AgentNet Documentation

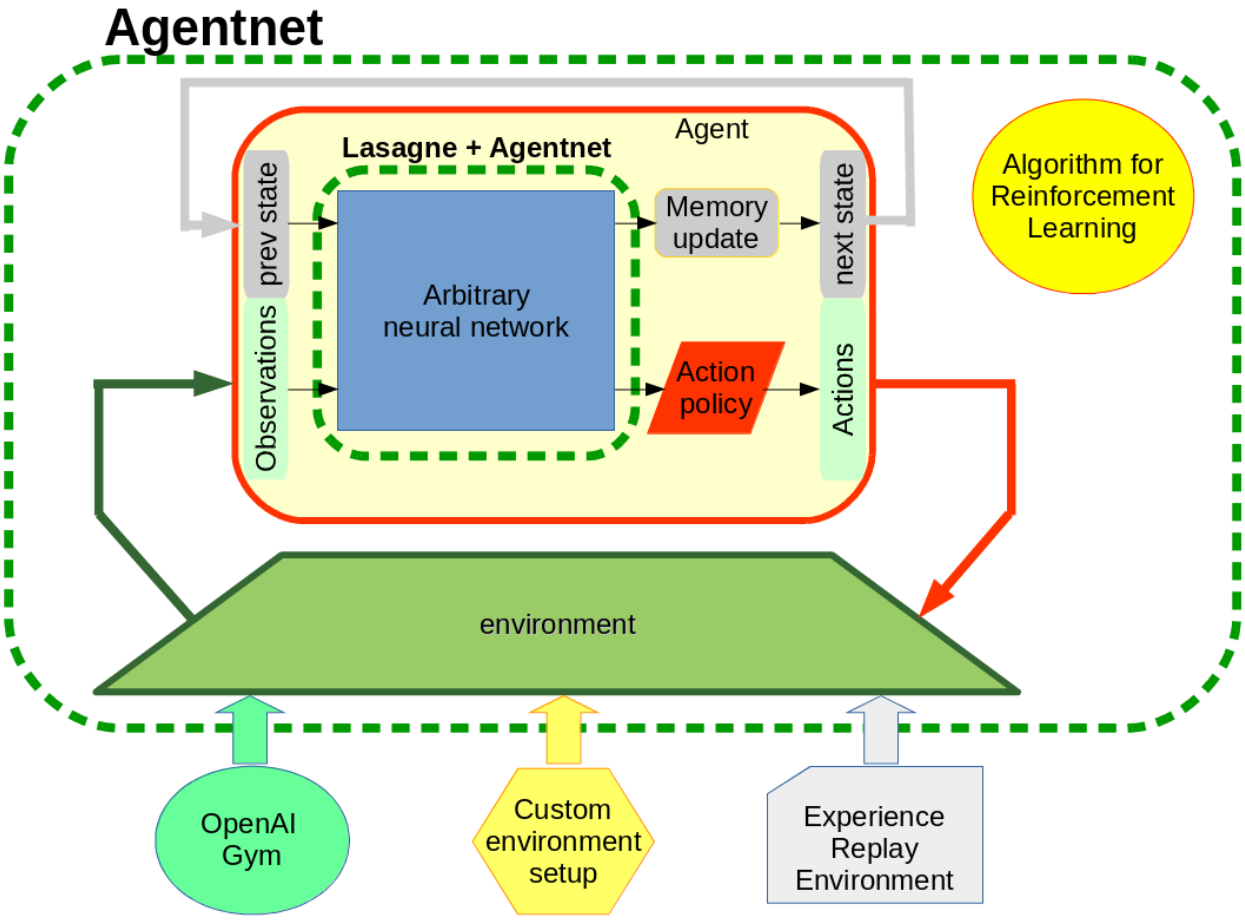
Release master

Lambda lab and contributors

Aug 17, 2017

Contents

AgentNet is a toolkit for Deep Reinforcement Learning agent design and training.



The core idea is to merge all the newest neural network layers and tools from Lasagne and Theano with Reinforcement Learning formulation and algorithms. The primary goal - make it easy and intuitive to fuse arbitrary neural network architectures into the world of reinforcement learning.

All techno-babble set aside, you can use AgentNet to __train your pet neural network to play games!__ [e.g. Atari, Doom] in a single notebook.

AgentNet has full in-and-out support for __Lasagne__ deep learning library, granting you access to all convolutions, maxouts, poolings, dropouts, etc. etc. etc.

AgentNet handles both discrete and continuous control problems and supports arbirary recurrent agent mempory structure. It also has an [experimental] support for hierarchical reinforcement learning.

The library implements numerous reinforcement learning algorithms including

- Q-learning (or deep Q-learning, since we support arbitrary complexity of network)
- N-step Q-learning
- SARSA
- N-step Advantage Actor-Critic (A2c)
- N-step Deterministic Policy Gradient (DPG)

As a side-quest, we also provide a boilerplate to custom long-term memory network architectures (see examples).

AgentNet is a library designed to create and evaluate deep reinforcement learning agents. The library is optimized for ease of prototyping and

User:

CHAPTER 1

Installing AgentNet

A native way to install AgentNet is by using pip We also provide a platform-agnostic Docker container with AgentNet and most popular analytical libraries.

Any assistance with AgentNet installation, as well as your feedback, ideas and bug reports are very welcome.

If you have a Windows-based or otherwise non-mainstream operating system and generally prefer avoiding trouble, we recommend using docker installation

Native

This far the installation was only tested on Ubuntu, Windows 7 and random Mac OS, yet an experienced user is unlikely to have problems installing it onto other Linux or Mac OS Machine

Currently the minimal dependencies are **bleeding edge** Theano and Lasagne. You can find a guide to installing them here

- http://lasagne.readthedocs.io/en/latest/user/installation.html#bleeding-edge-version

If you have both of them, you can install agentnet with

- `[sudo] pip install --upgrade https://github.com/yandexdataschool/AgentNet/archive/master.zip`

If you want to explore the examples folder, we recommend installing from repository

```
git clone https://github.com/justheuristic/AgentNet
cd AgentNet
python setup.py install
```

Developer installation

```
git clone https://github.com/justheuristic/AgentNet
cd AgentNet
python setup.py develop
```

Docker container

We use Yandex REP container to provide data analysis tools (Jupyter, Matplotlib, Pandas, etc)

To download/install/run the container, use

1. install Docker,
2. make sure `docker` daemon is running (`sudo service docker start`)
3. make sure no application is using port 1234 (this is the default port that can be changed)
4. `[sudo] docker run -d -p 1234:8888 justheuristic/agentnet`
5. Access via localhost:1234 or whatever port you chose This installation contains an installation of AgentNet, along with latest Theano and Lasagne libraries.

Notes for windows

We recommend running the docker container, using docker-machine (see docker install above).

Technically if you managed to get Theano and Lasagne working on Windows, you can follow the Linux instruction. However, we cannot guarantee that this works on all Windows distributions.

A generic guide on how to install lasagne on windows can be found in this awesome tutorial. If you already have Anaconda installed, we recommend these

- Anaconda python 2 here
- Anaconda python 3 here

CHAPTER 2

What's what

Here's a brief description of AgentNet structure, intended to help you understand where to find any particular functionality.

Using AgentNet usually involves designing agent architecture, interacting with an environment and training using reinforcement learning techniques.

The first stage (designing agent) requires you to build a neural network (or any Lasagne thing) representing **a single tick** of agent in a decision process. This process is described in `agentnet.agent` section below.

agentnet.agent

Core `Agent` abstraction to build your agent around. Lower level `Recurrence` to design custom recurrent architectures.

A tick in agent's life consists of

- seeing observation
- remembering previous memory state [if any]
- committing action
- updating memory state [if any]

Thus, the minimalistic Q-learning agent with no memory should look like this

```
import lasagne,agentnet
#where current observation arrives
observation_layer = lasagne.layers.InputLayer([None,n_observations], name=
↪"observation_input")

# q_values layer (estimated using linear model)
q_values = lasagne.layers.DenseLayer(observation_layer,num_units=n_actions,
↪nonlinearity=None)
```

```
#a layer that picks action in an epsilon-greedy manner (epsilon = 0 -> greedy manner)
action_resolver = agentnet.resolver.EpsilonGreedyResolver(q_values, epsilon=0, name=
↪"resolver")

# packing this into agent
agent = agentnet.agent.Agent(observation_layer,
                             policy_estimators=q_values,
                             action_layers=action_resolver)
```

One can that use `agent.get_sessions(...)` to produce sessions of environment interaction.

To use the trained agent, one can use `agent.get_react_function()` that compiles a one-step function that takes observations and previous memory states (if any) and returns actions and updates memory (if any).

To see these methods in action, take a look at some of the examples. The `agent.get_sessions(...)` is used everywhere and the `agent.get_react_function` is present in any of the Atari examples.

The agent supports arbitrary lasagne network architecture, so this is only the most basic usage example.

agentnet.resolver

The action-picker layers. These layers implement some action picking strategy (e.g. epsilon-greedy) fiven agent policy.

They are generally fed with action Qvalues or probabilities, predicted by the agent.

Unlike most lasagne layers, their output is usually of type int32 (for discrete action space problems), meaning the IDs of actions picked.

Here's a code snippet for epsilon-greedy resolver.

```
#a layer that estimates Qvalues. Note that there's no nonlinearity since Qvalues can␣
↪be arbitrary.
q_values = lasagne.layers.DenseLayer(<some_other_layers>,
                                     num_units=<n_actions>,
                                     nonlinearity=None)

# epsilon-greedy resolver. Returns a batch of action codes, representing actions␣
↪picked at this turn.
action_resolver = EpsilonGreedyResolver(q_values, name="action-picker")

#One can change the "epsilon" (probability of picking random action instead of␣
↪optimal one) like this
action_resolver.epsilon.set_value(0.5)
```

agentnet.memory

Memory layers used to give your agent a recurrent memory (e.g. LSTM) that can be trained via backpropagation through time.

Unlike `lasagne.layers.recurrent`, `agentnet.memory` layers provide a one-step update. For example, a `GRUCell` layer takes GRU cell from previous tick and input layer(s) and outputs an updated GRU cell state.

