374             self.draw_lines(poly=flag, color=(0, 0, 0), closed=True, width=2)
375 
376         def _close_renderer(self) -> None:
377             """Close and destroy the surface, if it exists"""
378             if self.surface is not None:
379                 self.surface = cast(pygame.Surface, None)
380 
381         def _draw_scene(self, viewport: Viewport, kwargs: dict[str, Any]) -> None:
382             """Draw the objects in the scene"""
383             self._draw_sky(viewport)
384             self._draw_clouds(viewport, self.cloud_poly)
385             self._draw_terrain(viewport, self.parent.terrain_poly)
386             self._draw_lidar(viewport, self.parent.lidar)
387             self._draw_objects(viewport)
388             self._draw_flag(viewport)
389 
390         def _render(self, close: bool = False, **kwargs: Any) -> Union[None, RenderFrame]:
391             """Render the current scene
392 
393             Parameters
394             ----------
395             close: bool, default = False
396                 Whether to close the surface. If True, the surface is closed and
397                 destroyed.
398 
399             Returns
400             -------
401             If close is True, None is returned.
```

```
        Otherwise, if self.render_mode is "rgb_array", the frame will be returned
        as a byte array.
        In any other case, the status of the viewing window will be returned.
        This assumes a view window that may be open, but not visible.
        """
        if close:
            self._close_renderer()
            return None

        self.scroll = self.parent.scroll
        x_min = self.scroll
        x_max = x_min + VIEWPORT_W / SCALE
        y_min = 0
        y_max = VIEWPORT_H / SCALE
        viewport = Viewport(x_min, x_max, y_min, y_max, SCALE)

        if self.surface is None:
            self.surface = pygame.Surface((VIEWPORT_W, VIEWPORT_H))

        self._draw_scene(viewport, kwargs)

        return self.get_image_data()

    def render_all(
        self, data: list[Any], result: Optional[tuple[int, bool]]
    ) -> list[RenderFrame]:
        """Render all frames of the environment."""
        raise NotImplementedError("render_all not supported by base renderer")
```

8.4 marco_polo/envs/BipedalWalker/bpw_renderer_updated.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Updated rendering class for bipedal walker"""

from typing import Any, Optional

import pygame

from marco_polo.envs.BipedalWalker.bpw_constants import (
    SCALE,
    TERRAIN_HEIGHT,
    TERRAIN_STARTPAD,
    TERRAIN_STEP,
    VIEWPORT_H,
    VIEWPORT_W,
)
from marco_polo.envs.BipedalWalker.bpw_renderer import (
    BipedalWalkerRenderer,
    PolygonType,
    shift_point,
    Viewport,
)


class BipedalWalkerRendererEnhanced(BipedalWalkerRenderer):
    """Updated renderer for BipedalWalkerCustom

    This will function as a drop in replacement for the original
    renderer. However, some optional features require minor changes
    to the calling code.

    This has the following changes over the orginal renderer:
    * scales the vertical position to keep the walker in frame.
    * draws a path, showing the full terrain with a moving viewport.
    * adds an end flag marker.
    * add a frame step progress bar showing how far the run has progressed
      in simulation steps.
    * displays the current reward.

    Lastly, this class supports render_all(), which renders all of the
    frames in one call. When using this, the renderer will make these
    additional changes:
    * padding frames will be added at the end to of the video to pause
      it briefly at the end when looping the video.
    * the outcome status of the run will be shown as a color coding on
      the progress bar.
    * The final reward will be shown throughout, if known
    """

    def __init__(
        self, parent: Any, n_padding_frames: int = 10, mode: str = "rgb_array"
    ) -> None:
        """Initialize the object

        Parameters
        ----------
        parent: BipedalWalkerCustom
            The environment to render
        n_padding_frames: int, default = 10
            The number of frames to pad the end of the video with
        mode: str, default = "rgb_array"
            The rending mode. If the value is "rgb_array", this will
            return the frame as data. Anything else will not.
        """
        super().__init__(parent, mode=mode)
```

```
79          self._n_padding_frames = n_padding_frames
80          self._colors = {
81              "black": (0, 0, 0),
82              "white": (255, 255, 255),
83              "gray": (127, 127, 127),
84          }
85          pygame.font.init()
86          self.font = pygame.font.SysFont(None, 24)  # type: ignore  # None is default font
87
88      def render_all(
89          self, data: list[Any], result: Optional[tuple[int, bool]]
90      ) -> list[Any]:
91          """Render all of the frames at once
92
93          Given the data for all the frames, this will return a list of
94          all frames for the simulation. See class docstring for deatils
95          or the changes made to rendering in this case
96
97          Parameters
98          ----------
99          data: list[(float, list[Any], list[Any], int, float)]
100             list of data for all frames. Each frame will have the following
101             in a tuple: (scroll, lidars, draw_list_data, step, reward), where
102             scroll is the x scroll position of the frame
103             lidars is a list of lidar-like objects
104             draw_list_data is a list of objectdrawing data
105             step is the simulation step (not the frame step) for this frame
106             reward is the current reward
107         result tuple[int, bool] or None:
108             The end result of the simulation run. This is a tuple
109             (end_step, pass_fail), where
110             end_step is the simulation step where this run ends
111             pass_fail is bool for whether the run passed
112
113         Return
114         ------
115         list[frames]
116             An ordered list of all frames for the renderering, possibly
117             including padded frames at the end. (see class docstring)
118         """
119         frames = []
120         # generate each image frame
121         for scroll, lidars, draw_list_data, step, reward in data:
122             frame = self.render(
123                 scroll=scroll,
124                 render_data=[lidars, draw_list_data, step, reward],
125                 result=result,
126             )
127             frames.append(frame)
128         # pad the end with duplicate frames as needed
129         for _ in range(self._n_padding_frames):
130             frames.append(frames[-1])
131         return frames
132
133     def _draw_scene(self, viewport: Viewport, kwargs: dict[str, Any]) -> None:
134         """Draw the objects in the scene
135
136         Parameters
137         ----------
138         viewport: Viewport
139             the current viewport
140         kwargs: dict[str, Any]
141             optional additional arguments. The following are supported:
142             * 'scroll': float -> replaces the viewpoint scroll with the
143               given value
144             * 'result': [last_step: int, pass_fail: bool] -> renders the
145               status of the run on the progress bar. last_step is the
146               index of the step where the run will end, pass_fail is
147               whether the run made it to the end of the course.
148             * 'render_data': [lidars: data, draw_list_data: data, step: int] ->
149               data to render in place of the current data. This is
150               used when batch rendering from cached data. lidars
151               is a list of FakeLidar objects contining the lidar
152               points. draw_list_data is a list of object data to draw.
153               See _draw_cached_objects for specifications.
154         """
155         # if there is a passed scroll value, overwrite the existing value
156         if "scroll" in kwargs:
157             self.scroll = kwargs["scroll"]
158             viewport.x_min = self.scroll
159             viewport.x_max = viewport.x_min + VIEWPORT_W / SCALE
```

```
160
161         step = self.parent.current_step
162
163         # calculate the vertical shift and apply it to vertical extents
164         y_shift = self._calc_y_shift()
165         viewport.y_min = viewport.y_min - y_shift
166         viewport.y_max = viewport.y_max - y_shift
167
168         # shift the clouds so they stay pinned to the top
169         shifted_clouds = self._shift_clouds(y_shift)
170         super()._draw_sky(viewport)
171         super()._draw_clouds(viewport, shifted_clouds)
172         super()._draw_terrain(viewport, self.parent.terrain_poly)
173
174         # render_all_data contains the data for the updated renderer
175         if "render_data" in kwargs:
176             lidars, draw_list_data, step, reward = kwargs["render_data"]
177             self._handle_lidar(viewport, lidars, kwargs)
178             self._draw_cached_objects(viewport, draw_list_data)
179         else:
180             self._handle_lidar(viewport, self.parent.lidar, kwargs)
181             super()._draw_objects(viewport)
182
183         super()._draw_flag(viewport)  # starting flag
184
185         self._draw_end_flag(viewport, y_shift)
186         self._draw_step_progress_bar(viewport, step, kwargs.get("result", []))
187         self._draw_path(viewport, y_shift)
188
189         if "render_data" in kwargs:
190             # reward was set earlier
191             total_reward = self.parent.final_reward
192             if reward is not None:
193                 self._draw_reward(
194                     viewport, current_reward=reward, total_reward=total_reward
195                 )
196         else:
197             reward = self.parent.reward_to_now
198             total_reward = self.parent.final_reward
199             if reward is not None:
200                 self._draw_reward(
201                     viewport, current_reward=reward, total_reward=total_reward
202                 )
203
204     def _handle_lidar(
205         self, viewport: Viewport, lidars: list[Any], kwargs: dict[str, Any]
206     ) -> None:
207         if kwargs.get("all_lidar", False):
208             self._draw_lidar(viewport, lidars)
209         else:
210             super()._draw_lidar(viewport, lidars)
211
212     def _draw_lidar(self, viewport: Viewport, lidar: list[Any]) -> None:
213         for lidar_ in lidar:
214             points = [
215                 self.scale_point(shift_point(point, delta_x=-viewport.x_min))
216                 for point in [lidar_.point1, lidar_.point2]
217             ]
218             self.draw_lines(points, color=(255, 0, 0), closed=False, width=1)
219
220     def _draw_end_flag(self, viewport: Viewport, y_shift: float) -> None:
221         """Draw a checkered flag at the end of the course
222
223         Parameters
224         ----------
225         viewport: Viewport
226             the current viewport
227         y_shift: float
228             the y shift factor for the current view
229         """
230         # if the flag is out of the viewport, don't draw it.
231         flag_x = TERRAIN_STEP * (self.parent.terrain_len - 1) - viewport.x_min
232         if flag_x > self.scroll + VIEWPORT_W / SCALE:
233             return
234
235         flag_y_top = TERRAIN_HEIGHT + 50 / SCALE
236
237         # flag pole
238         pole = [
239             self.scale_point((flag_x, viewport.pixel_y_min)),
240             self.scale_point((flag_x, flag_y_top)),
```

```
241             ]
242             self.draw_lines(poly=pole, color=self._colors["black"], closed=False, width=2)
243
244             #  Flag coordinates:
245             #
246             #  *                  <-  y1
247             #  |     *            <-  y2
248             #  *-----*------*     <-  y3
249             #  |     *            <-  y4
250             #  *                  <-  y5
251             #  ^     ^      ^
252             #  |     |      |
253             #  x1    x2     x3
254             #
255             x_vals = [flag_x + i / SCALE for i in [0, 10, 25]]
256             y_vals = [flag_y_top - i / SCALE for i in (0, 2, 5, 8, 10)]
257
258             # flag and checkers for flag
259             f_flag = [
260                 self.scale_point((x_vals[0], y_vals[0])),
261                 self.scale_point((x_vals[0], y_vals[4])),
262                 self.scale_point((x_vals[2], y_vals[2])),
263                 self.scale_point((x_vals[0], y_vals[0])),
264             ]
265             f_check1 = [
266                 self.scale_point((x_vals[0], y_vals[0])),
267                 self.scale_point((x_vals[0], y_vals[2])),
268                 self.scale_point((x_vals[1], y_vals[2])),
269                 self.scale_point((x_vals[1], y_vals[1])),
270             ]
271             f_check2 = [
272                 self.scale_point((x_vals[0], y_vals[2])),
273                 self.scale_point((x_vals[0], y_vals[4])),
274                 self.scale_point((x_vals[1], y_vals[3])),
275                 self.scale_point((x_vals[1], y_vals[2])),
276             ]
277             f_check3 = [
278                 self.scale_point((x_vals[1], y_vals[1])),
279                 self.scale_point((x_vals[1], y_vals[2])),
280                 self.scale_point((x_vals[2], y_vals[2])),
281             ]
282             f_check4 = [
283                 self.scale_point((x_vals[1], y_vals[3])),
284                 self.scale_point((x_vals[1], y_vals[2])),
285                 self.scale_point((x_vals[2], y_vals[2])),
286             ]
287
288             self.draw_polygon(f_check1, color=self._colors["black"])
289             self.draw_polygon(f_check2, color=self._colors["white"])
290             self.draw_polygon(f_check3, color=self._colors["white"])
291             self.draw_polygon(f_check4, color=self._colors["black"])
292             self.draw_lines(poly=f_flag, color=self._colors["gray"], closed=True, width=2)
293
294         def _shift_clouds(self, y_shift: float) -> list[tuple[PolygonType, float, float]]:
295             """Return the clouds shifted down by y_shift
296
297             Parameters
298             ----------
299             y_shift: float
300                 the y shift factor for the current view
301
302             Returns
303             -------
304             list[tuple[PolygonType, float, float]]
305                 A list of the clouds, with each cloud given as a list
306                 of points and the x_min and x_max value of the cloud.
307             """
308             clouds_new = [
309                 (
310                     [shift_point(point, delta_x=0, delta_y=-y_shift) for point in poly],
311                     x1,
312                     x2,
313                 )
314                 for poly, x1, x2 in self.cloud_poly
315             ]
316             return clouds_new
317
318         def _calc_y_shift(self) -> float:
319             """Return the shift factor for the y direction
320
321             Returns
```

```
        -------
        float
            the y shift factor for the current view
        """
        # find the terrain segment that the walker is on. The height is
        # the average of that segment's height.
        # calculate shift by forcing the shifted height to be in a
        # target range
        x_target = self.scroll + TERRAIN_STEP * TERRAIN_STARTPAD / 2
        y_target = 0
        for poly, _ in self.parent.terrain_poly:
            if poly[0][0] <= x_target <= poly[1][0]:
                y_target = 0.5 * (poly[0][1] + poly[1][1])
                break

        if y_target < 2.0:
            return 2.0 - y_target
        if y_target > 8.0:
            return 8.0 - y_target
        return 0.0  # within view box, no shift needed

    def _draw_cached_objects(
        self, viewport: Viewport, obj_data: list[tuple[Any]]
    ) -> None:
        """Draw the objects of the scene

        This differs from the orginal _draw_objects method by using
        cached data instead of current data.

        Parameters
        ----------
        viewport: Viewport
            the current viewport
        obj_data: list[object data]
            The cached data is an array of objects, where each object
            is a list of fixtures. The format of the fixture depends
            on the type of fixture.
            For a circle: "circle", radius, transform, color1, color2
            For others: "other", path, color1, color2
        """
        for obj in obj_data:
            for fixture in obj:
                if fixture[0] == "circle":
                    raise NotImplementedError("circle rendering not supported")
                _, path, color1, color2 = fixture
                path = [
                    self.scale_point(shift_point(point, delta_x=-viewport.x_min))
                    for point in path
                ]
                if len(path) > 3:
                    self.draw_polygon(path[:-1], color=color1)
                self.draw_lines(path, color=color2, closed=False, width=2)

    def _draw_step_progress_bar(
        self, viewport: Viewport, step: int, result: tuple[int, bool]
    ) -> None:
        """Draw a bar to indicate frame progress

        Parameters
        ----------
        vieport: Viewport
            the current viewport
        step: int
            the current frame step index, in range [0, parent.max_episode_steps]
        result: tuple[int, bool]
            the result of the simulation run. The tuple includes the
            step when the run ends and a bool indicating whether is
            passed or failed.
        """
        percent_done = step / self.parent.max_episode_steps

        # so bar doesn't take full width
        width, height = viewport.get_pixel_size()
        x_min = 0.03 * width
        x_max = 0.97 * width
        y_min = 0.92 * height
        y_max = 0.97 * height

        progress = 0.94 * width * percent_done + x_min
        progress_color = (0, 0, 204)
        bg_color = (204, 204, 204)
```

```
403
404         progress_box = [(x_min, y_min), (x_max, y_min), (x_max, y_max), (x_min, y_max)]
405         self.draw_polygon(poly=progress_box, color=bg_color)
406
407         # fill progress bar line
408         progress_bar = [
409             (x_min, y_min),
410             (progress, y_min),
411             (progress, y_max),
412             (x_min, y_max),
413         ]
414         self.draw_polygon(poly=progress_bar, color=progress_color)
415
416         # Draw the final result in the progress bar
417         # if the run succeeds, color green from the sucess point to the end
418         # if the run fails, color red from the sucess point to the end
419         if result:
420             color_pass_fail = {True: (179, 255, 179), False: (255, 179, 179)}
421             end_step, pass_fail = result
422             end_percent = end_step / self.parent.max_episode_steps
423             end_point = 0.94 * width * end_percent + x_min
424             result_bar = [
425                 (end_point, y_min),
426                 (x_max, y_min),
427                 (x_max, y_max),
428                 (end_point, y_max),
429             ]
430             self.draw_polygon(poly=result_bar, color=color_pass_fail[pass_fail])
431
432         # draw border around progress bar
433         self.draw_lines(
434             poly=progress_box, color=self._colors["black"], closed=True, width=2
435         )
436
437     def _draw_path(self, viewport: Viewport, y_shift: float) -> None:
438         """Draw the total path with a viewport indicator
439
440         Parameters
441         ----------
442         viewport: Viewport
443             the current viewport
444         y_shift: float
445             the y shift factor for the current view
446         """
447         x_min, x_max = viewport.x_min, viewport.x_max
448         _, p_height = viewport.get_pixel_size()
449         delta_x = x_max - x_min
450         y_shift -= 1
451
452         terrain_len = len(self.parent.terrain) + 1
453         percent_length = viewport.x_min / (TERRAIN_STEP * terrain_len)
454         mini_height = 0.2 * p_height
455         mini_width = mini_height * VIEWPORT_W / VIEWPORT_H
456         mini_x_min = percent_length * VIEWPORT_W
457         mini_y_min = p_height * 0.00
458
459         x_scale = delta_x / (TERRAIN_STEP * terrain_len)
460         new_poly = [poly[0] for poly, _ in self.parent.terrain_poly]
461         new_poly.append(self.parent.terrain_poly[-1][0][1])  # last point
462         heights = [p[1] for p in new_poly]
463         height_min, height_max = min(heights), max(heights)
464         y_zero = new_poly[0][1]
465         y_base = mini_height / 2 - y_zero
466         diff = max(abs(height_max - y_zero), abs(height_min - y_zero))
467         if diff <= mini_height / 2:  # the plot fits within the range
468             y_scale = 1
469         else:
470             y_scale = 2 / diff
471
472         # draw the terrain line
473         for points in zip(new_poly[:-1], new_poly[1:]):
474             point1, point2 = [
475                 (xx * x_scale, yy * y_scale + y_base) for xx, yy in points
476             ]
477             self.draw_lines(
478                 [point1, point2], color=self._colors["black"], closed=False, width=1
479             )
480         # draw viewpoint indicator
481         mini_viewport = Viewport(
482             mini_x_min,
483             mini_x_min + mini_width,
```

```
            mini_y_min,
            mini_y_min + mini_height,
            SCALE,
        )
        color_dark_gray = (77, 77, 77)
        color_dark_red = (204, 77, 77)
        box = [
            (mini_viewport.x_min, mini_viewport.y_min),
            (mini_viewport.x_max, mini_viewport.y_min),
            (mini_viewport.x_max, mini_viewport.y_max),
            (mini_viewport.x_min, mini_viewport.y_max),
        ]
        self.draw_lines(poly=box, color=color_dark_gray, closed=True, width=1)

        # draw bar on mini viewer to indicate where walker is
        point1 = (mini_x_min + 0.25 * mini_width, mini_y_min)
        point2 = (mini_x_min + 0.25 * mini_width, mini_y_min + mini_height)
        self.draw_lines(
            poly=[point1, point2], color=color_dark_red, closed=False, width=1
        )

    def _draw_reward(
        self,
        viewport: Viewport,
        current_reward: float,
        total_reward: Optional[float] = None,
    ) -> None:
        y_font = 0.875 * viewport.get_pixel_size()[1]
        x_font_start1 = 0.03 * viewport.get_pixel_size()[0]
        x_font_end2 = 0.97 * viewport.get_pixel_size()[0]  # end point of total reward

        # current reward
        text = self.font.render(f"Reward: {current_reward:.2f}", True, (0, 0, 0))
        text = pygame.transform.rotate(text, 180)
        text = pygame.transform.flip(text, True, False)
        self.surface.blit(text, [x_font_start1, y_font])

        if total_reward is not None:
            text = self.font.render(
                f"Final reward: {total_reward:.2f}", True, (0, 0, 0)
            )
            text = pygame.transform.rotate(text, 180)
            text = pygame.transform.flip(text, True, False)
            text_width = text.get_width()  # use to find where to put text
            self.surface.blit(text, [x_font_end2 - text_width, y_font])
```

8.5 marco_polo/envs/BipedalWalker/cppn.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.

import datetime
import json
import logging
import os
import pickle
import time
from argparse import Namespace
from collections import defaultdict, deque
from collections.abc import Callable
from typing import Any, cast, Optional, Union

# https://matplotlib.org/2.0.2/faq/usage_faq.html#non-interactive-example
# https://github.com/matplotlib/matplotlib/issues/3466/
# https://stackoverflow.com/questions/35737116/runtimeerror-invalid-display-variable
import matplotlib  # type: ignore[import] # pylint disable=wrong-import-position
import neat  # type: ignore[import]
import numpy as np
from numpy.random import PCG64DXSM, Generator

from marco_polo.tools.types import PathString

from marco_polo.envs._base.env_params import (
    EnvParams,
)  # pylint: disable=wrong-import-position

matplotlib.use("Agg")
import matplotlib.pyplot as plt  # type: ignore[import] # pylint: disable=wrong-import-position

LOGGER = logging.getLogger(__name__)


############################################################
## Factories
############################################################
def get_env_param_class(args: Namespace) -> Callable[..., Any]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env parameter objects
    """
    return CppnEnvParams


############################################################
## Auxiliary Class
############################################################
class PrettyGenome(neat.DefaultGenome):  # type: ignore[misc] # DefaultGenome is Any
    """DefaultGenome extended with a human readable string representations

    This can be used as a direct replacement for neat.DefaultGeome when a more
    useful string representatinos is desired. (see __str__ method for details)

    The class is intended to be accessed via the CppnEnvParams class
```

```
79
80      See Also
81      --------
82      neat.DefaultGenome for details of the class
83      """
84
85      def __str__(self) -> str:
86          """Return a string representation of the genome
87
88          An example of the output is:
89          Fitness: None
90          Nodes:
91              0 DefaultNodeGene(key=0, bias=-0.16533354, response=4.416987, activation=identity,
                    aggregation=sum)
92              1 DefaultNodeGene(key=1, bias=-0.19784433, response=1.0, activation=sin, aggregation=sum)
93          Connections:
94              DefaultConnectionGene(key=(-1, 1), weight=1.03887474446, enabled=True)
95              DefaultConnectionGene(key=(1, 0), weight=-0.12996966813, enabled=True)
96          """
97          connections = [c for c in self.connections.values() if c.enabled]
98          connections.sort()
99          output = "Fitness: {0}\nNodes:".format(self.fitness)
100         for key, node in self.nodes.items():
101             output += "\n\t{0} {1!s}".format(key, node)
102         output += "\nConnections:"
103         for conn in connections:
104             output += "\n\t" + str(conn)
105         return output
106
107
108 ######################################################
109 ## Main Class
110 ######################################################
111 class CppnEnvParams(EnvParams):
112     """Network that can evolve via a genetic algorithm
113
114     CPPN = Compositional pattern-producing network (see _[1] for details)
115
116     This class stores details of the genome and can spawn off mutated copies.
117     It is essentially a wrapper around PrettyGenome that adds input, output,
118     mutation, and plotting. The evolution process is done using NEAT _[2]
119
120     Class Attributes
121     ----------------
122     x: np.array[float]
123         The x values of the environment
124
125     Attributes
126     ----------
127     parent: PrettyGenome | None
128         The parent of the genome in this instance
129     seed: int | Arry[int]
130         random number seed
131     np_random: numpy.random.Generator
132         Rnadom number generator state
133     cppn_config_path: str
134         path to cppn config file
135     genome_path: str
136         path to genome file
137     cppn_config: neat.Config
138         Configuration for the CPPN system ????
139     cppn_genome: PrettyGenome
140         The actual genome
141     altitude_fn: Callable
142         the function to calculate the altitude
143
144     References
145     ----------
146     .. [1] Kenneth O. Stanley. 2007. Compositional pattern producing networks:
147            A novel abstraction of development. Genetic Programming and
148            Evolvable Machines 8, 2 (Jun 2007), 131  162  .
149            https://doi.org/10.1007/s10710-007-9028-8
150
151     .. [2] K. O. Stanley and R. Miikkulainen, "Efficient evolution of neural
152            network topologies," Proceedings of the 2002 Congress on
153            Evolutionary Computation. CEC'02 (Cat. No.02TH8600), Honolulu, HI,
154            USA, 2002, pp. 1757-1762 vol.2, doi: 10.1109/CEC.2002.1004508.
155     """
156
157     # the shared X values for the environments
158     x = np.array([(i - 200 / 2.0) / (200 / 2.0) for i in range(200)])
```

```
159     # note, this is the same as: x = np.linspace(-1, 0.99, 200)
160     # They will differ by ~10^-16 due to machine precision, so a direct output
161     # comparison will have minor differences.
162
163     def __init__(
164         self,
165         args: Namespace,
166         seed: Optional[int] = None,
167         cppn_config_path: str = "config-cppn",
168         genome_path: Optional[PathString] = None,
169     ) -> None:
170         """Create and intialize the genome wrapper
171
172         The object's evolution and other parameters are controlled by the
173         config file. Additionally, a path to an existing genome can be
174         specified. If so, it will be read in. If not, default parameters
175         will be used in constructing the object.
176
177         Parameters
178         ----------
179         args: Namespace
180             The program parameters given on startup
181         seed: int or None, default = None
182             seed to use for the random number generator
183         cppn_config_path: str, default = 'config-cppn'
184             path to use for the cppn config. This is relative to the location
185             of this file.
186         genome_path: str, optional
187             path to existing genome to use
188         """
189         # https://stackoverflow.com/questions/55231989/optional-union-in-type-hint
190
191         super().__init__(args)
192         self.parent: Union[str, None] = None
193         if seed:
194             self.seed = seed
195         else:
196             self.seed = args.master_seed
197         # These parameters are not currently stateful
198         #  objects are pulled from the fiber pull, but not reset in the pool
199         #  so it doesn't account for stateful computation.
200         # Updating calls in es.py to push the updated objects into the pool
201         # handles this issue.
202         # This is a mistake in the original POET code.
203         self.np_random = Generator(PCG64DXSM(seed=self.seed))
204         self.cppn_config_path = os.path.join(
205             os.path.dirname(__file__), cppn_config_path
206         )
207         self.genome_path = genome_path
208         self.cppn_config = neat.Config(
209             neat.DefaultGenome,
210             neat.DefaultReproduction,
211             neat.DefaultSpeciesSet,
212             neat.DefaultStagnation,
213             self.cppn_config_path,
214         )
215         self.altitude_fn = lambda x: x
216         if genome_path is not None:
217             with open(genome_path, mode="rb") as file:
218                 self.cppn_genome = pickle.load(file)
219         else:
220             start_cppn_genome = PrettyGenome("0")
221             start_cppn_genome.configure_new(self.cppn_config.genome_config)
222             start_cppn_genome.nodes[0].activation = "identity"
223             self.cppn_genome = start_cppn_genome
224         self.reset_altitude_fn()
225
226     def get_name(self) -> str:
227         """Return the name of the genome"""
228         # LOGGER.info(str(hash(str(self.cppn_genome))))
229         # LOGGER.info(str(self.cppn_genome))
230         return str(hash(str(self.cppn_genome)))
231
232     def reset_altitude_fn(self) -> None:
233         """Reset the altitude function of the genome from the network
234
235         This would be used when a new genome is created or mutated to
236         replace the default function.
237
238         Side-effects
239         ------------
```

```
240         Replaces the current altitude function
241         """
242         net = neat.nn.FeedForwardNetwork.create(self.cppn_genome, self.cppn_config)
243         self.altitude_fn = net.activate
244 
245     def get_mutated_params(self) -> "CppnEnvParams":
246         """Return a mutated copy of the genome
247 
248         The parameters for the mutation are given in the config file
249         used to create the genome.
250 
251         Side-effects
252         ------------
253         None
254 
255         Returns
256         -------
257         CppnEnvParams
258             A new CppnEnvParam object that is a mutated copy of the original
259         """
260         while True:
261             mutated = copy_genome(self.cppn_genome)
262             mutated.nodes[0].response = 1.0
263             # key is a unique identifier for this genome
264             mutated.key = datetime.datetime.utcnow().isoformat()
265             # this controls the actual mutation of the structure, connections,
266             # and genes
267             mutated.mutate(self.cppn_config.genome_config)
268             distance = self.cppn_genome.distance(
269                 mutated, self.cppn_config.genome_config
270             )
271             if not (is_genome_valid(mutated) and (distance > 0)):
272                 continue
273             net = neat.nn.FeedForwardNetwork.create(mutated, self.cppn_config)
274             yvals = np.array([net.activate((xi,)) for xi in self.x])
275             yvals -= yvals[0]  # normalize to start at altitude 0
276             # TODO: yvals[0] is always zero, so why np.abs for the threshold_?
277             threshold_ = np.abs(np.max(yvals))
278             if threshold_ <= 0:  # TODO: essentially theshold_ != 0 ??
279                 continue
280             # the significance of these numbers is not clear
281             if threshold_ < 0.25:
282                 mutated.nodes[0].response = (
283                     self.np_random.random() / 2 + 0.25
284                 ) / threshold_
285             if threshold_ > 16:
286                 mutated.nodes[0].response = (
287                     self.np_random.random() * 4 + 12
288                 ) / threshold_
289 
290             # now that the genome is created, make the wrapper
291             res = CppnEnvParams(
292                 self.args, seed=self.np_random.integers(low=2**32 - 1, dtype=int)
293             )
294             res.cppn_genome = mutated
295             res.reset_altitude_fn()
296             res.parent = self.get_name()
297 
298             return res
299 
300     def xy(self) -> dict[str, list[float]]:
301         """Return the x,y values of the environment
302 
303         Returns
304         -------
305         dict[str, list[float]]
306             a dict of lists of the x,y values in the form:
307             {'x': [x_values], 'y': [y_values]}
308         """
309         net = neat.nn.FeedForwardNetwork.create(self.cppn_genome, self.cppn_config)
310         yvals = np.array([net.activate((xi,))[0] for xi in self.x])
311         return {"x": self.x.tolist(), "y": yvals.tolist()}
312 
313     def save_xy(self, folder: PathString = "/tmp") -> None:
314         """Create and save a plot of x,y values for the environment
315 
316         Parameters
317         ----------
318         folder: PathString, default = "/tmp"
319             folder where the plot will be saved.
320 
```

