TensorForce Documentation

Release 0.2alpha

reinforce.io

Sep 23, 2017

Contents:

TensorForce is an open source reinforcement learning library focused on providing clear APIs, readability and modularisation to deploy reinforcement learning solutions both in research and practice. TensorForce is built on top on TensorFlow.

CHAPTER 1

Quick start

For a quick start, you can run one of our example scripts using the provided configurations, e.g. to run the TRPO agent on CartPole, execute from the examples folder:

```
python examples/openai_gym.py CartPole-v0 -a PPOAgent -c examples/configs/ppo_
↪cartpole.json -n examples/configs/ppo_cartpole_network.json
```

In python, it could look like this:

```
# examples/quickstart.py

import numpy as np

from tensorforce import Configuration
from tensorforce.agents import PPOAgent
from tensorforce.core.networks import layered_network_builder
from tensorforce.execution import Runner
from tensorforce.contrib.openai_gym import OpenAIGym

# Create an OpenAIgym environment
env = OpenAIGym('CartPole-v0')

# Create a Trust Region Policy Optimization agent
agent = PPOAgent(config=Configuration(
    log_level='info',
    batch_size=4096,

    gae_lambda=0.97,
    learning_rate=0.001,
    entropy_penalty=0.01,
    epochs=5,
    optimizer_batch_size=512,
    loss_clipping=0.2,

    states=env.states,
    actions=env.actions,
    network=layered_network_builder([
        dict(type='dense', size=32),
        dict(type='dense', size=32)
    ])
))

# Create the runner
runner = Runner(agent=agent, environment=env)


# Callback function printing episode statistics
def episode_finished(r):
    print("Finished episode {ep} after {ts} timesteps (reward: {reward})".format(ep=r.
↪episode, ts=r.timestep,
                                                                                  ␣
↪reward=r.episode_rewards[-1]))
    return True


# Start learning
runner.run(episodes=3000, max_timesteps=200, episode_finished=episode_finished)

# Print statistics
print("Learning finished. Total episodes: {ep}. Average reward of last 100 episodes:
↪{ar}.".format(ep=runner.episode,
                                                                                      ␣
↪             ar=np.mean(
                                                                                      ␣
↪                 runner.episode_rewards[
                                                                                      ␣
↪                 -100:])))
```

Agent and model overview

A reinforcement learning agent provides methods to process states and return actions, to store past observations, and to load and save models. Most agents employ a `Model` which implements the algorithms to calculate the next action given the current state and to update model parameters from past experiences.

Environment <-> Runner <-> Agent <-> Model

Parameters to the agent are passed as a `Configuration` object. The configuration is passed on to the `Model`.

Ready-to-use algorithms

We implemented some of the most common RL algorithms and try to keep these up-to-date. Here we provide an overview over all implemented agents and models.

Agent / General parameters

`Agent` is the base class for all reinforcement learning agents. Every agent inherits from this class.

class `tensorforce.agents.`**`Agent`**(*config*, *model=None*)

Basic Reinforcement learning agent. An agent encapsulates execution logic of a particular reinforcement learning algorithm and defines the external interface to the environment.

The agent hence acts an intermediate layer between environment and backend execution (value function or policy updates).

Each agent requires the following configuration parameters:

- `states`: dict containing one or more state definitions.
- `actions`: dict containing one or more action definitions.
- `preprocessing`: dict or list containing state preprocessing configuration.
- `exploration`: dict containing action exploration configuration.

The configuration is passed to the *Model* and should thus include its configuration parameters, too.

Examples:

act (*state*, *deterministic=False*)

Return action(s) for given state(s). First, the states are preprocessed using the given preprocessing configuration. Then, the states are passed to the model to calculate the desired action(s) to execute.

After obtaining the actions, exploration might be added by the agent, depending on the exploration configuration.

Parameters

- **state** – One state (usually a value tuple) or dict of states if multiple states are expected.
- **deterministic** – If true, no exploration and sampling is applied.

