PACS PROJECT REPORT

REINFORCEMENT LEARNING FOR ICD DEVICES

ANDREA ZANIN AND STEFANO PAGANI

1. Background

In the last years there has been a growing research interest in training agents to perform sequential decision making for medical applications; for example to control Implantable Cardioverter-Defibrillators (ICDs) [1] or to choose ablation sites [2] in the heart. These tasks require the agents to perform multiple subsequent decisions, whose effects are delayed in time; for this kind of tasks Reinforcement Learning (RL) is the state of the art approach and it is already widely used in other fields (e.g. autonomous driving).

1.1. **Reinforcement Learning**

Any reinforcement learning problem can be represented with the Agent-Environment model: the agent observes a state from the environment, does an action and receives a reward from the environment.

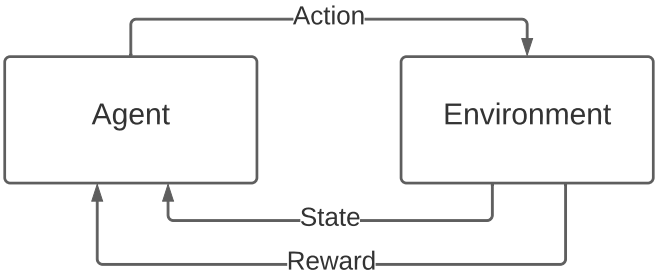

FIGURE 1. A representation of the Agent-Environment model.

The interaction between agent and environment happens at discrete time steps and the goal of the agent is maximizing the return, which frequently is the sum of all the rewards (the formulas below assume this is the case). The reward function $r(s, a)$ (s is the state of the environment and a is the action of the agent) is typically a design choice of the model, for example to train an agent to control an ICD we can provide a negative reward every time the agent releases an electric shock and a positive feedback at the end of the simulation if regular heart rhythm was restored.

1.2. **Optimal policy**

We call state space $\mathcal{S}$ the set of all possible environment states, action space $\mathcal{A}$ the set of all possible actions and policies all the possible functions $\pi : \mathcal{S} \to \mathcal{A}$. Our goal is finding the optimal policy $\pi^\star$, i.e. the policy which maximizes the expected return.
The Bellman optimality equations (Equation 1) provide a implicit expression for $\pi^\star$ provided that we know the reward function $r(s, a)$ and the probability $p(s' \mid s, a)$ of transitioning from state s to state s' after taking action a.

$$\begin{aligned} \pi^\star(s) &= \arg\max_a Q^\star(s,a) \\ Q^\star(s,a) &= Q_{\pi^\star}(s,a) = r(s,a) + \sum_{s' \in \mathcal{S}} p(s' \mid s,a) \max_{a'} Q^\star(s',a') \end{aligned} \tag{1}$$

The Q_π function is the action-value function of the policy π: it provides the expected return of taking action a at state s and then following policy π.

These equations can be solved with a policy iteration algorithm: start from an arbitrary initial policy and iteratively improving it until convergence. This process however requires multiple evaluations of all the possible states, which is only feasible for problems with small state spaces.

1.3. **Deep Q-Learning**

The state spaces for medical tasks are usually very large or infinite and the transition probabilities are not known in analytic form; in this situation the best approach is training a neural network to approximate the optimal policy.

In medical applications data from real patients is usually scarce and the agent can't be allowed to interact with real patients during training, due to obvious safety concerns; for this reason the agents are trained in a simulated environment. These environment simulations are very computationally expensive, so we want to use as little of them as needed.

In our implementation we chose the Deep Q-Learning (DQL) algorithm [3], which learns an approximation of $Q^\star$ observing many agent-environment interactions. To use DQL we don't need to know the state transition probabilities, because they are implicitly estimated from the observed interactions. Furthermore DQL is an off-policy algorithm, which means that it can learn the optimal policy even if all the agent-environment interactions use a different policy; this allows us to update the neural network and keep all the previously simulated interactions in the training dataset.

Training a RL agent for medical applications thus requires implementing the neural network model, implementing the environment simulation and allowing the two to interact.

2. Project goal

The goal of this project was developing a library to simplify the development and training of reinforcement learning agents interacting with a simulated physical environment. We identified tensorflow's tf-agents library [4] as the most widely used library in the reinforcement learning field; as far as physical environments are concerned we chose FENiCSx [5] as library to perform the numerical simulations.

2.1. **Problem**

The numerical simulations can be very computationally intensive and thus need to be executed in a distributed environment; FENiCSx achieves this using MPI to handle the low-level coordination of the processes. The tf-agents library on the other hand is not designed to work with MPI, this means that using it in conjunction with FENiCSx is not straightforward. Two key problems arise if we run tf-agent's code on all processes and use MPI to distribute the FENiCSx computation:

- tf-agent's internals and many of the libraries in its ecosystem have non-deterministic behaviour (e.g. usage of random number generators), to guarantee correctness we must ensure that this behaviour is identical across all processes. Doing this is often non-trivial or not at all possible, e.g. if the library does not allow seeding the RNG.
- the training of the agent is computationally expensive, so it is wasteful to do it on every process independently.

On top of solving those problems, we wanted to reduce the amount of boilerplate code needed to handle data collection and training of the agent, thus allowing faster iteration on research ideas.

2.2. **Solution**

We designed a system that ensures all of tf-agent's computation is done on a single process (the leader process) and the results of the computation are then replicated across all processes. This approach ensures that non-deterministic behaviour is identical in all processes and that no redundant computation is performed.

The system can be extended to allow distributing tf-agent's computation across several processes, with the only limitation that the only process (or thread if using python's async capabilities) that runs both tf-agent's computation and FENiCSx computation is the leader process. We deem this to be an acceptable limitation, because the overhead of the extra processes needed by this approach is tiny.

The classes of the physics-rl library have been structured to allow developers to use just a subset of the functionalities without necessarily using all the library. Those functionalities are organized in modules:

- **`coordinated.py` module** (Section 3): tools to handle a mix of distributed computation and single-node computation.
- **`fem.py` module** (Section 4): ready to use simulation of two physical environments governed by the heat diffusion equation (`HeatEnvironment` class) and by a monodomain Mitchell-Schaeffer model (`MonodomainMitchellSchaeffer` class). These environments use the `coordinated.py` module to distribute computation.
- **`dojo.py` module** (Section 5): a class that handles data collection and training using user-provided agents and environments with reasonable opinionated defaults (`Dojo` class).

