RETURNN Documentation

Release 1.0-dev

RETURNN contributors

Jul 24, 2017

Contents

RETURNN paper.

RETURNN - RWTH extensible training framework for universal recurrent neural networks, is a Theano/TensorFlow-based implementation of modern recurrent neural network architectures. It is optimized for fast and reliable training of recurrent neural networks in a multi-GPU environment.

Features include:

- Mini-batch training of feed-forward neural networks
- Sequence-chunking based batch training for recurrent neural networks
- Long short-term memory recurrent neural networks including our own fast CUDA kernel
- Multidimensional LSTM (GPU only, there is no CPU version)
- Memory management for large data sets
- Work distribution across multiple devices

See *Basic usage* and *Technological overview*.

There are many example demos which work on artificially generated data, i.e. they should work as-is.

There are some real-world examples such as setups for speech recognition on the Switchboard corpus.

Some benchmark setups against other frameworks can be found here. The results are in the RETURNN paper.

There is also a wiki.

CHAPTER 1

User guide

Installation

Installation is easy. Install all dependencies, which are just Theano and h5py (`pip install theano h5py`). For TensorFlow, use `pip install tensorflow-gpu` and make sure that CUDA and cuDNN can be found. Then just checkout the Git repository of RETURNN (https://github.com/rwth-i6/returnn/) and you are ready to go.

See *Basic usage* for the basic usage.

Basic usage

Install RETURNN, *Installation*.

Now `rnn.py` is the main entry point. Usage:

```
./rnn.py <config-file> [other-params]
```

where `config-file` is a config file for RETURNN. See here for an example, and many more examples from the demos. We support multiple config syntax, such as simple line-based `key value`, JSON based which is determined by a "`{`" at the beginning of the file or Python code which is determined by a "`#!`" at the beginning of the file. There is a variety of config params.

`rnn.py` will execute some task, such as `train` or `forward`. You must define the task and the train and dev datasets which it will use, the mini-batch construction variants, as well as the neural network topology, as well as some training parameters.

Some common config parameters:

task The task, such as `train` or `forward`.

device E.g. `gpu` or `cpu`. Can also be `gpu0,gpu1` for multi-GPU training.

use_tensorflow If you set this to `True`, TensorFlow will be used.

train / dev The datasets. This can be a filename to a hdf-file. Or it can be a dict with an entry `class` where you can choose a from a variety of other dataset implementations, including many synthetic generated data.

num_inputs / num_outputs Defines the source/target dimensions of the data. Both can be integers. num_outputs can also be a dict if your dataset has other data streams. The standard source data is called "`data`" by default, and the standard target data is called "`classes`" by default. You can also specify whether your data is dense or sparse (i.e. it is just the index), which is specified by the number of dimensions, i.e. 2 (time-dim + feature-dim) or 1 (just time-dim).

Example: `num_outputs = {"data": [100, 2], "classes": [5000, 1]}`. This defines an input dimension of 100, and the input is dense (2), and an output dimension of 5000, and the output provided by the dataset is sparse (1). If "`classes`" is provided by `num_outputs`, then you can omit `num_inputs`.

batching The sorting variant when the mini-batches are created. E.g. `random`.

batch_size The total number of frames. A mini-batch has at least a time-dimension and a batch-dimension (or sequence-dimension), and depending on dense or sparse, also a feature-dimension. `batch_size` is the upper limit for `time * sequences` during creation of the mini-batches.

max_seqs The maximum number of sequences in one mini-batch.

chunking You can chunk sequences of your data into parts, which will greatly reduce the amount of needed zero-padding. This option is a string of two numbers, separated by a comma, i.e. `chunk_size:chunk_step`, where `chunk_size` is the size of a chunk, and `chunk_step` is the step after which we create the next chunk. I.e. the chunks will overlap by `chunk_size - chunk_step` frames. Set this to `0` to disable it, or for example `100:75` to enable it.

network This is a dict which defines the network topology. It consists of layer-names as strings, mapped on dicts, which defines the layers. The layer dict consists of keys as strings and the value type depends on the key. The layer dict should contain the key `class` which defines the class or type of the layer, such as `hidden` for a feed-forward layer, `rec` for a recurrent layer (including LSTM) or `softmax` for the output layer (doesn't need to have the softmax activation). Usually it also contains the key `n_out` which defines the feature-dimension of the output of this layer, and the key `from` which defines the inputs to this layer, which is a list of other layers. If you omit `from`, it will automatically pass in the input data from the dataset. All layer dict keys are passed to the layer class `__init__`, so you have to refer to the code for all details.

Example of a 3 layer bidirectional LSTM:

```
network = {
"lstm0_fw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2
↪": 0.01, "direction": 1 },
"lstm0_bw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2
↪": 0.01, "direction": -1 },

"lstm1_fw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2
↪": 0.01, "direction": 1, "from" : ["lstm0_fw", "lstm0_bw"] },
"lstm1_bw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2
↪": 0.01, "direction": -1, "from" : ["lstm0_fw", "lstm0_bw"] },

"lstm2_fw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2
↪": 0.01, "direction": 1, "from" : ["lstm1_fw", "lstm1_bw"] },
"lstm2_bw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2
↪": 0.01, "direction": -1, "from" : ["lstm1_fw", "lstm1_bw"] },

"output" :   { "class" : "softmax", "loss" : "ce", "from" : ["lstm2_fw", "lstm2_bw
↪"] }
}
```

See *API* or the code itself for documentation of the arguments for each layer class type. The `rec` layer class in particular supports a wide range of arguments, and several units which can be used, e.g. you can choose between

different LSTM implementations, or GRU, or standard RNN, etc. See *TFNetworkRecLayer.RecLayer* or *NetworkRecurrentLayer.RecurrentUnitLayer*. See also *TensorFlow LSTM benchmark*.

learning_rate The learning rate during training, e.g. `0.01`.

adam / nadam / ... E.g. set `adam = True` to enable the Adam optimization during training. See in *Updater.py* for many more.

model Defines the model file where RETURNN will save all model params after an epoch of training. For each epoch, it will suffix the filename by the epoch number.

num_epochs The number of epochs to train.

log_verbosity An integer. Common values are 3 or 4. Starting with 4, you will get an output per mini-batch.

There are much more params, and more details to many of the listed ones. See the code for more details. All config params can also be passed as command line params. See the code for some usage. The generic form is `++param value`.

See *Technological overview* for more details and an overview how it all works.

Technological overview

RETURNN is mostly used as a tool where `rnn.py` (*rnn*) is the main entry point but you can also use it as a framework / Python module to use in your own Python code. To get an idea about how it works, it helps to follow roughly the execution path starting in *rnn*, esp. in *rnn.main()*. In all cases, the code itself should be checked for details and comments.

So far, there are two calculation backends: Theano and TensorFlow, where Theano was the first backend, thus Theano-specific code files are currently not prefix but TensorFlow specific files are prefixed with `TF`. This is implemented separately for both Theano and TensorFlow:

- The engine for high-level logic, although a bit is shared. *Engine*, *EngineTask* for Theano and *TFEngine* for TensorFlow.
- Network topology construction which constructs the computation graph for training or just forwarding. *Network*, *TFNetwork*.
- Network model update code for training, i.e. SGD etc. *Updater*, *TFUpdater*.
- All the individual layer implementations. *NetworkLayer*, *NetworkBaseLayer*, *NetworkHiddenLayer*, *NetworkRecurrentLayer* etc for Theano and *TFNetworkLayer*, *TFNetworkRecLayer* for TensorFlow. This also means that Theano and TensorFlow don't support the same layers and even parameters can be different.
- Some utilities *TheanoUtil* and *TFUtil*.
- Multi-GPU logic. *Device*, *EngineTask* for Theano and not yet implemented for TensorFlow.

All the rest is shared for all backends, which mostly is:

- The main entry point *rnn*.
- Config handling *Config*.
- Logging *Log*.
- Utilities *Util*.
- Dataset reading *Dataset* including all the different dataset implementations *HDFDataset*, *SprintDataset*, *LmDataset*, *GeneratingDataset*, *MetaDataset*, etc.
- Learning rate scheduling logic such as Newbob *LearningRateControl*.

- Pretrain network structure construction *Pretrain*.
- The native op code which generates code for ops for both CUDA and CPU shares a common base. *NativeOp*, where TensorFlow-specific code is in *TFNativeOp*.

Execution guide

- *rnn.main()* will parse command line arguments and read in a config.
- Then logging *Log* is initialized, based on verbosity and other settings.
- Then it initializes the datasets (`train`, `dev`, `eval` in config), i.e. *Dataset* instances.
- Theano-only: *Device* instances.
- The engine, i.e. a *Engine* or *TFEngine* instance.
- Depending on the `task` option, some engine initialization which also initializes the network computation graph, *Network structure construction*.
- Then, depending on the `task` option, it might start `engine.train`, `engine.forward` etc. (*Engine.Engine.train()* or *TFEngine.Engine.train()*), *Training*.

Network structure construction

The network structure which defines the model topology is defined by the config `network` option, which is a dict, where each entry is a layer specification, which itself is a dict containing the kwargs for the specific layer class. E.g.:

```
network = {
    "fw1": {"class": "linear", "activation": "relu", "dropout": 0.1, "n_out": 500},
    "fw2": {"class": "linear", "activation": "relu", "dropout": 0.1, "n_out": 500,
↪"from": ["fw1"]},
    "output": {"class": "softmax", "loss": "ce", "from": ["fw2"]}
}
```

The `"class"` key will get extracted from the layer arguments and the specific layer class will be used. For Theano, the base layer class is *NetworkBaseLayer.Container* and *NetworkBaseLayer.Layer*; for TensorFlow, it is *TFNetworkLayer.LayerBase*. E.g. that would use the *TFNetworkLayer.LinearLayer* class, and the `LinearLayer.__init__` will accepts arguments like `activation`. In the given example, all the remaining arguments will get handled by the base layer.

The construction itself can be found for TensorFlow in *TFNetwork.TFNetwork.construct_from_dict()*, which starts from the output layers goes over the sources of a layer, which are defined by `"from"`. If a layer does not define `"from"`, it will automatically get the input from the dataset data.

Here is a 2 layer unidirectional LSTM network:

```
network = {
    "lstm1": {"class": "rec", "unit": "lstm", "dropout": 0.1, "n_out": 500},
    "lstm2": {"class": "rec", "unit": "lstm", "dropout": 0.1, "n_out": 500, "from": [
↪"lstm1"]},
    "output": {"class": "softmax", "loss": "ce", "from": ["lstm2"]}
}
```

In TensorFlow, that would use the layer class *TFNetworkRecLayer.RecLayer* which will handle the argument `unit`.

And here is a 3 layer bidirectional LSTM network:

```
network = {
"lstm0_fw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2": 0.
↪01, "direction": 1 },
"lstm0_bw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2": 0.
↪01, "direction": -1 },

"lstm1_fw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2": 0.
↪01, "direction": 1, "from" : ["lstm0_fw", "lstm0_bw"] },
"lstm1_bw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2": 0.
↪01, "direction": -1, "from" : ["lstm0_fw", "lstm0_bw"] },

"lstm2_fw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2": 0.
↪01, "direction": 1, "from" : ["lstm1_fw", "lstm1_bw"] },
"lstm2_bw" : { "class": "rec", "unit": "lstm", "n_out" : 500, "dropout": 0.1, "L2": 0.
↪01, "direction": -1, "from" : ["lstm1_fw", "lstm1_bw"] },

"output" :   { "class" : "softmax", "loss" : "ce", "from" : ["lstm2_fw", "lstm2_bw"] }
}
```

Training

The engine will loop over the epochs and the individual batches / steps and loads and saves the model. The specific implementation is different in Theano and TensorFlow. See the code for more details, i.e. *Engine*, *EngineTask* for Theano and *TFEngine* for TensorFlow.

TensorFlow LSTM benchmark

There are multiple LSTM implementations/kernels available in TensorFlow, and we also have our own kernel. In this benchmark, we try to compare the runtime performance during training for each of the kernels. We try to measure in a way that it should be generic and not be specific for our Returnn framework. You can run this benchmark yourself with this script.

In Returnn with the TensorFlow backend, the `rec` layer (*TFNetworkRecLayer.RecLayer*) you can use these LSTM kernels via the `unit` argument:

- **`BasicLSTM` (GPU and CPU).** Uses tf.contrib.rnn.BasicLSTMCell via dynamic_rnn. I.e. the cell itself is pure TensorFlow, and the loop over time is done via `tf.while_loop`.
- **`StandardLSTM` (GPU and CPU).** Uses tf.contrib.rnn.LSTMCell via dynamic_rnn. I.e. the cell itself is pure TensorFlow, and the loop over time is done via `tf.while_loop`. This has some more options compared to `BasicLSTM`.
- **`LSTMBlock` (GPU and CPU).** Uses tf.contrib.rnn.LSTMBlockCell via dynamic_rnn. The time-step operation is implemented as a single TF operation, and the loop over time is done via `tf.while_loop`. Thus this should be faster than `BasicLSTM` and `StandardLSTM`.
- **`LSTMBlockFused` (GPU and CPU).** Uses tf.contrib.rnn.LSTMBlockFusedCell. The loop over time is part of the op ("fused" in TF terminology), thus this is like `NativeLSTM` and `CudnnLSTM` a single op for the whole calculation. This is based on `LSTMBlock` and should thus be faster than `LSTMBlock`.
- **`CudnnLSTM` (GPU only).** Uses the LSTM kernel from cuDNN, via tf.contrib.cudnn_rnn.CudnnLSTM. The loop over time is done internally ("fused"). If you import such a model on CPU, it will automatically convert it into a `LSTMBlockFused`.

- **`NativeLSTM` (GPU and CPU).** Uses our own CUDA kernel which can also be compiled on CPU. The loop over time is also done via C++ code inside the op ("fused"). See *TFNativeOp*.

If you just use `LSTM`, it will currently use `NativeLSTM` by default. Except of `NativeLSTM`, all of these kernels are part of the official TensorFlow framework. Note that these kernels are always use for a single direction in time and a single layer.

The cuDNN LSTM kernel can also work bidirectional and do multiple layers at once but `tf.contrib.cudnn_rnn.CudnnLSTM` currently does not support batches with sequences of different length, thus this is normally not an option to use. Note that most frameworks with cuDNN bindings do not support this correctly (see here), where CNTK is currently the only exception. In TensorFlow, this is issue 6633. Note that you still can use the cuDNN kernel in the way we do in Returnn, i.e. for a single layer in one time-direction.

For the benchmark, we build a multi-layer bidirectional network. Example of a 3 layer bidirectional LSTM:

```
network = {
"lstm1_fwd" : { "class": "rec", "unit": "lstm", "n_out" : 500, "direction": 1 },
"lstm1_bwd" : { "class": "rec", "unit": "lstm", "n_out" : 500, "direction": -1 },

"lstm2_fwd" : { "class": "rec", "unit": "lstm", "n_out" : 500, "direction": 1, "from"␣
↪: ["lstm1_fwd", "lstm1_bwd"] },
"lstm2_bwd" : { "class": "rec", "unit": "lstm", "n_out" : 500, "direction": -1, "from
↪" : ["lstm1_fwd", "lstm1_bwd"] },

"lstm3_fwd" : { "class": "rec", "unit": "lstm", "n_out" : 500, "direction": 1, "from"␣
↪: ["lstm2_fwd", "lstm2_bwd"] },
"lstm3_bwd" : { "class": "rec", "unit": "lstm", "n_out" : 500, "direction": -1, "from
↪" : ["lstm2_fwd", "lstm2_bwd"] },

"output" :   { "class" : "softmax", "loss" : "ce", "from" : ["lstm3_fwd", "lstm3_bwd
↪"] }
}
```

We use framewise cross entropy as a loss for training, and we use a very simple artificial dataset (*GeneratingDataset.Task12AXDataset*) with dense input with a very low number of dimensions (9) and single output class indices (sparse) with a very low number of class labels (2), so that the overhead of the final softmax layer should be minimal, as well as the whole input pipeline. We are not interested in the error performance on this task in this benchmark, as in theory the results should all be the same. In practice, they are not due to different implementations, and also the initialization is currently not the same in all cases. However, that has no effect on the runtime performance.

By default, we use chunking, i.e. not the full sequences but only slices of it of fixed size (50 frames), to reduce the amount of padding in a mini-batch and also to keep the maximum sequence length of a batch fixed, and also to be able to increase the amount of sequences in a batch to allow more parallelism (40 sequences). See our paper for more details about chunking. Thus, our mini-batch has in total 2000 frames.

Comparison

For a 5 layer bidirectional LSTM with dimension 500 in each time direction, on a GeForce GTX 980, using 8 CPU threads, we got these results:

```
GPU:CudnnLSTM: 0:00:08.8151
GPU:NativeLSTM: 0:00:08.8440
GPU:LSTMBlockFused: 0:00:16.9765
GPU:LSTMBlock: 0:00:33.4895
GPU:StandardLSTM: 0:00:39.5170
GPU:BasicLSTM: 0:00:41.7282
```

```
CPU:NativeLSTM: 0:04:05.4365
CPU:LSTMBlockFused: 0:04:35.1702
CPU:StandardLSTM: 0:04:57.7977
CPU:BasicLSTM: 0:05:00.5334
CPU:LSTMBlock: 0:05:07.5613
```

On a GeForce GTX 1080 Ti, using 8 CPU threads, for the same experiment we got:

```
GPU:NativeLSTM: 0:00:05.2728
GPU:CudnnLSTM: 0:00:05.3645
GPU:LSTMBlockFused: 0:00:09.3915
GPU:LSTMBlock: 0:00:15.3071
GPU:StandardLSTM: 0:00:17.8279
GPU:BasicLSTM: 0:00:22.3976
CPU:NativeLSTM: 0:05:09.6268
CPU:LSTMBlockFused: 0:07:45.5984
CPU:StandardLSTM: 0:08:02.5465
CPU:BasicLSTM: 0:08:16.3543
CPU:LSTMBlock: 0:08:18.1589
```

And on a GeForce GTX 1070, with 4 CPU threads, we got:

```
GPU:NativeLSTM: 0:00:03.9989
GPU:CudnnLSTM: 0:00:05.4496
GPU:LSTMBlockFused: 0:00:07.5233
GPU:LSTMBlock: 0:00:11.1515
GPU:StandardLSTM: 0:00:12.0605
GPU:BasicLSTM: 0:00:12.0833
CPU:LSTMBlockFused: 0:02:53.6482
CPU:BasicLSTM: 0:03:00.8289
CPU:StandardLSTM: 0:03:01.6320
CPU:LSTMBlock: 0:03:04.8836
CPU:NativeLSTM: 0:03:18.5375
```

On a CPU-only system with a single CPU thread, we got:

```
CPU:NativeLSTM: 0:15:55.7625
CPU:LSTMBlockFused: 0:24:53.1451
CPU:BasicLSTM: 0:26:28.2804
CPU:StandardLSTM: 0:27:10.0493
CPU:LSTMBlock: 0:27:58.8870
```

Each of those are executed on different hardware, so there might be small other differences due to that. Also the number of available CPU threads differs. Each of those were run on Ubuntu 16.04 with TensorFlow 1.2 (installed via `pip`), CUDA 8.0 and cuDNN 5.1.