```
        Side-effects
        ------------
        Writes the plot to disk, overwriting any plot in the folder that was
        previously created by this function.
        """
        # with open(folder + '/' + self.cppn_genome.key + '_xy.json', 'w') as f:
        xy_vals = self.xy()
        x_vals, y_vals = xy_vals["x"], xy_vals["y"]
        with open(os.path.join(folder, "_xy.json"), mode="w", encoding="utf8") as file:
            file.write(json.dumps({"x": x_vals, "y": y_vals}))

        # Plot environment function
        plt.plot(x_vals, y_vals)
        plt.savefig(os.path.join(folder, "terrain.png"))
        plt.close()

    def to_json(self) -> str:
        """Return JSON representation of the object

        Returns
        -------
        str
            A JSON dump of a dict with the following keys:
            'cppn_config_path' -  path to the configuration file for the genome
            'genome_path' - path to the genome
            'parent' - the parent of this genome
            'np_random' - the random state
        """
        return json.dumps(
            {
                "cppn_config_path": self.cppn_config_path,
                "genome_path": self.genome_path,
                "parent": self.parent,
                "np_random": self.np_random.bit_generator.state,
            }
        )

    def save_genome(self, file_path: Optional[PathString] = None) -> None:
        """Save the genome (not the whole class) to a file

        Parameters
        ----------
        file_path: PathString, optional
            path to write file to. If omitted, a deafult path is selected
        """
        if file_path is None:
            file_path = "/tmp/genome_{}_saved.pickle".format(time.time())

        with open(file_path, mode="wb") as file:
            pickle.dump(self.cppn_genome, file)

    def _save(self, folder: PathString) -> None:
        """Saves the object to the given folder

        The components saved are:
        * a pickle file of the genome
          saved to _genome.pkl
        * the raw xy data
          saved to _xy.json
        * an XY plot of the data
          saved to terrain.png
        * the configuration (see self.to_json)
          saved to _config.json
        * a human readble version of the genome (see PrettyGenome.__str__)
          saved to "genome.txt"

        Parameters
        ----------
        folder: PathString
            path to store data in
        """
        os.makedirs(folder, exist_ok=True)
        self.save_genome(os.path.join(folder, "_genome.pkl"))
        self.save_xy(folder)

        with open(
            os.path.join(folder, "_config.json"), mode="w", encoding="utf8"
        ) as file:
            file.write(self.to_json())

        with open(
```

```
            os.path.join(folder, "genome.txt"), mode="w", encoding="utf8"
        ) as file:
            file.write(str(self.cppn_genome))

    def checkpoint(self, folder: PathString) -> None:
        """Save the current state of the object

        Parameters
        ----------
        folder: PathString
            path to store data in
        """
        self._save(os.path.join(folder, "cppn"))

    def reload(self, folder: PathString) -> None:
        """Load an object from disk

        Parameters
        ----------
        folder: PathString
            path to load data from

        Side-Effects
        ------------
        Overwrites this instance with the data from disk
        """
        dir_path = os.path.join(folder, "cppn")

        with open(os.path.join(dir_path, "_genome.pkl"), mode="rb") as file:
            self.cppn_genome = pickle.load(file)

        # create prng, set state within open()
        self.np_random = Generator(PCG64DXSM())

        with open(
            os.path.join(dir_path, "_config.json"), mode="r", encoding="utf8"
        ) as file:
            json_dict = json.load(file)

        self.cppn_config_path = json_dict["cppn_config_path"]
        self.genome_path = json_dict["genome_path"]
        self.parent = json_dict["parent"]
        self.np_random.bit_generator.state = json_dict["np_random"]
        self.reset_altitude_fn()


def copy_genome(genome: PrettyGenome) -> PrettyGenome:
    """Return a duplicate of the genome

    Parameters
    ----------
    genome: PrettyGenome
        the genome to copy

    Side-Effects
    ------------
    The implementation will write the state to disk

    Returns
    -------
    PrettyGenome
        a copy of the genome
    """
    # write object to disk, then read back into a new object
    file_path = "/tmp/genome_{}.pickle".format(time.time())
    with open(file_path, mode="wb") as file:
        pickle.dump(genome, file)
    with open(file_path, mode="rb") as file:
        return cast(PrettyGenome, pickle.load(file))


def is_genome_valid(genome: PrettyGenome) -> bool:
    """Return true if the genome is valid

    'valid' in this sense appears to mean that the connections graph
    makes it from the input to the output. The current config-cppn has a
    single input and output (see, num_inputs=1 and num_outputs=1). It's
    unclear if/how this would need to change if the number of inputs/outputs
    was changed in the config file.

    Parameters
```

```
    ----------
    genome: PrettyGenome
        the genome to test

    Returns
    -------
    bool
        whether the genome is valid
    """
    # graph tracks forward connections. If a connects to b, c, and d:
    # graph[a] = [b, c, d]
    graph = defaultdict(list)
    for key in genome.connections.keys():
        graph[key[0]].append(key[1])
    # this is walking along the graph, starting at the input (key -1 by
    # convention for a single input), trying to find the output (key 0 by
    # convention for a single output)
    # This code may have potential for an infinite loop with a circular
    # network i.e. b in graph[a] and a in graph[b]
    # I assume the mutation code disallows that scenario, but I haven't
    # confirmed that.
    queue = deque([-1])
    while len(queue) > 0:
        cur = queue.popleft()
        if cur == 0:  # network leads to the output, this is valid
            return True
        if cur not in graph:  # dead end in network path, ignore
            continue
        for node in graph[cur]:  # add nodes connected to this one
            # could probably add a check for zero here too, by savings would
            # be minimal for the added complexity
            queue.append(node)
    return False
```

8.6 marco_polo/envs/BipedalWalker/env.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


"""Bipedal walker environment

This is simple 4-joints walker robot environment.

There are two versions:

- Normal, with slightly uneven terrain.

- Hardcore with ladders, stumps, pitfalls.

Reward is given for moving forward, total 300+ points up to the far end. If the robot
falls, it gets -100. Applying motor torque costs a small amount of points, more optimal
agent will get better score.

Heuristic is provided for testing, it's also useful to get demonstrations to
learn from. To run heuristic:

python gym/envs/box2d/bipedal_walker.py

State consists of hull angle speed, angular velocity, horizontal speed, vertical speed,
position of joints and joints angular speed, legs contact with ground, and 10 lidar
rangefinder measurements to help to deal with the hardcore version. There's no
coordinates in the state vector. Lidar is less useful in normal version, but it works.

To solve the game you need to get 300 points in 1600 time steps.

To solve hardcore version you need 300 points in 2000 time steps.

Created by Oleg Klimov. Licensed on the same terms as the rest of OpenAI Gym.
"""


import argparse  # for typing
import copy
import math
from collections import namedtuple
from collections.abc import Callable
from typing import Any, cast, Optional, Type, Union

import Box2D  # type: ignore[import]
import gymnasium as gym
import numpy as np
from Box2D.b2 import (  # type: ignore[import]
    contactListener,
    edgeShape,
    fixtureDef,
    polygonShape,
    revoluteJointDef,
)
from gymnasium import spaces
from gymnasium.utils import seeding
from pettingzoo.utils.env import ParallelEnv  # type: ignore[import]

from marco_polo.envs.BipedalWalker.bpw_constants import (
    SCALE,
    TERRAIN_GRASS,
    TERRAIN_HEIGHT,
    TERRAIN_STARTPAD,
    TERRAIN_STEP,
```

```
    VIEWPORT_H,
    VIEWPORT_W,
)
from marco_polo.envs.BipedalWalker.bpw_renderer import (
    BipedalWalkerRenderer,
    ColorType,
    PointType,
    PolygonType,
)
from marco_polo.envs.BipedalWalker.bpw_renderer_updated import (
    BipedalWalkerRendererEnhanced,
)
from marco_polo.envs.BipedalWalker.cppn import CppnEnvParams
from marco_polo.envs.BipedalWalker.terrain_generation import generate_terrain_coords
from marco_polo.tools.compiled import fast_clip, fast_sum
from marco_polo.tools.types import RenderFrame, Role


FPS = 50

MOTORS_TORQUE = 80
SPEED_HIP = 4
SPEED_KNEE = 6
LIDAR_RANGE = 160 / SCALE

INITIAL_RANDOM = 5

HULL_POLY = [(-30, +9), (+6, +9), (+34, +1), (+34, -8), (-30, -8)]
LEG_DOWN = -8 / SCALE
LEG_W, LEG_H = 8 / SCALE, 34 / SCALE

FRICTION = 2.5

HULL_FD = fixtureDef(
    shape=polygonShape(vertices=[(x / SCALE, y / SCALE) for x, y in HULL_POLY]),
    density=5.0,
    friction=0.1,
    categoryBits=0x0020,
    maskBits=0x001,  # collide only with ground
    restitution=0.0,
)  # 0.99 bouncy

LEG_FD = fixtureDef(
    shape=polygonShape(box=(LEG_W / 2, LEG_H / 2)),
    density=1.0,
    restitution=0.0,
    categoryBits=0x0020,
    maskBits=0x001,
)

LOWER_FD = fixtureDef(
    shape=polygonShape(box=(0.8 * LEG_W / 2, LEG_H / 2)),
    density=1.0,
    restitution=0.0,
    categoryBits=0x0020,
    maskBits=0x001,
)

# statics for the step function
HKHK_Vec = np.array(object=[SPEED_HIP, SPEED_KNEE, SPEED_HIP, SPEED_KNEE])
X_SCALE = 0.3 * (VIEWPORT_W / SCALE) / FPS
Y_SCALE = 0.3 * (VIEWPORT_H / SCALE) / FPS
SCROLL_SCALE = VIEWPORT_W / SCALE / 5
SHAPING_SCALE = 130 / SCALE
LIDAR_SCALE = 1.5 / 10.0

LIDAR_FULL = [
    (math.sin(i * LIDAR_SCALE) * LIDAR_RANGE, math.cos(i * LIDAR_SCALE) * LIDAR_RANGE)
    for i in range(10)
]

FakeLidar = namedtuple("FakeLidar", ["point1", "point2"])


def get_env_class(args: argparse.Namespace) -> Callable[..., Any]:
    """Returns the class to use, based on input arguments

    This function selects between BipedalWalkerCustom and
    BipedalWalkerCustomEnhanced based on the requested class

    The enhanced version is used if the input config contains:
```

```
    env_params:
      class: "BipedalWalkerCustomEnhanced"

    The regular version is used if the input config contains:
    env_params:
      class: "BipedalWalkerCustom"

    if anyother value for "class" is given, an error is raised.

    If env_params.class is omitted, BipedalWalkerCustom is used.

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env objects
    """
    if "env_params" in args:
        if "class" in args.env_params:
            class_str = args.env_params["class"]
            if class_str == "BipedalWalkerCustomEnhanced":
                return BipedalWalkerCustomEnhanced
            if class_str == "BipedalWalkerCustom":
                return BipedalWalkerCustom

            raise NotImplementedError(f"Unknown walker class: {class_str}")

    # Either no "env_params" or no "env_params.class".
    # That's fine, just return the default class
    return BipedalWalkerCustom


class ContactDetector(contactListener):  # type: ignore[misc] # contactListener is Any
    """Contact detector for Box2D physics."""

    def __init__(self, env: ParallelEnv) -> None:
        contactListener.__init__(self)
        self.env = env

    def BeginContact(self, contact: Any) -> None:  # pylint: disable=invalid-name
        """Handle contact starting."""
        if (
            self.env.hull == contact.fixtureA.body  # pylint: disable=consider-using-in
            or self.env.hull == contact.fixtureB.body
        ):
            self.env.game_over = True
        for leg in [self.env.legs[1], self.env.legs[3]]:
            if leg in [contact.fixtureA.body, contact.fixtureB.body]:
                leg.ground_contact = 1.0

    def EndContact(self, contact: Any) -> None:  # pylint: disable=invalid-name
        """Handle contact ending."""
        for leg in [self.env.legs[1], self.env.legs[3]]:
            if leg in [contact.fixtureA.body, contact.fixtureB.body]:
                leg.ground_contact = 1.0


class LidarCallback(Box2D.b2.rayCastCallback):  # type: ignore  # pylint: disable=too-few-public-methods
    """Callback class for lidar raycasting"""

    def ReportFixture(  # type: ignore
        self, fixture, point, _, fraction
    ):  # pylint: disable=invalid-name
        """Callback for Box2D"""
        if (fixture.filterData.categoryBits & 1) == 0:
            return 1
        self.point2 = point  # pylint: disable=attribute-defined-outside-init
        self.fraction = fraction  # pylint: disable=attribute-defined-outside-init
        return 0


class BipedalWalkerCustom(ParallelEnv):  # type: ignore[misc] # ParallelEnv is Any
    """BipedalWalkerCustom augmented gym environment

    Attributes
    ----------
    env_params: cppn.CppnEnvParams
```

```
        Parameters for this env, including mutation info
    self.renderer: BipedalWalkerRenderer
        Object to render visuals
    self.world: Box2D.b2World
        2D Physics world object
    self.terrain: list
        List of Box2D.b2World.StaticBody objects, ie objects with no dynamics
    self.hull: Box2D.b2World.CreateDynamicBody
        DynamicBody object for the walker, experiences physics like gravity
    self.prev_shaping: float
        If not None, subtract previous step reward off of current step reward each
        step resulting in reward that shows (pos or neg) improvement each step
    self.fd_polygon: Box2D.b2.fixtureDef
        Abstract fixture, I think this corresponds to the terrain the walker uses
        in the case where we're not using the cppn to create the terrain.
    self.fd_edge: Box2D.b2.fixtureDef
        Abstract fixture, I think this corresponds to the terrain the walker uses
        in the case where we are using the cppn to create the terrain.
    self.action_space = spaces.Box
        Gym action space
    """

    # multi-agent compatibility
    role_name = cast(Role, "default")
    possible_agents = [role_name]

    metadata = {"render.modes": ["human", "rgb_array"], "video.frames_per_second": FPS}

    def __repr__(self) -> str:
        return f"{self.__dict__}\nenv\n{self.__dict__['np_random'].get_state()}"

    def __init__(self) -> None:
        self.agents: list[Role] = []
        self.env_params: Optional[CppnEnvParams] = None
        self.seed()

        self.world = Box2D.b2World()
        self.terrain: list[Box2D.b2Body] = []
        self.terrain_poly: list[tuple[PolygonType, ColorType]] = []

        # components of walker
        self.hull: Optional[Box2D.b2Body] = None
        self.joints: list[Box2D.b2Joint] = []
        self.legs: list[Box2D.b2Body] = []

        self.current_step = 0  # the current step number in a simulation run

        self.prev_shaping = 0
        self.fd_polygon = fixtureDef(
            shape=polygonShape(vertices=[(0, 0), (1, 0), (1, -1), (0, -1)]),
            friction=FRICTION,
        )

        self.fd_edge = fixtureDef(
            shape=edgeShape(vertices=[(0, 0), (1, 1)]),
            friction=FRICTION,
            categoryBits=0x0001,
        )

        self.max_episode_steps = 2000  # default (may be changed in augment())

        # internal tracking of reward, only for rendering
        self.reward_to_now = 0.0  # sum of reward up to now
        self.final_reward: Optional[float] = None  # at completion of run

        self.terrain_len = 0  # will be updated in reset
        self.end_point = 0.0  # will be updated in reset
        self.finish = False

        self.scroll = 0.0

        self.render_mode = "rgb_array"
        self.renderer = self._RENDER_CLASS(self, mode=self.render_mode)
        self.reset()

        self.obs_buffer = np.zeros(24)  # to avoid creating new arrays constantly

    def augment(self, params: CppnEnvParams) -> None:
        """Add cppn environment parameters.

        Parameters
```

```
322             ----------
323             params: CppnEnvParams
324                 the cppn data to add
325             """
326             self.env_params = params
327             self.max_episode_steps = params.get("max_episode_steps", self.max_episode_steps)
328
329         def seed(self, seed: Optional[int] = None) -> None:
330             """Set the random number seed.
331
332             Parameters
333             ----------
334             seed: int, optional
335                 the random number seed
336             """
337             self.np_random, seed = seeding.np_random(seed)
338
339         def _destroy(self) -> None:
340             if not self.terrain:
341                 return
342             self.world.contactListener = None
343             for terrain in self.terrain:
344                 self.world.DestroyBody(terrain)
345             self.terrain = []
346             self.world.DestroyBody(self.hull)
347             self.hull = None
348             for leg in self.legs:
349                 self.world.DestroyBody(leg)
350             self.legs = []
351             self.joints = []
352             self.world = None
353
354         def generate_terrain(self) -> None:
355             """Generate terrain for environment.
356
357             To be useful, the env_params should be set first via `augment()`
358             """
359             terrain_x, terrain_y = generate_terrain_coords(self.env_params, self.np_random)
360
361             # use the x, y values to generate the terrain pieces
362             self.terrain = []
363             self.terrain_poly = []
364             for i_step in range(len(terrain_x) - 1):
365                 poly: list[PointType] = [
366                     (terrain_x[i_step], terrain_y[i_step]),
367                     (terrain_x[i_step + 1], terrain_y[i_step + 1]),
368                 ]
369                 self.fd_edge.shape.vertices = poly
370                 terrain = self.world.CreateStaticBody(fixtures=self.fd_edge)
371                 if i_step % 2 == 0:
372                     color = (77, 255, 77)
373                 else:
374                     color = (77, 204, 77)
375                 terrain.color1 = color
376                 terrain.color2 = color
377                 self.terrain.append(terrain)
378
379                 # Make the polygons for rendering. These are scaled to pixel sizes
380                 color = (102, 153, 77)
381                 render_polygon = [(point[0] * SCALE, point[1] * SCALE) for point in poly]
382                 # these points are the bottom of the polygon
383                 render_polygon += [(render_polygon[1][0], 0), (render_polygon[0][0], 0)]
384                 self.terrain_poly.append((render_polygon, color))
385             # ???: it's not clear why this is reversed
386             self.terrain.reverse()
387
388         def reset(
389             self, seed: Optional[int] = None, options: Optional[dict[str, Any]] = None
390         ) -> tuple[dict[Role, Any], dict[Role, dict[str, Any]]]:
391             """Reset the env for a new run."""
392             if seed is not None:
393                 self.seed(seed)
394             self._destroy()
395             self.world = Box2D.b2World()
396             self.world.contactListener_bug_workaround = ContactDetector(self)
397             self.world.contactListener = self.world.contactListener_bug_workaround
398             self.game_over = False
399             self.prev_shaping = 0
400             self.scroll = 0.0
401             self.agents = [self.possible_agents[0]]
402
```

```
        # The step() function increments this each time, including at the end of
        # this function. Setting this to -1 means it will correctly start at zero
        # for the actual simulation run.
        self.current_step = -1

        self.generate_terrain()
        assert self.terrain is not None
        self.terrain_len = len(self.terrain) + 1
        self.end_point = (self.terrain_len - TERRAIN_GRASS) * TERRAIN_STEP

        init_x = TERRAIN_STEP * TERRAIN_STARTPAD / 2
        init_y = TERRAIN_HEIGHT + 2 * LEG_H
        self.hull = self.world.CreateDynamicBody(
            position=(init_x, init_y), fixtures=HULL_FD
        )
        self.hull.color1 = (127, 102, 230)
        self.hull.color2 = (77, 77, 127)
        self.hull.ApplyForceToCenter(
            (self.np_random.uniform(-INITIAL_RANDOM, INITIAL_RANDOM), 0), True
        )

        self.legs = []
        self.joints = []
        for i in [-1, +1]:
            leg = self.world.CreateDynamicBody(
                position=(init_x, init_y - LEG_H / 2 - LEG_DOWN),
                angle=(i * 0.05),
                fixtures=LEG_FD,
            )
            leg.color1 = (153 - i * 26, 77 - i * 26, 127 - i * 26)
            leg.color2 = (102 - i * 26, 51 - i * 26, 77 - i * 26)
            rjd = revoluteJointDef(
                bodyA=self.hull,
                bodyB=leg,
                localAnchorA=(0, LEG_DOWN),
                localAnchorB=(0, LEG_H / 2),
                enableMotor=True,
                enableLimit=True,
                maxMotorTorque=MOTORS_TORQUE,
                motorSpeed=i,
                lowerAngle=-0.8,
                upperAngle=1.1,
            )
            self.legs.append(leg)
            self.joints.append(self.world.CreateJoint(rjd))

            lower = self.world.CreateDynamicBody(
                position=(init_x, init_y - LEG_H * 3 / 2 - LEG_DOWN),
                angle=(i * 0.05),
                fixtures=LOWER_FD,
            )
            lower.color1 = (152 - i * 26, 77 - i * 26, 127 - i * 26)
            lower.color2 = (102 - i * 26, 51 - i * 26, 77 - i * 26)
            rjd = revoluteJointDef(
                bodyA=leg,
                bodyB=lower,
                localAnchorA=(0, -LEG_H / 2),
                localAnchorB=(0, LEG_H / 2),
                enableMotor=True,
                enableLimit=True,
                maxMotorTorque=MOTORS_TORQUE,
                motorSpeed=1,
                lowerAngle=-1.6,
                upperAngle=-0.1,
            )
            lower.ground_contact = False
            self.legs.append(lower)
            self.joints.append(self.world.CreateJoint(rjd))

        self.drawlist = self.terrain + self.legs + [self.hull]

        self.lidar = [LidarCallback() for _ in range(10)]

        self.renderer.reset()
        self.obs_buffer = np.zeros(24)

        self.reward_to_now = 0.0
        self.final_reward = None  # None because we don't know this yet
        self.finish = False

        initial_obs, _, _, _, info = self.step({self.role_name: np.zeros(4)})
```

```
484
485         return initial_obs, info
486
487     def step(
488         self, actions: dict[str, Any]
489     ) -> tuple[
490         dict[Role, Any],
491         dict[Role, float],
492         dict[Role, bool],
493         dict[Role, bool],
494         dict[Role, dict[str, Any]],
495     ]:
496         self.current_step += 1
497         truncated = {self.role_name: False}
498         terminated = {self.role_name: False}
499         action = actions[self.role_name]
500
501         # Uncomment the next line to receive a bit of stability help
502         # self.hull.ApplyForceToCenter((0, 20), True)
503         control_speed = False  # Should be easier as well
504
505         motor_torques = MOTORS_TORQUE * np.clip(
506             a=np.abs(action), a_min=0, a_max=1, dtype=float
507         )
508         reward = -0.00035 * np.sum(a=motor_torques)
509
510         if control_speed:
511             motor_speeds = HKHK_Vec * np.clip(a=action, a_min=-1, a_max=1, dtype=float)
512             for i in range(4):
513                 self.joints[i].motorSpeed = motor_speeds[i]
514
515         else:
516             motor_speeds = HKHK_Vec * np.sign(action, dtype=float)
517             for i in range(4):
518                 self.joints[i].motorSpeed = motor_speeds[i]
519                 self.joints[i].maxMotorTorque = motor_torques[i]
520
521         # self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)
522         self.world.Step(0.02, 180, 60)
523
524         assert self.hull is not None
525         pos = self.hull.position
526         vel = self.hull.linearVelocity
527
528         lidar_frac = []
529         for i in range(10):
530             self.lidar[i].fraction = 1.0
531             self.lidar[i].point1 = pos
532             self.lidar[i].point2 = (
533                 pos[0] + LIDAR_FULL[i][0],
534                 pos[1] - LIDAR_FULL[i][1],
535             )
536             self.world.RayCast(
537                 self.lidar[i], self.lidar[i].point1, self.lidar[i].point2
538             )
539             lidar_frac.append(self.lidar[i].fraction)
540
541         state = [
542             # Normal angles up to 0.5 here, but sure more is possible.
543             self.hull.angle,
544             2.0 * self.hull.angularVelocity / FPS,
545             # Normalized to get -1..1 range
546             vel.x * X_SCALE,
547             vel.y * Y_SCALE,
548             # This will give 1.1 on high up, but it's still OK
549             # There should be spikes on hitting the ground, that's normal too
550             self.joints[0].angle,
551             self.joints[0].speed / SPEED_HIP,
552             self.joints[1].angle + 1.0,
553             self.joints[1].speed / SPEED_KNEE,
554             self.legs[1].ground_contact,
555             self.joints[2].angle,
556             self.joints[2].speed / SPEED_HIP,
557             self.joints[3].angle + 1.0,
558             self.joints[3].speed / SPEED_KNEE,
559             self.legs[3].ground_contact,
560         ]
561
562         state.extend(lidar_frac)
563
564         self.scroll = pos.x - SCROLL_SCALE
```

```
565
566         # moving forward is a way to receive reward (normalized to get 300 on completion)
567         shaping = pos[0] * SHAPING_SCALE
568         # keep head straight, other than that and falling, any behavior is unpunished
569         shaping -= 5.0 * abs(state[0])
570
571         reward += shaping - self.prev_shaping
572         self.prev_shaping = shaping
573
574         self.reward_to_now += reward
575
576         if self.game_over or pos[0] < 0:
577             reward = -100
578             terminated = {self.role_name: True}
579             self.reward_to_now = reward
580             self.final_reward = self.reward_to_now
581             self.agents = []
582
583         if pos[0] > self.end_point:
584             terminated = {self.role_name: True}
585             self.final_reward = self.reward_to_now
586             self.finish = True
587             self.agents = []
588
589         if self.current_step == self.max_episode_steps:
590             truncated = {self.role_name: True}
591             self.agents = []
592
593         # reshape for multiagent compatibility
594         observations = {self.role_name: np.array(state)}
595         rewards = {self.role_name: reward}
596         info = {self.role_name: {"finish": self.finish}}
597
598         return observations, rewards, terminated, truncated, info
599
600     def _step_noviz(
601         self, actions: dict[str, Any]
602     ) -> tuple[
603         dict[Role, Any],
604         dict[Role, float],
605         dict[Role, bool],
606         dict[Role, bool],
607         dict[Role, dict[str, Any]],
608     ]:
609         """Step the environment without visualization support
610
611         This may have a slight performance gain over step() as it
612         does not calculate/store data used for visualization.
613         It can otherwise be used just like step()
614         """
615         self.current_step += 1
616         truncated = {self.role_name: False}
617         terminated = {self.role_name: False}
618         action = actions[self.role_name]
619
620         # cache these lookups
621         assert self.hull is not None
622         hull = self.hull
623         joints = self.joints
624         world = self.world
625         state = self.obs_buffer
626
627         control_speed = False  # Should be easier as well
628
629         motor_torques = MOTORS_TORQUE * fast_clip(
630             np.abs(action).astype(float), min=0, max=1
631         )
632         reward = -0.00035 * fast_sum(motor_torques)
633
634         if control_speed:
635             motor_speeds = HKHK_Vec * fast_clip(action.astype(float), min=-1, max=1)
636             for i in range(4):
637                 joints[i].motorSpeed = motor_speeds[i]
638         else:
639             motor_speeds = HKHK_Vec * np.sign(action, dtype=float)
640             for i in range(4):
641                 joints[i].motorSpeed = motor_speeds[i]
642                 joints[i].maxMotorTorque = motor_torques[i]
643
644         # self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)
645         world.Step(0.02, 180, 60)
```

```
646
647         pos = hull.position
648         vel = hull.linearVelocity
649
650         # Normal angles up to 0.5 here, but sure more is possible.
651         state[0] = hull.angle
652         state[1] = 2.0 * hull.angularVelocity / FPS
653         # Normalized to get -1..1 range
654         state[2] = vel.x * X_SCALE
655         state[3] = vel.y * Y_SCALE
656         # This will give 1.1 on high up, but it's still OK
657         # (and there should be spikes on hiting the ground, that's normal too)
658         state[4] = joints[0].angle
659         state[5] = joints[0].speed / SPEED_HIP
660         state[6] = joints[1].angle + 1.0
661         state[7] = joints[1].speed / SPEED_KNEE
662         state[8] = self.legs[1].ground_contact
663         state[9] = joints[2].angle
664         state[10] = joints[2].speed / SPEED_HIP
665         state[11] = joints[3].angle + 1.0
666         state[12] = joints[3].speed / SPEED_KNEE
667         state[13] = self.legs[3].ground_contact
668
669         lidar = self.lidar[0]
670         for i in range(10):
671             lidar.fraction = 1.0
672             point2 = (pos[0] + LIDAR_FULL[i][0], pos[1] - LIDAR_FULL[i][1])
673             world.RayCast(lidar, pos, point2)
674             state[14 + i] = lidar.fraction
675
676         # moving forward is a way to receive reward (normalized to get 300 on completion)
677         shaping = pos[0] * SHAPING_SCALE
678         # keep head straight, other than that and falling, any behavior is unpunished
679         shaping -= 5.0 * abs(state[0])
680
681         reward += shaping - self.prev_shaping
682         self.prev_shaping = shaping
683
684         self.finish = False
685         if self.game_over or pos[0] < 0:
686             reward = -100
687             terminated = {self.role_name: True}
688         if pos[0] > self.end_point:
689             terminated = {self.role_name: True}
690             self.finish = True
691
692         if self.current_step == self.max_episode_steps:
693             truncated = {self.role_name: True}
694             self.agents = []
695
696         # reshape for multiagent compatibility
697         observations = {self.role_name: state}
698         rewards = {self.role_name: reward}
699         info = {self.role_name: {"finish": self.finish}}
700
701         return observations, rewards, terminated, truncated, info
702
703     def observation_space(self, agent: Role) -> gym.spaces.Box:
704         """Return observation space for the agent
705
706         Parameters
707         ----------
708         agent: Role
709             The agent to get the observation space for
710
711         Returns
712         -------
713         gym.spaces.Space
714             The observation space for the agent
715         """
716         return self.observation_spaces[agent]
717
718     def action_space(self, agent: Role) -> gym.spaces.Box:
719         """Return action space for the agent
720
721         Parameters
722         ----------
723         agent: Role
724             The agent to get the action space for
725
726         Returns
```

```
        -------
        gym.spaces.Space
            The action space for the agent
        """
        return self.action_spaces[agent]

    _RENDER_CLASS: Type[BipedalWalkerRenderer] = BipedalWalkerRenderer

    def render(self, *args: Any, **kwargs: Any) -> Union[None, RenderFrame]:
        """Render a frame of the environment run."""
        return self.renderer.render(*args, **kwargs)

    def render_all(self, *args: Any, **kwargs: Any) -> list[RenderFrame]:
        """Render all frames of the environment run."""
        return self.renderer.render_all(*args, **kwargs)

    def bundle_step_data(self) -> Any:
        """Return rendering data for the current situation

        This is used to save the current state when doing batch rendering.

        Returns
        -------
        tuple[scroll, lidars, draw_list_data, step]
            data for the frames.
            scroll is the x scroll position of the state
            lidars is a list of lidar-like objects storing the lida points
            draw_list_data is a list of object drawing data
            step is the simulation step (not the frame step) for this frame
        """
        lidars = [FakeLidar([l.point1[0], l.point1[1]], l.point2) for l in self.lidar]
        draw_list_data = []
        for obj in self.drawlist:
            fixtures = []
            for fixture in obj.fixtures:
                trans = fixture.body.transform
                if isinstance(fixture.shape, Box2D.b2.circleShape):
                    translation = trans * fixture.shape.pos
                    fixtures.append(
                        [
                            "circle",
                            fixture.shape.radius,
                            copy.deepcopy(translation),
                            obj.color1,
                            obj.color2,
                        ]
                    )
                else:
                    path = [trans * v for v in fixture.shape.vertices]
                    path.append(path[0])
                    fixtures.append(["other", path[:], obj.color1, obj.color2])
            draw_list_data.append(fixtures)
        return (
            self.scroll,
            lidars,
            draw_list_data,
            self.current_step,
            self.reward_to_now,
        )

    # these are taken from the BPW in gymnasium
    observation_spaces = {
        role_name: spaces.Box(
            low=np.array(
                [
                    -math.pi,
                    -5.0,
                    -5.0,
                    -5.0,
                    -math.pi,
                    -5.0,
                    -math.pi,
                    -5.0,
                    -0.0,
                    -math.pi,
                    -5.0,
                    -math.pi,
                    -5.0,
                    -0.0,
                ]
                + [-1.0] * 10
```

```
            ).astype(np.float32),
            high=np.array(
                [
                    math.pi,
                    5.0,
                    5.0,
                    5.0,
                    math.pi,
                    5.0,
                    math.pi,
                    5.0,
                    5.0,
                    math.pi,
                    5.0,
                    math.pi,
                    5.0,
                    5.0,
                ]
                + [1.0] * 10
            ).astype(np.float32),
        )
    }

    action_spaces = {
        role_name: spaces.Box(
            np.array([-1, -1, -1, -1]).astype(np.float32),
            np.array([1, 1, 1, 1]).astype(np.float32),
        )
    }


class BipedalWalkerCustomEnhanced(BipedalWalkerCustom):
    """Same as BipedalWalkerCustom, except using the enhanced renderer"""

    _RENDER_CLASS = BipedalWalkerRendererEnhanced
```