To add recurrent memory layers to a one-step network graph, one should

- Define where does the memory state from last tick go (typically InputLayer of your network).
- Define a layer that provides the "new state" of the recurrent memory.

- Connect these two layers when creating Agent (or Recurrence)

Here's an example of adding one RNNCell from the basic tutorial.

```
#layer where current observation goes
observation_layer = lasagne.layers.InputLayer(observation_size, name="observation_
↪input")

#layer for previous memory state (first dot)
prev_state_layer = lasagne.layers.InputLayer([None, n_hidden_neurons], name="prev_
↪state_input")

# new memory state (second dot)
rnn = agentnet.memory.RNNCell(prev_state_layer, observation_layer, name="rnn0")

#<... define Qvalues, resolver, etc>

# packing this into agent
agent = agentnet.agent.Agent(<...all inputs,actions,etc...>,
              agent_states={rnn:prev_state_layer})
```

agentnet.environment

`SessionPoolEnvironment` used to train on recorded sessions from any external environment. Also facilitates experience replay.

When using any external environment (e.g. OpenAI gym), one can go with this kind of environment alone.

If you want to implement Experience Replay-based training, take a closer look to the docs of `agentnet.environment.SessionPoolEnvironment`.

In case you want to implement a custom theano-based environment from scratch to train directly, use `agentnet.environment.BaseEnvironment` to inherit from.

agentnet.learning

A set of reinforcement learning objectives one can use to train `Agent`.

These objectives can be optimized using any optimization tool like `lasagne.updates` (see any of the examples).

agentnet.target_network

This module allows you to define the so called Target Networks - copies of your agent (or some parts of it) that use older weights from N epochs ago that slowly update towards the agent's current weights.

More details can be found in the module itself.

utils

This module stores a lot of helper functions, used in other AgentNet submodules.

It also contains a number of generally useful utility functions.

- `agentnet.utils.persistence` - contains `save` and `load` functions used to save all agent params to a file or read them from a previously saved file.
- `agentnet.utils.clone` - a function that allows you to quickly clone a subnetwork or apply some layers to a different input. Useful when sharing params.

That's it for the basics. To see this architecture in action, we recommend viewing **examples** section.

CHAPTER 3

Documentation and tutorials

One can find more-or-less structured documentation pages on AgentNet functionality here.

AgentNet also has full embedded documentation, so calling `help(some_function_or_object)` or pressing shift+tab in IPython yields a description of object/function.

A standard pipeline of AgentNet experiment is shown in following examples:

- Simple Deep Recurrent Reinforcement Learning setup .
  - Most basic demo, if a bit boring. Covers the problem of learning "If X1 than Y1 Else Y2".
  - Uses a single RNN memory and Q-learning algorithm
- Playing Atari SpaceInvaders with Convolutional NN via OpenAI Gym .
  - Step-by-step explanation of what you need to do to recreate DeepMind Atari DQN
  - Written in a generic way, so that adding recurrent memory or changing learning algorithm could be done in a couple of lines

CHAPTER 4

Demos

If you wish to get acquainted with the current library state, view some of the ./examples

- Playing Atari with Convolutional NN via OpenAI Gym
  - Can switch to any visual game thanks to awesome Gym interface
  - Very simplistic, non-recurrent suffering from atari flickering, etc.
- Deep Recurrent Kung-Fu training with GRUs and actor-critic
  - Uses the "Playing atari" example with minor changes
  - Trains via Advantage actor-critic (value+policy-based)
- Simple Deep Recurrent Reinforcement Learning setup
  - Trying to guess the interconnected hidden factors on a synthetic problem setup
- Stack-augmented GRU generator
  - Reproducing http://arxiv.org/abs/1503.01007 with less code
- MOAR deep recurrent value-based LR for wikipedia facts guessing
  - Trying to figure a policy on guessing musician attributes (genres, decades active, instruments, etc)
  - Using several hidden layers and 3-step Q-learning
- More to come

AgentNet is under active construction, so expect things to change. If you wish to join the development, we'd be happy to accept your help.

Modules:

CHAPTER 5

Agent and Recurrence

The Agent abstraction and core AgentNet functionality lies here.

Agent

MDPAgent provides a high-level interface for quick implementation of classic MDP agents with continuous, discrete or mixed action space, arbitrary recurrent agent memory and decision making policy.

If you are up to something more sophisticated, try agentnet.agent.recurrence.Recurrence, which is a lasagne layer for custom recurrent networks.

class agentnet.agent.mdp_agent.**Agent**
Alias for MDPAgent

class agentnet.agent.mdp_agent.**MDPAgent**(*observation_layers=()*, *agent_states={}*, *policy_estimators=()*, *action_layers=()*)

A generic agent within MDP (markov decision process) abstraction. Basically wraps Recurrence layer to interact between agent and environment. Note for developers: if you want to get acquainted with this code, we suggest reading [Recurrence](http://agentnet.readthedocs.io/en/master/modules/agent.html#module-agentnet.agent.recurrence) first.

Parameters

- **observation_layers** (*lasagne.layers.InputLayer or a list of such*) – agent observation(s)
- **action_layers** (*resolver.BaseResolver child instance or any appropriate layer or a tuple of such, that can be fed into environment to get next state and observation.*) – agent's action(s), or whatever is fed into your environment as agent actions.
- **agent_states** (*collections.OrderedDict or dict*) – OrderedDict{ memory_output: memory_input}, where memory_output: lasagne layer generates first agent state (before any interaction) determines new agent state given previous agent state and an

observation memory_input: lasagne.layers.InputLayer that is used as "previous state" input for memory_output

- **policy_estimators** (*lasagne.Layer child instance (e.g. Q-values) or a tuple of such instances (e.g. state value + action probabilities for a2c)*) – whatever determines agent policy (or whatever you want to work with later). - Q_values (and target network q-values) for q-learning - action probabilities for reinforce - action probabilities and state values (also possibly target network) for actor-critic - whatever intermediate state you want. e.g. if you want to penalize network for activations of layer *l_dense_1* later, you will need to add it to policy_estimators.

get_react_function(*output_flags={}*, *function_flags={'allow_input_downcast': True}*)
compiles and returns a function that performs one step of agent network

Returns a theano function. The returned function takes all observation inputs, followed by all agent memories. It's outputs are all actions, followed by all new agent memories By default, the function will have allow_input_downcast=True, you can override it in function parameters

Return type theano.function

Example

The regular use case would look something like this: (assuming agent is an MDPagent with single observation, single action and 2 memory slots) >> react = agent.get_react_function >> action, new_mem0, new_mem1 = react(observation, mem0, mem1)

get_zeros_like_memory(*batch_size=1*)
Returns a list of tensors matching initial agent memory, filled with zeros :param batch_size: how many parallel session memories to store :return: list of numpy arrays filled with zeros zeros with shape/dtype matching agent memory

get_all_params(***kwargs*)
calls lasagne.layers.get_all_params(all_agent_layers,**kwargs)

get_all_param_values(***kwargs*)
calls lasagne.layers.get_all_param_values(all_agent_layers,**kwargs)

set_all_param_values(*values*, ***kwargs*)
calls lasagne.layers.set_all_param_values(all_agent_layers,values,**kwargs)

get_sessions(*environment*, *session_length=10*, *batch_size=None*, *initial_env_states='zeros'*, *initial_observations='zeros'*, *initial_hidden='zeros'*, *experience_replay=False*, *unroll_scan=True*, *return_automatic_updates=False*, *optimize_experience_replay=None*, ***kwargs*)
Returns history of agent interaction with environment for given number of turns:

Parameters

- **environment** (BaseEnvironment) – an environment to interact with
- **session_length** (*int*) – how many turns of interaction shall there be for each batch
- **batch_size** (*int or theano.tensor.TensorVariable*) – amount of independent sessions [number or symbolic]. irrelevant if experience_replay=True (will be inferred automatically also irrelevant if there's at least one input or if you manually set any initial_*.
- **experience_replay** (*bool*) – whether or not to use experience replay if True, assumes environment to have a pre-defined sequence of observations and actions (as env.observations etc.) The agent will then observe sequence of observations and will be forced to take recorded actions via get_output(...,{action_layer=recorded_action} Saves

some time by directly using environment.observations (list of sequences) instead of calling environment.get_action_results via environment.as_layers(...). Note that if this parameter is false, agent will be allowed to pick any actions during experience replay

- **unroll_scan** – whether use theano.scan or lasagne.utils.unroll_scan
- **return_automatic_updates** – whether to append automatic updates to returned tuple (as last element)
- **kwargs** (*several kw flags (flag=value,flag2=value,..)*) – optional flags to be sent to NN when calling get_output (e.g. deterministic = True)
- **initial_something** – layers providing initial values for all variables at 0-th time step 'zeros' default means filling variables with zeros Initial values are NOT included in history sequences
- **optimize_experience_replay** – deprecated, use experience_replay

Returns state_seq,observation_seq,hidden_seq,action_seq,policy_seq, for environment state, observation, hidden state, chosen actions and agent policy respectively each of them having dimensions of [batch_i,seq_i,...] time synchronization policy: env_states[:,i] was observed as observation[:,i] BASED ON WHICH agent generated his policy[:,i], resulting in action[:,i], and also updated his memory from hidden[:,i-1] to hiden[:,i]

Return type tuple of Theano tensors

get_automatic_updates(*recurrent=True*)

Gets all random state updates that happened inside scan. :param recurrent: if True, appends automatic updates from previous layers :return: theano.OrderedUpdates with all automatic updates

as_recurrence(*environment*, *session_length=10*, *batch_size=None*, *initial_env_states='zeros'*, *initial_observations='zeros'*, *initial_hidden='zeros'*, *recurrence_name='AgentRecurrence'*, *unroll_scan=True*)

Returns a Recurrence lasagne layer that contains :

Parameters

- **environment** (BaseEnvironment) – an environment to interact with
- **session_length** (*int*) – how many turns of interaction shall there be for each batch
- **batch_size** (*int or theano.tensor.TensorVariable*) – amount of independent sessions [number or symbolic]. irrelevant if there's at least one input or if you manually set any initial_*.
- **initial_something** – layers providing initial values for all variables at 0-th time step 'zeros' default means filling variables with zeros Initial values are NOT included in history sequences flags: optional flags to be sent to NN when calling get_output (e.g. deterministic = True)
- **unroll_scan** – whether use theano.scan or lasagne.utils.unroll_scan

Returns Recurrence instance that returns [agent memory states] + [env states] + [env_observations] + [agent policy] + [action_layers outputs] all concatenated into one list

Return type *agentnet.agent.recurrence.Recurrence*

as_replay_recurrence(*environment*, *session_length=10*, *initial_hidden='zeros'*, *recurrence_name='ReplayRecurrence'*, *unroll_scan=True*)

returns a Recurrence lasagne layer that contains.