Analysis and discussion

We are quite proud that our own LSTM kernel (`NativeLSTM`) has a similar runtime than the cuDNN LSTM kernel (`CudnnLSTM`), sometimes even better. The implementation of it is quite straight-forward.

As expected, on GPU, both `NativeLSTM` and `CudnnLSTM` are faster than `LSTMBlockFused` (sometimes twice as fast).

Also as expected, on GPU, `LSTMBlockFused` is faster than `LSTMBlock` (up to 50%).

On GPU, `LSTMBlock` seems slightly faster than `BasicLSTM`/`StandardLSTM` but the difference is not so big.

Interestingly, on all experiments, on GPU, `StandardLSTM` seems to be slightly faster than `BasicLSTM`, which is not expected, as the `BasicLSTM` is simpler and also recommended by TensorFlow if you don't need the extended options which are available for `StandardLSTM`.

On CPU, it again looks different, and not as clear. This depends also on how much CPU threads will be used, and on the hardware. For example, `NativeLSTM` is currently not well optimized to use multiple threads (intra op parallelism). See also *`TFUtil.setup_tf_thread_pools()`* about intra and inter op parallelism.

We see that with a very low number of threads, on CPU, `NativeLSTM` can be the fastest, but not necessarily. Increasing the number of threads, `NativeLSTM` can become the slowest.

On CPU, `LSTMBlockFused` seems to be the fastest despite `NativeLSTM`, no matter the number of threads.

On CPU, interestingly, `BasicLSTM` and `StandardLSTM` seem to be slightly faster than `LSTMBlock`.

CHAPTER 2

API Reference

API

The main entry point is in *rnn*. This will initialize an instance of *Engine* which has the high level logic to iterate through epochs.

For each GPU (or CPU), an instance of *Device* will be created which does all the calculation and the model update.

The model is an instance of *Network.LayerNetwork* and consists of multiple hidden layers of class *NetworkBaseLayer.Layer* and one or more output layers of class *NetworkOutputLayer.OutputLayer*.

The model update code, i.e. the optimization methods such as SGD are implemented in *Updater*.

ActivationFunctions

ActivationFunctions.**relu**(*z*)

ActivationFunctions.**clipped01lu**(*z*)
0 for x <= 0 x for 0 <= x <= 1 1 for 1 <= x

ActivationFunctions.**clippedlu**(*z*)

-1 for x <= -1 x for -1 <= x <= 1 1 for 1 <= x

ActivationFunctions.**elu**(*z*)

ActivationFunctions.**identity**(*z*)

ActivationFunctions.**softsign**(*z*)

ActivationFunctions.**softsquare**(*z*)

ActivationFunctions.**maxout**(*z*)

ActivationFunctions.**softmax**(*z*)

ActivationFunctions.**gauss**(*z*)

ActivationFunctions.**cdf**(*z*)
Cumulative distribution function via erf (Error function)

ActivationFunctions.**constant_one**()

ActivationFunctions.**constant_zero**()

class ActivationFunctions.**Round3**(*output_types_preference=None*, *name=None*)

c_code(*node*, *name*, *x*, *z*, *sub*)

grad(*inputs*, *gout*)

ActivationFunctions.**hard_sigmoid**(*x*)

ActivationFunctions.**binary_tanh**(*x*)

ActivationFunctions.**binary_sigmoid**(*x*)

ActivationFunctions.**strtoact**(*act*)

Parameters **act** (*str | list[str]*) – activation function name, or multiple such as a list or separated by ":"

Return type theano.Op | list[theano.Op]

ActivationFunctions.**strtoact_single_joined**(*act*)

Parameters **act** (*str | None*) – activation function name, or multiple such as a list or separated by ":"

Return type theano.Op

BestPathDecoder

class BestPathDecoder.**BestPathDecodeOp**

make_node(*x*, *y*, *seq_lengths*)

c_code(*node*, *name*, *inp*, *out*, *sub*)

c_compile_args()

c_code_cache_version()

c_support_code()

BundleFile

class BundleFile.**BundleFile**(*filePath*)
Holds paths to HDF dataset files.

Reads paths to HDF dataset files from a bundle file. Example of contents of a bundle file:

/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_real_1_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_real_2_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_real_3_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_real_4_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_real_5_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_real_6_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_simu_1_100.hdf

/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_simu_2_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_simu_3_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_simu_4_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_simu_5_100.hdf
/work/asr2/ryndin/crnnRegressionSpeechEnhancemenent/data/data_tr05_simu_6_100.hdf

Parameters **filePath** (*str*) – path to a bundle file which contains paths to HDF dataset files. One path per line.

datasetFilePaths
Paths to HDF dataset files.

Return type list of str

Returns Paths to HDF dataset files.

numberOfDatasetFiles
Number of HDF dataset files.

Return type *int*

Returns Number of HDF dataset files.

CTC

class CTC.**CTCOp**

make_node(*x*, *y*, *seq_lengths*)

c_support_code()

c_compile_args()

c_code(*node*, *name*, *inp*, *out*, *sub*)

c_code_cache_version()

CachedDataset

class CachedDataset.**CachedDataset**(*cache_byte_size=0*, ***kwargs*)

initialize()

init_seq_order(*epoch=None*, *seq_list=None*)

Parameters **| None seq_list** (list *[str]*) – In case we want to set a predefined order.

Initialize lists: self.seq_index # sorted seq idx

batch_set_generator_cache_whole_epoch()

load_seqs(*start*, *end*, *with_cache=True*)
Load data sequences. As a side effect, will modify / fill-up:

self.alloc_intervals self.targets

This does some extra logic for the cache and calls self._load_seqs() for the real loading.

Parameters

- **start** (int) – start sorted seq idx
- **end** (int) – end sorted seq idx
- **with_cache** (bool) – handle cache

alloc_interval_index(*ids*)

Parameters **ids** (int) – sorted seq idx

:return index in self.alloc_intervals :rtype: int

insert_alloc_interval(*start*, *end=None*)

remove_alloc_interval(*start*, *end=None*)

delete(*nframes*)

Parameters **nframes** (*int|None*) – how much frames to delete max. Note that this limit is not strict. We can end up deleting more than nframes.

Returns number of frames deleted

Return type *int*

num_seqs

is_cached(*start*, *end*)

Parameters

- **start** (int) – like in load_seqs(), sorted seq idx
- **end** (int) – like in load_seqs(), sorted seq idx

Return type *bool*

:returns whether we have the full range (start,end) of sorted seq idx cached in self.alloc_intervals (end is exclusive).

get_seq_length_2d(*sorted_seq_idx*)

Return type (*int*,*int*)

get_seq_length(*seq_idx*)

Return type *NumbersDict*

get_seq_start(*sorted_seq_idx*)

Return type (*int*,*int*)

get_times(*sorted_seq_idx*)

get_input_data(*sorted_seq_idx*)

get_data_dim(*key*)

get_targets(*target*, *sorted_seq_idx*)

get_target_list()

get_ctc_targets(*sorted_seq_idx*)

has_ctc_targets()

get_tag(*sorted_seq_idx*)

CachedDataset2

class CachedDataset2.**CachedDataset2**(***kwargs*)

Somewhat like CachedDataset, but different. Simpler in some sense. And more generic. Caching might be worse.

If you derive from this class: - you must override *_collect_single_seq* - you must set *num_inputs* (dense-dim of "data" key) and *num_outputs* (dict key -> dim, ndim-1) - you should set *labels* - handle seq ordering by overriding *init_seq_order* - you can set *_estimated_num_seqs* - you can set *_num_seqs* or *_num_timesteps* if you know them in advance

init_seq_order(*epoch=None*, *seq_list=None*)

Parameters | **None** **seq_list** (list *[str]*) – In case we want to set a predefined order.

This is called when we start a new epoch, or at initialization. Call this when you reset the seq list.

is_cached(*start*, *end*)

num_seqs

is_less_than_num_seqs(*n*)

get_num_timesteps()

get_seq_length(*sorted_seq_idx*)

Return type *int*

get_input_data(*sorted_seq_idx*)

get_targets(*target*, *sorted_seq_idx*)

get_ctc_targets(*sorted_seq_idx*)

get_tag(*sorted_seq_idx*)

get_target_list()

is_data_sparse(*key*)

Return type *bool*

get_data_dim(*key*)

Return type *int*

Returns number of classes, no matter if sparse or not

get_data_dtype(*key*)

Config

class Config.**Config**

load_file(*f*)

Reads the configuration parameters from a file and adds them to the inner set of parameters :type f: string

add_line(*key*, *value*)

Adds one specific configuration (key,value) pair to the inner set of parameters :type key: str :type value: str

has(*key*)
Returns whether the given key is present in the inner set of parameters :type key: string :rtype: boolean :returns True if and only if the given key is in the inner set of parameters

is_typed(*key*)

Return type boolean

:returns True if and only if the value of the given key has a specified data type

is_true(*key*, *default=False*)

Parameters

- **key** (*str*) –
- **default** (*bool*) –

Returns bool(value) if it is set or default

Return type *bool*

is_of_type(*key*, *types*)

Parameters

- **key** (*str*) –
- **types** (*type | tuple[type]*) – for isinstance() check

Returns whether is_typed(key) is True and isinstance(value, types) is True

Return type *bool*

get_of_type(*key*, *types*, *default=None*)

Parameters

- **key** (*str*) –
- **types** (*type | list[type] | T*) – for isinstance() check
- **default** (*T | None*) –

Returns if is_of_type(key, types) is True, returns the value, otherwise default

Return type T

set(*key*, *value*)

update(*dikt*)

value(*key*, *default*, *index=None*, *list_join_str=', '*)

Return type str | T

typed_value(*key*, *default=None*, *index=None*)

Return type str | T

int(*key*, *default*, *index=0*)
Parses the value of the given key as integer, returning default if not existent :type key: str :type default: T :type index: int :rtype: int | T

bool(*key*, *default*, *index=0*)
Parses the value of the given key as boolean, returning default if not existent :type key: str :type default: T :type index: bool :rtype: bool | T

bool_or_other(*key*, *default*, *index=0*)

float(*key*, *default*, *index=0*)
Parses the value of the given key as float, returning default if not existent :type key: str :type default: T :type index: int :rtype: float | T

list(*key*, *default=None*)

Return type *list*[str] | T

int_list(*key*, *default=None*)

Return type *list*[*int*] | T

float_list(*key*, *default=None*)

Return type *list*[*float*] | T

int_pair(*key*, *default=None*)

Config.**get_global_config**()

Return type *Config*

CustomLSTMFunctions

CustomLSTMFunctions.**debug_make_theano_function_wrapper**(*f*, *name*, *hook*, *other_values*)

CustomLSTMFunctions.**make_bwd_fun**(*recurrent_transform*)

CustomLSTMFunctions.**make_fwd_fun**(*recurrent_transform*)

CustomLSTMFunctions.**print_wt**(*op*, *x*)

CustomLSTMFunctions.**setup_parent_functions**(*fn*, *recurrent_transform_id*)

Dataset

class Dataset.**Dataset**(*name='dataset'*, *window=1*, *context_window=None*, *chunking='0'*, *seq_ordering='default'*, *shuffle_frames_of_nseqs=0*, *estimated_num_seqs=None*)

Parameters

- **name** (*str*) – e.g. "train" or "eval"
- **window** (*int*) – features will be of dimension window * feature_dim, as we add a context-window around. not all datasets support this option.
- **context_window** (*None* | *int* | *dict* | *NumbersDict*) – will add this context for each chunk
- **chunking** (*str*) – "chunk_size:chunk_step"
- **seq_ordering** (*str*) – "batching"-option in config. e.g. "default", "sorted" or "random". See self.get_seq_order_for_epoch() for more details.
- **shuffle_frames_of_nseqs** (*int*) – shuffles the frames. not always supported

- **estimated_num_seqs** (*None | int*) – for progress reporting in case the real num_seqs is unknown

batch_set_generator_cache_whole_epoch()

The BatchSetGenerator can cache the list of batches which we generated across epochs. See self.generate_batches() and self._generate_batches(). In many cases, the dataset does not support this, and in that case, it is not needed to enable this cache and waste memory. Caching it together with option shuffle_batches could also mean that there will be self.load_seqs() calls with non-monotonic seq-idxs. The only dataset currently which enables this is CachedDataset and thus HDFDataset.

Returns whether we should enable this cache

Return type *bool*

calculate_priori(*target='classes'*)

estimated_num_seqs

classmethod **from_config**(*config*, ***kwargs*)

Parameters **kwargs** (*dict[str]*) – passed on to __init__

Return type *Dataset*

generate_batches(*recurrent_net*, *batch_size*, *max_seqs=-1*, *seq_drop=0.0*, *max_seq_length=9223372036854775807*, *shuffle_batches=False*, *used_data_keys=None*)

Parameters **used_data_keys** (set *(str)* | *None*) –

Return type *BatchSetGenerator*

classmethod **generic_complete_frac**(*seq_idx*, *num_seqs*)

Parameters

- **seq_idx** (*int*) – idx
- **num_seqs** (*int | None*) – None if not available

Returns Returns a fraction (float in [0,1], always > 0) of how far we have advanced for this seq in the dataset. This does not have to be exact. This is only for the user.

get_complete_frac(*seq_idx*)

Returns Returns a fraction (float in [0,1], always > 0) of how far we have advanced for this seq in the dataset. This does not have to be exact. This is only for the user.

get_ctc_targets(*sorted_seq_idx*)

get_data(*seq_idx*, *key*)

Parameters

- **seq_idx** (*int*) – sorted seq idx
- **key** (*str*) – data-key, e.g. "data" or "classes"

Return type numpy.ndarray

Returns features or targets format 2d (time,feature) (float)

get_data_dim(*key*)

Returns number of classes, no matter if sparse or not

get_data_dtype(*key*)

get_data_keys()

get_data_shape(*key*)
:returns get_data(*, key).shape[1:], i.e. num-frames excluded

get_data_slice(*seq_idx*, *key*, *start_frame*, *end_frame*)

get_input_data(*sorted_seq_idx*)

Return type numpy.ndarray

Returns features format 2d (time,feature) (float)

get_max_ctc_length()

get_num_codesteps()

get_num_timesteps()

get_seq_length(*seq_idx*)

Return type *NumbersDict*

get_seq_length_2d(*sorted_seq_idx*)

Return type numpy.array[*int*,*int*]

:returns the len of the input features and the len of the target sequence. For multiple target seqs, it is expected that they all have the same len. We support different input/target len for seq2seq/ctc and other models. Note: This is deprecated, better use get_seq_length().

get_seq_order_for_epoch(*epoch*, *num_seqs*, *get_seq_len=None*)
:returns the order for the given epoch. This is mostly a static method, except that is depends on the configured type of ordering,

such as 'default' (= as-is), 'sorted' or 'random'. 'sorted' also uses the sequence length.

Parameters

- **epoch** (`int`) – for 'random', this determines the random seed
- **get_seq_len** – function (originalSeqIdx: int) -> int

Return type *list*[*int*]

get_start_end_frames_full_seq(*seq_idx*)

Parameters **seq_idx** (`int`) –

Returns (start,end) frame, taking context_window into account

Return type (*NumbersDict*,*NumbersDict*)

get_tag(*sorted_seq_idx*)

get_target_list()

get_targets(*target*, *sorted_seq_idx*)

Return type numpy.ndarray

Returns targets format 1d (time) (int: idx of output-feature)

get_times(*sorted_seq_idx*)

has_ctc_targets()

have_seqs()

classmethod index_shape_for_batches(*batches*, *data_key='data'*)

init_seq_order(*epoch=None*, *seq_list=None*)

Parameters | **None** **seq_list** (list[*str*]) – In case we want to set a predefined order.

Return type *bool*

:returns whether the order changed

This is called when we start a new epoch, or at initialization. Call this when you reset the seq list.

initialize()
Does the main initialization before it can be used. This needs to be called before self.load_seqs() can be used.

is_cached(*start*, *end*)

Parameters

- **start** (int) – like in load_seqs(), sorted seq idx
- **end** (int) – like in load_seqs(), sorted seq idx

Return type *bool*

:returns whether we have the full range (start,end) of sorted seq idx.

is_data_sparse(*key*)

is_less_than_num_seqs(*n*)

Return type *bool*

:returns whether n < num_seqs. In case num_seqs is not known in advance, it will wait until it knows that n is behind the end or that we have the seq.

iterate_seqs(*chunk_size=None*, *chunk_step=None*, *used_data_keys=None*)
Takes chunking into consideration. :param int chunk_size: :param int chunk_step: :param set(str)|None used_data_keys: :return: generator which yields tuples (seq index, seq start, seq end) :rtype: list[(int,NumbersDict,NumbersDict)]

static kwargs_update_from_config(*config*, *kwargs*)

len_info()

Return type str

:returns a string to present the user as information about our len. Depending on our implementation, we can give some more or some less information.

load_seqs(*start*, *end*)
Load data sequences, such that self.get_data() & friends can return the data. :param int start: start sorted seq idx, inclusive :param int end: end sorted seq idx, exclusive

num_seqs

preprocess(*seq*)

Return type numpy.ndarray

sliding_window(*xr*)

Return type numpy.ndarray

class Dataset.**DatasetSeq**(*seq_idx*, *features*, *targets*, *ctc_targets=None*, *seq_tag=None*)

Parameters

- **seq_idx** (*int*) – sorted seq idx in the Dataset
- **features** (*numpy.ndarray*) – format 2d (time,feature) (float)
- **| numpy.ndarray | None targets** (*dict[str,numpy.ndarray]*) – name -> format 1d (time) (idx of output-feature)
- **| None ctc_targets** (*numpy.ndarray*) – format 1d (time) (idx of output-feature)
- **seq_tag** (*str*) – sequence name / tag

get_data(*key*)

get_data_keys()

num_frames

Return type *NumbersDict*

Dataset.**convert_data_dims**(*data_dims*, *leave_dict_as_is=False*)

This converts what we called num_outputs originally, from the various formats which were allowed in the past (just an int, or dict[str,int]) into the format which we currently expect. :param int | dict[str,int|(int,int)|dict] data_dims: what we called num_outputs originally :param bool leave_dict_as_is: :rtype: dict[str,(int,int)|dict] :returns dict data-key -> (data-dimension, len(shape) (1 ==> sparse))

(or potentially data-key -> dict, if leave_dict_as_is is True; for TensorFlow)

Dataset.**get_dataset_class**(*name*)

Dataset.**init_dataset**(*kwargs*)

Return type *Dataset*

Dataset.**init_dataset_via_str**(*config_str*, *config=None*, *cache_byte_size=None*, ***kwargs*)

Parameters

- **config_str** (*str*) – hdf-files, or "LmDataset:..." or so
- **config** (*Config.Config|None*) – optional, only for "sprint:..."
- **cache_byte_size** (*int|None*) – optional, only for HDFDataset

Return type *Dataset*

Dataset.**random**() → x in the interval [0, 1).

