rlgraph Documentation

Release 0.0.3

RLgraph authors

Oct 17, 2018

Contents

RLgraph is a library for designing flexible reinforcement learning graphs

CHAPTER 1

Introduction to RLgraph

CHAPTER 2

The Space Classes

2.1 What is a Space?

Spaces in RLgraph are used to define the types and shapes of data flowing into, between, and from the different machine learning components. For example, an environment may output an RGB color image (e.g. 80x80 pixels) at any time step, which our RL algorithm will use as its only input to determine the next action. An RGB color image is usually a member of an int-type space with the exact data type being uint8 (meaning the value can range from 0 to 255) and with a shape of [width x height x 3], where the 3 represents the three color channels: red, green, and blue. Values of 0 mean no intensity in a color channel, 255 means full intensity.

2.1.1 Difference between Space and Tensor

Whereas a space defines the constraints in terms of dimensionality and data type of some data, a tensor is an members (or an instance) of a Space. Tensors hold actual numeric values (other than a space), which obey the rules and bounds of the given Space.

2.1.2 Ranks, Dimensions and Shapes

The rank of an example image tensor is 3, since we have - for each image instance - a 3D cube of numbers defining the exact look of that image. The 3 ranks stand for the width, height and color depth of the image. An example rank-3 tensor is shown in the figure below.

A space's or a tensor's rank is often confused with it's dimensions. In RLgraph, we speak of dimensions only as the size of each rank. Compare this to the figure above: The dimensions for each rank are 4 (1st rank), 3 (2nd rank), and 3 (3rd rank). In the tensorflow documentation, you will often find the term nD tensor for a rank-n tensor, which is a little misleading.

Another often used term for the set of all dimensionality values is the "shape" and it's often provided as a tuple. The shape of our example image is (80, 80, 3) and this shape tuple is sufficient to determine both a space's rank (len(shape)) and its dimensions (shape[0], shape[1], and shape[2]).

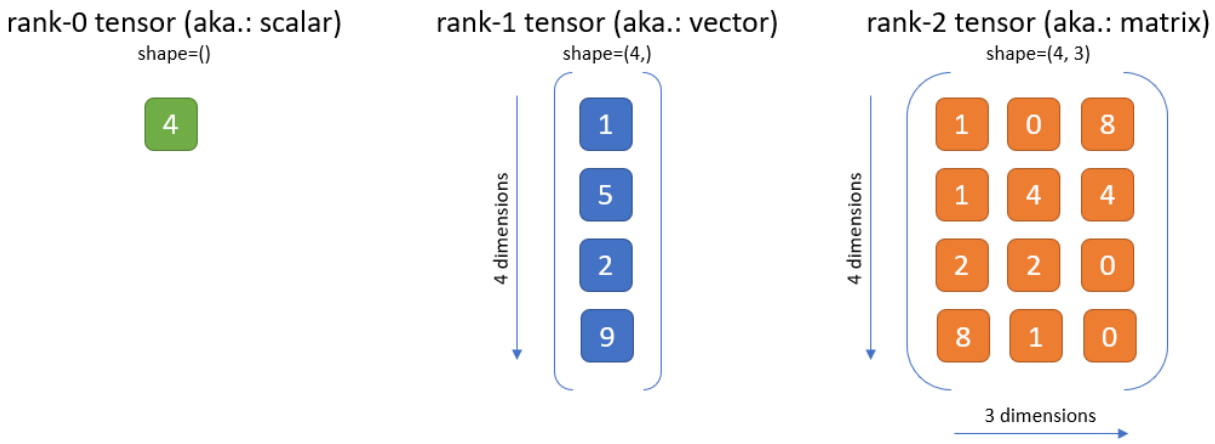

Fig. 1: Above: Examples for a rank-0 (scalar), a rank-1 (vector), and a rank-2 (matrix) int-type tensor.

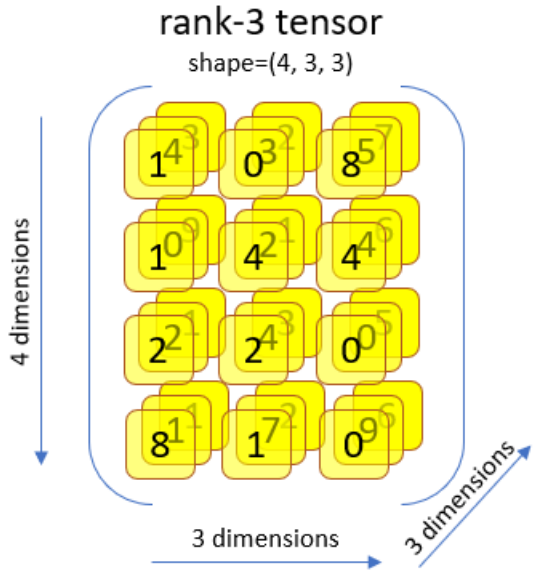

Fig. 2: Above: Example for a rank-3 tensor (e.g. an RGB-image).

2.2 There are two major types of Spaces: BoxSpaces and ContainerSpaces.

2.2.1 Box Spaces

A BoxSpace is simply an n-dimensional cube of numbers (or strings), where the numbers must all be of the same data type ("dtype" from here on). RLgraph's dtypes are based on the numpy type system and supported types are np.ints (such as np.int32 or np.uint8), np.floats (e.g. np.float32), np.bool_, as well as a box type for String data, which has the dtype np.string_. The rank of a box can be anything from rank-0 (a single scalar value), rank-1 (a vector of values), rank-2 (a matrix of values), rank-3 (a cube), to any higher dimensional box. All tensors shown in the figures on top are examples for box-space tensors.

2.2.2 Container Spaces

Container Spaces can contain Box Spaces as well as other Container Spaces in an arbitrarily nested fashion. The two supported container types are Tuple and Dict.

A Tuple is an ordered sequence of other Spaces (similar to a python tuple). For example: An environment that produces an RGB image and a text string at each time step could have the space: Tuple(IntBox(80,80,3, np.uint8), TextBox()).

Another example for a tuple space is shown below:

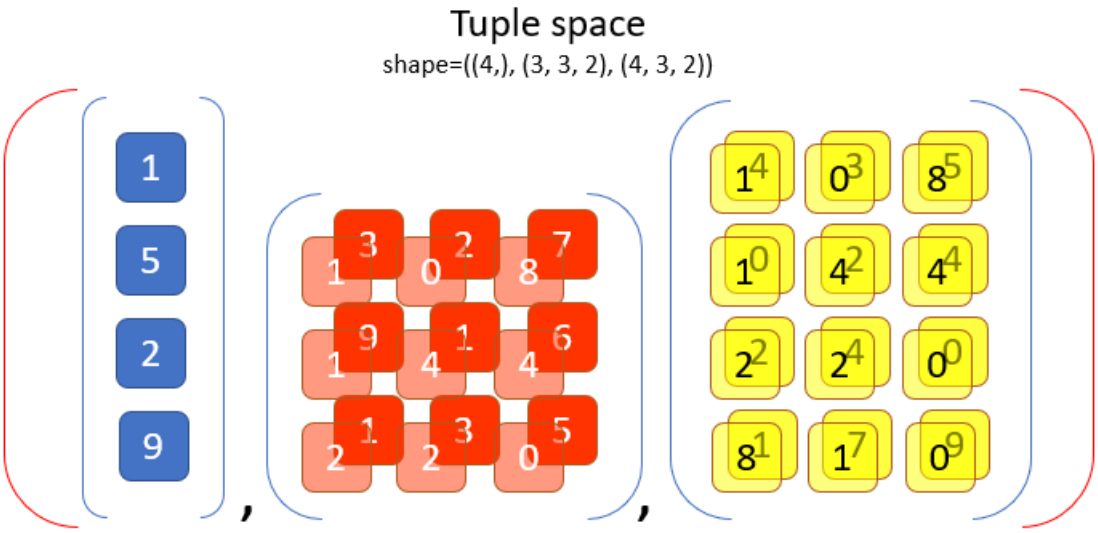

Fig. 3: Above: Example Tuple space with 3 box-type child-spaces. *Note* that a Tuple can contain other container spaces in a deeply nested fashion. Shown here is only a relatively simple Tuple with only box spaces as single elements.

Another way to describe this space is through a keyed Dict space (similar to python dicts), with the keys "image" and "text". For example: Dict({"image": IntBox(80,80,3, np.uint8), "text": TextBox()}).

Containers are fully equivalent to Box Space classes in that they also have shapes and dtypes. However, these are represented by heterogeneous tuples. Our Image-and-Text Dict space from above, for example, would have a shape of ((80,80,3),()), a rank of (3, 0) and a dtype of (np.uint8, np.string_).

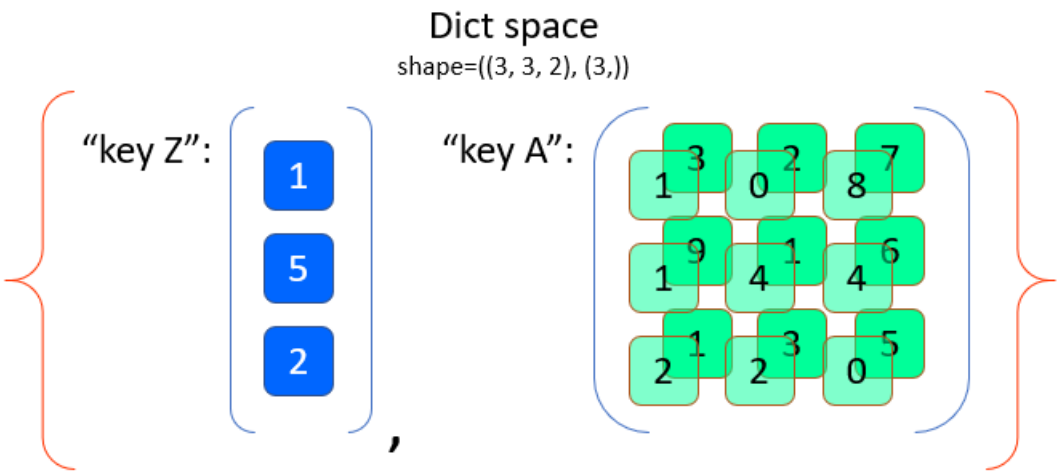

Fig. 4: Above: Example Dict space with 2 keys, each one holding a box-type child-space. Note here that for Dict spaces, the order of the keys are sorted alphabetically before generating shape, rank and dtype tuples. In this case: “key A” comes before “key Z”. For Tuple spaces, this order is given by the sequence of sub-Spaces inside the Tuple. Nested Container Spaces (e.g. a Dict inside another Dict) generate equally nested shape, rank and dtype tuples.

2.3 Special Ranks of BoxSpaces.

TODO: Describe the generic special ranks once we have them implemented (will replace the current special ranks: batch and time).

CHAPTER 3

The Environment Classes

3.1 What is an environment?

The generic reinforcement learning problem can be broken down into the following ever repeating sequence of steps. This sequence is also often referred to as a Markov Decision Process (MDP). An environment describes a particular MDP:

- An agent that "lives" in some environment observes the current state of that environment. The state could be anything from a simple x/y position tuple of the agent to an image of a camera (that the agent has access to) or a line of text from the agent's chat partner (the agent is a chat bot). The nature of this state signal - its data type and shape - is called the "state space".
- Based on that state observation, the agent now picks an action. The environment dictates from which space this action may be chosen. For example, one could think of a computer game environment, in which an agent has to move through a 2D maze and can pick the actions up, down, left, and right at each time step. This space from which to chose is called the "action space". In RLgraph, both state- and action spaces are usually given by the environment.
- The chosen action is now executed on the environment (e.g. the agent decided to move left) and the environment changes because of that action. This change is described by the transition function $P(S'|S, A)$, which outputs the probability for ending up in next state *S'* given that the agent chose action *A* after having observed state *S*.
- Besides producing a next state (*S'*), the environment also emits a reward signal (*R*), which is determined by the reward function $R(S'|S, A)$, which describes the expected reward when reaching state *S'* after having chosen action *A* in (the previous) state *S*.
- We now start this procedure again, using *S'* as our new observation. We pick another action, change the environment through that action (transition function *P*), observe the next state (*S''*) and receive another reward (R_{t+1}), and so on and so forth (see figure above).
- The collected rewards (R_t) can be used by the agent for learning to act smarter, in fact the reinforcement learning incentive is to pick actions in such a way as to maximize the accumulated reward over some amount of time (usually some episode, after which the environment must be reset to some initial state).

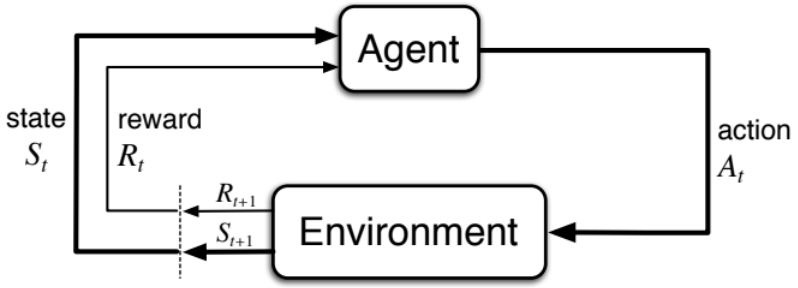

Fig. 1: Above: The basic cycle in reinforcement learning. |br| (source: Reinforcement Learning - An Introduction - 2nd Edition)

3.2 RLgraph's environment adapters.

RLGraph supports many popular environment types and standards and offers a common interface into all these. The base class is the *Environment* and its most important API-methods are *reset* (to reset the environment) and *step* (to execute an action). In the following, we will briefly describe the different supported environment types. If you are interested in writing your own environments for your own RL research, we will be very happy to receive your pull request. For more information on our environments, see the environment reference documentation.

3.2.1 OpenAI Gym

The OpenAI Gym standard is the most widely used type of environment in reinforcement learning research. It contains the famous set of Atari 2600 games (each game has a RAM state- and a 2D image version), simple text-rendered grid-worlds, a set of robotics tasks, continuous control tasks (via the MuJoCO physics simulator), and many others.

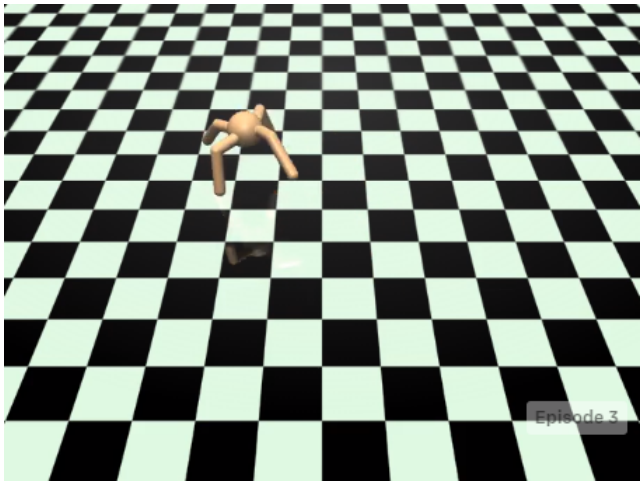

Fig. 2: Above: The "Ant-v2" environment, one of the many MuJoCo-simulator tasks of the OpenAI Gym. (source: openAI).

RLgraph's OpenAIGymEnv class serves as an adapter between RLgraph code and any of these openAI Gym environments. For example, in order to have your agent learn how to play Breakout from image pixels, you would create the environment under RLgraph via:

```
from rlgraph.environments import OpenAIGymEnv
# Create the env.
breakout_env = OpenAIGymEnv("Breakout-v0", visualize=True)
# Reset the env.
breakout_env.reset()
# Execute 100 random actions in the env.
for _ in range(100):
    state, reward, is_terminal, info = breakout_env.step(breakout_env.action_space.
↪sample())
    # Reset if terminal state was reached.
    if is_terminal:
        breakout_env.reset()
```

3.2.2 Deepmind Lab

Deepmind Lab is Google DeepMind's environment of choice for their advanced RL research. It's a fully customizable suite of 3D environments (including mazes and other interesting worlds), which are usually navigated by the agent through a 1st person's perspective.