Parameters

- **environment** (SessionBatchEnvironment *or* SessionPoolEnvironment) – an environment to interact with
- **session_length** (*int*) – how many turns of interaction shall there be for each batch
- **initial_something** – layers providing initial values for all variables at 0-th time step 'zeros' default means filling variables with zeros Initial values are NOT included in history sequences flags: optional flags to be sent to NN when calling get_output (e.g. deterministic = True)
- **unroll_scan** – whether use theano.scan or lasagne.utils.unroll_scan

Returns

an agentnet.agent.recurrence.Recurrence instance that returns

[agent memory states] + [env states] + [env_observations] [agent policy] + [action_layers outputs] all concatenated into one list

get_agent_reaction(*prev_states={}*, *current_observations=()*, ***kwargs*)

Symbolic expression for a one-tick agent reaction

Parameters

- **prev_states** (*a dict [memory output: prev memory state value]*) – values for previous states
- **current_observations** (*a list of inputs where i-th input corresponds to i-th input slot from self.observations*) – agent observations at this step
- **kwargs** – any flag that should be passed to the lasagne network for lasagne.layers.get_output method

Returns a tuple of [actions, new agent states] actions: a list of all action layer outputs new_states: a list of all new_state values, where i-th element corresponds to i-th self.state_variables key

Return type the return type description

Recurrence

AgentNet core abstraction is Recurrence - a lasagne container-layer that can hold arbitrary graph and roll it for specified number of steps.

Apart from from MDP Agent, recurrence is also useful for arbitrary recurrent constructs e.g. convolutional RNN, attentive and/or augmented architectures etc. etc.

As Recurrence is a lasagne layer, one recurrence can be used as a part of computational graph of another recurrence.

class agentnet.agent.recurrence.**Recurrence**(*input_nonsequences=OrderedDict()*, *input_sequences=OrderedDict()*, *tracked_outputs=()*, *state_variables=OrderedDict()*, *state_init='zeros'*, *unroll_scan=True*, *n_steps=None*, *batch_size=None*, *mask_input=None*, *delayed_states=()*, *verify_graph=True*, *force_cast_types=False*, *name='YetAnotherRecurrence'*)

A generic recurrent layer that works with a custom graph. Recurrence is a lasagne layer that takes an inner graph and rolls it for several steps using scan. Conversely, it can be used as any other lasagne layer, even as a part of another recurrence.

[tutorial on recurrence](https://github.com/yandexdataschool/AgentNet/blob/master

/examples/Custom%20rnn%20with%20recurrence.ipynb)

param input_nonsequences inputs that are same at each time tick. Technically it's a dictionary that maps InputLayer from one-step graph to layers from the outer graph.

param input_sequences layers that represent time sequences, fed into graph tick by tick. This has to be a dict (one-step input -> sequence layer). All such sequences are iterated over FIRST AXIS (axis=1), since we consider their shape to be [batch, time, whatever_else...]

param tracked_outputs any layer from the one-state graph which outputs should be recorded at every time tick. Note that all state_variables are tracked separately, so their inclusion is not needed.

param state_variables a dictionary that maps next state variables to their respective previous state keys (new states) must be lasagne layers and values (previous states) must be InputLayers

Note that state dtype is defined thus:

- if state key layer has output_dtype, than that type is used for the entire state
- otherwise, theano.config.floatX is used

param state_init what are the default values for state_variables. In other words, what is prev_state for the first iteration. By default it's T.zeros of the appropriate shape. Can be a dict mapping state OUTPUTS to their initialisations. if so, any states not mentioned in it will be considered zeros Can be a list of initializer layers for states in the order of dict.items() if so, it's length must match len(state_variables)

param mask_input Boolean mask for sequences (like the same param in lasagne.layers.RecurrentLayer). When mask==1, computes next item as usual. Elif mask==0, next item is the copy of previous one.

param unroll_scan whether or not to use lasagne.utils.unroll_scan instead of theano.scan. Note that if unroll_scan == False, one should use .get_rng_updates after .get_output to collect automatic updates

param n_steps how many time steps will the recurrence roll for. If n_steps=None, tries to infer it. n_steps == None will only work when unroll_scan==False and there are at least some input sequences

param batch_size if the process has no inputs, this expression (int or theano scalar), this variable defines the batch size

param delayed_states any states mentioned in this list will be shifted 1 turn backwards - from init to n_steps -1. They will be padded with their initial values This is intended to allow flipping the recurrence graph to synchronize corresponding values. E.g. for MDP, if environment reaction follows agent action, synchronize observations with actions [at i-th turn agent sees i-th observation, than chooses i-th action and gets i-th reward]

param verify_graph whether to assert that all inner graph input layers are registered for the recurrence as inputs or prev states and all inputs/prev states are actually needed to compute next states/outputs. NOT the same as theano.scan(strict=True).

param force_cast_types if True, automatically converts layer types for layers to the declared type. Otherwise raises an error.

returns

a tuple of sequences with shape [batch,tick, ...]

- state variable sequences in order of dict.items()
- tracked_outputs in given order

WARNING! this layer has a dictionary of outputs. It shouldn't used further as an atomic lasagne layer. Instead, consider using my_recurrence[one_of_states_or_outputs] (see code below)

```
>>> import numpy as np
>>> import theano
>>> import agentnet
>>> from agentnet.memory import RNNCell
>>> from lasagne.layers import *
>>> sequence = InputLayer((None,None,3),name='input sequence')
>>> #one step
>>> inp = InputLayer((None,3))
>>> prev_rnn = InputLayer((None,10))
>>> rnn = RNNCell(prev_rnn,inp,name='rnn')
>>> #recurrence roll of the one-step graph above.
>>> rec = agentnet.Recurrence(input_sequences={inp:sequence},
...                           state_variables={rnn:prev_rnn},
...                           unroll_scan=False)
>>> weights = get_all_params(rec) #get weights
>>> print(weights)
>>> rnn_states = rec[rnn] #get rnn state sequence
>>> run = theano.function([sequence.input_var], get_output(rnn_states))
↪#compile applier function as any lasagne network
>>> run(np.random.randn(5,25,3)) #demo run
```

get_sequence_layers()

returns history of agent interaction with environment for given number of turns. [state_sequences] , [output sequences] - a list of all state sequences and a list of all output sequences Shape of each such sequence is [batch, tick, shape_of_one_state_or_output...]

get_one_step(*prev_states={}*, *current_inputs={}*, ***get_output_kwargs*)

Applies one-step recurrence. :param prev_states: a dict {memory output: prev state} or a list of theano expressions for each prev state

Parameters

- **current_inputs** – a dictionary of inputs that maps {input layers -> theano expressions for them}, Alternatively, it can be a list where i-th input corresponds to i-th input slot from concatenated sequences and nonsequences self.input_nonsequences.keys() + self.input_sequences.keys()
- **get_output_kwargs** – any flag that should be passed to the lasagne network for lasagne.layers.get_output method

Returns new_states: a list of all new_state values, where i-th element corresponds to i-th self.state_variables key new_outputs: a list of all outputs where i-th element corresponds to i-th self.tracked_outputs key

get_automatic_updates(*recurrent=True*)
Gets all random state updates that happened inside scan. :param recurrent: if True, appends automatic updates from previous layers :return: theano.OrderedUpdates with all automatic updates

get_params(***tags*)
returns all params, including recurrent params from one-step network

get_output_for(*inputs*, *accumulate_updates='warn'*, *recurrence_flags={}*, ***kwargs*)
returns history of agent interaction with environment for given number of turns.

Parameters

- **-[state init] +[input_nonsequences] +[input_sequences]** (*inputs*) – Each part is a list of theano expressions for layers in the order they were provided when creating this layer.
- **- a set of flags to be passed to the one step agent** (*recurrence_flags*) – e.g. {deterministic=True}

Returns [state_sequences] + [output sequences] - a list of all states and all outputs sequences Shape of each such sequence is [batch, tick, shape_of_one_state_or_output...]

CHAPTER 6

Environment

Environment is an MDP abstraction that defines which observations does agent get and how does environment external state change given agent actions and previous state.

There's a base class for environment definition, as well as special environments for Experience Replay.

When designing your own experiment,

- if it is done entirely in theano, implement BaseEnvironment. See ./experiments/wikicat or boolean_reasoning for example.
- if it isn't (which is probably the case), use SessionPoolEnvironment to train from recorded interactions as in Atari examples

BaseEnvironment

class agentnet.environment.**BaseEnvironment**(*state_shapes*, *observation_shapes*, *action_shapes*, *state_dtypes=None*, *observation_dtypes=None*, *action_dtypes=None*)

Base for environment layers. This is the class you want to inherit from when designing your own custom environments.

To define an environment, one has to describe

- it's internal state(s),
- observations it sends to agent
- actions it accepts from agent
- environment inner logic

States, observations and actions are theano tensors (matrices, vectors, etc), their shapes should be defined via state_shape, state_dtype, action_shape, etc.

The default dtypes are floatX for state and observation, int32 for action. This suits most of the cases, one can usually use inherited dtypes.

Finally, one has to implement get_action_results, which converts a tuple of (old environment state, agent action) -> (new state, new observation)

Developer tips: [when playing with non-float observations and states] if you implemented a new environment, but keep getting a _.grad illegally returned an integer-valued variable exception. (Input index _, dtype _), please make sure that any non-float environment states are excluded from gradient computation or are cast to floatX.

To find out which variable causes the problem, find all expressions of the dtype mentioned in the expression and then iteratively replace their type with a similar one (like int8 -> uint8 or int32) until the error message dtype changes. Once id does, you have found the cause of the exception.

get_action_results(*last_states*, *actions*, ***kwargs*)

Computes environment state after processing agent's action.