Dataset.**shapes_for_batches**(*batches*, *data_keys*, *dataset=None*, *extern_data=None*)

Parameters

- **batches** (*list[EngineBatch.Batch]*) –
- **data_keys** (*list[str]*) –
- **dataset** (*Dataset*) –
- **extern_data** (*TFNetwork.ExternData*) – detailed data description. only used for TensorFlow

Return type dict[str,*list*[*int*]] | None

Debug

Debug.**auto_exclude_all_new_threads**(*func*)

Debug.**dumpAllThreadTracebacks**(*exclude_thread_ids=set([])*)

Debug.**setupWarnWithTraceback**()

Debug.**initBetterExchook**()

Debug.**format_signum**(*signum*)

Parameters **signum** (int) –

Returns string "signum (signame)"

Return type str

Debug.**signal_handler**(*signum*, *frame*)
Prints a message on stdout and dump all thread stacks.

Parameters

- **signum** (int) – e.g. signal.SIGUSR1
- **frame** – ignored, will dump all threads

Debug.**install_signal_handler_if_default**(*signum*, *exceptions_are_fatal=False*)

Parameters

- **signum** (int) – e.g. signal.SIGUSR1
- **exceptions_are_fatal** (bool) – if True, will reraise any exceptions. if False, will just print a message

Returns True iff no exception, False otherwise. not necessarily that we registered our own handler

Return type *bool*

Debug.**installNativeSignalHandler**()

Debug.**installLibSigSegfault**()

Debug.**initFaulthandler**(*sigusr1_chain=False*)
Maybe installs signal handlers, SIGUSR1 and SIGUSR2 and others. If no signals handlers are installed yet for SIGUSR1/2, we try to install our own Python handler. This also tries to install the handler from the fauldhandler module, esp for SIGSEGV and others.

Parameters **sigusr1_chain** (bool) – whether the default SIGUSR1 handler should also be called.

Debug.**initIPythonKernel**(**args*, ***kwargs*)

Debug.**initCudaNotInMainProcCheck**()

Debug.**debug_shell**(*user_ns=None*, *user_global_ns=None*, *exit_afterwards=True*)

DebugHelpers

This file is going to be imported by Debug.debug_shell() and available as interactive commands.

DebugHelpers.**find_obj_in_stack**(*cls*, *stack=None*, *all_threads=True*)

DebugHelpers.**get_device**()

Return type *Device.Device*

DebugHelpers.**compute**(*var*, *trainnet=True*)

Parameters

- **var** (*theano.Variable*) – variable which we should compute the value of

- **trainnet** (bool) – whether to make givens based on dev.trainnet or dev.testnet

Returns the computed value

Return type numpy.ndarray

This expects to calculate some value of the trainnet or testnet of the current Device.

class DebugHelpers.**DebugNn**(*filename*)

compile_forwarder()

forward(*data*, *output_index=0*)

class DebugHelpers.**SimpleHdf**(*filename*)

get_seq_tags()

get_data(*seq_idx*)

get_targets(*seq_idx*)

get_data_dict(*seq_idx*)

Device

class Device.**Device**(*device*, *config*, *blocking=False*, *num_batches=1*, *update_specs=None*)

Parameters

- **device** (*str*) – name, "gpu*" or "cpu*"
- **config** (Config.Config) – config
- **blocking** (bool) – False -> multiprocessing, otherwise its blocking
- **num_batches** (int) – num batches to train on this device

:param dict update_specs

alloc_data(*shapes*, *max_ctc_length=0*)

Parameters **shapes** (*dict[str,list[int]]*) – by data-key. format usually (time,batch,features)

clear_memory(*network*)

compute_run(*task*)

detect_nan(*i*, *node*, *fn*)

dump_model_broken_info(*info*)

fast_check_model_is_broken_from_result(*output*, *outputs_format*)

finish_epoch_stats()

forward(*use_trainnet=False*)

get_compute_func(*task*)

get_device_clock()

get_device_memory()

get_device_shaders()

get_memory_info()

get_net_train_params(*network*)

get_num_updates()

get_task_network()

Return type *LayerNetwork*

initialize(*config*, *update_specs=None*, *json_content=None*, *train_param_args=None*)

is_device_proc()

make_ce_ctc_givens(*network*)

make_ctc_givens(*network*)

make_givens(*network*)

make_input_givens(*network*)

static make_result_dict(*output*, *outputs_format*)

make_sprint_givens(*network*)

need_reinit(*json_content*, *train_param_args=None*)

prepare(*network*, *updater=None*, *train_param_args=None*, *epoch=None*)

Call this from the main proc before we do anything else. This is called before we start any training, e.g. at the begin of an epoch. :type network: LayerNetwork :type updater: Updater | None :type train_param_args: dict | None

process(*asyncTask*)

process_inner(*device*, *config*, *update_specs*, *asyncTask*)

reinit(*json_content*, *train_param_args=None*)

:returns len of train_params :rtype: int Reinits for a new network topology. This can take a while because the gradients have to be recomputed.

result()

Return type (*list*[numpy.ndarray], *list*[str] | None)

:returns the outputs and maybe a format describing the output list See self.make_result_dict() how to interpret this list. See self.initialize() where the list is defined.

run(*task*)

set_learning_rate(*learning_rate*)

set_net_encoded_params(*network_params*)

This updates *all* params, not just the train params.

set_net_params(*network*)

This updates *all* params, not just the train params.

startProc(**args*, ***kwargs*)

start_epoch_stats()

sync_net_train_params()

sync_used_targets()
Updates self.used_targets for the host.

terminate()

update_data()

update_memory()

Device.**getDevicesInitArgs**(*config*)

Return type *list*[dict[str]]

Device.**get_current_seq_index_mask**(*target*)

Device.**get_current_seq_tags**()

Returns current seq tags (seq names) of current batch. assumes is_device_host_proc()

Return type *list*[str]

Device.**get_device_attributes**()

Device.**get_gpu_names**()

Device.**get_num_devices**()

Device.**have_gpu**()

Device.**is_device_host_proc**()

Device.**is_using_gpu**()

Device.**sort_strint**(*txt*)

Device.**str2int**(*txt*)

Engine

class Engine.**Engine**(*devices*)

analyze(*data*, *statistics*)

Parameters

- **data** (Dataset.Dataset) –
- **statistics** (list[str]|None) –

Returns nothing, will print everything to log.v1

check_last_epoch()

classify(*data*, *output_file*)

compute_priors(*dataset*, *config*)

classmethod config_get_final_epoch(*config*)

daemon(*config*)

classmethod epoch_model_filename(*model_filename*, *epoch*, *is_pretrain*)

Return type str

eval_model()

format_score(*score*)

forward_to_hdf(*data*, *output_file*, *combine_labels=''*, *batch_size=0*)

classmethod get_epoch_model(*config*)

:returns (epoch, modelFilename) :rtype: (int|None, str|None)

get_epoch_model_filename()

get_epoch_str()

get_eval_datasets()

classmethod get_existing_models(*config*)

classmethod get_train_start_epoch_batch(*config*)
We will always automatically determine the best start (epoch,batch) tuple based on existing model files. This ensures that the files are present and enforces that there are no old outdated files which should be ignored. Note that epochs start at idx 1 and batches at idx 0. :type config: Config.Config :returns (epoch,batch) :rtype (int,int)

init_network_from_config(*config*)

init_train_epoch()

init_train_from_config(*config*, *train_data*, *dev_data=None*, *eval_data=None*)

is_first_epoch_after_pretrain()

is_pretrain_epoch()

classmethod model_filename_postfix()

network_dump_json(*json_filename*)

print_network_info()

save_model(*filename*, *epoch*)

Parameters

- **filename** (*str*) – full filename for model
- **epoch** (int) – save epoch idx

train()

train_epoch()

class Engine.**SeqTrainParallelControl**(*engine*, *config*, ***kwargs*)

Idea: Parallelize some stuff in seq training (e.g. sprint loss). Can use chunked training. We have these steps:

1.(forward:GPU) forward only, remember output

2.(calc_loss:CPU) calculate loss based on data from (1), error signal. store hat_y = y - grad_L (for stability).

3.(train:GPU) forward again, and backprop with data from (2).

1.and (3) are on the same GPU, use the same shared params.

2.is on CPU.

3.is done via the usual loop via EngineTask.TrainTaskThread.

It calls self.train_wait_for_seqs().

This class *SeqTrainParallelControl* is instantiated by the Engine and it has the these callbacks which are called by the engine (TrainTaskThread):

train_start_epoch() train_finish_epoch() train_wait_for_seqs()

Thus, this instance lives in the main proc and this code is executed in the main proc.

There is a counterpart of this code living in the device proc and we are calling it via Device.seq_train_parallel_control which is an instance of *SeqTrainParallelControlDevHost*. Most things are actually happening there.

forward_fill_queue()

Full sequence forwarding, no chunking (at the moment).

train_finish_epoch()

Called from TrainTaskThread at the end of an epoch.

train_start_epoch()

Called from TrainTaskThread at the beginning of a new epoch.

train_wait_for_seqs(*device*, *batches*)

Called from TrainTaskThread while doing training (forward + backprop). This will tell the device what set of batches we want to train next. :type device: Device.Device :type batches: list[EngineBatch.Batch]

EngineBatch

class EngineBatch.**BatchSeqCopyPart**(*seq_idx*, *seq_start_frame*, *seq_end_frame*, *batch_slice*, *batch_frame_offset*)

A batch used for training in CRNN can consist of several parts from sequences, ordered in various ways. The dataset, depending on the configuration, can generate these. For the non-recurrent case, we usually concatenate them together into one slice. For the recurrent case, we have a single slice per sequence, or even multiple slices for a sequence in case of chunking.

This class represents one single such part and where it is going to be stored in the batch.

frame_length

class EngineBatch.**Batch**

A batch can consists of several sequences (= segments). This is basically just a list of BatchSeqCopyPart.

try_sequence_as_slice(*length*)

Parameters **length** (NumbersDict) – number of (time) frames

Returns new shape which covers the old shape and one more data-batch, format (time,batch)

Return type (*NumbersDict*,*int*)

add_sequence_as_slice(*seq_idx*, *seq_start_frame*, *length*)

Adds one data-batch in an additional slice.

Parameters

- **seq_idx** (int) –
- **seq_start_frame** (NumbersDict) –
- **length** (NumbersDict) – number of (time) frames

add_frames(*seq_idx*, *seq_start_frame*, *length*, *frame_dim_corresponds=True*)

Adds frames to all data-batches. Will add one data-batch if we don't have one yet.

Parameters

- **seq_idx** (int) –
- **seq_start_frame** (*NumbersDict* | *int*) –
- **length** (NumbersDict) – number of (time) frames
- **frame_dim_corresponds** (bool) – if the batch frame offset should always be the same (max value) for all keys

init_with_one_full_sequence(*seq_idx*, *dataset*)

Parameters

- **seq_idx** (int) –
- **dataset** (Dataset.Dataset) –

get_all_slices_num_frames()

Note that this is only an upper limit in case of data_shape[1] > 1 because data_shape[0] is the max frame len of all seqs.

get_total_num_frames()

start_seq

end_seq

get_num_seqs()

class EngineBatch.**BatchSetGenerator**(*dataset*, *generator*, *shuffle_batches=True*, *cache_whole_epoch=True*)

This will give you the next batches (list[Batch]) such that you can use them for assign_dev_data(). We get those batches from a generator, i.e. lazily on-the-fly. This is the whole point of BatchSetGenerator - that we must not know the whole list of batches in advance. As assign_dev_data() can fail for various reasons, we buffer the list of batches and you call self.advance() explicitly to go forward to next batches.

reset()

Call this after one epoch to reuse the previously cached batches.

peek_next_n(*n*)

Return type *list*[*Batch*]

:returns it might return less. There is no way to know in advance. If self.has_more() is True, it will at least return one.

advance(*n*)

completed_frac()

Return type *float*

:returns 0-1, >0

has_more()

Return type *bool*

get_current_batch_idx()

Return type *int*

EngineTask

class EngineTask.**ClassificationTaskThread**(*network*, *devices*, *data*, *batches*)

evaluate(*batchess*, *results*, *result_format*, *num_frames*)

class EngineTask.**EvalTaskThread**(*network*, *devices*, *data*, *batches*, ***kwargs*)

initialize()

class EngineTask.**HDFForwardTaskThread**(*network*, *devices*, *data*, *batches*, *cache*, *compression='none'*)

evaluate(*batchess*, *results*, *result_format*, *num_frames*)

Parameters

- **batchess** (list[list[Batch]]) – batches per device
- **results** (list[list[numpy.ndarray]]) – results per device
- **result_format** (list[str]|None) – describes what we have in a result list

:returns some score or None :rtype: dict[str] | None

finalize()

initialize()

exception EngineTask.**ModelBrokenError**(*msg*, *batches*)
We got a nan/inf at the result somewhere. This means that something is broken.

class EngineTask.**PriorEstimationTaskThread**(*network*, *devices*, *data*, *batches*, *priori_file*, *target*, *extract_type*)

evaluate(*batchess*, *results*, *result_format*, *num_frames*)

finalize()

class EngineTask.**TaskThread**(*task*, *network*, *devices*, *data*, *batches*, *eval_batch_size=0*, *start_batch=0*, *share_batches=False*, *report_prefix=None*, *exclude=None*, *epoch=None*)

Parameters **report_prefix** (*str*) – such as epoch or so. only for reporting

class **DeviceBatchRun**(*parent*, *devices*)

allocate()

device_collect_results()

device_mem_usage_str(*devices*)

Return type str | None

device_run()

finish()

:returns whether everything is fine.

print_process()

run()

stop()

TaskThread.**allocate_devices**(*selected_devices=None*)

Sets the device data, i.e. the next batches, via self.batches. This calls Dataset.load_seqs() to get the data. This sets:

device.targets device.ctc_targets device.tags device.index

Return type *list*[*list*[*EngineBatch.Batch*]]

:returns list of batches per device

TaskThread.**assign_dev_data**(*device*, *batches*)

TaskThread.**epoch_norm_factor_for_result**(*key*)

TaskThread.**evaluate**(*batchess*, *results*, *result_format*, *num_frames*)

Parameters

- **batchess** (list[list[EngineBatch.Batch]]) – batches per device
- **results** (list[list[numpy.ndarray]]) – results per device
- **result_format** (list[str]|None) – describes what we have in a result list

:returns some score or None :rtype: dict[str] | None

TaskThread.**finalize**()

Called at the end of an epoch.

TaskThread.**get_device_prepare_args**()

TaskThread.**initialize**()

Called at the beginning of an epoch.

TaskThread.**maybe_wait_for_batches**(*device*, *batches*)

TaskThread.**prepare_device_for_batch**(*device*)

TaskThread.**reduce**(*num_frames*)

TaskThread.**run**()

TaskThread.**run_inner**()

class EngineTask.**TrainTaskThread**(*network*, *devices*, *data*, *batches*, *learning_rate*, *updater*, *seq_train_parallel=None*, ***kwargs*)

class CopyManager(*devices*)

class CopyThread(*device*, *network*, *copy_to_device*)

run()

TrainTaskThread.CopyManager.**copy_from_device**()

TrainTaskThread.CopyManager.**copy_to_device**(*network*)

TrainTaskThread.**finalize**()

TrainTaskThread.**get_device_prepare_args**()

TrainTaskThread.**initialize**()

TrainTaskThread.**maybe_wait_for_batches**(*device*, *batches*)

TrainTaskThread.**prepare_device_for_batch**(*device*)

TrainTaskThread.**reduce**(*num_frames*)

TrainTaskThread.**save_ctc_priors**(*filename*, *epoch_str*)

EngineUtil

EngineUtil.**assign_dev_data**(*device*, *dataset*, *batches*, *load_seqs=True*)

:returns successful and how much batch idx to advance. :rtype: (bool,int)

EngineUtil.**assign_dev_data_single_seq**(*device*, *dataset*, *seq*, *load_seqs=True*)

Parameters **seq** (int) – sorted seq idx

Returns whether we succeeded

Return type *bool*

EngineUtil.**maybe_subtract_priors**(*network*, *train*, *config*)

External

External.**norm_shape**(*shape*)
Normalize numpy array shapes so they're always expressed as a tuple, even for one-dimensional shapes.

Parameters shape - an int, or a tuple of ints

Returns a shape tuple

External.**sliding_window**(*a*, *ws*, *ss=None*, *flatten=True*)
Return a sliding window over a in any number of dimensions

Parameters: a - an n-dimensional numpy array ws - an int (a is 1D) or tuple (a is 2D or greater) representing the size

of each dimension of the window

ss - an int (a is 1D) or tuple (a is 2D or greater) representing the amount to slide the window in each dimension. If not specified, it defaults to ws.

flatten - if True, all slices are flattened, otherwise, there is an extra dimension for each dimension of the input.