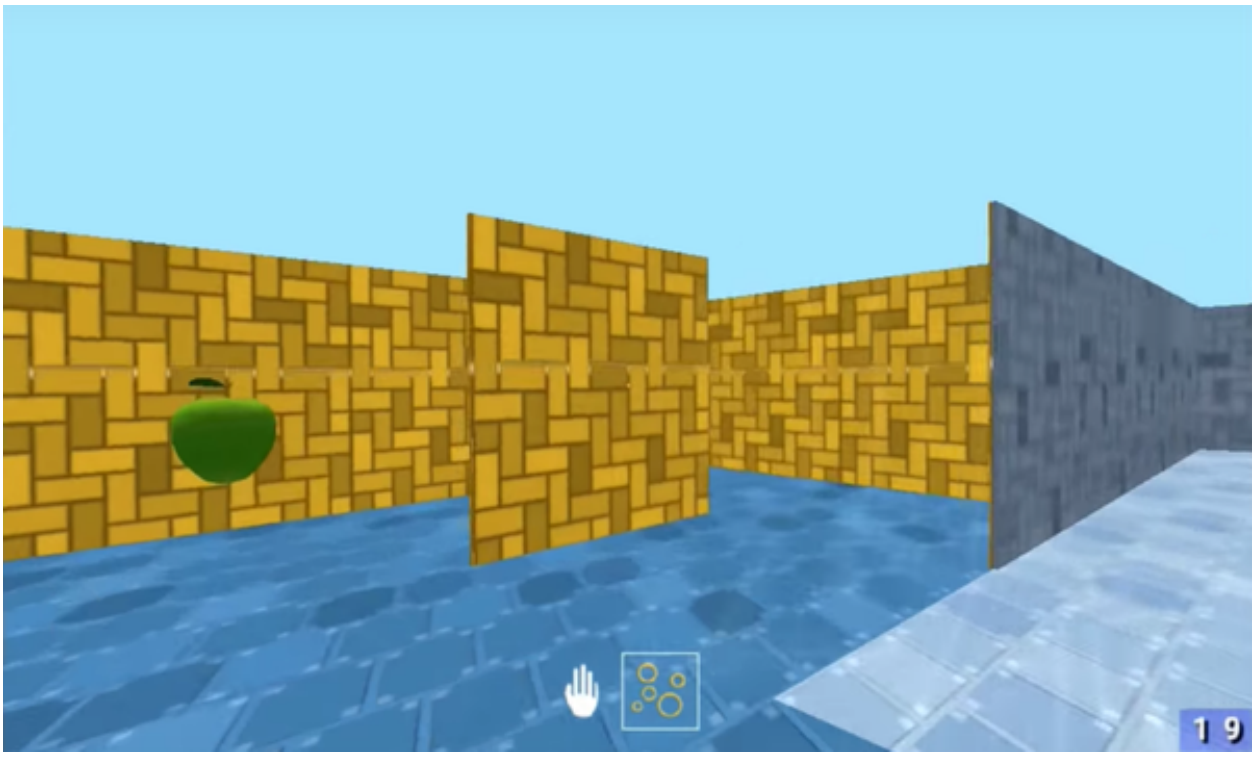

Fig. 3: Above: The "Nav Maze Arena" environment of the DM Lab. (source: deepmind).

Different state observation items can be configured as needed at environment construction time, e.g. an image capturing the 1st person view from inside the maze or a textual input offering instructions on where to go next (e.g. "blue ladder"). When using more than one state observation items, the Rlgraph state space will be a Dict with the keys describing the nature of the different observation items (e.g. "RGB_INTERLEAVED" for an RGB image, "INSTR" for the instruction string).

DM Lab itself (and hence also its RLgraph adapter) is somewhat hard to install and only runs on Linux and Mac. For details, you can take a look at our Docker file to see which steps are required in order to get it up and running.

3.2.3 Simple Grid Worlds

Grid worlds are a great way to quickly test the learning capabilities of our agents. They are simple worlds with square fields on which an agent can move up, down, left or right. There are walls, through which an agent cannot move, fire, on which a negative reward is collected, holes, into which an agent will fall to collect a negative reward and end the episode, a starting state, from which the agent starts after a reset, and a goal state, which the agent has to reach in order to end the episode and to collect a large reward.

RLgraph comes with its own GridWorld environment class that can be customized in its map (dimensions, locations of walls, etc..), the transition- and the reward function.

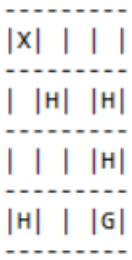

Fig. 4: Above: The 4x4 grid world showing the agent's position (X), some holes (H) and the to-be-reached goal state (G). Allowed actions are up, down, left and right.

Check out the reference documentation on the GridWorld and other supported environments here.

CHAPTER 4

What is an RLgraph Component?

Components are the basic building blocks, which you will use to build any machine learning and reinforcement learning models with. A component is the smallest unit, which can be run and tested in and by itself via RLgraph's different executor and testing classes. RLgraph components span from simple (and single) neural network layers to highly complex policy networks, memories, optimizers and mathematical components, such as loss functions.

Each component contains:

- ... any number of sub-components, each of which may again contain their own sub-components (also sometimes called "child components"). Hence components are arbitrarily nestable inside each other.
- ... at least one API-method, so that clients of the component (in the end this will be our reinforcement learning agent) can use it.
- ... any number of "graph functions", which are special component methods, which contain the actual computation code. These are the only places, where you will find backend (tensorflow, pytorch, etc..) specific code.
- ... any number of variables for the component to use for its computations (graph functions).

On the following page, we will walk through building our own custom component, which will include all of the above items. But let's first talk in some more detail about RLgraph's Component base class.

4.1 The Component Base Class

The *Component* base class contains the core functionality, which every RLgraph Component inherits from. Its most important methods are listed below. For a more detailed overview, please take a look at the Component reference documentation.

1. *add_components*: This method is used to add an arbitrary number of sub-components to the component itself.
2. *check_input_spaces*: Can be used to sanity check the incoming spaces (see the documentation on RLgraph's Space classes) of all API-method call arguments.

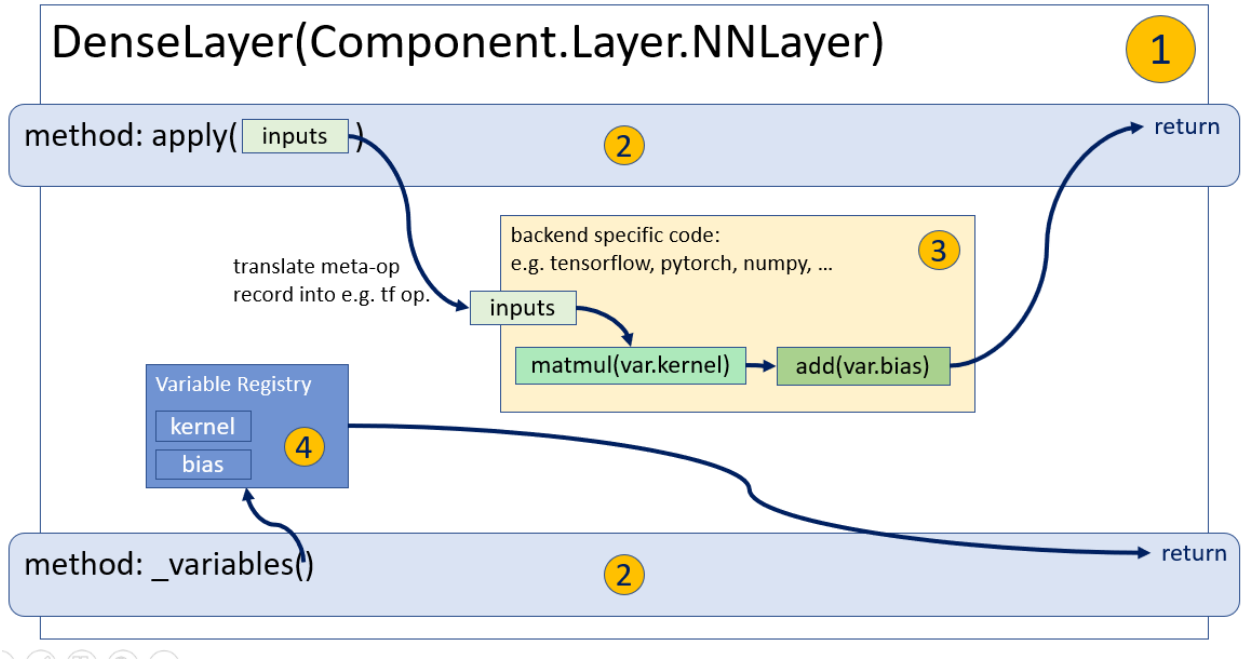

Fig. 1: Above: A DenseLayer component (1) with two API-methods (2), one graph function (3) and two variables (kernel and bias) (4).

3. *create_variables*: This method is called automatically by the RLgraph build system and can be implemented in order to create an arbitrary number of variables used by the component's computation functions ("graph functions").
4. *copy*: Copies the component and returns a new Component object that is identical to the original one. This is useful, for example, to create a target network as a copy of the main policy network in a DQN-type agent.

4.1.1 API-Methods

A component's API-methods are its outside facing handles through which clients of the component (either another component or an agent that contains the component in question) can access and control its behavior. For example, a typical memory component would need an *insert_records* API-method to insert some data into the memory, a *get_records* method to retrieve a certain number of already stored records, and maybe a *clear* method to wipe out all stored information from the memory.

API-methods are normal class methods, but must be tagged with the *@rlgraph_api* decorator, which can be imported as follows:

```
from rlgraph.utils.decorators import rlgraph_api
```

An API-method can have any arbitrary combination of regular python args and kwargs, as well as define default values for some of these. For example:

```
# inside some component class ...
...
@rlgraph_api
def my_api_method(self, a, b=5, c=None):
    # do and return something
```

Calling the above API-method (e.g. from its parent component) requires the call argument *a* to be provided, whereas *b* and *c* are optional arguments. As you may recall from the spaces chapter, information in RLgraph is passed around between components within fixed space constraints. In fact, each API-method call argument (*a*, *b*, and *c* in our example above) has a dedicated space after the final graph has been built from all components in it.

Important note: Up until now, if an API-method is called more than once by the component's client(s), the spaces of the provided call arguments (e.g. the space of *a*) in the different API-calls have to match. So if in the first call, *a* is an IntBox, in the second call, it has to be an IntBox as well. This is because of a possible dependency of the component's variables (see below) on these "input-spaces". We will try to further loosen this restriction in future releases and only require it if RLgraph knows for sure that the space of the argument in question is being used to define variables of the component.

4.1.2 Variables

Variables are the data that each component can store for the lifetime of the computation graph. A variable has a fixed data type and shape, hence a fixed Rlgraph space. As a matter of fact, variables are often created directly from *Space* instances via the practical *Space.get_variable()* method.

Variables can be accessed inside graph functions (see below) and can be read as well as be written to. Examples for variables are:

- The buffer of a memory that stores a certain part of a memory record, for example an image (rank-3 uint8 tensor).
- A memory component's index pointer (which record should we retrieve next?). This is usually a single int scalar.
- The weights matrix of some neural network layer. This is always a rank-2 float tensor.

Variables are created in a component's *create_variables* method, which gets called automatically, once all input spaces of the component (all its API-method arguments' spaces) are known to the RLgraph build system. In the next paragraph, we will explain how this stage of "input-completeness" is reached and why it's important for the component.

4.1.3 Input Spaces and the concept of "input-completeness"

Let's look at a Component's API-method and its variable generating code to understand the concept of "input-completeness".

```
# inside some component class ...
...
@rlgraph_api
def insert(self, record):
    # Call a graph function that will take care of the assignment.
    return self._graph_fn_insert(record)

def create_variables(input_spaces, action_space=None):
    """
    Override this base class method to create variables based on the
    spaces that are underlying each API-method's call argument
    (in our case, this is only the call arg "records" of the "insert" API-method).
    """
    # Lookup the input space by the name of the API-method's call arg ("record").
    in_space = input_spaces["record"]
    self.storage_buffer = in_space.get_variable(trainable=False, ... other options)
```

A component reaches input-completeness, if all spaces to all its unique call parameters (by their names) are known. A space for a call argument (e.g. *record*) gets known once the respective API-method (here: *insert*) gets called by a

client (a parent component). Only the outermost component, also called the "root", needs its spaces to be provided manually by the user, since its API-methods are only executed (called) at graph-execution time.

If a component has many API-methods, each with the only call argument *a* , which share the call parameter's names (e.g. a component has API-methods: *one(a, b)*)

A client of this component (a parent component or the RL agent directly) will eventually make a call to the component's API-method *insert()*. At that point, the space of the *record* argument will be known. If the component above only has that one API-method, and hence only that one API-method call argument (*record*), it is then input-complete.

4.1.4 Graph Functions

Every component serves a certain computation purpose within a machine learning model. A neural network layer maps input data to output data via, for example, a matrix-matrix multiplication (and adding maybe some bias). An optimizer calculates the gradient of a loss function over the weights of a trainable layer and applies the resulting gradients in a certain way to these weights. All these calculation steps happen inside a component's graph functions, the only place in RLgraph, were we can find backend specific code, such as calls to tensorflow's static graph building functions or computations on pytorch tensors.

Unlike API-methods, graph functions can only be called from within the same component that owns them (not by parents or grandparents of the component). These calls happen from within the component's different API-methods (similar to calling another API-method).

Graph functions are - similar to API-methods - regular python class methods, but must be tagged with the *@graph_fn* decorator as follows:

```
# inside some component class ...
...
@graph_fn
def _graph_fn_do_some_computation(self, a, b):
    # All backend-specific code in RLgraph goes into graph functions.
    if get_backend() == "tf":
        # Do some computation in tf.
        some_result = tf.add(a, b)

    elif get_backend() == "pytorch":
        # Do some computation in pytorch.
        some_result = a + b

    return some_result
```

Inside a graph function, any type of backend specific computations are allowed to be coded. A graph function then returns the result of the computation or many results as a tuple.

CHAPTER 5

How to Write Your Own Custom Component

In the following, we will build an entire component from scratch in RLgraph, including the component's API-methods, its graph functions, and its variable generating code.

5.1 A Simple Single-Value Memory Component

Our component, once done, will look as follows:

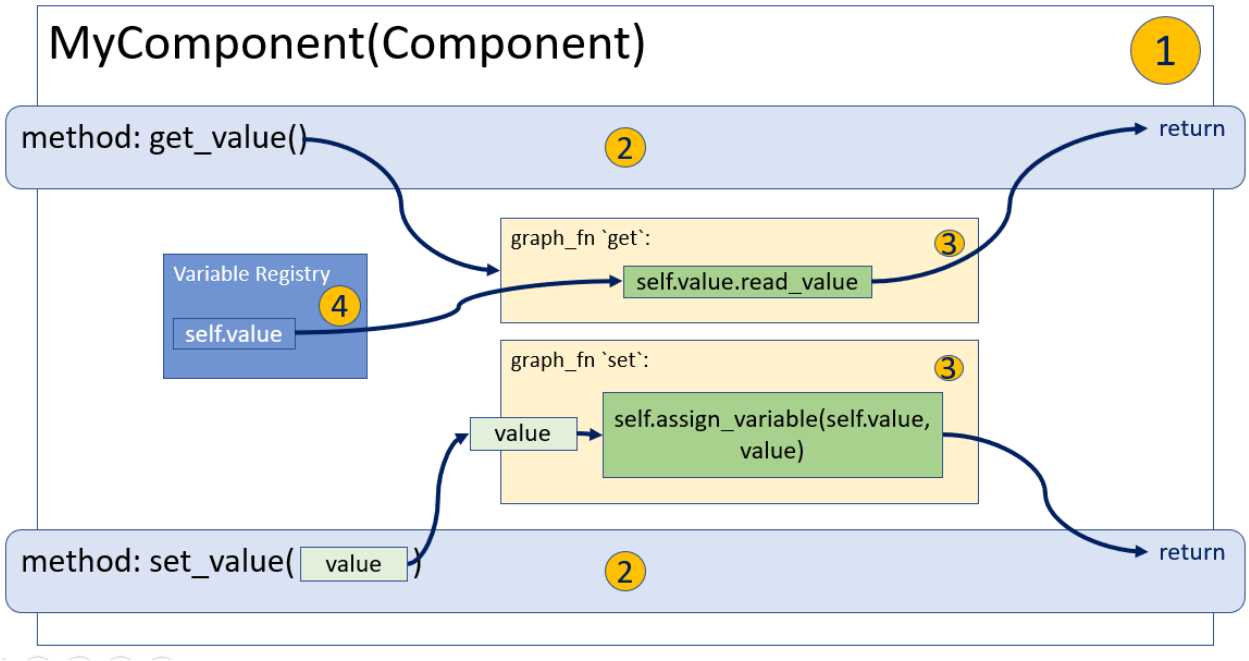

Fig. 1: Above: The custom memory component we are about to build from scratch.

