actor-critic Documentation

Release 0.1

Jan Robine

Aug 08, 2018

Contents

Documentation of the *actor-critic* repository on GitHub.

CHAPTER 1

API Documentation

actorcritic	The root package.

1.1 actorcritic

The root package.

agents	Contains *agents*, which are an abstraction from environments.
baselines	Contains *baselines*, which are used to compute the *advantage*.
kfac_utils	Contains utilities that concern K-FAC.
model	Contains the base class of actor-critic models.
multi_env	Contains classes that provide the ability to run multiple environments in subprocesses.
nn	Contains utilities that concern TensorFlow and neural networks.
objectives	Contains *objectives* that are used to optimize actor-critic models.
policies	Contains *policies* that determine the behavior of an *agent*.
envs	Contains functions that are dedicated to certain environments.
examples	Contains examples of how to use this project.

1.1.1 actorcritic.agents

Contains *agents*, which are an abstraction from environments.

Functions

transpose_list(values)	Transposes a list of lists.

Classes

Agent	Takes environments and a model (containing a policy) and provides `interact()`, which manages operations such as selecting actions from the model and stepping in the environments.
MultiEnvAgent(multi_env, model, num_steps)	An agent that maintains multiple environments (via *MultiEnv*) and samples multiple steps.
SingleEnvAgent(env, model, num_steps)	An agent that maintains a single environment and samples multiple steps.

class actorcritic.agents.**Agent**

Bases: object

Takes environments and a model (containing a policy) and provides *interact()*, which manages operations such as selecting actions from the model and stepping in the environments.

See also:

This allows to create multi-step agents, like *SingleEnvAgent* and *MultiEnvAgent*.

interact(*session*)

Samples actions from the model, and steps in the environments.

Parameters **session** (tf.Session) – A session that will be used to compute the actions.

Returns

tuple – A tuple of (*observations*, *actions*, *rewards*, *terminals*, *next_observations*, *infos*).

All values are in *batch-major* format, meaning that the rows determine the batch and the columns determine the time: [*batch*, *time*]. In our case the rows correspond to the environments and the columns correspond to the steps: [*environment*, *step*]. The opposite is the *time-major* format: [*time*, *batch*] or [*step*, *environment*].

Example:

If the agent maintains *3* environments and samples for *5* steps, the result would consist of a matrix (list of list) with shape [*3*, *5*]:

```
[ [step 1, step 2, step 3, step 4, step 5],   # environment 1
  [step 1, step 2, step 3, step 4, step 5],   # environment 2
  [step 1, step 2, step 3, step 4, step 5] ]  # environment 3
```

observations, *actions*, *rewards*, *terminals*, and *infos* are collected during sampling and have the shape [*environments*, *steps*].

next_observations contains the observations that the agent received at last, but did not use for selecting actions yet. These e.g. can be used to bootstrap the remaining returns. Has the shape [*environments*, *1*].

class actorcritic.agents.**MultiEnvAgent**(*multi_env*, *model*, *num_steps*)

Bases: *actorcritic.agents.Agent*

An agent that maintains multiple environments (via *MultiEnv*) and samples multiple steps.

__init__(*multi_env*, *model*, *num_steps*)

Parameters

- **multi_env** (*MultiEnv*) – Multiple environments.
- **model** (*ActorCriticModel*) – A model to sample actions.
- **num_steps** (int) – The number of steps to take in *interact()*.

interact(*session*)

Samples actions from the model, and steps in the environments.

Parameters **session** (tf.Session) – A session that will be used to compute the actions.

Returns

tuple – A tuple of (*observations*, *actions*, *rewards*, *terminals*, *next_observations*, *infos*).

All values are in *batch-major* format, meaning that the rows determine the batch and the columns determine the time: [*batch*, *time*]. In our case the rows correspond to the environments and the columns correspond to the steps: [*environment*, *step*]. The opposite is the *time-major* format: [*time*, *batch*] or [*step*, *environment*].

Example:

If the agent maintains *3* environments and samples for *5* steps, the result would consist of a matrix (list of list) with shape [*3*, *5*]:

```
[ [step 1, step 2, step 3, step 4, step 5],   # environment 1
  [step 1, step 2, step 3, step 4, step 5],   # environment 2
  [step 1, step 2, step 3, step 4, step 5] ]  # environment 3
```

observations, *actions*, *rewards*, *terminals*, and *infos* are collected during sampling and have the shape [*environments*, *steps*].

next_observations contains the observations that the agent received at last, but did not use for selecting actions yet. These e.g. can be used to bootstrap the remaining returns. Has the shape [*environments*, *1*].

class actorcritic.agents.**SingleEnvAgent**(*env*, *model*, *num_steps*)

Bases: *actorcritic.agents.Agent*

An agent that maintains a single environment and samples multiple steps.

__init__(*env*, *model*, *num_steps*)

Parameters

- **env** (gym.Env) – An environment.
- **model** (*ActorCriticModel*) – A model to sample actions.
- **num_steps** (int) – The number of steps to take in *interact()*.

interact(*session*)

Samples actions from the model and steps in the environment.

Parameters **session** (tf.Session) – A session that will be used to compute the actions.

Returns

tuple – A tuple (*observations*, *actions*, *rewards*, *terminals*, *next_observations*, *infos*).

All values are in *batch-major* format, meaning that the rows determine the batch and the columns determine the time: [*batch*, *time*]. In our case we have *one* environment so the row corresponds to the environment and the columns correspond to the steps: [*1*, *step*]. The opposite is the *time-major* format: [*time*, *batch*] or [*step*, *1*].

observations, *actions*, *rewards*, *terminals*, and *infos* are collected during sampling and have the shape [*1*, *steps*].

next_observations contains the observation that the agent received at last, but did not use for selecting an action yet. This e.g. can be used to bootstrap the remaining return. Has the shape [*1*, *1*].

actorcritic.agents.**transpose_list**(*values*)

Transposes a list of lists. Can be used to convert from *time-major* format to *batch-major* format and vice versa.

Example

Input:

```
[[1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]]
```

Output:

```
[[1, 5, 9],
 [2, 6, 10],
 [3, 7, 11],
 [4, 8, 12]]
```

Parameters **values** (list of list) – Values to transpose.