Returns an array containing each n-dimensional window from a

Fsa

class Fsa.**Edge**(*source_state_idx*, *target_state_idx*, *label*, *weight=0.0*)
class to represent an edge

Parameters

- **source_state_idx** (*int*) – the starting node of the edge
- **target_state_idx** (*int*) – the ending node od th edge
- **label** (*int | str | None*) – the label of the edge (normally a letter or a phoneme ...)
- **weight** (*float*) – probability of the word/phon in -log space

weight = None
int|None idx_word_in_sentence: index of word in the given sentence int|None idx_phon_in_word: index of phon in a word int|None idx: label index within the sentence/word/phon bool phon_at_word_begin: flag indicates if phon at the beginning of a word bool phon_at_word_end: flag indicates if phon at the end of a word float|None score: score of the edge bool is_loop: is the edge a loop within the graph

class Fsa.**Graph**(*lemma*)
class holds the Graph representing the Finite State Automaton holds the input and the created output (ASG, CTC, HMM) states between input and output may be held if necessary

Parameters **lemma** (*str | None*) – a sentence or word

list[str] lemma_list: input transformed into list if necessary

set_filename(*name*)
sets the filename, for use with saving :param str name: the filename, different stuff gets appended

make_single_state_graph()

save()

class Fsa.**Asg**(*graph*, *num_labels*, *asg_repetition=2*, *label_conversion=False*)
class to create ASG FSA

Parameters

- **fsa** (Graph) – represents the Graph on which the class operates
- **num_labels** (int) – number of labels without blank, silence, eps and repetitions
- **asg_repetition** (int) – asg repeat symbol which stands for x repetitions
- **label_conversion** (bool) – shall the labels be converted into numbers (only ASG and CTC)

set_asg_rep(*reps*)
sets the asg repeat symbol :param int reps: the asg repeat

set_num_labels(*numlab*)
sets number of labels :param int numlab: the number of labels

set_label_conversion(*onoff*)
sets label conversion on or off :param bool onoff: flag to set label conversion on/off

run()

class Fsa.**Ctc**(*graph*, *num_labels*, *label_conversion=False*)
class to create CTC FSA

Parameters

- **fsa** (Graph) – represents the Graph on which the class operates
- **num_labels** (int) – number of labels without blank, silence, eps and repetitions
- **label_conversion** (bool) – shall the labels be converted into numbers (only ASG and CTC)

set_num_labels(*numlab*)
sets number of labels :param int numlab: the number of labels

set_label_conversion(*onoff*)
sets label conversion on or off :param bool onoff: flag to set label conversion on/off

run()

class Fsa.**Hmm**(*graph*, *depth=6*, *allo_num_states=3*)
class to create HMM FSA

Parameters

- **fsa** (Graph) – represents the Graph on which the class operates
- **depth** (int) – the depth of the HMM FSA process
- **allo_num_states** (int) – number of allophone states

set_depth(*depth*)
sets the depth for the HMM FSA process :param int depth: the depth of the HMM FSA process

load_lexicon(*lexicon_name*)
loads Lexicon :param str lexicon_name: holds the path and name of the lexicon file

load_state_tying(*state_tying_name*)
loads StateTying :param state_tying_name: holds the path and name of the state tying file

class Fsa.**Fsa**
class to create Finite State Automaton

Parameters

- **lemma** (*str|list[str]*) – word or sentence
- **fsa_type** (*str*) – determines finite state automaton type: asg, ctc, hmm
- **num_states** (*int*) – number of states
- **edges** (*list*) – list of edges

where:

num_states: int, number of states. per convention, state 0 is start state, state (num_states - 1) is single final state

edges: list[(from,to,label_idx,weight)] from and to are state_idx >= 0 and < num_states, label_idx >= 0 and label_idx < num_labels –or– label_idx == num_labels for blank symbol weight is a float, in -log space

Parameters

- **filename** (*str*) – name of file to store graph
- **asg_repetition** (*int*) – repetition symbols for asg
- **num_labels** (*int*) – number of labels
- **label_conversion** (*bool*) – use chars or indexes
- **final_states** (*list[int]*) – list of final states
- **depth** (*int*) – depth / level of hmm
- **allo_num_states** (*int*) – number of allophone states
- **lexicon** (*str*) – lexicon file name
- **state_tying** (*str*) – state tying file name
- **phon_dict** (*dict*) – dictionary of phonemes, loaded from lexicon file

set_params(*asg_repetition=2*, *num_labels=256*, *label_conversion=False*, *depth=6*, *allo_num_states=3*, *lexicon_name=''*, *state_tying_name=''*, *single_state=False*)

sets the parameters for FSA generator checks if needed params for fsa type available otherwise erquests user input :param str filename: sets the output file name :param int asg_repetition:

if a label is repeated within the lemma how many repetitions will be substituted with a specific repetition symbol

Parameters

- **num_labels** (*int*) – total number of labels
- **label_conversion** (*bool*) – true: each label converted to index of its label false: no conversion
- **depth** (*int*) – depth of the hmm acceptor
- **allo_num_states** (*int*) – umber of allophone states
- **lexicon** (*str*) – lexicon file name
- **state_tying** (*str*) – state tyting file name
- **single_state** (*bool*) – produce additional fsa: single node

Returns

set_lemma(*lemma*)

Parameters **lemma** (*str*) – word or sentence

set_fsa_type(*fsa_type*)

Parameters **fsa_type** (*str*) – determines finite state automaton type: asg, ctc, hmm

set_filename(*filename*)

Parameters **filename** (*str*) – name of file to store graph

set_hmm_depth(*depth*)

set_lexicon(*lexicon_name=None*)
sets a new lexicon :param str lexicon_name: lexicon path

set_state_tying(*state_tying=None*)
sets a new state tying file :param str state_tying: state tying file/path

run()
runs the FSA

convert_label_seq_to_indices()
takes label sequence of chars and converts to indices (ascii numbering)

reduce_node_num()
takes the edges and nodes, then reduces all to one node

Fsa.**fsa_to_dot_format**(*file*, *num_states*, *edges*)

Parameters

- **num_states** –
- **edges** –

Returns

converts num_states and edges to dot file to svg file via graphviz

class Fsa.**BuildSimpleFsaOp**(*loop_emission_idxs=()*, *loop_scores=(0.0, 0.0)*)

itypes = (TensorType(int32, matrix),)

otypes = (TensorType(float32, matrix), TensorType(float32, vector), TensorType(float32, matrix))

perform(*node*, *inputs*, *output_storage*, *params=None*)

class Fsa.**FastBaumWelchBatchFsa**(*edges*, *weights*, *start_end_states*)
FSA(s) in representation format for FastBaumWelchOp.

Parameters

- **edges** (*numpy.ndarray*) – (4,num_edges), edges of the graph (from,to,emission_idx,sequence_idx)
- **weights** (*numpy.ndarray*) – (num_edges,), weights of the edges
- **start_end_states** (*numpy.ndarray*) – (2, batch), (start,end) state idx in automaton.

class Fsa.**FastBwFsaShared**
One FSA shared for all the seqs in one batch (i.e. across batch-dim). This is a simplistic class which provides the necessary functions to

add_edge(*source_state_idx*, *target_state_idx*, *emission_idx*, *weight=0.0*)

Parameters

- **source_state_idx** (int) –
- **target_state_idx** (int) –
- **emission_idx** (int) –
- **weight** (float) –

add_inf_loop(*state_idx*, *num_emission_labels*)

Parameters

- **state_idx** (int) –
- **num_emission_labels** (int) –

get_num_edges(*n_batch*)

Parameters **n_batch** (int) –

Return type *int*

get_edges(*n_batch*)

Parameters **n_batch** (int) –

Return edges (4,num_edges), edges of the graph (from,to,emission_idx,sequence_idx)

Return type numpy.ndarray

get_weights(*n_batch*)

Parameters **n_batch** (int) –

Return weights (num_edges,), weights of the edges

Return type numpy.ndarray

get_start_end_states(*n_batch*)

Parameters **n_batch** (int) –

Return start_end_states (2, batch), (start,end) state idx in automaton. there is only one single automaton.

Return type numpy.ndarray

get_fast_bw_fsa(*n_batch*)

Parameters **n_batch** (int) –

Return type *FastBaumWelchBatchFsa*

Fsa.**main**()

FunctionLoader

FunctionLoader.**make_funloader_code**(*recurrent_transform*, *fn_name*, *reset_fn_name=None*)

GeneratingDataset

class GeneratingDataset.**GeneratingDataset**(*input_dim*, *output_dim*, *num_seqs=inf*, *fixed_random_seed=None*, ***kwargs*)

Parameters

- **input_dim** (int) –
- **output_dim** (*int|dict[str,int|(int,int)|dict]*) –
- **num_seqs** (*int|float*) –
- **fixed_random_seed** (int) –

init_seq_order(*epoch=None*, *seq_list=None*)

Parameters **seq_list** – predefined order. doesn't make sense here

This is called when we start a new epoch, or at initialization.

is_cached(*start*, *end*)

generate_seq(*seq_idx*)

Return type *DatasetSeq*

get_num_timesteps()

num_seqs

get_seq_length(*sorted_seq_idx*)

get_input_data(*sorted_seq_idx*)

get_targets(*target*, *sorted_seq_idx*)

get_ctc_targets(*sorted_seq_idx*)

get_tag(*sorted_seq_idx*)

class GeneratingDataset.**Task12AXDataset**(***kwargs*)

12AX memory task. This is a simple memory task where there is an outer loop and an inner loop. Description here: http://psych.colorado.edu/~oreilly/pubs-abstr.html#OReillyFrank06

get_random_seq_len()

generate_input_seq(*seq_len*)

Somewhat made up probability distribution. Try to make in a way that at least some "R" will occur in the output seq. Otherwise, "R"s are really rare.

classmethod make_output_seq(*input_seq*)

Return type *list*[*int*]

estimate_output_class_priors(*num_trials*, *seq_len=10*)

Return type (*float*, *float*)

generate_seq(*seq_idx*)

class GeneratingDataset.**TaskEpisodicCopyDataset**(***kwargs*)

Episodic Copy memory task. This is a simple memory task where we need to remember a sequence. Described in: http://arxiv.org/abs/1511.06464 Also tested for Associative LSTMs. This is a variant where the lengths are random, both for the chars and for blanks.

generate_input_seq()

classmethod make_output_seq(*input_seq*)

Return type *list*[*int*]

generate_seq(*seq_idx*)

class GeneratingDataset.**TaskXmlModelingDataset**(*limit_stack_depth=4*, ***kwargs*)
XML modeling memory task. This is a memory task where we need to remember a stack. Defined in Jozefowicz et al. (2015). Also tested for Associative LSTMs.

generate_input_seq()

classmethod make_output_seq(*input_seq*)

Return type *list*[*int*]

generate_seq(*seq_idx*)

class GeneratingDataset.**TaskVariableAssignmentDataset**(***kwargs*)
Variable Assignment memory task. This is a memory task to test for key-value retrieval. Defined in Associative LSTM paper.

generate_input_seq()

classmethod make_output_seq(*input_seq*)

Return type *list*[*int*]

generate_seq(*seq_idx*)

class GeneratingDataset.**DummyDataset**(*input_dim*, *output_dim*, *num_seqs*, *seq_len=2*, *input_max_value=10.0*, *input_shift=None*, *input_scale=None*, ***kwargs*)

generate_seq(*seq_idx*)

class GeneratingDataset.**StaticDataset**(*data*, *target_list=None*, *output_dim=None*, *input_dim=None*, ***kwargs*)

generate_seq(*seq_idx*)

get_target_list()

class GeneratingDataset.**CopyTaskDataset**(*nsymbols*, *minlen=0*, *maxlen=0*, *minlen_epoch_factor=0*, *maxlen_epoch_factor=0*, ***kwargs*)

get_random_seq_len()

generate_seq(*seq_idx*)

Return type *DatasetSeq*

GeneratingDataset.**demo**()

HDFDataset

class HDFDataset.**HDFDataset**(**args*, ***kwargs*)

add_file(*filename*)

Setups data: self.seq_lengths self.file_index self.file_start self.file_seq_start

Use load_seqs() to load the actual data. :type filename: str

get_tag(*sorted_seq_idx*)

is_data_sparse(*key*)

get_data_dtype(*key*)

len_info()

Inv

class Inv.**InvOp**(*min_skip*, *max_skip*, *nstates*, *focus='last'*, *nil=-1*, *coverage=0*, *mode='viterbi'*)

make_node(*x*, *y*, *len_x*, *len_y*)

c_support_code()

c_compile_args()

c_code(*node*, *name*, *inp*, *out*, *sub*)

class Inv.**InvOpBackTrace**(*min_skip*, *max_skip*, *nstates*, *focus='last'*, *nil=-1*, *coverage=0*, *mode='viterbi'*)

make_node(*x*, *y*, *len_x*, *len_y*)

c_support_code()

c_compile_args()

c_code(*node*, *name*, *inp*, *out*, *sub*)

class Inv.**InvOpFull**(*min_skip*, *max_skip*, *nstates*, *focus='last'*, *mode='viterbi'*)

make_node(*x*, *y*, *len_x*, *len_y*)

c_support_code()

c_compile_args()

c_code(*node*, *name*, *inp*, *out*, *sub*)

class Inv.**AlignOp**

make_node(*x*, *y*, *len_x*, *len_y*)

c_support_code()

c_compile_args()

class Inv.**InvAlignOp**

c_code(*node*, *name*, *inp*, *out*, *sub*)

class Inv.**StdOpFull**(*skip_tdp*, *nstates*)

make_node(*x*, *y*, *len_x*, *len_y*)

c_support_code()

c_compile_args()

c_code(*node*, *name*, *inp*, *out*, *sub*)

LearningRateControl

class LearningRateControl.**ConstantLearningRate**(*defaultLearningRate*, *minLearningRate=0.0*, *defaultLearningRates=None*, *errorMeasureKey=None*, *relativeErrorAlsoRelativeToLearningRate=False*, *minNumEpochsPerNewLearningRate=0*, *filename=None*)

Parameters

- **defaultLearningRate** (float) – default learning rate. usually for epoch 1
- **| dict[int,float] defaultLearningRates** (list[float]) – learning rates
- **errorMeasureKey** (*str*) – for getEpochErrorValue() the selector for EpochData.error which is a dict
- **minNumEpochsPerNewLearningRate** (int) – if the lr was recently updated, use it for at least N epochs
- **filename** (*str*) – load from and save to file

calcLearningRateForEpoch(*epoch*)

Dummy constant learning rate. Returns initial learning rate. :type epoch: int :returns learning rate :rtype: float

need_error_info = False

class LearningRateControl.**LearningRateControl**(*defaultLearningRate*, *minLearningRate=0.0*, *defaultLearningRates=None*, *errorMeasureKey=None*, *relativeErrorAlsoRelativeToLearningRate=False*, *minNumEpochsPerNewLearningRate=0*, *filename=None*)

Parameters

- **defaultLearningRate** (float) – default learning rate. usually for epoch 1
- **| dict[int,float] defaultLearningRates** (list[float]) – learning rates
- **errorMeasureKey** (*str*) – for getEpochErrorValue() the selector for EpochData.error which is a dict
- **minNumEpochsPerNewLearningRate** (int) – if the lr was recently updated, use it for at least N epochs
- **filename** (*str*) – load from and save to file

class EpochData(*learningRate*, *error=None*)

LearningRateControl.**calcLearningRateForEpoch**(*epoch*)

:returns learning rate :rtype: float

LearningRateControl.**calcNewLearnignRateForEpoch**(*epoch*)

LearningRateControl.**calcRelativeError**(*oldEpoch*, *newEpoch*)

LearningRateControl.**getEpochErrorDict**(*epoch*)

LearningRateControl.**getEpochErrorValue**(*epoch*)

LearningRateControl.**getErrorKey**(*epoch*)

LearningRateControl.**getLastBestEpoch**(*last_epoch*, *first_epoch=1*, *filter_score=inf*, *only_last_n=-1*, *min_score_dist=0.0*)

Parameters

- **first_epoch** (int) – will check all epochs >= first_epoch
- **last_epoch** (int) – will check all epochs <= last_epoch
- **filter_score** (float) – all epochs which values over this score are not considered
- **only_last_n** (int) – if set, from the resulting list, we consider only the last only_last_n
- **min_score_dist** (float) – filter out epochs where the diff to the most recent is not big enough

Returns the last best epoch. to get the details then, you might want to use getEpochErrorDict.

Return type int|None

LearningRateControl.**getLastEpoch**(*epoch*)

LearningRateControl.**getLearningRateForEpoch**(*epoch*)

Return type *float*

LearningRateControl.**getMostRecentLearningRate**(*epoch*, *excludeCurrent=True*)

LearningRateControl.**load**()

classmethod LearningRateControl.**load_initial_from_config**(*config*)

Return type *LearningRateControl*

classmethod LearningRateControl.**load_initial_kwargs_from_config**(*config*)

Return type dict[str]

LearningRateControl.**need_error_info = True**

LearningRateControl.**save**()

LearningRateControl.**setDefaultLearningRateForEpoch**(*epoch*, *learningRate*)

LearningRateControl.**setEpochError**(*epoch*, *error*)

class LearningRateControl.**NewbobAbs**(*errorThreshold*, *learningRateDecayFactor*, ***kwargs*)

calcLearningRateForEpoch(*epoch*)
Newbob+ on train data. :type epoch: int :returns learning rate :rtype: float

classmethod load_initial_kwargs_from_config(*config*)

Return type dict[str]

class LearningRateControl.**NewbobMultiEpoch**(*numEpochs*, *updateInterval*, *relativeErrorThreshold*, *learningRateDecayFactor*, ***kwargs*)

Parameters **defaultLearningRate** (float) – learning rate for epoch 1+2

calcLearningRateForEpoch(*epoch*)
Newbob+ on train data. :type epoch: int :returns learning rate :rtype: float

classmethod load_initial_kwargs_from_config(*config*)

Return type dict[str]

class LearningRateControl.**NewbobRelative**(*relativeErrorThreshold*, *learningRateDecayFactor*, ***kwargs*)

Parameters **defaultLearningRate** (float) – learning rate for epoch 1+2

calcLearningRateForEpoch(*epoch*)
Newbob+ on train data. :type epoch: int :returns learning rate :rtype: float

classmethod load_initial_kwargs_from_config(*config*)

Return type dict[str]

LearningRateControl.**demo**()

LearningRateControl.**learningRateControlType**(*typeName*)

LearningRateControl.**loadLearningRateControlFromConfig**(*config*)

Return type *LearningRateControl*

LmDataset

class LmDataset.**AllophoneState**(*id=None*, *state=None*)

boundary = 0

context_future = ()

context_history = ()

format()

id = None

mark_final()

mark_initial()

state = None

class LmDataset.**Lexicon**(*filename*)

class LmDataset.**LmDataset**(*corpus_file*, *orth_symbols_file=None*, *orth_symbols_map_file=None*, *orth_replace_map_file=None*, *word_based=False*, *seq_end_symbol='[END]'*, *unknown_symbol='[UNKNOWN]'*, *parse_orth_opts=None*, *phone_info=None*, *add_random_phone_seqs=0*, *partition_epoch=1*, *auto_replace_unknown_symbol=False*, *log_auto_replace_unknown_symbols=10*, *log_skipped_seqs=10*, *error_on_invalid_seq=True*, *add_delayed_seq_data=False*, *delayed_seq_data_start_symbol='[START]'*, ***kwargs*)

Parameters

- **corpus_file** (*str|()->str*) – Bliss XML or line-based txt. optionally can be gzip.
- **phone_info** (*dict|None*) – if you want to get phone seqs, dict with lexicon_file etc. see PhoneSeqGenerator

