TensorForce Documentation

Release 0.2alpha

reinforce.io

Nov 16, 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 examples/configs/ppo.json -n examples/
↪configs/mlp2_network.json
```

In python, it could look like this:

```
# examples/quickstart.py

import numpy as np
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(
    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:])))
```

1.1 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`.

1.1.1 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.

Model

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

MemoryAgent

BatchAgent

Deep-Q-Networks (DQN)

Normalized Advantage Functions

Deep-Q-learning from demonostration (DQFD)

Vanilla Policy Gradient

Trust Region Policy Optimization (TRPO)

1.1.2 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*.

1.1.3 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.

1.2 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`(*actions*)
Executes action, observes next state(s) and reward.
Parameters **`actions`** – Actions to execute.
Returns (Dict of) next state(s), boolean indicating terminal, and reward signal.

`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).

1.2.1 Ready-to-use environments

OpenAI Gym

OpenAI Universe

Deepmind Lab

1.3 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.

1.3.1 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

```
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:

```
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_spec(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,
    # ...
))
```

1.3.2 Ready-to-use preprocessors

These are the preprocessors that come with TensorForce:

Center

Grayscale

ImageResize

Normalize

Sequence

1.3.3 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.

1.4 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

1.4.1 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)
    network_spec = [
        dict(type='dense', size=32, activation='tanh'),
        dict(type='dense', size=32, activation='tanh')
    ]

    agent = DQNAgent(
        states_spec=env.states,
        actions_spec=env.actions,
        network_spec=network_spec,
        batch_size=64
    )

    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()
```

1.4.2 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)

# 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