Returns list of list – The transposed values.

1.1.2 actorcritic.baselines

Contains *baselines*, which are used to compute the *advantage*.

Classes

Baseline	A wrapper class for the baseline that is subtracted from the target values to get the *advantage*.
StateValueFunction(value)	A baseline defined by a state-value function.

class actorcritic.baselines.**Baseline**

Bases: object

A wrapper class for the baseline that is subtracted from the target values to get the *advantage*.

register_predictive_distribution(*layer_collection*, *random_seed=None*)

Registers the predictive distribution of this baseline in the specified kfac.LayerCollection (required for K-FAC).

Parameters

- **layer_collection** (kfac.LayerCollection) – A layer collection used by the KfacOptimizer.
- **random_seed** (int, optional) – A random seed for sampling from the predictive distribution.

Raises NotImplementedError – If this baseline does not support K-FAC.

value
tf.Tensor – The output values of this baseline.

class actorcritic.baselines.**StateValueFunction**(*value*)
Bases: *actorcritic.baselines.Baseline*

A baseline defined by a state-value function.

__init__(*value*)

Parameters **value** (tf.Tensor) – The output values of this state-value function.

register_predictive_distribution(*layer_collection*, *random_seed=None*)
Registers the predictive distribution (normal distribution) of this state-value function in the specified kfac.LayerCollection (required for K-FAC).

Parameters

- **layer_collection** (kfac.LayerCollection) – A layer collection used by the KfacOptimizer.
- **random_seed** (int, optional) – A random seed for sampling from the predictive distribution.

value
tf.Tensor – The output values of this state-value function.

1.1.3 actorcritic.kfac_utils

Contains utilities that concern K-FAC.

Classes

ColdStartPeriodicInvUpdateKfacOpt(...)	A modified KfacOptimizer that runs the inverse operation periodically and uses a standard SGD optimizer for a few updates in the beginning, called *cold updates* and *cold optimizer*.

class actorcritic.kfac_utils.**ColdStartPeriodicInvUpdateKfacOpt**(*num_cold_updates*, *cold_optimizer*, *invert_every*, ***kwargs*)
Bases: kfac.python.ops.optimizer.KfacOptimizer

A modified KfacOptimizer that runs the inverse operation periodically and uses a standard SGD optimizer for a few updates in the beginning, called *cold updates* and *cold optimizer*.

This can be used to slowly initialize the parameters in the beginning before using the heavy K-FAC optimizer. The covariances get updated every step (after the *cold updates*).

See also:

- `kfac.PeriodicInvCovUpdateKfacOpt`
- The idea is taken from the original ACKTR implementation.

__init__(*num_cold_updates*, *cold_optimizer*, *invert_every*, ***kwargs*)

Parameters

- **num_cold_updates** (`int`) – The number of *cold updates* in the beginning before using the actual K-FAC optimizer.
- **cold_optimizer** (`tf.train.Optimizer`) – An optimizer that is used for the *cold updates*.
- **invert_every** (`int`) – The inverse operation gets called every *invert_every* steps (after the *cold updates* have finished).

apply_gradients(*grads_and_vars*, *global_step=None*, *name=None*)

Applies gradients to variables.

Parameters

- **grads_and_vars** – List of (gradient, variable) pairs.
- ***args** – Additional arguments for super.apply_gradients.
- ****kwargs** – Additional keyword arguments for super.apply_gradients.

Returns An *Operation* that applies the specified gradients.

1.1.4 actorcritic.model

Contains the base class of actor-critic models.

Classes

ActorCriticModel(observation_space, action_space)	Represents a model (e.g.

class actorcritic.model.**ActorCriticModel**(*observation_space*, *action_space*)

Bases: `object`

Represents a model (e.g. a neural net) that provides the functionalities required for actor-critic algorithms. Provides a policy, a baseline (that is subtracted from the target values to compute the advantage) and the values used for bootstrapping from next observations (ideally the values of the baseline), and the placeholders.

__init__(*observation_space*, *action_space*)

Parameters

- **observation_space** (`gym.spaces.Space`) – A space that determines the shape of the *observations_placeholder* and the *bootstrap_observations_placeholder*.
- **action_space** (`gym.spaces.Space`) – A space that determines the shape of the *actions_placeholder*.

actions_placeholder

`tf.Tensor` – The placeholder for the sampled actions.

baseline
Baseline – The baseline used by this model.

bootstrap_observations_placeholder
tf.Tensor – The placeholder for the sampled next observations. These are used to compute the *bootstrap_values*.

bootstrap_values
tf.Tensor – The bootstrapped values that are computed based on the observations passed to the *bootstrap_observations_placeholder*.

observations_placeholder
tf.Tensor – The placeholder for the sampled observations.

policy
Policy – The policy used by this model.

register_layers(*layer_collection*)
Registers the layers of this model (neural net) in the specified kfac.LayerCollection (required for K-FAC).

Parameters **layer_collection** (kfac.LayerCollection) – A layer collection used by the KfacOptimizer.

Raises NotImplementedError – If this model does not support K-FAC.

register_predictive_distributions(*layer_collection*, *random_seed=None*)
Registers the predictive distributions of the policy and the baseline in the specified kfac.LayerCollection (required for K-FAC).

Parameters

- **layer_collection** (kfac.LayerCollection) – A layer collection used by the KfacOptimizer.
- **random_seed** (int, optional) – A random seed used for sampling from the predictive distributions.

rewards_placeholder
tf.Tensor – The placeholder for the sampled rewards (scalars).

sample_actions(*observations*, *session*)
Samples actions from the policy based on the specified observations.

Parameters

- **observations** – The observations that will be passed to the *observations_placeholder*.
- **session** (tf.Session) – A session that will be used to compute the values.

Returns list of list – A list of lists of actions. The shape equals the shape of *observations*.

select_max_actions(*observations*, *session*)
Selects actions from the policy that have the highest probability (mode) based on the specified observations.

Parameters

- **observations** – The observations that will be passed to the *observations_placeholder*.
- **session** (tf.Session) – A session that will be used to compute the values.

Returns list of list – A list of lists of actions. The shape equals the shape of *observations*.

terminals_placeholder
tf.Tensor – The placeholder for the sampled terminals (booleans).

