Balloon Learning Environment

The Balloon Learning Environment Authors

Dec 19, 2022

GETTING STARTED:

Note: We are working hard to improve the Balloon Learning Environment documentation. Please let us know if there is a page you'd like to see here!

The Balloon Learning Environment (BLE) is a simulator for stratospheric balloons. It is designed as a challenge environment for deep reinforcement learning algorithms.

CHAPTER

ONE

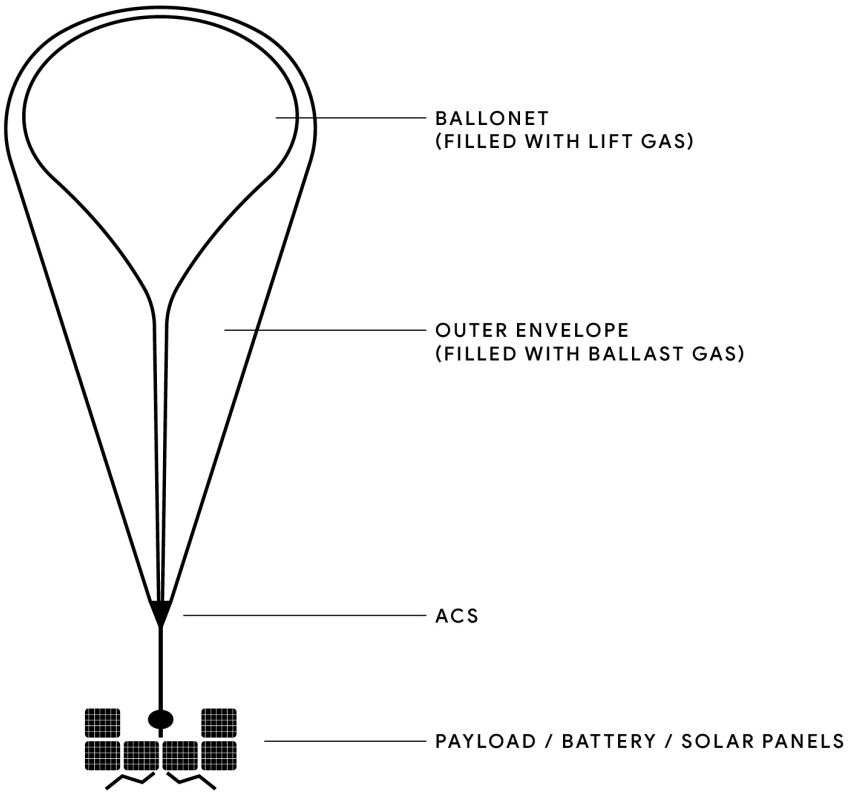
ABOUT THE BALLOON LEARNING ENVIRONMENT

This page gives a simple background into the problem of flying stratospheric balloons and station keeping. This is intended to be an introduction, and doesn't discuss all the nuances of the simulator or real-world complexities.

1.1 Stratospheric Balloons

The type of balloon that we consider in the BLE is pictured below. These balloons have an outer envelope and an inner envelope (ballonet). The ballonet is filled with a buoyant gas (for example, helium). The outer envelope contains air which acts as ballast. An Altitude Control System (ACS) is able to pump air in or out of the envelope. When air is pumped into the balloon the average density of the gases in the envelope increases, which in turn makes the balloon descend. On the other hand, if air is pumped out of the balloon then the average gas density decreases, resulting in the balloon ascending.

The balloons are also equipped with a battery to power the ACS, and solar panels to recharge the batteries. Power is used by communication systems on the balloon, so battery power is constantly draining when the solar panels aren't in use. This means that the balloons in the BLE are constrained at night—they cannot constantly use the ACS without running out of power. The energy required for running the ACS is asymmetric; the energy needed to release air from the envelope (ascend) is negligable.