An example of implementation: `# a dummy update rule where new state is equal to last state new_states = prev_states # mdp with full observability observations = new_states return prev_states, observations`

Parameters

- **last_states** (*list(float[batch_id, memory_id0,[memory_id1],..])*) – Environment state on previous tick.
- **actions** (*list(int[batch_id])*) – Agent action after observing last state.

Returns a tuple of new_states, actions new_states: Environment state after processing agent's action. observations: What agent observes after committing the last action.

Return type tuple of new_states: list(float[batch_id, memory_id0,[memory_id1],..]), observations: list(float[batch_id,n_agent_inputs])

as_layers(*prev_state_layers=None*, *action_layers=None*, *environment_layer_name='EnvironmentLayer'*)

Lasagne Layer compatibility method. Understanding this is not required when implementing your own environments.

Creates a lasagne layer that makes one step environment updates given agent actions.

Parameters

- **prev_state_layers** – a layer or a list of layers that provide previous environment state. None means create InputLayers automatically
- **action_layers** – a layer or a list of layers that provide agent's chosen action. None means create InputLayers automatically.
- **environment_layer_name** (*str*) – layer's name

Returns [new_states], [observations]: 2 lists of Lasagne layers new states - all states in the same order as in self.state_shapes observations - all observations in the order of self.observation_shapes

Experience Replay

class agentnet.environment.**SessionPoolEnvironment**(*observations=1*, *actions=1*, *agent_memories=1*, *default_action_dtype='int32'*, *rng_seed=1337*)

A generic pseudo-environment that replays sessions loaded via .load_sessions(...), ignoring agent actions completely.

This environment can be used either as a tool to run experiments with non-theano environments or to actually train via experience replay [http://rll.berkeley.edu/deeprlworkshop/papers/database_composition.pdf]

It has a single scalar integer env_state, corresponding to time tick.

The environment maintains it's own pool of sessions represented as (.observations, .actions, .rewards)

To load sessions into pool, use

- .load_sessions - replace existing sessions with new ones
- .append_sessions - add new sessions to existing ones up to a limited size
- .get_session_updates - a symbolic update of experience replay pool via theano.updates.

To use SessionPoolEnvironment for experience replay, one can

- feed it into agent.get_sessions (with optimize_experience_replay=True recommended) to use all sessions
- **subsample sessions via .select_session_batch or .sample_sessions_batch to use random session subsample** [this option creates SessionBatchEnvironment that can be used with agent.get_sessions]

During experience replay sessions

- states are replaced with a fake one-unit state
- observations, actions and rewards match original ones
- **agent memory states, Q-values and all in-agent expressions (but for actions) will correspond to** what agent thinks NOW about the replay.

Although it is possible to get rewards via the regular functions, it is usually faster to take self.rewards as rewards with no additional computation.

Parameters

- **observations** (*int or lasagne.layers.Layer or list of lasagne.layers.Layer*) – number of floatX flat observations or a list of observation inputs to mimic
- **actions** (*int or lasagne.layers.Layer or list of lasagne.layers.Layer*) – number of int32 scalar actions or a list of resolvers to mimic
- **agent_memories** (*int or lasagne.layers.Layer or a list of lasagne.layers.Layer*) – number of agent states [batch,tick,unit] each or a list of memory layers to mimic
- **default_action_dtype** (*string or dtype*) – if actions are given as lasagne layers with valid dtypes, this is a default dtype of action Otherwise agentnet.utils.layers.get_layer_dtype is used on a per-layer basis

To setup custom dtype, set the .output_dtype property of layers you send as actions, observations of memories.

WARNING! this session pool is stored entirely as a set of theano shared variables. GPU-users willing to store a __large__ pool of sessions to sample from are recommended to store them somewhere outside (e.g. as numpy arrays) to avoid overloading GPU memory.

load_sessions(*observation_sequences*, *action_sequences*, *reward_seq*, *is_alive=None*, *prev_memories=None*)

Load a batch of sessions into env. The loaded sessions are that used during agent interactions.

append_sessions(*observation_sequences*, *action_sequences*, *reward_seq*, *is_alive=None*, *prev_memories=None*, *max_pool_size=None*)

Add a batch of sessions to the existing sessions. The loaded sessions are that used during agent interactions.

If max_pool_size != None, only last max_pool_size sessions are kept.

get_session_updates(*observation_sequences*, *action_sequences*, *reward_seq*, *is_alive=None*, *prev_memory=None*, *cast_dtypes=True*)

Return a dictionary of updates that will set shared variables to argument state. If cast_dtypes is True, casts all updates to the dtypes of their respective variables.

select_session_batch(*selector*)

Returns SessionBatchEnvironment with sessions (observations, actions, rewards) from pool at given indices.

Parameters **selector** – An array of integers that contains all indices of sessions to take.

Note that if this environment did not load is_alive or preceding_memory, you won't be able to use them at the SessionBatchEnvironment

sample_session_batch(*max_n_samples*, *replace=False*, *selector_dtype='int32'*)

Return SessionBatchEnvironment with sessions (observations, actions, rewards) that will be sampled uniformly from this session pool.

If replace=False, the amount of samples is min(max_n_sample, current pool). Otherwise it equals max_n_samples.

The chosen session ids will be sampled at random using self.rng on each iteration. P.S. There is no need to propagate rng updates! It does so by itself. Unless you are calling it inside theano.scan, ofc, but i'd recommend that you didn't. unroll_scan works ~probably~ perfectly fine btw

agentnet.environment.**SessionBatchEnvironment**(*observations*, *single_observation_shapes*, *actions=None*, *single_action_shapes='all_scalar'*, *rewards=None*, *is_alive=None*, *preceding_agent_memories=None*)

A generic pseudo-environment that replays sessions defined on creation by theano expressions ignoring agent actions completely.

The environment takes symbolic expression for sessions represented as (.observations, .actions, .rewards) Unlike SessionPoolEnvironment, this one does not store it's own pool of sessions.

To create experience-replay sessions, call Agent.get_sessions with this as an environment.

Parameters

- **observations** (*theano tensor or a list of such*) – a tensor or a list of tensors matching agent observation sequence [batch, tick, whatever]
- **observation shapes** (*single*) – shapes of one-tick one-batch-item observations. E.g. if lasagne shape is [None, 25(ticks), 3,210,160], than single_observation_shapes must contain [3,210,160]
- **actions** (*theano tensor or a list of such*) – a tensor or a list of tensors matching agent actions sequence [batch, tick, whatever]
- **action shapes** (*single*) – shapes of one-tick one-batch-item actions. Similar to observations. All scalar means that each action has shape (,), lasagne sequence layer being of shape (None, seq_length)
- **rewards** (*theano tensor*) – a tensor matching agent rewards sequence [batch, tick]
- **is_alive** (*theano tensor or None*) – whether or not session has still not finished by a particular tick. Always alive by default.
- **preceding_agent_memory** – a tensor or a list of such storing what was in agent's memory prior to the first tick of the replay session.

How does it tick:

During experience replay sessions,

- observations, actions and rewards match original ones
- agent memory states, Q-values and all in-agent expressions (but for actions) will correspond to what agent thinks NOW about the replay (not what it thought when he commited actions)
- preceding_agent_memory [optional] - what was agent's memory state prior to the first tick of the replay session.

Although it is possible to get rewards via the regular functions, it is usually faster to take self.rewards as rewards with no additional computation.

CHAPTER 7

Memory layers

This module contains a number of Lasagne layers useful when designing agent memory.

Memory layers can be vaguely divided into classical recurrent layers (e.g. RNN, LSTM, GRU) and augmentations (Stack augmentation, window augmentation, etc.).

Technically memory layers are lasagne layers that take previous memory state and some optional inputs to return new memory state.

For example, to build RNN with 36 units one has to define

#RNN input rnn_input = some_lasagne_layer

#rnn state from previous tick prev_rnn = InputLayer((None,36)) #None for batch size

#new RNN state (i.e. sigma(Wi * rnn_input + Wh * prev_rnn + b)) new_rnn = RNNCell(prev_rnn, rnn_input)

When using inside Agent (MDPAgent) or Recurrence, one must register them as agent_states (for agent) or state_variables (for recurrence), e.g.

from agentnet.agent import Agent agent = Agent(observations,{new_rnn : prev_rnn},...)

Standard recurrent layers

`agentnet.memory.`**`RNNCell`**(*prev_state*, *input_or_inputs=()*, *nonlinearity=<function tanh>*, *num_units=None*, *name=None*, *grad_clipping=0*, *Whid=<lasagne.init.Uniform object>*, *Winp=<lasagne.init.Uniform object>*, *b=<lasagne.init.Constant object>*)

Implements a one-step recurrent neural network (RNN) with arbitrary number of units.

Parameters

- **`prev_state`** – input that denotes previous state (shape must be (None, n_units))
- **`input_or_inputs`** – a single layer or a list/tuple of layers that go as inputs
- **`nonlinearity`** – which nonlinearity to use

- **num_units** – how many recurrent cells to use. None means "as in prev_state"
- **grad_clipping** – maximum gradient absolute value. 0 or None means "no clipping"

Returns updated memory layer

Return type lasagne.layers.Layer

for developers: Works by stacking DenseLayers with ElemwiseSumLayer. is a function mock, not actual class.

agentnet.memory.**GRUCell**(*prev_state*, *input_or_inputs=()*, *num_units=None*, *weight_init=<lasagne.init.Normal object>*, *bias_init=<lasagne.init.Constant object>*, *forgetgate_nonlinearity=<function sigmoid>*, *updategate_nonlinearity=<function sigmoid>*, *hidden_update_nonlinearity=<function tanh>*, *dropout=0*, *name='YetAnotherGRULayer'*, *grad_clipping=0*)

Implements a one-step gated recurrent unit (GRU) with arbitrary number of units.