- **orth_symbols_file** (*str | () -> str | None*) – list of orthography symbols, if you want to get orth symbol seqs
- **orth_symbols_map_file** (*str | () -> str | None*) – list of orth symbols, each line: "symbol index"
- **orth_replace_map_file** (*str | () -> str | None*) – JSON file with replacement dict for orth symbols
- **word_based** (bool) – whether to parse single words, or otherwise will be char-based
- **seq_end_symbol** (*str | None*) – what to add at the end, if given. will be set as postfix=[seq_end_symbol] or postfix=[] for parse_orth_opts.
- **parse_orth_opts** (*dict[str] | None*) – kwargs for parse_orthography()
- **add_random_phone_seqs** (int) – will add random seqs with the same len as the real seq as additional data
- **log_auto_replace_unknown_symbols** (*bool | int*) – write about auto-replacements with unknown symbol. if this is an int, it will only log the first N replacements, and then keep quiet.
- **log_skipped_seqs** (*bool | int*) – write about skipped seqs to logging, due to missing lexicon entry or so. if this is an int, it will only log the first N entries, and then keep quiet.
- **error_on_invalid_seq** (bool) – if there is a seq we would have to skip, error
- **add_delayed_seq_data** (bool) – will add another data-key "delayed" which will have the sequence delayed_seq_data_start_symbol + original_sequence[:-1]
- **delayed_seq_data_start_symbol** (*str*) – used for add_delayed_seq_data
- **partition_epoch** (int) – whether to partition the epochs into multiple parts. like epoch_split

get_data_dtype(*key*)

get_target_list()

init_seq_order(*epoch=None*, *seq_list=None*)

class LmDataset.**PhoneSeqGenerator**(*lexicon_file*, *allo_num_states=3*, *allo_context_len=1*, *state_tying_file=None*, *add_silence_beginning=0.1*, *add_silence_between_words=0.1*, *add_silence_end=0.1*, *repetition=0.9*, *silence_repetition=0.95*)

Parameters

- **lexicon_file** (*str*) – lexicon XML file
- **allo_num_states** (int) – how much HMM states per allophone (all but silence)
- **allo_context_len** (int) – how much context to store left and right. 1 -> triphone
- **| None state_tying_file** (*str*) – for state-tying, if you want that
- **add_silence_beginning** (float) – prob of adding silence at beginning
- **add_silence_between_words** (float) – prob of adding silence between words
- **add_silence_end** (float) – prob of adding silence at end
- **repetition** (float) – prob of repeating an allophone
- **silence_repetition** (float) – prob of repeating the silence allophone

generate_garbage_seq(*target_len*)

Parameters **target_len** (*int*) – len of the returned seq

Return type *list*[*AllophoneState*]

:returns allophone state list. those will have repetitions etc. It will randomly generate a sequence of phonemes and transform that into a list of allophones in a similar way than generate_seq().

generate_seq(*orth*)

Parameters **orth** (*str*) – orthography as a str. orth.split() should give words in the lexicon

Return type *list*[*AllophoneState*]

:returns allophone state list. those will have repetitions etc

get_class_labels()

orth_to_phones(*orth*)

random_seed(*seed*)

seq_to_class_idxs(*phones*, *dtype=None*)

Parameters

- **phones** (*list[AllophoneState]*) – list of allophone states
- **dtype** (*str*) – eg "int32"

Return type numpy.ndarray

:returns 1D numpy array with the indices

class LmDataset.**StateTying**(*state_tying_file*)

Log

class Log.**Stream**(*log*, *lvl*)

write(*msg*)

flush()

class Log.**Log**

initialize(*logs=[]*, *verbosity=[]*, *formatter=[]*)

write(*msg*)

MetaDataset

class MetaDataset.**ChunkShuffleDataset**(*dataset*, *chunk_shuffle_cache=1000*, *batch_gen_batch_size=5000*, *batch_gen_max_seqs=1*, *batch_gen_recurrent_net=True*, ***kwargs*)

This goes through a dataset, caches some recent chunks

Parameters **dataset** (*dict[str]*) – kwargs for init_dataset

get_target_list()

init_seq_order(*epoch=None*, *seq_list=None*)

Parameters | **None** **seq_list** (list[str]) – In case we want to set a predefined order.

is_less_than_num_seqs(*seq_idx*)

Return type *bool*

:returns whether seq_idx < num_seqs. In case num_seqs is not known in advance, it will wait until it knows that n is behind the end or that we have the seq.

class MetaDataset.**ClusteringDataset**(*dataset*, *cluster_map_file*, *n_clusters*, *single_cluster=False*, ***kwargs*)

This is a special case of MetaDataset, with one main subdataset, and we add a cluster-idx for each seq. We will read the cluster-map (seq-name -> cluster-idx) here directly.

get_data_dtype(*key*)

get_data_keys()

get_tag(*seq_idx*)

init_seq_order(*epoch=None*, *seq_list=None*)

is_less_than_num_seqs(*n*)

num_seqs

class MetaDataset.**CombinedDataset**(*datasets*, *data_map*, *data_dims*, *data_dtypes=None*, *window=1*, ***kwargs*)

This combines multiple different datasets, which provide different data-sources. E.g. one can provide am-dataset with data:acoustic-features -> classes:characters (acoustic model train data), and lm-dataset provides just data:characters (language model train data). Note: The mapping has been inverted. We now expect (dataset-key, dataset-data-key) -> self-data-key am-dataset:data -> am-data, am-dataset:classes -> am-classes, lm-dataset:data -> lm-data. For each sequence idx, it will select one of the given datasets, fill in the data-keys of this dataset and will return empty sequences for the remaining datasets. The selection of the dataset will be random and equally distributed, over the sum of num-seqs.

Parameters

- **datasets** (dict[str,dict[str]]) – dataset-key -> dataset-kwargs. including keyword 'class' and maybe 'files'
- **data_map** (dict[(str,str),str]) – (dataset-key, dataset-data-key) -> self-data-key. Should contain 'data' as key. Also defines the target-list, which is all except 'data'.
- **data_dims** (dict[str,(int,int)]) – self-data-key -> data-dimension, len(shape) (1 ==> sparse repr).
- **data_dtypes** (dict[str,str]) – self-data-key -> dtype. automatic if not specified

get_data_dim(*key*)

get_data_dtype(*key*)

get_target_list()

init_seq_order(*epoch=None*, *seq_list=None*)

is_less_than_num_seqs(*n*)

class MetaDataset.**ConcatDataset**(*datasets*, ***kwargs*)

This concatenates multiple datasets. They are expected to provide the same data-keys and data-dimensions. It will go through the datasets always in order.

Parameters **datasets** (list[dict[str]]) – list of kwargs for init_dataset

get_target_list()

init_seq_order(*epoch=None*, *seq_list=None*)

Parameters | **None seq_list** (*list[str]*) – In case we want to set a predefined order.

num_seqs

class MetaDataset.**MetaDataset**(*seq_list_file*, *seq_lens_file*, *datasets*, *data_map*, *data_dims*, *data_dtypes=None*, *window=1*, ***kwargs*)

This wraps around one or multiple datasets and might provide extra information. Every dataset is expected to provide the the same sequences, where the sequence list is given by a file.

Parameters

- **seq_list_file** (*str*) – filename. line-separated
- **seq_lens_file** (*str*) – filename. json. dict[str,dict[str,int]], seq-tag -> data-key -> len
- **datasets** (*dict[str,dict[str]]*) – dataset-key -> dataset-kwargs. including keyword 'class' and maybe 'files'
- **data_map** (*dict[str,(str,str)]*) – self-data-key -> (dataset-key, dataset-data-key). Should contain 'data' as key. Also defines the target-list, which is all except 'data'.
- **data_dims** (*dict[str,(int,int)]*) – self-data-key -> data-dimension, len(shape) (1 ==> sparse repr).
- **data_dtypes** (*dict[str,str]*) – self-data-key -> dtype. automatic if not specified

get_data_dtype(*key*)

get_seq_length(*sorted_seq_idx*)

get_tag(*sorted_seq_idx*)

get_target_list()

init_seq_order(*epoch=None*, *seq_list=None*)

MultiBatchBeam

MultiBatchBeam.**multi_batch_beam**(*array*, *start_idxs*, *batch_lens*, *beam_width*, *wrap_mode*, *pad_left=0*, *pad_right=0*, *idx_dim=0*, *batch_dim=1*)

Parameters

- **array** – ndarray, at least 2D. symbolic
- **start_idxs** – ndarray, 1D. symbolic. can be float (for gpu)
- **batch_lens** – ndarray, 1D. symbolic. len of each batch. can be float (for gpu)
- **beam_width** – scalar. symbolic.
- **wrap_mode** – "wrap_around" or "pad". static.
- **idx_dim** – int. where to apply each start_idxs[i]. static.
- **batch_dim** – the same dim as in start_idxs. static.
- **pad_value** – used in wrap_mode "pad". automatically broadcasted. symbolic.

Returns ndarray like array, but shape[idx_dim] == beam_width

See also *_naive_multi_batch_beam* for one naive reference implementation.

class MultiBatchBeam.**MultiBatchBeamOp**(*wrap_mode*, *idx_dim=0*, *batch_dim=1*)

make_node(*array*, *start_idxs*, *batch_lens*, *beam_width*, *pad_left*, *pad_right*)

perform(*node*, *inputs*, *output_storage*)

infer_shape(*node*, *input_shapes*)

grad(*inputs*, *output_grads*)

connection_pattern(*node*)

class MultiBatchBeam.**MultiBatchBeamGradAddOp**(*wrap_mode*, *idx_dim=0*, *batch_dim=1*, *inplace=False*, *zero_with_shape=False*, *array_ndim=None*)

(D_array / D_array_shape, start_idxs, batch_lens, beam_width, D_beam) -> D_array + grad

Parameters

- **wrap_mode** (*str*) – "wrap_around" or "pad"
- **idx_dim** (int) – usually that's time dim
- **batch_dim** (int) – batch dim
- **inplace** (bool) – operate inplace on input
- **zero_with_shape** (bool) – we get D_array_shape as the first input and init D_array with zero
- **array_ndim** (int) – ndim of array/D_array. needed for zero_with_shape

make_node(*D_array_or_shape*, *start_idxs*, *batch_lens*, *beam_width*, *D_beam*)

infer_shape(*node*, *input_shapes*)

perform(*node*, *inputs*, *output_storage*)

class MultiBatchBeam.**GpuMultiBatchBeamOp**

NativeOp

Generic interface which automatically creates: * CPU and GPU op * inplace and not inplace * grad variants

class NativeOp.**NativeOpBaseMixin**(*in_info*, *out_info*, *c_fw_code*, *c_bw_code=None*, *c_extra_support_code=None*, *code_version=None*, *cpu_support=True*, *grad_input_map=None*, *name=None*)

The purpose of having this as a separate base class is to make this independent of any Theano specific functionality so that we can also use this base for example for TensorFlow.

Parameters

- **in_info** (list[dict(str)]) – each dict describes one input var. attribs in the dict:

  int ndim: the ndim. tuple shape: tuple and can contain None for specific dimensions.

  optional attribs: str dtype: "float32" by default. bool need_contiguous: false by default. int want_inplace: -1 by default. try to optimize to destroy input, on output-index.

  "dummy_out" is a special value which will add another output.

  bool is_inplace: false by default. whether the optimization was applied. str gradient: can be "disconnected". see grad(). bool bw_input: True by default. add this param to the bw input.

  other attribs are just ignored.

- **out_info** (*list[dict(str)]*) – like in_info. slightly different behavior for:

  shape: we also allow refs to the in_info in the form (in-idx,dim). see infer_shape(). need_contiguous/want_inplace: used for bw, in case for bw_input == True.

- **c_fw_code** (*str*) – C code for forward pass
- **c_extra_support_code** (*str|dict[str]*) – C support code (for c_support_code)
- **c_bw_code** (*str|None*) – C code for backward pass (for gradient)
- **code_version** (*tuple[int]*) – will be returned by c_code_cache_version.
- **cpu_support** (*bool*) –
- **grad_input_map** (*tuple[int]|callable*) – selection of grad inputs. by default, we get all inputs + all outputs + all grad outputs.
- **name** (*str*) – name

infer_shape(*node*, *input_shapes*)

kwargs_for_grad_op()

Returns the kwargs for creating a NativeOp for the gradient op. e.g. includes in_info, out_info, etc

Return type dict[str]

Note: The inputs of the gradient are by default: fwd_op.inputs + fwd_op.outputs + output_grads. We filter them via self._filter_grad_inputs.

make_results_of_gradient(*grad_op_outputs*, *disconnected_type=None*)

Parameters

- **grad_op_outputs** (*list[T]*) – this is already with dummy outputs removed
- **disconnected_type** (*S*) –

Returns gradient for each input of our op

Return type *list*[T|S]

class NativeOp.**NativeOp**(*custom_grad=None*, ***kwargs*)

We wrap some C code which can define a forward pass and optionally a backward pass (for gradient calculation). The C code should be Numpy and CUDA compatible. See NativeOp.cpp. We also support inplace operations, i.e. we can operate inplace on some inputs. You can define in a flexible way all the inputs and the outputs. See __init__() for the details.

All output variables are created automatically with the right shape but their content is not initialized, except when its used by some input variable as the inplace output - in that case, it is either the input variable or it has a copy of its data.

Parameters

- **custom_grad** (*function*) – if given, will use this instead for self.grad
- **kwargs** (*dict[str]*) – all passed to NativeOpBaseMixin

classmethod **as_tensor_var**(*v*)

classmethod **tensor_type**(*dtype*, *ndim*)

classmethod **contiguous**(*v*)

grad(*inputs*, *output_grads*)

connection_pattern(*node*)

make_node(**args*)

perform(*node*, *inputs*, *output_storage*)

c_code_cache_version()

c_support_code()

c_libraries()

c_compile_args()

c_lib_dirs()

c_header_dirs()

c_code(*node*, *name*, *inputs*, *outputs*, *sub*)

class NativeOp.**GpuNativeOp**(*custom_grad=None*, ***kwargs*)

Parameters

- **custom_grad** (*function*) – if given, will use this instead for self.grad
- **kwargs** (*dict[str]*) – all passed to NativeOpBaseMixin

classmethod as_tensor_var(*v*)

classmethod tensor_type(*dtype*, *ndim*)

classmethod contiguous(*v*)

c_support_code()

class NativeOp.**NativeOpGenBase**

Base interface for op generation. See NativeOp.__init__() for attribs.

in_info = None

out_info = None

c_fw_code = None

c_bw_code = None

c_extra_support_code = None

code_version = None

grad_input_map = None

custom_grad = None

cpu_support = True

make_op()

classmethod map_layer_inputs_to_op(**inputs*)

classmethod map_layer_output_from_op(**outputs*)

class NativeOp.**LstmGenericBase**

inputs:

param Z {input,output,forget} gate + cell state. 3d (time,batch,dim*4)

param V_h recurrent matrix. 2d (dim,dim*4)

param c initial cell state. 2d (batch,dim)

param i index. 2d (time,batch) -> 0 or 1

outputs:

param Y output. 3d (time,batch,dim)

param H gates and cell state. 3d (time,batch,dim*4)

param d final cell state. 2d (batch,dim)

in_info = ({'ndim': 3, 'bw_out_var': {'shape': ((2, 0), (2, 1), (0, 1))}, 'name': 'Z', 'need_contiguous': True, 'shape': (No

out_info = ({'ndim': 3, 'shape': ((0, 0), (0, 1), (1, 0)), 'need_contiguous': True, 'name': 'Y', 'bw_grad_var': {'want_inp

classmethod **grad_input_map**(*Z*, *V_h*, *c*, *i*, *Y*, *H*, *d*, *DY*, *DH*, *Dd*)

classmethod **map_layer_inputs_to_op**(*Z*, *V_h*, *i*)

c_extra_support_code = {'lstm_kernel': '\n DEF_KERNEL\n void lstm_kernel(float* data, const float* old_state, bo

c_fw_code = '\n // Z*, V_h, c, i = input_names (*: inplace)\n // Y, H, d = output_names\n assert(n_inputs == 4);\n assert(

c_bw_code = '\n // V_h, c, i, Y, H*, DY*, Dd = input_names (*: inplace)\n // DZ, DV_h, Dc, tmpDc = output_names\n ass

code_version = ()

class NativeOp.**Chunking**

Given an input in 3d (n_time,n_batch,n_dim), we chunk up the time dimension in chunks of size chunk_size, every chunk_step frames. This results in an 3d output (chunk_size, n_batch * n_chunks, n_dim) where n_chunks = floor(max(n_time - chunk_size + chunk_step - 1, 0) / chunk_step) + 1. Examples:

n_time=1, chunk_size=50, chunk_step=10 -> n_chunks=1 n_time=49, chunk_size=50, chunk_step=10 -> n_chunks=1 n_time=50, chunk_size=50, chunk_step=10 -> n_chunks=1 n_time=51, chunk_size=50, chunk_step=10 -> n_chunks=2 n_time=60, chunk_size=50, chunk_step=10 -> n_chunks=2 n_time=61, chunk_size=50, chunk_step=10 -> n_chunks=3 n_time=99, chunk_size=50, chunk_step=10 -> n_chunks=6 n_time=100, chunk_size=50, chunk_step=10 -> n_chunks=6 n_time=101, chunk_size=50, chunk_step=10 -> n_chunks=7

in_info = ({'ndim': 3, 'shape': (None, None, None), 'name': 'input'}, {'ndim': 2, 'gradient': 'disconnected', 'shape': (N

out_info = ({'ndim': 3, 'shape': ((2, 0), (2, 1), (2, 2)), 'name': 'output'}, {'ndim': 2, 'shape': ((3, 0), (3, 1)), 'name': 'oin

c_extra_support_code = {'copy_kernel': '\n DEF_KERNEL\n void copy_kernel(\n float* chunk_params,\n float* in

c_fw_code = '\n assert_cmp(n_inputs, ==, 5);\n assert_cmp(n_outputs, ==, 2);\n Ndarray* input = inputs[0];\n Ndarray

code_version = ()

static **naive_chunk_start_frames**(*n_time*, *chunk_size*, *chunk_step*)

This is just for documentation / demonstration. Also used by testing code.

classmethod **custom_grad**(*op*, *inputs*, *output_grads*)

NativeOp.**chunk**(*x*, *index*, *chunk_size*, *chunk_step*)

class NativeOp.**UnChunking**

This reverses the output from *Chunking*, i.e. chunking the time dimension. We get a 3d input (chunk_size, n_batch * n_chunks, n_dim) and return an 3d output (n_time, n_batch, n_dim) where the chunks are of size chunk_size, every chunk_step frames. Because of overlaps, we have to combine the overlapping chunks somehow. We will do that with a uniform distribution, i.e. take the mean of all overlaps per frame.