8.7 marco_polo/envs/BipedalWalker/terrain_generation.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Functions to generate terrain for the Bipedal Walker"""
import json
from typing import Any, Optional

import numpy as np


from marco_polo.envs.BipedalWalker.bpw_constants import (
    TERRAIN_GRASS,
    TERRAIN_HEIGHT,
    TERRAIN_LENGTH,
    TERRAIN_STARTPAD,
    TERRAIN_STEP,
)
from marco_polo.envs.BipedalWalker.cppn import CppnEnvParams
from marco_polo.tools.types import FloatArray


################################################################################
## Main Functions
################################################################################
def generate_terrain_coords(
    env_params: Optional[CppnEnvParams], random: np.random.Generator
) -> tuple[FloatArray, FloatArray]:
    """Generate and return the coordinates of the terrain

    The method used for terrain depends on the values in env_params

    Parameters
    ----------
    env_params: Optional[CppnEnvParams]
        the environment parameters
    random: np.random.Generator
        The random number object to use for generating terrain

    Returns
    -------
    FloatArray, FloatArray
        arrays of x and y values of terrain
    """
    if env_params is None:
        return _flat_terrain()

    # env_params.terrain_func is optional. If missing, default to original
    terrain_func = env_params.get("terrain_function", "original")

    if terrain_func == "original":
        return _default_simple_terrain(env_params, random)
    elif terrain_func.startswith("file:"):
        return _terrain_from_file(terrain_func)

    raise ValueError("Unknown value for env_params.terrain_func")


def _flat_terrain() -> tuple[FloatArray, FloatArray]:
    """Flat line at y=0, used when there are no env_params."""
    terrain_x = np.arange(TERRAIN_LENGTH) * TERRAIN_STEP
    terrain_y = np.zeros(TERRAIN_LENGTH)
    return terrain_x, terrain_y


def _terrain_from_file(terrain_func: str) -> tuple[FloatArray, FloatArray]:
    """Read terrain from json file"""
```

```
    filename = terrain_func.replace("file:", "", 1)
    with open(filename, mode="r", encoding="utf8") as file:
        data = json.load(file)

    terrain_x = np.array([94 * (float(i) + 1) for i in data["x"]])
    terrain_y = np.array(float(i[0]) + TERRAIN_HEIGHT for i in data["y"])

    return terrain_x, terrain_y


def _default_simple_terrain(
    env_params: CppnEnvParams, random: np.random.Generator
) -> tuple[FloatArray, FloatArray]:
    """Original terrain generation, used when no generation function is given

    Parameters
    ----------
    env_params: Optional[CppnEnvParams]
        the environment parameters
    random: np.random.Generator
        The random number object to use for generating terrain

    Returns
    -------
    FloatArray, FloatArray
        arrays of x and y values of terrain
    """
    # create proper sized arrays (x array is the same as the flat terrain case)
    terrain_x, terrain_y = _flat_terrain()

    # function_x is only used for the calculation of terrain_y
    function_x = np.linspace(-np.pi, np.pi, 200, endpoint=False)

    y_val = TERRAIN_HEIGHT
    # next_change_step is to enable perioidic modifications of the terrain.
    # When i == change_step, a change is applied to the terrain.
    # For this model, the change is only to skip applying the altitude
    # function for a step. This creates minor roughness in the terrain.
    # In the orginal Gym environment, it also could change the type of
    # the terrain.
    next_change_step = TERRAIN_STARTPAD
    for i_step, x_val in enumerate(function_x):
        if i_step == next_change_step:
            # this is a "change step". The "change" here is to skip
            # updating the y value for this step. So, doing nothing.
            # Now, update the index to the next change step.
            next_change_step += random.integers(TERRAIN_GRASS // 2, TERRAIN_GRASS)
        else:
            # this is not a 'change step', so do the standard altitude
            # calculation if we are past the starting padding
            if i_step > TERRAIN_STARTPAD:
                # mypy doesn't like the altitude_fn function pointer
                y_val = TERRAIN_HEIGHT + env_params.altitude_fn((x_val,))[0]  # type:
                    ignore[no-untyped-call]
                # scale the values vertically so they start at the baseline
                # after the padding steps.
                if i_step == TERRAIN_STARTPAD + 1:
                    # mypy doesn't like the altitude_fn function pointer
                    y_norm = env_params.altitude_fn((x_val,))[0]  # type: ignore[no-untyped-call]
                y_val -= y_norm
        terrain_y[i_step] = y_val

    return terrain_x, terrain_y
```

9 marco_polo/envs/PettingZoo

9.1 marco_polo/envs/PettingZoo/__init__.py

```
1  # Copyright (c) 2023 Mobius Logic, Inc.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #    http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15
16 from .env import get_env_class
17 from .params import get_env_param_class
```

9.2 marco_polo/envs/PettingZoo/env.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from Farama-Foundation/PettingZoo
# (https://github.com/Farama-Foundation/PettingZoo)
# under the MIT License.

"""PettingZoo Connect 4 """

from pettingzoo.classic import connect_four_v3  # type: ignore[import]
import argparse  # for type hinting
from typing import Any, cast  # for type hinting
from collections.abc import Callable  # for type hinting

import logging

logging.getLogger("pettingzoo.utils.env_logger").setLevel(logging.WARNING)
import numpy as np

from marco_polo.envs.PettingZoo.params import PettingZooEnvParams
from marco_polo.tools.types import FloatArray


################################################################################
## Auxiliary Functions
################################################################################
def softmax(x: FloatArray) -> FloatArray:
    exp = np.exp(x)
    return exp / cast(float, np.exp(x).sum())


########################################################
## Factories
########################################################
def get_env_class(args: argparse.Namespace) -> Callable[..., Any]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env objects
    """
    return PettingZooEnv


################################################################################
## Main Class
################################################################################
class PettingZooEnv:
    def __init__(self) -> None:
        """ """
        self.env = connect_four_v3.env(render_mode="rgb_array")

    def __getattr__(self, name):
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""

        return getattr(self.env, name)

    def last(self):
        observation, reward, termination, truncation, info = self.env.last()
        # self.action_mask = observation["action_mask"].reshape(-1)
        # observation = np.concatenate([observation["observation"].reshape(-1),self.action_mask])
```

```
79
80          return (observation, reward, termination, truncation, info)
81
82      def step(self, act):
83          # act = np.random.sample(act)
84          act = np.random.choice(range(len(act)), p=softmax(act))
85          # if self.action_mask[act]==0:
86          #     return(True)
87
88          self.env.step(act)
89          # return(False)
90
91      def augment(self, params):
92          pass
93
94      def seed(self, seed):
95          pass
96
97      def render(self, *args, **kwargs):
98          if kwargs.get("close", False):
99              self.env.close()
100             return
101
102         return self.env.render()
```

9.3 marco_polo/envs/PettingZoo/params.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""PettingZoo Parameter Class """

from typing import Any  # for type hinting
from collections.abc import Callable  # for type hinting
from argparse import Namespace  # for type hinting
import logging

from pettingzoo.classic import connect_four_v3  # type: ignore[import]

from marco_polo.envs._base.env_params import EnvParams

logging.getLogger("pettingzoo.utils.env_logger").setLevel(logging.WARNING)


# logger = logging.getLogger(__name__)


################################################################################
## Auxiliary Functions
################################################################################
def get_env_param_class(args: Namespace) -> Callable[..., Any]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env parameter objects
    """
    return PettingZooEnvParams


################################################################################
## Main Class
################################################################################
class PettingZooEnvParams(EnvParams):
    """Parameters for PettingZoo Environments"""

    def __init__(self) -> None:
        tmp = connect_four_v3.env()
        tmp.reset()
        self.agents = tmp.agents

    def __getattr__(self, name: str) -> Any:
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
        if name.startswith("_"):
            raise AttributeError(f"accessing private attribute '{name}' is prohibited")

        if name in self.__dict__.keys():
            return self.__dict__[name]

        return getattr(self._params, name)
```

10 marco_polo/envs/_base

10.1 marco_polo/envs/_base/env_params.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Defines base class for environment parameter classes."""

from typing import Any, Optional


class EnvParams:
    """Base class for environment params."""

    def __init__(self, args: Any, param_section_name: str = "env_params") -> None:
        self.args = args
        self._params = getattr(args, param_section_name, {})

    def __getitem__(self, key: str) -> Any:
        """Return value stored for key from the params dict."""
        return self._params[key]

    def __setitem__(self, key: str, value: Any) -> None:
        """Set the value for key in the params dict."""
        self._params[key] = value

    def get(self, key: str, default: Optional[Any] = None) -> Any:
        """Return value for key from the params dict, or None if it doesn't exist."""
        try:
            return self._params[key]
        except KeyError:
            return default

    def get_mutated_params(self) -> "EnvParams":
        """Return a mutated copy of the params"""
        raise NotImplementedError(
            f"get_mutated_params has not been implemented in {type(self)}"
        )

    def checkpoint(self, folder: str) -> None:
        """Save a checkpoint in the given folder."""
        raise NotImplementedError(
            f"checkpoint has not been implemented in {type(self)}"
        )

    def reload(self, folder: str) -> None:
        """Read a checkpoint from the given folder."""
        raise NotImplementedError(f"reload has not been implemented in {type(self)}")
```

11 marco_polo/optimizers/ARS

11.1 marco_polo/optimizers/ARS/__init__.py

```
1  # Copyright (c) 2023 Mobius Logic, Inc.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #    http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15
16 from .manager import get_opt_class
17 from .modules.model import get_model
```

11.2 marco_polo/optimizers/ARS/manager.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Augmented Random Search (ARS) Optimization Manager"""

import time
import argparse
from collections import OrderedDict  # for type hinting
import json
import os
from typing import Optional, Type, Union  # for type hinting

from numpy.random import PCG64DXSM, Generator

from marco_polo.tools.types import Role, PathString  # for type hinting
from marco_polo.tools.wrappers import (
    EnvHistory,
    TeamHistory,
    RunTask,
)  # for type hinting
from marco_polo.optimizers.uber_es.manager import Manager as Base_Manager
from marco_polo.optimizers.ARS.modules.opt import ESTeamWrapper
from marco_polo.optimizers.ARS.modules.rollouts import run_optim_batch_distributed
from marco_polo.optimizers.uber_es.modules.opt import StepStats
from marco_polo.base_classes.serialCrew import SerialCrew
from marco_polo.tools.iotools import NumpyDecoder


##########################################################
## Factories
##########################################################
def get_opt_class(
    args: argparse.Namespace,
) -> Union[Type["Manager"], Type["SerialManager"]]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args : argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    Union[Type["Manager"], Type["SerialManager"]]
        the class to use in creating env objects
    """
    return Manager


##########################################################################################
## Main Class
##########################################################################################
class Manager(Base_Manager):
    """
    Compute Manager Class

    This class handles all compute by providing a standardized interface for
    optimization and evaluation. This manager subclasses the uber_es manager.
    """

    def __init__(self, setup_args: argparse.Namespace) -> None:
        """
        Manager Initializer

        Stores simulation args and sets up the compute pool.
        This primarily calls the super constructor from uber_es.
```

```
79
80          Parameters
81          ----------
82          setup_args : args.Namespace
83              Simulation parameters
84
85          Side-Effects
86          ------------
87          None
88
89          Returns
90          -------
91          None
92          """
93
94          # nothing new, just call parent
95          super().__init__(setup_args=setup_args)
96
97      ########################################
98      ## Aux Funcs
99      ########################################
100     def reload(
101         self,
102         folder: PathString,
103         cur_opts: OrderedDict[EnvHistory, TeamHistory],
104     ) -> None:
105         """
106         Reload Objects from Disk
107
108         Reads all objects to disk, calls the appropriate "Reload" functions
109         for internal objects.
110
111         Parameters
112         ----------
113         folder : PathString
114             Directory to reload objects
115         cur_opts : OrderedDict[EnvHistory, TeamHistory]
116             Dictionary of active env/team pairs
117
118         Side-Effects
119         ------------
120         Yes
121             Creates all internal objects
122
123         Returns
124         -------
125         None
126         """
127         # This is a copy from the uber_es manager
128         #  The ESTeamsWrapper needed overwritten
129         #  Adds some backwards compatability.
130
131         # Manager directory
132         if os.path.isdir(os.path.join(folder, "Manager")):
133             folder = os.path.join(folder, "Manager")
134
135         # Grab parameters
136         optArcList = []
137         with open(
138             os.path.join(folder, "Manager.json"), mode="r", encoding="utf-8"
139         ) as f:
140             dct = json.load(f, cls=NumpyDecoder)
141             self.np_random.bit_generator.state = dct["np_random"]
142             if "optimArchive" in dct.keys():
143                 optArcList = dct["optimArchive"]
144
145         # Build optimizers
146         #  env.id was used for the archive, but the bracket uses the whole env
147         #  so building a dictionary to translate env.id to env
148         envIDDict = {k.id: k for k in cur_opts.keys()}
149         for kv in optArcList:
150             # grab team from active env/team dict
151             team = cur_opts[envIDDict[kv[0]]]
152             # safety - these should match
153             assert kv[1] == team.id, "Team ID does not match optimizer archive."
154
155             # build tmp optimizer
156             tmpOpt = ESTeamWrapper(args=self.setup_args, team=team)
157             # reload
158             tmpOpt.reload(folder=os.path.join(folder, "".join(kv)))
159             # store
```

```
            self.optimArchive[(kv[0], kv[1])] = tmpOpt

    ########################################
    ## Optim Funcs
    ########################################
    def optimize_step(
        self,
        tasks: list[RunTask],
        epoch: Optional[int] = None,
        opt_iter: Optional[int] = None,
    ) -> list[dict[Role, StepStats]]:
        """
        Single Optimization Step

        This function performs a single optimization step for each active environment
        and agent pair in "tasks". "epoch" and "opt_iter" are for logging only.
        This function passes work off to "start_chunk(), then grabs results
        asynchronously and processes them as they come in.

        Parameters
        ----------
        tasks : list[RunTask]
            List of tuple["EnvHistory", "TeamHistory"] pairs, where TeamHistory is a
            dictionary of agents indexed by their roles
        epoch : int, default=None
            Current simulation time
        opt_iter : int, default=None
            Which iteration in a loop we are on

        Side-Effects
        ------------
        Yes
            This function updates environments with the optimization stats, and
            the optimization updates the agents.

        Returns
        -------
        list[dict[Role, StepStats]]
            Returns a list of length "tasks" with statistics calculated in "ESAgt.combine_and_update()"
        """
        # setup new shared list
        job_handles = []
        start_times = []
        ret_list = []

        # loop over tasks
        for env, team in tasks:
            # check if optimizer already exists for task
            if (env.id, team.id) not in self.optimArchive:
                self.optimArchive[(env.id, team.id)] = ESTeamWrapper(
                    args=self.setup_args, team=team
                )

            # grab optimizer to continue task
            ESTeam = self.optimArchive[(env.id, team.id)]

            # pass off work
            job_handles.append(
                self.start_chunk(
                    # start_chunk args
                    eval_func=run_optim_batch_distributed,
                    num_jobs=self.setup_args.uber_es["optim_jobs"],
                    num_tasks_per_job=self.setup_args.uber_es["rollouts_per_optim_job"],
                    # eval_func args
                    env_params=env.env_param,
                    env_creator_func=env.get_env_creator(),
                    team=ESTeam,
                    setup_args=self.setup_args,
                    noise_std=self.setup_args.ARS["noise_std"],
                )
            )
            # append start time
            start_times.append(time.time())

        # loop over job handles, analyze
        for sT, jH, (env, team) in zip(start_times, job_handles, tasks):
            # get results for this env/agt optim
            task_results = [handle.get() for handle in jH]

            # grab esTeam
            ESTeam = self.optimArchive[(env.id, team.id)]
```

```
241
242             # update esTeam
243             stepstats = ESTeam.combine_and_update(
244                 step_results=task_results, step_t_start=sT
245             )
246
247             # update env
248             self.optupdate(env=env, stats=stepstats)
249             env.stats.iterations_lifetime += 1
250             env.stats.iterations_current += 1
251
252             # append results
253             ret_list.append(stepstats)
254
255         # return all results
256         return ret_list
257
258
259 ################################################################################
260 ## Secondary Class
261 ################################################################################
262 class SerialManager(SerialCrew, Manager):
263     """Manager that uses a single process instead of multiprocessing
264
265     This should be a drop in replacement for the Manager class.
266
267     This is functionally different than using a Manager with one
268     worker in that the async calls are direct calls to the specified
269     function so debugging and profiling will work as expected.
270
271     Generally this would only be used for testing.
272
273     The class is intentionally blank. Using multi-inheritance, it has
274     all the methods of the Manager class, except that it uses the
275     get_crew() method from the SerialCrew class.
276     """
```

12 marco_polo/optimizers/ARS/modules

12.1 marco_polo/optimizers/ARS/modules/filter.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""ARS Observation Filter"""

# Code in this file is copied and adapted from
# https://github.com/modestyachts/ARS/blob/master/code/filter.py
# https://github.com/iamsuvhro/Augmented-Random-Search/blob/master/ars.py
#
    https://github.com/sourcecode369/Augmented-Random-Search-/blob/master/Augmented%20Random%20Search/ars.py

# So, none of these implementations are entirely correct.
# Modestyachts implements the derivation from https://www.johndcook.com/blog/skewness_kurtosis/
#  These look to be Chan's formulae for 3rd/4th central moments
# The other two implement the cited one from modestyachts, but mess up the variance
# Without going back to the original derivation, I'm just going to implement the
#  cited algorithm properly.


import numpy as np
import os
import json

from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder
from marco_polo.tools.types import FloatArray, PathString


##############################################################################
## Filter Class
##############################################################################
class Filter:
    """Observation Filter

    This class performs centering and normalization on input observations.
    This comes from the Modestyachts ARS implementation, though the algorithms
    have been rederived for correctness.
    """

    def __init__(self, num_inputs: int) -> None:
        # obs counter
        self._n = 0
        # current mean
        self._M = np.zeros(num_inputs, dtype=np.float64)
        # previous mean
        self._M_old = np.zeros(num_inputs, dtype=np.float64)
        # dunno what "S" is, possibly "sigma" 'cause it's the standard deviation
        self._S = np.zeros(num_inputs, dtype=np.float64)

        # things for normalizing obs
        self.mean = np.zeros(num_inputs, dtype=np.float64)
        self.std = np.zeros(num_inputs, dtype=np.float64)

        # update stats
        self.update_stats()

    def reset(self) -> None:
        self._n = 0
        # use subsetting to set all values
        self._M[...] = 0.0
        self._M_old[...] = 0.0
        self._S[...] = 0.0

        # update stats
```

```
75          self.update_stats()
76
77      def copy(self) -> "Filter":
78          # setup new filter
79          retFilt = Filter(num_inputs=self._M.size)
80
81          # copy internals
82          retFilt._n = self._n
83          retFilt._M = self._M.copy()
84          retFilt._M_old = self._M_old.copy()
85          retFilt._S = self._S.copy()
86
87          # update stats
88          retFilt.update_stats()
89
90          # return
91          return retFilt
92
93      def checkpoint(self, filename: PathString, newname: str) -> None:
94          # split filename
95          folder = os.path.dirname(filename)
96          # replace file name and save
97          with open(os.path.join(folder, newname), mode="w", encoding="utf-8") as f:
98              json.dump(self.__dict__, f, cls=NumpyEncoder)
99
100     def reload(self, filename: PathString, newname: str) -> None:
101         # split filename
102         folder = os.path.dirname(filename)
103         # replace file name and save
104         with open(os.path.join(folder, newname), mode="r", encoding="utf-8") as f:
105             self.__dict__ = json.load(f, cls=NumpyDecoder)
106
107     def observe(self, x: FloatArray) -> None:
108         """
109         Welford's basic online algorithm
110
111         https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_online_algorithm
112         """
113         # increment number of samples
114         self._n += 1
115
116         # update
117         # mean
118         #  at this point, "M" and "M_old" are the same, so we can use "+=" operator
119         self._M += (x - self._M_old) / self._n
120         # "S"
121         #  this requires "M" and "M_old", which is why we have 2
122         self._S += (x - self._M_old) * (x - self._M)
123         # old mean
124         #  at the next iteration, the current will be "old"
125         self._M_old = self._M.copy()
126
127     def update_stats(self) -> None:
128         # std deviation
129         #  modestyachts use _M as the default std
130         #   maybe to avoid zeros?
131         self.std = np.sqrt(self._S / (self._n - 1)) if (self._n > 1) else self._M.copy()
132
133         # modestyachts safety check
134         # Set values for std less than 1e-7 to +inf to avoid
135         # dividing by zero. State elements with zero variance
136         # are set to zero as a result.
137         self.std[self.std < 1e-7] = float("inf")
138
139         # mean
140         self.mean = self._M
141
142     def normalize(self, x: FloatArray) -> FloatArray:
143         return (x - self.mean) / self.std
144
145     def sync(self, other: "Filter") -> None:
146         """
147         Syncs fields from other filter
148
149         Updates the internal state to match the supplied, "other", filter
150         """
151         self._n = other._n
152         self._M = other._M.copy()
153         self._M_old = other._M_old.copy()
154         self._S = other._S.copy()
155
```

```
156         # update stats
157         self.update_stats()
158 
159     def update(self, other: "Filter") -> None:
160         """
161         Chan's general solution of Welford's Algorithm
162 
163         This is actually a blend, 'cause Chan's mean calculation is unstable.
164         I can't find anything about stability issues with S.
165         Also can't tell if it's a population variance or a sample variance.
166         (I think it's population and it's biased, but not positive).
167 
168         http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf
169         https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_algorithm
170         """
171         # grab number of samples
172         n_self = self._n
173         n_other = other._n
174 
175         # difference of means
176         delta = self._M - other._M
177 
178         # update total observation count
179         self._n = n_self + n_other
180 
181         # update mean
182         self._M = ((n_self * self._M) + (n_other * other._M)) / self._n
183 
184         # update S
185         #  I have numerical stability questions about this
186         self._S = self._S + other._S + delta * delta * n_self * n_other / self._n
187 
188     def __repr__(self) -> str:
189         return f"(n={self._n}, mean_M={self._M.mean()}, mean_S={self._S.mean()})"
```

12.2 marco_polo/optimizers/ARS/modules/model.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


"""ARS Basic Agent Model"""

import json

# import logging
import argparse  # for type hinting

import numpy as np
from numpy.random import PCG64DXSM, Generator
from os import path as osp
from typing import Any, Optional, Union

from marco_polo.base_classes.model import BaseModel
from marco_polo.optimizers.uber_es.modules.model import Model as Uber_Model

from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder
from marco_polo.optimizers.ARS.modules.filter import Filter
from pettingzoo.utils.env import ParallelEnv  # type: ignore[import]
from marco_polo.tools.types import FloatArray, Role, PathString  # for type hinting

# logger = logging.getLogger(__name__)


################################################################################
## Agent Factory
################################################################################
def get_model(
    env: ParallelEnv, role: Role, args: argparse.Namespace, seed: int
) -> BaseModel:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    env : pettingzoo.util.env.ParallelEnv
        A parallel env that holds to pettingzoos's interface, not used here
    role : Role
        Agent role in env, not used here
    args : argparse.Namespace
        arguments that were passed to the `main()` function
    seed : int
        seed for agent creation

    Returns
    -------
    BaseModel
        agent policy model
    """
    return Model(model_params=args.model_params, seed=seed)


################################################################################
## Aux Funcs
################################################################################


################################################################################
## Model Class
################################################################################
class Model(Uber_Model):
```

```
    """simple feedforward model"""

    def __init__(
        self,
        model_params: dict[str, Any],
        seed: int,
        args: Optional[argparse.Namespace] = None,
    ) -> None:
        """Constructor for ARS Model

        Child class of the Uber ES model.

        Parameters
        ----------
        model_params : dict[str, Any]
            Dictionary defining model structure
        seed : int
            Initial value for internal PRNG
        args : Optional[argparse.Namespace], default=None
            Values to parameters the rest of the model

        Side-Effects
        ------------
        Yes
            Sets all class attributes

        Returns
        -------
        None
        """
        # super constructor
        super().__init__(model_params=model_params, seed=seed, args=args)

        # setup observation filter and storage buffer
        self.save_obs = False
        self.normalize_obs = model_params.get("normalize_obs", True)
        self.obs_buf = Filter(num_inputs=self.input_size)
        self.obs_filt = Filter(num_inputs=self.input_size)

    # aux getters/setters
    #  These are incredibly important due to the use of a wrapper class
    def set_obs_flag(self, flag: bool) -> None:
        self.save_obs = flag

    def get_obs_flag(self) -> bool:
        return self.save_obs

    def get_obs_buf(self) -> Filter:
        return self.obs_buf

    def get_obs_filt(self) -> Filter:
        return self.obs_filt

    def get_action(
        self, x: FloatArray, t: float = 0, mean_mode: bool = False
    ) -> Union[np.int_, FloatArray]:
        """Generate Model Action

        Parameters
        ----------
        x : FloatArray
            Input observations
        t : float, default=0
            Time scale
        mean_mode : bool, default=False
            Generate noise in the network and maintain the average?