1.1.5 actorcritic.multi_env

Contains classes that provide the ability to run multiple environments in subprocesses.

Functions

create_subprocess_envs(env_fns)	Utility function that creates environments by calling the functions in *env_fns* and wrapping the returned environments in *SubprocessEnv*s.

Classes

MultiEnv(envs)	An environment that maintains multiple *SubprocessEnv*s and executes them in parallel.
SubprocessEnv(env_fn)	Maintains a gym.Env inside a subprocess, so it can run concurrently.

class actorcritic.multi_env.**MultiEnv**(*envs*)
Bases: object

An environment that maintains multiple *SubprocessEnv*s and executes them in parallel.

The environments will be reset automatically when a terminal state is reached. That means that *reset()* actually only has to be called once in the beginning.

__init__(*envs*)

Parameters **envs** (list of *SubprocessEnv*) – The environments. The observation and action spaces must be equal across the environments.

action_space
gym.spaces.Space – The action space used by all environments.

close()
Closes all environments.

envs
list of gym.Env – The environments.

observation_space
gym.spaces.Space – The observation space used by all environments.

reset()
Resets all environments.

Returns list – A list of observations received from each environment.

step(*actions*)
Proceeds one step in all environments.

Parameters **actions** (list) – A list of actions to be executed in the environments.

Returns tuple – A tuple of (*observations*, *rewards*, *terminals*, *infos*). Each element is a list containing the values received from the environments.

class actorcritic.multi_env.**SubprocessEnv**(*env_fn*)

Bases: gym.core.Env

Maintains a gym.Env inside a subprocess, so it can run concurrently. If the subprocess ends unexpectedly, it will be recreated automatically without interrupting the execution.

To use the subprocess *start()* has to be called first. After that *initialize()* has to be called to retrieve the observation space and the action space from the underlying environment. The purpose of these methods is that multiple *SubprocessEnv*s can be created and started in parallel without blocking the execution, which creates the underlying gym.Env already. Afterwards *start()*, which blocks the execution, can be called in parallel. See *create_subprocess_envs()* which implements this idea.

__init__(*env_fn*)

Parameters **env_fn** (callable) – A function that returns a gym.Env. It will be called inside the subprocess, so watch out for referencing variables on the main process or the like. It possibly will be called multiple times, since the subprocess will be recreated when it unexpectedly ends.

action_space

gym.spaces.Space – The action space of the underlying environment. Does not block the execution. *start()* and *initialize()* must have been called.

close()

Closes the subprocess.

initialize()

Retrieves the observation space and the action space from the environment in the subprocess. This method blocks until the execution is finished. *start()* must have been called.

observation_space

gym.spaces.Space – The observation space of the underlying environment. Does not block the execution. *start()* and *initialize()* must have been called.

render(*mode='human'*)

Remotely calls gym.Env.render() in the subprocess. This methods blocks until execution is finished. *start()* and *initialize()* must have been called.

Parameters **mode** (str) – The *mode* argument passed to gym.Env.render().

Returns The value returned by gym.Env.render().

reset(***kwargs*)

Remotely calls gym.Env.reset() in the underlying environment. This method blocks until execution is finished. *start()* and *initialize()* must have been called.

Parameters **kwargs** (dict) – Keyword arguments passed to gym.Env.reset().

Returns The value returned by gym.Env.reset().

start()

Starts the subprocess. Does not block. You should call *initialize()* afterwards.

step(*action*)

Remotely calls gym.Env.step() in the underlying environment. This method blocks until execution is finished. *start()* and *initialize()* must have been called.

Parameters **action** – The *action* argument passed to gym.Env.step().

Returns `tuple` – A tuple of (*observation*, *reward*, *terminal*, *info*). The values returned by `gym.Env.step()`.

class actorcritic.multi_env.**_AutoResetWrapper**(*env*)

Bases: `gym.core.Wrapper`

reset(***kwargs*)

Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step(*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

actorcritic.multi_env.**create_subprocess_envs**(*env_fns*)

Utility function that creates environments by calling the functions in *env_fns* and wrapping the returned environments in *SubprocessEnv*s. They will be started and initialized in parallel.

Parameters **env_fns** (`list` of `callable`) – A list of functions that return a `gym.Env`. They should not be instances of *SubprocessEnv*.

Returns `list` of *SubprocessEnv* – A list of the created environments.

1.1.6 actorcritic.nn

Contains utilities that concern TensorFlow and neural networks.

Functions

conv2d(input, params, stride, padding)	Creates a 2D convolutional layer with bias (without activation).
conv2d_params(num_input_channels, ...)	Creates weights and bias variables for a 2D convolutional layer.
flatten(input)	Flattens inputs but keeps the batch size.
fully_connected(input, params)	Creates a fully connected layer with bias (without activation).
fully_connected_params(input_size, ...)	Creates weights and bias variables for a fully connected layer.
linear_decay(start_value, end_value, step, ...)	Applies linear decay from *start_value* to *end_value*.

Classes

ClipGlobalNormOptimizer(optimizer, clip_norm)	A `tf.train.Optimizer` that wraps around another optimizer and minimizes the loss by clipping gradients using the global norm (`tf.clip_by_global_norm()`).

class actorcritic.nn.**ClipGlobalNormOptimizer**(*optimizer*, *clip_norm*, *name=None*)

Bases: `tensorflow.python.training.optimizer.Optimizer`

A `tf.train.Optimizer` that wraps around another optimizer and minimizes the loss by clipping gradients using the global norm (`tf.clip_by_global_norm()`).

See also:

- https://www.tensorflow.org/versions/r1.2/api_docs/python/tf/clip_by_global_norm
- https://stackoverflow.com/questions/36498127/how-to-apply-gradient-clipping-in-tensorflow/43486487#43486487

__init__(*optimizer*, *clip_norm*, *name=None*)

Parameters

- **optimizer** (`tf.train.Optimizer`) – An optimizer whose gradients will be clipped.
- **clip_norm** (`tf.Tensor` or `float`) – Value for the global norm (passed to `tf.clip_by_global_norm()`).
- **name** (`string`, optional) – A name for this optimizer.

apply_gradients(*grads_and_vars*, *global_step=None*, *name=None*)

Apply gradients to variables.