Parameters

- **prev_state** (*lasagne.layers.Layer*) – input that denotes previous state (shape must be (None, n_units))
- **input_or_inputs** (*lasagne.layers.Layer or list of such*) – a single layer or a list/tuple of layers that go as inputs
- **num_units** (*int*) – how many recurrent cells to use. None means "as in prev_state"
- **weight_init** – either a lasagne initializer to use for every gate weights or a list of two initializers: - first used for all weights from hidden -> <any>_gate and hidden update - second used for all weights from input(s) -> <any>_gate weights and hidden update or a list of two objects elements: - second list is hidden -> forget gate, update gate, hidden update - second list of lists where list[i][0,1,2] = input[i] -> [forget gate, update gate, hidden update]
- **<any>_nonlinearity** – which nonlinearity to use for a particular gate
- **dropout** – dropout rate as per https://arxiv.org/abs/1603.05118
- **grad_clipping** – maximum gradient absolute value. 0 or None means "no clipping"

Returns updated memory layer

Return type lasagne.layers.Layer

for developers: Works by stacking other lasagne layers; is a function mock, not actual class.

agentnet.memory.**LSTMCell**(*prev_cell*, *prev_out*, *input_or_inputs=()*, *num_units=None*, *peepholes=True*, *weight_init=<lasagne.init.Normal object>*, *bias_init=<lasagne.init.Constant object>*, *peepholes_W_init=<lasagne.init.Normal object>*, *forgetgate_nonlinearity=<function sigmoid>*, *ingate_nonlinearity=<function sigmoid>*, *outgate_nonlinearity=<function sigmoid>*, *cell_nonlinearity=<function tanh>*, *output_nonlinearity=<function tanh>*, *dropout=0.0*, *name=None*, *grad_clipping=0.0*)

Implements a one-step LSTM update. Note that LSTM requires both c_t (private memory) and h_t aka output.

Parameters

- **prev_cell** (*lasagne.layers.Layer*) – input that denotes previous "private" state (shape must be (None, n_units))

- **prev_out** (*lasagne.layers.Layer*) – input that denotes previous "public" state (shape must be (None,n_units))
- **input_or_inputs** (*lasagne.layers.Layer or list of such*) – a single layer or a list/tuple of layers that go as inputs
- **num_units** (*int*) – how many recurrent cells to use. None means "as in prev_state"
- **peepholes** (*bool*) – If True, the LSTM uses peephole connections. When False, peepholes_W_init are ignored.
- **bias_init** – either a lasagne initializer to use for every gate weights or a list of 4 initializers for [input gate, forget gate, cell, output gate]
- **weight_init** – either a lasagne initializer to use for every gate weights: or a list of two initializers, - first used for all weights from hidden -> <all>_gate and cell - second used for all weights from input(s) -> <all>_gate weights and cell or a list of two objects elements, - second list is hidden -> input gate, forget gate, cell, output gate, - second list of lists where list[i][0,1,2] = input[i] -> [input_gate, forget gate, cell,output gate]
- **peepholes_W_init** – either a lasagne initializer or a list of 3 initializers for [input_gate, forget gate,output gate] weights. If peepholes=False, this is ignored.
- **<any>_nonlinearity** – which nonlinearity to use for a particular gate
- **dropout** – dropout rate as per https://arxiv.org/pdf/1603.05118.pdf
- **grad_clipping** – maximum gradient absolute value. 0 or None means "no clipping"

Returns a tuple of (new_cell,new_output) layers

Return type (lasagne.layers.Layer,lasagne.layers.Layer)

for developers: Works by stacking other lasagne layers; is a function mock, not actual class.

Augmentations

agentnet.memory.**AttentionLayer**(*input_sequence*, *query*, *num_units*, *mask_input=None*, *key_sequence=None*, *nonlinearity=<theano.tensor.elemwise.Elemwise object>*, *probs_nonlinearity=<function softmax>*, *W_enc=<lasagne.init.Normal object>*, *W_dec=<lasagne.init.Normal object>*, *W_out=<lasagne.init.Normal object>*, ***kwargs*)

A layer that implements basic Bahdanau-style attention. Implementation is inspired by tfnn@yandex.

Kurzgesagt, attention lets network decide which fraction of sequence/image should it view now by using small one-layer block that predicts (input_element,query) -> do i want to see input_element for all input_elements. You can read more about it here - http://distill.pub/2016/augmented-rnns/ .

AttentionLayer also allows you to have separate keys and values: it computes logits with keys, then converts them to weights(probs) and averages _values_ with those weights.

This layer outputs a dict with keys "attn" and "probs" - attn - inputs processed with attention, shape [batch_size, enc_units] - probs - probabilities for each activation [batch_size, seq_length]

This layer assumes input sequence/image/video/whatever to have 1 spatial dimension (see below). - rnn/emb format [batch,seq_len,units] works out of the box - 1d convolution format [batch,units,seq_len] needs dimshuffle(conv,[0,2,1]) - 2d convolution format [batch,units,dim1,dim2] needs two-step procedure - step1 = dimshuffle(conv,[0,2,3,1]) - step2 = reshape(step1,[-1,dim1*dim2,units]) - higher dimensionality follows the same prin-

ciple as 2d example above - reshape and dimshuffle can both be found in lasagne.layers (aliases to ReshapeLayer and DimshuffleLayer)

When calling get_output, you can pass flag hard_attention=True to replace attention with argmax over logits.

Parameters

- **input_sequence** (*lasagne.layers.Layer with shape [batch, seq_length, units]*) – sequence of inputs to be processed with attention
- **query** (*lasagne.layers.Layer with shape [batch, units]*) – single time-step state of decoder (usually lstm/gru/rnn hid)
- **num_units** (*int*) – number of hidden units in attention intermediate activation
- **key_sequence** (*lasagne.layers.Layer with shape [batch, seq_length, units] or None*) – a sequence of keys to compute dot_product with. By default, uses input_sequence instead.
- **nonlinearity** (*function(x) -> x that works with theano tensors*) – nonlinearity in attention intermediate activation
- **probs_nonlinearity** (*function(x) -> x that works with theano tensors*) – nonlinearity that converts logits of shape [batch,seq_length] into attention weights of same shape (you can provide softmax with tunable temperature or gumbel-softmax or anything of the sort)
- **mask_input** (*lasagne.layers.Layer with shape [batch, seq_length]*) – mask for input_sequence (like other lasagne masks). Default is no mask

Other params can be theano shared variable, expression, numpy array or callable. Initial value, expression or initializer for the weights. These should be a matrix with shape `(num_inputs, num_units)`. See `lasagne.utils.create_param()` for more information.

The roles of those params are: W_enc - weights from encoder (each state) to hidden layer W_dec - weights from decoder (each state) to hidden layer W_out - hidden to logit weights No logit biases are introduces because softmax is invariant to adding bias to each logit

`agentnet.memory.`**`DotAttentionLayer`**(*input_sequence*, *query*, *key_sequence=None*, *mask_input=None*, *scale=False*, *use_dense_layer=False*, *probs_nonlinearity=<function softmax>*, ***kwargs*)

A layer that implements multiplicative (Dotproduct) attention. Implementation is inspired by tfnn@yandex.

Unlike AttentionLayer, DotAttention requires you to provide query in the same shape as one time-step of the input sequence. Otherwise it does so via DenseLayer.

DotAttention also allows you to have separate keys and values: it computes logits with keys, then converts them to weights(probs) and averages _values_ with those weights.

Kurzgesagt, attention lets network decide which fraction of sequence/image should it view now by using small one-layer block that predicts (input_element,query) -> do i want to see input_element for all input_elements. You can read more about it here - http://distill.pub/2016/augmented-rnns/ .

This layer outputs a dict with keys "attn" and "probs" - attn - inputs processed with attention, shape [batch_size, enc_units] - probs - probabilities for each activation [batch_size, seq_length]

This layer assumes input sequence/image/video/whatever to have 1 spatial dimension (see below). - rnn/emb format [batch,seq_len,units] works out of the box - 1d convolution format [batch,units,seq_len] needs dimshuffle(conv,[0,2,1]) - 2d convolution format [batch,units,dim1,dim2] needs two-step procedure

•step1 = dimshuffle(conv,[0,2,3,1])

•step2 = reshape(step1,[-1,dim1*dim2,units])

•higher dimensionality follows the same principle as 2d example above

•reshape and dimshuffle can both be found in lasagne.layers (aliases to ReshapeLayer and DimshuffleLayer)

When calling get_output, you can pass flag hard_attention=True to replace attention with argmax over logits.

Parameters

- **input_sequence** (*lasagne.layers.Layer with shape [batch, seq_length, units]*) – sequence of inputs to be processed with attention
- **query** (*lasagne.layers.Layer with shape [batch, units]*) – single time-step state of decoder that is used as query (usually custom layer or lstm/gru/rnn hid) If it matches input_sequence one-step size, query is used as is. Otherwise, DotAttention is performed from DenseLayer(query,input_units,nonlinearity=None).
- **key_sequence** (*lasagne.layers.Layer with shape [batch, seq_length, units] or None*) – a sequence of keys to compute dot_product with. By default, uses input_sequence instead.
- **mask_input** (*lasagne.layers.Layer with shape [batch, seq_length]*) – mask for input_sequence (like other lasagne masks). Default is no mask
- **scale** – if True, scales query.dot(key) by key_size**-0.5 to maintain variance. Otherwise does nothing.
- **use_dense_layer** – if True, forcibly creates intermediate dense layer on top of query
- **probs_nonlinearity** (*function(x) -> x that works with theano tensors*) – nonlinearity that converts logits of shape [batch,seq_length] into attention weights of same shape (you can provide softmax with tunable temperature or gumbel-softmax or anything of the sort)

agentnet.memory.**StackAugmentation**(*observation_input*, *prev_state_input*, *controls_layer*, ***kwargs*)

A special kind of memory augmentation that implements end-to-end diferentiable stack in accordance to this paper: http://arxiv.org/abs/1503.01007

Parameters

- **observation_input** (*lasagne.layers.Layer*) – an item that can be pushed into the stack (e.g. RNN state)
- **prev_state_input** (*lasagne.layers.Layer (usually InputLayer)*) – revious stack state of shape [batch,stack depth, stack item size]
- **controls_layer** (*lasagne.layers.layer (usually DenseLayer with softmax nonlinearity)*) – a layer with 3 channels: PUSH_OP, POP_OP and NO_OP accordingly (must sum to 1)

A simple snippet that runs that augmentation from the Stack RNN example stack_width = 3 stack_depth = 50 # previous stack goes here prev_stack_layer = InputLayer((None,stack_depth,stack_width)) # Stack controls - push, pop and no-op stack_controls_layer = DenseLayer(<rnn>,3, nonlinearity=lasagne.nonlinearities.softmax,) # stack input stack_input_layer = DenseLayer(<rnn>,stack_width) #new stack state next_stack = StackAugmentation(stack_input_layer,prev_stack_layer,stack_controls_layer)

agentnet.memory.**WindowAugmentation**(*new_value_input*, *prev_state_input*, ***kwargs*)

An augmentation that holds K previous items in memory, used in DeepMind Atari architecture from original article

Supports a window of K last values of new_value_input. Each time new element is pushed into the window, the oldest one gets out.