1.2 Navigating a Windfield

The balloons in the BLE have no way of moving themselves laterally. They are only capable of moving up, down, or staying at the current altitude. To navigate they must "surf" the wind to get where they need to go. The balloons are flown in a 4d windfield, where each different x, y, altitude, time position gives a different wind vector. A balloon must learn how to navigate this windfield to achieve its goal.

1.3 Station-Keeping

The goal of station-keeping is to remain within a fixed distance of a ground station. This distance is only measured in the x, y plane, i.e. the altitude of the balloon is not taken into account. In the BLE, the task is to keep a balloon within 50km of the ground station. We call the proportion of time that a balloon remains within 50km of the ground station TWR50 (for Time Within a Radius of 50km). In the BLE, each episode lasts two days.

1.4 Failure Modes

A balloon can have a critical failure by running out of power, flying too low, or having a superpressure that is too high or too low. Each of these are partially protected by a safety layer, but in extreme conditions there can still be critical failures.

CHAPTER

TWO

GETTING STARTED

2.1 Installation

Note: The Balloon Learning Environment requires python >= 3.7.

To get started with the Balloon Learning Environment, install the package with pip. First, ensure your pip version is up to date:

```
pip install --upgrade pip
```

and then install the balloon_learning_environment package:

```
pip install balloon_learning_environment
```

To install with the acme package, run:

```
pip install balloon_learning_environment[acme]
```

To install from GitHub directly, run the following commands from the root directory where you cloned the repository:

```
pip install .[acme]
```

2.2 Ensure the BLE is Using Your GPU/TPU

The BLE contains a VAE for generating winds, which you will probably want to run on your accelerator. See the jax documentation for installing with GPU or TPU.

As a sanity check, you can open interactive python and run:

```
from balloon_learning_environment.env import balloon_env
env = balloon_env.BalloonEnv()
```

If you are not running with GPU/TPU, you should see a log like:

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and␣
↪rerun for more info.)
```

If you don't see this log, you should be good to go!

2.3 Training Baseline Agents

Once the Balloon Learning Environment has been installed you can start training the baseline agents. The BLE has a training script for the baseline agents that you can call with

```
python -m balloon_learning_environment.train \
  --base_dir=/tmp/ble/train \
  --agent=finetune_perciatelli \
  --renderer=matplotlib
```

Using the matplotlib renderer allows you to see in real time how the agent is performing. However, the renderer also has a significant overhead. We recommend using *–renderer=None*, which is the default, for speed.

Other agents you could train are:

- *dqn*: A simple DQN agent. This is not a strong baseline, and is mostly to demonstrate that solving the BLE is not a simple task!
- *quantile*: A Quantile-based agent in JAX that uses the same architecture as Perciatelli44, but starts training from a fresh initialization.
- *finetune_perciatelli*: The same as *quantile*, but reloads the *perciatelli44* weights, and is a great way to warm-start agent training.

For more options for using the train script, see the flags at the top of the train.py file.

2.4 Evaluating Baseline Agents

The BLE also comes with an evaluation suite. This lets us run our agent on a large number of environment seeds and aggregate the results. To run an evaluation suite on a benchmark agent, use the following example command:

```
python -m balloon_learning_environment.eval.eval \
  --output_dir=/tmp/ble/eval \
  --agent=random \
  --suite=micro_eval \
  --renderer=matplotlib
```

This will evaluate the random agent on 1 seed and write the result to */tmp/ble/eval* as a json file. This file can be loaded in the summarize_eval notebook to summarize statistics about the flight.

Other agents to evaluate (including agents mentioned above) are:

- *perciatelli44*: A state-of-the-art learned agent reported in "Autonomous navigation of stratospheric balloons using reinforcement learning".
- *station_seeker*: A rule-based agent that achieves good performance, also reported in "Autonomous navigation of stratospheric balloons using reinforcement learning".

You can also try evaluation on other suites:

- *big_eval*: This suite containes 10,000 seeds and gives a good signal of how well an agent station-keeps. However, this suite may take up to 300 hours on a single GPU! We suggest splitting the work up over multiple shards using the *shard_idx* and *num_shards* flags.
- *small_eval*: This is a very useful evaluation suite to run. It contains 100 seeds and gives a rough view into how well an agent performs. On abui single GPU, it may take around 3 hours.

For more options for evaluation runs, see eval.py. For more available suites, see suites.py.

2.5 Training Acme Agents

The BLE also includes examples for training acme agents, but these use their own binary. To train the QR-DQN agent with acme on a single-GPU machine, use the following command:

```
python -m balloon_learning_environment.train_acme_qrdqn
```

Experimental: To run distributed training of acme's QR-DQN with Google cloud's Vertex AI, first follow the instructions here, and then run the following command:

```
python -m balloon_learning_environment.distributed_train_acme_qrdqn \
  --num_actors=10 \
  --lp_launch_type=vertex_ai
```

CHAPTER

THREE

TRAINING AND EVALUATING A NEW AGENT

There are two options for training/evaluating a new agent:

1. Create an agent following the Agent interface and use our scripts.
2. Use the Balloon Learning Environment gym environment in your own framework.

We recommend following our agent interface for single GPU training. For more complicated use cases, such as distributed training, it may be easier to use the gym environment in your own framework.

3.1 Using the Agent Interface

To make a new agent using our framework, you should do the following:

1. Create an agent following the Agent interface.
2. Create a training script that uses train_lib.run_training_loop.
3. Create an evaluation script that uses eval_lib.eval_agent.

3.1.1 Creating an Agent

First, create an agent following the Agent interface. For example:

```
from typing import Sequence
from balloon_learning_environment.agents import agent

class MyNewAgent(agent.Agent):
    def __init__(self, num_actions: int, observation_shape: Sequence[int]):
        super(MyNewAgent, self).__init__(num_actions, observation_shape)

    ...
```

Make sure to override all the functions required and recommended by the Agent There are several good examples in the agents module.

Alternatively, you can use one of the agents provided with the BLE:

```
from balloon_learning_environment import train_lib
from balloon_learning_environment.env import balloon_env
from balloon_learning_environment.utils import run_helpers

agent_name = 'quantile'
```

(continues on next page)

(continued from previous page)

```
env = gym.make('BalloonLearningEnvironment-v0')
run_helpers.bind_gin_variables(agent_name)
agent = run_helpers.create_agent(agent_name,
                                 env.action_space.n,
                                 env.observation_space.shape)
```

3.1.2 Create Training Script

Once you have created your agent, it should be ready to train by calling train_lib.run_training_loop. You'll need to create a launch script that sets up the environment and agent and calls this function.

For an example, take a look at our train.py which we use to train the benchmark agents. A slimmed down version would look something like this:

```
from balloon_learning_environment import train_lib
from balloon_learning_environment.env import balloon_env  # Registers the environment.
import gym

env = gym.make('BalloonLearningEnvironment-v0')
agent = YourAgent(env.action_space.n, env.observation_space.shape)

train_lib.run_training_loop(
    '/tmp/ble/train/my_experiment',  # The experiment root path.
    env,
    agent,
    num_iterations=2000,
    max_episode_length=960,  # 960 steps is 2 days, the default amount.
    collector_constructors=[])  # Specify some collectors to log training stats.
```

You can optionally add Collectors to generate statistics during training. We include a set of collectors to

1. print statistics to the console with ConsoleCollector.
2. save statistics to a pickle file with PickleCollector.
3. write to Tensorboard event files with TensorboardCollector.

You can create your own by following the Collector interface and passing its constructor to the CollectorDispatcher.

3.1.3 Create Evaluation Script

If your agent follows the Agent interface, you can also make use of eval_lib.eval_agent. Once again, you'll need to create a launch script that sets up the environment and agent, and then calls the function.

For an example, take a look at our eval.py which we use to evaluate the benchmark agents. A slimmed down version would look something like this:

```
from balloon_learning_environment.env import balloon_env  # Registers the environment.
from balloon_learning_environment.eval import eval_lib
import gym

env = gym.make('BalloonLearningEnvironment-v0')
agent = YourAgent(env.action_space.n, env.observation_space.shape)
```

(continues on next page)

(continued from previous page)

```
 eval_results = eval_agent(
     agent,
     env,
     eval_suite=suites.get_eval_suite('small_eval'))

do_something_with_eval_results(eval_results)  # Write to disk, for example.
```

'small_eval' uses 100 seeds, which may take around 3 GPU hours, depending on the GPU. 'small_eval' is great for determining the progress of an agent. Once you are satisfied with an agent, we recommend reporting 'big_eval' results where feasible. 'big_eval' uses 10,000 seeds, and takes around 300 GPU hours. This work can be parallelized and spread out across multiple shards, as we demonstrate in eval.py.

3.2 Using a Different Framework

If you choose to use a different framework for training an agent, simply create an environment and interact with it in the way that makes sense for your framework or experiment.

```
from balloon_learning_environment.env import balloon_env  # Registers the environment.
import gym

env = gym.make('BalloonLearningEnvironment-v0')
# Do what you want with the environment now it has been created.
```

The environment follows the standard gym interface. The type of the returned environment object is BalloonEnv.

CHAPTER

FOUR

USING AND CONFIGURING THE ENVIRONMENT

4.1 Basic Usage

The main entrypoint to the BLE for most users is the gym environment. To use the environment, import the balloon environment and use gym to create it:

```
import balloon_learning_environment.env.balloon_env  # Registers the environment.
import gym

env = gym.make('BalloonLearningEnvironment-v0')
```

This will give you a new BalloonEnv object that follows the gym environment interface. Before we run the environment we can inspect the observation and action spaces:

```
>>> print(env.observation_space)
Box([0. 0. 0. ... 0. 0. 0.], [1. 1. 1. ... 1. 1. 1.], (1099,), float32)
>>> print(env.action_space)
Discrete(3)
```

Here we can see that the observation space is a 1099 dimensional array, and the action space has 3 discrete actions. We can use the environment as follows:

```
env.seed(0)
observation_0 = env.reset()
observation_1, reward, is_terminal, info = env.step(0)
```

In this snippet, we seeded the environment to give it a deterministic initialization. This is useful for replicating results (for example, in evaluation), but most of the time you'll want to skip this line to have a random initialization. After seeding the environment we reset it and stepped once with action 0.

We expect the observations to have the shape specified by observation_space:

```
>>> print(type(observation_0), observation_0.shape)
<class 'numpy.ndarray'> (1099,)
```

The reward, is_terminal, and info objects are as follows:

```
>>> print(reward, is_terminal, info, sep='\n')
0.26891435801077535
False
{'out_of_power': False, 'envelope_burst': False, 'zeropressure': False, 'time_elapsed':␣
↪datetime.timedelta(seconds=180)}
```

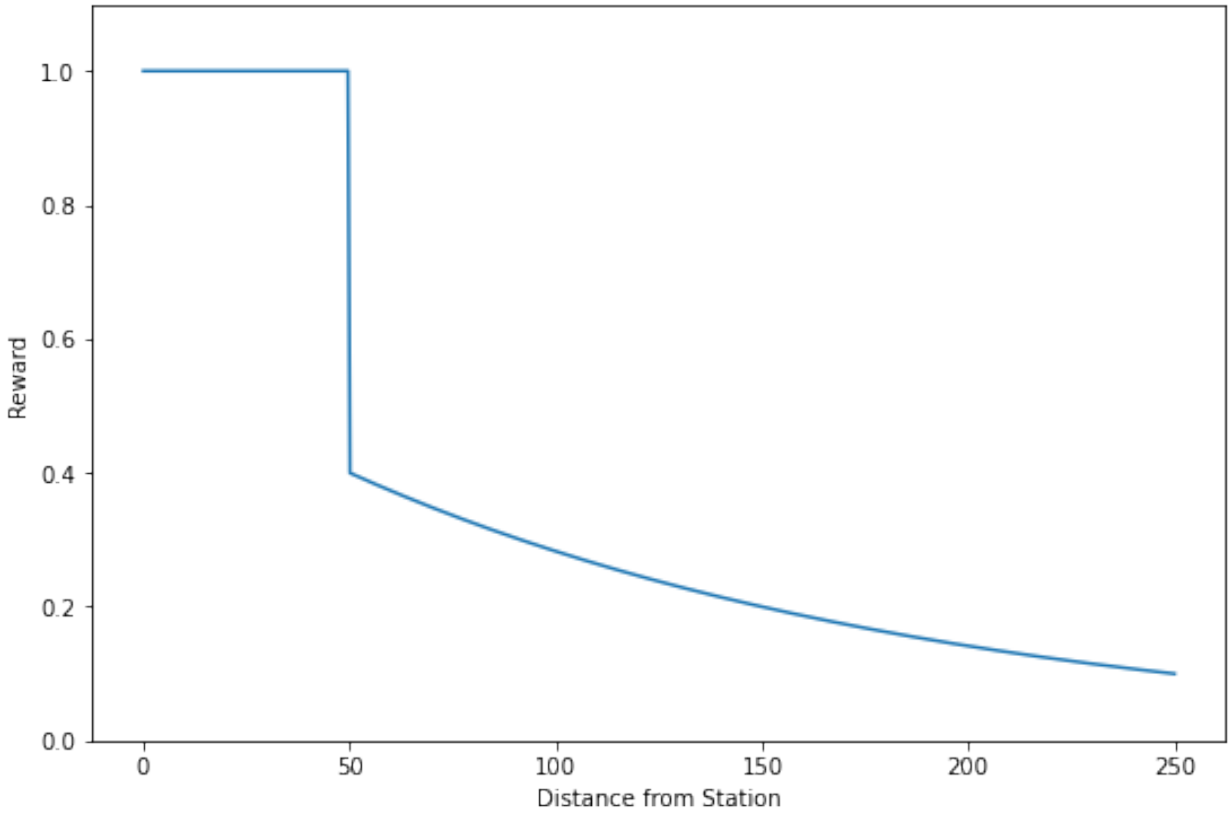
These should be enough to start training an RL agent.

4.2 Configuring the Environment

The environment may be configured to give custom behavior. To see all options for configuring the environment, see the BalloonEnv constructor. Here, we highlight important options.

First, the FeatureConstructor class may be swapped out. The feature constructor receives observations from the simulator at each step, and returns features when required. This setup allows a feature constructor to maintain its own state, and use the simulator history to create a feature vector. The default feature constructor is the PerciatelliFeatureConstructor.

The reward function can also be swapped out. The default reward function, perciatelli_reward_function gives a reward of 1.0 as long as the agent is in the stationkeeping readius. The reward decays exponentially outside of this radius.



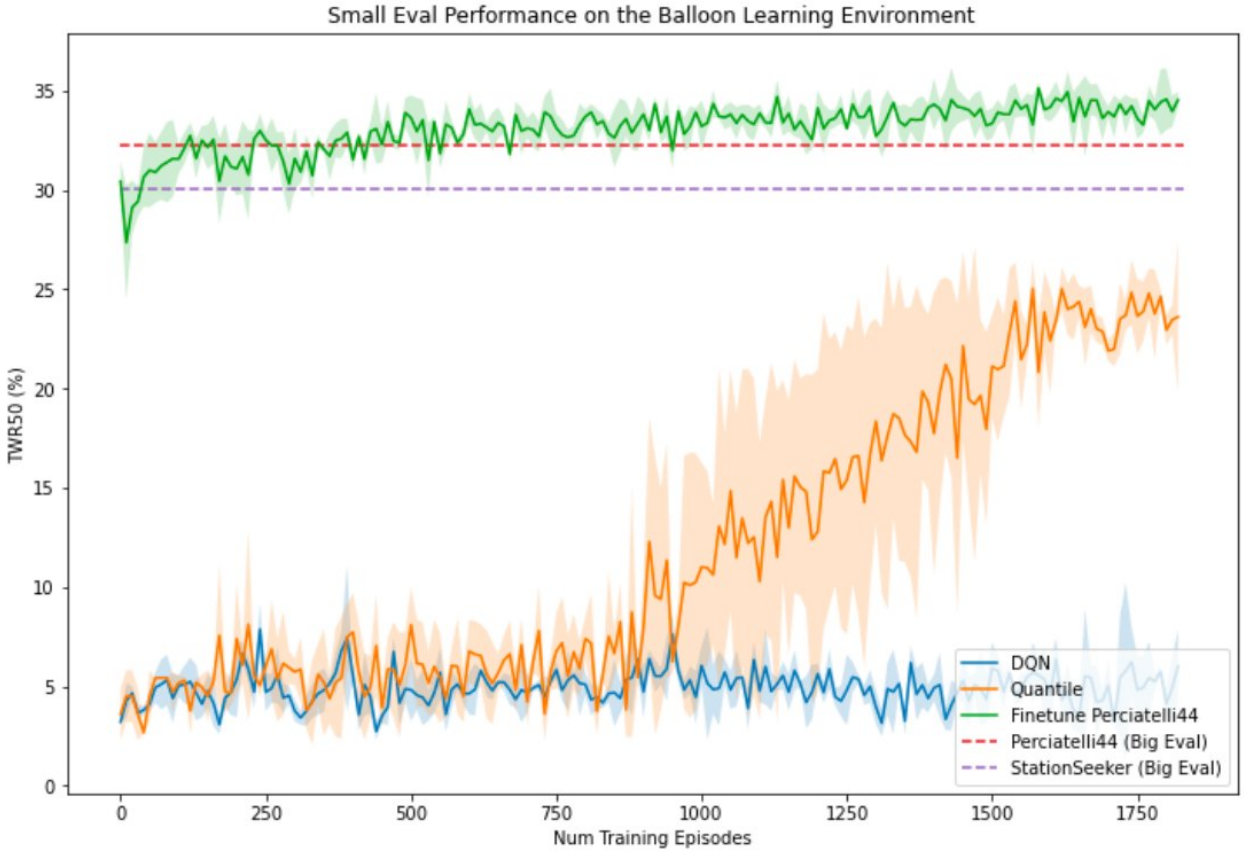
CHAPTER

FIVE

BENCHMARK RESULTS

This page will be udpated soon with even more data

The following graph shows evaluation results on the "small eval" evaluation suite throughout training for the DQN, quantile, and finetuned Perciatelli44 agents. The horizontal lines are evaluation results for "big eval" for Perciatelli44 and station seeker.



CHAPTER

SIX

CHANGE LIST

6.1 v1.0.2

- Added GridBasedWindField and GenerativeWindFieldSampler.
- Deprecated GenerativeWindField in favor of GridBasedWindField and GenerativeWindFieldSampler.
- Bug fix: previously the train script would load the latest checkpoint when restarting but then resume from the previous iteration. It is now fixed, so if reloading checkpoint i, the agent will continue working on iteration i + 1.
- Vectorized wind column feature calculations. This gives about a 16% speed increase.
- Cleanups in balloon.py, by removing staticfunctions.
- Moved wind field creation to run_helpers, since it is common for training and evaluation.
- Added a flag for calculating the flight path to eval_lib, to allow for faster evaluation if you don't need flight paths.
- Improvements to AcmeEvalAgent by making it more configurable.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search