We are building a simplistic memory that holds some value (or a tensor of values) in a variable stored under *self.value*. Clients of our component will be able to read the current value via the API-method *get_value*, overwrite it via *set_value*, and do some simple calculation by calling *get_value_plus_n* (which is not shown in the figure above), which adds some number (*n*) to the current value of the variable and returns the result of that computation.

5.1.1 Class Definition and Constructor

First we will create a new python file called *my_component.py* and will import all necessary RLgraph modules as well as *tensorflow*, which will be the only supported backend of this component for simplicity reasons.

```
import tensorflow as tf
from rlgraph.components.component import Component
from rlgraph.utils.decorators import rlgraph_api, graph_fn

# Define our new Component class.
class MyComponent(Component):
    # Ctor.
    def __init__(self, initial_value=1.0, scope="my-component", **kwargs):
        # It is good practice to pass through **kwargs to parent class.
        super(MyComponent, self).__init__(scope, **kwargs)
        # Store the initial value.
        # This will be assigned equally to all items in the memory.
        self.initial_value = initial_value
        # Placeholder for our variable (will be created inside self.create_variables).
        self.value = None
```

5.1.2 API-Methods and Input Spaces

Let's now define all our API-methods. Each of these will simply make a call to an underlying graph function in which the actual magic is implemented. Note that all API-methods must be tagged with the *@rlgraph_api* decorator:

```
@rlgraph_api
def get_value(self):
    return self._graph_fn_get()

@rlgraph_api
def set_value(self, value):
    return self._graph_fn_set(value)

@rlgraph_api
def get_value_plus_n(self, n):
    return self._graph_fn_get_value_plus_n(n)
```

Note that the set of our API-method call arguments is now: *value* and *n*. The spaces of both *value* and *n* must thus be known to the RLgraph build system, before the *create_variables()* method will be called automatically. In case our component is the root component, the user will have to provide these spaces manually in the Agent (which is the owner of the root). Remember that this manual space is always necessary for all of the root component's API-method call arguments).

5.1.3 The Single Value Variable

Now it's time to specify, which variables our component needs. All variables should be generated inside a component's *create_variables* method, which is called automatically, once all input spaces are known. In our case, the input space

for the *value* arg is important as that signals us, which type of variable we want (rank, dtype, etc.). We can apply some restrictions to this space by implementing the *check_input_spaces()* method, which gets called (automatically) right before *create_variables*. For example:

```
# Add this to the import section at the top of the file
from rlgraph.spaces import ContainerSpace

# Then, in our component class ...

def check_input_spaces(self, input_spaces, action_space=None):
    # Make sure, we have a non-container space.
    in_space = input_spaces["value"]
    assert not isinstance(in_space, ContainerSpace), "ERROR: No containers allowed!"
```

The above code will make sure that only simple spaces are allowed as our variable space (e.g. a FloatBox with some arbitrary shape).

Now that we have sanity checked our variable space, let's write the code to create the variable:

```
def create_variables(self, input_spaces, action_space=None):
    in_space = input_spaces["value"]
    # Create the variable as non-trainable and with
    # the given initial value (from the c'tor).
    self.value = in_space.get_variable(
        trainable=False, initializer=self.initial_value
    )
```

5.1.4 Under the Hood Coding: Our Graph Functions

Finally, we need to implement the actual under-the-hood computation magic using our favourite machine learning backend. We currently support tensorflow and pytorch, but if you are interested in other backends, we would love to receive your contributions on this and PRs (see here for our contrib guidelines).

We will implement three graph functions, exactly those three that we have already been calling from within our API-methods. Graph functions usually start with *_graph_fn_* and we should stick to that convention here as well. The exact code for all three is shown below. Note the sudden change from abstract glue-code like coding to actual tensorflow code. Graph functions can return one or more (a tuple) tensorflow ops. But we will also later learn (when we write an entire algorithm from scratch) how to bundle and nest these ops into more complex tuple and dict structures and return these to the API-method caller.

```
@graph_fn
def _graph_fn_get(self):
    # Note: read_value() is the tf way to make sure a read op is added to the graph.
    # (remember that self.value is an actual tf.Variable).
    return self.value.read_value()

@graph_fn
def _graph_fn_set(self, value):
    # We use the RLgraph `Component.assign_variable()` helper here.
    assign_op = self.assign_variable(self.value, value)
    # Make sure the value gets assigned via the no_op trick
    # (no_op is now dependent on the assignment op).
    with tf.control_dependencies([assign_op]):
        return tf.no_op()

@graph_fn
```

(continues on next page)

(continued from previous page)

```
def _graph_fn_get_value_plus_n(self, n):
    # Simple tf.add operation as return value.
    return tf.add(self.value, n)
```

It might seem a little strange that our API-methods in this example are only very thin wrappers around the actual computations (graph functions). However, in a later chapter on agent implementations, we will see how useful API-methods really are, not for wrapping calls to graph functions, but to delegate and connect different graph functions and also other API-methods with each other.

CHAPTER 6

The Complete Code for Our Custom Component

Here you can see the complete code for our custom component. On the next page, we will talk about how we can test this component via RLgraph's special *ComponentTest* class.

```
import tensorflow as tf
from rlgraph.components.component import Component
from rlgraph.utils.decorators import rlgraph_api, graph_fn
# To be able to do input-space sanity checking.
from rlgraph.spaces import ContainerSpace

# Define our new Component class.
class MyComponent(Component):
    # Ctor.
    def __init__(self, initial_value=1.0, scope="my-component", **kwargs):
        # It is good practice to pass through **kwargs to parent class.
        super(MyComponent, self).__init__(scope, **kwargs)
        # Store the initial value.
        # This will be assigned equally to all items in the memory.
        self.initial_value = initial_value
        # Placeholder for our variable (will be created inside self.create_variables).
        self.value = None

@rlgraph_api
def get_value(self):
    return self._graph_fn_get()

@rlgraph_api
def set_value(self, value):
    return self._graph_fn_set(value)

@rlgraph_api
def get_value_plus_n(self, n):
    return self._graph_fn_get_value_plus_n(n)

def check_input_spaces(self, input_spaces, action_space=None):
```

(continues on next page)

(continued from previous page)

```
    # Make sure, we have a non-container space.
    in_space = input_spaces["value"]
    assert not isinstance(in_space, ContainerSpace), "ERROR: No containers allowed!"

def create_variables(self, input_spaces, action_space=None):
    in_space = input_spaces["value"]
    # Create the variable as non-trainable and with
    # the given initial value (from the c'tor).
    self.value = in_space.get_variable(
        trainable=False, initializer=self.initial_value
    )

@graph_fn
def _graph_fn_get(self):
    # Note: read_value() is the tf way to make sure a read op is added to the graph.
    # (remember that self.value is an actual tf.Variable).
    return self.value.read_value()

@graph_fn
def _graph_fn_set(self, value):
    # We use the RLgraph `Component.assign_variable()` helper here.
    assign_op = self.assign_variable(self.value, value)
    # Make sure the value gets assigned via the no_op trick
    # (no_op is now dependent on the assignment op).
    with tf.control_dependencies([assign_op]):
       return tf.no_op()

@graph_fn
def _graph_fn_get_value_plus_n(self, n):
    # Simple tf.add operation as return value.
    return tf.add(self.value, n)
```

CHAPTER 7

How to Test Your Components

Now we will show you, how one can very easily test a single component via RLgraph's testing system. As an example, we will use our custom component built from scratch in this chapter here.

7.1 Writing a New Test Case with Python's Unittest Module

7.1.1 Test 1: Writing a New Value

7.1.2 Test 2: Retrieving the Value

7.1.3 Test 3: Testing for the Correct Computation Results

CHAPTER 8

RLgraph API Reference Documentation

8.1 RLgraph Core API

rlgraph.**get_backend**()

rlgraph.**get_distributed_backend**()

8.2 Space Classes and Space Utilities

8.2.1 Space Base Class Reference

class rlgraph.spaces.space.**Space**(*add_batch_rank=False*, *add_time_rank=False*, *time_major=False*)

Bases: rlgraph.utils.specifiable.Specifiable

Space class (based on and compatible with openAI Spaces). Provides a classification for state-, action-, reward- and other spaces.

contains(*sample*)

Checks whether this space contains the given sample. This is more for testing purposes.

Args: sample: The element to check.

Returns: bool: Whether sample is a valid member of this space.

flat_dim

Returns: int: The length of a flattened vector derived from this Space.

flatten(*mapping=None*, *custom_scope_separator='/'*, *scope_separator_at_start=True*, *scope_=None*, *list_=None*)

A mapping function to flatten this Space into an OrderedDict whose only values are primitive (non-container) Spaces. The keys are created automatically from Dict keys and Tuple indexes.

Args:

mapping (Optional[callable]): A mapping function that takes a flattened auto-generated key and a primitive Space and converts the primitive Space to something else. Default is pass through.

custom_scope_separator (str): The separator to use in the returned dict for scopes. Default: '/'.

scope_separator_at_start (bool): Whether to add the scope-separator also at the beginning. Default: True.

scope_ (Optional[str]): For recursive calls only. Used for automatic key generation.

list_ (Optional[list]): For recursive calls only. The list so far.

Returns:

OrderedDict: The OrderedDict using auto-generated keys and containing only primitive Spaces (or whatever the mapping function maps the primitive Spaces to).

force_batch(*samples*)

Makes sure that *samples* is always returned with a batch rank no matter whether it already has one or not (in which case this method returns a batch of 1) or whether this Space has a batch rank or not.

Args: samples (any): The samples to be batched. If already batched, return as-is.

Returns: any: The batched sample.

get_shape(*with_batch_rank=False*, *with_time_rank=False*, *time_major=None*, ***kwargs*)

Returns the shape of this Space as a tuple with certain additional ranks at the front (batch) or the back (e.g. categories).

Args:

with_batch_rank (Union[bool,int]): Whether to include a possible batch-rank as *None* at 0th (or 1st) position. If *with_batch_rank* is an int (e.g. -1), the possible batch-rank is returned as that number (instead of None) at the 0th (or 1st if time_major is True) position. Default: False.

with_time_rank (Union[bool,int]): Whether to include a possible time-rank as *None* at 1st (or 0th) position. If *with_time_rank* is an int, the possible time-rank is returned as that number (instead of None) at the 1st (or 0th if time_major is True) position. Default: False.

time_major (bool): Overwrites *self.time_major* if not None. Default: None (use *self.time_major*).

Returns: tuple: The shape of this Space as a tuple.

get_variable(*name*, *is_input_feed=False*, *add_batch_rank=None*, *add_time_rank=None*, *time_major=False*, *is_python=False*, *local=False*, ***kwargs*)

Returns a backend-specific variable/placeholder that matches the space's shape.

Args: name (str): The name for the variable.

is_input_feed (bool): Whether the returned object should be an input placeholder, instead of a full variable.

add_batch_rank (Optional[bool,int]): If True, will add a 0th (or 1st) rank (None) to the created variable. If it is an int, will add that int (-1 means None). If None, will use the Space's default value: *self.has_batch_rank*. Default: None.

add_time_rank (Optional[bool,int]): If True, will add a 1st (or 0th) rank (None) to the created variable. If it is an int, will add that int (-1 means None). If None, will use the Space's default value: *self.has_time_rank*. Default: None.

time_major (bool): Only relevant if both *add_batch_rank* and *add_time_rank* are True. Will make the time-rank the 0th rank and the batch-rank the 1st rank. Otherwise, batch-rank will be 0th and time-rank will be 1st. Default: False.

is_python (bool): Whether to create a python-based variable (list) or a backend-specific one.

local (bool): Whether the variable must not be shared across the network. Default: False.

Keyword Args: To be passed on to backend-specific methods (e.g. trainable, initializer, etc..).

Returns: any: A Tensor Variable/Placeholder.

rank

Returns: int: The rank of the Space not including batch- or time-ranks (e.g. 3 for a space with shape=(10, 7, 5)).

sample(*size=None*, *fill_value=None*)

Uniformly randomly samples an element from this space. This is for testing purposes, e.g. to simulate a random environment.

Args:

size (Optional[int]): The number of samples or batch size to sample. If size is > 1: Returns a batch of size samples with the 0th rank being the batch rank (even if *self.has_batch_rank* is False). If size is None or (1 and self.has_batch_rank is False): Returns a single sample w/o batch rank. If size is 1 and self.has_batch_rank is True: Returns a single sample w/ the batch rank.

fill_value (Optional[any]): The number or initializer specifier to fill the sample. Can be used to create a (non-random) sample with a certain fill value in all elements. TODO: support initializer spec-strings like 'normal', 'truncated_normal', etc..

Returns: any: The sampled element(s).

shape

Returns: tuple: The shape of this Space as a tuple. Without batch or time ranks.

with_batch_rank(*add_batch_rank=True*)

Returns a deepcopy of this Space, but with *has_batch_rank* set to the provided value.

Args: add_batch_rank (Union[bool,int]): The fixed size of the batch-rank or True or False.

Returns: Space: The deepcopy of this Space, but with *has_batch_rank* set to True.

with_extra_ranks(*add_batch_rank=True*, *add_time_rank=True*, *time_major=False*)

Returns a deepcopy of this Space, but with *has_batch_rank* and *has_time_rank* set to the provided value. Use None to leave whatever value this Space has already.

Args:

add_batch_rank (Optional[bool]): If True or False, set the *has_batch_rank* property of the new Space to this value. Use None to leave the property as is.

add_time_rank (Optional[bool]): If True or False, set the *has_time_rank* property of the new Space to this value. Use None to leave the property as is.

time_major (Optional[bool]): Whether the time-rank should be the 0th rank (instead of the 1st by default). Not important if either batch_rank or time_rank are not set. Use None to leave the property as is.

Returns: Space: The deepcopy of this Space, but with *has_batch_rank* set to True.

with_time_rank(*add_time_rank=True*)
Returns a deepcopy of this Space, but with *has_time_rank* set to the provided value.

Args: add_time_rank (Union[bool,int]): The fixed size of the time-rank or True or False.

Returns: Space: The deepcopy of this Space, but with *has_time_rank* set to True.

zeros(*size=None*)

Args: size (Optional): Same as *Space.sample()*.

Returns: np.ndarray: *size* zero samples where all values are zero and have the correct type.

8.2.2 Box Spaces

class rlgraph.spaces.box_space.**BoxSpace**(*low*, *high*, *shape=None*, *add_batch_rank=False*, *add_time_rank=False*, *time_major=False*, *dtype=<class 'numpy.float32'>*)
Bases: *rlgraph.spaces.space.Space*

A box in R^n with a shape tuple of len n. Each dimension may be bounded.

bounds

contains(*sample*)
Checks whether this space contains the given sample. This is more for testing purposes.

Args: sample: The element to check.

Returns: bool: Whether sample is a valid member of this space.

flat_dim
Returns: int: The length of a flattened vector derived from this Space.

force_batch(*samples*)
Makes sure that *samples* is always returned with a batch rank no matter whether it already has one or not (in which case this method returns a batch of 1) or whether this Space has a batch rank or not.