This is the second part of *minimize()*. It returns an *Operation* that applies gradients.

Parameters

- **grads_and_vars** – List of (gradient, variable) pairs as returned by *compute_gradients()*.
- **global_step** – Optional *Variable* to increment by one after the variables have been updated.
- **name** – Optional name for the returned operation. Default to the name passed to the *Optimizer* constructor.

Returns An *Operation* that applies the specified gradients. If *global_step* was not None, that operation also increments *global_step*.

Raises

- `TypeError` – If *grads_and_vars* is malformed.
- `ValueError` – If none of the variables have gradients.

actorcritic.nn.**conv2d**(*input*, *params*, *stride*, *padding*)

Creates a 2D convolutional layer with bias (without activation).

Parameters

- **input** (`tf.Tensor`) – The input values.

- **params** (tuple of (tf.Variable, tf.Variable)) – A tuple of (*weights*, *bias*). Probably obtained by *conv2d_params()*.
- **stride** (int) – The stride of the convolution.
- **padding** (string) – The padding of the convolution. One of *'VALID'*, *'SAME'*.

Returns tf.Tensor – The output values.

actorcritic.nn.**conv2d_params**(*num_input_channels*, *num_filters*, *filter_extent*, *dtype*, *weights_initializer*, *bias_initializer*)

Creates weights and bias variables for a 2D convolutional layer. These can be used in *conv2d()*.

Parameters

- **num_input_channels** (int) – The size of the input layer.
- **num_filters** (int) – The output size. Number of filters to apply.
- **filter_extent** (int) – The spatial extent of the filters. Determines the size of the weights.
- **dtype** (tf.DType) – The data type of the variables.
- **weights_initializer** (tf.keras.initializers.Initializer) – An initializer for the weights.
- **bias_initializer** (tf.keras.initializers.Initializer) – An initializer for the bias.

Returns tuple of (tf.Variable, tf.Variable) – A tuple of (*weights*, *bias*).

actorcritic.nn.**flatten**(*input*)

Flattens inputs but keeps the batch size.

Parameters **input** (tf.Tensor) – Input values of shape [*batch_size*, *d_1*, ..., *d_n*].

Returns tf.Tensor – Flattened input values of shape [*batch_size*, *d1 * ... * d_n*].

actorcritic.nn.**fully_connected**(*input*, *params*)

Creates a fully connected layer with bias (without activation).

Parameters

- **input** (tf.Tensor) – The input values.
- **params** (tuple of (tf.Variable, tf.Variable)) – A tuple of (*weights*, *bias*). Probably obtained by *fully_connected_params()*.

Returns tf.Tensor – The output values.

actorcritic.nn.**fully_connected_params**(*input_size*, *output_size*, *dtype*, *weights_initializer*, *bias_initializer*)

Creates weights and bias variables for a fully connected layer. These can be used in *fully_connected()*.

Parameters

- **input_size** (int) – The size of the input layer.
- **output_size** (int) – The output size. Number of units.
- **dtype** (tf.DType) – The data type of the variables.
- **weights_initializer** (tf.keras.initializers.Initializer) – An initializer for the weights.
- **bias_initializer** (tf.keras.initializers.Initializer) – An initializer for the bias.

Returns tuple of (tf.Variable, tf.Variable) – A tuple of (*weights*, *bias*).

actorcritic.nn.**linear_decay**(*start_value*, *end_value*, *step*, *total_steps*, *name=None*)

Applies linear decay from *start_value* to *end_value*. The value at a specific step is computed as:

```
value = (start_value - end_value) * (1 - step / total_steps) + end_value
```

Parameters

- **start_value** (tf.Tensor or float) – The start value.
- **end_value** (tf.Tensor or float) – The end value.
- **step** (tf.Tensor) – The current step (e.g. global_step).
- **total_step** (int or tf.Tensor) – The total number of steps. Steps to reach end_value.
- **name** (string, optional) – A name for the operation.

Returns tf.Tensor – The linear decayed value.

1.1.7 actorcritic.objectives

Contains *objectives* that are used to optimize actor-critic models.

Classes

A2CObjective(model[, discount_factor, ...])	An objective that defines the loss of the policy and the baseline according to the A3C and A2C/ACKTR papers.
ActorCriticObjective	An objective takes an *ActorCriticModel* and determines how it is optimized.

class actorcritic.objectives.**A2CObjective**(*model*, *discount_factor=0.99*, *entropy_regularization_strength=0.01*, *name=None*)

Bases: *actorcritic.objectives.ActorCriticObjective*

An objective that defines the loss of the policy and the baseline according to the A3C and A2C/ACKTR papers.

The rewards are discounted and the policy loss uses entropy regularization. The baseline is optimized using a squared error loss.

The policy objective uses entropy regularization:

```
J(theta) = log(policy(state, action | theta)) * (target_values - baseline) + beta
↪* entropy(policy)
```

where *beta* determines the strength of the entropy regularization.

See also:

- https://arxiv.org/pdf/1602.01783.pdf (A3C)
- https://arxiv.org/pdf/1708.05144.pdf (A2C/ACKTR)

__init__ (*model*, *discount_factor=0.99*, *entropy_regularization_strength=0.01*, *name=None*)

Parameters

- **model** (*ActorCriticModel*) – A model that provides the policy and the baseline that will be optimized.
- **discount_factor** (float) – Used for discounting the rewards. Should be between [0, 1].
- **entropy_regularization_strength** (float or tf.Tensor) – Determining the strength of the entropy regularization. Corresponds to the *beta* parameter in A3C.
- **name** (string, optional) – A name for this objective.

baseline_loss
tf.Tensor – The current loss of the baseline of the model.

mean_entropy
tf.Tensor – The current mean entropy of the policy of the model.

policy_loss
tf.Tensor – The current loss of the policy of the model.

class actorcritic.objectives.**ActorCriticObjective**
Bases: object

An objective takes an *ActorCriticModel* and determines how it is optimized. It defines the loss of the policy and the loss of the baseline, and can create train operations based on these losses.

baseline_loss
tf.Tensor – The current loss of the baseline of the model.

optimize_separate(*policy_optimizer*, *baseline_optimizer*, *policy_kwargs=None*, *baseline_kwargs=None*)
Creates an operation that minimizes the policy loss and the baseline loss separately. This means that it minimizes the losses using two different optimizers.