In Figure 2 are represented the interactions between the various modules: the `fem.py` module contains the implementations of the simulated environments, which use the `coordinated.py` module internally, the `dojo.py` module manages the training of the agent (defined using `tf-agents`) and in particular the interactions between agent and environment.

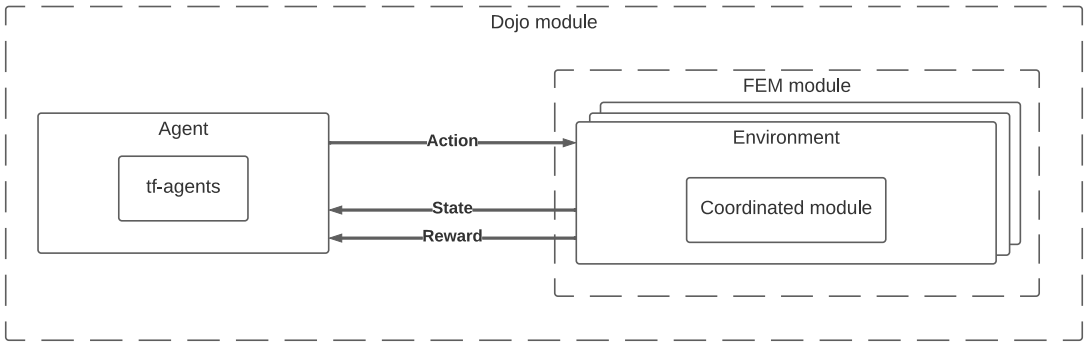

FIGURE 2. A representation of the interaction between the various modules in the library.

3. Coordinated module

The `coordinated.py` module implements the system that handles the coordination of the mix of distributed and single-process computation needed to use reinforcement learning in a simulated physical environment.

The implementation is based on two classes: `Coordinator` and `Coordinated`; the former will be instantiated as is in most cases, while the latter is meant to be used as base class to implement distributed computations.

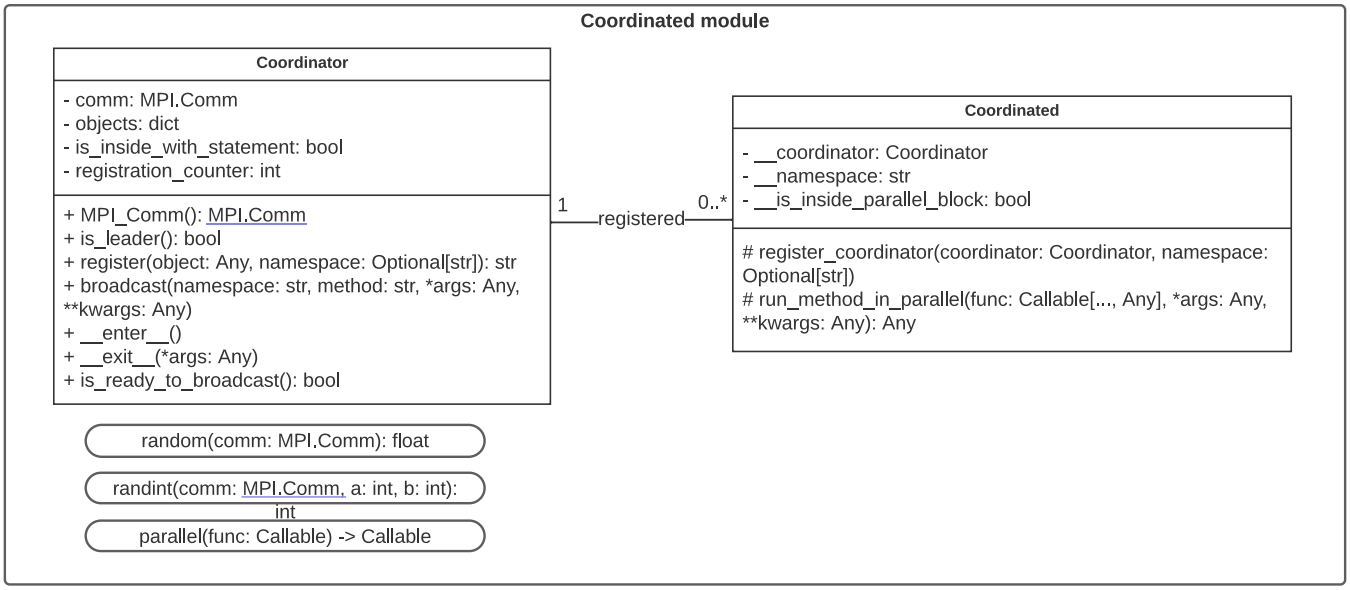

FIGURE 3. UML diagram of the Coordinated module

3.1. Module usage

Any class that needs to run distributed FENiCSx computation should inherit from the `Coordinated` class and the methods that need to be run on all processes should be marked with the `@parallel` decorator. The result is a class that can be used in the leader process as if it was running in a single process (thus allowing compatibility with tf-agent's ecosystem), while still running on all processes the necessary computations.

All processes should have a `Coordinator` instance; in most cases there should be exactly one per process, but in some cases more than one may be necessary (see Section 3.4.1). All instances inheriting from `Coordinated` must be registered in the `Coordinator`, the corresponding instances of the same class in different processes are registered in a common namespace (the namespace can be manually set, but in most cases it is inferred automatically). This allows a function call in an instance in the leader process to be replicated on the correct instances in the follower processes.

3.2. Method call replication

The system that guarantees that a method call on the leader instance is replicated on all follower instances is the following:

1. The `@parallel` decorator wraps the method implementation so that when the method is called the `Coordinator` is also notified (deriving from `Coordinated` is necessary so that the decorator can get access to the `Coordinator` instance).
2. When the leader `Coordinator` is notified of the method call it broadcasts the method name, the method arguments and the instance namespace to the Coordinators of the other processes.
3. When a follower `Coordinator` receives the method call broadcast it calls the method with the received name on the instance in the received namespace.