Args: samples (any): The samples to be batched. If already batched, return as-is.

Returns: any: The batched sample.

get_shape(*with_batch_rank=False*, *with_time_rank=False*, *time_major=None*, ***kwargs*)
Returns the shape of this Space as a tuple with certain additional ranks at the front (batch) or the back (e.g. categories).

Args:

with_batch_rank (Union[bool,int]): Whether to include a possible batch-rank as *None* at 0th (or 1st) position. If *with_batch_rank* is an int (e.g. -1), the possible batch-rank is returned as that number (instead of None) at the 0th (or 1st if time_major is True) position. Default: False.

with_time_rank (Union[bool,int]): Whether to include a possible time-rank as *None* at 1st (or 0th) position. If *with_time_rank* is an int, the possible time-rank is returned as that number (instead of None) at the 1st (or 0th if time_major is True) position. Default: False.

time_major (bool): Overwrites *self.time_major* if not None. Default: None (use *self.time_major*).

Returns: tuple: The shape of this Space as a tuple.

get_variable(*name*, *is_input_feed=False*, *add_batch_rank=None*, *add_time_rank=None*, *time_major=None*, *is_python=False*, *local=False*, ***kwargs*)
Returns a backend-specific variable/placeholder that matches the space's shape.

Args: name (str): The name for the variable.

is_input_feed (bool): Whether the returned object should be an input placeholder, instead of a full variable.

add_batch_rank (Optional[bool,int]): If True, will add a 0th (or 1st) rank (None) to the created variable. If it is an int, will add that int (-1 means None). If None, will use the Space's default value: *self.has_batch_rank*. Default: None.

add_time_rank (Optional[bool,int]): If True, will add a 1st (or 0th) rank (None) to the created variable. If it is an int, will add that int (-1 means None). If None, will use the Space's default value: *self.has_time_rank*. Default: None.

time_major (bool): Only relevant if both *add_batch_rank* and *add_time_rank* are True. Will make the time-rank the 0th rank and the batch-rank the 1st rank. Otherwise, batch-rank will be 0th and time-rank will be 1st. Default: False.

is_python (bool): Whether to create a python-based variable (list) or a backend-specific one.

local (bool): Whether the variable must not be shared across the network. Default: False.

Keyword Args: To be passed on to backend-specific methods (e.g. trainable, initializer, etc..).

Returns: any: A Tensor Variable/Placeholder.

zeros(*size=None*)

Args: size (Optional): Same as *Space.sample()*.

Returns: np.ndarray: *size* zero samples where all values are zero and have the correct type.

class rlgraph.spaces.int_box.**IntBox**(*low=None*, *high=None*, *shape=None*, *dtype='int32'*, ***kwargs*)

Bases: *rlgraph.spaces.box_space.BoxSpace*

A box in Z^n (only integers; each coordinate is bounded) e.g. an image (w x h x RGB) where each color channel pixel can be between 0 and 255.

contains(*sample*)

Checks whether this space contains the given sample. This is more for testing purposes.

Args: sample: The element to check.

Returns: bool: Whether sample is a valid member of this space.

flat_dim_with_categories

If we were to flatten this Space and also consider each single possible int value (assuming global bounds) as one category, what would the dimension have to be to represent this Space.

get_shape(*with_batch_rank=False*, *with_time_rank=False*, ***kwargs*)

Keyword Args:

with_category_rank (bool): Whether to include a category rank for this IntBox (if all dims have equal lower/upper bounds).

sample(*size=None*, *fill_value=None*)

Uniformly randomly samples an element from this space. This is for testing purposes, e.g. to simulate a random environment.

Args:

size (Optional[int]): The number of samples or batch size to sample. If size is > 1: Returns a batch of size samples with the 0th rank being the batch rank (even if *self.has_batch_rank* is False). If size is None or (1 and self.has_batch_rank is False): Returns a single sample w/o batch rank. If size is 1 and self.has_batch_rank is True: Returns a single sample w/ the batch rank.

fill_value (Optional[any]): The number or initializer specifier to fill the sample. Can be used to create a (non-random) sample with a certain fill value in all elements. TODO: support initializer specstrings like 'normal', 'truncated_normal', etc..

Returns: any: The sampled element(s).

class rlgraph.spaces.float_box.**FloatBox**(*low=None*, *high=None*, *shape=None*, *dtype='float32'*, ***kwargs*)

Bases: *rlgraph.spaces.box_space.BoxSpace*

sample(*size=None*, *fill_value=None*)

Uniformly randomly samples an element from this space. This is for testing purposes, e.g. to simulate a random environment.

Args:

size (Optional[int]): The number of samples or batch size to sample. If size is > 1: Returns a batch of size samples with the 0th rank being the batch rank (even if *self.has_batch_rank* is False). If size is None or (1 and self.has_batch_rank is False): Returns a single sample w/o batch rank. If size is 1 and self.has_batch_rank is True: Returns a single sample w/ the batch rank.

fill_value (Optional[any]): The number or initializer specifier to fill the sample. Can be used to create a (non-random) sample with a certain fill value in all elements. TODO: support initializer specstrings like 'normal', 'truncated_normal', etc..

Returns: any: The sampled element(s).

class rlgraph.spaces.bool_box.**BoolBox**(*shape=None*, ***kwargs*)

Bases: *rlgraph.spaces.box_space.BoxSpace*

contains(*sample*)

Checks whether this space contains the given sample. This is more for testing purposes.

Args: sample: The element to check.

Returns: bool: Whether sample is a valid member of this space.

sample(*size=None*, *fill_value=None*)

Uniformly randomly samples an element from this space. This is for testing purposes, e.g. to simulate a random environment.

Args:

size (Optional[int]): The number of samples or batch size to sample. If size is > 1: Returns a batch of size samples with the 0th rank being the batch rank (even if *self.has_batch_rank* is False). If size is None or (1 and self.has_batch_rank is False): Returns a single sample w/o batch rank. If size is 1 and self.has_batch_rank is True: Returns a single sample w/ the batch rank.

fill_value (Optional[any]): The number or initializer specifier to fill the sample. Can be used to create a (non-random) sample with a certain fill value in all elements. TODO: support initializer specstrings like 'normal', 'truncated_normal', etc..

Returns: any: The sampled element(s).

class rlgraph.spaces.text_box.**TextBox**(*shape=()*, ***kwargs*)

Bases: *rlgraph.spaces.box_space.BoxSpace*

A text box in TXT^n where the shape means the number of text chunks in each dimension.

contains(*sample*)

Checks whether this space contains the given sample. This is more for testing purposes.

Args: sample: The element to check.

Returns: bool: Whether sample is a valid member of this space.

sample(*size=None*, *fill_value=None*)
Uniformly randomly samples an element from this space. This is for testing purposes, e.g. to simulate a random environment.

Args:

size (Optional[int]): The number of samples or batch size to sample. If size is > 1: Returns a batch of size samples with the 0th rank being the batch rank (even if *self.has_batch_rank* is False). If size is None or (1 and self.has_batch_rank is False): Returns a single sample w/o batch rank. If size is 1 and self.has_batch_rank is True: Returns a single sample w/ the batch rank.

fill_value (Optional[any]): The number or initializer specifier to fill the sample. Can be used to create a (non-random) sample with a certain fill value in all elements. TODO: support initializer spec-strings like 'normal', 'truncated_normal', etc..

Returns: any: The sampled element(s).

8.2.3 Container Spaces

class rlgraph.spaces.containers.**ContainerSpace**(*add_batch_rank=False*, *add_time_rank=False*, *time_major=False*)
Bases: *rlgraph.spaces.space.Space*

A simple placeholder class for Spaces that contain other Spaces.

sample(*size=None*, *horizontal=False*)
Child classes must overwrite this one again with support for the *horizontal* parameter.

Args:

horizontal (bool): False: Within this container, sample each child-space *size* times. True: Produce *size* single containers in an np.array of len *size*.

class rlgraph.spaces.containers.**Dict**(*spec=None*, ***kwargs*)
Bases: *rlgraph.spaces.containers.ContainerSpace*, dict

A Dict space (an ordered and keyed combination of n other spaces). Supports nesting of other Dict/Tuple spaces (or any other Space types) inside itself.

contains(*sample*)
Checks whether this space contains the given sample. This is more for testing purposes.

Args: sample: The element to check.

Returns: bool: Whether sample is a valid member of this space.

dtype

flat_dim
Returns: int: The length of a flattened vector derived from this Space.

force_batch(*samples*)
Makes sure that *samples* is always returned with a batch rank no matter whether it already has one or not (in which case this method returns a batch of 1) or whether this Space has a batch rank or not.

Args: samples (any): The samples to be batched. If already batched, return as-is.

Returns: any: The batched sample.

get_shape(*with_batch_rank=False*, *with_time_rank=False*, *time_major=None*, *with_category_rank=False*)

Returns the shape of this Space as a tuple with certain additional ranks at the front (batch) or the back (e.g. categories).

Args:

with_batch_rank (Union[bool,int]): Whether to include a possible batch-rank as *None* at 0th (or 1st) position. If *with_batch_rank* is an int (e.g. -1), the possible batch-rank is returned as that number (instead of None) at the 0th (or 1st if time_major is True) position. Default: False.

with_time_rank (Union[bool,int]): Whether to include a possible time-rank as *None* at 1st (or 0th) position. If *with_time_rank* is an int, the possible time-rank is returned as that number (instead of None) at the 1st (or 0th if time_major is True) position. Default: False.

time_major (bool): Overwrites *self.time_major* if not None. Default: None (use *self.time_major*).

Returns: tuple: The shape of this Space as a tuple.

get_variable(*name*, *is_input_feed=False*, *add_batch_rank=None*, *add_time_rank=None*, *time_major=None*, ***kwargs*)

Returns a backend-specific variable/placeholder that matches the space's shape.

Args: name (str): The name for the variable.

is_input_feed (bool): Whether the returned object should be an input placeholder, instead of a full variable.

add_batch_rank (Optional[bool,int]): If True, will add a 0th (or 1st) rank (None) to the created variable. If it is an int, will add that int (-1 means None). If None, will use the Space's default value: *self.has_batch_rank*. Default: None.

add_time_rank (Optional[bool,int]): If True, will add a 1st (or 0th) rank (None) to the created variable. If it is an int, will add that int (-1 means None). If None, will use the Space's default value: *self.has_time_rank*. Default: None.

time_major (bool): Only relevant if both *add_batch_rank* and *add_time_rank* are True. Will make the time-rank the 0th rank and the batch-rank the 1st rank. Otherwise, batch-rank will be 0th and time-rank will be 1st. Default: False.

is_python (bool): Whether to create a python-based variable (list) or a backend-specific one.

local (bool): Whether the variable must not be shared across the network. Default: False.

Keyword Args: To be passed on to backend-specific methods (e.g. trainable, initializer, etc..).

Returns: any: A Tensor Variable/Placeholder.

rank

Returns: int: The rank of the Space not including batch- or time-ranks (e.g. 3 for a space with shape=(10, 7, 5)).

sample(*size=None*, *horizontal=False*)

Child classes must overwrite this one again with support for the *horizontal* parameter.

Args:

horizontal (bool): False: Within this container, sample each child-space *size* times. True: Produce *size* single containers in an np.array of len *size*.

shape

Returns: tuple: The shape of this Space as a tuple. Without batch or time ranks.

zeros(*size=None*)

Args: size (Optional): Same as *Space.sample()*.

Returns: np.ndarray: *size* zero samples where all values are zero and have the correct type.

class rlgraph.spaces.containers.**Tuple**(**components*, ***kwargs*)
Bases: *rlgraph.spaces.containers.ContainerSpace*, tuple

A Tuple space (an ordered sequence of n other spaces). Supports nesting of other container (Dict/Tuple) spaces inside itself.

contains(*sample*)
Checks whether this space contains the given sample. This is more for testing purposes.

Args: sample: The element to check.

Returns: bool: Whether sample is a valid member of this space.

dtype

flat_dim
Returns: int: The length of a flattened vector derived from this Space.

force_batch(*samples*)
Makes sure that *samples* is always returned with a batch rank no matter whether it already has one or not (in which case this method returns a batch of 1) or whether this Space has a batch rank or not.

Args: samples (any): The samples to be batched. If already batched, return as-is.

Returns: any: The batched sample.

get_shape(*with_batch_rank=False*, *with_time_rank=False*, *time_major=None*, *with_category_rank=False*)
Returns the shape of this Space as a tuple with certain additional ranks at the front (batch) or the back (e.g. categories).

Args:

with_batch_rank (Union[bool,int]): Whether to include a possible batch-rank as *None* at 0th (or 1st) position. If *with_batch_rank* is an int (e.g. -1), the possible batch-rank is returned as that number (instead of None) at the 0th (or 1st if time_major is True) position. Default: False.

with_time_rank (Union[bool,int]): Whether to include a possible time-rank as *None* at 1st (or 0th) position. If *with_time_rank* is an int, the possible time-rank is returned as that number (instead of None) at the 1st (or 0th if time_major is True) position. Default: False.

time_major (bool): Overwrites *self.time_major* if not None. Default: None (use *self.time_major*).

Returns: tuple: The shape of this Space as a tuple.

get_variable(*name*, *is_input_feed=False*, *add_batch_rank=None*, *add_time_rank=None*, *time_major=None*, ***kwargs*)
Returns a backend-specific variable/placeholder that matches the space's shape.

Args: name (str): The name for the variable.

is_input_feed (bool): Whether the returned object should be an input placeholder, instead of a full variable.

add_batch_rank (Optional[bool,int]): If True, will add a 0th (or 1st) rank (None) to the created variable. If it is an int, will add that int (-1 means None). If None, will use the Space's default value: *self.has_batch_rank*. Default: None.

add_time_rank (Optional[bool,int]): If True, will add a 1st (or 0th) rank (None) to the created variable. If it is an int, will add that int (-1 means None). If None, will use the Space's default value: *self.has_time_rank*. Default: None.

time_major (bool): Only relevant if both *add_batch_rank* and *add_time_rank* are True. Will make the time-rank the 0th rank and the batch-rank the 1st rank. Otherwise, batch-rank will be 0th and time-rank will be 1st. Default: False.

is_python (bool): Whether to create a python-based variable (list) or a backend-specific one.

local (bool): Whether the variable must not be shared across the network. Default: False.

Keyword Args: To be passed on to backend-specific methods (e.g. trainable, initializer, etc..).

Returns: any: A Tensor Variable/Placeholder.

rank
Returns: int: The rank of the Space not including batch- or time-ranks (e.g. 3 for a space with shape=(10, 7, 5)).

sample(*size=None*, *horizontal=False*)
Child classes must overwrite this one again with support for the *horizontal* parameter.

Args:

horizontal (bool): False: Within this container, sample each child-space *size* times. True: Produce *size* single containers in an np.array of len *size*.

shape
Returns: tuple: The shape of this Space as a tuple. Without batch or time ranks.

zeros(*size=None*)

Args: size (Optional): Same as *Space.sample()*.

Returns: np.ndarray: *size* zero samples where all values are zero and have the correct type.

8.2.4 Space Utilities

rlgraph.spaces.space_utils.**check_space_equivalence**(*space1*, *space2*)
Compares the two input Spaces for equivalence and returns the more generic Space of the two. The more generic Space is the one that has the properties has_batch_rank and/or has _time_rank set (instead of hard values in these ranks). E.g.: FloatBox((64,)) is equivalent with FloatBox((), +batch-rank). The latter will be returned.

NOTE: FloatBox((2,)) and FloatBox((3,)) are NOT equivalent.

Args: space1 (Space): The 1st Space to compare. space2 (Space): The 2nd Space to compare.

Returns:

Union[Space,False]: False is the two spaces are not equivalent. The more generic Space of the two if they are equivalent.

rlgraph.spaces.space_utils.**get_list_registry**(*from_space*, *capacity=None*, *initializer=0*, *flatten=True*, *add_batch_rank=False*)
Creates a list storage for a space by providing an ordered dict mapping space names to empty lists.