Parameters

- **policy_optimizer** (tf.train.Optimizer) – An optimizer that is used for the policy loss.
- **baseline_optimizer** (tf.train.Optimizer) – An optimizer that is used for the baseline loss.
- **policy_kwargs** (dict, optional) – Keyword arguments passed to the minimize() method of the *policy_optimizer*.
- **baseline_kwargs** (dict, optional) – Keyword arguments passed to the minimize() method of the *baseline_optimizer*.

Returns tf.Operation – An operation that updates both the policy and the baseline.

optimize_shared(*optimizer*, *baseline_loss_weight=0.5*, ***kwargs*)
Creates an operation that minimizes both the policy loss and the baseline loss using the same optimizer. This is used for models that share parameters between the policy and the baseline. The shared loss is defined as:

```
shared_loss = policy_loss + baseline_loss_weight * baseline_loss
```

where *baseline_loss_weight* determines the *'learning rate'* relative to the policy loss.

Parameters

- **optimizer** (`tf.train.Optimizer`) – An optimizer that is used for both the policy loss and the baseline loss.
- **baseline_loss_weight** (`float` or `tf.Tensor`) – Determines the relative *'learning rate'*.
- **kwargs** (`dict`, optional) – Keyword arguments passed to the `minimize()` method of the optimizer.

Returns `tf.Operation` – An operation that updates both the policy and the baseline.

policy_loss
`tf.Tensor` – The current loss of the policy of the model.

1.1.8 actorcritic.policies

Contains *policies* that determine the behavior of an *agent*.

Classes

DistributionPolicy(distribution, actions[, ...])	Base class for stochastic policies that follow a concrete `tf.distributions.Distribution`.
Policy	Base class for stochastic policies.
SoftmaxPolicy(logits, actions[, ...])	A stochastic policy that follows a categorical distribution.

class actorcritic.policies.**DistributionPolicy**(*distribution*, *actions*, *random_seed=None*)

Bases: *actorcritic.policies.Policy*

Base class for stochastic policies that follow a concrete `tf.distributions.Distribution`. Implements the required methods based on this distribution.

__init__(*distribution*, *actions*, *random_seed=None*)

Parameters

- **distribution** (`tf.distributions.Distribution`) – The distribution.
- **actions** (`tf.Tensor`) – The input actions used to compute the log-probabilities. Must have the same shape as the inputs.
- **random_seed** (`int`, optional) – A random seed used for sampling.

entropy
`tf.Tensor` – Computes the entropy of this policy based on the inputs that are provided for computing the probabilities. The shape equals the shape of the inputs.

log_prob
`tf.Tensor` – Computes the log-probability of the given actions based on the inputs that are provided for computing the probabilities. The shape equals the shape of the actions and the inputs.

mode
`tf.Tensor` – Selects actions from this policy which have the highest probability (mode) based on the inputs that are provided for computing the probabilities. The shape equals the shape of the inputs.

sample
`tf.Tensor` – Samples actions from this policy based on the inputs that are provided for computing the

probabilities. The shape equals the shape of the inputs.

class actorcritic.policies.**Policy**
Bases: object

Base class for stochastic policies.

entropy
tf.Tensor – Computes the entropy of this policy based on the inputs that are provided for computing the probabilities. The shape equals the shape of the inputs.

log_prob
tf.Tensor – Computes the log-probability of the given actions based on the inputs that are provided for computing the probabilities. The shape equals the shape of the actions and the inputs.

mode
tf.Tensor – Selects actions from this policy which have the highest probability (mode) based on the inputs that are provided for computing the probabilities. The shape equals the shape of the inputs.

register_predictive_distribution(*layer_collection*, *random_seed=None*)
Registers the predictive distribution of this policy in the specified kfac.LayerCollection (required for K-FAC).

Parameters

- **layer_collection** (kfac.LayerCollection) – A layer collection used by the KfacOptimizer.
- **random_seed** (int, optional) – A random seed for sampling from the predictive distribution.

Raises NotImplementedError – If this policy does not support K-FAC.

sample
tf.Tensor – Samples actions from this policy based on the inputs that are provided for computing the probabilities. The shape equals the shape of the inputs.

class actorcritic.policies.**SoftmaxPolicy**(*logits*, *actions*, *random_seed=None*, *name=None*)
Bases: *actorcritic.policies.DistributionPolicy*

A stochastic policy that follows a categorical distribution.

__init__(*logits*, *actions*, *random_seed=None*, *name=None*)

Parameters

- **logits** (tf.Tensor) – The input logits (or 'scores') used to compute the probabilities.
- **actions** (tf.Tensor) – The input actions used to compute the log-probabilities. Must have the same shape as *logits*.
- **random_seed** (int, optional) – A random seed used for sampling.
- **name** (string, optional) – A name for this policy.

register_predictive_distribution(*layer_collection*, *random_seed=None*)
Registers the predictive distribution of this policy in the specified kfac.LayerCollection (required for K-FAC).

Parameters

- **layer_collection** (kfac.LayerCollection) – A layer collection used by the KfacOptimizer.

- **random_seed** (int, optional) – A random seed for sampling from the predictive distribution.

1.1.9 actorcritic.envs

Contains functions that are dedicated to certain environments.

atari	Contains functions that are dedicated to Atari environments.

actorcritic.envs.atari

Contains functions that are dedicated to Atari environments.

model	An implementation of an actor-critic model that is aimed at Atari games.
wrappers	Contains *wrappers* that can wrap around environments to modify their functionality.

actorcritic.envs.atari.model

An implementation of an actor-critic model that is aimed at Atari games.

Classes

AtariModel(observation_space, action_space)	An *ActorCriticModel* that follows the A3C and ACKTR paper.

class actorcritic.envs.atari.model.**AtariModel**(*observation_space*, *action_space*, *conv3_num_filters=64*, *random_seed=None*, *name=None*)

Bases: *actorcritic.model.ActorCriticModel*

An *ActorCriticModel* that follows the A3C and ACKTR paper.