in_info = ({'ndim': 3, 'shape': (None, None, None), 'name': 'input'}, {'ndim': 2, 'gradient': 'disconnected', 'shape': (N

out_info = ({'ndim': 3, 'shape': ((2, 0), (2, 1), (2, 2)), 'name': 'output'}, {'ndim': 2, 'shape': ((3, 0), (3, 1)), 'name': 'oin

c_extra_support_code = {'unchunk_kernel': '\n DEF_KERNEL\n void unchunk_kernel(\n float* chunk_params,\n

c_fw_code = '\n assert_cmp(n_inputs, ==, 6);\n assert_cmp(n_outputs, ==, 3);\n Ndarray* input = inputs[0];\n Ndarray

code_version = ()

classmethod custom_grad(*op*, *inputs*, *output_grads*)

NativeOp.**unchunk**(*x*, *index*, *chunk_size*, *chunk_step*, *n_time*, *n_batch*)

class NativeOp.**SubtensorBatchedIndex**

Consider you have: idx: 2d (n_time, n_batch) -> idx (in [0..n_dim-1]) x: 3d (n_time, n_batch, n_dim)

Then, this op will calculate: x[..., idx[...]]: 2d (n_time, n_batch)

in_info = ({'ndim': 3, 'shape': (None, None, None), 'name': 'x', 'bw_in_var': {'want_inplace': 0}}, {'ndim': 2, 'gradien

out_info = ({'ndim': 2, 'shape': ((0, 0), (0, 1)), 'name': 'y'},)

classmethod grad_input_map(*x*, *idx*, *y*, *DY*)

c_extra_support_code = {'select_kernel': '\n DEF_KERNEL\n void select_kernel(\n float* x, long x_dim0, long x_di

c_fw_code = '\n assert_cmp(n_inputs, ==, 2);\n assert_cmp(n_outputs, ==, 1);\n Ndarray* x = inputs[0];\n Ndarray* idx

c_bw_code = '\n assert_cmp(n_inputs, ==, 3);\n assert_cmp(n_outputs, ==, 1);\n Ndarray* x = inputs[0];\n Ndarray* idx

NativeOp.**subtensor_batched_index**(*x*, *idx*)

class NativeOp.**SparseToDense**

Expects a sparse matrix in COOrdinate format, where W[s0[i,b],b,s1[i]] = weight[i,b] for all i, and all batches b. Will return W (time,batch,dim).

in_info = ({'ndim': 3, 'shape': (None, None, None), 'need_contiguous': True, 'name': '_initial_W', 'want_inplace': 0},

out_info = ({'ndim': 3, 'shape': ((0, 0), (0, 1), (0, 2)), 'name': 'W'},)

c_extra_support_code = {'assign_kernel': '\n DEF_KERNEL\n void assign_kernel(\n float* out, float* s0, float* s1,

c_fw_code = '\n assert(n_inputs == 5);\n assert(n_outputs == 1);\n Ndarray* s0 = inputs[1];\n Ndarray* s1 = inputs[2];

NativeOp.**sparse_to_dense**(*s0*, *s1*, *weight*, *mask*, *n_time*, *n_dim*)

NativeOp.**onehot_to_sparse**(*y*, *mask*)

NativeOp.**sparse_slice_offset**(*s0*, *idx*)

Parameters

- **s0** – 1D tensor, ordered indices for sparse coo-format matrix (without batch)
- **idx** – scalar, index to find in s0

Returns s0_idx, such that s0[i] >= idx for all i >= s0_idx, s0[i] < idx for all i < s0_idx.

This assumes that the indices in s0 are ordered.

NativeOp.**sparse_splice_offset_numpy**(*s0*, *idx*)

Like sparse_slice_offset().

class NativeOp.**MaxAndArgmaxSparse**

Expects a sparse matrix in COOrdinate format, where W[s0[i,b],s1[i],b] = weight[i,b] for all i, and all batches b. It will return the max and argmax for all W[:,:,b] over the second axis.

in_info = ({'ndim': 2, 'gradient': 'disconnected', 'shape': (None, None), 'need_contiguous': True, 'name': 's0'}, {'ndim

out_info = ({'ndim': 2, 'shape': ((4, 0), (4, 1)), 'name': 'out_max'}, {'ndim': 2, 'shape': ((5, 0), (5, 1)), 'name': 'out_arg

c_extra_support_code = {'doit_kernel': '\n DEF_KERNEL\n void doit_kernel(\n long n_batch, long n_in_time, long

c_fw_code = '\n assert(n_inputs == 6);\n assert(n_outputs == 2);\n Ndarray* s0 = inputs[0];\n Ndarray* s1 = inputs[1];

code_version = ()

NativeOp.**max_and_argmax_sparse**(*s0*, *s1*, *weight*, *mask*, *out_max*, *out_arg*)

class NativeOp.**CrossEntropySoftmaxAndGradientZSparse**

y_target is given in sparse COOrdinate format. We will calculate CE[t,b] = sum_i y_target[t,b,i] * log(softmax(z[t,b])[i]), for any timeframe t and batch b, and grad(CE[t,b], z[t,b]) = softmax(z[t,b]) - y_target[t,b]. We also support an index-mask for z, i.e. for the possible [t,b].

in_info = ({'ndim': 3, 'shape': (None, None, None), 'need_contiguous': True, 'name': 'z'}, {'ndim': 2, 'shape': (None, N

out_info = ({'ndim': 2, 'shape': ((0, 0), (0, 1)), 'name': 'out_ce'}, {'ndim': 3, 'shape': ((0, 0), (0, 1), (0, 2)), 'name': 'out_

c_extra_support_code = {'softmax_kernel': '\n DEF_KERNEL\n void softmax_kernel(\n float* out_softmax,\n float

c_fw_code = '\n assert(n_inputs == 6);\n assert(n_outputs == 3);\n Ndarray* z = inputs[0];\n Ndarray* z_mask = inputs

NativeOp.**crossentropy_softmax_and_gradient_z_sparse**(*z*, *z_mask*, *y_target_t*, *y_target_i*, *y_target_w*, *y_target_mask*)

NativeOp.**crossentropy_softmax_and_gradient_z_sparse__slow**(*z*, *z_mask*, *y_target_t*, *y_target_i*, *y_target_w*, *y_target_mask*)

class NativeOp.**FastBaumWelchOp**

inputs:

param am_scores scores in -log space. 3d (time,batch,dim)

param edges edges of the graph (from,to,emission_idx,sequence_idx)

param weights weights of the edges

outputs:

param output Baum-Welch alignment, scores in -log space. 3d (time,batch,dim), like am_scores

in_info = ({'ndim': 3, 'gradient': 'disconnected', 'shape': (None, None, None), 'need_contiguous': True, 'name': 'am_s

out_info = ({'ndim': 3, 'shape': ((0, 0), (0, 1), (0, 2)), 'need_contiguous': True, 'name': 'output'}, {'ndim': 2, 'shape': (

c_extra_support_code = {'21_normalize_2': '\n __global__\n void normalize_2(float* buffer, unsigned* sequence_id

c_fw_code = '\n // am_scores, edges, weights, start_end_states, index, state_buffer* = input_names (*: inplace)\n // outp

c_bw_code = None

code_version = 55

cpu_support = False

class NativeOp.**SegmentFastBaumWelchOp**(*segmentwise_normalization=False*, *dump_targets_interval=None*)

in_info = ({'ndim': 3, 'gradient': 'disconnected', 'shape': (None, None, None), 'need_contiguous': True, 'name': 'am_s

out_info = ({'ndim': 3, 'shape': ((0, 0), (0, 1), (0, 2)), 'need_contiguous': True, 'name': 'output'}, {'ndim': 2, 'shape': (

c_extra_support_code = {'10_fill_array': '\n __global__\n void fill_array(float* array, float value, unsigned size) {\n

cpu_support = False

c_fw_code = '\n // inputs: am_scores, batch_idxs, edges, weights, length_models, start_end_states, index, am_score_scal

Network

class Network.**LayerNetwork**(*n_in=None*, *n_out=None*, *base_network=None*, *data_map=None*, *data_map_i=None*, *shared_params_network=None*, *mask=None*, *sparse_input=False*, *target='classes'*, *train_flag=False*, *eval_flag=False*)

Parameters

- **n_in** (int) – input dim of the network
- **n_out** (*dict[str, (int, int)]*) – output dim of the network. first int is num classes, second int is 1 if it is sparse, i.e. we will get the indices.
- **data_map** (*dict[str,theano.Variable]*) – if specified, this will be used for x/y (and it expects data_map_i)
- **data_map_i** (*dict[str,theano.Variable]*) – if specified, this will be used for i/j
- **base_network** (*LayerNetwork|None*) – optional base network where we will derive x/y/i/j/n_in/n_out from. data_map will have precedence over base_network.
- **shared_params_network** (*LayerNetwork|()->LayerNetwork|None*) – optional network where we will share params with. we will error if there is a param which cannot be shared.
- **mask** (*str*) – e.g. "unity" or None ("dropout")
- **sparse_input** (bool) – for SourceLayer
- **target** (*str*) – default target
- **train_flag** (bool) – marks that we are used for training
- **eval_flag** (bool) – marks that we are used for evaluation

add_cost_and_constraints(*layer*)

add_layer(*layer*)

:rtype NetworkHiddenLayer.Layer

declare_train_params(***kwargs*)
Kwargs as in self.get_params(), or default values.

classmethod epoch_from_hdf_model(*model*)

:returns last epoch the model was trained on :rtype: int

classmethod epoch_from_hdf_model_filename(*model_filename*)

:returns last epoch the model was trained on :rtype: int

classmethod from_base_network(*base_network*, *json_content=None*, *share_params=False*, *base_as_calc_step=False*, ***kwargs*)

Parameters

- **base_network** (LayerNetwork) – base network to derive from

- **json_content** (*dict[str]|None*) – JSON content for subnetwork. if None, will use from base network
- **share_params** (bool) – will use the same params as the base network
- **base_as_calc_step** (bool) – base is calc step 0. see below
- **kwargs** (*dict[str]*) – kwargs for __init__

Return type *LayerNetwork*

classmethod from_config_topology(*config*, *mask=None*, ***kwargs*)

Parameters **mask** (*str*) – e.g. "unity" or None ("dropout"). "unity" is for testing.

Return type *LayerNetwork*

classmethod from_description(*description*, *mask=None*, ***kwargs*)

Parameters **mask** (*str*) – e.g. "unity" or None ("dropout")

Return type *LayerNetwork*

classmethod from_hdf(*filename=None*, *model=None*, *load_params=True*, ***kwargs*)

Gets the JSON from the hdf file, initializes the network and loads the network params. :param str|None filename: filename of hdf :param h5py.File|None model: hdf, if no filename is provided :param bool load_params: whether to load the params

classmethod from_hdf_model_topology(*model*, ***kwargs*)

Return type *LayerNetwork*

classmethod from_json(*json_content*, *n_in=None*, *n_out=None*, *network=None*, ***kwargs*)

Parameters **| None network** (LayerNetwork) – optional already existing instance

Return type *LayerNetwork*

classmethod from_json_and_config(*json_content*, *config*, ***kwargs*)

Return type *LayerNetwork*

get_all_layers()

get_all_params_vars()

get_calc_step(*i*)

Parameters **i** (int) – calc step, 0 to n

Return type *LayerNetwork*

Used by CalcStepLayer. Will automatically create the requested calc step. Calc step 0 is the base network (calc_step_base).

get_layer(*layer_name*)

get_layer_param(*layer_name*, *param_name*, *param*)

Used by Container.add_param() to maybe substitute a parameter instead of creating a new shared var. :param str layer_name: the layer name where this param will be added :param str param_name: the name of the param :param theano.SharedVariable param: the already created shared var :rtype None | theano.Variable If we return None, Container.add_param() will continue as usual.

get_objective()

get_params_dict()

Return type dict[str,dict[str,numpy.ndarray|theano.sandbox.cuda.CudaNdArray]]

get_params_shared_flat_dict()

Return type dict[str,theano.shared]

This will collect all vars of all layers in one dict. We extend the param name with our custom scheme.

get_params_vars(*hidden_layer_selection*, *with_output*)

Return type *list*[theano.compile.sharedvalue.SharedVariable]

:returns list (with well-defined order) of shared variables

get_train_param_args_default()
:returns default kwargs for self.get_params(), which returns all params with this.

get_used_data_keys()

init_args()

classmethod init_args_from_config(*config*)

Return type dict[str]

:returns the kwarg for cls.from_json()

classmethod json_from_config(*config*, *mask=None*)

Parameters **mask** (*str*) – "unity", "none" or "dropout"

Return type dict[str]

load_hdf(*model*)

:returns last epoch this was trained on :rtype: int

make_classifier(*name='output'*, *target='classes'*, ***kwargs*)

Parameters

- **sources** (*list[NetworkBaseLayer.Layer]*) – source layers
- **loss** (*str*) – loss type, "ce", "ctc" etc

new_subnetwork(*json_content*, *n_out*, *data_map*, *data_map_i*)

Parameters

- **json_content** (*dict[str,dict]*) – subnetwork specification
- **n_out** (*dict[str,list[int,int]]*) – n_out info for subnetwork
- **data_map** (*dict[str,theano.Variable]*) – data
- **data_map_i** (*dict[str,theano.Variable]*) – indices for data

Return type *LayerNetwork*

The data input for the subnetwork is not derived from ourselves but specified explicitly through n_out & data_map.

num_params()

print_network_info(*name='Network'*)

save_hdf(*model*, *epoch*)

set_cost_constraints_and_objective()

set_params_by_dict(*params*)

to_json()

to_json_content()

use_target(*target*, *dtype*)

NetworkBaseLayer

class NetworkBaseLayer.**Container**(*layer_class=None*, *name=''*, *network=None*, *train_flag=False*, *eval_flag=False*, *depth=1*, *consensus='flat'*, *forward_weights_init=None*, *bias_init=None*, *weight_clip=0.0*, *cost=None*, *recurrent_weights_init=None*, *substitute_param_expr=None*)

Parameters

- **layer_class** (*str*) – name of layer type, e.g. "hidden", "recurrent", "lstm" or so. see LayerClasses.
- **name** (*str*) – custom layer name, e.g. "hidden_2"
- **network** (Network.LayerNetwork) – the network which we will be part of
- **forward_weights_init** (*str*) – see self.create_forward_weights()
- **bias_init** (*str*) – see self.create_bias()

add_param(*param*, *name=''*)

Return type theano.SharedVariable

create_bias(*n*, *prefix='b'*, *name=''*, *init_eval_str=None*)

Parameters **n** (int) – output dimension

Return type theano.shared

create_forward_weights(*n*, *m*, *name=None*)

Parameters

- **n** (int) – input dimension
- **m** (int) – output dimension
- **name** (*str|None*) – layer name

Return type theano.shared

create_random_normal_weights(*n*, *m*, *scale=None*, *name=None*)

create_random_uniform_weights(*n*, *m*, *p=None*, *p_add=None*, *l=None*, *name=None*, *depth=None*)

create_random_uniform_weights1(*n*, *m*, *p=None*, *l=None*, *name=None*)

create_random_uniform_weights2(*n*, *m=None*, *name=None*)

create_random_unitary_tiled_weights(*n*, *m*, *name=None*)

create_random_unitary_weights(*n*, *m*, *name=None*)

create_recurrent_weights(*n*, *m*, *name=None*)

Parameters

- **n** (*int*) – input dimension
- **m** (*int*) – output dimension
- **name** (*str|None*) – layer name

Return type theano.shared

dot(*vec*, *mat*)

get_params_dict()

Return type dict[str,numpy.ndarray|theano.sandbox.cuda.CudaNdArray]

get_params_vars()
:returns list of shared vars in a well-defined order

classmethod guess_source_layer_name(*layer_name*)

classmethod initialize_rng()

layer_class = None

load(*head*)

num_params()

rng_seed = 1234

save(*head*)

set_attr(*name*, *value*)

Parameters

- **name** (*str*) – key name
- **value** (*bool|int|float|str|list|dict*) – value

This will be stored in to_json() and save() (in HDF). More complex types like list or dict will be encoded as a JSON-str when saved to HDF.

set_params_by_dict(*params*)

shared(*value*, *name*, *borrow=True*)

to_json()

class NetworkBaseLayer.**Layer**(*sources*, *n_out*, *index*, *y_in=None*, *target=None*, *target_index=None*, *sparse=False*, *cost_scale=1.0*, *input_scale=1.0*, *L1=0.0*, *L2=0.0*, *L2_eye=None*, *varreg=0.0*, *output_L2_reg=0.0*, *output_entropy_reg=0.0*, *output_entropy_exp_reg=0.0*, *with_bias=True*, *mask='unity'*, *dropout=0.0*, *batch_drop=False*, *batch_norm=False*, *bn_use_sample=False*, *layer_drop=0.0*, *residual=False*, *carry=False*, *sparse_filtering=False*, *gradient_scale=1.0*, *trainable=True*, *device=None*, *dtype='float32'*, ***kwargs*)

Parameters

- **sources** (*list[NetworkBaseLayer.Layer]*) – list of source layers

- **n_out** (int) – output dim of W_in and dim of bias
- **L1** (float) – l1-param-norm regularization
- **L2** (float) – l2-param-norm regularization
- **mask** (*str*) – "unity" or "dropout"

add_param(*param*, *name=''*, *constraints=True*, *custom_update=None*, *custom_update_normalized=False*, *custom_update_exp_average=0*, *custom_update_condition=None*, *custom_update_accumulate_batches=None*)

Return type theano.SharedVariable

batch_norm(*h*, *dim*, *use_shift=True*, *use_std=True*, *use_sample=0.0*, *force_sample=False*, *index=None*, *sample_mean=None*, *gamma=None*, *beta=None*, *depth_norm=False*)

concat_units(*other*, *axis=1*)

cost()

Return type (theano.Variable | None, dict[theano.Variable,theano.Variable] | None)

Returns cost, known_grads

cost_scale()

Return type theano.Variable

errors()

Return type theano.Variable

find_data_layer()

get_branching()

get_energy()

make_consensus(*networks*, *axis=2*)

make_constraints()

make_output(*output*, *collapse=True*, *sample_mean=None*, *gamma=None*)

output_index()

recurrent = False

to_json()

transfer_output(*device*)

class NetworkBaseLayer.**SourceLayer**(*n_out*, *x_out=None*, *delay=0*, *sparse=False*, *name=''*, *network=None*, *eval_flag=False*, *data_key=None*, *sources=None*, *dropout=0*, *train_flag=None*, *mask=None*, *index=None*, *y_in=None*, *dtype=None*)

cost()

errors()

Return type theano.Variable

layer_class = 'source'

make_constraints()

recurrent = False

transfer_output(*device*)

NetworkCNNLayer

class NetworkCNNLayer.**CNN**(*n_features=1*, *filter=1*, *d_row=-1*, *border_mode='valid'*, *conv_stride=(1, 1)*, *pool_size=(1, 1)*, *filter_dilation=(1, 1)*, *ignore_border=1*, *pool_stride=0*, *pool_padding=(0, 0)*, *mode='max'*, *activation='tanh'*, *dropout=0.0*, *factor=1.0*, *base=None*, *transpose=False*, *force_sample=False*, ***kwargs*)

Parameters

- **n_features** (int) – integer the number of feature map(s), e.g. 32, 64, or so on. the input will be interpret as (width|time, batch, height * n_in_features) and the output will be (width|time, batch, height * n_features).
- **filter** (*int | (int, int)*) – integer or tuple of length 2 the filter size/shape, i.e. the number of row(s) and/or columns(s) from the filter shape. when this filter type is integer, it means the number of rows the same as the number of columns. e.g. 3, 5, (1,3), or so on.
- **d_row** (int) – integer the number of row(s) from the input the default value is -1, which the dimension comes from the n_out of the input. otherwise, this has to be filled only for the first convolutional layer and the rest layer will use the number of rows from the previous layer.
- **border_mode** (*str*) – string "valid" – only apply filter to complete patches of the image.

