TensorForce Documentation

Release 0.2alpha

reinforce.io

Nov 09, 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 ppo_agent -c examples/configs/ppo_agent.
↪json -n examples/configs/ppo_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:])))
```

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.

class `tensorforce.agents.`**`Agent`**(*states_spec*, *actions_spec*, *config*)

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(*states*, *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

- **states** – 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.

static from_spec(*spec*, *kwargs*)

Creates an agent from a specification dict.

observe(*terminal*, *reward*)

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

Parameters

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

reset()

Reset the agent to its initial state on episode start. Updates internal episode and timestep counter, internal states, and resets preprocessors.

restore_model(*directory=None*, *file=None*)

Restore TensorFlow model. If no checkpoint file is given, the latest checkpoint is restored. If no checkpoint directory is given, the model's default saver directory is used (unless file specifies the entire path).

Parameters

- **directory** – Optional checkpoint directory.
- **file** – Optional checkpoint file, or path if directory not given.

save_model(*directory=None*, *append_timestep=True*)

Save TensorFlow model. If no checkpoint directory is given, the model's default saver directory is used. Optionally appends current timestep to prevent overwriting previous checkpoint files. Turn off to be able to load model from the same given path argument as given here.

Parameters

- **directory** – Optional checkpoint directory.
- **use_global_step** – Appends the current timestep to the checkpoint file if true.

:param If this is set to True, the load path must include the checkpoint timestep suffix.: :param For example, if stored to models/ and set to true, the exported file will be of the: :param form models/model.ckpt-X where X is the last timestep saved. The load path must: :param precisely match this file name. If this option is turned off, the checkpoint will: :param always overwrite the file specified in path and the model can always be loaded under: :param this path.:

Returns Checkpoint path were the model was saved.

Model

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

class `tensorforce.models.`**`Model`**(*states_spec*, *actions_spec*, *config*, ***kwargs*)

Bases: `object`

Base class for all (TensorFlow-based) models.

`create_output_operations`(*states*, *internals*, *actions*, *terminal*, *reward*, *update*, *deterministic*)

Calls all the relevant TensorFlow functions for this model and hence creates all the TensorFlow operations involved.

Parameters

- **`states`** – Dict of state tensors.
- **`internals`** – List of prior internal state tensors.
- **`actions`** – Dict of action tensors.
- **`terminal`** – Terminal boolean tensor.
- **`reward`** – Reward tensor.
- **`update`** – Boolean tensor indicating whether this call happens during an update.
- **`deterministic`** – Boolean tensor indicating whether action should be chosen deterministically.

`get_optimizer_kwargs`(*states*, *internals*, *actions*, *terminal*, *reward*, *update*)

Returns the optimizer arguments including the time, the list of variables to optimize, and various argument-free functions (in particular `fn_loss` returning the combined 0-dim batch loss tensor) which the optimizer might require to perform an update step.

Parameters

- **`states`** – Dict of state tensors.
- **`internals`** – List of prior internal state tensors.
- **`actions`** – Dict of action tensors.
- **`terminal`** – Terminal boolean tensor.
- **`reward`** – Reward tensor.
- **`update`** – Boolean tensor indicating whether this call happens during an update.

Returns Loss tensor of the size of the batch.

`get_summaries`()

Returns the TensorFlow summaries reported by the model

Returns List of summaries

`get_variables`(*include_non_trainable=False*)

Returns the TensorFlow variables used by the model.

Returns List of variables.

`initialize`(*custom_getter*)

Creates the TensorFlow placeholders and functions for this model. Moreover adds the internal state placeholders and initialization values to the model.

Parameters **`custom_getter`** – The `custom_getter_` object to use for `tf.make_template` when creating TensorFlow functions.

reset()
Resets the model to its initial state on episode start.
Returns Current episode and timestep counter, and a list containing the internal states initializations.

restore(*directory=None*, *file=None*)
Restore TensorFlow model. If no checkpoint file is given, the latest checkpoint is restored. If no checkpoint directory is given, the model's default saver directory is used (unless file specifies the entire path).
Parameters
- **directory** – Optional checkpoint directory.
- **file** – Optional checkpoint file, or path if directory not given.

save(*directory=None*, *append_timestep=True*)
Save TensorFlow model. If no checkpoint directory is given, the model's default saver directory is used. Optionally appends current timestep to prevent overwriting previous checkpoint files. Turn off to be able to load model from the same given path argument as given here.
Parameters
- **directory** – Optional checkpoint directory.
- **append_timestep** – Appends the current timestep to the checkpoint file if true.

Returns Checkpoint path were the model was saved.

tf_actions_and_internals(*states*, *internals*, *update*, *deterministic*)