When a method A decorated with `@parallel` calls another method B also decorated with `@parallel` the algorithm stated above would lead to duplicate calls to B on follower processes: the method is called both by method A running in the follower process and by the broadcasted B call coming from the leader process. For this reason the `Coordinated` class tracks whether a parallel method is being executed and in that case it does not broadcast nested calls.

The follower processes should be waiting for broadcast messages coming from the leader process, furthermore they should shutdown when the leader process terminates; this is all handled using the `with coordinator` construct.

3.3. Coordinated class

The `Coordinated` class exposes a method `register_coordinator` that derived classes should call in the constructor to register the instance in the `Coordinator`. This behaviour could have been implemented accepting a field `coordinator` in the constructor of `Coordinated`, but this would have complicated the use of multiple inheritance for the library users; we expect most classes that derive from Coordinated to also derive from tf-agent's PyEnvironment, so we chose to implement the `register_coordinator` method instead of the alternative solution.
This method initializes the `__coordinator` and `__namespace` variables which will be used by `run_method_in_parallel` to call the `Coordinator.broadcast` method.

The other method implemented by `Coordinated` is `run_method_in_parallel` which executes the provided function in parallel on all the processes. The module also provides a decorator `parallel` which can be used on any method of a class deriving from `Coordinated` to wrap that method in a call to `run_method_in_parallel`; in most cases this is the suggested way to use `run_method_in_parallel`.

The method `run_method_in_parallel` checks that the instance has been setup correctly, which means that `register_coordinator` has been previously called and the `Coordinator` is ready to broadcast (checked using the `Coordinator.is_ready_to_broadcast` method).
After the checks are successful the `run_method_in_parallel` method runs the provided function and if needed broadcasts the call to all other processes using the `Coordinator.broadcast` method. The call is broadcasted only if the process is the leader process (i.e. `rank=0`) and there isn't another call to `run_method_in_parallel` in the call stack, the latter constraint ensures that the call isn't broadcasted multiple times. The latter check is implemented by setting `__is_inside_parallel_block = True` before broadcasting and switching it back to `False` after running the function.

3.4. **Coordinator class**

Each `Coordinator` instance stores a dictionary `objects` that matches namespaces (which are just convenience names) with object instances, this allows the `Coordinator` instances on different processes to refer to corresponding objects using their namespace.

The `register` method associates an instance to a namespace by saving that key-value pair in the `objects` dictionary; if the method `register` is called with corresponding instances in the same order in all processes (as happens in most cases) then you don't need to specify the namespace, because the implementation of `register` automatically uses an incrementing integer as namespace.

The `broadcast` method can be called only from the leader process (i.e. `rank=0`) and it broadcasts to all other processes a namespace and the name and arguments of the method to run on the instances in that namespace; `Coordinator` instances in the follower processes will receive the broadcast, load the instance corresponding to the namespace from their `objects` variable and call the method corresponding to the provided name with the provided arguments.
The arguments are serialized and deserialized with `pickle` in order to be transmitted using MPI; most python variables can be serialized with `pickle` (with the notable exception of `lambda` functions) so this process is generally transparent to the library user.

The broadcasting of information from the leader process to the follower processes is implemented using MPI Bcast, which must be called on all instances to succeed. The follower `Coordinator` instances run an infinite loop which calls MPI Bcast to receive a command and then executes that command, they exit the loop when they receive a stop command instead of a method execution command. The leader `Coordinator` instance issues broadcasts whenever its `broadcast` method is called (usually due to a call to a method with the `@parallel` decorator); when all the computation is

finished it broadcasts the stop command.
To implement this behaviour in an easy to use way we used Python's `with` construct, that automatically calls the `__enter__` and `__exit__` methods respectively at the start and end of a `with constructor_instance` block. The `__enter__` method runs the infinite loop previously described if the process is a follower and it sets `is_leader_inside_with_statement = True` if it is the leader instead. In the `__exit__` method the leader process broadcasts the stop command and `is_leader_inside_with_statement` is set back to `False`.

The `is_ready_to_broadcast` method is a getter for the variable `is_leader_inside_with_statement`, it is used only by the `Coordinated` class to check whether a `broadcast` can be issued. It has been implemented as a method of `Coordinator` and not of `Coordinated` to allow modifications or extensions of the `Coordinator` class that change the setup to be implemented without changes to `Coordinated`.

3.4.1. **Coordinator instead of multiple communicators**

Instead of using the `Coordinator` class to distinguish messages in different namespaces we could have used a different communicator for each namespace (e.g. running `MPI_Comm_dup` in the constructor of `Coordinated`), but then the follower replicas would need to be multithreaded if several `Coordinated` instances are needed, this would have made the library more complex for the end user since multithreading is still a relatively niche python feature.

We decided to introduce the `Coordinator` class that multiplexes the messages of all namespaces in a single communicator, this has the advantage that no multithreading is needed to support several `Coordinated` instances. If a user of the library needs the `Coordinated` classes to operate in parallel, they can still use multithreading and different communicators: they just need to instantiate multiple Coordinator classes.

3.5. **Benchmark**

To test the performance of the `coordinated.py` module we implemented a class `PingEnvironment` which inherits from `Coordinated` and has a single parallel method `ping`, that is a no-op. This class allowed us to test the latency introduced by the coordination system alone (i.e. without the computational time needed to run the actual simulation which is generally much higher than the latency introduced by the `coordinated.py` module).

In particular we measured the wall time elapsed during 10 million executions of the `ping` method running on 1, 2, 4 and 8 MPI processes on the same machine, more specifically on one machine of the gigat cluster, part of the HPC resources of MOX; the collected data is shown in Table 1.
As shown in Figure 4 the function call overhead increases linearly with the number of MPI processes: fitting a linear model the R^2 is 99.9%.

Extrapolating from the measured data we obtain that even with 16 processes the overhead is below 10 µs per parallel function call, which is negligible for the intended applications of the module.