The observations are sent to three convolutional layers followed by a fully connected layer, each using rectifier activation functions (ReLU). The policy and the baseline use fully connected layers built on top of the last hidden fully connected layer separately. The policy layer has one unit for each action and its outputs are used as logits for a categorical distribution (softmax). The baseline layer has only one unit which represents its value.

The weights of the layers are orthogonally initialized.

Detailed network architecture:

- Conv2D: 32 filters 8x8, stride 4
- ReLU
- Conv2D: 64 filters 4x4, stride 2
- ReLU
- Conv2D: 64 filters 3x3, stride 1 (number of filters based on argument *conv3_num_filters*)

- Flatten
- Fully connected: 512 units
- ReLU
- Fully connected (policy): units = number of actions / Fully connected (baseline): 1 unit

A2C uses 64 filters in the third convolutional layer. ACKTR uses 32.

The policy is a *SoftmaxPolicy*. The baseline is a *StateValueFunction*.

See also:

This network architecture was originally used in: https://www.nature.com/articles/nature14236

__init__(*observation_space*, *action_space*, *conv3_num_filters=64*, *random_seed=None*, *name=None*)

Parameters

- **observation_space** (gym.spaces.Space) – A space that determines the shape of the observations_placeholder and the bootstrap_observations_placeholder.
- **action_space** (gym.spaces.Space) – A space that determines the shape of the actions_placeholder.
- **conv3_num_filters** (int, optional) – Number of filters used for the third convolutional layer, defaults to 64. ACKTR uses 32.
- **random_seed** (int, optional) – A random seed used for sampling from the *~actor-critic.policies.SoftmaxPolicy*.
- **name** (string, optional) – A name for this model.

register_layers(*layer_collection*)

Registers the layers of this model (neural net) in the specified kfac.LayerCollection (required for K-FAC).

Parameters **layer_collection** (kfac.LayerCollection) – A layer collection used by the KfacOptimizer.

actorcritic.envs.atari.wrappers

Contains *wrappers* that can wrap around environments to modify their functionality.

The implementations of these wrappers are adopted from OpenAI.

Classes

AtariClipRewardWrapper(env)	A wrapper that clips the rewards between -1 and 1.
AtariEpisodicLifeWrapper(env)	A wrapper that ends episodes (returns *terminal* = True) after a life in the Atari game has been lost.
AtariFireResetWrapper(env)	A wrapper that executes the *'FIRE'* action after the environment has been reset.
AtariFrameskipWrapper(env, frameskip)	A wrapper that skips frames.
AtariInfoClearWrapper(env)	A wrapper that removes unnecessary data in the *info* returned by gym.Env.step().

Continued on next page

Table 17 – continued from previous page

AtariNoopResetWrapper(env, noop_max)	A wrapper that executes a random number of *'NOOP'* actions.
AtariPreprocessFrameWrapper(env)	A wrapper that scales the observations from 210x160 down to 84x84 and converts from RGB to grayscale by extracting the luminance.
EpisodeInfoWrapper(env)	A wrapper that stores episode information in the *info* returned by `gym.Env.step()` at the end of an episode.
FrameStackWrapper(env, num_stacked_frames)	A wrapper that stacks the last observations.
RenderWrapper(env[, fps])	A wrapper that calls `gym.Env.render()` every step.

class actorcritic.envs.atari.wrappers.**AtariClipRewardWrapper**(*env*)
Bases: gym.core.RewardWrapper

A wrapper that clips the rewards between -1 and 1.

__init__(*env*)

Parameters **env** (gym.Env) – An environment that will be wrapped.

class actorcritic.envs.atari.wrappers.**AtariEpisodicLifeWrapper**(*env*)
Bases: gym.core.Wrapper

A wrapper that ends episodes (returns *terminal* = True) after a life in the Atari game has been lost.

__init__(*env*)

Parameters **env** (gym.Env) – An environment that will be wrapped.

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

class actorcritic.envs.atari.wrappers.**AtariFireResetWrapper**(*env*)
Bases: gym.core.Wrapper

A wrapper that executes the *'FIRE'* action after the environment has been reset.

__init__(*env*)

Parameters **env** (gym.Env) – An environment that will be wrapped.

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

class actorcritic.envs.atari.wrappers.**AtariFrameskipWrapper**(*env*, *frameskip*)
Bases: gym.core.Wrapper

A wrapper that skips frames.

__init__(*env*, *frameskip*)

Parameters

- **env** (gym.Env) – An environment that will be wrapped.
- **frameskip** (int) – Every *frameskip*-th frame is used. The remaining frames are skipped.

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

class actorcritic.envs.atari.wrappers.**AtariInfoClearWrapper**(*env*)
Bases: gym.core.Wrapper

A wrapper that removes unnecessary data in the *info* returned by gym.Env.step(). This reduces the amount of inter-process data.

> **Warning:** *AtariEpisodicLifeWrapper* does not work afterwards, so it should be used *before*.

__init__(*env*)

Parameters **env** (gym.Env) – An environment that will be wrapped.

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

class actorcritic.envs.atari.wrappers.**AtariNoopResetWrapper**(*env*, *noop_max*)
Bases: gym.core.Wrapper

A wrapper that executes a random number of *'NOOP'* actions.

__init__(*env*, *noop_max*)

Parameters

- **env** (gym.Env) – An environment that will be wrapped.
- **noop_max** (int) – The maximum number of *'NOOP'* actions. The number is selected randomly between 1 and *noop_max*.

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

class actorcritic.envs.atari.wrappers.**AtariPreprocessFrameWrapper**(*env*)
Bases: gym.core.ObservationWrapper

A wrapper that scales the observations from 210x160 down to 84x84 and converts from RGB to grayscale by extracting the luminance.

__init__(*env*)

Parameters **env** (gym.Env) – An environment that will be wrapped.

class actorcritic.envs.atari.wrappers.**EpisodeInfoWrapper**(*env*)
Bases: gym.core.Wrapper

A wrapper that stores episode information in the *info* returned by gym.Env.step() at the end of an episode. More specifically, if an episode is terminal, *info* will contain the key *'episode'* which has a dict value containing the *'total_reward'*, which is the cumulative reward of the episode.