Creates the TensorFlow operations for retrieving the actions (and posterior internal states) in reaction to the given input states (and prior internal states).
Parameters
- **states** – Dict of state tensors.
- **internals** – List of prior internal state tensors.
- **update** – Boolean tensor indicating whether this call happens during an update.
- **deterministic** – Boolean tensor indicating whether action should be chosen deterministically.

Returns Actions and list of posterior internal state tensors.

tf_discounted_cumulative_reward(*terminal*, *reward*, *discount*, *final_reward=0.0*)
Creates the TensorFlow operations for calculating the discounted cumulative rewards for a given sequence of rewards.
Parameters
- **terminal** – Terminal boolean tensor.
- **reward** – Reward tensor.
- **discount** – Discount factor.
- **final_reward** – Last reward value in the sequence.

Returns Discounted cumulative reward tensor.

tf_loss_per_instance(*states*, *internals*, *actions*, *terminal*, *reward*, *update*)
Creates the TensorFlow operations for calculating the loss per batch instance of the given input states and actions.
Parameters
- **states** – Dict of state tensors.
- **internals** – List of prior internal state tensors.
- **actions** – Dict of action tensors.
- **terminal** – Terminal boolean tensor.
- **reward** – Reward tensor.
- **update** – Boolean tensor indicating whether this call happens during an update.

Returns Loss tensor.

tf_optimization(*states*, *internals*, *actions*, *terminal*, *reward*, *update*)

Creates the TensorFlow operations for performing an optimization update step based on the given input states and actions batch.

Parameters

- **states** – Dict of state tensors.
- **internals** – List of prior internal state tensors.
- **actions** – Dict of action tensors.
- **terminal** – Terminal boolean tensor.
- **reward** – Reward tensor.
- **update** – Boolean tensor indicating whether this call happens during an update.

Returns The optimization operation.

tf_regularization_losses(*states*, *internals*, *update*)

Creates the TensorFlow operations for calculating the regularization losses for the given input states.

Parameters

- **states** – Dict of state tensors.
- **internals** – List of prior internal state tensors.
- **update** – Boolean tensor indicating whether this call happens during an update.

Returns Dict of regularization loss tensors.

MemoryAgent

class tensorforce.agents.**MemoryAgent**(*states_spec*, *actions_spec*, *config*)

Bases: tensorforce.agents.agent.Agent

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

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**(*states_spec*, *actions_spec*, *config*)

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_timestep`: bool optionally keep the last observation for use in the next batch

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

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`**(*states_spec*, *actions_spec*, *network_spec*, *config*)

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_q_model`: boolean indicating whether to use a double q-model.
- `clip_loss`: float if not 0, uses the huber loss with clip_loss as the linear bound
- *scope*: TensorFlow variable scope name (default: 'vpg')
- `batch_size`: Positive integer (**mandatory**)
- *learning_rate*: positive float (default: 1e-3)
- *discount*: Positive float, at most 1.0 (default: 0.99)
- *normalize_rewards*: Boolean (default: false)
- *entropy_regularization*: None or positive float (default: none)
- *optimizer*: Specification dict (default: Adam with learning rate 1e-3)
- *state_preprocessing*: None or dict with (default: none)
- *exploration*: None or dict with (default: none)
- *reward_preprocessing*: None or dict with (default: none)
- *log_level*: Logging level, one of the following values (default: 'info')
  - 'info', 'debug', 'critical', 'warning', 'fatal'
- *summary_logdir*: None or summary directory string (default: none)
- *summary_labels*: List of summary labels to be reported, some possible values below (default: 'total-loss')
  - 'total-loss'
  - 'losses'
  - 'variables'
  - 'activations'
  - 'relu'
- *summary_frequency*: Positive integer (default: 1)

Normalized Advantage Functions

class `tensorforce.agents.`**`NAFAgent`**(*states_spec*, *actions_spec*, *network_spec*, *config*)
Bases: `tensorforce.agents.memory_agent.MemoryAgent`

NAF: https://arxiv.org/abs/1603.00748

- *scope*: TensorFlow variable scope name (default: 'vpg')
- `batch_size`: Positive integer (**mandatory**)
- *learning_rate*: positive float (default: 1e-3)
- *discount*: Positive float, at most 1.0 (default: 0.99)
- *normalize_rewards*: Boolean (default: false)
- *entropy_regularization*: None or positive float (default: none)
- *optimizer*: Specification dict (default: Adam with learning rate 1e-3)
- *state_preprocessing*: None or dict with (default: none)
- *exploration*: None or dict with (default: none)
- *reward_preprocessing*: None or dict with (default: none)
- *log_level*: Logging level, one of the following values (default: 'info')
  - 'info', 'debug', 'critical', 'warning', 'fatal'
- *summary_logdir*: None or summary directory string (default: none)
- *summary_labels*: List of summary labels to be reported, some possible values below (default: 'total-loss')
  - 'total-loss'
  - 'losses'
  - 'variables'
  - 'activations'
  - 'relu'
- *summary_frequency*: Positive integer (default: 1)

Deep-Q-learning from demonostration (DQFD)

class `tensorforce.agents.`**`DQFDAgent`**(*states_spec*, *actions_spec*, *network_spec*, *config*)
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 .. code-block:

```
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

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** –

pretrain(*steps*)
Computes pretrain updates.

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

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.

Parameters **batch** –

Vanilla Policy Gradient

class tensorforce.agents.**VPGAgent**(*states_spec*, *actions_spec*, *network_spec*, *config*)
Bases: tensorforce.agents.batch_agent.BatchAgent

Vanilla Policy Gradient agent as described by [Sutton et al. (1999)] (https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf).

- *scope*: TensorFlow variable scope name (default: 'vpg')
- batch_size: Positive integer (**mandatory**)
- *discount*: Positive float, at most 1.0 (default: 0.99)
- *normalize_rewards*: Boolean (default: false)
- *entropy_regularization*: None or positive float (default: none)
- *gae_lambda*: None or float between 0.0 and 1.0 (default: none)
- *optimizer*: Specification dict (default: Adam with learning rate 1e-3)
- *baseline_mode*: None, or one of 'states' or 'network' specifying the baseline input (default: none)
- *baseline*: None or specification dict, or per-state specification for aggregated baseline (default: none)
- *baseline_optimizer*: None or specification dict (default: none)
- *state_preprocessing*: None or dict with (default: none)
- *exploration*: None or dict with (default: none)
- *reward_preprocessing*: None or dict with (default: none)
- *log_level*: Logging level, one of the following values (default: 'info')
  - 'info', 'debug', 'critical', 'warning', 'fatal'
- *summary_logdir*: None or summary directory string (default: none)

- *summary_labels*: List of summary labels to be reported, some possible values below (default: 'total-loss')
  - 'total-loss'
  - 'losses'
  - 'variables'
  - 'activations'
  - 'relu'
- *summary_frequency*: Positive integer (default: 1)

Trust Region Policy Optimization (TRPO)

class `tensorforce.agents.`**`TRPOAgent`**(*states_spec*, *actions_spec*, *network_spec*, *config*)
Bases: `tensorforce.agents.batch_agent.BatchAgent`

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

- *scope*: TensorFlow variable scope name (default: 'trpo')
- `batch_size`: Positive integer (**mandatory**)
- *learning_rate*: Max KL divergence, positive float (default: 1e-2)
- *discount*: Positive float, at most 1.0 (default: 0.99)
- *entropy_regularization*: None or positive float (default: none)
- *gae_lambda*: None or float between 0.0 and 1.0 (default: none)
- *normalize_rewards*: Boolean (default: false)
- *likelihood_ratio_clipping*: None or positive float (default: none)
- *baseline_mode*: None, or one of 'states' or 'network' specifying the baseline input (default: none)
- *baseline*: None or specification dict, or per-state specification for aggregated baseline (default: none)
- *baseline_optimizer*: None or specification dict (default: none)
- *state_preprocessing*: None or dict with (default: none)
- *exploration*: None or dict with (default: none)
- *reward_preprocessing*: None or dict with (default: none)
- *log_level*: Logging level, one of the following values (default: 'info')
  - 'info', 'debug', 'critical', 'warning', 'fatal'
- *summary_logdir*: None or summary directory string (default: none)
- *summary_labels*: List of summary labels to be reported, some possible values below (default: 'total-loss')
  - 'total-loss'
  - 'losses'

  - 'variables'
  - 'activations'
  - 'relu'
- *summary_frequency*: Positive integer (default: 1)

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

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 minimal value and divide by range.

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.

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)

    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()
```

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, 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

V