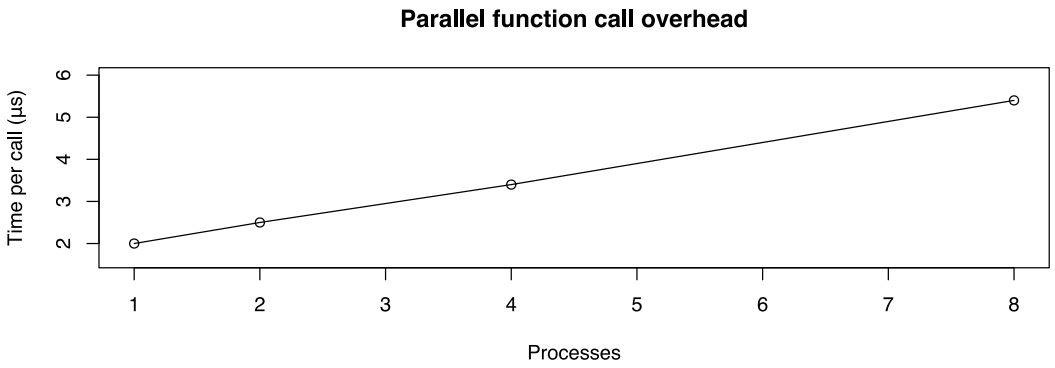

FIGURE 4. Average time per call to the `ping` function depending on the number of MPI processes; the average is obtained from 10 million successive calls.

Number of processes	1	2	4	8
Average function call time	1.961 µs	2.517 µs	3.410 µs	5.353 µs

TABLE 1. Average time per call to the `ping` function depending on the number of MPI processes; the average is obtained from 10 million successive calls.

4. FEM module

The `fem.py` module contains various classes that simplify the development of simulations of physical environments using FEniCSx. The `TimeProblem` class allows the users to iteratively update the PDE one piece at a time (the linear form, the boundary conditions, …), the `HeatEnvironment` and `MonodomainMitchellSchaeffer` implement two physical environments modeled respectively by the heat equation and the monodomain Mitchell-Schaeffer model.

`HeatEnvironment` and `MonodomainMitchellSchaeffer` derive from the `Coordinated` class of the `coordinated.py` module, so that they can be easily used in a reinforcement learning setting.

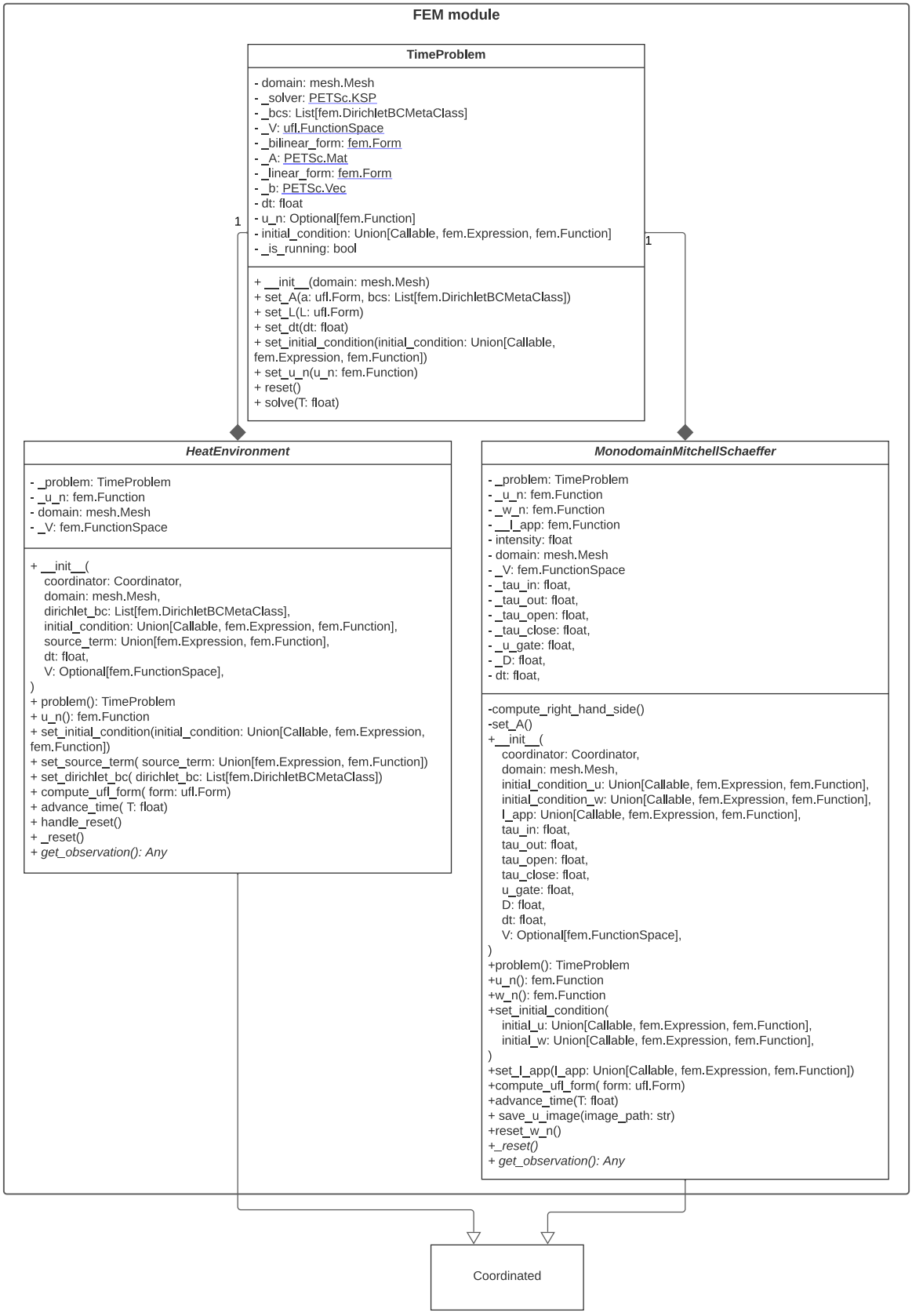

FIGURE 5. UML diagram of the FEM module

4.1. **TimeProblem class**

In the context of training an agent with reinforcement learning we need to iteratively update the PDE that we want to solve in response to the agent's actions. To allow user's of the library to do this easily we implemented the `TimeProblem` class, which follows the builder pattern: it allows the user to set the bilinear form, the right hand side, the boundary conditions, the initial condition and the time step of the problem each with its own setter function; the setters can be called again at any point in the simulation to change the PDE without needing to restart the simulation.