Returns Scalar value of the action or dict of multiple actions the agent wants to execute.

observe (*reward*, *terminal*)

Observe experience from the environment to learn from. Optionally preprocesses rewards Child classes should call super to get the processed reward EX: reward, terminal = super()...

Parameters

- **reward** – scalar reward that resulted from executing the action.
- **terminal** – boolean indicating if the episode terminated after the observation.

Returns processed_reward terminal

reset ()

Reset agent after episode. Increments internal episode count, internal states and preprocessors.

Returns void

Model

The `Model` class is the base class for reinforcement learning models.

class `tensorforce.models.`**Model** (*config*)

Bases: `object`

Base model class.

Each model requires the following configuration parameters:

- `discount`: float of discount factor (gamma).
- `learning_rate`: float of learning rate (alpha).
- `optimizer`: string of optimizer to use (e.g. 'adam').
- `device`: string of tensorflow device name.
- `tf_summary`: string directory to write tensorflow summaries. Default None

- `tf_summary_level`: int indicating which tensorflow summaries to create.
- `tf_summary_interval`: int number of calls to get_action until writing tensorflow summaries on update.
- `log_level`: string containing log level (e.g. 'info').
- `distributed`: boolean indicating whether to use distributed tensorflow.
- `global_model`: global model.
- `session`: session to use.

`create_tf_operations`(*config*)

Creates generic TensorFlow operations and placeholders required for models.

Parameters **`config`** – Model configuration which must contain entries for states and actions.

Returns:

`load_model`(*path*)

Import model from path using tf.train.Saver.

Parameters **`path`** – Path to checkpoint

Returns:

`reset`()

Resets the internal state to the initial state. Returns: A list containing the internal_inits field.

`save_model`(*path*, *use_global_step=True*)

Export model using a tf.train.Saver. Optionally append current time step as to not overwrite previous checkpoint file. Set to 'false' to be able to load model from exact path it was saved to in case of restarting program.

Parameters
- **`path`** – Model export directory
- **`use_global_step`** – Whether to append the current timestep to the checkpoint path.

Returns:

`update`(*batch*)

Generic batch update operation for Q-learning and policy gradient algorithms. Takes a batch of experiences,

Parameters **`batch`** – Batch of experiences.

Returns:

MemoryAgent

class `tensorforce.agents.`**`MemoryAgent`**(*config*, *model=None*)

Bases: `tensorforce.agents.agent.Agent`

The `MemoryAgent` class implements a replay memory, from which it samples batches to update the value function.

Each agent requires the following `Configuration` parameters:

- `states`: dict containing one or more state definitions.
- `actions`: dict containing one or more action definitions.
- `preprocessing`: dict or list containing state preprocessing configuration.
- `exploration`: dict containing action exploration configuration.

The `MemoryAgent` class additionally requires the following parameters:

- `batch_size`: integer of the batch size.
- `memory_capacity`: integer of maximum experiences to store.
- `memory`: string indicating memory type ('replay' or 'prioritized_replay').
- `update_frequency`: integer indicating the number of steps between model updates.
- `first_update`: integer indicating the number of steps to pass before the first update.
- `repeat_update`: integer indicating how often to repeat the model update.

`import_observations`(*observations*)

Load an iterable of observation dicts into the replay memory.

Parameters

- **`observations`** – An iterable with each element containing an observation. Each
- **`requires keys 'state','action','reward','terminal', 'internal'.`** (*`observation`*) –
- **`an empty list[] for 'internal' if internal state is irrelevant.`** (*`Use`*) –

Returns:

BatchAgent

class `tensorforce.agents.`**`BatchAgent`**(*config*, *model=None*)

Bases: `tensorforce.agents.agent.Agent`

The `BatchAgent` class implements a batch memory, which is cleared after every update.

Each agent requires the following `Configuration` parameters:

- `states`: dict containing one or more state definitions.
- `actions`: dict containing one or more action definitions.
- `preprocessing`: dict or list containing state preprocessing configuration.
- `exploration`: dict containing action exploration configuration.