Args: from_space: Space to create registry from. capacity (Optional[int]): Optional capacity to initalize list. initializer (Optional(any)): Optional initializer for list if capacity is not None. flatten (bool): Whether to produce a FlattenedDataOp with auto-keys.

add_batch_rank (Optional[bool,int]): If from_space is given and is True, will add a 0th rank (None) to the created variable. If it is an int, will add that int instead of None. Default: False.

Returns: dict: Container dict mapping spaces to empty lists.

rlgraph.spaces.space_utils.**get_space_from_op**(*op*)

Tries to re-create a Space object given some DataOp. This is useful for shape inference when passing a Socket's ops through a GraphFunction and auto-inferring the resulting shape/Space.

Args: op (DataOp): The op to create a corresponding Space for.

Returns: Space: The inferred Space object.

rlgraph.spaces.space_utils.**sanity_check_space**(*space*, *allowed_types=None*, *non_allowed_types=None*, *must_have_batch_rank=None*, *must_have_time_rank=None*, *must_have_batch_or_time_rank=False*, *must_have_categories=None*, *num_categories=None*, *rank=None*)

Sanity checks a given Space for certain criteria and raises exceptions if they are not met.

Args: space (Space): The Space object to check. allowed_types (Optional[List[type]]): A list of types that this Space must be an instance of. non_allowed_types (Optional[List[type]]): A list of type that this Space must not be an instance of.

must_have_batch_rank (Optional[bool]): Whether the Space must (True) or must not (False) have the *has_batch_rank* property set to True. None, if it doesn't matter.

must_have_time_rank (Optional[bool]): Whether the Space must (True) or must not (False) have the *has_time_rank* property set to True. None, if it doesn't matter.

must_have_batch_or_time_rank (Optional[bool]): Whether the Space must (True) or must not (False) have either the *has_batch_rank* or the *has_time_rank* property set to True.

must_have_categories (Optional[bool]): For IntBoxes, whether the Space must (True) or must not (False) have global bounds with *num_categories* > 0. None, if it doesn't matter.

num_categories (Optional[int,tuple]): An int or a tuple (min,max) range within which the Space's *num_categories* rank must lie. Only valid for IntBoxes. None if it doesn't matter.

rank (Optional[int,tuple]): An int or a tuple (min,max) range within which the Space's rank must lie. None if it doesn't matter.

Raises: RLGraphError: Various RLGraphErrors, if any of the conditions is not met.

8.3 Agent Classes

8.3.1 Agent Base Class Reference

class rlgraph.agents.agent.**Agent**(*state_space*, *action_space*, *discount=0.98*, *preprocessing_spec=None*, *network_spec=None*, *internal_states_space=None*, *action_adapter_spec=None*, *exploration_spec=None*, *execution_spec=None*, *optimizer_spec=None*, *observe_spec=None*, *update_spec=None*, *summary_spec=None*, *saver_spec=None*, *auto_build=True*, *name='agent'*)

Bases: rlgraph.utils.specifiable.Specifiable

Generic agent defining RLGraph-API operations and parses and sanitizes configuration specs.

build(*build_options=None*)
Builds this agent. This method call only be called if the agent parameter "auto_build" was set to False.

Args: build_options (Optional[dict]): Optional build options, see build doc.

call_api_method(*op*, *inputs=None*, *return_ops=None*)
Utility method to call any desired api method on the graph, identified via output socket. Delegate this call to the RLGraph graph executor.

Args: op (str): Name of the api method.

inputs (Optional[dict,np.array]): Dict specifying the provided api_methods for (key=input space name, values=the values that should go into this space (e.g. numpy arrays)).

Returns: any: Result of the op call.

define_api_methods(*policy_scope*, *pre_processor_scope*, *optimizer_scope*, **params*)
Can be used to specify and then *self.define_api_method* the Agent's CoreComponent's API methods. Each agent implements this to build its algorithm logic.

Args: policy_scope (str): The global scope of the Policy within the Agent. pre_processor_scope (str): The global scope of the PreprocessorStack within the Agent. params (any): Params to be used freely by child Agent implementations.

export_graph(*filename=None*)
Any algorithm defined as a full-graph, as opposed to mixed (mixed Python and graph control flow) should be able to export its graph for deployment.

Args: filename (str): Export path. Depending on the backend, different filetypes may be required.

get_action(*states*, *internals=None*, *use_exploration=True*, *apply_preprocessing=True*, *extra_returns=None*)
Returns action(s) for the passed state(s). If *states* is a single state, returns a single action, otherwise, returns a batch of actions, where batch-size = number of states passed in.

Args: states (Union[dict,np.ndarray]): States dict/tuple or numpy array. internals (Union[dict,np.ndarray]): Internal states dict/tuple or numpy array.

use_exploration (bool): If False, no exploration or sampling may be applied when retrieving an action.

apply_preprocessing (bool): If True, apply any state preprocessors configured to the action. Set to false if all pre-processing is handled externally both for acting and updating.

extra_returns (Optional[Set[str]]): Optional set of Agent-specific strings for additional return values (besides the actions). All Agents must support "preprocessed_states".

Returns:

any: Action(s) as dict/tuple/np.ndarray (depending on *self.action_space*). Optional: The preprocessed states as a 2nd return value.

get_policy_weights()
Returns all weights relevant for the agent's policy for syncing purposes.

Returns: any: Weights and optionally weight meta data for this model.

import_observations(*observations*)
Bulk imports observations, potentially using device pre-fetching. Can be optionally implemented by agents requiring pre-training.

Args: observations (dict): Dict or list of observation data.

load_model(*path=None*)
Load model from serialized format.

Args: path (str): Path to checkpoint directory.

observe(*preprocessed_states*, *actions*, *internals*, *rewards*, *next_states*, *terminals*, *env_id=None*)
Observes an experience tuple or a batch of experience tuples. Note: If configured, first uses buffers and then internally calls _observe_graph() to actually run the computation graph. If buffering is disabled, this just routes the call to the respective *_observe_graph()* method of the child Agent.

Args: preprocessed_states (Union[dict, ndarray]): Preprocessed states dict or array. actions (Union[dict, ndarray]): Actions dict or array containing actions performed for the given state(s). internals (Union[list]): Internal state(s) returned by agent for the given states.Must be

empty list if no internals available.

rewards (float): Scalar reward(s) observed. terminals (bool): Boolean indicating terminal. next_states (Union[dict, ndarray]): Preprocessed next states dict or array. env_id (Optional[str]): Environment id to observe for. When using vectorized execution and

buffering, using environment ids is necessary to ensure correct trajectories are inserted. See *SingleThreadedWorker* for example usage.

preprocess_states(*states*)
Applies the agent's preprocessor to one or more states, e.g. to preprocess external data before inserting to memory without acting. Returns identity if no preprocessor defined.

Args: states (np.array): State(s) to preprocess.

Returns: np.array: Preprocessed states.

reset()
Must be implemented to define some reset behavior (before starting a new episode). This could include resetting the preprocessor and other Components.

reset_env_buffers(*env_id=None*)
Resets an environment buffer for buffered *observe* calls.

Args: env_id (Optional[str]): Environment id to reset. Defaults to a default environment if None provided.

set_policy_weights(*weights*)
Sets policy weights of this agent, e.g. for external syncing purporses.

Args: weights (any): Weights and optionally meta data to update depending on the backend.

Raises: ValueError if weights do not match graph weights in shapes and types.

store_model(*path=None*, *add_timestep=True*)
Store model using the backend's check-pointing mechanism.

Args: path (str): Path to model directory.

add_timestep (bool): Indiciates if current training step should be appended to exported model. If false, may override previous checkpoints.

terminate()
Terminates the Agent, so it will no longer be usable. Things that need to be cleaned up should be placed into this function, e.g. closing sessions and other open connections.

update(*batch=None*)
Performs an update on the computation graph either via externally experience or by sampling from an internal memory.

Args:

batch (Optional[dict]): Optional external data batch to use for update. If None, the agent should be configured to sample internally.

Returns: float: The loss value calculated in this update.

8.3.2 DQN Agent

class rlgraph.agents.dqn_agent.**DQNAgent**(*double_q=True*, *dueling_q=True*, *huber_loss=False*, *n_step=1*, *memory_spec=None*, *store_last_memory_batch=False*, *store_last_q_table=False*, ***kwargs*)

Bases: *rlgraph.agents.agent.Agent*

A collection of DQN algorithms published in the following papers: [1] Human-level control through deep reinforcement learning. Mnih, Kavukcuoglu, Silver et al. - 2015 [2] Deep Reinforcement Learning with Double Q-learning. v. Hasselt, Guez, Silver - 2015 [3] Dueling Network Architectures for Deep Reinforcement Learning, Wang et al. - 2016 [4] https://en.wikipedia.org/wiki/Huber_loss

define_api_methods(*policy_scope*, *pre_processor_scope*, *optimizer_scope*, **sub_components*)

Can be used to specify and then *self.define_api_method* the Agent's CoreComponent's API methods. Each agent implements this to build its algorithm logic.

Args: policy_scope (str): The global scope of the Policy within the Agent. pre_processor_scope (str): The global scope of the PreprocessorStack within the Agent. params (any): Params to be used freely by child Agent implementations.

get_action(*states*, *internals=None*, *use_exploration=True*, *apply_preprocessing=True*, *extra_returns=None*)

Args:

extra_returns (Optional[Set[str],str]): Optional string or set of strings for additional return values (besides the actions). Possible values are: - 'preprocessed_states': The preprocessed states after passing the given states through the preprocessor stack. - 'internal_states': The internal states returned by the RNNs in the NN pipeline. - 'used_exploration': Whether epsilon- or noise-based exploration was used or not.

Returns:

tuple or single value depending on *extra_returns*:

- action
- the preprocessed states

reset()

Resets our preprocessor, but only if it contains stateful PreprocessLayer Components (meaning the PreprocessorStack has at least one variable defined).

update(*batch=None*)

Performs an update on the computation graph either via externally experience or by sampling from an internal memory.

Args:

batch (Optional[dict]): Optional external data batch to use for update. If None, the agent should be configured to sample internally.

Returns: float: The loss value calculated in this update.

8.3.3 ApeX Agent

class rlgraph.agents.apex_agent.**ApexAgent**(*memory_spec=None*, ***kwargs*)
Bases: *rlgraph.agents.dqn_agent.DQNAgent*

Ape-X is a DQN variant designed for large scale distributed execution where many workers share a distributed prioritized experience replay.

Paper: https://arxiv.org/abs/1803.00933

The distinction to standard DQN is mainly that Ape-X needs to provide additional operations to enable external updates of priorities. Ape-X also enables per default dueling and double DQN.

get_td_loss(*batch*)
Utility method that just returns the td-loss from a batch without applying an update.

Args: batch (dict): Input batch.

Returns: Tuple: Total loss and loss per item.

update(*batch=None*)
Performs an update on the computation graph either via externally experience or by sampling from an internal memory.

Args:

batch (Optional[dict]): Optional external data batch to use for update. If None, the agent should be configured to sample internally.

Returns: float: The loss value calculated in this update.

8.3.4 IMPALA Agent

class rlgraph.agents.impala_agent.**IMPALAAgent**(*discount=0.99*, *fifo_queue_spec=None*, *architecture='large'*, *environment_spec=None*, *weight_pg=None*, *weight_baseline=None*, *weight_entropy=None*, *worker_sample_size=100*, *dynamic_batching=False*, ***kwargs*)
Bases: *rlgraph.agents.agent.Agent*

An Agent implementing the IMPALA algorithm described in [1]. The Agent contains both learner and actor API-methods, which will be put into the graph depending on the type ().

[1] IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures - Espeholt, Soyer, Munos et al. - 2018 (https://arxiv.org/abs/1802.01561)

default_environment_spec = {'frameskip': 4, 'level_id': 'seekavoid_arena_01', 'obser

default_internal_states_space = Tuple(("Floatbox((256,) <class 'numpy.float32'>)", "F

define_api_methods(**sub_components*)
Can be used to specify and then *self.define_api_method* the Agent's CoreComponent's API methods. Each agent implements this to build its algorithm logic.

Args: policy_scope (str): The global scope of the Policy within the Agent. pre_processor_scope (str): The global scope of the PreprocessorStack within the Agent. params (any): Params to be used freely by child Agent implementations.

define_api_methods_actor(*env_stepper*, *env_output_splitter*, *internal_states_slicer*, *merger*, *states_dict_splitter*, *fifo_queue*)

Defines the API-methods used by an IMPALA actor. Actors only step through an environment (n-steps at a time), collect the results and push them into the FIFO queue. Results include: The actions actually taken, the discounted accumulated returns for each action, the probability of each taken action according to the behavior policy.

Args:

env_stepper (EnvironmentStepper): The EnvironmentStepper Component to setp through the Env n steps in a single op call.

fifo_queue (FIFOQueue): The FIFOQueue Component used to enqueue env sample runs (n-step).

define_api_methods_learner(*fifo_output_splitter*, *fifo_queue*, *states_dict_splitter*, *transpose_states*, *transpose_terminals*, *transpose_action_probs*, *staging_area*, *preprocessor*, *policy*, *loss_function*, *optimizer*)

Defines the API-methods used by an IMPALA learner. Its job is basically: Pull a batch from the FIFOQueue, split it up into its components and pass these through the loss function and into the optimizer for a learning update.

Args: fifo_queue (FIFOQueue): The FIFOQueue Component used to enqueue env sample runs (n-step).

splitter (ContainerSplitter): The DictSplitter Component to split up a batch from the queue along its items.

policy (Policy): The Policy Component, which to update. loss_function (IMPALALossFunction): The IMPALALossFunction Component. optimizer (Optimizer): The optimizer that we use to calculate an update and apply it.

define_api_methods_single(*fifo_output_splitter*, *fifo_queue*, *queue_runner*, *transpose_actions*, *transpose_rewards*, *transpose_terminals*, *transpose_action_probs*, *preprocessor*, *staging_area*, *concat*, *policy*, *loss_function*, *optimizer*)

get_action(*states*, *internal_states=None*, *use_exploration=True*, *extra_returns=None*)

Returns action(s) for the passed state(s). If *states* is a single state, returns a single action, otherwise, returns a batch of actions, where batch-size = number of states passed in.

Args: states (Union[dict,np.ndarray]): States dict/tuple or numpy array. internals (Union[dict,np.ndarray]): Internal states dict/tuple or numpy array.

use_exploration (bool): If False, no exploration or sampling may be applied when retrieving an action.

apply_preprocessing (bool): If True, apply any state preprocessors configured to the action. Set to false if all pre-processing is handled externally both for acting and updating.

extra_returns (Optional[Set[str]]): Optional set of Agent-specific strings for additional return values (besides the actions). All Agents must support "preprocessed_states".

Returns:

any: Action(s) as dict/tuple/np.ndarray (depending on *self.action_space*). Optional: The preprocessed states as a 2nd return value.

update(*batch=None*)

Performs an update on the computation graph either via externally experience or by sampling from an internal memory.

Args:

batch (Optional[dict]): Optional external data batch to use for update. If None, the agent should be configured to sample internally.

Returns: float: The loss value calculated in this update.

8.4 Components Reference

8.4.1 Component Base Class Reference

class rlgraph.components.component.**Component**(**sub_components*, ***kwargs*)

Bases: rlgraph.utils.specifiable.Specifiable

Base class for a graph component (such as a layer, an entire function approximator, a memory, an optimizers, etc..).

A component can contain other components and/or its own graph-logic (e.g. tf ops). A component's sub-components are connected to each other via in- and out-Sockets (similar to LEGO blocks and deepmind's sonnet).

This base class implements the interface to add sub-components, create connections between different sub-components and between a sub-component and this one and between this component and an external component.

A component also has a variable registry, the ability to save the component's structure and variable-values to disk, and supports adding its graph_fns to the overall computation graph.

add_components(**components*, ***kwargs*)

Adds sub-components to this one.

Args: components (List[Component]): The list of Component objects to be added into this one.

Keyword Args:

expose_apis (Optional[Set[str]]): An optional set of strings with API-methods of the child component that should be exposed as the parent's API via a simple wrapper API-method for the parent (that calls the child's API-method).