        Side-Effects
        ------------
        Yes
            Updates observation buffer
            Uses internal PRNG

        Returns
        -------
        Union[np.int_, FloatArray]
            Agent action
        """
        # flatten observations
        #  this will get done twice, the second time in super().get_action()
        h = np.array(x).flatten()
```

```
160
161         # when doing optimization, save the observations for update later
162         if self.save_obs:
163             self.obs_buf.observe(x=h)
164
165         # normalize observations?
166         if self.normalize_obs:
167             h = self.obs_filt.normalize(x=h)
168
169         return super().get_action(x=h, t=t, mean_mode=mean_mode)
170
171     def reload(self, filename: PathString) -> None:
172         # read state
173         with open(filename, mode="r", encoding="utf-8") as f:
174             data = json.load(f, cls=NumpyDecoder)
175
176         # load state into object
177         self.theta = data["theta"]
178         self.np_random.bit_generator.state = data["np_random"]
179         self.set_model_params(model_params=self.theta)
180
181         self.save_obs = data.get("save_obs", False)
182         self.normalize_obs = data.get("normalize_obs", False)
183
184         # load observation filters if exist
185         if osp.isfile(osp.join(osp.dirname(filename), "obs_buf.json")):
186             self.obs_buf.reload(filename=filename, newname="obs_buf.json")
187
188         if osp.isfile(osp.join(osp.dirname(filename), "obs_filt.json")):
189             self.obs_filt.reload(filename=filename, newname="obs_filt.json")
190
191     def checkpoint(self, filename: PathString) -> None:
192         # build dict to save
193         manifest = {
194             "theta": self.theta,
195             "np_random": self.np_random.bit_generator.state,
196             "save_obs": self.save_obs,
197             "normalize_obs": self.normalize_obs,
198         }
199
200         # save state
201         with open(filename, mode="w", encoding="utf-8") as f:
202             json.dump(manifest, f, cls=NumpyEncoder)
203
204         # save observation filters
205         self.obs_buf.checkpoint(filename=filename, newname="obs_buf.json")
206         self.obs_filt.checkpoint(filename=filename, newname="obs_filt.json")
207
208     def shift_weights(self, noise_std: float, seed: int) -> None:
209         # set random state
210         random_state = Generator(PCG64DXSM(seed=seed))
211         # draw normal
212         t = noise_std * random_state.standard_normal(size=self.actor_param_count)
213         # update local theta
214         theta = self.theta + t
215         # set model parameters to this theeta
216         #  NOTE: it no longer matches the model theta!
217         self.set_model_params(theta)
```

12.3 marco_polo/optimizers/ARS/modules/opt.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""ARS Team Wrapper and Optimization Controller"""

import json
import logging
import os
import time
from typing import Any, Union  # for type hinting
import argparse  # for type hinting

import numpy as np
from numpy.random import PCG64DXSM, Generator

from marco_polo.optimizers.uber_es.modules.opt import ESTeamWrapper as Base_TeamWrapper
from marco_polo.optimizers.uber_es.modules.opt import StepStats
from marco_polo.tools.stats import compute_weighted_sum

from marco_polo.optimizers.ARS.modules.result_objects import (
    POResult,
)  # for type hinting
from marco_polo.optimizers.ARS.modules.filter import Filter  # for type hinting
from marco_polo.tools.types import FloatArray, Role, PathString  # for type hinting
from marco_polo.tools.wrappers import TeamHistory  # for type hinting
from marco_polo.tools.iotools import NumpyDecoder


logger = logging.getLogger(__name__)


################################################################################
## Aux Objects
################################################################################


################################################################################
## Main Class
################################################################################
class ESTeamWrapper(Base_TeamWrapper):
    """Wrapper for the optimizers that apply to a given team

    By default, the neural network models of all agents are updated by
    the methods in this class. However, a set of roles can be marked
    as 'frozen' to prevent the agents from being updated. This can be
    done via the parameter file using the key uber_es.freeze.
    Alternatively, the list of frozen roles can be changed later
    (see add_freeze(), remove_freeze(), and remove_all_freeze())
    """

    def __init__(self, args: argparse.Namespace, team: TeamHistory) -> None:
        """ES Team Wrapper for ARS Optimizer

        Parameters
        ----------
        args : argparse.Namespace
            Input arguments to parameterize Team and Optimizer
        team : TeamHistory
            Team of agents

        Returns
        -------
        None
        """
        self.team = team
        self.args = args
```

```
79
80          # see if freeze key is in args
81          try:
82              self._freeze = set(args.uber_es["freeze"])
83          except (AttributeError, KeyError):
84              self._freeze = set()
85
86          # build optimizer dict
87          self.optimizers = {}
88          for role in team.roles():
89              self.optimizers[role] = ARS(args=args)
90
91      def combine_and_update(
92          self,
93          step_results: list[dict[Role, POResult]],
94          step_t_start: float,
95          decay_noise: bool = False,
96          propose_only: bool = False,
97      ) -> dict[Role, StepStats]:
98          """Combine results from rollouts, pass to the optimizer, and update models.
99
100         Agents with roles that are listed in the freeze set will have the statistics
101         calculated but will not have their models updated.
102
103         Parameters
104         ----------
105         step_results : list[dict[Role, POResult]]
106             Results from runs
107         step_t_start : float
108             Time when this set of steps started (used for calculating runtime)
109         decay_noise : bool, default=True
110             Whether to have the applied noise decay during the optimization
111         propose_only : bool, default=False
112             If true, only the stats will be calculated and no updates will be made
113
114         Return
115         ------
116         dict[Role, StepStats]
117             Statistics from the runs for each role
118         """
119         # return dict
120         #  organize by agent key
121         stats = dict()
122
123         # loop over agents, update their network
124         for role in self.team.roles():
125             # grab sim results for this agent
126             results = [r[role] for r in step_results]
127
128             # freeze is implemented by using propose_only
129             if role in self._freeze:
130                 _propose_only = True
131             else:
132                 _propose_only = propose_only
133
134             # calculate stats and new theta
135             new_theta, stats[role] = self.optimizers[role].combine_steps(
136                 step_results=results,
137                 theta=self.team[role].theta,
138                 obs_filt=self.team[role].get_obs_filt(),
139                 step_t_start=step_t_start,
140                 propose_only=_propose_only,
141             )
142
143             # update agent
144             if not _propose_only:
145                 self.team[role].update_theta(new_theta=new_theta)
146                 self.team[role].set_model_params(model_params=new_theta)
147                 # obs_filt was updated in-place
148                 self.team[role].get_obs_filt().update_stats()
149             self.team[role].get_obs_buf().reset()  # should be redundant
150
151         # return results
152         return stats
153
154
155 ################################################################################
156 ## Optimization Controller
157 ################################################################################
158 class ARS:
159     """
```

```
    Augmented Random Search

    Algorithm V2-t from https://arxiv.org/abs/1803.07055
    """

    def __init__(self, args: argparse.Namespace) -> None:
        """


        Parameters
        ----------
        args : argparse.Namespace
            Input arguments


        Side-Effects
        ------------
        Yes
            Sets class attributes

        Return
        ------
        None
        """
        #
        self.alpha = args.ARS["learning_rate"]  # learning-rate/step-size
        self.noise_std = args.ARS["noise_std"]  # std deviation of the exploration noise

        # check top fraction to keep, then convert to int for use
        assert args.ARS["top_frac"] <= 1, "top_frac must be <= 1"
        assert args.ARS["top_frac"] >= 0, "top_frac must be >= 0"
        self.top_frac = args.ARS["top_frac"]
        self.top_n = int(
            round(
                args.ARS["top_frac"]
                * args.uber_es["optim_jobs"]
                * args.uber_es["rollouts_per_optim_job"]
            )
        )

    def __str__(self) -> str:
        return "ARS Optimization Controller"

    def checkpoint(self, folder: PathString, role: Role) -> None:
        """Save ARS Optimizer to Disk

        Parameters
        ----------
        folder : PathString
            Directory to store output
        role : Role
            Agent role for file naming

        Side-Effects
        ------------
        None

        Return
        ------
        None
            Saves output to disk
        """
        with open(
            os.path.join(folder, f"{role}_opt.json"), mode="w", encoding="utf-8"
        ) as f:
            json.dump(self.__dict__, f)

    def reload(self, folder: PathString, role: Role) -> None:
        """Load ARS Optimizer From Disk

        Parameters
        ----------
        folder : PathString
            Directory to read input
        role : Role
            Agent role for file naming

        Side-Effects
        ------------
        Yes
            Sets optimizer attributes
```

```
241
242             Return
243             ------
244             None
245             """
246             with open(
247                 os.path.join(folder, f"{role}_opt.json"), mode="r", encoding="utf-8"
248             ) as f:
249                 self.__dict__ = json.load(f, cls=NumpyDecoder)
250
251         def get_noise(self, seed: int, theta_len: int) -> FloatArray:
252             """Generate Random Noise for Gradient Estimation
253
254             This function generates exploratory noise for simulation rollouts.
255             The noise follows a standard normal distribution.
256
257             Parameters
258             ----------
259             seed : int
260                 Initial value for the random generator.
261             theta_len : int
262                 How many random numbers to return
263
264             Side-Effects
265             ------------
266             None
267
268             Return
269             ------
270             FloatArray
271                 Array of random floats of length theta
272             """
273             # set random state
274             random_state = Generator(PCG64DXSM(seed=seed))
275             # draw normal distribution
276             return random_state.normal(scale=self.noise_std, size=theta_len)
277
278         def calc_max_min_theta(
279             self, noise_seeds: FloatArray, returns: FloatArray, theta: FloatArray
280         ) -> tuple[FloatArray, FloatArray]:
281             """
282             Holdover from the original POET Code
283
284             Poet calculated max/min theta values after the update, but differently from
285             how it actually calculates the gradient. I'm separating it out, adding a min,
286             and then keeping it for historic purposes only.
287
288             Parameters
289             ----------
290             noise_seeds : FloatArray
291                 Array of seed values used to generate the random noise
292             returns : FloatArray
293                 Rewards values from the simulation rollouts
294             theta : FloatArray
295                 Current parameterization of the model
296
297             Side-Effects
298             ------------
299             None
300
301             Returns
302             -------
303             tuple[FloatArray, FloatArray]
304                 Min/Max point estimates of network parameterizations
305             """
306             # used for both
307             theta_len = len(theta)
308             retList: list[FloatArray] = []
309
310             # loop over max/min
311             for i in range(0, 2):
312                 # setup
313                 pos_row, neg_row = (
314                     returns.argmax(axis=0) if (i == 0) else returns.argmin(axis=0)
315                 )
316                 noise_sign = 1.0
317                 noise_seed = noise_seeds[pos_row]
318
319                 # check
320                 if returns[pos_row, 0] < returns[neg_row, 1]:
321                     noise_sign = -1.0
```

```
                noise_seed = noise_seeds[neg_row]

            # calculate
            retList.append(
                theta
                + noise_sign * self.get_noise(seed=noise_seed, theta_len=theta_len)
            )

        return retList[0], retList[1]

    def combine_steps(
        self,
        step_results: list[POResult],
        theta: FloatArray,
        obs_filt: Filter,
        step_t_start: float,
        propose_only: bool = False,
    ) -> tuple[FloatArray, StepStats]:
        """Calculate Statistics from All Simulation Rollouts

        If "propose_only" is true, then only statistics are calculated but no
        updates are performed.


        Parameters
        ----------
        step_results : list[POResult]
            Results from the simulation rollouts
        theta : FloatArray
            Current network parameterization
        obs_filt : Filter
            The model observation filter
        step_t_start : float
            Time that this calculation started
        propose_only : bool, default=False
            Do updates or only calculate statistics?

        Side-Effects
        ------------
        Yes
            Updates the observation filter for the model

        Returns
        -------
        tuple[FloatArray, StepStats]
            Tuple containing the new network parameterization and statistics from
            the simulations
        """
        # constants
        theta_len = len(theta)

        # Extract results
        nList = []
        rList = []
        lList = []
        oList = []
        for r in step_results:
            nList.append(r.noise_inds)
            rList.append(r.returns)
            lList.append(r.lengths)
            oList.append(r.obs_buf)

        # reshape results
        noise_inds = np.concatenate(nList)
        po_returns = np.concatenate(rList)
        po_lengths = np.concatenate(lList)
        po_obs = np.concatenate(oList)

        # sort and subset results
        #  get max of +/- rollouts
        #  partition top n - idk why the "-" in the kth position
        #  grab just the top n
        top_n_idx = po_returns.max(axis=1).argpartition(kth=-self.top_n)[-self.top_n :]

        noise_inds = noise_inds[top_n_idx]
        po_returns = po_returns[top_n_idx, :]
        po_lengths = po_lengths[top_n_idx]
        po_obs = po_obs[top_n_idx]

        # get max/min possible grads
        #  has to be after subsetting results, but before theta update
```

```
        po_theta_max, po_theta_min = self.calc_max_min_theta(
            noise_seeds=noise_inds, returns=po_returns, theta=theta
        )

        # calculate gradients and update theta
        ret_std = po_returns.std()
        w_sum, _ = compute_weighted_sum(
            weights=(po_returns[:, 0] - po_returns[:, 1]) / ret_std,
            vec_generator=(
                self.get_noise(seed=seed, theta_len=theta_len) for seed in noise_inds
            ),
            theta_size=theta_len,
        )

        # Calculate gradient
        grads = w_sum * self.alpha / self.top_n  # need for logging

        # are we updating?
        if not propose_only:
            # update theta
            theta += grads

            # update observations
            for obs in po_obs:
                obs_filt.update(other=obs)

        # grab final time
        step_t_end = time.time()

        # return new theta and stats
        return theta, StepStats(
            po_returns_mean=po_returns.mean(),
            po_returns_median=np.median(po_returns),
            po_returns_std=po_returns.std(),
            po_returns_max=po_returns.max(),
            po_theta_max=po_theta_max,
            po_returns_min=po_returns.min(),
            po_len_mean=po_lengths.mean(),
            po_len_std=po_lengths.std(),
            noise_std=self.noise_std,
            learning_rate=self.alpha,
            theta_norm=np.square(theta).sum(),
            grad_norm=float(np.square(grads).sum()),
            update_ratio=float(0.0),
            episodes_this_step=self.top_n,
            timesteps_this_step=po_lengths.sum(),
            time_elapsed_this_step=step_t_end - step_t_start,
        )
```

12.4 marco_polo/optimizers/ARS/modules/result_objects.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""ARS Result Objects"""
from collections import namedtuple

# this adds obs_buf to the UBER_ES POResult object
POResult = namedtuple(
    "POResult", ["noise_inds", "returns", "lengths", "rollouts", "obs_buf"]
)
```

12.5 marco_polo/optimizers/ARS/modules/rollouts.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


"""ARS Simulation Rollouts"""

import logging
from collections import namedtuple
import argparse  # for type hinting
from collections.abc import Callable  # for type hinting
from typing import Any  # for type hinting

import numpy as np
from numpy.random import PCG64DXSM, Generator

from marco_polo.optimizers.uber_es.modules.rollouts import _simulate_basic
from marco_polo.optimizers.ARS.modules.opt import ESTeamWrapper  # for type hinting
from marco_polo.tools.types import Role  # for type hinting
from marco_polo.envs._base.env_params import EnvParams  # for type hinting
from marco_polo.optimizers.ARS.modules.result_objects import POResult

logger = logging.getLogger(__name__)


################################################################################
## Aux Objects
################################################################################


########################################
## Optim Funcs
########################################
def run_optim_batch_distributed(
    num_rollouts: int,
    rs_seed: int,
    env_params: EnvParams,
    env_creator_func: Callable[..., Any],
    team: ESTeamWrapper,
    setup_args: argparse.Namespace,
    noise_std: float,
) -> dict[Role, POResult]:
    """
    Distributed Evaluation Routine

    This fuction is design for asynchronous evaluations. It does not track frames
    or output images, just tracks actions, rewards, and simulation length. It calls
    "_simulate_basic()" to perform simulations, then handles results for each agent.

    Parameters
    ----------
    num_rollouts : int
        How many evaluations to perform
        The actual amount is 2x this, since we eval + and - gradient perturbations
    rs_seed : int
        Random seed for PRNG
    env_params : EnvParams
        Parameter object specifying the environment
    env_creator_func : Callable[..., Any]
        Function to create the environment from the CPPN
        This is an artificant of pickle, because the full env can't be sent over pipes
    team : ESTeamWrapper
        Dictionary of agents to play
    setup_args : argparse.Namespace
```

```
        Input args for the agent wrapper
    noise_std : float
        Standard deviation when drawing gradient perturbations

    Side-Effects
    ------------
    None
        When run asynchronously, everything is copied to this func, so no side-effects

    Returns
    -------
    dict[role, POResult]
        Dictionary of POResult objects for each agent
    """
    # set new prng
    random_state = Generator(PCG64DXSM(seed=rs_seed))
    # draw seeds for weight shifting
    seeds = random_state.integers(low=0, high=(2**63), size=(num_rollouts,))

    # setup simulator
    env = env_creator_func()
    env.augment(env_params)

    # turn on observation saving
    for agt in team.values():
        agt.set_obs_flag(flag=True)

    # simulation return objects
    returns = {role: np.zeros(shape=(num_rollouts, 2)) for role in team.roles()}
    observations = {role: [None] * num_rollouts for role in team.roles()}
    lengths = np.zeros(shape=(num_rollouts, 2), dtype="int")
    agt_seeds = []
    results = dict()

    # loop over rollouts, do + and - point estimates
    for i, seed in enumerate(seeds):
        # reset observation buffer
        for agt in team.values():
            agt.get_obs_buf().reset()

        # shift weights and store seed
        agt_seeds.append(team.shift_weights(noise_std=noise_std, seed=seed))

        # first rollout
        r, lengths[i, 0], _ = _simulate_basic(
            env=env, team=team, random_state=random_state
        )

        # update returns
        for role in team.roles():
            returns[role][i, 0] = r[role]

        # shift weights in other direction
        _ = team.shift_weights(noise_std=-noise_std, seed=seed)

        # second rollout
        r, lengths[i, 1], _ = _simulate_basic(
            env=env, team=team, random_state=random_state
        )

        # store returns and observations
        for role, agt in team.items():
            returns[role][i, 1] = r[role]
            observations[role][i] = agt.get_obs_buf().copy()

    # turn off observation saving
    for agt in team.values():
        agt.set_obs_flag(flag=False)

    # loop over agents, fill return
    for role in team.keys():
        results[role] = POResult(
            returns=returns[role],
            noise_inds=[a[role] for a in agt_seeds],
            lengths=lengths,
            rollouts=[],
            obs_buf=observations[role],
        )

    # return results
    return results
```

13 marco_polo/optimizers/tianshou

13.1 marco_polo/optimizers/tianshou/__init__.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


from .manager import get_opt_class
from .modules.model import get_model
```

13.2 marco_polo/optimizers/tianshou/manager.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Tianshou Manager"""

import logging
import os
import pprint
import torch
import numpy as np
import argparse  # for type hinting
from tianshou.data import Batch, Collector, ReplayBuffer, VectorReplayBuffer  # type: ignore[import]
from tianshou.env import DummyVectorEnv, ShmemVectorEnv, SubprocVectorEnv  # type: ignore[import]
from tianshou.policy import MultiAgentPolicyManager  # type: ignore[import]
from tianshou.trainer import offpolicy_trainer, onpolicy_trainer  # type: ignore[import]

from collections import OrderedDict  # for type hinting
from collections.abc import Callable  # for type hinting
from typing import Any, Optional, Union, Type  # for type hinting

from marco_polo.tools.wrappers import (
    EnvHistory,
    TeamHistory,
    RunTask,
)  # for type hinting
from marco_polo.tools.types import Role, PathString  # for type hinting


from marco_polo.optimizers.uber_es.manager import Manager as Base_Manager
from marco_polo.tools.wrappers import MP_parallel_to_aec, MP_PettingZooEnv
from marco_polo.envs._base.env_params import EnvParams


logger = logging.getLogger(__name__)


############################################################
## Factories
############################################################
def get_opt_class(
    args: argparse.Namespace,
) -> Type["Manager"]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    class
        the class to use in creating env objects
    """
    return Manager


############################################################
## Utilities
############################################################
def flatten_dict_of_lists(dct: dict[Any, list[Any]]) -> list[Any]:
    """Flatten a dictionary of lists into a single list."""
    flattened_lst = []
    for key, value in dct.items():
        if isinstance(value, list):
            flattened_lst.extend(value)  # Append the list values to the flattened list
```

```
79          else:
80              flattened_lst.append(value)
81      return flattened_lst
82
83
84  ## Needs to be worked on - there's something else going
85  # on that's killing randomity...
86  class SeedingCollector(Collector):  # type: ignore[misc] # Collector is Any
87      def collect(self, *args: Any, **kwargs: Any) -> dict[str, Any]:
88          kwargs["gym_reset_kwargs"] = {"seed": 0}
89          return super().collect(*args, **kwargs)  # type: ignore[no-any-return] # mypy can't see
                superclass
90
91
92  ################################################################################
93  ## Main Class
94  ################################################################################
95  class Manager(Base_Manager):
96      """
97      Compute Manager Class
98
99      This class handles all compute by providing a standardized interface for
100     optimization and evaluation. This manager is based on tianshou and is under
101     development.
102
103     Attributes
104     ----------
105     setup_args : argparse.Namespace
106         Seems to be like a dictionary
107         This stores all of the simulation parameters
108     np_random : Generator
109         Numpy random generator
110     crew : Multiprocessing Pool
111         Worker pool for asynchronous evaluation
112
113     Methods
114     -------
115     Manager(argparse.Namespace, list[Func])
116         Initializer, stores functions, creates worker pool
117     checkpoint(str)
118         Store output
119     reload(str)
120         Reload output from checkpoint
121     evalupdate(env, stat)
122         Pass eval stats to the appropriate environment update function
123     optupdate(env, stat)
124         Pass optimization stats to the appropriate environment update function
125     start_chunk(func, int, int, **kwargs)
126         Passes func off to worker pool, returns job handles
127     evaluate(list[(env, team)], int, bool)
128         Main evaluation function
129     _visualize(list[(env, team)], int)
130         Non-blocking video generating function
131     optimize_step(list[(env, team)], int, int)
132         Single optimization step
133     optimize_chunk(list[(env, team)], int, bool)
134         Main optimization function, with evals and video generation
135     """
136
137     def __init__(self, setup_args: argparse.Namespace) -> None:
138         """
139         Manager Initializer
140
141         Stores simulation args and sets up the compute pool
142
143         Parameters
144         ----------
145         setup_args : args.NAmespace
146             Simulation parameters
147
148         Side-Effects
149         ------------
150         None
151
152         Returns
153         -------
154         None
155         """
156         # torch sees through pods, and schedules based on the blade resources
157         # this lets us control how many resources torch claims
158         torch.manual_seed(setup_args.master_seed)
```

```
159             CORES = setup_args.tianshou["torch_threads"]
160             torch.set_num_threads(CORES)
161             torch.set_num_interop_threads(CORES)
162             super().__init__(setup_args=setup_args)
163
164         ########################################
165         ## Aux Funcs
166         ########################################
167         def checkpoint(self, folder: PathString, cur_opts: list[RunTask]) -> None:
168             pass
169
170         def reload(
171             self, folder: PathString, cur_opts: OrderedDict[EnvHistory, TeamHistory]
172         ) -> None:
173             pass
174
175         ########################################
176         ## Optim Funcs
177         ########################################
178         def optimize_step(
179             self,
180             tasks: list[RunTask],
181             epoch: Optional[int] = None,
182             opt_iter: Optional[int] = None,
183         ) -> list[dict[Role, list[Any]]]:
184             # idk what is in the list at the bottom
185             #  maybe floats?
186             """ """
187             return self.optimize_chunk(tasks=tasks, epoch=1)
188
189         def _create_env(
190             self,
191             args: argparse.Namespace,
192             env_creator: Callable[..., Any],
193             env_params: EnvParams,
194             **kwargs: dict[str, Any],
195         ) -> MP_PettingZooEnv:
196             env = env_creator(**kwargs)
197             env.augment(params=env_params)
198             env.seed(seed=self.np_random.integers(low=2**31 - 1, dtype=int))
199             return MP_PettingZooEnv(args, MP_parallel_to_aec(env))
200
201         def optimize_chunk(
202             self,
203             tasks: list[RunTask],
204             epoch: Optional[int] = None,
205             verbose: Optional[bool] = False,
206         ) -> list[dict[Role, list[Any]]]:
207             results = []
208             args = self.setup_args
209
210             epoch_to_run = args.tianshou["epoch"]
211
212             returns = [{k: [] for k in agt.keys()} for (_, agt) in tasks]
213
214             for i, (env, agts) in enumerate(tasks):
215                 logger.info(f"Optimizing team: {agts.id} on env: {env.id}")
216
217                 # if (env.id, agts.id) in self.optimArchive.keys():
218                 #     agts = self.optimArchive[(env.id, agts.id)]
219
220                 env_creator = env.get_env_creator()
221
222                 # print(agts.keys())
223                 # if len(agts.keys())>1:
224                 tmp_env = self._create_env(
225                     args=self.setup_args, env_creator=env_creator, env_params=env.env_param
226                 )
227                 agts_list = [agts[k] for k in tmp_env.agents]
228                 policy = MultiAgentPolicyManager(agts_list, tmp_env)
229                 agents = tmp_env.agents
230
231                 train_envs = SubprocVectorEnv(
232                     [
233                         lambda: self._create_env(
234                             args=self.setup_args,
235                             env_creator=env_creator,
236                             env_params=env.env_param,
237                         )
238                         for _ in range(args.tianshou["train_num"])
239                     ]
```

```
        )

        test_envs = SubprocVectorEnv(
            [
                lambda: self._create_env(
                    args=self.setup_args,
                    env_creator=env_creator,
                    env_params=env.env_param,
                )
                for _ in range(args.tianshou["test_num"])
            ]
        )

        if args.tianshou["train_num"] > 1:
            buffer = VectorReplayBuffer(
                args.tianshou["step_per_collect"], len(train_envs)
            )
        else:
            buffer = ReplayBuffer(args.tianshou["step_per_collect"])

        train_collector = SeedingCollector(
            policy, train_envs, buffer, exploration_noise=True
        )

        # At some point we may want to write a test collector
        reward_list: dict[Role, dict[Any, Any]] = {
            agt: dict() for agt in tmp_env.agents
        }
        steps: dict[str, list[int]] = dict()

        ####################
        ## Inner Funcs
        ####################
        def save_best_fn(policy) -> None:
            torch.save(policy.state_dict(), os.path.join(log_path, "policy.pth"))

        def setup_eval_collect(num_epoch: int, step_idx: int) -> None:
            pass
            # reward_list.append(dict())

        def save_collecter_rewards(**kwargs: Any) -> dict[Any, Any]:
            returns = dict()
            ## Grab the reward and save it to a external buffer
            env_id = kwargs["env_id"][0]

            for rl in reward_list.values():
                if not env_id in rl.keys():
                    rl[env_id] = [0]

            if not env_id in steps.keys():
                steps[env_id] = [0]

            if "rew" not in kwargs.keys():
                ## At beg/end of run, set up next storage
                ## only if positve number of steps
                if steps[env_id][-1] > 0:
                    for rl in reward_list.values():
                        rl[env_id].append(0)

                    steps[env_id].append(0)
            else:
                agt = kwargs["obs_next"].agent_id[0]
                steps[env_id][-1] += 1
                reward_list[agt][env_id][-1] += kwargs["rew"][0][0]

            return returns

        ####################
        ## End Inner Funcs
        ####################
        test_collector = SeedingCollector(
            policy, test_envs, preprocess_fn=save_collecter_rewards
        )

        # trainer
        result = onpolicy_trainer(
            policy,
            train_collector,
            test_collector,
            epoch_to_run,
            args.tianshou["step_per_epoch"],
```

```
                args.tianshou["repeat_per_collect"],
                args.tianshou["test_num"],
                args.tianshou["batch_size"],
                step_per_collect=args.tianshou["step_per_collect"],
                test_fn=setup_eval_collect,
                # save_best_fn=save_best_fn,
                # logger=logger,
                test_in_train=False,
            )

            pprint.pprint(result)

            # print(reward_list, flatten_dict_of_lists(steps))

            flat_steps = np.array(flatten_dict_of_lists(steps))
            for agt, rwds_ in reward_list.items():
                rwds = np.array(flatten_dict_of_lists(rwds_))
                rwd_value = np.mean(rwds[flat_steps > 0])
                returns[i][agt].append(rwd_value)

            # self.optimArchive[(env.id, agts.id)] = agts
            logger.info(f"Finished Optimizing team: {agts.id} on env: {env.id}")

    return returns
```

14 marco_polo/optimizers/tianshou/modules

14.1 marco_polo/optimizers/tianshou/modules/model.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


# import pickle
import os

# from argparse import ArgumentParser
import argparse
from typing import Any, Union
import gymnasium as gym
import numpy as np
import copy

# import torch
from tianshou.data import Batch  # type: ignore[import]
from pettingzoo.utils.env import ParallelEnv  # type: ignore[import]
from tianshou.policy import BasePolicy, PPOPolicy  # type: ignore[import]
from tianshou.utils.net.common import Net  # type: ignore[import]
from tianshou.utils.net.continuous import ActorProb, Critic  # type: ignore[import]

import torch
from torch import nn
from torch.distributions import Independent, Normal
from torch.distributions.distribution import Distribution
from torch.distributions.uniform import Uniform
from torch.optim.lr_scheduler import LambdaLR

from marco_polo.tools.types import FloatArray, Role, PathString


############################################################
## Factories
############################################################
def get_model(
    env: ParallelEnv, role: Role, args: argparse.Namespace, seed: int
) -> BasePolicy:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    env: pettingzoo.util.env.ParallelEnv
        A parallel env that holds to pettingzoos's interface
    role: String
        Agent role in env
    args: argparse.Namespace
        arguments that were passed to the `main()` function
    seed: int
        seed for agent creation

    Returns
    -------
    policy
        agent policy model
    """

    actor, critic = construct_actor_critic(env=env, player_name=role, args=args)
    policy = construct_policy(actor=actor, critic=critic, args=args)

    return policy


############################################################
## Auxiliary Functions
```

```
76  ######################################################
77  def dist(*logits: Any) -> Distribution:
78      return Independent(Normal(logits[0], scale=logits[1] * 0.1), 1)
79
80
81  def construct_actor_critic(
82      env: ParallelEnv, player_name: str, args: argparse.Namespace
83  ) -> tuple[ActorProb, Critic]:
84      observation_space = (
85          env.observation_spaces["observation"]
86          if isinstance(env.observation_spaces, gym.spaces.Dict)
87          else env.observation_spaces
88      )
89
90      action_space = (
91          env.action_spaces["action"]
92          if isinstance(env.action_spaces, gym.spaces.Dict)
93          else env.action_spaces
94      )
95
96      # print(observation_space)
97      # print(action_space)
98      max_action = action_space[player_name].high[0]
99
100     # model
101     a_net = Net(
102         state_shape=observation_space[player_name].shape or observation_space.n,
103         hidden_sizes=[40, 40],
104         activation=nn.Tanh,
105         device="cuda" if torch.cuda.is_available() else "cpu",
106     ).to("cuda" if torch.cuda.is_available() else "cpu")
107
108     actor = ActorProb(
109         a_net,
110         action_shape=action_space[player_name].shape or action_space[player_name].n,
111         max_action=max_action,
112         unbounded=True,
113         device="cuda" if torch.cuda.is_available() else "cpu",
114     ).to("cuda" if torch.cuda.is_available() else "cpu")
115
116     c_net = Net(
117         state_shape=observation_space[player_name].shape
118         or observation_space[player_name].n,
119         hidden_sizes=[40, 40],
120         activation=nn.Tanh,
121         device="cuda" if torch.cuda.is_available() else "cpu",
122     ).to("cuda" if torch.cuda.is_available() else "cpu")
123
124     critic = Critic(c_net, device="cuda" if torch.cuda.is_available() else "cpu").to(
125         "cuda" if torch.cuda.is_available() else "cpu"
126     )
127
128     torch.nn.init.constant_(actor.sigma_param, -0.5)
129     for m in list(actor.modules()) + list(critic.modules()):
130         if isinstance(m, torch.nn.Linear):
131             # orthogonal initialization
132             torch.nn.init.orthogonal_(m.weight, gain=np.sqrt(2))
133             torch.nn.init.zeros_(m.bias)
134
135     # do last policy layer scaling, this will make initial actions have (close to)
136     # 0 mean and std, and will help boost performances,
137     # see https://arxiv.org/abs/2006.05990, Fig.24 for details
138     for m in actor.mu.modules():
139         if isinstance(m, torch.nn.Linear):
140             torch.nn.init.zeros_(m.bias)
141             m.weight.data.copy_(0.01 * m.weight.data)
142
143     return (actor, critic)
144
145
146 def construct_policy(
147     actor: ActorProb, critic: Critic, args: argparse.Namespace
148 ) -> BasePolicy:
149     optim = torch.optim.Adam(
150         list(actor.parameters()) + list(critic.parameters()), lr=args.tianshou["lr"]
151     )
152
153     lr_scheduler = None
154     if args.tianshou["lr_decay"]:
155         # decay learning rate to 0 linearly
156         max_update_num = (
```