The `BatchAgent` class additionally requires the following parameters:

- `batch_size`: integer of the batch size.
- `keep_last`: bool optionally keep the last observation for use in the next batch

`observe`(*reward*, *terminal*)

Adds an observation and performs an update if the necessary conditions are satisfied, i.e. if one batch of experience has been collected as defined by the batch size.

In particular, note that episode control happens outside of the agent since the agent should be agnostic to how the training data is created.

Parameters

- **`reward`** – float of a scalar reward
- **`terminal`** – boolean whether episode is terminated or not

Returns: void

Deep-Q-Networks (DQN)

class `tensorforce.agents.`**`DQNAgent`**`(`*config*, *model=None*`)`
Bases: `tensorforce.agents.memory_agent.MemoryAgent`

Deep-Q-Network agent (DQN). The piece de resistance of deep reinforcement learning as described by Minh et al. (2015). Includes an option for double-DQN (DDQN; van Hasselt et al., 2015)

DQN chooses from one of a number of discrete actions by taking the maximum Q-value from the value function with one output neuron per available action. DQN uses a replay memory for experience playback.

Configuration:

Each agent requires the following configuration parameters:

- `states`: dict containing one or more state definitions.
- `actions`: dict containing one or more action definitions.
- `preprocessing`: dict or list containing state preprocessing configuration.
- `exploration`: dict containing action exploration configuration.

The `MemoryAgent` class additionally requires the following parameters:

- `batch_size`: integer of the batch size.
- `memory_capacity`: integer of maximum experiences to store.
- `memory`: string indicating memory type (‘replay’ or ‘prioritized_replay’).
- `update_frequency`: integer indicating the number of steps between model updates.
- `first_update`: integer indicating the number of steps to pass before the first update.
- `repeat_update`: integer indicating how often to repeat the model update.

Each model requires the following configuration parameters:

- `discount`: float of discount factor (gamma).
- `learning_rate`: float of learning rate (alpha).
- `optimizer`: string of optimizer to use (e.g. ‘adam’).
- `device`: string of tensorflow device name.
- `tf_summary`: string directory to write tensorflow summaries. Default None
- `tf_summary_level`: int indicating which tensorflow summaries to create.
- `tf_summary_interval`: int number of calls to get_action until writing tensorflow summaries on update.
- `log_level`: string containing logleve (e.g. ‘info’).
- `distributed`: boolean indicating whether to use distributed tensorflow.
- `global_model`: global model.
- `session`: session to use.

The DQN agent expects the following additional configuration parameters:

- `target_update_frequency`: int of states between updates of the target network.
- `update_target_weight`: float of update target weight (tau parameter).

- `double_dqn`: boolean indicating whether to use double-dqn.
- `clip_loss`: float if not 0, uses the huber loss with clip_loss as the linear bound

Normalized Advantage Functions

class `tensorforce.agents.`**`NAFAgent`**(*config*, *model=None*)

Bases: `tensorforce.agents.memory_agent.MemoryAgent`

Normalized Advantage Functions (NAF) agent (Gu et al., 2016), a.k.a. DQN for continuous actions.

Configuration:

Each agent requires the following configuration parameters:

- `states`: dict containing one or more state definitions.
- `actions`: dict containing one or more action definitions.
- `preprocessing`: dict or list containing state preprocessing configuration.
- `exploration`: dict containing action exploration configuration.

The `MemoryAgent` class additionally requires the following parameters:

- `batch_size`: integer of the batch size.
- `memory_capacity`: integer of maximum experiences to store.
- `memory`: string indicating memory type ('replay' or 'prioritized_replay').
- `update_frequency`: integer indicating the number of steps between model updates.
- `first_update`: integer indicating the number of steps to pass before the first update.
- `repeat_update`: integer indicating how often to repeat the model update.