#exposed_must_be_complete (bool): Whether the exposed API methods must be input-complete or not.

static assign_variable(*ref*, *value*)

Assigns a variable to a value.

Args: ref (any): The variable to assign to. value (any): The value to use for the assignment.

Returns: Optional[op]: None or the graph operation representing the assginment.

call_count = 0

call_times = []

check_input_completeness()

Checks whether this Component is "input-complete" and stores the result in self.input_complete. Input-completeness is reached (only once and then it stays that way) if all API-methods of this component (whose *must_be_complete* field is not set to False) have all their input Spaces defined.

Returns: bool: Whether this Component is input_complete or not.

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

check_variable_completeness()
Checks, whether this Component is input-complete AND all our sub-Components are input-complete. At that point, all variables are defined and we can run the *_variables* graph_fn.

Returns: bool: Whether this Component is "variables-complete".

copy(*name=None*, *scope=None*, *device=None*, *trainable=None*, *reuse_variable_scope=None*, *reuse_variable_scope_for_sub_components=None*)
Copies this component and returns a new component with possibly another name and another scope. The new component has its own variables (they are not shared with the variables of this component as they will be created after this copy anyway, during the build phase). and is initially not connected to any other component.

Args: name (str): The name of the new Component. If None, use the value of scope. scope (str): The scope of the new Component. If None, use the same scope as this component. device (str): The device of the new Component. If None, use the same device as this one.

trainable (Optional[bool]): Whether to make all variables in this component trainable or not. Use None for no specific preference.

reuse_variable_scope (Optional[str]): If not None, variables of the copy will be shared under this scope.

reuse_variable_scope_for_sub_components (Optional[str]): If not None, variables only of the sub-components of the copy will be shared under this scope.

Returns: Component: The copied component object.

create_summary(*name*, *values*, *type_='histogram'*)
Creates a summary op (and adds it to the graph). Skips those, whose full name does not match *self.summary_regexp*.

Args:

name (str): The name for the summary. This has to match *self.summary_regexp*. The name should not contain a "summary"-prefix or any global scope information (both will be added automatically by this method).

values (op): The op to summarize.

type_ (str): The summary type to create. Currently supported are: "histogram", "scalar" and "text".

create_variables(*input_spaces*, *action_space=None*)
Should create all variables that are needed within this component, unless a variable is only needed inside a single _graph_fn-method, in which case, it should be created there. Variables must be created via the backend-agnostic self.get_variable-method.

Note that for different scopes in which this component is being used, variables will not(!) be shared.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

get_all_sub_components(*list_=None*, *level_=0*)

Returns all sub-Components (including self) sorted by their nesting-level (... grand-children before children before parents).

Args: list_ (Optional[List[Component]])): A list of already collected components to append to. level_ (int): The slot indicating the Component level depth in *list_* at which we are currently.

Returns: List[Component]: A list with all the sub-components in *self* and *self* itself.

get_number_of_allowed_inputs(*api_method_name*)

Returns the number of allowed input args for a given API-method.

Args: api_method_name (str): The API-method to analyze.

Returns:

Tuple[int,int]: A tuple with the range (lower/upper bound) of allowed input args for the given API-method. An upper bound of None means that the API-method accepts any number of input args equal or larger than the lower bound.

get_parents()

Returns a list of parent and grand-parents of this component.

Returns: List[Component]: A list (may be empty if this component has no parents) of all parent and grand-parents.

get_sub_component_by_global_scope(*scope*)

Returns a sub-Component (or None if not found) by scope. The sub-coponent's scope should be given as global scope of the sub-component (not local scope with respect to this Component).

Args: scope (str): The global scope of the sub-Component we are looking for.

Returns: Component: The sub-Component with the given global scope if found, None if not found.

get_sub_component_by_name(*name*)

Returns a sub-Component (or None if not found) by its name (local scope). The sub-Component must be a direct sub-Component of *self*.

Args: name (str): The name (local scope) of the sub-Component we are looking for.

Returns: Component: The sub-Component with the given name if found, None if not found.

Raises: RLGraphError: If a sub-Component by that name could not be found.

get_variable(*name=''*, *shape=None*, *dtype='float'*, *initializer=None*, *trainable=True*, *from_space=None*, *add_batch_rank=False*, *add_time_rank=False*, *time_major=False*, *flatten=False*, *local=False*, *use_resource=False*)

Generates or returns a variable to use in the selected backend. The generated variable is automatically registered in this component's (and all parent components') variable-registry under its global-scoped name.

Args: name (str): The name under which the variable is registered in this component.

shape (Optional[tuple]): The shape of the variable. Default: empty tuple.

dtype (Union[str,type]): The dtype (as string) of this variable.

initializer (Optional[any]): Initializer for this variable.

trainable (bool): Whether this variable should be trainable. This will be overwritten, if the Component has its own *trainable* property set to either True or False.

from_space (Optional[Space,str]): Whether to create this variable from a Space object (shape and dtype are not needed then). The Space object can be given directly or via the name of the in-Socket holding the Space.

add_batch_rank (Optional[bool,int]): If True and *from_space* is given, will add a 0th (1st) rank (None) to the created variable. If it is an int, will add that int instead of None. Default: False.

add_time_rank (Optional[bool,int]): If True and *from_space* is given, will add a 1st (0th) rank (None) to the created variable. If it is an int, will add that int instead of None. Default: False.

time_major (bool): Only relevant if both *add_batch_rank* and *add_time_rank* are True. Will make the time-rank the 0th rank and the batch-rank the 1st rank. Otherwise, batch-rank will be 0th and time-rank will be 1st. Default: False.

flatten (bool): Whether to produce a FlattenedDataOp with auto-keys.

local (bool): Whether the variable must not be shared across the network. Default: False.

use_resource (bool): Whether to use the new tf resource-type variables. Default: False.

Returns:

DataOp: The actual variable (dependent on the backend) or - if from a ContainerSpace - a FlattenedDataOp or ContainerDataOp depending on the Space.

`get_variables`(**names*, ***kwargs*)

Utility method to get one or more component variable(s) by name(s).

Args: names (List[str]): Lookup name strings for variables. None for all.

Keyword Args:

collections (set): A set of collections to which the variables have to belong in order to be returned here. Default: tf.GraphKeys.TRAINABLE_VARIABLES

custom_scope_separator (str): The separator to use in the returned dict for scopes. Default: '/'.

global_scope (bool): Whether to use keys in the returned dict that include the global-scopes of the Variables. Default: False.

Returns: dict: A dict mapping variable names to their get_backend variables.

`get_variables_by_name`(**names*, ***kwargs*)

Retrieves this components variables by name.

Args: names (List[str]): List of names of Variable to return.

Keyword Args:

custom_scope_separator (str): The separator to use in the returned dict for scopes. Default: '/'.

global_scope (bool): Whether to use keys in the returned dict that include the global-scopes of the Variables. Default: False.

Returns: dict: Dict containing the requested names as keys and variables as values.

`propagate_scope`(*sub_component*)

Fixes all the sub-Component's (and its sub-Component's) global_scopes.

Args:

sub_component (Optional[Component]): The sub-Component object whose global_scope needs to be updated. Use None for this Component itself.

propagate_sub_component_properties (*properties*, *component=None*)
Recursively updates properties of component and its sub-components.

Args:

properties (dict): Dict with names of properties and their values to recursively update sub-components with.

component (Optional([Component])): Component to recursively update. Uses self if None.

propagate_summary (*key_*)
Propagates a single summary op of this Component to its parents' summaries registries.

Args: key_ (str): The lookup key for the summary to propagate.

propagate_variables (*keys=None*)
Propagates all variable from this Component to its parents' variable registries.

Args:

keys (Optional[List[str]]): An optional list of variable names to propagate. Should only be used in internal, recursive calls to this same method.

static read_variable (*variable*, *indices=None*)
Reads a variable.

Args: variable (DataOp): The variable whose value to read. indices (Optional[np.ndarray,tf.Tensor]): Indices (if any) to fetch from the variable.

Returns: any: Variable values.

register_api_methods_and_graph_fns ()
Detects all methods of the Component that should be registered as API-methods for this Component and complements *self.api_methods* and *self.api_method_inputs*. Goes by the @api decorator before each API-method or graph_fn that should be auto-thin-wrapped by an API-method.

register_variables (**variables*)
Adds already created Variables to our registry. This could be useful if the variables are not created by our own *self.get_variable* method, but by some backend-specific object (e.g. tf.layers). Also auto-creates summaries (regulated by *self.summary_regexp*) for the given variables.

Args: # TODO check if we warp PytorchVariable variables (Union[PyTorchVariable, SingleDataOp]): The Variable objects to register.

remove_sub_component_by_name (*name*)
Removes a sub-component from this one by its name. Thereby sets the *parent_component* property of the removed Component to None. Raises an error if the sub-component does not exist.

Args: name (str): The name of the sub-component to be removed.

Returns: Component: The removed component.

static reset_profile ()
Sets profiling values to 0.

static scatter_update_variable (*variable*, *indices*, *updates*)
Updates a variable. Optionally returns the operation depending on the backend.

Args: variable (any): Variable to update. indices (array): Indices to update. updates (any): Update values.

Returns: Optional[op]: The graph operation representing the update (or None).

sub_component_by_name(*scope_name*)
Returns a sub-component of this component by its name.

Args: scope_name (str): Name of the component. This is typically its scope.

Returns: Component: Sub-component if it exists.

Raises: ValueError: Error if no sub-component with this name exists.

when_input_complete(*input_spaces=None*, *action_space=None*, *device=None*, *summary_regexp=None*)
Wrapper that calls both *self.check_input_spaces* and *self.create_variables* in sequence and passes the dict with the input_spaces for each argument (key=arg name) and the action_space as parameter.

Args:

input_spaces (Optional[Dict[str,Space]]): A dict with Space/shape information. keys=in-argument name (str); values=the associated Space. Use None to take *self.api_method_inputs* instead.

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and con more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

device (str): The device to use for the variables generated.

summary_regexp (Optional[str]): A regexp (str) that defines, which summaries should be generated and registered.

8.4.2 Action Adapters

Action Adapter Base Class

class rlgraph.components.action_adapters.action_adapter.**ActionAdapter**(*action_space*, *add_units=0*, *units=None*, *weights_spec=None*, *biases_spec=None*, *activation=None*, *scope='action-adapter'*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A Component that cleans up a neural network's flat output and gets it ready for parameterizing a Distribution Component. Processing steps include: - Sending the raw, flattened NN output through a Dense layer whose number of units matches the flattened action space. - Reshaping (according to the action Space). - Translating the reshaped outputs (logits) into probabilities (by softmaxing) and log-probabilities (log).

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

get_action_layer_output (**args*, ***kwargs*)

get_logits (**args*, ***kwargs*)

get_logits_probabilities_log_probs (**args*, ***kwargs*)

Dueling Action Adapter

class rlgraph.components.action_adapters.dueling_action_adapter.**DuelingActionAdapter**(*units_st units_ac weights_ bi- ases_sp ac- ti- va- tion_sta weights_ bi- ases_sp ac- ti- va- tion_ad scope=' action- adapter **kwarg*

Bases: *rlgraph.components.action_adapters.action_adapter.ActionAdapter*

An ActionAdapter that adds a dueling Q calculation to the flattened output of a neural network.

API:

get_dueling_output(nn_output) (Tuple[SingleDataOp x 3]): The state-value, advantage-values (reshaped) and q-values (reshaped) after passing action_layer_output through the dueling layer.

get_action_layer_output (**args*, ***kwargs*)

get_logits_probabilities_log_probs (**args*, ***kwargs*)

Baseline Action Adapter

class rlgraph.components.action_adapters.baseline_action_adapter.**BaselineActionAdapter**(*scope actio adap **kw*

Bases: *rlgraph.components.action_adapters.action_adapter.ActionAdapter*

An ActionAdapter that adds 1 node to its action layer for an additional state-value output per batch item.

API: get_state_values_and_logits(nn_output) (Tuple[SingleDataOp x 2]): The state-value and action logits (reshaped).

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

get_logits(**args*, ***kwargs*)

get_logits_probabilities_log_probs(**args*, ***kwargs*)

get_state_values_and_logits(**args*, ***kwargs*)

8.4.3 Distributions

Distribution Base Class

class rlgraph.components.distributions.distribution.**Distribution**(*scope='distribution'*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A distribution wrapper class that can incorporate a backend-specific distribution object that gets its parameters from an external source (e.g. a NN).

API: get_distribution(parameters): The backend-specific distribution object. sample_stochastic(parameters): Returns a stochastic sample from the distribution. sample_deterministic(parameters): Returns the max-likelihood value (deterministic) from the distribution.

draw(parameters, max_likelihood): Draws a sample from the distribution (if *max_likelihood* is True, this is will be a deterministic draw, otherwise a stochastic sample).

entropy(parameters): The entropy value of the distribution. log_prob(parameters): The log probabilities for given values.

kl_divergence(parameters, other_parameters): The Kullback-Leibler Divergence between a Distribution and another one.

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

draw(*parameters*, *max_likelihood=True*)

entropy(*parameters*)

kl_divergence(**args*, ***kwargs*)

log_prob(**args*, ***kwargs*)

sample_deterministic(*parameters*)

sample_stochastic(*parameters*)

Normal Distribution

class rlgraph.components.distributions.normal.**Normal**(*scope='normal'*, ***kwargs*)
Bases: *rlgraph.components.distributions.distribution.Distribution*

A Gaussian Normal distribution object defined by a tuple: mean, variance, which is the same as "loc_and_scale".

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

Bernoulli Distribution

class rlgraph.components.distributions.bernoulli.**Bernoulli**(*scope='bernoulli'*, ***kwargs*)
Bases: *rlgraph.components.distributions.distribution.Distribution*

A Bernoulli distribution object defined by a single value p, the probability for True (rather than False).

Categorical Distribution

class rlgraph.components.distributions.categorical.**Categorical**(*scope='categorical'*, ***kwargs*)
Bases: *rlgraph.components.distributions.distribution.Distribution*

A categorical distribution object defined by a n values {p0, p1, ... } that add up to 1, the probabilities for picking one of the n categories.

Beta Distribution

class rlgraph.components.distributions.beta.**Beta**(*scope='beta'*, ***kwargs*)
Bases: *rlgraph.components.distributions.distribution.Distribution*

A Beta distribution is defined on the interval [0, 1] and parameterized by shape parameters alpha and beta (also called concentration parameters).

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

8.4.4 Explorations

Exploration Base Class

class rlgraph.components.explorations.exploration.**Exploration**(*epsilon_spec=None*, *noise_spec=None*, *scope='exploration'*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A Component that can be plugged on top of a Policy's output to produce action choices. It includes noise and/or epsilon-based exploration options as well as an out-Socket to draw actions from the Policy's distribution - either by sampling or by deterministically choosing the max-likelihood value.

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

EpsilonExploration Helper Class

class rlgraph.components.explorations.epsilon_exploration.**EpsilonExploration**(*decay_spec=None*, *scope='epsilon-exploration'*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A component to handle epsilon-exploration functionality. It takes the current time step and outputs a bool on whether to explore (uniformly random) or not (greedy or sampling). The time step is used by a epsilon-decay component to determine the current epsilon value between 1.0 and 0.0. The result of this decay is the probability, with which we output "True" (meaning: do explore), vs "False" (meaning: do not explore).

API: ins:

time_step (int): The current time step.

outs:

do_explore (bool): The decision whether to explore (do_explore=True; pick uniformly randomly) or whether to use a sample (or max-likelihood value) from a distribution (do_explore=False).

check_input_spaces(*input_spaces*, *action_space=None*)

Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

do_explore(**args*, ***kwargs*)

8.4.5 Helper Components Reference

class rlgraph.components.helpers.**MemSegmentTree**(*values*, *capacity*, *operator=<built-in function add>*)

Bases: object

In-memory Segment tree for prioritized replay.

Note: The pure TensorFlow segment tree is much slower because variable updating is expensive, and in scenarios like Ape-X, memory and update are separated processes, so there is little to be gained from inserting into the graph.

get(*index*)

Reads an item from the segment tree.