The constructor of the class accepts as the only parameter the mesh on which the simulation should be computed, this is also the only parameter that cannot be changed during the simulation.

The `solve` method advances the simulation of `T` time units in steps determined by `dt`.
The `TimeProblem` class is designed to handle problems that have been discretized in time with a finite differences scheme and then the semidiscretized problem has been expressed in weak formulation, so that it can be solved by FENiCSx. The problem in weak formulation has as unknown `u`, that is the solution one time step in the future, and has a known term `u_n`, that is the solution at the current time.
Each simulation step solves a PDE using FEniCSx and then updates the term `u_n` with the newly obtained solution, this update provides the PDE to solve in the next step. To perform this update efficiently we assume that `u_n` and `u` belong to the same function space, so that their representations as vectors can be directly copied; if the assumption doesn't hold the code can be easily updated to interpolate `u` into `u_n`, at the expense of some additional computation.

The `reset` method allows the user to restart the simulation from the initial condition.

4.2. **HeatEnvironment class**

`HeatEnvironment` is an abstract class implementing an environment modeled by the heat equation (Equation 2) that can be used to train a tensorflow agent, because it implements the `PyEnvironment` abstract class. The derived classes should implement the methods `get_observation`, `action_spec`, `observation_spec` and `_step` which define how the agent interacts with the environment, the class `HeatEnvironment` handles the physical simulation and exposes methods to interact with it.

$$\begin{cases} \frac{\partial u}{\partial t} = \nabla^2 u + f & \text{in}\Omega \times (0, T] \\ u = u_D & \text{in } \partial\Omega \times (0, T] \\ u = u_0 & \text{at } t = 0 \end{cases} \tag{2}$$

The constructor accepts as parameters the various terms that appear in the equation (initial condition, boundary conditions and source term), the simulation parameters (time step, mesh and function space) and a `Coordinator` instance. The constructor handles registering the instance with the provided coordinator and sets up a `TimeProblem` instance to simulate the PDE.

`HeatEnvironment` has setters for the source term, the boundary conditions and the initial condition, so that they can be changed in response to agent actions or to simulate different environments in different runs.

The method `advance_time` can be called to advance the simulation by T time units, in most cases this will be called by the `_step` function implemented in classes deriving from `HeatEnvironment`.

The method `_reset` resets the simulation to the initial condition, it is called automatically by the tf-agents library. Since the reset changes the simulation's state, this method is implemented using the `parallel` decorator; this is a good example of how the `coordinated.py` module allows distributed computation to be seamlessly integrated with single-process computation. To implement custom reset behavior the child classes can override `handle_reset`, but shouldn't override `_reset` directly, because the latter implements the behaviour required by tf-agents.

The method `compute_ufl_form` is a utility function to compute the value of a variational form. This method can for example be used to compute observations or rewards that will be used for reinforcement learning.

4.2.1. **Usage example**

In the example `example_dojo_fenics.py` we extended `HeatEnvironment` implementing the methods `get_observation`, `action_spec`, `observation_spec` and `_step` to simulate an agent that can observe some metrics derived from the environment (the temperature in the 4 quadrants). The agent can stop the simulation at any time, it receives a penalty for waiting and at the end gets a reward proportional to the average temperature.

The task is simple, but this example allows us to showcase how our library can be used to easily train a tensorflow agent with custom interactions with the environment; indeed the whole example is less then 150 lines of code.

The `System` class in the example derives from `HeatEnvironment` and the constructor of the former provides the parameters of the simulation that `HeatEnvironment` requires (e.g. the mesh).

The `get_observation` method is implemented using `HeatEnvironment.compute_ufl_form` to compute the integral of the temperature in each quadrant, these 4 numbers are provided to the agent as observations.
The `_step` method advances or stops the simulation depending on the action performed by the agent, when the simulation is stopped the integral of the temperature over the whole domain is returned as reward.
The `handle_reset` method chooses a random initial condition each time, so that the agent has to make different choices every time.

To train the agent we use the `Dojo` class contained in the `dojo.py` module.

4.3. MonodomainMitchellSchaeffer class

4.3.1. Model

To demonstrate the flexibility of the library we also implemented a simulation of the monodomain Mitchell-Schaeffer equation, which models the trans-membrane potential in the heart. In particular we reproduced the typical spiral shape of reentrant arrhythmias.

The model from which we started is the following:

$$\begin{aligned}
\frac{\partial u}{\partial t} &= D\nabla^2 u + J_{\text{in}}(w,u) + J_{\text{out}}(u) + I_{\text{app}} \\
\frac{dw}{dt} &= \begin{cases} \frac{1-w}{\tau_{\text{open}}} & \text{if } u < u_{\text{gate}} \\ -\frac{w}{\tau_{\text{close}}} & \text{if } u \geq u_{\text{gate}} \end{cases} \\
J_{\text{in}}(w,u) &= \frac{wu^2(1-u)}{\tau_{\text{in}}} \\
J_{\text{out}}(u) &= -\frac{u}{\tau_{\text{out}}}
\end{aligned} \tag{3}$$

We discretized in time using forward euler for w and backward euler for u:

$$\begin{aligned}
u^{n+1} - D\Delta t\nabla^2 u^{n+1} &= u^n + \Delta t\big(J_{\text{in}}(w^n,u^n) + J_{\text{out}}(u^n) + I_{\text{app}}\big) \\
\frac{w^{n+1}-w^n}{\Delta t} &= \begin{cases} \frac{1-w^n}{\tau_{\text{open}}} & \text{if } u^n < u_{\text{gate}} \\ -\frac{w^n}{\tau_{\text{close}}} & \text{if } u^n \geq u_{\text{gate}} \end{cases}
\end{aligned} \tag{4}$$

Finally we expressed this problem in weak formulation:

$$\begin{gathered}
\text{find } u \in \mathbb{V} \text{ such that} \\
\int uv - D\Delta t(\nabla u)(\nabla v)\ dx = \int \big(u^n + \Delta t\big(J_{\text{in}} + J_{\text{out}} + I_{\text{app}}\big)\big)v\ dx \qquad \forall v \in \mathbb{V}
\end{gathered} \tag{5}$$

We are using Neumann boundary conditions (which are the default in FENiCSx) and the initial conditions will be specified by the users of the class.