Each model requires the following configuration parameters:

- `discount`: float of discount factor (gamma).
- `learning_rate`: float of learning rate (alpha).
- `optimizer`: string of optimizer to use (e.g. 'adam').
- `device`: string of tensorflow device name.
- `tf_summary`: string directory to write tensorflow summaries. Default None
- `tf_summary_level`: int indicating which tensorflow summaries to create.
- `tf_summary_interval`: int number of calls to get_action until writing tensorflow summaries on update.
- `log_level`: string containing logleve (e.g. 'info').
- `distributed`: boolean indicating whether to use distributed tensorflow.
- `global_model`: global model.
- `session`: session to use.

The NAF agent expects the following additional configuration parameters:

- `target_update_frequency`: int of states between updates of the target network.
- `update_target_weight`: float of update target weight (tau parameter).
- `clip_loss`: float if not 0, uses the huber loss with clip_loss as the linear bound

Deep-Q-learning from demonostration (DQFD)

class `tensorforce.agents.`**`DQFDAgent`**(*config*, *model=None*)

Bases: `tensorforce.agents.memory_agent.MemoryAgent`

Deep Q-learning from demonstration (DQFD) agent (Hester et al., 2017). This agent uses DQN to pre-train from demonstration data.

Configuration:

Each agent requires the following configuration parameters:

- `states`: dict containing one or more state definitions.
- `actions`: dict containing one or more action definitions.
- `preprocessing`: dict or list containing state preprocessing configuration.
- `exploration`: dict containing action exploration configuration.

Each model requires the following configuration parameters:

- `discount`: float of discount factor (gamma).
- `learning_rate`: float of learning rate (alpha).
- `optimizer`: string of optimizer to use (e.g. 'adam').
- `device`: string of tensorflow device name.
- `tf_summary`: string directory to write tensorflow summaries. Default None
- `tf_summary_level`: int indicating which tensorflow summaries to create.
- `tf_summary_interval`: int number of calls to get_action until writing tensorflow summaries on update.
- `log_level`: string containing logleve (e.g. 'info').
- `distributed`: boolean indicating whether to use distributed tensorflow.
- `global_model`: global model.
- `session`: session to use.

The `DQFDAgent` class additionally requires the following parameters:

- `batch_size`: integer of the batch size.
- `memory_capacity`: integer of maximum experiences to store.
- `memory`: string indicating memory type ('replay' or 'prioritized_replay').
- `min_replay_size`: integer of minimum replay size before the first update.
- `update_rate`: float of the update rate (e.g. 0.25 = every 4 steps).
- `target_network_update_rate`: float of target network update rate (e.g. 0.01 = every 100 steps).
- `use_target_network`: boolean indicating whether to use a target network.
- `update_repeat`: integer of how many times to repeat an update.
- `update_target_weight`: float of update target weight (tau parameter).
- `demo_sampling_ratio`: float, ratio of expert data used at runtime to train from.
- `supervised_weight`: float, weight of large margin classifier loss.

- `expert_margin`: float of difference in Q-values between expert action and other actions enforced by the large margin function.
- `clip_loss`: float if not 0, uses the huber loss with clip_loss as the linear bound

`import_demonstrations`(*demonstrations*)
Imports demonstrations, i.e. expert observations. Note that for large numbers of observations, set_demonstrations is more appropriate, which directly sets memory contents to an array an expects a different layout.

Parameters **`demonstrations`** – List of observation dicts

Returns:

`observe`(*reward*, *terminal*)
Adds observations, updates via sampling from memories according to update rate. DQFD samples from the online replay memory and the demo memory with the fractions controlled by a hyper parameter p called 'expert sampling ratio.

Parameters

- **`reward`** –
- **`terminal`** –

Returns:

`pretrain`(*steps*)
Computes pretrain updates.

Parameters **`steps`** – Number of updates to execute.

Returns:

`set_demonstrations`(*batch*)
Set all demonstrations from batch data. Expects a dict wherein each value contains an array containing all states, actions, rewards, terminals and internals respectively. of :param batch:

Returns:

Vanilla Policy Gradient

class `tensorforce.agents.`**`VPGAgent`**(*config*, *model=None*)
Bases: `tensorforce.agents.batch_agent.BatchAgent`

Vanilla Policy Gradient agent as described by Sutton et al. (1999).