Parameters

- **new_value_input** (*lasagne.layers.Layer, shape must be compatible with prev_state_input (see next)*) – a newest item to be stored in the window
- **prev_state_input** (*lasagne.layers.Layer, normally InputLayer*) – previous window state of shape [batch, window_length, item_size]

Shapes of new_value_input and prev_state_input must match: if new_value_input has shape (batch, a, b, c) than prev_state_input must have shape (batch, window_size, a, b, c)

where a,b,c stands for arbitrary shapes (e.g. channel, width and height of an image). An item can have arbitrary number of dimensions as long as first one is batch_size

Window shape and K are defined as prev_state_input.output_shape

The state shape consists of (batch_i, relative_time_inverted, new_value shape) So, last inserted value would be at state[:,0], pre-last - at state[:,1] etc.

And yes, K = prev_state_input.output_shape[1].

agentnet.memory.**CounterLayer**(*prev_counter*, *k=None*, *name=None*)

A simple counter Layer that increments it's state by 1 each turn and loops each k iterations

Parameters

- **prev_counter** (*lasagne.layers.Layer, normally InputLayer*) – previous state of counter
- **k** – if not None, resets counter to zero each k timesteps

Returns incremented counter

Return type lasagne.layers.Layer

agentnet.memory.**SwitchLayer**(*condition*, *than_branch*, *else_branch*, *name=None*)

a simple layer that implements an 'if-than-else' logic

Parameters

- **condition** (*lasagne.layers.Layer*) – a layer with [batch_size] boolean conditions (dtype int*)
- **than_branch** – branch that happens if condition != 0 for particular element of a batch
- **else_branch** – branch that happens if condition == 0 for particular element of a batch

Shapes and dtypes of the two branches must match.

Returns a layer where i-th batch sample will take than_branch value if condition, else else_branch value

Return type lasagne.layers.Layer

Low-level layers

agentnet.memory.**GateLayer**(*gate_controllers*, *channels*, *gate_nonlinearities=<function sigmoid>*, *bias_init=<lasagne.init.Constant object>*, *weight_init=<lasagne.init.Normal object>*, *channel_names=None*, ***kwargs*)

An overly generic interface for one-step gate, stacked gates or gate applier. If several channels are given, stacks them for quicker execution.

Parameters

- **gate_controllers** – a single layer or a list/tuple of such layers that gate depends on (for most RNNs, that's input and previous memory state)
- **channels** – a single layer or integer or a list/tuple of layers/integers if a layer, that defines a layer that should be multiplied by the gate output if an integer that defines a number of units of a gate – and these are the units to be returned
- **gate_nonlinearities** – a single function or a list of such(channel-wise), - defining nonlinearities for gates on corresponding channels
- **bias_init** –
  - an initializer or a list (channel-wise) of initializers for bias(b) parameters
  - (None, lasagne.init, theano variable or numpy array)
  - None means no bias
- **weight_init** –
  - an initializer OR a list of initializers for (channel-wise)
  - OR a list of lists of initializers (channel, controller)
  - (lasagne.init, theano variable or numpy array)

CHAPTER 8

Resolvers

Layers that convert policy or Q-value vectors into action ids

agentnet.resolver.**BaseResolver**(*incoming*, *name='BaseResolver'*, *output_dtype='int32'*)

Special Lasagne Layer instance, that determines actions agent takes given policy (e.g. Q-values),

agentnet.resolver.**EpsilonGreedyResolver**(*incoming*, *epsilon=None*, *seed=1234*, *name='EpsilonGreedyResolver'*, ***kwargs*)

Epsilon-greedy resolver:

- determines which action should be taken given agent's policy,
- takes maximum policy action with probability 1 - epsilon
- takes random action with probability epsilon

agentnet.resolver.**ProbabilisticResolver**(*incoming*, *assume_normalized=False*, *seed=1234*, *output_dtype='int32'*, *name='ProbabilisticResolver'*)

instance, that:

- determines which action should be taken given policy
- samples actions with probabilities given by input layer

CHAPTER 9

Learning Algorithms

This module contains implementations of various reinforcement learning algorithms.

The core API of each learning algorithm is .get_elementwise_objective that returns the per-tick loss that you can minimize over NN weights using e.g. lasagne.updates.your_favorite_method.

Q-learning

Q-learning implementation. Works with discrete action space. Supports n-step updates and custom state value function (max(Q(s,a)), double q-learning, boltzmann, mellowmax, expected value sarsa,...)

agentnet.learning.qlearning.**get_elementwise_objective**(*qvalues*, *actions*, *rewards*, *is_alive='always'*, *qvalues_target=None*, *state_values_target=None*, *n_steps=1*, *gamma_or_gammas=0.99*, *crop_last=True*, *state_values_target_after_end='zeros'*, *consider_reference_constant=True*, *aggregation_function='deprecated'*, *force_end_at_last_tick=False*, *return_reference=False*, *loss_function=<function squared_error>*)

Returns squared error between predicted and reference Q-values according to n-step Q-learning algorithm

Qreference(state,action) = reward(state,action) + gamma*reward(state_1,action_1) + ... + gamma^n * max[action_n](Q(state_n,action_n) loss = mean over (Qvalues - Qreference)**2

Parameters

- **qvalues** – [batch,tick,actions] - predicted qvalues
- **actions** – [batch,tick] - commited actions
- **rewards** – [batch,tick] - immediate rewards for taking actions at given time ticks
- **is_alive** – [batch,tick] - whether given session is still active at given tick. Defaults to always active.
- **qvalues_target** – Q-values used when computing reference (e.g. r+gamma*Q(s',a_max). shape [batch,tick,actions] examples: (default) If None, uses current Qvalues. Older snapshot Qvalues (e.g. from a target network)
- **state_values_target** – state values V(s), used when computing reference (e.g. r+gamma*V(s'), shape [batch_size,seq_length,1] double q-learning V(s) = Q_old(s,argmax Q_new(s,a)) expected_value_sarsa V(s) = E_a~pi(als) Q(s,a) state values from teacher network (knowledge transfer)

Must provide either nothing or qvalues_target or state_values_target, not both at once

Parameters

- **n_steps** – if an integer is given, uses n-step q-learning algorithm If 1 (default), this works exactly as normal q-learning If None: propagating rewards throughout the whole sequence of state-action pairs.
- **gamma_or_gammas** – delayed reward discounts: a single value or array[batch,tick](can broadcast dimensions).
- **crop_last** – if True, zeros-out loss at final tick, if False - computes loss VS Qvalues_after_end
- **state_values_target_after_end** – [batch,1] - symbolic expression for "next best q-values" for last tick used when computing reference Q-values only. Defaults at T.zeros_like(Q-values[:,0,None,0]). if crop_last=True, simply does not penalize at last tick. If you wish to simply ignore the last tick, use defaults and crop output's last tick (qref[:,:-1])
- **consider_reference_constant** – whether or not zero-out gradient flow through reference_qvalues (True is highly recommended)
- **force_end_at_last_tick** – if True, forces session end at last tick unless ended otehrwise
- **return_reference** – if True, returns reference Qvalues. If False, returns squared_error(action_qvalues, reference_qvalues)
- **loss_function** – loss_function(V_reference,V_predicted). Defaults to (V_reference-V_predicted)**2. Use to override squared error with different loss (e.g. Huber or MAE)

Returns mean squared error over Q-values (using formula above for loss)

SARSA

State-Action-Reward-State-Action (sars'a') learning algorithm implementation. Supports n-step eligibility traces. This is an on-policy SARSA. To use off-policy Expected Value SARSA, use agentnet.learning.qlearning with custom aggregation_function

agentnet.learning.sarsa.**get_elementwise_objective**(*qvalues*, *actions*, *rewards*, *is_alive='always'*, *qvalues_target=None*, *n_steps=1*, *gamma_or_gammas=0.99*, *crop_last=True*, *state_values_target_after_end='zeros'*, *consider_reference_constant=True*, *force_end_at_last_tick=False*, *return_reference=False*, *loss_function=<function squared_error>*)

Returns squared error between predicted and reference Q-values according to n-step SARSA algorithm Qreference(state,action) = reward(state,action) + gamma*reward(state_1,action_1) + ... + gamma^n*Q(state_n,action_n) loss = mean over (Qvalues - Qreference)**2

Parameters

- **qvalues** – [batch,tick,action_id] - predicted qvalues
- **actions** – [batch,tick] - commited actions
- **rewards** – [batch,tick] - immediate rewards for taking actions at given time ticks
- **is_alive** – [batch,tick] - whether given session is still active at given tick. Defaults to always active.
- **qvalues_target** – Q-values[batch,time,actions] or V(s)[batch_size,seq_length,1] used for reference. Some examples: (default) If None, uses current Qvalues. Older snapshot Qvalues (e.g. from a target network) Double q-learning V(s) = Q_old(s,argmax Q_new(s,a))[:,:,None] State values from teacher network (knowledge transfer)
- **n_steps** – if an integer is given, uses n-step sarsa algorithm If 1 (default), this works exactly as normal SARSA If None: propagating rewards throughout the whole sequence of state-action pairs.
- **gamma_or_gammas** – delayed reward discounts: a single value or array[batch,tick](can broadcast dimensions).
- **crop_last** – if True, zeros-out loss at final tick, if False - computes loss VS Qvalues_after_end
- **state_values_target_after_end** – [batch,1] - symbolic expression for "best next state q-values" for last tick used when computing reference Q-values only. Defaults at T.zeros_like(Q-values[:,0,None,0]) If you wish to simply ignore the last tick, use defaults and crop output's last tick (qref[:,:-1])
- **consider_reference_constant** – whether or not zero-out gradient flow through reference_qvalues (True is highly recommended)
- **force_end_at_last_tick** – if True, forces session end at last tick unless ended otehrwise
- **loss_function** – loss_function(V_reference,V_predicted). Defaults to (V_reference-V_predicted)**2. Use to override squared error with different loss (e.g. Huber or MAE)
- **return_reference** – if True, returns reference Qvalues. If False, returns squared_error(action_qvalues, reference_qvalues)

Returns loss [squared error] over Q-values (using formula above for loss)

Advantage actor-critic

Advantage Actor-Critic (A2c or A3c) implementation. Follows the article http://arxiv.org/pdf/1602.01783v1.pdf Supports K-step advantage estimation as in https://arxiv.org/pdf/1506.02438v5.pdf

Agent should output action probabilities and state values instead of Q-values. Works with discrete action space only.