4.3.2. **Implementation**

The class `MonodomainMitchellSchaeffer` implements this model leveraging the `Coordinated` and `TimeProblem` classes.

The constructor accepts the parameters defining the PDE (initial conditions, τ_{in}, τ_{out}, τ_{open}, τ_{close}, D and I_{app}), the simulation parameters (time step, function space and mesh) and a `Coordinator` instance. This method initializes a `TimeProblem` instance to simulate the PDE.

The class `MonodomainMitchellSchaeffer` exposes setters for the initial condition and the applied current, which are specialized versions of the setters exposed by `TimeProblem`. The initial conditions and the applied current can be provided to the setters as `fem.Expression`, `fem.Function` or as python functions; in the case of python functions they have to be defined with the `def` syntax and not as `lambda` function, because the setters are parallel methods and thus their arguments are serialized with `pickle` to be sent to the follower processes and `pickle` can't serialize `lambda` functions.

The `advance_time` method advances the simulation by T time units, each time step involves two phases: the forward euler phase to update w and the FEniCSx phase to update u.
To compute w_{n+1} we express the forward euler update formula as a `fem.Expression` and then interpolate it to obtain the `fem.Function` corresponding to w_{n+1}.
Then we use `TimeProblem.solve` to solve the semidiscretized problem and find the new value of u.

4.3.3. **Benchmark**

In the file `example_ms.py` we use the class `MonodomainMitchellSchaeffer` to simulate reentrant arrhythmias. We added a mock implementation of the methods required by tf-agents, because in this example we want to benchmark only our implementation without the computation required to train the neural network.

We use the following initialization for u and w, which approximates the state of the cardiac tissue in between two pulses:

$$\begin{aligned} u(x,y) &\equiv 0 \\ v(x,y) &= \begin{cases} \min(-x,1) & \text{if } x < 0 \\ 0 & \text{if } x \geq 0 \end{cases} \end{aligned} \tag{6}$$

To stimulate a reentrant activity we apply a non-zero current only for $t \in [0, 30]$ and $(x, y) \in (-5, 5) \times (0, L)$.

Running the simulation and plotting u we verified that the qualitative behaviour of the model is the expected spiral pattern that is known to characterize reentrant activity (as seen in Figure 6).

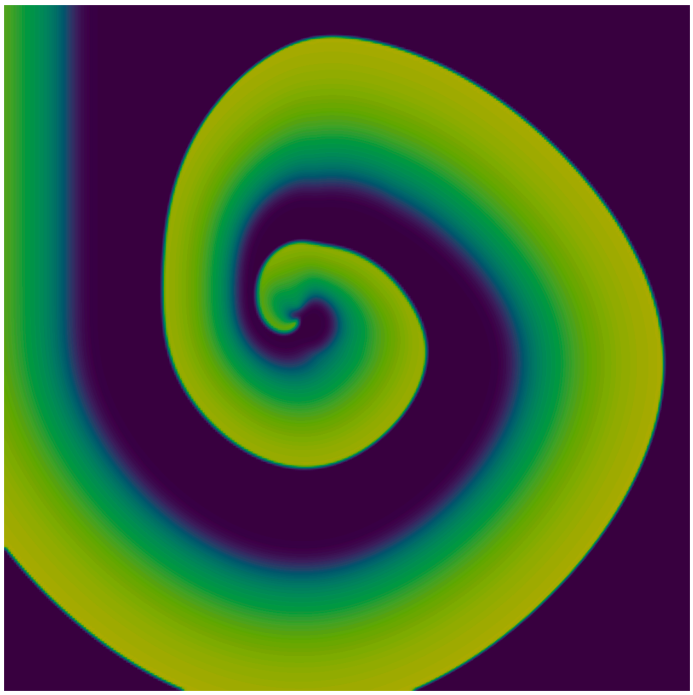

FIGURE 6. The transmembrane potential u at $t = 1000$ in the simulation of reentrant activity.

To assess the scalability of our code we measured the execution time of 100 simulation steps with a grid of resolution 3000×3000 using 1, 2, 4 and 8 MPI processes; we repeated the tests 3 times to compute the variance of the obtained measurements (see Figure 7 and Table 2 for the resulting data). We used the gigat cluster to run the benchmark.

Using the collected data we fitted a linear model of the form $\text{Time} = A \cdot \frac{1}{\text{Number of processes}} + B$ obtaining as estimates for the parameters $A = 695$ seconds and $B = 694$ seconds; the R^2 of the model was 96.6%.
The high value of B, that is the serial part of the program, indicates that there is room for more efficient implementations of the simulation of this PDE; this remains as an open problem for future works.

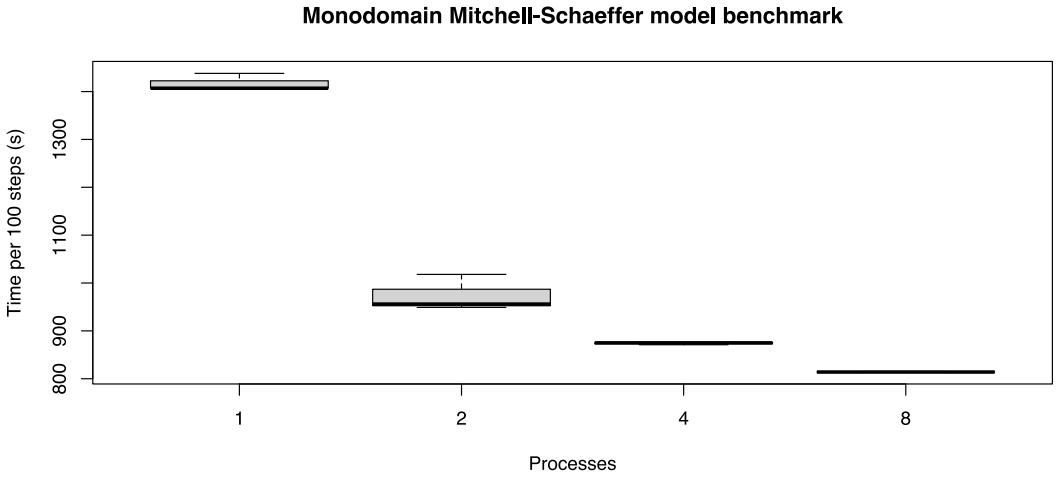

FIGURE 7. Wall clock time to compute 100 steps simulations of the Monodomain Mitchell-Schaeffer model using 1, 2, 4 and 8 MPI processes.