```
            np.ceil(args.tianshou["step_per_epoch"] / args.tianshou["step_per_collect"])
            * args.tianshou["epoch"]
        )

        _lr = _lr_scheduler(max_update_num)
        lr_scheduler = LambdaLR(optim, lr_lambda=_lr.lr)

    policy = MC_PPOPolicy(
        args,
        actor,
        critic,
        optim,
        dist,
        # discount_factor=args.gamma,
        # gae_lambda=args.gae_lambda,
        # max_grad_norm=args.max_grad_norm,
        # vf_coef=args.vf_coef,
        # ent_coef=args.ent_coef,
        # reward_normalization=args.rew_norm,
        action_scaling=True,
        # action_bound_method=args.bound_action_method,
        lr_scheduler=lr_scheduler,
        # action_space=env.action_space,
        # eps_clip=args.eps_clip,
        # value_clip=args.value_clip,
        # dual_clip=args.dual_clip,
        # advantage_normalization=args.norm_adv,
        # recompute_advantage=args.recompute_adv
    )

    return policy


########################################################
## Auxiliary Classes
########################################################
class Identify:
    def __init__(self, weights: FloatArray) -> None:
        self.weights = weights

    def sample(self) -> FloatArray:
        return self.weights


class _lr_scheduler:
    def __init__(self, max_update_num: float) -> None:
        self.max_update_num = max_update_num

    def lr(self, epoch: int) -> float:
        return 1 - epoch / self.max_update_num


########################################################
## Main Class
########################################################
## Add saving and loading using the MC checkpoint lanauge.
class MC_PPOPolicy(PPOPolicy):  # type: ignore[misc] # PPOPolicy is Any
    def __init__(
        self, init_args: argparse.Namespace, *args: Any, **kwargs: Any
    ) -> None:
        super().__init__(*args, **kwargs)
        self.init_args = init_args

    def checkpoint(self, folder: PathString) -> None:
        path = os.path.join(os.path.dirname(folder), "Model.pth")
        torch.save(self.state_dict(), path)

    def reload(self, folder: PathString) -> None:
        path = os.path.join(os.path.dirname(folder), "Model.pth")
        self.load_state_dict(torch.load(path))

    def get_action(
        self, obs: FloatArray, t: float = 0, mean_mode: bool = False
    ) -> FloatArray:
        # obs.info = dict()
        obs_bat = Batch(obs=obs.reshape(1, -1), info=dict())
        act_bat = self.forward(obs_bat)
        return act_bat.act.numpy().reshape(-1)  # type: ignore[no-any-return] # from base class

    def get_theta(self) -> FloatArray:
        theta = []
```

```
        for param in self.actor.parameters():
            theta.append(param.detach().numpy().reshape(-1))

        for param in self.critic.parameters():
            theta.append(param.detach().numpy().reshape(-1))

        return np.concatenate(theta)

    def __deepcopy__(self, memo: dict[int, Any]) -> BasePolicy:
        result = construct_policy(
            actor=copy.deepcopy(self.actor),
            critic=copy.deepcopy(self.critic),
            args=self.init_args,
        )
        memo[id(self)] = result

        return result
```

15 marco_polo/optimizers/uber_es

15.1 marco_polo/optimizers/uber_es/__init__.py

```
1  # Copyright (c) 2023 Mobius Logic, Inc.
2  #
3  # Licensed under the Apache License, Version 2.0 (the "License");
4  # you may not use this file except in compliance with the License.
5  # You may obtain a copy of the License at
6  #
7  #    http://www.apache.org/licenses/LICENSE-2.0
8  #
9  # Unless required by applicable law or agreed to in writing, software
10 # distributed under the License is distributed on an "AS IS" BASIS,
11 # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 # See the License for the specific language governing permissions and
13 # limitations under the License.
14
15
16 from .manager import get_opt_class
17 from .modules.model import get_model
```

15.2 marco_polo/optimizers/uber_es/manager.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Uber ES Optimization Manager"""

import json
import logging
import multiprocessing as mp
import os
import time
import argparse  # for type hinting
from collections import OrderedDict  # for type hinting
from collections.abc import Callable  # for type hinting
from multiprocessing.pool import AsyncResult  # for type hinting
from typing import Any, Optional, Type, Union  # for type hinting

import numpy as np
from numpy.random import PCG64DXSM, Generator

from marco_polo.optimizers.uber_es.modules.opt import ESTeamWrapper, StepStats
from marco_polo.optimizers.uber_es.modules.result_objects import EvalResult
from marco_polo.optimizers.uber_es.modules.rollouts import (
    run_eval_batch_distributed,
    run_optim_batch_distributed,
    run_viz_distributed,
)
from marco_polo.base_classes.serialCrew import SerialCrew
from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder, Telemetry, TBWriter
from marco_polo.tools.types import EnvId, Role, TeamId, PathString  # for type hinting
from marco_polo.tools.wrappers import EnvHistory, TeamHistory, RunTask

logger = logging.getLogger(__name__)


######################################################
## Factories
######################################################
def get_opt_class(
    args: argparse.Namespace,
) -> Union[Type["Manager"], Type["SerialManager"]]:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    args: argparse.Namespace
        arguments that were passed to the `main()` function

    Returns
    -------
    Union[Type["Manager"], Type["SerialManager"]]
        the class to use in creating env objects
    """
    if getattr(args, "parallel", True):
        return Manager
    return SerialManager


################################################################################
## Main Class
################################################################################
class Manager:
    """
    Compute Manager Class

    This class handles all compute by providing a standardized interface for
    optimization and evaluation. This manager is based on the multiprocessing
```

```
    asynchronous pool.
    """

    def __init__(self, setup_args: argparse.Namespace) -> None:
        """
        Manager Initializer

        Stores simulation args and sets up the compute pool

        Parameters
        ----------
        setup_args : args.Namespace
            Simulation parameters

        Side-Effects
        ------------
        None

        Returns
        -------
        None
        """

        self.setup_args = setup_args
        self.np_random = Generator(PCG64DXSM(seed=setup_args.master_seed))

        # initialize worker pool
        self.crew = self.get_crew()

        # dictionary to store optimizers between optimization chunks
        # key is (env.id, team.id)
        self.optimArchive: dict[tuple[EnvId, TeamId], ESTeamWrapper] = {}

        # Tensorboard:
        self.Telemetry = Telemetry()
        if hasattr(setup_args, "telemetry"):
            t_args = setup_args.telemetry
            if "Tensorboard" in t_args.keys():
                tb = TBWriter(t_args["Tensorboard"])
                self.Telemetry.add_logger(tb)

        # # Saving this as notes
        # #  This is how to share objects between workers in a parallel pool.
        # #  It still copies complex objects, so think of this more as a safety/convenience
        # #  idea than a performance idea.
        # # create manager for this context
        # manager = mp_ctx.Manager()
        # self.manager = manager
        # # This is pedantic
        # #  The recommendation is to keep all shared objects together for programmer
        # #  convenience, therefore, POET built a dict of them.
        # self.fiber_shared = {
        #         "envs": manager.dict(),
        #         "agents": manager.dict(),
        #         "tasks": manager.dict(),
        # }
        # # setup workers, provide reference to the shared objects
        # #  This works in tandem with a "global" call within initialize_worker_fiber
        # #  It's not the standard way of passing shared args to workers.
        # self.crew = mp_ctx.Pool(setup_args.num_workers, initializer=initialize_worker_fiber,
        #         initargs=(self.fiber_shared["agents"],
        #                   self.fiber_shared["envs"],
        #                   self.fiber_shared["tasks"],))

    ########################################
    ## Aux Funcs
    ########################################
    def get_crew(self) -> mp.pool.Pool:
        """Return a Pool of processes used to run tasks

        Parameters
        ----------
        None

        Side-Effects
        ------------
        None

        Returns
        -------
        mp.Pool
```

```
            Pool of workers to run tasks
        """
        # Get a specific context
        #  this avoids changing defaults in MP
        mp_ctx = mp.get_context("spawn")

        return mp_ctx.Pool(processes=self.setup_args.num_workers)

    def checkpoint(self, folder: PathString, cur_opts: list[RunTask]) -> None:
        """
        Store Objects to Disk

        Writes all objects to disk, calls the appropriate "checkpoint" functions
        for internal objects.

        Parameters
        ----------
        folder : PathString
            Directory to store objects
        cur_opts : list[RunTask]
            List of Tuples of currently active EnvHistory/TeamHistory pairs

        Side-Effects
        ------------
        Yes
            Cleans the optimizer archive before storing

        Returns
        -------
        None
            All objects are saved to disk
        """
        # clean dictionary
        #  Last-minute cleaning so we don't bloat the checkpoints
        #  This assume "cur_opts" is the current active set of env/team pairs
        #  If that's not true, this will fail
        self._clean_archive(cur_opts=cur_opts)

        tmp: dict[str, Any] = {"np_random": self.np_random.bit_generator.state}
        tmp["optimArchive"] = list(self.optimArchive.keys())

        # Manager directory
        folder = os.path.join(folder, "Manager")
        os.makedirs(folder, exist_ok=True)

        # store params
        with open(
            os.path.join(folder, "Manager.json"), mode="w", encoding="utf-8"
        ) as f:
            json.dump(tmp, f, cls=NumpyEncoder)

        # log optim archive
        for k, v in self.optimArchive.items():
            # setup/create directory
            opt_folder = os.path.join(folder, "".join(k))
            os.makedirs(opt_folder, exist_ok=True)
            # save optimizer
            v.checkpoint(folder=opt_folder)

    def reload(
        self,
        folder: PathString,
        cur_opts: OrderedDict[EnvHistory, TeamHistory],
    ) -> None:
        """
        Reload Objects from Disk

        Reads all objects to disk, calls the appropriate "Reload" functions
        for internal objects.

        Parameters
        ----------
        folder : PathString
            Directory to reload objects
        cur_opts : OrderedDict[EnvHistory, TeamHistory]
            Dictionary of active EnvHistory/TeamHistory pairs

        Side-Effects
        ------------
        Yes
            Creates all internal objects
```

```
241
242             Returns
243             -------
244             None
245             """
246             ## Adds some backwards compatability.
247             # Manager directory
248             if os.path.isdir(os.path.join(folder, "Manager")):
249                 folder = os.path.join(folder, "Manager")
250
251             # grab parameters
252             optArcList = []
253             with open(
254                 os.path.join(folder, "Manager.json"), mode="r", encoding="utf-8"
255             ) as f:
256                 dct = json.load(f, cls=NumpyDecoder)
257                 self.np_random.bit_generator.state = dct["np_random"]
258                 if "optimArchive" in dct.keys():
259                     optArcList = dct["optimArchive"]
260
261             # build optimizers
262             #  env.id was used for the archive, but the bracket uses the whole env
263             #  so building a dictionary to translate env.id to env
264             envIDDict = {k.id: k for k in cur_opts.keys()}
265             for kv in optArcList:
266                 # grab team from active env/team dict
267                 team = cur_opts[envIDDict[kv[0]]]
268                 # safety - these should match
269                 assert kv[1] == team.id, "Team ID does not match optimizer archive."
270
271                 # build tmp optimizer
272                 tmpOpt = ESTeamWrapper(args=self.setup_args, team=team)
273                 # reload
274                 tmpOpt.reload(folder=os.path.join(folder, "".join(kv)))
275                 # store
276                 self.optimArchive[(kv[0], kv[1])] = tmpOpt
277
278         # def evalupdate(self, env: EnvHistory, stats) -> None:
279         #     """
280         #     Update Environment with Evaluation Data
281
282         #     Calls the appropriate environment update function with evaluation statistics.
283
284         #     Parameters
285         #     ----------
286         #     env : Gym.environment
287         #         The simulation environment
288         #     stats : EvalResult
289         #         EvalResult object
290
291         #     Side-Effects
292         #     ------------
293         #     Possibly
294         #         If the environment has an update function, it is called and changes
295         #         things within each environment
296
297         #     Returns
298         #     -------
299         #     None
300         #     """
301         #     ## check if env has update function
302         #     if hasattr(env.__class__, "evalupdate") and callable(
303         #         getattr(env.__class__, "evalupdate")
304         #     ):
305         #         env.evalupdate(stats)
306
307         def optupdate(self, env: EnvHistory, stats: dict[Role, StepStats]) -> None:
308             """
309             Update Environment with Optimization Data
310
311             Calls the appropriate environment update function with optimization statistics.
312
313             Parameters
314             ----------
315             env : EnvHistory
316                 The simulation environment
317             stats : dict[Role, StepStats]
318                 Whatever is returned from the optimizer update
319
320             Side-Effects
321             ------------
```

```
322             Possibly
323                 If the environment has an update function, it is called and changes
324                 things within each environment
325 
326             Returns
327             -------
328             None
329             """
330             ## check if env has update function
331             if hasattr(env.__class__, "optupdate") and callable(
332                 getattr(env.__class__, "optupdate")
333             ):
334                 env.optupdate(stats)
335 
336         def _clean_archive(self, cur_opts: list[RunTask]) -> None:
337             """
338             Remove Old Entries From Optimizer Dictionary
339 
340             This function keeps the optimizer dictionary from growing without bounds.
341             It makes sure that only the active EnvHistory/TeamHistory pairings are maintained.
342 
343             Parameters
344             ----------
345             cur_opts : list[RunTask]
346                 List of tuples of current EnvHistory/TeamHistory pairings
347 
348             Side-Effects
349             ------------
350             Yes
351                 Cleans all items from optimArchive except those in cur_opts
352 
353             Returns
354             -------
355             None
356             """
357             # new archive object
358             newArchive = dict()
359 
360             # loop through things to save and add to dictionary
361             for key, val in cur_opts:
362                 new_key = (key.id, val.id)
363                 if new_key in self.optimArchive:
364                     newArchive[new_key] = self.optimArchive[new_key]
365 
366             # replace
367             self.optimArchive = newArchive
368 
369         ########################################
370         ## Eval Funcs
371         ########################################
372         def start_chunk(
373             self,
374             eval_func: Callable[..., Any],
375             num_jobs: int,
376             num_tasks_per_job: int,
377             **kwargs: Any,
378         ) -> list[AsyncResult[Any]]:
379             """
380             Multi-processing Handoff Function
381 
382             This function passes "eval_func" to the multiprocessing pool for asynchronous
383             evaluation. It sets up the appropriate seed, passes "num_job" copies of
384             "eval_func", and then passes arguments.
385 
386             Parameters
387             ----------
388             eval_func : Callable[..., Any]
389                 Function to run on pool
390             num_jobs : int
391                 Number of tasks to start
392             num_tasks_per_job : int
393                 Size of each job, generally refers to simulation rollouts
394             **kwargs : Any
395                 Additional args to pass to functios
396 
397             Side-Effects
398             ------------
399             None
400                 May write videos to disk, but all objects are copied and not returned,
401                 so no internal changes from these calls
402 
```

```
        Returns
        -------
        list[AsyncResult[Any]]
            List of async objects. Results of function can be gotten with asyc.get()
        """
        # debug logs
        logger.debug(f"Spawning {num_jobs} batches of size {num_tasks_per_job}")

        # pull seeds for rollouts
        rs_seeds = self.np_random.integers(
            low=2**31 - 1, size=num_jobs, dtype=np.int32
        )

        # return obj
        chunk_tasks = []

        # put tasks on pool
        #  append task handles to return obj
        for i in range(num_jobs):
            chunk_tasks.append(
                self.crew.apply_async(
                    func=eval_func, args=(num_tasks_per_job, rs_seeds[i]), kwds=kwargs
                )
            )

        return chunk_tasks

    def evaluate(
        self, tasks: list[RunTask], epoch: Optional[int] = None, verbose: bool = False
    ) -> list[dict[Role, EvalResult]]:
        """
        Main Evaluation Function

        This function runs all "tasks" provided. "epoch" and "verbose" are for
        logging purposes only. This function passes evaluation off to "start_chunk",
        then asynchronously gets results and processes them.

        Parameters
        ----------
        tasks : list[(RunTask]
            List of EnvHistory and the TeamHistory to run on them
        epoch : Optional[int], default=None
            Current simulation time, for logging only
        verbose : bool, default=False
            Log to console?

        Side-Effects
        ------------
        None
            All objects are copied and not returned, so no internal changes

        Returns
        -------
        list[dict[role, EvalResult]]
            List the same length as "tasks", with a dictionary holding the results
            of each agent on a team
        """
        # start time
        eval_time = time.time()

        # job handle list
        eval_handles = []

        # assign jobs
        for env, team in tasks:
            eval_handles.append(
                self.start_chunk(
                    # start_chunk args
                    eval_func=run_eval_batch_distributed,
                    num_jobs=self.setup_args.eval_jobs,
                    num_tasks_per_job=self.setup_args.rollouts_per_eval_job,
                    # eval_func args
                    env_params=env.env_param,
                    env_creator_func=env.get_env_creator(),
                    team=team,
                )
            )

        # return object
        combined_results = []

```

```
        # loop through tasks
        # for task in eval_results_done:
        for task_handles, (env, team) in zip(eval_handles, tasks):
            # get task returns for this task
            #  We have to get all of these 'cause then we cycle through the roles
            task = [job.get() for job in task_handles]

            # return obj
            combined = dict()

            # Loop through roles
            for role in team.roles():
                # return objects for this role
                returns_ = []
                lengths_ = []

                # loop through this role's results from all tasks
                for job in task:
                    returns_.extend(job[role].returns)
                    lengths_.extend(job[role].lengths)

                # reshape list as numpy object
                returns = np.array(returns_)
                lengths = np.array(lengths_)

                # create eval result for this role
                #  recompute metrics
                combined[role] = EvalResult(
                    returns=returns,
                    lengths=lengths,
                    eval_returns_mean=returns.mean(),
                    eval_returns_max=returns.max(),
                    eval_returns_min=returns.min(),
                )

            # All roles are finished, put into final return object
            combined_results.append(combined)

        # if verbose, log results to terminal
        if verbose:
            for task_result, (env, team) in zip(combined_results, tasks):
                for role, stats in task_result.items():
                    logger.info(
                        f"Epoch={epoch} - Environment={env.id} - "
                        f"Team={team.id} - Role={role} - "
                        f"Eval_mean={stats.eval_returns_mean} - "
                        f"Best_eval={stats.eval_returns_max} - "
                        f"Lengths={stats.lengths} - "
                        f"Iterations_lifetime={env.stats.iterations_lifetime} - "
                        f"Iterations_current={env.stats.iterations_current}"
                    )

        # log
        logger.debug(
            f"evaluate() Time to complete evaluation: {time.time() - eval_time}"
        )

        # return results list
        return combined_results

    def _visualize(self, tasks: list[RunTask], epoch: Optional[int] = None) -> None:
        """
        Hidden Vizualization Function

        This function is specifically for simulation vizualization. It is a blocking
        function with no return. It runs all "tasks" and outputs .gifs for each.
        This function is used in "optimize_chunk()"

        Parameters
        ----------
        tasks : list[RunTask]
            List of envHistory and the TeamHistory to run on them
        epoch : Optional[int], default=None
            Current simulation time, for logging only

        Side-Effects
        ------------
        None
            All objects are copied and not returned, so no internal changes

        Returns
```

```
        -------
        None
        """
        # pass off tasks
        vis_handles = []
        for env, team in tasks:
            vis_handles.extend(
                self.start_chunk(
                    # start_chunk args
                    eval_func=run_viz_distributed,
                    num_jobs=1,
                    num_tasks_per_job=1,
                    # eval_func args
                    env_params=env.env_param,
                    env_creator_func=env.get_env_creator(),
                    team=team,
                    env_id=env.id,
                    epoch=epoch,
                    vidpath=self.setup_args.vidpath,
                    frame_skip=self.setup_args.frame_skip,
                )
            )

        # Wait for these to complete, or else the program can end before saving images
        _ = [job.get() for job in vis_handles]

    ########################################
    ## Optim Funcs
    ########################################
    def optimize_step(
        self,
        tasks: list[RunTask],
        epoch: Optional[int] = None,
        opt_iter: Optional[int] = None,
    ) -> list[dict[Role, StepStats]]:
        """
        Single Optimization Step

        This function performs a single optimization step for each active environment
        and team pair in "tasks". "epoch" and "opt_iter" are for logging only.
        This function passes work off to "start_chunk(), then grabs results
        asynchronously and processes them as they come in.

        Parameters
        ----------
        tasks : list[(RunTask]
            List of tuple of EnvHistory/TeamHistory pairs
        epoch : Optional[int], default=None
            Current simulation time
        opt_iter : Optional[int], default=None
            Which iteration in a loop we are on

        Side-Effects
        ------------
        Yes
            This function updates environments with the optimization stats, and
            the optimization updates the agents.

        Returns
        -------
        list[dict[Role, StepStats]]
            Returns a list of length "tasks" with statistics calculated in "combine_and_update()"
        """
        # setup new shared list
        job_handles = []
        start_times = []
        ret_list = []

        # loop over tasks
        for env, team in tasks:
            # check if optimizer already exists for task
            if (env.id, team.id) not in self.optimArchive:
                self.optimArchive[(env.id, team.id)] = ESTeamWrapper(
                    args=self.setup_args, team=team
                )

            # grab optimizer to continue task
            es_team = self.optimArchive[(env.id, team.id)]

            # pass off work
            job_handles.append(
```

```
                self.start_chunk(
                    # start_chunk args
                    eval_func=run_optim_batch_distributed,
                    num_jobs=self.setup_args.uber_es["optim_jobs"],
                    num_tasks_per_job=self.setup_args.uber_es["rollouts_per_optim_job"],
                    # eval_func args
                    env_params=env.env_param,
                    env_creator_func=env.get_env_creator(),
                    team=es_team,
                    setup_args=self.setup_args,
                    noise_std=self.setup_args.uber_es["noise_std"],
                )
            )
            # append start time
            start_times.append(time.time())

        # loop over job handles, analyze
        for sT, jH, (env, team) in zip(start_times, job_handles, tasks):
            # get results for this env/team optim
            task_results = [handle.get() for handle in jH]

            # grab optimizer
            es_team = self.optimArchive[(env.id, team.id)]

            # update optimizers
            stepstats = es_team.combine_and_update(
                step_results=task_results,
                step_t_start=sT,
                decay_noise=True,
                propose_only=False,
            )

            # update env
            self.optupdate(env=env, stats=stepstats)
            env.stats.iterations_lifetime += 1
            env.stats.iterations_current += 1

            # append results
            ret_list.append(stepstats)

        # return all results
        return ret_list

    def optimize_chunk(
        self, tasks: list[RunTask], epoch: Optional[int] = None, verbose: bool = False
    ) -> list[dict[Role, list[float]]]:
        """
        Multiple Optimization Step, Evaluation, and Video Generation

        This function performs several optimization steps, as well as evaluation
        of the optimized teams after each optimization. When appropriate, it
        generates videos of agent performance as well.

        Parameters
        ----------
        tasks : list[RunTask]
            List of tuples of EnvHistory/TeamHistory pairs
        epoch : Optional[int], default=None
            Current simulation time
        verbose : bool, default=False
            Log results to console?

        Side-Effects
        ------------
        Yes
            This function calls "optimize_step()", which updates environments with
            the optimization stats, and the optimization updates the agents.

        Returns
        -------
        list[dict[Role, list[float]]]
            Returns a list of length "tasks" with evaluation means for each team
        """
        # Start time
        opt_time = time.time()

        # list of dict{key: list} for each agent Role in TeamHistory
        returns: list[dict[Role, list[float]]] = [
            {role: [] for role in team.roles()} for (_, team) in tasks
        ]

```

```
        # loop over optimization iterations
        for i in range(self.setup_args.uber_es["optim_iters"]):
            # log iter
            logger.info(f"optim_iter {i+1} of {self.setup_args.uber_es['optim_iters']}")

            # single optimization step
            _ = self.optimize_step(tasks=tasks, epoch=epoch, opt_iter=i)

            # eval for one opt
            eval_results = self.evaluate(tasks=tasks, epoch=epoch, verbose=verbose)

            for j, (env, team) in enumerate(tasks):
                for role in team.roles():
                    returns[j][role].append(eval_results[j][role].eval_returns_mean)

                self.Telemetry.write_items(
                    name=f"{env.id}:{team.id}:Scores",
                    data=returns[j],
                    step=epoch * self.setup_args.uber_es["optim_iters"] + i,
                )

                # TODO: Does this actually work?
                #       Seems like "role" will just be the last agent in the team
                L = eval_results[j][role].lengths
                self.Telemetry.write_items(
                    name=f"{env.id}:{team.id}:Lengths",
                    data={
                        "mean": np.mean(L),
                        "+std": np.mean(L) + np.std(L),
                        "-std": np.mean(L) - np.std(L),
                        "max": np.max(L),
                        "min": np.min(L),
                    },
                    step=epoch * self.setup_args.uber_es["optim_iters"] + i,
                )

        # check for visualization
        if (
            self.setup_args.visualize_freq > 0
            and epoch % self.setup_args.visualize_freq == 0
        ):
            self._visualize(tasks=tasks, epoch=epoch)

        # clean dictionary
        #  this isn't strictly necessary here, it just needs to run regularly
        #  Doing it here keeps it within the Manager object
        #  This assume "tasks" is the current active set of env/team pairs
        #  If that's not true, this will fail
        self._clean_archive(cur_opts=tasks)

        # Log total time
        logger.debug(f"optimize_chunk() Full Optstep Time: {time.time() - opt_time}")

        # return list of dict of list of mean eval result
        return returns


################################################################################
## Secondary Class
################################################################################
class SerialManager(SerialCrew, Manager):
    """Manager that uses a single process instead of multiprocessing

    This should be a drop in replacement for the Manager class.

    This is functionally different than using a Manager with one
    worker in that the async calls are direct calls to the specified
    function so debugging and profiling will work as expected.

    Generally this would only be used for testing.

    The class is intentionally blank. Using multi-inheritance, it has
    all the methods of the Manager class, except that it uses the
    get_crew() method from the SerialCrew class.
    """
```

16 marco_polo/optimizers/uber_es/modules

16.1 marco_polo/optimizers/uber_es/modules/model.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


"""Uber ES Basic Agent Model"""

import argparse  # for type hinting
import json
import logging
from typing import Any, cast, Optional, Union  # for type hinting

import numpy as np
from numpy.random import PCG64DXSM, Generator

from pettingzoo.utils.env import ParallelEnv  # type: ignore [import]

from marco_polo.base_classes.model import BaseModel
from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder
from marco_polo.tools.types import (
    ActivationType,
    FloatArray,
    Role,
    PathString,
)  # for type hinting

logger = logging.getLogger(__name__)


################################################################################
## Agent Factory
################################################################################
def get_model(
    env: ParallelEnv, role: Role, args: argparse.Namespace, seed: int
) -> BaseModel:
    """Returns the class to use, based on input arguments

    Parameters
    ----------
    env: pettingzoo.util.env.ParallelEnv
        A parallel env that holds to pettingzoos's interface, not used here
    role: Role
        Agent role in env, not used here
    args: argparse.Namespace
        arguments that were passed to the `main()` function
    seed: int
        seed for model creation

    Returns
    -------
    BaseModel
        The model
    """
    return Model(model_params=args.model_params, seed=seed)


################################################################################
## Aux Funcs
################################################################################
# possible activation functions for each layer of the NN
def sigmoid(x: ActivationType) -> ActivationType:
```

```
    return 1 / (1 + np.exp(-x))


def relu(x: ActivationType) -> ActivationType:
    return np.maximum(x, 0)


def passthru(x: ActivationType) -> ActivationType:
    return x


def softmax(x: ActivationType) -> ActivationType:
    arg: ActivationType = x - np.max(x)
    e_x = np.exp(arg)
    return e_x / cast(float, e_x.sum(axis=0))  # cast is to help mypy


################################################################################
## Model Class
################################################################################
class Model(BaseModel):
    """simple feedforward model"""

    def __init__(
        self,
        model_params: dict[str, Any],
        seed: int,
        args: Optional[argparse.Namespace] = None,
    ) -> None:
        """Constructor for Uber ES Model

        Much of this construction came from Uber's original POET implementation.
        Primary additions include documentation and model checkpointing.