`agentnet.learning.a2c.`**`get_elementwise_objective`**(*policy*, *state_values*, *actions*, *rewards*, *is_alive='always'*, *state_values_target=None*, *n_steps=1*, *n_steps_advantage='same'*, *gamma_or_gammas=0.99*, *crop_last=True*, *state_values_target_after_end='zeros'*, *state_values_after_end='zeros'*, *consider_value_reference_constant=True*, *force_end_at_last_tick=False*, *return_separate=False*, *treat_policy_as_logpolicy=False*, *loss_function=<function squared_error>*)

returns cross-entropy-like objective function for Actor-Critic method L_policy = - log(policy) * A(s,a) L_V = (V - Vreference)^2 where A(s,a) is an advantage term (e.g. [r+gamma*V(s') - V(s)]) and Vreference is reference state values as per Temporal Difference.

Parameters

- **`policy`** – [batch,tick,action_id] or [batch,tick] - predicted probabilities for all actions (3-dim) or chosen actions (2-dim).
- **`state_values`** – [batch,tick] - predicted state values
- **`actions`** – [batch,tick] - committed actions
- **`rewards`** – [batch,tick] - immediate rewards for taking actions at given time ticks
- **`is_alive`** – [batch,tick] - binary matrix whether given session is active at given tick. Defaults to all ones.
- **`state_values_target`** – there should be state values used to compute reference (e.g. older network snapshot) If None (defualt), uses current Qvalues to compute reference
- **`n_steps`** – if an integer is given, the STATE VALUE references are computed in loops of 3 states. If 1 (default), this uses a one-step TD, i.e. reference_V(s) = r+gamma*V(s') If None: propagating rewards throughout the whole session and only taking V(s_last) at the session end.
- **`n_steps_advantage`** – same as n_steps, but for advantage term A(s,a) (see above). Defaults to same as n_steps
- **`gamma_or_gammas`** – a single value or array[batch,tick](can broadcast dimensions) of discount for delayed reward
- **`crop_last`** – if True, zeros-out loss at final tick, if False - computes loss VS Qvalues_after_end
- **`force_values_after_end`** – if true, sets reference policy at session end to rewards[end] + qvalues_after_end

- **state_values_target_after_end** – [batch,1] - "next target state values" after last tick; used for reference. Defaults at T.zeros_like(state_values_target[:,0,None,:])
- **state_values_after_end** – [batch,1] - "next state values" after last tick; used for reference. Defaults at T.zeros_like(state_values[:,0,None,:])
- **consider_value_reference_constant** – whether or not to zero-out critic gradients through the reference state values term
- **return_separate** – if True, returns a tuple of (actor loss , critic loss) instead of their sum.
- **treat_policy_as_logpolicy** – if True, policy is used as log(pi(a|s)). You may want to do this for numerical stability reasons.
- **loss_function** – loss_function(V_reference,V_predicted) used for CRITIC. Defaults to (V_reference-V_predicted)**2 Use to override squared error with different loss (e.g. Huber or MAE)
- **force_end_at_last_tick** – if True, forces session end at last tick unless ended otherwise

Returns elementwise sum of policy_loss + state_value_loss [batch,tick]

Deterministic policy gradient

Deterministic policy gradient loss, Also used for model-based acceleration algorithms. Supports regular and k-step implementation. Based on: - http://arxiv.org/abs/1509.02971 - http://arxiv.org/abs/1603.00748 - http://jmlr.org/proceedings/papers/v32/silver14.pdf

agentnet.learning.dpg.**get_elementwise_objective_critic**(*action_qvalues*, *state_values*, *rewards*, *is_alive='always'*, *n_steps=1*, *gamma_or_gammas=0.99*, *crop_last=True*, *state_values_after_end='zeros'*, *force_end_at_last_tick=False*, *consider_reference_constant=True*, *return_reference=False*, *loss_function=<function squared_error>*, *scan_dependencies=()*, *scan_strict=True*)

Returns squared error between action values and reference (r+gamma*V(s')) according to deterministic policy gradient.

This function can also be used for any model-based acceleration like Qlearning with normalized advantage functions.

- Original article: http://arxiv.org/abs/1603.00748
- **Since you can provide any state_values, you can technically use any other advantage function shape** as long as you can compute V(s).

If n_steps > 1, the algorithm will use n-step Temporal Difference updates V_reference(state,action) = reward(state,action) + gamma*reward(state_1,action_1) + ... + gamma^n * V(state_n)

Parameters

- **action_qvalues** – [batch,tick,action_id] - predicted qvalues
- **state_values** – [batch,tick] - predicted state values (aka qvalues for best actions)
- **rewards** – [batch,tick] - immediate rewards for taking actions at given time ticks
- **is_alive** – [batch,tick] - whether given session is still active at given tick. Defaults to always active. Default value of is_alive implies a simplified computation algorithm for Qlearning loss
- **n_steps** – if an integer is given, uses n-step TD algorithm If 1 (default), this works exactly as normal TD If None: propagating rewards throughout the whole sequence of state-action pairs.
- **gamma_or_gammas** – delayed reward discounts: a single value or array[batch,tick](can broadcast dimensions).
- **crop_last** – if True, zeros-out loss at final tick, if False - computes loss VS Qvalues_after_end
- **state_values_after_end** – [batch,1] - symbolic expression for "best next state q-values" for last tick used when computing reference Q-values only. Defaults at T.zeros_like(Q-values[:,0,None,0]) If you wish to simply ignore the last tick, use defaults and crop output's last tick (qref[:,:-1])
- **force_end_at_last_tick** – if True, forces session end at last tick unless ended otehrwise
- **consider_reference_constant** – whether or not zero-out gradient flow through reference_qvalues (True is highly recommended)
- **return_reference** – if True, returns reference Qvalues. If False, returns loss_function(action_Qvalues, reference_qvalues)
- **loss_function** – loss_function(V_reference,V_predicted). Defaults to (V_reference-V_predicted)**2. Use to override squared error with different loss (e.g. Huber or MAE)

Returns mean squared error over Q-values (using formula above for loss)

Implements layers required to train qlearning with normalized advantage functions. All the math taken from the original article: http://arxiv.org/abs/1603.00748 Loss function is exactly same as deterministic policy gradient (agentnet.learning.dpg) Usage example: https://github.com/yandexdataschool/AgentNet/blob/master/examples/Continuous%20LunarLander%20%20using%20normalized%20advantage%20functions.ipynb

agentnet.learning.qlearning_naf.**LowerTriangularLayer**(*incoming*, *matrix_diag=None*, ***kwargs*)

agentnet.learning.qlearning_naf.**NAFLayer**(*action_layer*, *mean_layer*, *L_layer*, ***kwargs*)

Generic

Several helper functions used in various reinforcement learning algorithms.

agentnet.learning.generic.**get_values_for_actions**(*values_for_all_actions*, *actions*)

Auxiliary function to select policy/Q-values corresponding to chosen actions. :param values_for_all_actions: qvalues or similar for all actions: floatX[batch,tick,action] :param actions: action ids int32[batch,tick] :returns: values selected for the given actions: float[batch,tick]

agentnet.learning.generic.**get_end_indicator**(*is_alive*, *force_end_at_t_max=False*)

Auxiliary function to transform session alive indicator into end action indicator :param force_end_at_t_max: if True, all sessions that didn't end by the end of recorded sessions are ended at the last recorded tick.

agentnet.learning.generic.**get_n_step_value_reference**(*state_values*, *rewards*, *is_alive='always'*, *n_steps=None*, *gamma_or_gammas=0.99*, *crop_last=True*, *state_values_after_end='zeros'*, *end_at_tmax=False*, *force_n_step=False*)

Computes the reference for state value function via n-step TD algorithm:

Vref = r(t) + gamma*r(t+1) + gamma^2*r(t+2) + ... + gamma^n*V(s[t+n]) where n == n_steps

Used by all n_step methods, including Q-learning, a2c and dpg

Works with both Q-values and state values, depending on aggregation_function

Parameters

- **state_values** – float[batch,tick] predicted state values V(s) at given batch session and time tick - for Q-learning, it's max over Q-values - for state-value based methods (a2c, dpg), it's same as state_values
- **rewards** –
  - float[batch,tick] rewards achieved by commiting actions at [batch,tick]
- **is_alive** – whether the session is still active at given tick, int[batch_size,time] of ones and zeros
- **n_steps** – if an integer is given, the references are computed in loops of n_steps Every n_steps'th step reference is set to V = r + gamma * next V_predicted On other steps, reference is propagated V = r + gamma * next V reference Defaults to None: propagating rewards throughout the whole session. Widely known as "lambda" in RL community (TD-lambda, Q-lambda) plus or minus one :) If n_steps equals 1, this works exactly as regular TD (though a less efficient one) If you provide symbolic integer here AND strict = True, make sure you added the variable to dependencies.
- **gamma_or_gammas** – delayed reward discount number, scalar or vector[batch_size]
- **crop_last** – if True, ignores loss for last tick(default)
- **state_values_after_end** –
  - symbolic expression for "next state values" for last tick used for reference only.