Number of processes	Average function call time
1	1438
1	1407
1	1407
2	1018
2	949
2	956
4	871
4	875
4	876
8	814
8	815
8	814

TABLE 2. Wall clock time to compute 100 steps simulations of the Monodomain Mitchell-Schaeffer model.

4.3.4. **ATP training**

We then used the `MonodomainMitchellSchaeffer` simulation to train an agent to perform Antitachycardia Pacing (ATP): the agent could decide when to apply an electric stimulus to the heart using a simulated ICD with the goal

of terminating a reentrant arrhythmia. The source code for this model is found in the `example_dojo_ms.py` file.

We provide a negative reward every time the agent applies a shock and a large positive reward for successfully terminating the arrhythmia. The negative reward is needed to ensure that the agent only applies the minimum necessary shocks, in real world applications this is important to limit the discomfort of the person and also to prolong the battery life of the ICD.

The input of the agent is a simulated version of the pentaray catheter commonly used in cardiology to create electric activation maps of the heart. The catheter has 5 splines with 4 electrodes each placed as shown in Figure 8. The potential measured by the catheter at position $(\mathrm{cx}, \mathrm{cy})$ is defined as follows

$$\int_{\Omega} \exp\left(-\mu * \sqrt{(x-\mathrm{cx})^2 + (y-\mathrm{cy})^2}\right) u(x,y) \qquad d\Omega$$
$$\mu = 33 \tag{7}$$

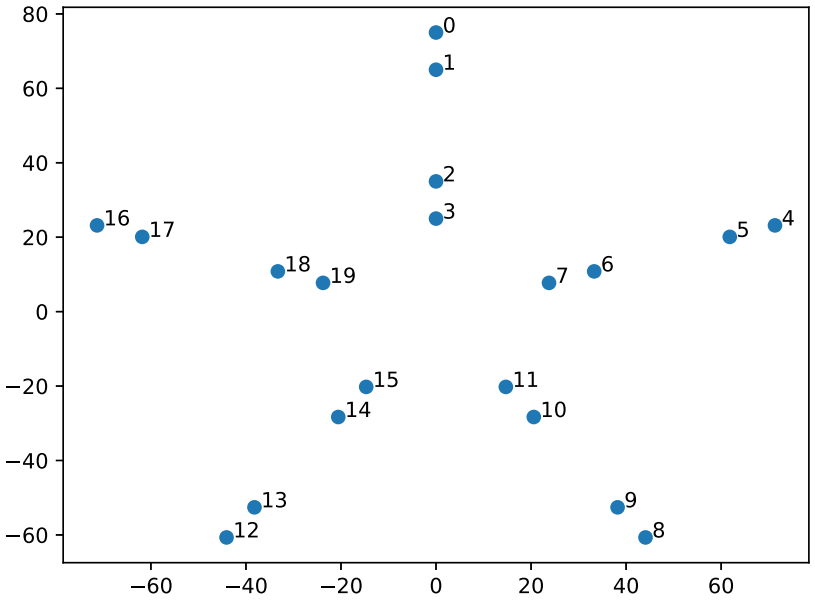

FIGURE 8. Positions of the electrodes of a pentaray catheter

The class `System` implements the `_reset` method by randomly choosing the shape (height and width of the rectangle) and location of the shock used to induce a reentrant activation, thus each simulation requires different actions to stop the arrhythmia.

The agent can interact with the simulation during the first 1000 time steps: every 50 time steps the agent can choose to apply a shock; following this period the simulation runs for up to 1000 steps to check whether the integral of u falls below a threshold, which means that the agent has successfully stopped the arrhythmia.

This application of reinforcement learning showed promising results, as it was able to learn to stop arrhythmias when the variability in the shape and location of the initial shock was moderate. Further research is needed to apply this technique to the wider range of reentrant activity and heart topologies which are present in human patients; in particular our simulation approximates the heart surface with a 2D square, while more realistic approaches should account for the 3D geometry of the heart.

5. Dojo module

The `dojo.py` module contains a class `Dojo` which implements a training pipeline for a tensorflow DQN agent. The class handles data collection, agent training and metrics production.