        Parameters
        ----------
        model_params : dict[str, Any]
            Dictionary defining model structure
        seed : int
            Initial value for internal PRNG
        args : Optional[argparse.Namespace], default=None
            Values to parameters the rest of the model

        Side-Effects
        ------------
        Yes
            Sets all class attributes

        Returns
        -------
        None
        """
        self.args = args
        self.np_random = Generator(PCG64DXSM(seed=seed))

        self.output_noise = model_params["output_noise"]

        self.rnn_mode = False  # in the future will be useful
        self.time_input = 0  # use extra sinusoid input
        self.sigma_bias = model_params["noise_bias"]  # bias in stdev of output
        self.sigma_factor = 0.5  # multiplicative in stdev of output
        if model_params["time_factor"] > 0:
            self.time_factor = float(model_params["time_factor"])
            self.time_input = 1
        self.input_size = model_params["input_size"]
        self.output_size = model_params["output_size"]

        self.shapes = [(self.input_size + self.time_input, model_params["layers"][0])]
        self.shapes += [
            (model_params["layers"][i], model_params["layers"][i + 1])
            for i in range(len(model_params["layers"]) - 1)
        ]
        self.shapes += [(model_params["layers"][-1], self.output_size)]

        self.sample_output = model_params["sample_output"]

        if len(model_params["activations"]) != (len(model_params["layers"]) + 1):
            raise ValueError(
                "The model activations need to be 1 function greater \
                than the number of layers"
            )
```

```
157
158         self.activations = [eval(x) for x in model_params["activations"]]
159
160         self.actor_weight = []
161         self.actor_bias = []
162         self.actor_bias_log_std = []
163         self.actor_bias_std = []
164         self.actor_param_count = 0
165
166         idx = 0
167         for shape in self.shapes:
168             self.actor_weight.append(np.zeros(shape=shape))
169             self.actor_bias.append(np.zeros(shape=shape[1]))
170             self.actor_param_count += np.product(shape) + shape[1]
171             if self.output_noise[idx]:
172                 self.actor_param_count += shape[1]
173             log_std = np.zeros(shape=shape[1])
174             self.actor_bias_log_std.append(log_std)
175             out_std = np.exp(self.sigma_factor * log_std + self.sigma_bias)
176             self.actor_bias_std.append(out_std)
177             idx += 1
178
179         # initialize action scale
180         self.scale = np.ones(self.output_size)
181         # declare if properly specified
182         if model_params.get("action_scale", False):
183             self.scale = np.array(model_params["action_scale"])
184             # self.scale = np.asarray(a=model_params["action_scale"], dtype=float)
185
186         self.render_mode = False
187         self.theta = self.get_random_model_params()
188
189     # aux getters/setters
190     #  These are incredibly important due to the use of a wrapper class
191     def __repr__(self) -> str:
192         return "{}".format(self.__dict__)
193
194     def __getstate__(self) -> dict[str, Any]:
195         return self.__dict__
196
197     def __setstate__(self, state: dict[str, Any]) -> None:
198         self.__dict__ = state
199
200     def get_action(
201         self, x: ActivationType, t: float = 0, mean_mode: bool = False
202     ) -> Union[np.int_, FloatArray]:
203         """Generate Model Action
204
205         Parameters
206         ----------
207         x : FloatArray
208             Input observations
209         t : float, default=0
210             Time scale
211         mean_mode : bool, default=False
212             Generate noise in the network and maintain the average?
213
214         Side-Effects
215         ------------
216         Yes
217             Uses internal PRNG
218
219         Returns
220         -------
221         Union[np.int_, FloatArray]
222             Agent action
223         """
224         # if mean_mode = True, ignore sampling.
225         h = np.array(x).flatten()
226         if self.time_input == 1:
227             time_signal = float(t) / self.time_factor
228             h = np.concatenate([h, [time_signal]])
229         num_layers = len(self.actor_weight)
230         for i in range(num_layers):
231             w = self.actor_weight[i]
232             b = self.actor_bias[i]
233             h = np.matmul(h, w) + b
234             if self.output_noise[i] and (not mean_mode):
235                 out_size = self.shapes[i][1]
236                 out_std = self.actor_bias_std[i]
237                 output_noise = self.np_random.standard_normal(size=out_size) * out_std
```

```
                h += output_noise
            h = self.activations[i](h)

        if self.sample_output:
            return np.argmax(self.np_random.multinomial(n=1, pvals=h, size=1))

        return h * self.scale

    def set_model_params(self, model_params: FloatArray) -> None:
        pointer = 0
        for i in range(len(self.shapes)):  # pylint: disable=consider-using-enumerate
            w_shape = self.shapes[i]
            b_shape = self.shapes[i][1]
            s_w = np.product(w_shape)
            s = s_w + b_shape
            chunk = np.array(model_params[pointer : pointer + s])
            self.actor_weight[i] = chunk[:s_w].reshape(w_shape)
            self.actor_bias[i] = chunk[s_w:].reshape(b_shape)
            pointer += s
            if self.output_noise[i]:
                s = b_shape
                self.actor_bias_log_std[i] = np.array(
                    model_params[pointer : pointer + s]
                )
                self.actor_bias_std[i] = np.exp(
                    self.sigma_factor * self.actor_bias_log_std[i] + self.sigma_bias
                )
                if self.render_mode:
                    logger.info(f"bias_std: {self.actor_bias_std[i]}, layer: {i}")
                pointer += s

    def reload(self, filename: PathString) -> None:
        with open(filename, mode="r", encoding="utf-8") as f:
            data = json.load(f, cls=NumpyDecoder)
        self.theta = data["theta"]
        self.np_random.bit_generator.state = data["np_random"]
        self.set_model_params(model_params=self.theta)

    def checkpoint(self, filename: PathString) -> None:
        manifest = {
            "theta": self.theta,
            "np_random": self.np_random.bit_generator.state,
        }
        with open(filename, mode="w", encoding="utf-8") as f:
            json.dump(manifest, f, cls=NumpyEncoder)

    def shift_weights(self, noise_std: float, seed: int) -> None:
        # set random state
        random_state = Generator(PCG64DXSM(seed=seed))
        # draw uniform, shift and scale
        # BUG: This is supposed to be a normal distribution
        #      However, it performs better this way, so we leave it
        #      Should read: noise_std * random_state.standard_normal(size=self.actor_param_count)
        t = 2 * noise_std * (random_state.random(self.actor_param_count) - 0.5)
        # update local theta
        theta = self.theta + t
        # set model parameters to this theeta
        #  NOTE: it no longer matches the model theta!
        self.set_model_params(theta)

    def update_theta(self, new_theta: FloatArray) -> None:
        self.theta = new_theta

    def get_theta(self) -> FloatArray:
        return self.theta

    def get_random_model_params(self, stdev: float = 0.1) -> FloatArray:
        return self.np_random.normal(scale=stdev, size=self.actor_param_count)

    def get_zeroed_model_params(self) -> FloatArray:
        return np.zeros(self.actor_param_count)
```

16.2 marco_polo/optimizers/uber_es/modules/opt.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


"""Uber ES Team Wrapper and Optimization Controller"""


import json
import logging
import os
import time
from collections import namedtuple
from typing import Any, cast  # for type hinting
import argparse  # for type hinting

import numpy as np
from numpy.random import PCG64DXSM, Generator

from marco_polo.optimizers.uber_es.modules.optimizers import Adam
from marco_polo.optimizers.uber_es.modules.result_objects import POResult
from marco_polo.tools.stats import compute_CS_ranks, compute_weighted_sum
from marco_polo.tools.types import FloatArray, Role, PathString
from marco_polo.tools.wrappers import TeamHistory, AgtHistory

logger = logging.getLogger(__name__)


################################################################################
## Aux Objects
################################################################################
StepStats = namedtuple(
    "StepStats",
    [
        "po_returns_mean",
        "po_returns_median",
        "po_returns_std",
        "po_returns_max",
        "po_theta_max",
        "po_returns_min",
        "po_len_mean",
        "po_len_std",
        "noise_std",
        "learning_rate",
        "theta_norm",
        "grad_norm",
        "update_ratio",
        "episodes_this_step",
        "timesteps_this_step",
        "time_elapsed_this_step",
    ],
)


################################################################################
## Optimization Controller
################################################################################
class ESTeamWrapper:
    """Wrapper for the optimizers that apply to a given team

    By default, the neural network models of all agents are updated by
    the methods in this class. However, a set of roles can be marked
    as 'frozen' to prevent the agents from being updated. This can be
    done via the parameter file using the key uber_es.freeze.
```

```
    Alternatively, the list of frozen roles can be changed later
    (see add_freeze(), remove_freeze(), and remove_all_freeze())
    """

    def __init__(self, args: argparse.Namespace, team: TeamHistory) -> None:
        self.team = team
        self.args = args

        # see if freeze key is in args
        try:
            self._freeze = set(args.uber_es["freeze"])
        except (AttributeError, KeyError):
            self._freeze = set()

        # build optimizer dict
        self.optimizers = {}
        for role in team.roles():
            self.optimizers[role] = OptimizationController(args=args, agent=team[role])

    def __getattr__(self, name: str) -> Any:
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
        if name.startswith("_"):
            raise AttributeError(f"accessing private attribute '{name}' is prohibited")
        return getattr(self.team, name)

    def __len__(self) -> int:
        return len(self.team)

    def get_freeze(self) -> list[Role]:
        """Return list of frozen agent roles

        Parameters
        ----------
        None

        Returns
        -------
        list[Role]
            copy of the list of roles
        """
        return [cast(Role, i) for i in self._freeze]

    def remove_freeze(self, role: Role) -> None:
        """Remove role from set of frozen agent roles

        Parameters
        ----------
        role : Role
            role to remove

        Raises
        ------
        KeyError:
            If role is not contained in the set of frozen roles.
        """
        self._freeze.remove(role)

    def add_freeze(self, role: Role) -> None:
        """Adds role to set of frozen agent roles

        If the role already exists, this has no effect.

        Parameters
        ----------
        role : Role
            role to add

        Returns
        -------
        None
        """
        self._freeze.add(role)

    def remove_all_freeze(self) -> None:
        """Remove all roles from set of frozen agent roles"""
        self._freeze.clear()

    def checkpoint(self, folder: str) -> None:
        """Store Objects to Disk

        This function stores the optimizers to disk. It does not store the agents
```

```
        or the args - those get stored elsewhere and can be reloaded.
        Additionally, the set of roles to freeze is stored.

        Parameters
        ----------
        folder : str
            Directory to store objects

        Side-Effects
        ------------
        None

        Returns
        -------
        None
            All objects are saved to disk
        """
        # convert freeze to list as a set is not JSON serializable
        obj_data = {"_freeze": self.get_freeze()}
        file_path = os.path.join(folder, "es_team_wrapper.json")
        with open(file_path, mode="w", encoding="utf-8") as file:
            json.dump(obj_data, file)

        # loop through agent optimizers, store by agent role
        for role, optimizer in self.optimizers.items():
            optimizer.checkpoint(folder=folder, role=role)

    def reload(self, folder: str) -> None:
        """Reload Objects from Disk

        Reloads the freeze set and the optimizers.

        Parameters
        ----------
        folder : str
            Directory to reload objects

        Side-Effects
        ------------
        All
            Creates all internal objects

        Returns
        -------
        None
        """
        file_path = os.path.join(folder, "es_team_wrapper.json")
        try:
            with open(file_path, mode="r", encoding="utf-8") as file:
                json_dict = json.load(file)
                self._freeze = set(json_dict["_freeze"])
        except FileNotFoundError:
            pass  # backwards compatibility for before freeze was added/saved

        # the optimizer dictionary is created on initialization
        #  this goes through and reset parameters
        for role, optimizer in self.optimizers.items():
            optimizer.reload(folder=folder, role=role)

    def shift_weights(self, noise_std: float, seed: int) -> dict[Role, int]:
        """Randomly shift the weights of the agents

        Values will not be changed for any role listed in this object's
        freeze set.

        Parameters
        ----------
        noise_std: float
            scaling factor used for random value adjustments
        seed: int
            random number seed used for generating each agent's seed

        Returns
        -------
        dict[Role, int]
            seeds for each role (calculated from the seed that was given)
        """
        random_state = Generator(PCG64DXSM(seed=seed))
        seeds = dict()
        for role, agent in self.team.items():
            seed = random_state.integers(low=0, high=2**63)
```

```
            if role not in self._freeze:
                agent.shift_weights(noise_std=noise_std, seed=seed)
            seeds[role] = seed
        return seeds

    def combine_and_update(
        self,
        step_results: list[dict[Role, POResult]],
        step_t_start: float,
        decay_noise: bool = True,
        propose_only: bool = False,
    ) -> dict[Role, StepStats]:
        """Combine results from rollouts, pass to the optimizer, and update models.

        Agents with roles that are listed in the freeze set will have the statistics
        calculated but will not have their models updated.

        Parameters
        ----------
        step_results : list[dict[Role, POResult]]
            Results from runs
        step_t_start : float
            Time when this set of steps started (used for calculating runtime)
        decay_noise : bool, default=True
            Whether to have the applied noise decay during the optimization
        propose_only : bool, default=False
            If true, only the stats will be calculated and no updates will be made

        Return
        ------
        dict[Role, StepStats]
            Statistics from the runs for each role
        """
        # return dict
        #  organize by agent key
        stats = dict()

        # loop over agents, update their network
        for role in self.team.roles():
            # grab sim results for this agent
            results = [r[role] for r in step_results]

            # freeze is implemented by using propose_only
            if role in self._freeze:
                _propose_only = True
            else:
                _propose_only = propose_only

            # calculate stats and new theta
            new_theta, stats[role] = self.optimizers[role].combine_steps(
                step_results=results,
                theta=self.team[role].theta,
                step_t_start=step_t_start,
                decay_noise=decay_noise,
                propose_only=_propose_only,
            )

            # update agent
            if not _propose_only:
                self.team[role].update_theta(new_theta=new_theta)
                self.team[role].set_model_params(model_params=new_theta)

        # return results
        return stats


################################################################################
## what is this? Nate, name this plz.
################################################################################
class OptimizationController:
    """
    OptimizationController

    This class maintains all of the Adam learning parameters for each model.
    """

    def __init__(self, args: argparse.Namespace, agent: AgtHistory) -> None:
        # Meta options
        self.l2_coeff = args.uber_es["l2_coeff"]
        self.returns_normalization = args.uber_es["returns_normalization"]
        self.normalize_grads_by_noise_std = args.uber_es["normalize_grads_by_noise_std"]
```

```
322
323         # learning rate
324         self.learning_rate = args.uber_es["learning_rate"]
325         self.lr_decay = args.uber_es["lr_decay"]
326         self.lr_limit = args.uber_es["lr_limit"]
327
328         # noise
329         self.noise_std = args.uber_es["noise_std"]
330         self.noise_decay = args.uber_es["noise_decay"]
331         self.noise_limit = args.uber_es["noise_limit"]
332         self.init_noise_std = self.noise_std
333
334         # setup optimizer
335         self.optimizer = Adam(
336             theta=agent.get_zeroed_model_params(), stepsize=self.learning_rate
337         )
338
339     def __str__(self) -> str:
340         return "Optimization Controller"
341
342     def checkpoint(self, folder: PathString, role: Role) -> None:
343         json_dict = self.__dict__.copy()
344         del json_dict["optimizer"]
345
346         file_path = os.path.join(folder, f"{role}_opt.json")
347         with open(file_path, mode="w", encoding="utf-8") as f:
348             json.dump(json_dict, f)
349
350         self.optimizer.checkpoint(folder=folder, role=role)
351
352     def reload(self, folder: PathString, role: Role) -> None:
353         file_path = os.path.join(folder, f"{role}_opt.json")
354         with open(file_path, mode="r", encoding="utf-8") as f:
355             json_dict = json.load(f)
356             for k, v in json_dict.items():
357                 self.__dict__[k] = v
358
359         self.optimizer.reload(folder=folder, role=role)
360
361     def reset_optimizers(self) -> None:
362         self.optimizer.reset()
363         self.noise_std = self.init_noise_std
364
365     def get_noise(self, seed: int, theta_len: int) -> FloatArray:
366         random_state = Generator(PCG64DXSM(seed=seed))
367         # BUG: This is supposed to be a normal distribution
368         #      However, it performs better, so we leave it.
369         #      Should read: random_state(scale=noise_std, size=self.actor_param_count)
370         return 2 * (random_state.random(theta_len) - 0.5)
371
372     def compute_grads(
373         self, noise_seeds: FloatArray, returns: FloatArray, theta: FloatArray
374     ) -> tuple[FloatArray, FloatArray]:
375         """Computes and returns gradients for the thetas based on run results
376
377         Function takes the run results, normalized them, and computes the weighted sum of the
378         noise vectors to construct the gradient G = sum_n E(theta + epsilon)epsilon.
379         It also returns the theta of the largest return value.
380
381         Parameters
382         ----------
383         noise_seeds : FloatArray
384             Seed for each theta estimate
385         returns : FloatArray
386             Returns from the simulations
387         theta : FloatArray
388             Current network parameterization
389
390         Returns
391         -------
392         tuple[FloatArray, FloatArray]
393             Gradient vector for update, and theta estimate with maximum reward
394         """
395
396         theta_len = len(theta)
397         pos_row, neg_row = returns.argmax(axis=0)
398         noise_sign = 1.0
399         po_noise_seed_max = noise_seeds[pos_row]
400
401         if returns[pos_row, 0] < returns[neg_row, 1]:
402             noise_sign = -1.0
```

```
            po_noise_seed_max = noise_seeds[neg_row]

        # BUG: This is supposed to be a normal distribution
        #      Be careful if get_noise() is updated, the scale here will be off
        po_theta_max = theta + noise_sign * self.noise_std * self.get_noise(
            po_noise_seed_max, theta_len
        )

        if self.returns_normalization == "centered_ranks":
            proc_returns = compute_CS_ranks(returns)
        elif self.returns_normalization == "normal":
            proc_returns = (returns - returns.mean()) / (returns.std() + 1e-5)
        else:
            raise NotImplementedError(
                "Invalid return normalization `{}`".format(self.returns_normalization)
            )

        # BUG: This is supposed to be a normal distribution
        #      This is currently incorrect, but if get_noise() is fixed, it will
        #      be fine.
        grads, _ = compute_weighted_sum(
            weights=proc_returns[:, 0] - proc_returns[:, 1],
            vec_generator=(self.get_noise(seed, theta_len) for seed in noise_seeds),
            theta_size=theta_len,
        )

        # NOTE: I don't think this is correct - JB
        #       returns is an array, either 1D (positive rollouts only)
        #       or 2D (positive and negative). But, we combine those into a single
        #       weighted sum, so we only use len(returns)/2 values.
        grads /= len(returns)
        if self.normalize_grads_by_noise_std:
            grads /= self.noise_std
        return grads, po_theta_max

    def combine_steps(
        self,
        step_results: list[POResult],
        theta: FloatArray,
        step_t_start: float,
        decay_noise: bool = True,
        propose_only: bool = False,
    ) -> tuple[FloatArray, StepStats]:
        # Extract
        nList = []
        rList = []
        lList = []
        for r in step_results:
            nList.append(r.noise_inds)
            rList.append(r.returns)
            lList.append(r.lengths)

        # Reshape results
        noise_inds = np.concatenate(nList)
        po_returns = np.concatenate(rList)
        po_lengths = np.concatenate(lList)

        # Calculate gradients
        grads, po_theta_max = self.compute_grads(
            noise_seeds=noise_inds, returns=po_returns, theta=theta
        )

        # update
        if not propose_only:
            update_ratio, theta = self.optimizer.update(
                theta, -grads + self.l2_coeff * theta
            )

            self.optimizer.stepsize = max(
                self.optimizer.stepsize * self.lr_decay, self.lr_limit
            )

            if decay_noise:
                self.noise_std = max(
                    self.noise_std * self.noise_decay, self.noise_limit
                )

        else:  # only make proposal
            update_ratio = 0.0
            # leave theta alone

```

```
        step_t_end = time.time()

        # return new theta and stats
        return theta, StepStats(
            po_returns_mean=po_returns.mean(),
            po_returns_median=np.median(po_returns),
            po_returns_std=po_returns.std(),
            po_returns_max=po_returns.max(),
            po_theta_max=po_theta_max,
            po_returns_min=po_returns.min(),
            po_len_mean=po_lengths.mean(),
            po_len_std=po_lengths.std(),
            noise_std=self.noise_std,
            learning_rate=self.optimizer.stepsize,
            theta_norm=np.square(theta).sum(),
            grad_norm=float(np.square(grads).sum()),
            update_ratio=float(update_ratio),
            episodes_this_step=len(po_returns),
            timesteps_this_step=po_lengths.sum(),
            time_elapsed_this_step=step_t_end - step_t_start,
        )
```

16.3 marco_polo/optimizers/uber_es/modules/optimizers.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


import json
import os
from collections.abc import Sized
from typing import Any

import numpy as np

from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder
from marco_polo.tools.types import FloatArray, Role, PathString  # for type hinting


################################################################################
## Parent Optimizer
################################################################################
class Optimizer(object):
    def __init__(self, theta: Sized) -> None:
        self.dim = len(theta)
        self.t = 0

    def update(
        self, theta: FloatArray, globalg: FloatArray
    ) -> tuple[np.float_, FloatArray]:
        self.t += 1
        step = self._compute_step(globalg)
        ratio = np.linalg.norm(step) / np.linalg.norm(theta)
        return ratio, theta + step

    def _compute_step(self, globalg: FloatArray) -> FloatArray:
        raise NotImplementedError

    def checkpoint(self, folder: PathString) -> None:
        raise NotImplementedError

    def reload(self, folder: PathString) -> None:
        raise NotImplementedError


################################################################################
## Simple Stochastic Gradient Descent
################################################################################
class SimpleSGD(Optimizer):
    def __init__(self, stepsize: float) -> None:
        self.stepsize = stepsize

    def compute(
        self, theta: FloatArray, globalg: FloatArray
    ) -> tuple[np.float_, FloatArray]:
        step = -self.stepsize * globalg
        ratio = np.linalg.norm(step) / np.linalg.norm(theta)
        return ratio, theta + step

    # why is there no step method?
    # what is this compute method, if not mostly the step method?


################################################################################
## Full Stochastic Gradient Descent
################################################################################
class SGD(Optimizer):
```

```
    def __init__(
        self, theta: FloatArray, stepsize: float, momentum: float = 0.9
    ) -> None:
        # parent constructor
        Optimizer.__init__(self, theta)

        # set vars
        self.v = np.zeros(self.dim, dtype=np.float32)
        self.stepsize, self.momentum = stepsize, momentum

    def _compute_step(self, globalg: FloatArray) -> FloatArray:
        self.v = self.momentum * self.v + (1.0 - self.momentum) * globalg
        step = -self.stepsize * self.v
        return step

    def checkpoint(self, folder: PathString) -> None:
        file_path = os.path.join(folder, "_SGD.json")
        with open(file_path, mode="w", encoding="utf-8") as f:
            json.dump(self.__dict__, f, cls=NumpyEncoder)

    def reload(self, folder: PathString) -> None:
        file_path = os.path.join(folder, "_SGD.json")
        with open(file_path, mode="r", encoding="utf-8") as f:
            self.__dict__ = json.load(f, cls=NumpyDecoder)


################################################################################
## Adam
################################################################################
class Adam(Optimizer):
    def __init__(
        self,
        theta: FloatArray,
        stepsize: float,
        beta1: float = 0.9,
        beta2: float = 0.999,
        epsilon: float = 1e-08,
    ) -> None:
        # parent constructor
        super().__init__(theta)

        # vars
        self.stepsize = stepsize
        self.init_stepsize = stepsize
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = np.zeros(self.dim, dtype=np.float32)
        self.v = np.zeros(self.dim, dtype=np.float32)

    def checkpoint(self, folder: PathString, role: Role = Role("")) -> None:
        file_path = os.path.join(folder, f"{role}_Adam.json")
        with open(file_path, mode="w", encoding="utf-8") as file:
            json.dump(self.__dict__, file, cls=NumpyEncoder)

    def reload(self, folder: PathString, role: Role = Role("")) -> None:
        file_path = os.path.join(folder, f"{role}_Adam.json")
        with open(file_path, mode="r", encoding="utf-8") as file:
            self.__dict__ = json.load(file, cls=NumpyDecoder)

    def reset(self) -> None:
        self.m = np.zeros(self.dim, dtype=np.float32)
        self.v = np.zeros(self.dim, dtype=np.float32)
        self.t = 0
        self.stepsize = self.init_stepsize

    def _compute_step(self, globalg: FloatArray) -> FloatArray:
        a: float = (
            self.stepsize
            * np.sqrt(1 - self.beta2**self.t)
            / (1 - self.beta1**self.t)
        )
        self.m = self.beta1 * self.m + (1 - self.beta1) * globalg
        self.v = self.beta2 * self.v + (1 - self.beta2) * (globalg * globalg)
        step = -a * self.m / (np.sqrt(self.v) + self.epsilon)
        return step

    # Artifact from POET, not deleting but probably not using.
    # def propose(
    #     self, theta: FloatArray, globalg: FloatArray
    # ) -> tuple[np.float_, FloatArray]:
```

```
        #     a = self.stepsize
        #     # Unlike _compute_step(), propose may not have t updated prior to
        #     # being called. If t==0, the scaling factor is effectively set to 1.
        #     if self.t > 0:
        #         a *= np.sqrt(1 - self.beta2**self.t) / (1 - self.beta1**self.t)
        #
        #     m = self.beta1 * self.m + (1 - self.beta1) * globalg
        #     v = self.beta2 * self.v + (1 - self.beta2) * (globalg * globalg)
        #     step = -a * m / (np.sqrt(v) + self.epsilon)
        #     ratio: np.floating[Any] = np.linalg.norm(step) / np.linalg.norm(theta)
        #     return ratio, theta + step
```

16.4 marco_polo/optimizers/uber_es/modules/result_objects.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Result of runs"""
from collections import namedtuple


POResult = namedtuple("POResult", ["noise_inds", "returns", "lengths", "rollouts"])

EvalResult = namedtuple(
    "EvalResult",
    ["returns", "lengths", "eval_returns_mean", "eval_returns_max", "eval_returns_min"],
)
```

16.5 marco_polo/optimizers/uber_es/modules/rollouts.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


"""Uber ES Simulation Rollouts"""

import argparse
import logging
import os
import subprocess
from collections.abc import Callable
from typing import Any, Optional, Union

import imageio
import numpy as np
from numpy.random import PCG64DXSM, Generator

from marco_polo.envs._base.env_params import EnvParams

# from marco_polo.optimizers.uber_es.modules.model import Model
from marco_polo.base_classes.model import BaseModel  # for type hinting
from marco_polo.optimizers.uber_es.modules.opt import ESTeamWrapper  # for type hinting
from marco_polo.optimizers.uber_es.modules.result_objects import EvalResult, POResult
from marco_polo.tools.wrappers import EnvHistory, TeamHistory
from marco_polo.tools.types import Role  # for type hinting

logger = logging.getLogger(__name__)


################################################################################
## Aux Multithreading Funcs
################################################################################
# # this has to be here!
# #  "globals" are actually only module global, not program or namespace global.
# #  so, to share globals between functions, they must all come from the
# #  same module and be imported to wherever they are getting used.

# # also, this is not necessary
# #  This is a cheap trick to leave a reference to shared objects in each worker,
# #  then we don't have to pass the objects as arguments.
# #  The standard way is to pass shared objects as arguments.
# #  There is no behavioral difference between these methods.
# def initialize_worker_fiber(arg_thetas, arg_niches, arg_tasks):
#     global tasks, thetas, niches
#     #from .noise_module import noise
#     thetas = arg_thetas
#     niches = arg_niches
#     tasks = arg_tasks


################################################################################
## Eval Funcs
################################################################################
########################################
## Eval Funcs
########################################
def run_viz_distributed(
    num_rollouts: int,
    rs_seed: int,
    env_params: EnvParams,
    env_creator_func: Callable[..., Any],
    env_id: str,
    team: TeamHistory,
```

```
    epoch: Optional[int],
    vidpath: str,
    frame_skip: int,
) -> None:
    """
    Distributed Vizualization Routine

    This fuction is design for asynchronous simulation vizualization. It takes a
    single team, performs a single rollout, and saves the video. There is no return
    and it is designed to be non-blocking.