Note: If you want to get the cumulative reward of the entire episode, *AtariEpisodicLifeWrapper* should be used *after* this wrapper.

__init__(*env*)

Parameters **env** (gym.Env) – An environment that will be wrapped.

static get_episode_rewards_from_info_batch(*infos*)
Utility function that extracts the episode rewards, that are inserted by the *EpisodeInfoWrapper*, out of the *infos*.

Parameters **infos** (list of list) – A batch-major list of *infos* as returned by *interact()*.

Returns numpy.ndarray – A batch-major array with the same shape as infos. It contains the episode reward of an *info* at the corresponding position. If no episode reward was in an *info*, the result will contain numpy.nan respectively.

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

class actorcritic.envs.atari.wrappers.**FrameStackWrapper**(*env*, *num_stacked_frames*)
Bases: gym.core.Wrapper

A wrapper that stacks the last observations. The observations returned by this wrapper consist of the last frames.

__init__(*env*, *num_stacked_frames*)

Parameters

- **env** (gym.Env) – An environment that will be wrapped.
- **num_stacked_frames** (int) – The number of frames that will be stacked.

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

class actorcritic.envs.atari.wrappers.**RenderWrapper**(*env*, *fps=None*)
Bases: gym.core.Wrapper

A wrapper that calls gym.Env.render() every step.

__init__ (*env*, *fps=None*)

Parameters

- **env** (gym.Env) – An environment that will be wrapped.
- **fps** (int, float, optional) – If it is not None, the steps will be slowed down to run at the specified frames per second by waiting 1.0/*fps* seconds every step.

reset (**kwargs*)

Resets the state of the environment and returns an initial observation.

Returns: observation (object): the initial observation of the space.

step (*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the environment

Returns *observation (object)* – agent's observation of the current environment reward (float) : amount of reward returned after previous action done (boolean): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

1.1.10 actorcritic.examples

Contains examples of how to use this project.

atari	Contains examples that deal with Atari environments.

actorcritic.examples.atari

Contains examples that deal with Atari environments.

a2c_acktr	An example of how to use *A2C* and *ACKTR* to learn to play an Atari game.

actorcritic.examples.atari.a2c_acktr

An example of how to use *A2C* and *ACKTR* to learn to play an Atari game.

Functions

create_environments(env_id, num_envs)	Creates multiple Atari environments that run in subprocesses.
create_optimizer(acktr, model, learning_rate)	Creates an optimizer based on whether *ACKTR* or *A2C* is used.
load_model(saver, checkpoint_path, session)	Loads the latest model checkpoint (with the neural network parameters) from a directory.

Continued on next page

Table 20 – continued from previous page

make_atari_env(env_id, render)	Creates a `gym.Env` and wraps it with all Atari wrappers in *actorcritic.envs.atari.wrappers*.
save_model(saver, checkpoint_path, ...)	Saves a model checkpoint to a directory.
train_a2c_acktr(acktr, env_id, num_envs, ...)	Trains an Atari model using *A2C* or *ACKTR*.

actorcritic.examples.atari.a2c_acktr.**create_environments**(*env_id*, *num_envs*)
Creates multiple Atari environments that run in subprocesses.

Parameters

- **env_id** (`string`) – An id passed to `gym.make()` to create the environments.
- **num_envs** (`int`) – The number of environments (and subprocesses) that will be created.

Returns `list` of `gym.Wrapper` – The environments.

actorcritic.examples.atari.a2c_acktr.**create_optimizer**(*acktr*, *model*, *learning_rate*)
Creates an optimizer based on whether *ACKTR* or *A2C* is used. *A2C* uses the RMSProp optimizer, *ACKTR* uses the K-FAC optimizer. This function is not restricted to Atari models and can be used generally.

Parameters

- **acktr** (`bool`) – Whether to use the optimizer of *ACKTR* or *A2C*.
- **model** (*ActorCriticModel*) – A model that is needed for K-FAC to register the neural network layers and the predictive distributions.
- **learning_rate** (`float` or `tf.Tensor`) – A learning rate for the optimizer.

actorcritic.examples.atari.a2c_acktr.**load_model**(*saver*, *checkpoint_path*, *session*)
Loads the latest model checkpoint (with the neural network parameters) from a directory.

Parameters

- **saver** (`tf.train.Saver`) – A saver object to restore the model.
- **checkpoint_path** (`string`) – A directory where the checkpoint is loaded from.
- **session** (`tf.Session`) – A session which will contain the loaded variable values.

actorcritic.examples.atari.a2c_acktr.**make_atari_env**(*env_id*, *render*)
Creates a `gym.Env` and wraps it with all Atari wrappers in *actorcritic.envs.atari.wrappers*.

Parameters

- **env_id** (`string`) – An id passed to `gym.make()`.
- **render** (`bool`) – Whether this environment should be rendered.

Returns `gym.Env` – The environment.

actorcritic.examples.atari.a2c_acktr.**save_model**(*saver*, *checkpoint_path*, *model_name*, *step*, *session*)
Saves a model checkpoint to a directory.

Parameters

- **saver** (`tf.train.Saver`) – A saver object to save the model.
- **checkpoint_path** (`string`) – A directory where the model checkpoint will be saved.
- **model_name** (`string`) – A name of the model. The checkpoint file in the *checkpoint_path* directory will have this name.
- **step** (`int` or `tf.Tensor`) – A number that is appended to the checkpoint file name.

- **session** (tf.Session) – A session whose variables will be saved.

actorcritic.examples.atari.a2c_acktr.**train_a2c_acktr**(*acktr*, *env_id*, *num_envs*, *num_steps*, *checkpoint_path*, *model_name*, *summary_path=None*)

Trains an Atari model using *A2C* or *ACKTR*. Automatically saves and loads the trained model.

Parameters

- **acktr** (bool) – Whether the *ACKTR* or the *A2C* algorithm should be used. *A2C* uses the RMSProp optimizer and 64 filters in the third convolutional layer of the neural network. *ACKTR* uses the K-FAC optimizer and 32 filters.
- **env_id** (string) – An id passed to gym.make() to create the environments.
- **num_envs** (int) – The number of environments that will be used (so *num_envs* subprocesses will be created). *A2C* normally uses 16. *ACKTR* normally uses 32.
- **num_steps** (int) – The number of steps to take in each iteration. *A2C* normally uses 5. *ACKTR* normally uses 20.
- **checkpoint_path** (string) – A directory where the model's checkpoints will be loaded and saved.
- **model_name** (string) – A name of the model. The files in the *checkpoint_path* directory will have this name.
- **summary_path** (string, optional) – A directory where the TensorBoard summaries will be saved. If not specified, no summaries will be saved.