  Defaults at T.zeros_like(values[:,0,None,:]) If you wish to simply ignore the last tick, use defaults and crop output's last tick (qref[:,:-1])
- **end_at_tmax** – if True, forces session end at last tick if there was no other session end.
- **force_n_step** – if True, does NOT fall back to 1-step algorithm if n_steps = 1

Returns V reference [batch,action_at_tick] according n-step algorithms ~ eligibility traces e.g. mentioned here http://arxiv.org/pdf/1602.01783.pdf as A3c and k-step Q-learning also described here https://arxiv.org/pdf/1506.02438v5.pdf for k-step advantage

agentnet.learning.generic.**get_1_step_value_reference**(*state_values*, *rewards*, *is_alive='always'*, *gamma_or_gammas=0.99*, *crop_last=True*, *state_values_after_end='zeros'*, *end_at_tmax=False*)

Computes the reference for state value function via 1-step TD algorithm:

Vref = r(t) + gamma*V(s')

Used as a fall-back by n-step algorithm when n_steps=1 (performance reasons)

Parameters

- **state_values** – float[batch,tick] predicted state values V(s) at given batch session and time tick - for Q-learning, it's max over Q-values - for state-value based methods (a2c, dpg), it's same as state_values
- **rewards** –
  - float[batch,tick] rewards achieved by committing actions at [batch,tick]
- **is_alive** – whether the session is still active int/bool[batch_size,time]
- **gamma_or_gammas** – delayed reward discount number, scalar or vector[batch_size]
- **crop_last** – if True, ignores loss for last tick(default)
- **state_values_after_end** –
  - symbolic expression for "next state values" for last tick used for reference only.

  Defaults at T.zeros_like(values[:,0,None,:]) If you wish to simply ignore the last tick, use defaults and crop output's last tick (qref[:,:-1])
- **end_at_tmax** – if True, forces session end at last tick if there was no other session end.

Returns V reference [batch,action_at_tick] = r + gamma*V(s')

CHAPTER 10

Target Network

Implements the target network techniques in deep reinforcement learning. In short, the idea is to estimate reference Qvalues not from the current agent state, but from an earlier snapshot of weights. This is done to decorrelate target and predicted Qvalues/state_values and increase stability of learning algorithm.

Some notable alterations of this technique: - Standard approach with older NN snapshot – https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

- Moving average of weights

– http://arxiv.org/abs/1509.02971

- Double Q-learning and other clever ways of training with target network

– http://arxiv.org/pdf/1509.06461.pdf

Here we implement a generic TargetNetwork class that supports both standard and moving average approaches through "moving_average_alpha" parameter of "load_weights".

class `agentnet.target_network.`**`TargetNetwork`**(*original_network_outputs*, *bottom_layers=()*, *share_inputs=True*, *name='target_net.'*)

A generic class for target network techniques. Works by creating a deep copy of the original network and synchronizing weights through "load_weights" method.

If you just want to duplicate lasagne layers with or without sharing params, use agentnet.utils.clone.clone_network

Parameters

- **`original_network_outputs`** (*`lasagne.layers.Layer or a list/tuple of such`*) – original network outputs to be cloned for target network
- **`bottom_layers`** (*`lasagne.layers.Layer or a list/tuple/dict of such.`*) – the layers that should be shared between networks.
- **`share_inputs`** (*`bool`*) – if True, all InputLayers will still be shared even if not mentioned in bottom_layers

Snippet

#build network from lasagne.layers l_in = InputLayer([None,10]) l_d0 = DenseLayer(l_in,20) l_d1 = DenseLayer(l_d0,30) l_d2 = DenseLayer(l_d1,40) other_l_d2 = DenseLayer(l_d1,41)

TargetNetwork that copies all the layers BUT FOR l_in full_clone = TargetNetwork([l_d2,other_l_d2]) clone_d2, clone_other_d2 = full_clone.output_layers

only copy l_d2 and l_d1, keep l_d0 and l_in from original network, do not clone other_l_d2 partial_clone = TargetNetwork(l_d2,bottom_layers=(l_d0)) clone_d2 = partial_clone.output_layers

do_something_with_l_d2_weights()

#synchronize parameters with original network partial_clone.load_weights()

#OR set clone_params = 0.33*original_params + (1-0.33)*previous_clone_params partial_clone.load_weights(0.33)

load_weights(*moving_average_alpha=1*)

Loads the weights from original network into target network. Should usually be called whenever you want to synchronize the target network with the one you train.

When using moving average approach, one should specify which fraction of new weights is loaded through moving_average_alpha param (e.g. moving_average_alpha=0.1)

Parameters **moving_average_alpha** – If 1, just loads the new weights. Otherwise target_weights = alpha*original_weights + (1-alpha)*target_weights

CHAPTER 11

Utilities

Helper functions for symbolic theano code

persistence

agentnet.utils.persistence.**save**(*nn*, *filename*)

Saves lasagne network weights to the target file. Does not store the architecture itself.

Basic usage: >> nn = lasagne.layers.InputLayer(...) >> nn = lasagne.layers.SomeLayer(...) >> nn = lasagne.layers.SomeLayer(...) >> train_my_nn() >> save(nn,"nn_weights.pcl")

Loading weights is possible through .load function in the same module.

Parameters

- **nn** – neural network output layer(s)
- **filename** – weight filename

agentnet.utils.persistence.**load**(*nn*, *filename*)

Loads lasagne network weights from the target file into NN you provided. Requires that NN architecture is exactly same as NN which weights were saved. Minor alterations like changing hard-coded batch size will probably work, but are not guaranteed.

Basic usage: >> nn = lasagne.layers.InputLayer(...) >> nn = lasagne.layers.SomeLayer(...) >> nn = lasagne.layers.SomeLayer(...) >> train_my_nn() >> save(nn,"previously_saved_weights.pcl") >> crash_and_lose_progress() >> nn = the_same_nn_as_before() >> load(nn,"previously_saved_weights.pcl")

Parameters

- **nn** – neural network output layer(s)
- **filename** – weight filename

Returns

the network with weights loaded

WARNING! the load() function is inplace, meaning that weights are loaded in the NN instance you provided and NOT in a copy.

clone network

Utility functions that can clone lasagne network layers in a custom way. [Will be] used for: - target networks, e.g. older copies of NN used for reference Qvalues. - DPG-like methods where critic has to process both optimal and actual actions

agentnet.utils.clone.**clone_network**(*original_network*, *bottom_layers=None*, *share_params=False*, *share_inputs=True*, *name_prefix=None*)

Creates a copy of lasagne network layer(s) provided as original_network.

If bottom_layers is a list of layers or a single layer, function won't copy these layers, using existing ones instead.

Else, if bottom_layers is a dictionary of {existing_layer:new_layer}, each time original network would have used existing_layer, cloned network uses new_layer

It is possible to either use existing weights or clone them via share_weights flag. If weights are shared, target_network will always have same weights as original one. Any changes (e.g. loading or training) will affect both original and cloned network. This is useful if you want both networks to train together (i.e. you have same network applied twice) One example of such case is Deep DPG algorithm: http://arxiv.org/abs/1509.02971

Otherwise, if weights are NOT shared, the cloned network will begin with same weights as the original one at the moment it was cloned, but than the two networks will be completely independent. This is useful if you want cloned network to deviate from original. One example is when you need a "target network" for your deep RL agent, that stores older weights snapshot. The DQN that uses this trick can be found here: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

Parameters

- **original_network** (*lasagne.layers.Layer or list/tuple/dict/any_iterable of such. If list, layers must be VALUES, not keys.*) – A network to be cloned (all output layers)
- **bottom_layers** (*lasagne.layers.Layer or a list/tuple/dict of such.*) – the layers which you don't want to clone. See description above. This parameter can also contain ARBITRARY objects within the original_network that you want to share.
- **share_params** – if True, cloned network will use same shared variables for weights. Otherwise new shared variables will be created and set to original NN values. WARNING! shared weights must be accessible via lasagne.layers.get_all_params with no flags If you want custom other parameters to be shared, use bottom_layers
- **share_inputs** (*bool*) – if True, all InputLayers will still be shared even if not mentioned in bottom_layers
- **name_prefix** (*string or None*) – if not None, adds this prefix to all the layers and params of the cloned network

Returns a clone of original_network (whether layer, list, dict, tuple or whatever

reapply

layers

agentnet.utils.layers.**DictLayer**(*incomings*, *output_shapes*, *output_dtypes=None*, ***kwargs*)
A base class for Lasagne layer that returns several outputs.

For a custom dictlayer you should implement get_output_for so that it returns a dict of {key:tensor_for_that_key}

By default it just outputs all the inputs IF their number matches, otherwise it raises an exception.

In other words, if you return 'foo' and 'bar' of shapes (None, 25) and (None,15,5,7), self.get_output_shape must be {'foo': (None,25), 'bar': (None,15,5,7)}

warning: this layer is needed for the purpose of graph optimization, it slightly breaks Lasagne conventions, so it is hacky.

Parameters

- **incomings** (*lasagne.layers.Layer or a list of such*) – Incoming layers.
- **output_shapes** (*dict of { output_key: tuple of shape dimensions (like input layer shape) } or a list of shapes, in which case keys are integers from 0 to len(output_shapes)*) – Shapes of key-value outputs from the DictLayer.
- **output_dtypes** (*None, dict of {key:dtype of output} or a list of dtypes. Key names must match those in output_shapes.*) – If provided, defines the dtypes of all key-value outputs. None means all float32.

agentnet.utils.layers.**get_layer_dtype**(*layer*, *default=None*)
takes layer's output_dtype property if it is defined, otherwise defaults to default or (if it's not given) theano.config.floatX

agentnet.utils.layers.**clip_grads**(*layer*, *clipping_bound*)
Clips grads passing through a lasagne.layers.layer

agentnet.utils.layers.**mul**(**args*, ***kwargs*)
Element-wise multiply layers

agentnet.utils.layers.**add**(**args*, ***kwargs*)
Element-wise sum of layers

format

agentnet.utils.format.**check_list**(*variables*)
Ensure that variables is a list or converts to one. If naive conversion fails, throws an error :param variables: sequence expected

agentnet.utils.format.**check_tuple**(*variables*)
Ensure that variables is a list or converts to one. If naive conversion fails, throws an error :param variables: sequence expected

agentnet.utils.format.**check_ordered_dict**(*variables*)
Ensure that variables is an OrderedDict :param variables: dictionary expected

agentnet.utils.format.**unpack_list**(*array*, *parts_lengths*)

Returns slices of the input list a. unpack_list(a, [2,3,5]) -> a[:2], a[2:2+3], a[2+3:2+3+5]

Parameters

- **array** – array-like or tensor variable
- **parts_lengths** – lengths of subparts

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index