    Parameters
    ----------
    num_rollouts: int
        number of visualization runs to do
        (this will usually be 1)
    rs_seed : int
        Random seed for PRNG
    env_params : EnvParams
        environment parameter object specifying the environment
    env_creator_func : Callable[..., Any]
        Function to create the environment from the environment parameters
        This is an artificant of pickle, because the full env can't be sent over pipes
    env_id : str
        ID of environment
    team : TeamHistory
        dictionary of agents to play
    epoch : Optional[int]
        Current simulation time
    vidpath : str
        Full path to directory for video storage
    frame_skip : int
        How many frames to skip when recording the gif

    Side-Effects
    ------------
    None
        When run asynchronously, everything is copied to this func, so no side-effects

    Returns
    -------
    None
        Saved gif to directory
    """
    # set new prng
    random_state = Generator(PCG64DXSM(seed=rs_seed))

    # setup simulator
    env = env_creator_func()
    env.augment(env_params)

    # set file name
    run_name = f"Epoch:{epoch}_envID:{env_id}"

    # simulate and write .gif
    for _ in range(num_rollouts):
        _simulate_viz(
            env=env,
            team=team,
            random_state=random_state,
            run_name=run_name,
            vidpath=vidpath,
            frame_skip=frame_skip,
        )


def _simulate_viz(
    env: EnvHistory,
    team: TeamHistory,
    random_state: Generator,
    run_name: str = "",
    vidpath: str = "",
    frame_skip: int = 1,
    batch_render: bool = False,
) -> None:
    """
    Simulation Function for Vizualization Only

    This function is similar to "_simulate_basic()" but specialized for
    vizualization only. It tracks frames and outputs a gif, but does not return
    rewards or observations.
```

```
160
161         Parameters
162         ----------
163         env : EnvHistory
164             Built from parameters and a gym creation function
165         team : TeamHistory
166             Agent team playing the environment
167         random_state : Numpy Generator
168             PRNG
169         run_name : str, default=""
170             Name of gif
171         vidpath : str, default=""
172             Full path to file output
173         frame_skip : int, default=1
174             render frame if i_framet % frame_skip == 0
175         batch_render: bool, default=False
176             whether to use batch rendering (i.e. run entire run first,
177             then render the results). Requires that the renderer
178             supports the render_all function.
179
180         Side-Effects
181         ------------
182         None
183             When run asynchronously, everything is copied to this func, so no side-effects
184
185         Returns
186         -------
187         None
188             Saved gif to directory
189         """
190         max_episode_length = getattr(env, "max_episode_steps", 2000)
191
192         # set environment seed
193         env.seed(random_state.integers(2**31 - 1, dtype=int))
194
195         # initial obs, and to clear the env
196         obs, _ = env.reset()
197
198         # rendering data structures
199         render_data = []
200         frames = []
201         result = [max_episode_length, False]  # default result
202
203         # Perform Run
204         for t in range(max_episode_length):
205             # if appropriate time, save frame
206             if t % frame_skip == 0:
207                 if batch_render:
208                     render_data.append(env.bundle_step_data())
209                 else:
210                     frames.append(env.render())
211
212             # action object for each agent
213             action = dict()
214
215             # loop over agents, send obs, get action
216             for role, agent in team.items():
217                 action[role] = agent.get_action(obs[role], t=t, mean_mode=False)
218
219             # send actions to game, get observations and etc
220             obs, _, term, trunc, _ = env.step(action)
221
222             # check if done
223             if all([term[k] or trunc[k] for k in term.keys()]):
224                 break
225
226         # Lets be sure to always capture the last frame.
227         if batch_render:
228             render_data.append(env.bundle_step_data())
229         else:
230             frames.append(env.render())
231
232         if batch_render:
233             frames = env.render_all(data=render_data, result=result)
234
235         # if there are any frames, store as video
236         if frames:
237             filename = os.path.join(vidpath, f"{run_name}.gif")
238             write_video(filename=filename, frames=frames, frame_skip=frame_skip)
239
240         # close viewer, if applicable
```

```
    env.render(close=True)


def write_video(filename: str, frames: list[Any], frame_skip: int) -> None:
    """Write the frames to a video

    Parameters
    ----------
    filename: str
        name of file to write
    frames: list[Any]
        frames of video to write
    frame_skip: int
        frame skipped in output - used to set the frame rate

    Side-Effects
    ------------
    None

    Returns
    -------
    None
        Saved gif to directory
    """
    # write gif
    #  original mimwrite() took fps, which we defined as 60/frame_skip
    #  update takes "duration", defined as 1000/fps
    #  therefore, we implement duration=int(1000*frame_skip/60)
    imageio.mimwrite(uri=filename, ims=frames, duration=int(1000 * frame_skip / 60))

    # compress gif
    # using direct function call because I can set the optimization that way
    # don't need to remove extensions - imagio doesn't add any
    # no point in reducing colorspace (--colors xx)- BPW videos have <32 colors anyway
    # no need to increase lossiness (--lossy=xx)- not enough going on in BPW
    # References:
    #  imagio notes -
         https://imageio.readthedocs.io/en/stable/examples.html#optimizing-a-gif-using-pygifsicle
    #  https://github.com/LucaCappelletti94/pygifsicle
    #   pygifesicle just uses subprocess to call gifsicle - no point in using
    #  https://www.lcdf.org/gifsicle/
    subprocess.call(
        ["gifsicle", "--no-warnings", "--optimize=2", filename, "--output", filename]
    )

    # debugging
    logger.debug(f"capturing gif: {filename}")


########################################
## Eval Funcs
########################################
def run_eval_batch_distributed(
    num_rollouts: int,
    rs_seed: int,
    env_params: EnvParams,
    env_creator_func: Callable[..., Any],
    team: dict[Role, BaseModel],
) -> dict[Role, EvalResult]:
    """
    Distributed Evaluation Routine

    This fuction is design for asynchronous evaluations. It does not track frames
    or output images, just tracks actions, rewards, and simulation length. It calls
    "_simulate_basic()" to perform simulations, then handles results for each agent.

    Parameters
    ----------
    num_rollouts : int
        How many evaluations to perform
    rs_seed : int
        Random seed for PRNG
    env_params : EnvParams
        Environment parameter object specifying the environment
    env_creator_func : Callable[..., Any]
        Function to create the environment from the envrionment parameters
        This is an artificant of pickle, because the full env can't be sent over pipes
    team : dict[role, BaseModel]
        Dictionary of agents to play

    Side-Effects
```

```
    ------------
    None
        When run asynchronously, everything is copied to this func, so no side-effects

    Returns
    -------
    dict[role, EvalResult]
        Dictionary of EvalResult objects for each agent
    """
    # set new prng
    random_state = Generator(PCG64DXSM(seed=rs_seed))

    # setup simulator
    env = env_creator_func()
    env.augment(env_params)

    # simulation return objects
    returns = {role: np.zeros(shape=(num_rollouts,)) for role in team.keys()}
    lengths = np.zeros(shape=(num_rollouts,), dtype="int")
    results = {}

    # loop over number of rollouts
    for i in range(num_rollouts):
        # simulate
        # store results
        r, lengths[i], _ = _simulate_basic(
            env=env, team=team, random_state=random_state
        )
        # update returns
        for role in team.keys():
            returns[role][i] = r[role]

    # loop over agents, fill EvalResult
    for role in team.keys():
        results[role] = EvalResult(
            returns=returns[role],
            lengths=lengths,
            eval_returns_mean=returns[role].mean(),
            eval_returns_max=returns[role].max(),
            eval_returns_min=returns[role].min(),
        )

    # return results
    return results


########################################
## Optim Funcs
########################################
def run_optim_batch_distributed(
    num_rollouts: int,
    rs_seed: int,
    env_params: EnvParams,
    env_creator_func: Callable[..., Any],
    team: ESTeamWrapper,
    setup_args: argparse.Namespace,
    noise_std: float,
) -> dict[Role, POResult]:
    """
    Distributed Optimization Routine

    This fuction is designed for asynchronous optimization. It does not track frames
    or output images, just tracks actions, rewards, and simulation length. It calls
    "_simulate_basic()" to perform simulations, then handles results for each agent.

    Parameters
    ----------
    num_rollouts : int
        How many evaluations to perform
        The actual amount is 2x this, since we eval + and - gradient perturbations
    rs_seed : int
        Random seed for PRNG
    env_params : EnvParams
        Environment parameter object specifying the environment
    env_creator_func : Callable[..., Any]
        Function to create the environment from the environment parameters
        This is an artificant of pickle, because the full env can't be sent over pipes
    team : ESTeamWrapper
        Dictionary of agents to play
    setup_args : args.Namespace
        Input args for the agent wrapper
```

```
    noise_std : float
        Standard deviation when drawing gradient perturbations

    Side-Effects
    ------------
    None
        When run asynchronously, everything is copied to this func, so no side-effects

    Returns
    -------
    dict[role, POResult]
        Dictionary of POResult objects for each agent
    """
    # set new prng
    random_state = Generator(PCG64DXSM(seed=rs_seed))
    # draw seeds for weight shifting
    seeds = random_state.integers(low=0, high=(2**63), size=(num_rollouts,))

    # setup simulator
    env = env_creator_func()
    env.augment(env_params)

    # simulation return objects
    returns = {role: np.zeros(shape=(num_rollouts, 2)) for role in team.roles()}
    lengths = np.zeros(shape=(num_rollouts, 2), dtype="int")
    agt_seeds = []
    results = dict()

    # loop over rollouts, do + and - point estimates
    for i, seed in enumerate(seeds):
        # shift weights and store seed
        agt_seeds.append(team.shift_weights(noise_std=noise_std, seed=seed))

        # first rollout
        r, lengths[i, 0], _ = _simulate_basic(
            env=env, team=team, random_state=random_state
        )

        # update returns
        for role in team.roles():
            returns[role][i, 0] = r[role]

        # shift weights in other direction
        _ = team.shift_weights(noise_std=-noise_std, seed=seed)

        # second rollout
        r, lengths[i, 1], _ = _simulate_basic(
            env=env, team=team, random_state=random_state
        )

        # store returns
        for role in team.roles():
            returns[role][i, 1] = r[role]

    # loop over agents, fill return
    for role in team.roles():
        results[role] = POResult(
            returns=returns[role],
            noise_inds=[a[role] for a in agt_seeds],
            lengths=lengths,
            rollouts=[],
        )

    # return results
    return results


def _simulate_basic(
    env: Any, team: Union[ESTeamWrapper, dict[Role, BaseModel]], random_state: Generator
) -> tuple[dict[Role, float], int, list[dict[Role, Any]]]:
    """
    Simulation Function for Evalutions Only

    This function is similar to "_simulate_viz()" but specialized for
    evaluation only. It tracks actions, rewards, and simulation time, but does
    not track frames or output video.

    Parameters
    ----------
    env : Any
        Built from parameters and a gym creation function
```

```
    team : Union[ESTeamWrapper, dict[Role, BaseModel]]
        Team playing the environment
    random_state : Numpy Generator
        PRNG

    Side-Effects
    ------------
    None
        When run asynchronously, everything is copied to this func, so no side-effects

    Returns
    -------
    tuple[dict[Role, float], int, list[dict[Role, Any]]]
        total_reward : dict[role, score]
            Dictionary of scores for each agent
        t : int
            Simulation time when finished
        actions : list[dict[role, action]]
            List of actions from each agent on each timestep
    """
    max_episode_length = getattr(env, "max_episode_steps", 2000)

    # set environment seed
    env.seed(random_state.integers(2**31 - 1, dtype=int))

    # initial obs, and to clear the env
    obs, _ = env.reset()

    # setup return things
    actions = []
    total_reward = {role: 0.0 for role in team.roles()}

    # function that does the step
    take_step = env.step

    # check if there is an optimized version for no visualization case
    # if so, use that instead
    take_step_no_viz = getattr(env, "step_noviz", None)
    if callable(take_step_no_viz):
        take_step = take_step_no_viz

    # Perform Run
    for t in range(max_episode_length):
        # action object for each agent
        action = dict()

        # loop over agents, send obs, get action
        for role, agent in team.items():
            action[role] = agent.get_action(obs[role], t=t, mean_mode=False)

        # store actions for later
        actions.append(action)

        # send actions to game, get observations and etc
        obs, reward, term, trunc, info = take_step(action)

        # update reward
        for role in total_reward:
            total_reward[role] += reward[role]

        # check if done
        if all([term[k] or trunc[k] for k in term.keys()]):
            break

    # return
    return total_reward, t, actions
```

17 marco_polo/tools

17.1 marco_polo/tools/compiled.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


from numba import njit  # type: ignore[import]


@njit
def fast_clip(array: list[float], min: float, max: float) -> list[float]:
    """Clip all values in array to be in [min, max]

    Any vaule less than `min` will be set to `min`
    Any value more than `max` will be set to `max`

    Parameters
    ----------
    array: list[float]
        iterable of data to clip
    min: float
        minimum value
    min: float
        minimum value

    Returns
    -------
    list[float]
        data after clipping
    """
    n = len(array)
    assert max > min, "max must be greater than min"
    for i in range(n):
        if array[i] < min:
            array[i] = min
        elif array[i] > max:
            array[i] = max
    return array


@njit
def fast_sum(array: list[float]) -> float:
    """Return the sum of the given array

    Parameters
    ----------
    array: list[float]
        iterable of data to sum

    Returns
    -------
    float:
        sum of array
    """
    result = 0.0
    n = len(array)
    for i in range(n):
        result += array[i]
    return result
```

17.2 marco_polo/tools/extraneous.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Various uncategorized code"""
import multiprocessing
import sys
from collections.abc import Iterable, Mapping
from multiprocessing.pool import AsyncResult
from typing import Any, Optional


def get_size(obj: Any, seen: Optional[set[Any]] = None) -> int:
    """
    Recursively finds size of objects

    This function has some issues with reference freeing.
    Just be careful calling several times, idk what's going on.

    """
    # https://goshippo.com/blog/measure-real-size-any-python-object/
    size = sys.getsizeof(obj)
    if seen is None:
        seen = set()
    obj_id = id(obj)
    if obj_id in seen:
        return 0
    # Important mark as seen *before* entering recursion to gracefully handle
    # self-referential objects
    seen.add(obj_id)
    if isinstance(obj, dict):
        size += sum([get_size(v, seen) for v in obj.values()])
        size += sum([get_size(k, seen) for k in obj.keys()])
    elif hasattr(obj, "__dict__"):
        size += get_size(obj.__dict__, seen)
    elif hasattr(obj, "__iter__") and not isinstance(obj, (str, bytes, bytearray)):
        size += sum([get_size(i, seen) for i in obj])
    return size


class DummyAsyncResult(AsyncResult[Any]):
    """Dummy 'async' result that is always ready

    This is intended to match the interface so that a non-async item
    can seemlessly replace an async one. This would be used to allow
    a single process execution to use the same code as a multiprocess
    execution.

    Example usage:
    handle = DummyAsyncResult(None, None, None)
    handle.set_result(function())  # function is the 'async' function
    ...
    result = handle.get()  # always ready
    """

    def __init__(self, pool: Any, callback: Any, error_callback: Any) -> None:
        self._pool = pool
        self._event = None
        self._job = None
        self._cache = None
        self._callback = callback
        self._error_callback = error_callback
        self._result = None

    def set_result(self, data: Any) -> None:
        """Set the output of the 'async' run"""
        self._result = data
```

```
79
80      def ready(self) -> bool:
81          """Return True"""
82          return True
83
84      def successful(self) -> bool:
85          """Return True"""
86          return True
87
88      def wait(self, timeout: Any = None) -> Any:  # pylint: disable=unused-argument
89          """Return the result"""
90          return self._result
91
92      def get(self, timeout: Any = None) -> Any:  # pylint: disable=unused-argument
93          """Return the result"""
94          return self._result
95
96      def _set(self, i: Any, obj: Any) -> None:
97          pass
98
99
100 class SingleProcessPool(multiprocessing.pool.Pool):  # pylint: disable=abstract-method
101     """This is a fake Pool with only a single worker.
102
103     The async jobs are done directly by the main process
104     This exists for testing purposes
105     """
106
107     def __init__(self) -> None:  # pylint: disable=super-init-not-called
108         # this allows the object to be correctly deleted
109         self._state = multiprocessing.pool.CLOSE
110
111     def apply_async(  # pylint: disable=too-many-arguments,dangerous-default-value
112         self,
113         func: Any,
114         args: Iterable[Any] = (),
115         kwds: Mapping[str, Any] = {},
116         callback: Any = None,
117         error_callback: Any = None,
118     ) -> DummyAsyncResult:
119         handle = DummyAsyncResult(None, None, None)
120         result = func(*args, **kwds)
121         handle.set_result(result)
122         return handle
```

17.3 marco_polo/tools/iotools.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Contains various utility functions/classes used in the project"""
import csv
import json
from collections.abc import Callable  # for type hinting
from typing import Any, cast, Union  # for type hinting
import os

import numpy as np

from marco_polo.tools.types import (
    EnvId,
    FloatArray,
    PathString,
    Role,
)  # for type hinting


################################################################################
## Auxiliary Functions
################################################################################
### Turn dictionary with keyed tuple into csv
def save_keyed_tuple(
    dct: dict[tuple[EnvId, EnvId], float],
    filename: PathString,
    do_sort: bool = True,
) -> None:
    """
    Writes a csv file of a dict that is indexed by a tuple (a matrix)

    Parameters
    ----------
    dct : dict
        Dict of values with keys given as tuples
    filename : str
        Name of file to save to
    do_sort : boolean, optional
        whether to sort the output keys
        This is assuming they are of the form "Env_#"

    Side-Effects
    ------------
    None

    Returns
    -------
    None

    Notes
    -----
    As an example, for a dict of:
    dct = {(a,a): val_aa, (a,b): val_ab,
           (b,a): val_ba, (b,b): val_bb,
           (c,a): val_ca, (c,b): val_cb,
           (d,a): val_da, (d,b): val_db}
    The ouput is:
        ,a,b
        a,val_aa,val_ab
        b,val_ba,val_bb
        c,val_ca,val_cb
        d,val_da,val_db
    Note that the ',' at the start of the first line is intentional
    to provide an empty cell in the csv format so the data forms a
    rectangular matrix.
```

```
    """
    # split (l, r) keys into lists of l and r
    l_keys = list({first for first, second in dct})
    r_keys = list({second for first, second in dct})

    # sort by the numeric part of Env_X
    if do_sort:
        l_keys.sort(key=lambda x: int(x.split("_")[1]))
        r_keys.sort(key=lambda x: int(x.split("_")[1]))

    with open(filename, mode="w", newline="", encoding="utf8") as file:
        csvfile = csv.writer(
            file, delimiter=",", quoting=csv.QUOTE_MINIMAL, lineterminator="\n"
        )
        header = [""] + r_keys  # [""] to add blank entry to csv row
        csvfile.writerow(header)

        for first in l_keys:
            row = [first] + [dct.get((first, second), "") for second in r_keys]
            csvfile.writerow(row)


def load_keyed_tuple(
    filename: PathString,
    format_function: Callable[[str], Any] = lambda x: x,
) -> dict[tuple[EnvId, EnvId], Any]:
    """
    Read a tuple-indexed dict (a matrix) from a csv file.

    This is the reverse of save_keyed_tuple
    The format_function is used to convert the values from string to
    whatever format is desirable. The default leaves it as a string.
    A typical choice would be float.

    Parameters
    ----------
    filename : str
        Name of file to save to
    format_function: callable, optional
        This is applied to values (but not keys) read from the file

    Returns
    -------
    dict
        The dict that was read

    Notes
    -----
    For a file with the following:
        ,a,b
        a,val_aa,val_ab
        b,val_ba,val_bb
        c,val_ca,val_cb
        d,val_da,val_db
    The output is:
    dct = {(a,a): val_aa, (a,b): val_ab,
           (b,a): val_ba, (b,b): val_bb,
           (c,a): val_ca, (c,b): val_cb,
           (d,a): val_da, (d,b): val_db}
    """
    dct: dict[tuple[EnvId, EnvId], Any] = {}
    with open(filename, mode="r", newline="", encoding="utf8") as file:
        csvfile = csv.reader(file, delimiter=",")
        # first line has an empty spot to account for alignment
        # of columns. So, we ignore that
        r_keys = next(csvfile)[1:]
        for l_key, *values in csvfile:
            for r_key, value in zip(r_keys, values):
                # appease type checker
                l_key = cast(EnvId, l_key)
                r_key = cast(EnvId, r_key)
                dct[(l_key, r_key)] = format_function(value)
    return dct


################################################################################
## Numpy Encoders for JSON
################################################################################
## Recursivly encodes objects with a reprJSON function
# https://stackoverflow.com/questions/5160077/encoding-nested-python-object-in-json
#
```

```python
# Encdoing numpy objects:
# https://stackoverflow.com/questions/26646362/numpy-array-is-not-json-serializable
#
# Decoding objects
# https://stackoverflow.com/questions/48991911/how-to-write-a-custom-json-decoder-for-a-complex-object

# Usage
# with open(filename, 'w') as jsonfile:
#     json.dump(edge, jsonfile, cls=NumpyEncoder)
# with open(filename, 'r') as jsonfile:
#     edge1 = json.load(jsonfile, cls=NumpyDecoder)


class NumpyEncoder(json.JSONEncoder):
    """Encode numpy data for JSON writer"""

    def default(self, o):  # type: ignore  # this is called by the JSON library
        if isinstance(o, np.integer):
            return {"np.integer": int(o)}
        if isinstance(o, np.floating):
            return {"np.floating": float(o)}
        if isinstance(o, np.ndarray):
            return {"np.array": o.tolist()}
        return json.JSONEncoder.default(self, o)


class NumpyDecoder(json.JSONDecoder):
    """Decode numpy data from JSON"""

    def __init__(self, *args, **kwargs):  # type: ignore  # this is called by the JSON library
        json.JSONDecoder.__init__(self, object_hook=self.numpy_hook, *args, **kwargs)

    def numpy_hook(self, dct):  # type: ignore  # this is called by the JSON library
        """Convert dict with numpy data to numpy object"""
        if "np.integer" in dct:
            return np.int_(dct["np.integer"])
        if "np.floating" in dct:
            return np.float_(dct["np.floating"])
        if "np.array" in dct:
            return np.array(dct["np.array"])
        return dct


################################################################################
## TensorFlow Logging
################################################################################
class TBWriter:
    def __init__(self, args: dict[str, Any]) -> None:
        """ """
        from tensorflow import summary  # type: ignore[import]
        import time

        self.summary = summary
        log_dir = args["log_dir"]

        now = time.localtime()
        subdir = time.strftime("%d-%b-%Y_%H.%M.%S", now)

        self.summary_dir = os.path.join(log_dir, subdir)
        self.summary_writer: dict[str, summary.SummaryWriter] = {}

    def create_scalar_writer(self, name: str) -> None:
        new_dir = os.path.join(self.summary_dir, name)
        self.summary_writer[name] = self.summary.create_file_writer(new_dir)

    def write_item(self, name: str, label: str, data: Any, step: int) -> None:
        if label not in self.summary_writer.keys():
            self.create_scalar_writer(label)

        with self.summary_writer[label].as_default():
            if isinstance(data, (np.ndarray, np.generic)):
                data = data.item()

            if hasattr(data, "__len__"):
                data = data[0]

            self.summary.scalar(name=name, data=data, step=step)
        self.summary_writer[label].flush()

    def write_items(self, name: str, data: dict[str, list[float]], step: int) -> None:
        for label in data.keys():
```

```
            self.write_item(name, label, data[label], step)


class Telemetry:
    def __init__(self) -> None:
        self.loggers: list[TBWriter] = []

    def add_logger(self, logger: TBWriter) -> None:
        self.loggers.append(logger)

    def write_item(self, name: str, label: str, data: Any, step: int) -> None:
        for logr in self.loggers:
            logr.write_item(name, label, data, step)

    def write_items(self, name: str, data: dict[str, list[float]], step: int) -> None:
        for logr in self.loggers:
            logr.write_items(name, data, step)
```

17.4 marco_polo/tools/logger.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


import csv
import logging
from pprint import pformat

from marco_polo.tools.types import PathString  # for type hinting

logger = logging.getLogger(__name__)


class CSVLogger:
    """
    CSVLogger Class Docs

    Attributes
    ----------

    Methods
    -------

    """

    def __init__(self, fnm: PathString, col_names: list[str]) -> None:
        """ """
        logger.info("Creating data logger at {}".format(fnm))
        self.fnm = fnm
        self.col_names = col_names

        with open(fnm, mode="a", newline="", encoding="utf8") as f:
            writer = csv.writer(f, delimiter=",")
            writer.writerow(col_names)

        # hold over previous values if empty
        self.vals = {name: None for name in col_names}

    def log(self, **cols: str) -> None:
        """ """
        self.vals.update(cols)  # type: ignore  # complicated to fix
        logger.info(pformat(self.vals))

        if any(key not in self.col_names for key in self.vals):
            raise Exception("CSVLogger given invalid key")

        with open(self.fnm, mode="a", newline="", encoding="utf8") as f:
            writer = csv.writer(f, delimiter=",")
            writer.writerow([self.vals[name] for name in self.col_names])
```

17.5 marco_polo/tools/section_logging.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Class to log sections of runs

Using this class cleans up code elsewhere

"""
import logging  # for type hinting
import time
from typing import Any


class SectionLogger:
    """Class tracks time and prints banners

    Example usage:
    section = SectionLogger(logger, sectin_name="NAME", prefix="step 1")
    ...
    section.print_status_banner("update msg")
    section.print_raw("some message")
    section.print_time()
    section.print_end_banner()

    This will print (via the logger's info stream):
    step 1 - ################ NAME Start ################
    step 1 - ################ NAME update msg ################
    step 1 - some message
    step 1 - NAME Step Time: 12.4 seconds
    step 1 - ################ NAME Complete ################

    Alternate usage with context manager (gives same output):
    with SectionLogger(logger, sectin_name="NAME", prefix="step 1") as section:
        ...
        section.print_status_banner("update msg")
        section.print_raw("some message")
        section.print_time()

    """

    def __init__(
        self,
        logger: logging.Logger,
        section_name: str,
        prefix: str = "",
        print_start: bool = True,
    ) -> None:
        """Create the object and print the start banner

        Parameters
        ----------
        logger: logging.Logger
            the logging object to use for output
        section_name: str
            name of the section to print in banners
        prefix: str, default = ""
            prefix to prepend to each output line
            default is no prefix
        print_start: bool, default = True
            Whether to print the start banner when object is created
            default is True
        """
        self.start_time = time.time()
        self.logger = logger
        self.section_name = section_name
        self.prefix = prefix
```

```
        # if there is a prefix, add a spacer after it
        if self.prefix:
            self.prefix += " - "
        if print_start:
            self.print_start_banner()

    def print_start_banner(self) -> None:
        """Print the start banner."""
        self.logger.info(
            f"{self.prefix}############### {self.section_name} Start ###############"
        )

    def print_time(self) -> None:
        """Print the current time elapsed"""
        delta_time = time.time() - self.start_time
        self.logger.info(
            f"{self.prefix}{self.section_name} Step Time: {delta_time:.2f} seconds"
        )

    def print_end_banner(self) -> None:
        """Print the end banner."""
        self.logger.info(
            f"{self.prefix}############### {self.section_name} Complete ###############\n"
        )

    def print_status_banner(self, status: str) -> None:
        """Print a status banner."""
        self.logger.info(
            f"{self.prefix}############### {self.section_name} {status} ###############"
        )

    def print_raw(self, line: str) -> None:
        """Print a line with the prefix."""
        self.logger.info(f"{self.prefix}{line}")

    def __enter__(self) -> "SectionLogger":
        """Enter context, return object."""
        return self

    def __exit__(self, *args: Any) -> None:
        """Exit context, print end banner."""
        self.print_end_banner()
```

17.6 marco_polo/tools/stats.py

```
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# The following code is modified from uber-research/poet
# (https://github.com/uber-research/poet)
# under the Apache 2.0 License.


from typing import Iterator

import numpy as np
from numpy.typing import NDArray

from marco_polo.tools.types import FloatArray, IntArray


def _comp_ranks(x: FloatArray) -> IntArray:
    """
    Returns ranks in [0, len(x))

    Note: This is different from scipy.stats.rankdata, which returns ranks in [1, len(x)].
    """

    # init empty vector for return
    ranks = np.empty(len(x), dtype=int)

    # Compute ranks of x
    #  argsort() returns indices of ranks
    #  arange() returns a range 0:x
    #  This is the fastest way to do this.
    ranks[x.argsort()] = np.arange(len(x))

    return ranks


def compute_CS_ranks(x: FloatArray) -> FloatArray:
    """Compute Centered and Scaled Ranks

    This function takes an object (presumbably the scores), centers the scores
    on 0, and normalizes between -0.5 and 0.5, inclusive.

    Parameters
    ----------
    x: 2-D Numpy array
        This is a [`optim_jobs`*`rollouts_per_optim_job`, 2] array of evaluation scores

    Returns
    -------
    y: Numpy array
        Array of ranks linearly spaces on the closed interval [-0.5, 0.5]

    Reference
    ---------
    https://stats.stackexchange.com/questions/164833/how-do-i-normalize-a-vector-of-numbers-so-they-are-between-0-and-1

    """

    # Safety check
    if x.size == 1:
        # print("stats.computer_centered_ranks:: x is of size 1")
        return np.zeros(shape=1, dtype=np.float32)

    # Compute ranks
    #  object is the same dimensions as x
    #  has values [0, len(x))
    y = _comp_ranks(x.ravel()).reshape(x.shape).astype(np.float32)

```

```
    # Normalize to (0, 1)
    #  This works because the values are ranks, 0 to len(x)
    #  so max(x) - min(x) = size - 1
    y /= x.size - 1

    # Center vector at 0
    #  The vector is already normed (0,1), so shift it by -0.5,
    #  it's now centered at 0, with range (-0.5, 0.5)
    y -= 0.5

    return y


def compute_weighted_sum(
    weights: FloatArray, vec_generator: Iterator[FloatArray], theta_size: int
) -> tuple[NDArray[np.float32], int]:
    """Calculates a weighted sum

    This function calculates a weighted sum using the weights input list as the value for
    each vector from vec_generator. There is no safety if vec_generator is longer
    than weights.

    Parameters
    ----------
    weights: FloatArray
        Array of numeric weights
    vec_generator: Iterator[FloatArray]
        Iterator of numpy arrays, where the generator is the same length as the weights
    theta_size: int
        Size of arrays created by vec_generator

    Returns
    -------
    total: NDArray[np.float32]
        Weighted sum of vectors, length theta_size
    num_items_summed: int
        Count of the items combined. This should be the same length as weights, but
        is defined as the number of items from vec_generator
    """

    # setup return objects
    #  use the summation as a counter
    total = np.zeros(theta_size, dtype=np.float32)  # TODO: why float32 specifically??
    num_items = 0

    # loop through noise generator
    for vec in vec_generator:
        # grab weight and multiply by noise
        total += weights[num_items] * vec
        # increment counter
        num_items += 1

    # return total vector and number of estimates combined
    return total, num_items
```

17.7 marco_polo/tools/types.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""types used in MarcoPolo"""
from typing import NewType, Union
from typing_extensions import LiteralString
from os import PathLike

import numpy as np
from numpy.typing import NDArray


FloatArray = NDArray[np.float_]
ActivationType = FloatArray

IntArray = NDArray[np.int_]

RenderFrame = NDArray[np.uint8]

# using NewType can help find logical errors within MarcoPolo
# however, the type will conflict with typing in pettingzoo
# If that's a problem, comment out that definition and uncomment
# the one after it
Role = NewType("Role", str)
# Role = str

AgentId = NewType("AgentId", str)
EnvId = NewType("EnvId", str)
TeamId = NewType("TeamId", str)

# PathString = NewType(
#     "PathString", Union[str, PathLike[str], bytes, PathLike[bytes], LiteralString]
# )
# PathString = Union[str, PathLike[str], bytes, PathLike[bytes], LiteralString]
PathString = str
```

17.8 marco_polo/tools/wrappers.py

```python
# Copyright (c) 2023 Mobius Logic, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