CHAPTER 2

Quickstart Guide

The basic idea of reinforcement learning is to find a behavior for an *agent* inside an *environment* that leads to a maximal *reward*. Such a behavior is called a *policy* and it decides what *action* to take based on the current *observation* (also called *state*).

For example, the environment can be an Atari game. In this case the reward is the score, the actions are the controller actions, and the current frame/image of the game is an observation.

The gym library (GitHub) by OpenAI provides several types of environments. A basic reinforcement learning setup to learn a policy for the *Breakout* environment could look like this:

```
import gym

# create the environment
env = gym.make('BreakoutNoFrameskip-v4')

# receive an initial observation (frame) to select the first action
observation = env.reset()

while True:
    # let the current policy select an action
    action = policy(observation)

    # execute the action and take one step in the environment (go to next frame)
    next_observation, reward, terminal, info = env.step(action)

    # improve the policy based on this experience
    improve_policy(observation, action, reward, terminal, next_observation)

    observation = next_observation

    if terminal:
        observation = env.reset()
```

`terminal` indicates whether the game ended, so the game has to be reset. `reward` is just a number that represents the points that were achieved in this step. `info` contains debug information (the current number of lives).

A2C and *ACKTR* actually use *multiple environments* at once by running them in multiple subprocesses. This means that we can improve the policy faster, since we simply have more observations and rewards available. For that reason there is *MultiEnv*:

```
from actorcritic.multi_env import MultiEnv

envs = create_environments()  # create multiple environments
multi_env = MultiEnv(envs)
```

Yet the crucial parts are `policy(observation)` and `improve_policy(observation, action, reward, next_observation)`. We need to know how to define a policy and especially how to improve it.

Actor-critic methods define the policy as a probability distribution, such that it computes the probability of every action based on the current observation. Then these probabilities are used to sample one of the actions. For example, if the ball approaches the bottom in *Breakout*, the probability to move the paddle towards the ball should be high.

We typically use a neural network to compute these probabilities. Then the observations (frames) are sent into the network, which produces a score for every action. These scores can be passed in the softmax function to obtain probabilities. *AtariModel* provides a neural network and a policy made for Atari environments:

```
from actorcritic.envs.atari.model import AtariModel

# observation_space and action_space define the type and shape of the observations␣
↪and actions
# e.g. the size of the frames
model = AtariModel(multi_env.observation_space, multi_env.action_space)
```

Additionally *A2C* and *ACKTR* do not take one step only and improve the policy immediately. Instead they take multiple steps and use all the experienced observations and rewards to improve the policy. A *MultiEnvAgent* simplifies this process. It takes the neural network and the policy (the *'model'*), and the environments. Then we just have to call *interact()* and it uses the policy to take multiple steps:

```
from actorcritic.agents import MultiEnvAgent

agent = MultiEnvAgent(multi_env, model, num_steps=5)

while True:
    # take 5 steps in all environments
    # session is a tf.Session used to compute the values of the neural network
    observations, actions, rewards, terminals, next_observations, infos = agent.
↪interact(session)

    # improve the policy based on this experience
    improve_policy(observations, actions, rewards, terminals, next_observations)
```

In *actor-critic* methods we do not define a loss function directly, but a *policy objective* function to optimize the neural network. It needs the observations, the actions, and the rewards that the agent experienced. Then we can learn through the policy objective, which looks at the rewards in order to decide whether the actions were good or not.

Furthermore we need a *baseline* function that enhances the policy objective. It should express how much reward we can expect if we would follow our policy proceeding from the observations we just have seen. This helps the policy to decide whether the actions it has taken actually were better or worse than expected. This *baseline* function is the *'critic'* of *actor-critic* (the policy is the *'actor'*). It distinguishes actor-critic methods from *policy gradient* methods which just have an *'actor'*.

Unfortunately we do not have such a *baseline* function. That is why we will learn the *baseline*, too, at the same time as the policy. Therefore an *ActorCriticModel* like the *AtariModel* has to provide a baseline. *A2C* and *ACKTR* use the *state-value function* which indeed tells us how much reward we can expect from a given observation.

It can be beneficial to use the same neural network as the policy for the baseline. *AtariModel* does exactly this.

In summary we need a *ActorCriticObjective*. The policy objective of *A2C* and *ACKTR* is implemented in *A2CObjective*. It *discounts* the rewards and uses *entropy regularization* (see *A2CObjective*).

```
from actorcritic.objectives import A2CObjective

objective = A2CObjective(model, discount_factor=0.99, entropy_regularization_
↪strength=0.01)
```

Next we need an optimizer for our neural network:

```
import tensorflow as tf

# A2C uses the RMSProp optimizer
optimizer = tf.train.RMSPropOptimizer(learning_rate=0.0007)

# create an 'optimize' operation that we can call
# use optimize_shared() since we share the network between the policy and the baseline
optimize_op = objective.optimize_shared(optimizer)
```

That is all. We can use all variables defined above to run the A2C algorithm:

```
while True:
    # take multiple steps in all environments
    observations, actions, rewards, terminals, next_observations, infos = agent.
↪interact(session)

    # improve the policy and the baseline
    session.run(optimize_op, feed_dict={
        model.observations_placeholder: observations,
        model.bootstrap_observations_placeholder: next_observations,
        model.actions_placeholder: actions,
        model.rewards_placeholder: rewards,
        model.terminals_placeholder: terminals
    })
```

bootstrap_observations_placeholder is needed to compute the *bootstrap_values*, which are used in the policy objective.

In order to use *ACKTR* we just have to change the optimizer to a `kfac.KfacOptimizer`.

See a2c_acktr.py for a full implementation, especially how to implement `create_environments()` and how to use the K-FAC optimizer.

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G

I

L

M

O

P

R

S

T

V