  Generates output of shape: (image_shape - filter_shape + 1).

  "full" – zero-pads image to multiple of filter shape to generate output of shape: (image_shape + filter_shape - 1). "same" – keep the dimension of convolutional layer output the same as the input dimension.
- **conv_stride** (*(int, int)*) – tuple of length 2 factor by which to subsample the convolutional layer output. this stride is writen in (rows,columns).
- **pool_size** (*(int, int)*) – tuple of length 2 factor by which to downscale in pooling layer. this is written in (rows,columns). the default value is (2,2), it will halve the input in each dimension.
- **filter_dilation** (*(int, int)*) – tuple of length 2 factor by which to subsample (stride) the convolutional layer input.
- **ignore_border** (*int | bool*) – integer or boolean 1 or True – (5, 5) input with pool_size = (2, 2), will generate a (2, 2) pooling layer output. 0 or False – (5, 5) input with pool_size = (2, 2), will generate a (3, 3) pooling layer output.
- **pool_stride** (*(int, int)*) – tuple of length 2 stride size, which is the number of shifts over rows/cols to get the next pool region. the default value is 0, it will set equal to pool_size, which means no overlap on pooling regions.
- **pool_padding** (*(int, int)*) – tuple of length 2 pad zeros to extend beyond four borders of the images. this is writen in (pad_h,pad_w), where pad_h is the size of the top and bottom margins, and pad_w is the size of the left and right margins.
- **mode** (*str*) – string pooling layer mode that excludes the padding in the computation. "max" – max pooling "sum" – sum pooling "avg" – average pooling "fmp" – fractional max pooling
- **activation** (*str*) – string activation function, e.g. "tanh", "sigmoid", "relu", "elu", "maxout", and so on.

- **factor** (float) – float factor by which scale the initial weights

recurrent = True

get_status(*sources*)

get_dim(*input*, *filters*, *pools*, *border_mode*, *stride*, *pool_stride*, *ignore_border*, *pad*)

calculate_index(*inputs*)

calculate_dropout(*dropout*, *inputs*)

convolution(*inputs*, *filter_shape*, *stride*, *border_mode*, *factor*, *pool_size*, *filter_dilation*)

pooling(*inputs*, *pool_size*, *ignore_border*, *stride*, *pad*, *mode*)

bias_term(*inputs*, *n_features*, *activation*)

run_cnn(*inputs*, *filter_shape*, *filter_dilation*, *params*, *modes*, *others*)

class NetworkCNNLayer.**NewConv**(***kwargs*)

layer_class = 'conv'
this class is for standard CNN and inception

class NetworkCNNLayer.**ConcatConv**(*padding=False*, ***kwargs*)

layer_class = 'conv_1d'
this class is for the CNN that processes an entire line image as the input by concatenated several frames by time axis.

class NetworkCNNLayer.**ResNet**(***kwargs*)

layer_class = 'resnet'
this class is for resnet connection.

NetworkCopyUtils

exception NetworkCopyUtils.**LayerDoNotMatchForCopy**

NetworkCopyUtils.**intelli_copy_layer**(*old_layer*, *new_layer*)

Copies from old_layer to new_layer.

We support slightly different param names. That can happen because the param names could encode the source/target layer number, e.g. named "hidden_N". Thus we need to translate the parameter names for the new network.

For the translation, we expect that a sorted list of the old output source layer names matches the related list of new output source layer names.

NetworkCtcLayer

NetworkCtcLayer.**log_add**(*a*, *b*)

NetworkCtcLayer.**log_sum**(*a*, *axis=None*, *keepdims=False*)

NetworkCtcLayer.**log_mul**(*a*, *b*)

NetworkCtcLayer.**log_div**(*a*, *b*)

NetworkCtcLayer.**log_path_probs**(*log_pcx_y*, *time_mask*, *seq_lens*, *forward=True*)
No blanks. Calculates the forward/backward probabilities.

Parameters

- **log_pcx_y** – softmax output for labels, log-space. shape (time,batch,seqlen) -> log prob
- **time_mask** – (time,batch) -> 0 or 1
- **seq_lens** – (batch,) -> seqlen

Returns log probabilities. shape (time,batch,seqlen)

NetworkCtcLayer.**ctc**(*log_pcx*, *time_mask*, *targets*, *seq_lens*)
No blanks. Calculates the CTC cost.

Parameters

- **log_pcx** – softmax output, log-space. shape (time,batch,label) -> log prob
- **time_mask** – (time,batch) -> 0 or 1
- **targets** – target seq, shape (seqlen,batch) -> label. seqlen <= time.
- **seq_lens** – (batch,) -> seqlen

Returns probs: (time,batch,seqlen) -> log prob

NetworkCtcLayer.**ctc_cost**(**args*, ***kwargs*)
:returns total negative log probability (scalar)

NetworkCtcLayer.**uniq_with_lengths**(*seq*, *time_mask*)

Parameters

- **seq** – (time,batch) -> label
- **time_mask** – (time,batch) -> 0 or 1

Returns out_seqs, seq_lens.

out_seqs is (max_seq_len,batch) -> label, where max_seq_len <= time. seq_lens is (batch,) -> len.

NetworkDescription

class NetworkDescription.**LayerNetworkDescription**(*num_inputs*, *num_outputs*, *hidden_info*, *output_info*, *default_layer_info*, *bidirectional=True*, *sharpgates='none'*, *truncation=-1*, *entropy=0*)
This class is used as a description to build up the LayerNetwork. The other options to build up a LayerNetwork are JSON or from a HDF model.

Parameters

- **hidden_info** (*list[dict[str]]*) – list of (layer_type, size, activation, name)
- **sharpgates** (*str*) – see LSTM layers
- **truncation** (*int*) – number of steps to use in truncated BPTT or -1. see theano.scan
- **entropy** (*float*) – ...

init_args()

copy()

classmethod from_config(*config*)

Return type *LayerNetworkDescription*

classmethod loss_from_config(*config*)

Return type str

classmethod tf_extern_data_types_from_config(*config*)

Parameters **config** (Config.Config) –

Returns dict data_key -> kwargs of Data

Return type dict[str,dict[str]]

classmethod num_inputs_outputs_from_config(*config*)

:returns (num_inputs, num_outputs), where num_inputs is like num_outputs["data"][0], and num_outputs is a dict of data_key -> (dim, ndim),

where data_key is e.g. "classes" or "data", dim is the feature dimension or the number of classes, and ndim is the ndim counted without batch-dim, i.e. ndim=1 means usually sparse data and ndim=2 means dense data.

Return type (*int*,dict[str,(*int*,*int*)])

to_json_content(*mask=None*)

Parameters **| str mask** (*None*) – mask

Return type dict

NetworkHiddenLayer

class NetworkHiddenLayer.**AdaptiveDepthLayer**(*eps=0.01*, *tau=0.01*, *bias=-1.0*, *damping='graves'*, ***kwargs*)

cost()

cost_scale()

layer_class = 'adaptive_depth'

class NetworkHiddenLayer.**AddZeroRowsLayer**(*row_index*, *number=1*, ***kwargs*)

layer_class = 'add_zero_rows'

class NetworkHiddenLayer.**AlignmentLayer**(*direction='inv'*, *tdps=None*, *nstates=1*, *nstep=1*, *min_skip=0*, *max_skip=30*, *search='align'*, *train_skips=False*, *base=None*, *output_attention=False*, *output_z=False*, *reduce_output=True*, *blank=False*, ***kwargs*)

cost()

errors()

layer_class = 'align'

class NetworkHiddenLayer.**AttentionLayer**(*base*, *conv_x=None*, *conv_y=None*, ***kwargs*)

layer_class = 'attention'

class NetworkHiddenLayer.**AttentionReshapeLayer**(*conf=0.3*, *pad=1*, *cap=1*, ***kwargs*)

layer_class = 'attention_reshape'

class NetworkHiddenLayer.**AttentionVectorLayer**(*base*, *template*, ***kwargs*)

cost()

layer_class = 'attention_vector'

class NetworkHiddenLayer.**BaseInterpolationLayer**(*base=None*, *method='softmax'*, *output_weights=False*, ***kwargs*)

layer_class = 'base'

class NetworkHiddenLayer.**BatchToTimeLayer**(*base*, ***kwargs*)

layer_class = 'batch_to_time'

class NetworkHiddenLayer.**BinOpLayer**(*op=None*, *n_out=None*, ***kwargs*)

static get_bin_op(*op*)

Return type theano.Op

layer_class = 'bin_op'

class NetworkHiddenLayer.**BlurLayer**(*ctx=5*, *p=1.0*, ***kwargs*)

RandomStreams
alias of MRG_RandomStreams

layer_class = 'blur'

rng = <theano.sandbox.rng_mrg.MRG_RandomStreams object>

class NetworkHiddenLayer.**CAlignmentLayer**(*direction='inv'*, *tdps=None*, *nstates=1*, *nstep=1*, *min_skip=1*, *max_skip=30*, *search='align'*, *train_skips=False*, *train_emission=False*, *clip_emission=1.0*, *base=None*, *coverage=0*, *output_z=False*, *reduce_output=True*, *blank=None*, *nil=None*, *focus='last'*, *mode='viterbi'*, ***kwargs*)

cost()

errors()

layer_class = 'calign'

class NetworkHiddenLayer.**CalcStepLayer**(*n_out=None*, *from_prev=''*, *apply=False*, *step=None*, *initial='zero'*, ***kwargs*)

layer_class = 'calc_step'

class NetworkHiddenLayer.**ChunkingLayer**(*chunk_size=1*, *method='concat'*, ***kwargs*)

layer_class = 'chunking'

class NetworkHiddenLayer.**ChunkingSublayer**(*n_out*, *sublayer*, *chunk_size*, *chunk_step*, *chunk_distribution='uniform'*, *add_left_context=0*, *add_right_context=0*, *normalize_output=True*, *trainable=False*, ***kwargs*)

cost()

layer_class = 'chunking_sublayer'

make_constraints()

recurrent = True

class NetworkHiddenLayer.**ClippingLayer**(*sparse_window=1*, ***kwargs*)

layer_class = 'clip'

class NetworkHiddenLayer.**ClusterDependentSubnetworkLayer**(*n_out*, *subnetwork*, *n_clusters*, *load='<random>'*, *data_map=None*, *trainable=True*, *concat_sources=True*, ***kwargs*)

Parameters

- **n_out** (*int*) – output dimension of output layer
- **network** (*dict[str,dict]*) – subnetwork as dict (JSON content)
- **data_map** (*list[str]*) – maps the sources (from) of the layer to data input. the list should be as long as the sources. default is ["data"], i.e. it expects one source and maps it as data in the subnetwork.
- **load** (*str*) – load string. filename but can have placeholders via str.format. Or "<random>" for no load.
- **trainable** (*bool*) – if we take over all params from the subnetwork

cost()

layer_class = 'clustersubnet'

make_constraints()

recurrent = True

update_cluster_target(*seq_tag*)

class NetworkHiddenLayer.**CollapseLayer**(*axis=0*, ***kwargs*)

layer_class = 'collapse'

class NetworkHiddenLayer.**ConcatBatchLayer**(***kwargs*)

layer_class = 'concat_batch'

class NetworkHiddenLayer.**ConstantLayer**(*value*, *n_out*, *dtype='float32'*, ***kwargs*)

layer_class = 'constant'

class NetworkHiddenLayer.**ConvPoolLayer**(*dx*, *dy*, *fx*, *fy*, ***kwargs*)

layer_class = 'convpool'

class NetworkHiddenLayer.**CopyLayer**(*activation=None*, ***kwargs*)
It's mostly the Identity function. But it will make sparse to non-sparse.

layer_class = 'copy'

class NetworkHiddenLayer.**CorruptionLayer**(*noise='gaussian'*, *p=0.0*, *clip=False*, ***kwargs*)

RandomStreams
alias of MRG_RandomStreams

layer_class = 'corruption'

rng = <theano.sandbox.rng_mrg.MRG_RandomStreams object>

class NetworkHiddenLayer.**DetectionLayer**(*label_idx*, ***kwargs*)

cost()

layer_class = 'detection'

class NetworkHiddenLayer.**DftLayer**(*dftLength=512*, *windowName='hamming'*, *flag_useSqrtWindow=False*, ***kwargs*)
This layer is applying the DFT of the input vector. The input is expected to be a segment of the time signal cut out with the rectangular function (so no windowing has been done) The output of the layer is the absolute values of the complex DFT coefficients. Only non negative coefficients are returned because of symmetric spectrum

layer_class = 'dft_layer_abs'

recurrent = True

class NetworkHiddenLayer.**DiscriminatorLayer**(*base=None*, *pgen=0.5*, *alpha=1*, *forge=False*, *ncritic=0*, *n_tmp=1*, *dynamic_scaling=False*, *error_scaling=False*, *loss='ce'*, ***kwargs*)

cost()

cost_scale()

errors()

layer_class = 'disc'

class NetworkHiddenLayer.**DownsampleLayer**(*factor*, *axis*, *method='average'*, *padding=False*, *sample_target=False*, *base=None*, ***kwargs*)
E.g. method == "average", axis == 0, factor == 2 -> each 2 time-frames are averaged. See TheanoUtil.downsample. You can also use method == "max".

layer_class = 'downsample'

class NetworkHiddenLayer.**DualStateLayer**(*acts='relu'*, *acth='tanh'*, ***kwargs*)

layer_class = 'dual'

class NetworkHiddenLayer.**DumpLayer**(*filename*, *with_grad=True*, *n_out=None*, ***kwargs*)

global_debug_container = None

layer_class = 'dump'

class NetworkHiddenLayer.**DuplicateIndexBatchLayer**(***kwargs*)

layer_class = 'duplicate_index_batch'

class NetworkHiddenLayer.**EmbeddingLayer**(***kwargs*)

layer_class = 'embedding'

class NetworkHiddenLayer.**EnergyNormalization**(***kwargs*)
This layer expects a (chunkted) time signal at the input. It normalizes the signal energy of the input chunk.

layer_class = 'energy_normalization_layer'

recurrent = True

class NetworkHiddenLayer.**ErrorsLayer**(*target*, ***kwargs*)

errors()

Return type theano.Variable

layer_class = 'errors'

class NetworkHiddenLayer.**FAlignmentLayer**(*direction='inv'*, *tdps=None*, *nstates=1*, *nstep=1*, *min_skip=1*, *max_skip=10*, *search='align'*, *train_skips=False*, *base=None*, *output_attention=False*, *output_z=False*, *reduce_output=True*, *blank=False*, *focus='last'*, *mode='viterbi'*, ***kwargs*)

layer_class = 'falign'

make_tdps(*tdps*, *max_skip*)

class NetworkHiddenLayer.**FStdAlignmentLayer**(*direction='inv'*, *base=None*, *nstates=3*, *skip_tdp=0*, ***kwargs*)

cost()

errors()

layer_class = 'fstdalign'

class NetworkHiddenLayer.**ForwardLayer**(*sparse_window=1*, ***kwargs*)

layer_class = 'hidden'

class NetworkHiddenLayer.**FrameConcatZeroLayer**(*num_frames*, *left=True*, ***kwargs*)
Concats zero at the start (left=True) or end in the time-dimension. I.e. you can e.g. delay the input by N frames. See also FrameConcatZeroLayer (frame_cutoff).

layer_class = 'frame_concat_zero'

class NetworkHiddenLayer.**FrameCutoffLayer**(*num_frames*, *left=True*, ***kwargs*)
Cutoffs frames at the start (left=True) or end in the time-dimension. You should use this when you used FrameConcatZeroLayer(frame_concat_zero).

layer_class = 'frame_cutoff'

class NetworkHiddenLayer.**GaussianFilter1DLayer**(*sigma*, *axis*, *window_radius=40*, ***kwargs*)

layer_class = 'gaussian_filter_1d'

recurrent = True

class NetworkHiddenLayer.**GenericCodeLayer**(*code*, *n_out*, ***kwargs*)