"""Wrappers used in MarcoPolo"""
import argparse
import copy
import json
import logging
import os
from collections import deque
from collections.abc import Callable, Iterable, Iterator  # for type hinting
from typing import Any, cast, ClassVar, Optional, Union  # for type hinting

from pettingzoo.utils import agent_selector  # type: ignore[import]
from pettingzoo.utils.conversions import (  # type: ignore[import]
    aec_to_parallel_wrapper,
    parallel_to_aec_wrapper,
)
from pettingzoo.utils.wrappers import OrderEnforcingWrapper  # type: ignore[import]
from tianshou.env.pettingzoo_env import PettingZooEnv  # type: ignore[import]

from marco_polo.base_classes.model import BaseModel
from marco_polo.envs._base.env_params import EnvParams
from marco_polo.tools.iotools import NumpyDecoder, NumpyEncoder
from marco_polo.tools.types import (
    AgentId,
    EnvId,
    FloatArray,
    PathString,
    RenderFrame,
    Role,
    TeamId,
)  # for type hinting

logger = logging.getLogger(__name__)


RunTask = tuple["EnvHistory", "TeamHistory"]


################################################################################
## Environment Stats Class
################################################################################
class EnvHistoryStats:
    """Stats for an EnvHistory object

    Attributes
    ----------
    team : TeamHistory
        current team of agents for the env
    iterations_lifetime : int
        Number of iterations in enviroments entire history across epochs
    iterations_current : int
        Current iteration in epoch
    created_at : int
        Epoch created at
    recent_scores : deque(maxlen=5)
        The 5 most recent scores achieved
    transfer_threshold : float
        Score threshold for transfers to happen
    best_score : float
        Lifetime best score on this env
        default: -infinity if no run has been done
    best_team : TeamHistory
        Team of agents that scored the best score on this env
    """
```

```
79
80      def __init__(self) -> None:
81          self.team: TeamHistory = NoneTeam(None)
82
83          self.iterations_lifetime: int = 0
84          self.iterations_current: int = 0
85          self.created_at: int = 1
86
87          #  https://stackoverflow.com/q/5944708/how-to-force-a-list-to-a-fixed-size
88          self.recent_scores = deque[float](maxlen=5)
89          self.transfer_threshold = float("-inf")
90
91          self.best_score = float("-inf")
92          self.best_team: TeamHistory = NoneTeam(None)
93
94      def toJSON(self) -> str:  # pylint: disable=invalid-name
95          """Return a json string version of the object"""
96          tmp = {k: v for k, v in self.__dict__.items() if k not in {"team", "best_team"}}
97
98          tmp["team"] = self.team.id
99          tmp["best_team"] = self.best_team.id
100         tmp["recent_scores"] = list(self.recent_scores)
101
102         return json.dumps(tmp, cls=NumpyEncoder)
103
104     def fromJSON(
105         self, data: str, team_dict: dict[str, "TeamHistory"]
106     ) -> None:  # pylint: disable=invalid-name
107         """Fill this object with data from the given JSON data
108
109         In addition to reading the JSON data, teams are pulled by
110         name from the given dict of teams
111
112         Parameters
113         ----------
114         data : str
115             JSON string with object data
116         team_dict : dict[str, TeamHistory]
117             dictionary of known teams
118
119         Side-Effects
120         ------------
121         object data is completely replaced by data from the JSON data
122         """
123         for key, val in json.loads(data).items():
124             if key == "team":
125                 self.team = team_dict[val]
126             elif key == "best_team":
127                 self.best_team = team_dict.get(val, NoneTeam(None))
128             elif key == "best_score":
129                 self.best_score = float(val)
130             elif key == "transfer_threshold":
131                 self.transfer_threshold = float(val)
132             elif key == "recent_scores":
133                 for i in val:
134                     self.recent_scores.append(i)
135             else:
136                 self.__dict__[key] = val
137
138     def __str__(self) -> str:
139         return "\n".join([f"{key}: {val}" for key, val in self.__dict__.items()])
140
141
142 ################################################################################
143 ## Environment History Class
144 ################################################################################
145 class EnvHistory:
146     """Env wrapper class
147
148     This is a wrappper around the env_params object, with the
149     information need to create an instance of the env class.
150     Attributes accessed for this class that don't exist are instead
151     pulled from the underlying env_params
152
153     Parameters
154     ----------
155     env_param : EnvParams
156         parameters to use for this environment
157     env_creator : class/function
158         Class or function that can be used to create the env instance
159
```

```
    Attributes
    ----------
    id : str
        A human readable ID of the object
        format is "Env_X" where X is the numeric index of the object
    env_param : EnvParams
        parameter object for the environment
    env_creator : class/function
        Class or function that can be used to create the env instance
    stats : EnvHistoryStats
        Statistics for this env
    """

    # this is a counter of the total number of Envs that have been
    # created (not just the number that are currently active). It
    # is used to form the id string of a new env object
    id_counter: ClassVar[int] = 0

    def __init__(self, env_param: EnvParams, env_creator: Callable[..., Any]) -> None:
        EnvHistory.id_counter += 1
        self.id = cast(
            EnvId, f"Env_{EnvHistory.id_counter}"
        )  # pylint: disable=invalid-name
        self.env_param = env_param
        self.env_creator = env_creator
        self.stats = EnvHistoryStats()

    def __getitem__(self, key: str) -> Any:
        return self.env_param[key]

    def __setitem__(self, key: str, value: Any) -> None:
        self.env_param[key] = value

    def __getattr__(self, name: str) -> Any:
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
        if name.startswith("_"):
            raise AttributeError(f"accessing private attribute '{name}' is prohibited")

        if name in self.__dict__:
            return self.__dict__[name]

        return getattr(self.env_param, name)

    def __repr__(self) -> str:
        return f"Env {self.id}"

    def get_env_creator(self) -> Callable[..., Any]:
        """Return the callable (function/class) to create a new env"""
        return self.env_creator

    def get_mutated_env(self) -> "EnvHistory":
        """Return a mutated copy of this object

        The mutation method is completely controlled by the
        env_params object

        Returns
        -------
        EnvHistory
            New object, mutated from this object
        """

        # Generate new parameters
        new_param = self.env_param.get_mutated_params()
        # wrap in new environment
        new_env = EnvHistory(env_param=new_param, env_creator=self.env_creator)
        # return new environment
        return new_env

    def log(self) -> None:
        """Write the object to the logger"""
        logger.info(f"EnvHistory.log() Environment {self.id}: " + str(self.stats))

    def checkpoint(self, folder: PathString) -> None:
        """Save the object to the given folder"""
        folder = os.path.join(folder, "Environments", self.id)
        os.makedirs(folder, exist_ok=True)

        tmp = {
            k: v
            for k, v in self.__dict__.items()
```

```
            if k not in {"env_param", "env_creator"}
        }
        tmp["stats"] = tmp["stats"].toJSON()

        checkpoint_path = os.path.join(folder, "EnvHistory.json")
        with open(checkpoint_path, mode="w", encoding="utf8") as file:
            json.dump(tmp, file, cls=NumpyEncoder)

        self.env_param.checkpoint(folder)

    def reload(self, folder: PathString, team_dict: dict[str, "TeamHistory"]) -> None:
        """Replace the data in this object with data from the given folder"""
        checkpoint_path = os.path.join(folder, "EnvHistory.json")
        with open(checkpoint_path, mode="r", encoding="utf8") as file:
            data = json.load(file, cls=NumpyDecoder)

        for key, val in data.items():
            self.__dict__[key] = val

        self.stats = EnvHistoryStats()
        self.stats.fromJSON(data["stats"], team_dict)
        self.env_param.reload(folder)


################################################################################
## Team Stats Class
################################################################################
class TeamHistoryStats:
    """Currently does nothing"""

    def __init__(self) -> None:
        pass

    def toJSON(self) -> str:  # pylint: disable=invalid-name
        """Return a json string version of the object - currently {}"""
        return json.dumps(self, default=lambda o: o.__dict__, sort_keys=True)

    def fromJSON(self, data: str) -> None:  # pylint: disable=invalid-name
        """Currently does nothing"""


################################################################################
## Team History Class
################################################################################
class TeamHistory:
    """Team wrapper class

    This is a wrappper around the team object.
    Attributes accessed for this class that don't exist are instead
    pulled from the underlying team

    Parameters
    ----------
    agent_group : Optional[dict[Role, "AgtHistory"]]
        dict mapping agents to roles

    Attributes
    ----------
    id : str
        A human readable ID of the object
        format is "Team_X" where X is the numeric index of the object
    parent :
        The parent of this team. Updated on copy
    agent_group : Optional[dict[Role, "AgtHistory"]]
        Agents in this team
    stats : TeamHistoryStats
        Statistics for this team
    """

    # this is a counter of the total number of teams that have been
    # created (not just the number that are currently active). It
    # is used to form the id string of a new team object
    id_counter: ClassVar[int] = 0

    def __init__(self, agent_group: Optional[dict[Role, "AgtHistory"]] = None) -> None:
        TeamHistory.id_counter += 1
        self.id = cast(
            TeamId, f"Team_{TeamHistory.id_counter}"
        )  # pylint: disable=invalid-name
        self.parent: Union[str, None] = None
```

```
322             if agent_group is not None:
323                 self.agent_group = agent_group
324             self.stats = TeamHistoryStats()
325 
326         def __getattr__(self, name: str) -> Any:
327             """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
328             if name.startswith("_"):
329                 raise AttributeError(f"accessing private attribute '{name}' is prohibited")
330 
331             if name in self.__dict__:
332                 return self.__dict__[name]
333 
334             return getattr(self.agent_group, name)
335 
336         def __getitem__(self, key: Role) -> "AgtHistory":
337             return self.agent_group[key]
338 
339         def __setitem__(self, key: Role, value: "AgtHistory") -> None:
340             self.agent_group[key] = value
341 
342         def __repr__(self) -> str:
343             id_info = {
344                 ", ".join(
345                     f"{role}: {str(agent)}" for role, agent in self.agent_group.items()
346                 )
347             }
348             return f"Team {self.id} - {id_info}"
349 
350         def roles(self) -> Iterable[Role]:
351             """Return the roles in the team"""
352             return self.agent_group.keys()
353 
354         def agents(self) -> Iterable["AgtHistory"]:
355             """Return the roles in the team"""
356             return self.agent_group.values()
357 
358         def items(self) -> Iterable[tuple[Role, "AgtHistory"]]:
359             """Return the roles/agent pairs in the team"""
360             return self.agent_group.items()
361 
362         def copy(self) -> "TeamHistory":
363             """return a copy of the object"""
364             agent_group_copy = {
365                 role: agent.copy() for role, agent in self.agent_group.items()
366             }
367             new_team = TeamHistory(agent_group_copy)
368             new_team.stats = copy.deepcopy(self.stats)
369             new_team.parent = self.id
370             return new_team
371 
372         def checkpoint(self, folder: PathString) -> None:
373             """Save the object to the Teams subdirectory of the given folder"""
374             folder = os.path.join(folder, "Teams")
375             os.makedirs(folder, exist_ok=True)
376 
377             tmp = {k: v for k, v in self.__dict__.items() if k != "agent_group"}
378             tmp["stats"] = tmp["stats"].toJSON()
379             tmp["team"] = {role: agent.id for role, agent in self.agent_group.items()}
380 
381             checkpoint_path = os.path.join(folder, f"{self.id}.json")
382             with open(checkpoint_path, mode="w", encoding="utf8") as file:
383                 json.dump(tmp, file, cls=NumpyEncoder)
384 
385         def reload(
386             self,
387             team_file: PathString,
388             agent_dict: dict[AgentId, "AgtHistory"],
389         ) -> None:
390             """Replace the data in this object with data from the given file"""
391             with open(team_file, mode="r", encoding="utf8") as file:
392                 data = json.load(file, cls=NumpyDecoder)
393 
394             for key, val in data.items():
395                 self.__dict__[key] = val
396 
397             self.stats = TeamHistoryStats()
398             self.stats.fromJSON(data["stats"])
399             self.agent_group = {k: agent_dict[v] for k, v in data["team"].items()}
400 
401 
402     class NoneTeam(TeamHistory):
```

```
    """The team version of None

    This can be used in place of None when assigning a team.

    This isn't a great solution, but it simplifies some typing and
    checkpoint/reload operations

    Attributes
    ----------
    id : None
        the id is None instead of a string
    """

    def __init__(
        self, agent_group: Optional[dict[str, "AgtHistory"]] = None
    ) -> None:  # pylint: disable=super-init-not-called
        # Do NOT call super init(). It will mess up the team count
        # appease mypy with the change in type of id
        self.id = cast(TeamId, None)  # pylint: disable=invalid-name
        self.agent_group = {}

    def __getattr__(self, name: str) -> Any:
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
        raise TypeError("Trying to get attribute from NoneTeam")

    def __repr__(self) -> str:
        return f"Team {self.id} - "

    def copy(self) -> "NoneTeam":
        """return a new NoneTeam (they are all the same)"""
        return NoneTeam(agent_group=None)

    def checkpoint(self, folder: PathString) -> None:
        """Save the object to the Teams subdirectory of the given folder"""
        raise TypeError("Trying to checkpoint NoneTeam")

    def reload(
        self,
        team_file: PathString,
        agent_dict: dict[AgentId, "AgtHistory"],
    ) -> None:
        """Replace the data in this object with data from the given file"""
        raise TypeError("Trying to reload from NoneTeam")


################################################################################
## Agent Stats Class
################################################################################
class AgtHistoryStats:
    """Stats for an AgtHistory object - currently does nothing"""

    def __init__(self) -> None:
        pass

    def toJSON(self) -> str:  # pylint: disable=invalid-name
        """Return a json string version of the object - currently {}"""
        return json.dumps(self, default=lambda o: o.__dict__, sort_keys=True)

    def fromJSON(self, data: str) -> None:  # pylint: disable=invalid-name
        """currently does nothing"""


################################################################################
## Agent History Class
################################################################################
class AgtHistory:
    """Agent wrapper class

    This is a wrappper around the agent object.
    Attributes accessed for this class that don't exist are instead
    pulled from the underlying agent.

    Parameters
    ----------
    model : BaseModel
        The model defining this agent

    Attributes
    ----------
    id : str
        A human readable ID of the object
```

```
        format is "Agent_X" where X is the numeric index of the object
    parent : Optional[str]
        The id of this agent's parent
    model : BaseModel
        underlying model object
    stats : AgentHistoryStats
        Statistics for this agent
    """

    # this is a counter of the total number of agents that have been
    # created (not just the number that are currently active). It
    # is used to form the id string of a new agent object
    id_counter: ClassVar[int] = 0

    def __init__(self, model: BaseModel) -> None:
        AgtHistory.id_counter += 1
        self.id = cast(
            AgentId, f"Agent_{AgtHistory.id_counter}"
        )  # pylint: disable=invalid-name
        self.parent: Union[str, None] = None

        self.model = model
        self.stats = AgtHistoryStats()

    def __call__(self, *args: Any, **kwargs: Any) -> Any:
        return self.model(*args, **kwargs)

    def __getattr__(self, name: str) -> Any:
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
        if name.startswith("_"):
            raise AttributeError(f"accessing private attribute '{name}' is prohibited")

        if name in self.__dict__:
            return self.__dict__[name]

        return getattr(self.model, name)

    def __repr__(self) -> str:
        return f"Agent {self.id}"

    def copy(self) -> "AgtHistory":
        """return a copy of the object"""
        new_agent = AgtHistory(copy.deepcopy(self.model))
        new_agent.stats = copy.deepcopy(self.stats)
        new_agent.parent = self.id
        return new_agent

    def checkpoint(self, folder: PathString) -> None:
        """Save the object to the Agents subdirectory of the given folder"""
        folder = os.path.join(folder, "Agents", f"{self.id}")
        os.makedirs(folder, exist_ok=True)

        tmp = {k: v for k, v in self.__dict__.items() if k != "model"}
        tmp["stats"] = tmp["stats"].toJSON()

        checkpoint_path = os.path.join(folder, "AgentHistory.json")
        with open(checkpoint_path, mode="w", encoding="utf8") as file:
            json.dump(tmp, file, cls=NumpyEncoder)

        self.model.checkpoint(os.path.join(folder, "Model.json"))

    def reload(self, folder: PathString) -> None:
        """Replace the data in this object with data from the given file"""
        checkpoint_path = os.path.join(folder, "AgentHistory.json")
        with open(checkpoint_path, mode="r", encoding="utf8") as file:
            data = json.load(file, cls=NumpyDecoder)

        for key, val in data.items():
            self.__dict__[key] = val

        self.stats = AgtHistoryStats()
        self.stats.fromJSON(data["stats"])

        self.model.reload(os.path.join(folder, "Model.json"))


################################################################################
## No Augment Wrapper - For using vanilla gym envs
################################################################################
class NoAugmentEnv:
    def __init__(self, env: Any) -> None:
```

```
565         self.env = env
566
567     def __getattr__(self, name: str) -> Any:
568         """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
569         if name.startswith("_"):
570             raise AttributeError(f"accessing private attribute '{name}' is prohibited")
571
572         if name in self.__dict__:
573             return self.__dict__[name]
574
575         return getattr(self.env, name)
576
577     def augment(self, env_params: EnvParams) -> None:
578         """Apply env parameters to the env (does nothing for this class)"""
579
580     def seed(self, seed: int) -> None:
581         """Seed the env (does nothing for this class)"""
582
583
584 ################################################################################
585 ## Convert Wrappers
586 ################################################################################
587 class SingleAgentGame:
588     """
589     Game wrapper
590
591     This sets up a wrapper to combine multiple agents to an input format for a game.
592     It also reshapes returns from the game for multiple agents.
593
594     Attributes
595     ----------
596     env : Gym environment
597         Gym environemnt to wrap.
598     rn : str
599         Role-name to give agent
600
601     Methods
602     -------
603     step(dict[role, action])
604         Pass agent actions to the environment, get step return
605     reset()
606         Reset the env, return initial obs for each role
607     """
608
609     def __init__(self, env: Any, rolename: Role = Role("role1")) -> None:
610         """
611         Init Function
612
613         Parameters
614         ----------
615         env : Gym environment
616         rolename : str
617             role for each agent?
618
619         Side-Effects
620         ------------
621         None
622
623         Returns
624         -------
625         None
626         """
627
628         # self.env = env
629         # if hasattr(env, "possible_agents"):
630         #     if len(env.possible_agents) > 1:
631         #         logger.info("Trying to apply wrapper to multiagent game!")
632         #     else:
633         #         self.rolename = env.possible_agents[0]
634         # else:
635         #     self.rolename = "default"
636         #     self.possible_agents = ["default"]
637
638         self.env = env
639         self.rolename = rolename
640         self.possible_agents = [self.rolename]
641         self.agents = [self.rolename]
642         self.observation_space = {self.rolename: env.observation_space}
643         self.action_space = {self.rolename: env.action_space}
644         if not hasattr(env, "render_mode"):
645             self.render_mode = "rgb_array"
```

```
646
647     def __getattr__(self, name: str) -> Any:
648         """
649         Returns an attribute with ``name``, unless ``name`` starts with an underscore.
650
651         Parameters
652         ----------
653         name : str
654             action name to peform
655
656         Side-Effects
657         ------------
658         Possibly
659             Environment performs action
660
661         Returns
662         -------
663         Possibly
664             Environment handles returns
665         """
666         # check if protected
667         if name.startswith("_"):
668             raise AttributeError(f"accessing private attribute '{name}' is prohibited")
669         # pass down to environment to handle
670         return getattr(self.env, name)
671
672     def step(self, actions: dict[str, FloatArray]) -> Iterator[dict[str, Any]]:
673         """
674         Take one step of environment.
675
676         The idea of this function is to get the action of every agent, pass it to the
677         environment, then reshape the return to match observations and rewards to
678         the appropriate agent.
679
680         This function currently only handles 1 agent, though with looping could be
681         setup to handle multiple.
682
683         Parameters
684         ----------
685         actions : dict[role, actVec]
686             Dictionary of actions for each agent
687
688         Side-Effects
689         ------------
690         Steps the environment
691
692         Returns
693         -------
694         list[dict[role, obsVec], dict[role, rewardVal], bool, "info"]
695             Returns a list of observations and rewards for every agent
696         """
697         # get "all" actions
698         action = list(actions.values())[0]
699
700         # pass action to environment, get returns
701         returns = self.env.step(action)
702
703         # reshape for each agent and return
704         return ({self.rolename: r} for r in returns)
705
706     def reset(self) -> dict[str, Any]:
707         """
708         Reset the Environment
709
710         Returns the appropriate observation vector
711
712         Parameters
713         ----------
714         None
715
716         Side-Effects
717         ------------
718         Resets the environment
719
720         Returns
721         -------
722         dict[role, obsVec]
723             Returns an observation vector for each role from the environment
724         """
725         # reset env, get default observations
726         returns = self.env.reset()
```

```
727             # reshape for each agent and return
728             return {self.rolename: returns}
729
730
731     class Gym_to_Gymnasium:
732         """Wrapper to convert Gym env to Gymnasium"""
733
734         def __init__(self, env: Any) -> None:
735             self.env = env
736
737         def __getattr__(self, name: str) -> Any:
738             """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
739             if name.startswith("_"):
740                 raise AttributeError(f"accessing private attribute '{name}' is prohibited")
741
742             if name == "observation_spaces":
743                 return self.env.observation_space
744
745             if name == "action_spaces":
746                 return self.env.action_space
747
748             if name in self.__dict__:
749                 return self.__dict__[name]
750
751             return getattr(self.env, name)
752
753         def step(self, *args: Any, **kwargs: Any) -> Any:
754             """Perform step and return gymnasium format results"""
755             obs, reward, done, info = self.env.step(*args, **kwargs)
756             return (obs, reward, done, done, info)
757
758         def reset(self, *args: Any, **kwargs: Any) -> tuple[Any, dict[str, Any]]:
759             """Perform reset and return gymnasium format results"""
760             obs = self.env.reset()
761             return obs, {}
762
763         def observation_space(self, agent: Role) -> FloatArray:
764             """Return the observation space for the given agent"""
765             return self.env.observation_space[agent]  # type: ignore  # from base class
766
767         def action_space(self, agent: Role) -> FloatArray:
768             """Return the action space for the given agent"""
769             return self.env.action_space[agent]  # type: ignore  # from base class
770
771
772     class Gymnasium_To_GymEnv:
773         """
774         Wrapper to convert Gymnasium env's to gym envs. Legacy.
775         """
776
777         def __init__(self, env: Any) -> None:
778             self.env = env
779
780         def __getattr__(self, name: str) -> Any:
781             """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
782             if name.startswith("_"):
783                 raise AttributeError(f"accessing private attribute '{name}' is prohibited")
784
785             if name == "observation_space":
786                 return self.env.observation_spaces
787
788             if name == "action_space":
789                 return self.env.action_spaces
790
791             if name in self.__dict__:
792                 return self.__dict__[name]
793
794             return getattr(self.env, name)
795
796         def step(
797             self, *args: Any, **kwargs: Any
798         ) -> tuple[FloatArray, float, bool, dict[str, Any]]:
799             obs, reward, term, _, info = self.env.step(*args, **kwargs)
800             return (obs, reward, term, info)
801
802         def reset(self, *args: Any, **kwargs: Any) -> Any:
803             obs, _ = self.env.reset()
804             return obs
805
806
807     ################################################################################
```

```python
## Tianshou Wrappers
################################################################################
## Taken from Petting Zoo:
     https://github.com/Farama-Foundation/PettingZoo/blob/master/pettingzoo/utils/agent_selector.py
class MP_AEC_to_Parallel(PettingZooEnv):  # type: ignore[misc] # PettingZooEnv is Any
    def __init__(self, env: Any) -> None:
        self.env = env

    def __getattr__(self, name: str) -> Any:
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
        if name.startswith("_"):
            raise AttributeError(f"accessing private attribute '{name}' is prohibited")

        return getattr(self.env, name)

    def _observe(self) -> dict[Role, Any]:
        obs = dict()
        for agt in self.env.agents:
            obs[agt] = self.env.observe(agt)["observation"]

        return obs

    def reset(self, close: bool = False) -> tuple[Any, Any]:
        r = self.env.reset()

        if not r:
            obs = self._observe()

        self.term = {a: False for a in self.env.agents}
        self.trunc = {a: False for a in self.env.agents}

        return (obs, None)

    def _last(self, agent: Role) -> tuple[Any, float, bool, bool, dict[str, Any]]:
        """Returns observation, cumulative reward, terminated, truncated, info for the current agent
            (specified by self.agent_selection)."""
        observation = self.env.observe(agent)["observation"]
        return (
            observation,
            self.env._cumulative_rewards[agent],
            self.env.terminations[agent],
            self.env.truncations[agent],
            self.env.infos[agent],
        )

    def step(
        self, action: dict[str, Any]
    ) -> tuple[
        dict[str, Any],
        dict[str, float],
        dict[str, bool],
        dict[str, bool],
        dict[str, dict[str, Any]]],
    ]:
        # agt = next(self.a_itter)
        agt = self.env.agent_selection
        # print("Curent Agent:?", agt)
        # print(action)
        self.env.step(action[agt])
        obss = dict()
        rewards = dict()
        terms = dict()
        truncs = dict()
        infos = dict()

        for a in self.env.agents:
            obs, reward, term, trunc, info = self._last(a)
            obss[a] = obs
            rewards[a] = reward
            terms[a] = term
            truncs[a] = trunc
            infos[a] = info

        return obss, rewards, terms, truncs, infos

    def render(self, close: bool = False) -> Union[None, RenderFrame]:
        if close:
            return None
        else:
            # since env isn't defined, we can't tell what this returns
            return self.env.render()  # type: ignore[no-any-return]
```

```
887
888
889 class MP_PettingZooEnv(PettingZooEnv):  # type: ignore[misc] # PettingZooEnv is Any
890     def __init__(
891         self, setup_args: argparse.Namespace, *args: Any, **kwargs: Any
892     ) -> None:
893         self.setup_args = setup_args
894
895         self.check_max_steps_per_ep = False
896         if "max_steps_per_ep" in setup_args.tianshou.keys():
897             self.check_max_steps_per_ep = True
898             self.max_steps_per_ep = setup_args.tianshou["max_steps_per_ep"]
899
900         super().__init__(*args, **kwargs)
901
902     def __getattr__(self, name: str) -> Any:
903         """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
904         if name.startswith("_"):
905             raise AttributeError(f"accessing private attribute '{name}' is prohibited")
906
907         if name in self.__dict__:
908             return self.__dict__[name]
909
910         return getattr(self.env, name)
911
912     def augment(self, env_params: EnvParams) -> None:
913         """Apply env parameters to the env"""
914         self.env.augment(env_params)
915
916     def seed(self, seed: Optional[int] = None) -> None:
917         """Seed the env"""
918         self.env.seed(seed)
919
920     def step(self, *args: Any, **kwargs: Any) -> Any:
921         """Do step, returning output in petting zoo format"""
922         result = super().step(*args, **kwargs)
923         self.steps += 1
924         if self.check_max_steps_per_ep:
925             if self.steps == self.max_steps_per_ep:
926                 result = (result[0], result[1], result[2], True, result[4])
927
928         return result
929
930     def reset(self, *args: Any, **kwargs: Any) -> Any:
931         self.steps = 0
932         r = super().reset(*args, **kwargs)
933         self.env.seed(0)
934         return r[0], r[1]
935
936
937 def MP_parallel_to_aec(par_env: Any) -> aec_to_parallel_wrapper:
938     if isinstance(par_env, aec_to_parallel_wrapper):
939         return par_env.aec_env
940     aec_env = MP_parallel_to_aec_wrapper(par_env)
941     ordered_env = MP_OrderEnforcingWrapper(aec_env)
942     return ordered_env
943
944
945 class MP_parallel_to_aec_wrapper(parallel_to_aec_wrapper):  # type: ignore[misc] #
        parallel_to_aec_wrapper is Any
946     def __getattr__(self, name: str) -> Any:
947         """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
948         if name.startswith("_"):
949             raise AttributeError(f"accessing private attribute '{name}' is prohibited")
950
951         if name in self.__dict__:
952             return self.__dict__[name]
953
954         return getattr(self.env, name)
955
956     def augment(self, env_params: EnvParams) -> None:
957         self.env.augment(env_params)
958
959     def seed(self, seed: int) -> None:
960         self.env.seed(seed)
961
962     # If the system is failing beceause it can't convert a list to an action, try
963     # uncommenting this code.
964     def step(self, action: FloatArray) -> None:
965         # I don't know why this is here...
966         # if action is not None:
```

```
        #     action = int(action)
        if (
            self.terminations[self.agent_selection]  # type: ignore  # types from base class
            or self.truncations[self.agent_selection]  # type: ignore  # types from base class
        ):
            del self._actions[self.agent_selection]  # type: ignore  # types from base class
            # TODO: this may be wrong
            assert action is not None
            self._was_dead_step(action)
            return

        self._actions[self.agent_selection] = action

        if self._agent_selector.is_last():
            obss, rews, terminations, truncations, infos = self.env.step(self._actions)

            self._observations = copy.copy(obss)
            self.terminations = copy.copy(terminations)
            self.truncations = copy.copy(truncations)
            self.infos = copy.copy(infos)
            self.rewards = copy.copy(rews)
            self._cumulative_rewards = copy.copy(rews)

            env_agent_set = set(self.env.agents)

            self.agents = self.env.agents + [
                agent
                for agent in sorted(self._observations.keys())
                if agent not in env_agent_set
            ]

            if len(self.env.agents):
                self._agent_selector = agent_selector(self.env.agents)
                self.agent_selection = self._agent_selector.reset()

            self._deads_step_first()
        else:
            if self._agent_selector.is_first():
                self._clear_rewards()

            self.agent_selection = self._agent_selector.next()


class MP_OrderEnforcingWrapper(OrderEnforcingWrapper):  # type: ignore[misc] # OrderEnforcingWrapper is
    Any
    def __getattr__(self, name: str) -> Any:
        """Returns an attribute with ``name``, unless ``name`` starts with an underscore."""
        if name.startswith("_"):
            raise AttributeError(f"accessing private attribute '{name}' is prohibited")

        if name in self.__dict__:
            return self.__dict__[name]

        return getattr(self.env, name)

    def augment(self, env_params: EnvParams) -> None:
        """Apply env parameters to the env"""
        self.env.augment(env_params)

    def seed(self, seed: int) -> None:
        """Seed the env"""
        self.env.seed(seed)
```