Configuration:

Each agent requires the following `Configuration` parameters:

- `states`: dict containing one or more state definitions.
- `actions`: dict containing one or more action definitions.
- `preprocessing`: dict or list containing state preprocessing configuration.
- `exploration`: dict containing action exploration configuration.

The `BatchAgent` class additionally requires the following parameters:

- `batch_size`: integer of the batch size.
- `keep_last`: bool optionally keep the last observation for use in the next batch

A Policy Gradient Model expects the following additional configuration parameters:

- `baseline`: string indicating the baseline value function (currently 'linear' or 'mlp').
- `gae_rewards`: boolean indicating whether to use GAE reward estimation.

- `gae_lambda`: GAE lambda.
- `normalize_rewards`: boolean indicating whether to normalize rewards.

The VPG agent does not require any additional configuration parameters.

Trust Region Policy Optimization (TRPO)

class `tensorforce.agents.`**`TRPOAgent`**(*config*, *model=None*)

Bases: `tensorforce.agents.batch_agent.BatchAgent`

Trust Region Policy Optimization (Schulman et al., 2015) agent.

Configuration:

Each agent requires the following `Configuration` parameters:

- `states`: dict containing one or more state definitions.
- `actions`: dict containing one or more action definitions.
- `preprocessing`: dict or list containing state preprocessing configuration.
- `exploration`: dict containing action exploration configuration.

The `BatchAgent` class additionally requires the following parameters:

- `batch_size`: integer of the batch size.
- `keep_last`: bool optionally keep the last observation for use in the next batch

A Policy Gradient Model expects the following additional configuration parameters:

- `baseline`: string indicating the baseline value function (currently 'linear' or 'mlp').
- `gae_rewards`: boolean indicating whether to use GAE reward estimation.
- `gae_lambda`: GAE lambda.
- `normalize_rewards`: boolean indicating whether to normalize rewards.

The TRPO agent expects the following additional configuration parameters:

- `learning_rate`: float of learning rate (alpha).
- `optimizer`: string of optimizer to use (e.g. 'adam').
- `cg_damping`: float of the damping factor for the conjugate gradient method.
- `line_search_steps`: int of how many steps to take during line search.
- `max_kl_divergence`: float indicating the maximum kl divergence to allow for updates.
- `cg_iterations`: int of count of conjugate gradient iterations.

State preprocessing

The agent handles state preprocessing. A preprocessor takes the raw state input from the environment and modifies it (for instance, image resize, state concatenation, etc.). You can find information about our ready-to-use preprocessors *here*.

Building your own agent

If you want to build your own agent, it should always inherit from `Agent`. If your agent uses a replay memory, it should probably inherit from `MemoryAgent`, if it uses a batch replay that is emptied after each update, it should probably inherit from `BatchAgent`.

We distinguish between agents and models. The `Agent` class handles the interaction with the environment, such as state preprocessing, exploration and observation of rewards. The `Model` class handles the mathematical operations, such as building the tensorflow operations, calculating the desired action and updating (i.e. optimizing) the model weights.

To start building your own agent, please refer to this blogpost to gain a deeper understanding of the internals of the TensorForce library. Afterwards, have look on a sample implementation, e.g. the DQN Agent and DQN Model.

Environments

A reinforcement learning environment provides the API to a simulated or real environment as the subject for optimization. It could be anything from video games (e.g. Atari) to robots or trading systems. The agent interacts with this environment and learns to act optimally in its dynamics.

Environment <-> Runner <-> Agent <-> Model

class `tensorforce.environments.`**`Environment`**
Base environment class.

`actions`
Return the action space. Might include subdicts if multiple actions are available simultaneously.

Returns: dict of action properties (continuous, number of actions)

`close``()`
Close environment. No other method calls possible afterwards.