Args: index (int):

Returns: The element.

get_min_value(*start=0*, *stop=None*)

Returns min value of storage variable.

get_sum(*start=0*, *stop=None*)

Returns sum value of storage variable.

index_of_prefixsum(*prefix_sum*)

Identifies the highest index which satisfies the condition that the sum over all elements from 0 till the index is <= prefix_sum.

Args: prefix_sum .float): Upper bound on prefix we are allowed to select.

Returns: int: Index/indices satisfying prefix sum condition.

insert(*index*, *element*)

Inserts an element into the segment tree by determining its position in the tree.

Args: index (int): Insertion index. element (any): Element to insert.

reduce(*start*, *limit*, *reduce_op=<built-in function add>*)

Applies an operation to specified segment.

Args: start (int): Start index to apply reduction to. limit (end): End index to apply reduction to. reduce_op (Union(operator.add, min, max)): Reduce op to apply.

Returns: Number: Result of reduce operation

class rlgraph.components.helpers.**SegmentTree**(*storage_variable*, *capacity=1048*)
Bases: object

TensorFlow Segment tree for prioritized replay.

get(*index*)
Reads an item from the segment tree.

Args: index (int):

Returns: The element.

get_min_value()
Returns min value of storage variable.

get_sum()
Returns sum value of storage variable.

index_of_prefixsum(*prefix_sum*)
Identifies the highest index which satisfies the condition that the sum over all elements from 0 till the index is <= prefix_sum.

Args: prefix_sum .float): Upper bound on prefix we are allowed to select.

Returns: int: Index/indices satisfying prefix sum condition.

insert(*index*, *element*, *insert_op=<function add>*)
Inserts an element into the segment tree by determining its position in the tree.

Args: index (int): Insertion index. element (any): Element to insert. insert_op (Union(tf.add, tf.minimum, tf, maximum)): Insert operation on the tree.

reduce(*start*, *limit*, *reduce_op=<function add>*)
Applies an operation to specified segment.

Args: start (int): Start index to apply reduction to. limit (end): End index to apply reduction to. reduce_op (Union(tf.add, tf.minimum, tf.maximum)): Reduce op to apply.

Returns: Number: Result of reduce operation

class rlgraph.components.helpers.**SoftMax**(*scope='softmax'*, ***kwargs*)
Bases: *rlgraph.components.component.Component*

A simple softmax component that translates logits into probabilities (and log-probabilities).

API: apply(logits) -> returns probabilities (softmaxed) and log-probabilities.

class rlgraph.components.helpers.**VTraceFunction**(*rho_bar=1.0*, *rho_bar_pg=1.0*, *c_bar=1.0*, *device='/device:CPU:0'*, *scope='v-trace-function'*, ***kwargs*)
Bases: *rlgraph.components.component.Component*

A Helper Component that contains a graph_fn to calculate V-trace values from importance ratios (rhos). Based on [1] and coded analogously to: https://github.com/deepmind/scalable_agent

[1] IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures - Espeholt, Soyer, Munos et al. - 2018 (https://arxiv.org/abs/1802.01561)

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

8.4.6 Layer Classes

Layer Base Class

class rlgraph.components.layers.layer.**Layer**(***kwargs*)

Bases: *rlgraph.components.component.Component*

A Layer is a simple Component that implements the *apply* method with n inputs and m return values.

API: apply(*inputs): Applies the layer's logic to the inputs and returns one or more result values.

get_preprocessed_space(*space*)

Returns the Space obtained after pushing the space input through this layer.

Args: space (Space): The incoming Space object.

Returns: Space: The Space after preprocessing.

Preprocessing Layers

class rlgraph.components.layers.preprocessing.preprocess_layer.**PreprocessLayer**(*scope='preprocess'*, ***kwargs*)

Bases: *rlgraph.components.layers.layer.Layer*

A Layer that - additionally to *apply* - implements the *reset* API-method. *apply* is usually used for preprocessing inputs. *reset* is used to reset some state information of this preprocessor (e.g reset/reinitialize a variable).

Neural Network Layers

Activation Functions

rlgraph.components.layers.nn.activation_functions.**get_activation_function**(*activation_function=No*
**other_parameters*)

Returns an activation function (callable) to use in a NN layer.

Args:

activation_function (Optional[callable,str]): The activation function to lookup. Could be given as:

- already a callable (return just that)
- a lookup key (str)
- None: Use linear activation.

other_parameters (any): Possible extra parameter(s) used for some of the activation functions.

Returns: callable: The backend-dependent activation function.

NNLayer Base Class

class rlgraph.components.layers.nn.nn_layer.**NNLayer**(***kwargs*)

Bases: *rlgraph.components.layers.layer.Layer*

A generic NN-layer object implementing the *apply* graph_fn and offering additional activation function support. Can be used in the following ways:

- **Thin wrapper around a backend-specific layer object (normal use case):** Create the backend layer in the *create_variables* method and store it under *self.layer*. Then register the backend layer's variables with the RLgraph Component.
- **Custom layer (with custom computation):** Create necessary variables in *create_variables* (e.g. matrices), then override *_graph_fn_apply*, leaving *self.layer* as None.
- **Single Activation Function:** Leave *self.layer* as None and do not override *_graph_fn_apply*. It will then only apply the activation function.

check_input_spaces(*input_spaces*, *action_space=None*)

Do some sanity checking on the incoming Space: Must not be Container (for now) and must have a batch rank.

Concat Layer

class rlgraph.components.layers.nn.concat_layer.**ConcatLayer**(*axis=-1*, *scope='concat-layer'*, ***kwargs*)

Bases: *rlgraph.components.layers.nn.nn_layer.NNLayer*

A simple concatenation layer wrapper. The ConcatLayer is a Layer without sub-components but with n api_methods and 1 output, where input data is concatenated into one output by its GraphFunction.

check_input_spaces(*input_spaces*, *action_space=None*)

Do some sanity checking on the incoming Space: Must not be Container (for now) and must have a batch rank.

Conv2D Layer

class rlgraph.components.layers.nn.conv2d_layer.**Conv2DLayer**(*filters*, *kernel_size*, *strides*, *padding='valid'*, *data_format='channels_last'*, *kernel_spec=None*, *biases_spec=None*, ***kwargs*)

Bases: *rlgraph.components.layers.nn.nn_layer.NNLayer*

A Conv2D NN-layer.

create_variables(*input_spaces*, *action_space=None*)

Should create all variables that are needed within this component, unless a variable is only needed inside a single _graph_fn-method, in which case, it should be created there. Variables must be created via the backend-agnostic self.get_variable-method.

Note that for different scopes in which this component is being used, variables will not(!) be shared.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

Dense Layer

class rlgraph.components.layers.nn.dense_layer.**DenseLayer**(*units*, *weights_spec=None*, *biases_spec=None*, ***kwargs*)

Bases: *rlgraph.components.layers.nn.nn_layer.NNLayer*

A dense (or "fully connected") NN-layer.

create_variables(*input_spaces*, *action_space=None*)

Should create all variables that are needed within this component, unless a variable is only needed inside a single _graph_fn-method, in which case, it should be created there. Variables must be created via the backend-agnostic self.get_variable-method.

Note that for different scopes in which this component is being used, variables will not(!) be shared.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

LSTM Layer

class rlgraph.components.layers.nn.lstm_layer.**LSTMLayer**(*units*, *use_peepholes=False*, *cell_clip=None*, *static_loop=False*, *forget_bias=1.0*, *parallel_iterations=32*, *swap_memory=False*, *time_major=False*, ***kwargs*)

Bases: *rlgraph.components.layers.nn.nn_layer.NNLayer*

An LSTM layer processing an initial internal state vector and a batch of sequences to produce a final internal state and a batch of output sequences.

apply(**args*, ***kwargs*)

check_input_spaces(*input_spaces*, *action_space=None*)

Do some sanity checking on the incoming Space: Must not be Container (for now) and must have a batch rank.

create_variables(*input_spaces*, *action_space=None*)

Should create all variables that are needed within this component, unless a variable is only needed inside a single _graph_fn-method, in which case, it should be created there. Variables must be created via the backend-agnostic self.get_variable-method.

Note that for different scopes in which this component is being used, variables will not(!) be shared.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

MaxPool2D Layer

class rlgraph.components.layers.nn.maxpool2d_layer.**MaxPool2DLayer**(*pool_size*, *strides*, *padding='valid'*, *data_format='channels_last'*, ***kwargs*)

Bases: *rlgraph.components.layers.nn.nn_layer.NNLayer*

A max-pooling 2D layer.

Residual Layer

class rlgraph.components.layers.nn.residual_layer.**ResidualLayer**(*residual_unit*, *repeats=2*, *scope='residual-layer'*, ***kwargs*)

Bases: *rlgraph.components.layers.nn.nn_layer.NNLayer*

A residual layer that adds the input value to some calculation. Based on:

[1] Identity Mappings in Deep Residual Networks - He, Zhang, Ren and Sun (Microsoft) 2016 (https://arxiv.org/pdf/1603.05027.pdf)

API: apply(input_) ->

String/Text Processing Layers

class rlgraph.components.layers.strings.string_layer.**StringLayer**(***kwargs*)

Bases: *rlgraph.components.layers.layer.Layer*

A generic string processing layer object.

check_input_spaces(*input_spaces*, *action_space=None*)

Do some sanity checking on the incoming Space: Must be string type.

8.4.7 Loss Functions

Loss Function Base Class

class rlgraph.components.loss_functions.loss_function.**LossFunction**(*discount=0.98*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A loss function component offers a simple interface into some error/loss calculation function.

API: loss_per_item(*inputs) -> The loss value vector holding single loss values (one per item in a batch). loss_average(loss_per_item) -> The average value of the input *loss_per_item*.

loss(**args*, ***kwargs*)

DQN Loss Function

class rlgraph.components.loss_functions.dqn_loss_function.**DQNLossFunction**(*double_q=False*, *huber_loss=False*, *importance_weights=False*, *n_step=1*, *scope='dqn-loss-function'*, ***kwargs*)

Bases: *rlgraph.components.loss_functions.loss_function.LossFunction*

The classic 2015 DQN Loss Function: L = Expectation-over-uniform-batch(r + gamma x max_a'Qt(s',a') - Qn(s,a))^2 Where Qn is the "normal" Q-network and Qt is the "target" net (which is a little behind Qn for stability purposes).

API:

loss_per_item(q_values_s, actions, rewards, terminals, qt_values_sp, q_values_sp=None): The DQN loss per batch item.

check_input_spaces(*input_spaces*, *action_space=None*)
Do some sanity checking on the incoming Spaces:

loss(**args*, ***kwargs*)

IMPALA Loss Function

class rlgraph.components.loss_functions.impala_loss_function.**IMPALALossFunction**(*discount=0.99*, *reward_clipping=* *weight_pg=No* *weight_baselin* *weight_entropy* ***kwargs*)

Bases: *rlgraph.components.loss_functions.loss_function.LossFunction*

The IMPALA loss function based on v-trace off-policy policy gradient corrections, described in detail in [1].

The three terms of the loss function are: 1) The policy gradient term:

L[pg] = (rho_pg * advantages) * nabla log(pi(als)), where (rho_pg * advantages)=pg_advantages in code below.

2. **The value-function baseline term:** L[V] = 0.5 (vs - V(xs))^2, such that dL[V]/dtheta = (vs - V(xs)) nabla V(xs)
3. **The entropy regularizer term:** L[E] = - SUM[all actions a] pi(als) * log pi(als)

[1] IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures - Espeholt, Soyer, Munos et al. - 2018 (https://arxiv.org/abs/1802.01561)

check_input_spaces(*input_spaces*, *action_space=None*)
Should check on the nature of all in-Sockets Spaces of this Component. This method is called automatically by the Model when all these Spaces are know during the Model's build time.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

loss(*logits_actions_pi*, *action_probs_mu*, *values*, *actions*, *rewards*, *terminals*)
API-method that calculates the total loss (average over per-batch-item loss) from the original input to per-item-loss.

Args: see *self._graph_fn_loss_per_item*.

Returns: SingleDataOp: The tensor specifying the final loss (over the entire batch).

8.4.8 Memories

Memory Base Class

class rlgraph.components.memories.memory.**Memory**(*capacity=1000*, *scope='memory'*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

Abstract memory component.

API: insert_records(records) -> Triggers an insertion of records into the memory. get_records(num_records) -> Returns *num_records* records from the memory.

create_variables(*input_spaces*, *action_space=None*)

Should create all variables that are needed within this component, unless a variable is only needed inside a single _graph_fn-method, in which case, it should be created there. Variables must be created via the backend-agnostic self.get_variable-method.

Note that for different scopes in which this component is being used, variables will not(!) be shared.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

ReplayMemory

class rlgraph.components.memories.replay_memory.**ReplayMemory**(*capacity=1000*, *scope='replay-memory'*, ***kwargs*)

Bases: *rlgraph.components.memories.memory.Memory*

Implements a standard replay memory to sample randomized batches.

create_variables(*input_spaces*, *action_space=None*)

Should create all variables that are needed within this component, unless a variable is only needed inside a single _graph_fn-method, in which case, it should be created there. Variables must be created via the backend-agnostic self.get_variable-method.

Note that for different scopes in which this component is being used, variables will not(!) be shared.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

PrioritizedReplay

class rlgraph.components.memories.prioritized_replay.**PrioritizedReplay**(*capacity=1000*, *alpha=1.0*, *beta=0.0*, *scope='prioritized-replay'*, ***kwargs*)

Bases: *rlgraph.components.memories.memory.Memory*

Implements pure TensorFlow prioritized replay.

API: update_records(indices, update) -> Updates the given indices with the given priority scores.

create_variables(*input_spaces*, *action_space=None*)

Should create all variables that are needed within this component, unless a variable is only needed inside a single _graph_fn-method, in which case, it should be created there. Variables must be created via the backend-agnostic self.get_variable-method.

Note that for different scopes in which this component is being used, variables will not(!) be shared.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

FIFOQueue

class rlgraph.components.memories.fifo_queue.**FIFOQueue**(*record_space=None*, *only_insert_single_records=False*, ***kwargs*)

Bases: *rlgraph.components.memories.memory.Memory*

A wrapper for a simple in-graph FIFOQueue.

create_variables(*input_spaces*, *action_space=None*)

Should create all variables that are needed within this component, unless a variable is only needed inside a single _graph_fn-method, in which case, it should be created there. Variables must be created via the backend-agnostic self.get_variable-method.

Note that for different scopes in which this component is being used, variables will not(!) be shared.

Args:

input_spaces (Dict[str,Space]): A dict with Space/shape information. keys=in-Socket name (str); values=the associated Space

action_space (Optional[Space]): The action Space of the Agent/GraphBuilder. Can be used to construct and connect more Components (which rely on this information). This eliminates the need to pass the action Space information into many Components' constructors.

QueueRunner

class rlgraph.components.memories.queue_runner.**QueueRunner**(*queue*, *api_method_name*, *return_slot*, *env_output_splitter*, *fifo_input_merger*, *next_states_slicer*, *internal_states_slicer*, **data_producing_components*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A queue runner that contains n sub-components, of which an API-method is called. The return values are bundled into a FIFOQueue as inputs. Queue runner uses multi-threading and is started after session creation.

API: enqueue() -> Returns a noop, but creates the enqueue ops for enqueuing data into the queue and hands these

to the underlying queue-runner object.

8.4.9 Neural Networks

Stack Class

class rlgraph.components.neural_networks.stack.**Stack**(**sub_components*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A component container stack that incorporates one or more sub-components some of whose API-methods (default: only *apply*) are automatically connected with each other (in the sequence the sub-Components are given in the c'tor), resulting in an API of the Stack. All sub-components' API-methods need to match in the number of input and output values. E.g. the third sub-component's api-metehod's number of return values has to match the forth sub-component's api-method's number of input parameters.

classmethod from_spec(*spec=None*, ***kwargs*)

Uses the given spec to create an object. If *spec* is a dict, an optional "type" key can be used as a "constructor hint" to specify a certain class of the object. If *spec* is not a dict, *spec*'s value is used directly as the "constructor hint".