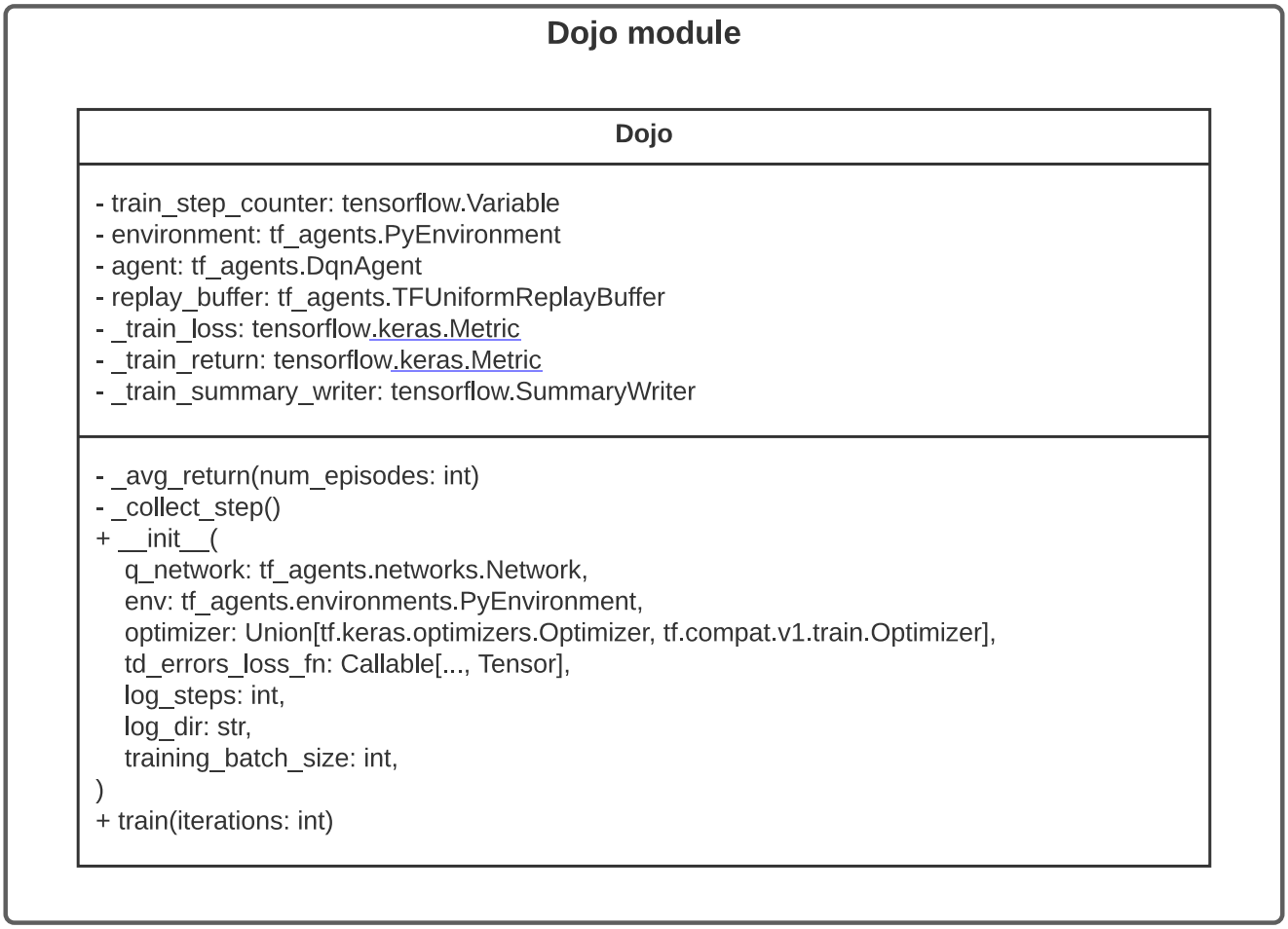

FIGURE 9. UML diagram of the Dojo module

The constructor takes as parameters the network structure that the agent should use, the environment with which it should interact, some training hyperparameters (optimizer, loss function and batch size) and logging settings (frequency of the logs and folder in which to save the logs).

The `train` method iterates data collection and training for as many cycles as specified by the `iterations` parameter. At each step it collects a batch of data, it runs a training step of the neural network contained in the agent using the agent's `train` method and updates the training loss metric. Every `_log_steps` steps the method computes the average return and logs the metrics.

The average return computation is implemented in the function `Dojo._avg_return`, which runs several episodes (the number of episodes is a parameter of the method) of agent-environment interactions and computes the average return over those episodes. This metric cannot be computed from the episodes used for training, because during training the agent is using a collection policy which may be different from the target policy (e.g. the collection policy may include some randomness to increase exploration).

The logs are printed to console, but they are also logged to a tensorboard directory, so the training of the model can be monitored using tensorboard's tooling.

The collected data is stored in a replay buffer so that the agent is trained on all the generated episodes and not only the last batch.

6. Using the library

For ease of use we packaged the library and published it on PyPI, so it can be installed simply running the following command:

```
pip install physicsrl
```

Installation of MPI and FENiCSx is still needed, detailed instructions to do this are available in the GitHub repository.

The library can then be imported as any Python module:

```
from physicsrl import coordinated, fem, dojo
```

The only boilerplate code required to use the library to train a tf-agent model with an environment simulated using FENiCSx is the following

```
# ...
# Define the class System deriving from coordinated.Coordinated
# ...

coordinator = coordinated.Coordinator()
env = System(coordinator)
with coordinator:
    if coordinator.is_leader():
        env = tf_py_environment.TFPyEnvironment(env)
        q_net = QNetwork(env.observation_spec(), env.action_spec())

        dojo = Dojo(q_net, env)

        dojo.train(100)
```

The GitHub repository's folder `examples` contains several examples of how to use the various modules of the library.

References

1. Swenson, D. J., Taepke, R. T., Blauer, J. J., Kwan, E., Ghafoori, E., Plank, G., Vigmond, E., MacLeod, R. S., DeGroot, P., Ranjan, R.: Direct comparison of a novel antitachycardia pacing algorithm against present methods using virtual patient modeling. Heart Rhythm. 17, 1602–1608 (2020). https://doi.org/https://doi.org/10.1016/j.hrthm.2020.05.009

2. Muizniece, L., Bertagnoli, A., Qureshi, A., Zeidan, A., Roy, A., Muffoletto, M., Aslanidi, O.: Reinforcement Learning to Improve Image-Guidance of Ablation Therapy for Atrial Fibrillation. Frontiers in Physiology. 12, (2021). https://doi.org/10.3389/fphys.2021.733139

3. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with Deep Reinforcement Learning, (2013)

4. Guadarrama, S., Korattikara, A., Ramirez, O., Castro, P., Holly, E., Fishman, S., Wang, K., Gonina, E., Wu, N., Kokiopoulou, E., Sbaiz, L., Smith, J., Bartók, G., Berent, J., Harris, C., Vanhoucke, V., Brevdo, E.: TF-Agents: A library for Reinforcement Learning in TensorFlow, https://github.com/tensorflow/agents

5. Logg, A., Mardal, K.-A., Wells, G. N., others: Automated Solution of Differential Equations by the Finite Element Method. Springer (2012)