`execute``(`*action*`)`
Executes action, observes next state and reward.
Parameters **`action`** – Action to execute.
Returns: tuple of state (tuple), reward (float), and terminal_state (bool).

`reset``()`
Reset environment and setup for new episode.

Returns: initial state of resetted environment.

`states`
Return the state space. Might include subdicts if multiple states are available simultaneously.

Returns: dict of state properties (shape and type).

Ready-to-use environments

OpenAI Gym

OpenAI Universe

Deepmind Lab

Preprocessing

Often it is necessary to modify state input tensors before passing them to the reinforcement learning agent. This could be due to various reasons, e.g.:

- Feature scaling / input normalization,
- Data reduction,
- Ensuring the Markov property by concatenating multiple states (e.g. in Atari)

TensorForce comes with a number of ready-to-use preprocessors, a preprocessing stack and easy ways to implement your own preprocessors.

Usage

The

Each preprocessor implements three methods:

1. The constructor (`__init__`) for parameter initialization
2. `process(state)` takes a state and returns the processed state
3. `processed_shape(original_shape)` takes a shape and returns the processed shape

The preprocessing stack iteratively calls these functions of all preprocessors in the stack and returns the result.

Using one preprocessor

```python
from tensorforce.core.preprocessing import Sequence

pp_seq = Sequence(4)  # initialize preprocessor (return sequence of last 4 states)

state = env.reset()  # reset environment
processed_state = pp_seq.process(state)  # process state
```

Using a preprocessing stack

You can stack multipe preprocessors:

```python
from tensorforce.core.preprocessing import Preprocessing, Grayscale, Sequence

pp_gray = Grayscale()  # initialize grayscale preprocessor
pp_seq = Sequence(4)  # initialize sequence preprocessor

stack = Preprocessing()  # initialize preprocessing stack
stack.add(pp_gray)  # add grayscale preprocessor to stack
stack.add(pp_seq)  # add maximum preprocessor to stack
```

```
state = env.reset()  # reset environment
processed_state = stack.process(state)  # process state
```

Using a configuration dict

If you use configuration objects, you can build your preprocessing stack from a config:

```
from tensorforce.core.preprocessing import Preprocessing

preprocessing_config = [
    {
        "type": "image_resize",
        "kwargs": {
            "width": 84,
            "height": 84
        }
    }, {
        "type": "grayscale"
    }, {
        "type": "center"
    }, {
        "type": "sequence",
        "kwargs": {
            "length": 4
        }
    }
]

stack = Preprocessing.from_config(preprocessing_config)
config.state_shape = stack.shape(config.state_shape)
```

The `Agent` class expects a *preprocessing* configuration parameter and then handles preprocessing automatically:

```
from tensorforce.agents import DQNAgent

agent = DQNAgent(config=dict(
    states=...,
    actions=...,
    preprocessing=preprocessing_config,
    # ...
))
```

Ready-to-use preprocessors

These are the preprocessors that come with TensorForce:

Center

class `tensorforce.core.preprocessing.`**`Center`**

Bases: `tensorforce.core.preprocessing.preprocessor.Preprocessor`

Center/standardize state. Subtract minimal value and divide by range.

Grayscale

class `tensorforce.core.preprocessing.`**`Grayscale`**(*weights=(0.299, 0.587, 0.114)*)
Bases: `tensorforce.core.preprocessing.preprocessor.Preprocessor`

Turn 3D color state into grayscale.

ImageResize

class `tensorforce.core.preprocessing.`**`ImageResize`**(*width*, *height*)
Bases: `tensorforce.core.preprocessing.preprocessor.Preprocessor`

Resize image to width x height.

Normalize

class `tensorforce.core.preprocessing.`**`Normalize`**
Bases: `tensorforce.core.preprocessing.preprocessor.Preprocessor`

Normalize state. Subtract mean and divide by standard deviation.

Sequence

class `tensorforce.core.preprocessing.`**`Sequence`**(*length=2*)
Bases: `tensorforce.core.preprocessing.preprocessor.Preprocessor`

Concatenate `length` state vectors. Example: Used in Atari problems to create the Markov property.