The rest of *spec* (if it's a dict) will be used as kwargs for the (to-be-determined) constructor. Additional keys in **kwargs will always have precedence (overwrite keys in *spec* (if a dict)). Also, if the spec-dict or **kwargs contains the special key "_args", it will be popped from the dict and used as *args list to be passed separately to the constructor.

The following constructor hints are valid: - None: Use *cls* as constructor. - An already instantiated object: Will be returned as is; no constructor call. - A string or an object that is a key in *cls*'s *__lookup_classes__* dict: The value in *__lookup_classes__*

for that key will be used as the constructor.

- A python callable: Use that as constructor.
- **A string: Either a json filename or the name of a python module+class (e.g. "rlgraph.components.Component")** to be Will be used to

Args: spec (Optional[dict]): The specification dict.

Keyword Args:

kwargs (any): Optional possibility to pass the c'tor arguments in here and use spec as the type-only info. Then we can call this like: from_spec([type]?, [**kwargs for ctor]) If *spec* is already a dict, then *kwargs* will be merged with spec (overwriting keys in *spec*) after "type" has been popped out of *spec*. If a constructor of a Specifiable needs an *args list of items, the special key *_args* can be passed inside *kwargs* with a list type value (e.g. kwargs={"_args": [arg1, arg2, arg3]}).

Returns: The object generated from the spec.

rlgraph.components.neural_networks.stack.**force_tuple**(*elements=None*, *, *to_tuple=True*)

Makes sure *elements* is returned as a list, whether *elements* is a single item, already a list, or a tuple.

Args:

elements (Optional[any]): The inputs as single item, list, or tuple to be converted into a list/tuple. If None, returns empty list/tuple.

to_tuple (bool): Whether to use tuple (instead of list).

Returns:

Union[list,tuple]: All given elements in a list/tuple depending on *to_tuple*'s value. If elements is None, returns an empty list/tuple.

PreprocessorStack

class rlgraph.components.neural_networks.preprocessor_stack.**PreprocessorStack**(**preprocessors*, ***kwargs*)

Bases: *rlgraph.components.neural_networks.stack.Stack*

A special Stack that only carries PreprocessLayer Components and bundles all their *reset* output ops into one exposed *reset* output op. Otherwise, behaves like a Stack in feeding the outputs of one sub-Component to the inputs of the next sub-Component, etc..

API: preprocess(input_): Outputs the preprocessed input after sending it through all sub-Components of this Stack. reset(): An op to trigger all PreprocessorLayers of this Stack to be reset.

get_preprocessed_space(*space*)

Returns the Space obtained after pushing the input through all layers of this Stack.

Args: space (Space): The incoming Space object.

Returns: Space: The Space after preprocessing.

reset(**args*, ***kwargs*)

DictPreprocessorStack

class rlgraph.components.neural_networks.dict_preprocessor_stack.**DictPreprocessorStack**(*prepr*, ***kw*

Bases: *rlgraph.components.neural_networks.preprocessor_stack.PreprocessorStack*

A generic PreprocessorStack that can handle Dict/Tuple Spaces and parallely preprocess different Spaces within different (and separate) single PreprocessorStack components. The output is again a dict of preprocessed inputs.

API: preprocess(input_): Outputs the preprocessed input after sending it through all sub-Components of this Stack. reset(): An op to trigger all PreprocessorStacks of this Vector to be reset.

get_preprocessed_space(*space*)
Returns the Space obtained after pushing the input through all layers of this Stack.

Args: space (Dict): The incoming Space object.

Returns: Space: The Space after preprocessing.

reset(**args*, ***kwargs*)

NeuralNetwork

class rlgraph.components.neural_networks.neural_network.**NeuralNetwork**(**layers*, ***kwargs*)

Bases: *rlgraph.components.neural_networks.stack.Stack*

A NeuralNetwork is a Stack, in which the apply method is defined either by custom-API-method OR by connecting through all sub-Components' *apply* methods. In both cases, a dict should be returned with at least the *output* key set. Possible further keys could be *last_internal_states* for RNN-based NNs and other keys.

has_rnn()

rlgraph.components.neural_networks.neural_network.**force_tuple**(*elements=None*, ***, *to_tuple=True*)

Makes sure *elements* is returned as a list, whether *elements* is a single item, already a list, or a tuple.

Args:

elements (Optional[any]): The inputs as single item, list, or tuple to be converted into a list/tuple.
If None, returns empty list/tuple.

to_tuple (bool): Whether to use tuple (instead of list).

Returns:

Union[list,tuple]: All given elements in a list/tuple depending on *to_tuple*'s value. If elements is None,
returns an empty list/tuple.

Policy

class rlgraph.components.neural_networks.policy.**Policy**(*network_spec*, *action_space=None*, *action_adapter_spec=None*, *max_likelihood=True*, *scope='policy'*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A Policy is a wrapper Component that contains a NeuralNetwork, an ActionAdapter and a Distribution Component.

get_action(**args*, ***kwargs*)

get_action_layer_output(**args*, ***kwargs*)

get_entropy(**args*, ***kwargs*)

get_logits_probabilities_log_probs(**args*, ***kwargs*)

get_max_likelihood_action(**args*, ***kwargs*)

get_nn_output(**args*, ***kwargs*)

get_stochastic_action(**args*, ***kwargs*)

ActorComponent

class rlgraph.components.neural_networks.actor_component.**ActorComponent**(*preprocessor_spec*, *policy_spec*, *exploration_spec*, *max_likelihood=None*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A Component that incorporates an entire pipeline from env state to an action choice. Includes preprocessor, policy and exploration sub-components.

API: get_preprocessed_state_and_action(state, time_step, use_exploration) ->

get_preprocessed_state_action_and_action_probs(**args*, ***kwargs*)

get_preprocessed_state_and_action(**args*, ***kwargs*)

RLcore

8.4.10 Optimizers

Optimizer Base Class

class rlgraph.components.optimizers.optimizer.**Optimizer**(*learning_rate=None*, ***kwargs*)

Bases: *rlgraph.components.component.Component*

A component that takes a tuple of variables as in-Sockets and optimizes them according to some loss function or another criterion or method.

get_optimizer_variables()

Returns this optimizer's variables. This extra utility function is necessary because some frameworks like TensorFlow create optimizer variables "late", e.g. Adam variables, so they cannot be fetched at graph build time yet.

Returns: list: List of variables.

Local Optimizer

class rlgraph.components.optimizers.local_optimizers.**AdadeltaOptimizer**(*learning_rate*, ***kwargs*)

Bases: *rlgraph.components.optimizers.local_optimizers.LocalOptimizer*

Adadelta optimizer which adapts learning rate over time:

https://arxiv.org/abs/1212.5701

class rlgraph.components.optimizers.local_optimizers.**AdagradOptimizer**(*learning_rate*, ***kwargs*)

Bases: *rlgraph.components.optimizers.local_optimizers.LocalOptimizer*

Adaptive gradient optimizer which sets small learning rates for frequently appearing features and large learning rates for rare features:

http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

class rlgraph.components.optimizers.local_optimizers.**AdamOptimizer**(*learning_rate*, ***kwargs*)

Bases: *rlgraph.components.optimizers.local_optimizers.LocalOptimizer*

Adaptive momentum optimizer: https://arxiv.org/abs/1412.6980

class rlgraph.components.optimizers.local_optimizers.**GradientDescentOptimizer**(*learning_rate*, ***kwargs*)

Bases: *rlgraph.components.optimizers.local_optimizers.LocalOptimizer*

Classic gradient descent optimizer: "Stochastic Estimation of the Maximum of a Regression Function." - Kiefer and Wolfowitz, 1952

class rlgraph.components.optimizers.local_optimizers.**LocalOptimizer**(*learning_rate*, *clip_grad_norm=None*, ***kwargs*)

Bases: *rlgraph.components.optimizers.optimizer.Optimizer*

A local optimizer performs optimization irrespective of any distributed semantics, i.e. it has no knowledge of other machines and does not implement any communications with them.

get_optimizer_variables()

Returns this optimizer's variables. This extra utility function is necessary because some frameworks like TensorFlow create optimizer variables "late", e.g. Adam variables, so they cannot be fetched at graph build time yet.

Returns: list: List of variables.

class rlgraph.components.optimizers.local_optimizers.**NadamOptimizer**(*learning_rate*, ***kwargs*)

Bases: *rlgraph.components.optimizers.local_optimizers.LocalOptimizer*

Nesterov-adaptive momentum optimizer which applies Nesterov's accelerated gradient to Adam:

http://cs229.stanford.edu/proj2015/054_report.pdf

class rlgraph.components.optimizers.local_optimizers.**RMSPropOptimizer**(*learning_rate*, ***kwargs*)

Bases: *rlgraph.components.optimizers.local_optimizers.LocalOptimizer*

RMSProp Optimizer as discussed by Hinton:

https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

class rlgraph.components.optimizers.local_optimizers.**SGDOptimizer**(*learning_rate*, ***kwargs*)

Bases: *rlgraph.components.optimizers.local_optimizers.LocalOptimizer*

Stochastic gradient descent optimizer from tf.keras including support for momentum, learning-rate-decay and Nesterov momentum.

Horovod Optimizer

class rlgraph.components.optimizers.horovod_optimizer.**HorovodOptimizer**(*local_optimizer=None*, ***kwargs*)

Bases: *rlgraph.components.optimizers.optimizer.Optimizer*

This Optimizer provides a wrapper for the horovod optimizer package:

https://github.com/uber/horovod

Horovod is meant to be used as an alternative to distributed TensorFlow as it implements communication in a different way, as explained in the Horovod paper:

arXiv:1802.05799

This Horovod Optimizer expects a local LocalOptimizer spec (tensorflow) as input.

8.4.11 RLgraph Components from Select Papers Reference

8.4.12 Queues Reference

8.5 Environment Classes

8.5.1 Environment Base Class Reference

class rlgraph.environments.environment.**Environment**(*state_space*, *action_space*, *seed=None*)

Bases: rlgraph.utils.specifiable.Specifiable

An Env class used to run experiment-based RL.

render()

Should render the Environment in its current state. May be implemented or not.

reset()

Resets the state of the environment, returning an initial observation.

Returns: tuple: The Env's state after the reset.

seed(*seed=None*)

Sets the random seed of the environment to the given value.

Args: seed (int): The seed to use (default: current epoch seconds).

Returns: int: The seed actually used.

step(***kwargs*)

Run one time step of the environment's dynamics. When the end of an episode is reached, reset() should be called to reset the environment's internal state.

Args:

kwargs (any): The action(s) to be executed by the environment. Actions have to be members of this Environment's action_space (a call to self.action_space.contains(action) must return True)

Returns:

tuple:

- The state s' after(!) executing the given actions(s).
- The reward received after taking a in s.
- Whether s' is a terminal state.

- Some Environment specific info.

terminate()
Clean up operation. May be implemented or not.

8.5.2 Random Environment

class rlgraph.environments.random_env.**RandomEnv**(*state_space*, *action_space*, *reward_space=None*, *terminal_prob=0.1*, *deterministic=False*)
Bases: *rlgraph.environments.environment.Environment*

An Env producing random states no matter what actions come in.

reset()
Resets the state of the environment, returning an initial observation.

Returns: tuple: The Env's state after the reset.

reset_for_env_stepper()

seed(*seed=None*)
Sets the random seed of the environment to the given value.

Args: seed (int): The seed to use (default: current epoch seconds).

Returns: int: The seed actually used.

step(*actions=None*)
Run one time step of the environment's dynamics. When the end of an episode is reached, reset() should be called to reset the environment's internal state.

Args:

kwargs (any): The action(s) to be executed by the environment. Actions have to be members of this Environment's action_space (a call to self.action_space.contains(action) must return True)

Returns:

tuple:

- The state s' after(!) executing the given actions(s).
- The reward received after taking a in s.
- Whether s' is a terminal state.
- Some Environment specific info.

step_for_env_stepper(*actions=None*)

8.5.3 GridWorld Environments

class rlgraph.environments.grid_world.**GridWorld**(*world='4x4'*, *save_mode=False*, *reward_function='sparse'*, *state_representation='discr'*)
Bases: *rlgraph.environments.environment.Environment*

A classic grid world where the action space is up,down,left,right and the field types are: 'S' : starting point ' ' : free space 'W' : wall (blocks) 'H' : hole (terminates episode) (to be replaced by W in save-mode) 'F' : fire (usually causing negative reward) 'G' : goal state (terminates episode) TODO: Create an option to introduce a continuous action space.

MAPS = {'16x16': ['S H ', ' HH ', ' FF W W', ' W ', 'WWW FF H ', ' W ', ' FFFF W ', '

get_discrete_pos(*x*, *y*)

Returns a single, discrete int-value. Calculated by walking down the rows of the grid first (starting in upper left corner), then along the col-axis.

Args: x (int): The x-coordinate. y (int): The y-coordinate.

Returns: int: The discrete pos value corresponding to the given x and y.

get_dist_to_goal()

get_possible_next_positions(*discrete_pos*, *action*)

Given a discrete position value and an action, returns a list of possible next states and their probabilities. Only next states with non-zero probabilities will be returned. For now: Implemented as a deterministic MDP.

Args: discrete_pos (int): The discrete position to return possible next states for. action (int): The action choice.

Returns:

List[Tuple[int,float]]: A list of tuples (s', p(s'|s,a)). Where s' is the next discrete position and p(s'|s,a) is the probability of ending up in that position when in state s and taking action a.

refresh_state()

render()

Should render the Environment in its current state. May be implemented or not.

reset(*randomize=False*)

Args:

randomize (bool): Whether to start the new episode in a random position (instead of "S"). This could be an empty space (" "), the default start ("S") or a fire field ("F").

seed(*seed=None*)

Sets the random seed of the environment to the given value.

Args: seed (int): The seed to use (default: current epoch seconds).

Returns: int: The seed actually used.

step(*actions*, *set_discrete_pos=None*)

Action map: 0: up 1: right 2: down 3: left

Args: actions (int): An integer 0-3 that describes the next action. set_discrete_pos (Optional[int]): An integer to set the current discrete position to before acting.

Returns: tuple: State Space (Space), reward (float), is_terminal (bool), info (usually None).

update_cam_pixels()

x

y

8.5.4 OpenAI Gym Environments

class rlgraph.environments.openai_gym.**OpenAIGymEnv**(*gym_env*, *frameskip=None*, *max_num_noops=0*, *noop_action=0*, *episodic_life=False*, *fire_reset=False*, *monitor=None*, *monitor_safe=False*, *monitor_video=0*, *visualize=False*, ***kwargs*)

Bases: *rlgraph.environments.environment.Environment*

OpenAI Gym adapter for RLgraph: https://gym.openai.com/.

episodic_reset()

noop_reset()
Steps through reset and warm-start.

render()
Should render the Environment in its current state. May be implemented or not.

reset()
Resets the state of the environment, returning an initial observation.

Returns: tuple: The Env's state after the reset.

reset_for_env_stepper()

seed(*seed=None*)
Sets the random seed of the environment to the given value.

Args: seed (int): The seed to use (default: current epoch seconds).

Returns: int: The seed actually used.

step(*actions*)
Run one time step of the environment's dynamics. When the end of an episode is reached, reset() should be called to reset the environment's internal state.

Args:

kwargs (any): The action(s) to be executed by the environment. Actions have to be members of this Environment's action_space (a call to self.action_space.contains(action) must return True)

Returns:

tuple:

- The state s' after(!) executing the given actions(s).
- The reward received after taking a in s.
- Whether s' is a terminal state.
- Some Environment specific info.

step_for_env_stepper(*actions*)

terminate()
Clean up operation. May be implemented or not.

static translate_space(*space*, *dtype=None*)
Translates openAI spaces into RLGraph Space classes.

Args: space (gym.spaces.Space): The openAI Space to be translated.

Returns: Space: The translated rlgraph Space.

8.5.5 DeepMind Lab Environments

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

A

B

C

D

H

I

K

L

S

T

U

V

W

X

Y

Z