Building your own preprocessor

All preprocessors should inherit from `tensorforce.core.preprocessing.Preprocessor`.

For a start, please refer to the source of the Grayscale preprocessor.

Runners

A "runner" manages the interaction between the Environment and the Agent. TensorForce comes with ready-to-use runners. Of course, you can implement your own runners, too. If you are not using simulation environments, the runner is simply your application code using the Agent API.

Environment <-> Runner <-> Agent <-> Model

Ready-to-use runners

We implemented a standard runner, a threaded runner (for real-time interaction e.g. with OpenAI Universe) and a distributed runner for A3C variants.

Runner

This is the standard runner. It requires an agent and an environment for initialization:

```
from tensorforce.execution import Runner

runner = Runner(
    agent = agent,  # Agent object
    environment = env  # Environment object
)
```

A reinforcement learning agent observes states from the environment, selects actions and collect experience which is used to update its model and improve action selection. You can get information about our ready-to-use agents *here*.

The environment object is either the "real" environment, or a proxy which fulfills the actions selected by the agent in the real world. You can find information about environments *here*.

The runner is started with the `Runner.run(...)` method:

```
runner.run(
    episodes = int,  # number of episodes to run
    max_timesteps = int,  # maximum timesteps per episode
    episode_finished = object,  # callback function called when episode is finished
)
```

You can use the episode_finished callback for printing performance feedback:

```
def episode_finished(r):
    if r.episode % 10 == 0:
        print("Finished episode {ep} after {ts} timesteps".format(ep=r.episode + 1,␣
↪ts=r.timestep + 1))
        print("Episode reward: {}".format(r.episode_rewards[-1]))
        print("Average of last 10 rewards: {}".format(np.mean(r.episode_rewards[-
↪10:])))
    return True
```

Using the Runner

Here is some example code for using the runner (without preprocessing).

```
from tensorforce.config import Configuration
from tensorforce.environments.openai_gym import OpenAIGym
from tensorforce.agents import DQNAgent
from tensorforce.execution import Runner

def main():
    gym_id = 'CartPole-v0'
    max_episodes = 10000
    max_timesteps = 1000

    env = OpenAIGym(gym_id)

    config = Configuration({
        'actions': env.actions,
        'states': env.states
        # ...
    })
    agent = DQNAgent(config)

    runner = Runner(agent, env)

    def episode_finished(r):
        if r.episode % report_episodes == 0:
            logger.info("Finished episode {ep} after {ts} timesteps".format(ep=r.
↪episode, ts=r.timestep))
            logger.info("Episode reward: {}".format(r.episode_rewards[-1]))
            logger.info("Average of last 100 rewards: {}".format(sum(r.episode_
↪rewards[-100:]) / 100))
        return True

    print("Starting {agent} for Environment '{env}'".format(agent=agent, env=env))

    runner.run(max_episodes, max_timesteps, episode_finished=episode_finished)

    print("Learning finished. Total episodes: {ep}".format(ep=runner.episode))

if __name__ == '__main__':
    main()
```

Building your own runner

There are three mandatory tasks any runner implements: Obtaining an action from the agent, passing it to the environment, and passing the resulting observation to the agent.

```
# Get action
action = agent.act(state, self.episode)

# Execute action in the environment
state, reward, terminal_state = environment.execute(action)

# Pass observation to the agent
agent.observe(state, action, reward, terminal_state)
```

The key idea here is the separation of concerns. External code should not need to manage batches or remember network features, this is that the agent is for. Conversely, an agent need not concern itself with how a model is implemented and the API should facilitate easy combination of different agents and models.

If you would like to build your own runner, it is probably a good idea to take a look at the source code of our Runner class.

CHAPTER 2

More information

You can find more information at our TensorForce GitHub repository.

We have a seperate repository available for benchmarking our algorithm implementations [here](https://github.com/reinforceio/tensorforce-benchmark).

Index