Parameters **code** (*str*) – generic Python code used for eval(). must return some output

layer_class = 'generic_code'

class NetworkHiddenLayer.**HDF5DataLayer**(*filename*, *dset*, ***kwargs*)

layer_class = 'hdf5'

recurrent = True

class NetworkHiddenLayer.**HiddenLayer**(*activation='sigmoid'*, ***kwargs*)

get_linear_forward_output(*with_bias=True*, *sources=None*)

class NetworkHiddenLayer.**IndexToVecLayer**(*n_out*, ***kwargs*)

layer_class = 'idx_to_vec'

class NetworkHiddenLayer.**InputBase**(***kwargs*)

layer_class = 'input_base'

class NetworkHiddenLayer.**InterpolationLayer**(*n_out*, ***kwargs*)

layer_class = 'interp'

class NetworkHiddenLayer.**InvAlignSegmentationLayer**(*window=0*, *win=20*, *base=None*, ***kwargs*)

layer_class = 'invalignsegment'

class NetworkHiddenLayer.**InvAlignSegmentationLayer2**(*window=0*, *win=20*, *base=None*, *join_states=False*, ***kwargs*)

find_diff_array(*att*)

layer_class = 'invalignsegment2'

set_yout(*y_out*, *diff*, *maxdiff*)

class NetworkHiddenLayer.**InvBacktrackLayer**(*direction='inv'*, *tdps=None*, *nstates=1*, *nstep=1*, *min_skip=1*, *max_skip=30*, *search='align'*, *train_skips=False*, *train_emission=False*, *clip_emission=1.0*, *base=None*, *coverage=0*, *output_z=False*, *reduce_output=True*, *blank=None*, *nil=None*, *focus='last'*, *mode='viterbi'*, ***kwargs*)

cost()

errors()

layer_class = 'ibt'

class NetworkHiddenLayer.**KernelLayer**(*kernel='gauss'*, *base=None*, *sigma=4.0*, ***kwargs*)

layer_class = 'kernel'

class NetworkHiddenLayer.**LengthLayer**(*min_len=0.0*, *max_len=1.0*, *use_real=0.0*, *err='ce'*, *oracle=False*, *pad=0*, ***kwargs*)

cost()

cost_scale()

layer_class = 'length'

class NetworkHiddenLayer.**LengthProjectionLayer**(*use_real=1.0*, *oracle=True*, *eval_oracle=False*, *pad=0*, *smo=0.0*, *avg=10.0*, *method='mapq'*, ***kwargs*)

cost()

cost_scale()

errors()

layer_class = 'length_projection'

class NetworkHiddenLayer.**LengthUnitLayer**(*min_len=1*, *max_len=32*, ***kwargs*)

layer_class = 'length_unit'

class NetworkHiddenLayer.**LinearCombLayer**(*n_out*, *n_comb*, *activation=None*, ***kwargs*)
Linear combination of each *n_comb* elements with bias.

layer_class = 'linear_comb'

class NetworkHiddenLayer.**LossLayer**(*loss*, *copy_input=None*, ***kwargs*)

Parameters

- **index** (*theano.Variable*) – index for batches
- **loss** (*str*) – e.g. 'ce'

layer_class = 'loss'

class NetworkHiddenLayer.**MfccLayer**(*dftSize=512*, *samplingFrequency=16000.0*, *fl=0*, *fh=None*, *nrOfFilters=40*, *nrOfMfccCoefficients=None*, ***kwargs*)
The source layer of this layer should be the DftLayer

batch_norm(*h*, *dim*, *use_shift=False*, *use_std=False*, *use_sample=0.0*, *force_sample=True*, *index=None*, ***kwargs*)
overwrite function from Layer to change default parameters of batch_norm: use_shift, use_std and foce_sample

getMfccFilterMatrix(*samplingFrequency*, *fl*, *fh*, *dftSize*, *nrOfFilters*, *flag_areaNormalized=0*)
returns the filter bank matrix used for the MFCCs For mathematical details see the book "speech language processing" by Huang et. al. pp. 314

#TBD !!! :type dftSize: int :param dftSize: size of dft :type nrOfFilters: int :param nrOfFilters: the number of filters used for the filterbank :type flag_areaNormalized: int :param flag_areaNormalized: flag that specifies which filter bank will be returned

0 - not normalized filter bank 1 - normalized filter bank where each filter covers an area of 1

invMelScale(*melVal*)
returns the respective value in the frequency domain

Parameters **melVal** (float) – value in mel domain

Return type *float*

layer_class = 'mfcc_layer'

melScale(*freq*)
returns the respective value on the mel scale

Parameters **freq** (float) – frequency value to transform onto mel scale

Return type *float*

recurrent = True

class NetworkHiddenLayer.**NativeLayer**(*n_out*, *native_class*, *params*, ***kwargs*)

layer_class = 'native'

recurrent = True

class NetworkHiddenLayer.**PolynomialExpansionLayer**(*n_degree*, *n_out=None*, ***kwargs*)

layer_class = 'polynomial_expansion'

class NetworkHiddenLayer.**Preemphasis**(*alpha=1.0*, ***kwargs*)
This layer is expecting a time signal as input and applying the preemphasis to the segment. (This is not completely correct application of preemphasis, since the first element of the segment does not know its predecessor in the time signal, therefore the effect is different than applying preemphasis on the complete signal beforehand)

layer_class = 'preemphasis_layer'

recurrent = True

class NetworkHiddenLayer.**RBFLayer**(*n_out*, ***kwargs*)
Use radial basis function.

layer_class = 'rbf'

class NetworkHiddenLayer.**RNNBlockLayer**(*num_layers=1*, *direction=0*, ***kwargs*)

cost()

errors()

layer_class = 'rnnblock'

recurrent = True

class NetworkHiddenLayer.**RandomRouteLayer**(*p=None*, *test_route=-1*, *n_out=None*, ***kwargs*)

layer_class = 'random_route'

class NetworkHiddenLayer.**RandomSelectionLayer**(*n_out*, ***kwargs*)

layer_class = 'random_selection'

class NetworkHiddenLayer.**RemoveRowsLayer**(*row_index*, *number=1*, ***kwargs*)

layer_class = 'remove_rows'

class NetworkHiddenLayer.**ReshapeLayer**(*base=None*, ***kwargs*)

layer_class = 'reshape'

class NetworkHiddenLayer.**ReverseAttentionLayer**(*base=None*, ***kwargs*)

layer_class = 'reverse_attention'

class NetworkHiddenLayer.**ReverseLayer**(***kwargs*)
Reverses the time-dimension.

layer_class = 'reverse'

class NetworkHiddenLayer.**RoutingLayer**(*base*, *p=0.5*, *oracle=False*, ***kwargs*)

cost()

cost_scale()

layer_class = 'signal_router'

class NetworkHiddenLayer.**ScaleGradLayer**(*scale=1.0*, *disconnect=False*, ***kwargs*)

layer_class = 'scale_grad'

class NetworkHiddenLayer.**ScaleGradientOp**(*scale*)

grad(*input*, *output_gradients*)

make_node(*x*)

perform(*node*, *inputs*, *output_storage*)

view_map = {0: [0]}

class NetworkHiddenLayer.**SegmentClassTargets**(*num_classes*, *window=15*, ***kwargs*)

class BuildClassesOp

itypes = (TensorType(int32, scalar), TensorType(int32, scalar), TensorType(int32, matrix), TensorType(int8, matri

otypes = (TensorType(float32, 3D), TensorType(int8, matrix))

perform(*node*, *inputs*, *output_storage*)

SegmentClassTargets.**layer_class = 'segment_class_targets'**

class NetworkHiddenLayer.**SegmentFinalStateLayer**(*base=None*, *use_full_label=False*, ***kwargs*)

layer_class = 'segfinal'

class NetworkHiddenLayer.**SegmentInputLayer**(*window=15*, ***kwargs*)

class ReinterpretCastOp

itypes = (TensorType(int32, matrix),)

otypes = (TensorType(float32, matrix),)

perform(*node*, *inputs*, *output_storage*)

SegmentInputLayer.**layer_class = 'segment_input'**

class NetworkHiddenLayer.**SegmentLayer**(***kwargs*)

layer_class = 'segment'

class NetworkHiddenLayer.**SharedForwardLayer**(*base=None*, *sparse_window=1*, ***kwargs*)

layer_class = 'hidden_shared'

class NetworkHiddenLayer.**SigmoidToTanhLayer**(***kwargs*)

layer_class = 'sigmoid_to_tanh'

class NetworkHiddenLayer.**SignalSplittingLayer**(*base*, *p=0.5*, *oracle=False*, ***kwargs*)

cost()

cost_scale()

layer_class = 'signal_splitter'

class NetworkHiddenLayer.**SignalValue**(*begin=0*, *sidx=0*, *reduce=False*, *copy_output=None*, ***kwargs*)

cost()

cost_scale()

errors()

layer_class = 'sigval'

class NetworkHiddenLayer.**SourceAttentionLayer**(*base*, *n_tmp=64*, ***kwargs*)

layer_class = 'source_attention'

class NetworkHiddenLayer.**SplitBatchLayer**(*n_parts=1*, *part=0*, ***kwargs*)

layer_class = 'split_batch'

class NetworkHiddenLayer.**StateAlignmentLayer**(*target*, *prior_scale=0.0*, ***kwargs*)

layer_class = 'state_alignment'

class NetworkHiddenLayer.**StateToAct**(*dual=False*, ***kwargs*)

layer_class = 'state_to_act'

class NetworkHiddenLayer.**StateVector**(*output_activation='identity'*, *idx=-1*, ***kwargs*)

layer_class = 'state_vector'

class NetworkHiddenLayer.**SubnetworkLayer**(*n_out*, *subnetwork*, *load='<random>'*, *data_map=None*, *trainable=True*, *concat_sources=True*, ***kwargs*)

Parameters

- **n_out** (int) – output dimension of output layer
- **network** (*dict[str,dict]*) – subnetwork as dict (JSON content)
- **data_map** (list*[str]*) – maps the sources (from) of the layer to data input. the list should be as long as the sources. default is ["data"], i.e. it expects one source and maps it as data in the subnetwork.
- **concat_sources** (bool) – if we concatenate all sources into one, like it is standard for most other layers
- **load** (*str*) – load string. filename but can have placeholders via str.format. Or "<random>" for no load.
- **trainable** (bool) – if we take over all params from the subnetwork

cost()

layer_class = 'subnetwork'

make_constraints()

recurrent = True

class NetworkHiddenLayer.**SumLayer**(***kwargs*)

layer_class = 'sum'

class NetworkHiddenLayer.**TanhToSigmoidLayer**(***kwargs*)

layer_class = 'tanh_to_sigmoid'

class NetworkHiddenLayer.**TimeBlurLayer**(*t_start*, *t_end*, *t_step*, *distribution*, ***kwargs*)

layer_class = 'time_blur'

recurrent = True

class NetworkHiddenLayer.**TimeChunkingLayer**(*n_out*, *chunk_size*, *chunk_step*, ***kwargs*)

layer_class = 'time_chunking'

class NetworkHiddenLayer.**TimeConcatLayer**(***kwargs*)

layer_class = 'time_concat'

class NetworkHiddenLayer.**TimeFlatLayer**(*chunk_size*, *chunk_step*, ***kwargs*)

layer_class = 'time_flat'

class NetworkHiddenLayer.**TimeShift**(*base=None*, *n_shift=1*, ***kwargs*)

layer_class = 'time_shift'

class NetworkHiddenLayer.**TimeToBatchLayer**(***kwargs*)

layer_class = 'time_to_batch'

class NetworkHiddenLayer.**TimeUnChunkingLayer**(*n_out*, *chunking_layer*, ***kwargs*)

layer_class = 'time_unchunking'

class NetworkHiddenLayer.**TimeWarpGlobalLayer**(*n_out=None*, *renorm_time=True*, *window_size=30*, *sigma2=0.5*, ***kwargs*)

Similar to *TimeWarpLayer* but different. This warp is cumulative and applied globally.

add_var_random_mat(*n*, *m*, *name*)

layer_class = 'time_warp_global'

recurrent = True

class NetworkHiddenLayer.**TimeWarpLayer**(*t_start*, *t_end*, *t_step*, *sigma*, *input_window*, *input_proj=None*, ***kwargs*)

Like https://en.wikipedia.org/wiki/Image_warping, controlled by NN. A bit like simple local feed-forward attention, where the attention is controlled by the input (encoder) and not output (decoder). Maybe similar: A Hybrid Dynamic Time Warping-Deep Neural Network Architecture for Unsupervised Acoustic Modeling, http://ewan.website/interspeech_2015_dnn_dtw.pdf Implementation is very similar to TimeBlurLayer except that the weight distribution is different every time frame and controlled by a NN. Note that this warp is applied locally. See also *TimeWarpGlobalLayer*.

layer_class = 'time_warp'

recurrent = True

class NetworkHiddenLayer.**TorchLayer**(*n_out*, *lua_fw_func*, *lua_bw_func*, *params*, *lua_file=None*, ***kwargs*)

layer_class = 'torch'

recurrent = True

class NetworkHiddenLayer.**TruncationLayer**(*n_trunc*, ***kwargs*)

layer_class = 'trunc'

class NetworkHiddenLayer.**UnsegmentInputLayer**(*original_output*, ***kwargs*)

class **UnsegmentInputOp**

itypes = (TensorType(float32, 3D), TensorType(int8, matrix))

otypes = (TensorType(float32, 3D),)

perform(*node*, *inputs*, *output_storage*)

UnsegmentInputLayer.**layer_class = 'unsegment_input'**

class NetworkHiddenLayer.**UpsampleLayer**(*factor*, *axis*, *time_like_last_source=False*, *method='nearest-neighbor'*, ***kwargs*)

layer_class = 'upsample'

class NetworkHiddenLayer.**WindowContextLayer**(*window*, *average='uniform'*, *scan=False*, *n_out=None*, ***kwargs*)

layer_class = 'window_context'

class NetworkHiddenLayer.**WindowLayer**(*window*, *delta=0*, *delta_delta=0*, ***kwargs*)

layer_class = 'window'

NetworkHiddenLayer.**concat_sources**(*sources*, *masks=None*, *mass=None*, *unsparse=False*, *expect_source=True*)

Parameters

- **unsparse** (bool) – whether to make sparse sources into 1-of-k
- **expect_source** (bool) – whether to throw an exception if there is no source

:returns (concatenated sources, out dim) :rtype: (theano.Variable, int)

NetworkLayer

NetworkLayer.**get_layer_class**(*name*, *raise_exception=True*)

Return type type(*NetworkHiddenLayer.HiddenLayer*)

NetworkLstmLayer

class NetworkLstmLayer.**RecurrentLayer**(*reverse=False*, *truncation=-1*, *compile=True*, *projection=0*, *sampling=1*, ***kwargs*)

recurrent = True

layer_class = 'recurrent'

compile()

create_recurrent_weights(*n*, *m*)

class NetworkLstmLayer.**LstmLayer**(*n_out*, *sharpgates='none'*, ***kwargs*)

layer_class = 'lstm'

class NetworkLstmLayer.**OptimizedLstmLayer**(*n_out*, *sharpgates='none'*, *encoder=None*, *n_dec=0*, ***kwargs*)

layer_class = 'lstm_opt'

get_branching()

get_energy()

make_constraints()

class NetworkLstmLayer.**SimpleLstmLayer**(*n_out*, *sharpgates='none'*, *encoder=None*, *n_dec=0*, ***kwargs*)

layer_class = 'lstm_simple'

NetworkLstmLayer.**make_lstm_step**(*n_cells*, *W_re*, *W_out_proj=None*, *W_re_proj=None*, *W_peep_i=None*, *W_peep_f=None*, *W_peep_o=None*, *grad_clip=None*, *CI=None*, *CO=None*, *G=None*)

NetworkLstmLayer.**lstm**(*z*, *i*, *W_re*, *W_out_proj=None*, *W_re_proj=None*, *W_peep_i=None*, *W_peep_f=None*, *W_peep_o=None*, *CI=None*, *CO=None*, *G=None*, *grad_clip=None*, *direction=1*)

class NetworkLstmLayer.**Lstm2Layer**(*n_out*, *n_cells=None*, *n_proj=None*, *peepholes=False*, *direction=1*, *activation=None*, *grad_clip=None*, *truncation=None*, ***kwargs*)

recurrent = True

layer_class = 'lstm2'

class NetworkLstmLayer.**Lstm3Layer**(*n_out*, *direction=1*, *grad_clip=None*, ***kwargs*)
Like lstm2 but even simpler.

recurrent = True

layer_class = 'lstm3'

class NetworkLstmLayer.**LayerNormLstmLayer**(*n_out*, *direction=1*, *grad_clip=None*, ***kwargs*)
Layer Normalization, https://arxiv.org/abs/1607.06450

recurrent = True

layer_class = 'ln_lstm'

class NetworkLstmLayer.**NativeLstmLayer**(*n_out*, *direction=1*, *truncation=None*, ***kwargs*)

recurrent = True

layer_class = 'native_lstm'

class NetworkLstmLayer.**GenericLstmLayer**(*n_out*, *sublayer*, *out_sublayer=None*, *n_cells=None*, *activation=None*, *direction=1*, *grad_clip=None*, *truncation=None*, ***kwargs*)
LSTM implementation which allows a custom input+recurrent function (n_in + n_out -> n_cells * 4) and a custom output function (n_cells -> n_out) which is identity by default. You specify it as a sub layer.

recurrent = True

layer_class = 'generic_lstm'

class NetworkLstmLayer.**AssociativeLstmLayer**(*n_out*, *n_copies*, *activation='tanh'*, *direction=1*, *grad_clip=None*, ***kwargs*)
Associative Long Short-Term Memory http://arxiv.org/abs/1602.03032

recurrent = True

layer_class = 'associative_lstm'

class NetworkLstmLayer.**LstmHalfGatesLayer**(*n_out*, *direction=1*, *activation='tanh'*, *grad_clip=None*, ***kwargs*)

recurrent = True

layer_class = 'lstm_half_gates'

class NetworkLstmLayer.**LstmProjGatesLayer**(*n_out*, *n_gate_proj*, *direction=1*, *activation='relu'*, *grad_clip=None*, ***kwargs*)

recurrent = True

layer_class = 'lstm_proj_gates'

class NetworkLstmLayer.**LstmComplexLayer**(*n_out*, *direction=1*, *activation='tanh'*, *use_complex='1:1:1:1'*, *grad_clip=None*, ***kwargs*)

recurrent = True

layer_class = 'lstm_complex'

class NetworkLstmLayer.**ActLstmLayer**(*n_out*, *n_max_calc_steps=10*, *time_penalty=0.01*, *time_penalty_type='linear_p'*, *total_halt_penalty=0.0*, *total_halt_penalty_type='inv'*, *direction=1*, *eps=0.01*, *grad_clip=None*, *unroll_inner_scan=False*, ***kwargs*)

Adaptive Computation Time for Recurrent Neural Networks, Graves

recurrent = True

layer_class = 'act_lstm'

class NetworkLstmLayer.**GRULayer**(*n_out*, *encoder=None*, *mode='cho'*, *n_dec=0*, ***kwargs*)

layer_class = 'gru'

class NetworkLstmLayer.**SRULayer**(*n_out*, *encoder=None*, *psize=0*, *pact='relu'*, *pdepth=1*, *carry_time=False*, *n_dec=0*, ***kwargs*)

layer_class = 'sru'

class NetworkLstmLayer.**SRALayer**(*n_out*, *encoder=None*, *psize=0*, *pact='relu'*, *pdepth=1*, *n_dec=0*, ***kwargs*)

layer_class = 'sra'

NetworkOutputLayer

class NetworkOutputLayer.**OutputLayer**(*loss*, *y*, *dtype=None*, *reshape_target=False*, *copy_input=None*, *copy_output=None*, *time_limit=0*, *use_source_index=False*, *auto_fix_target_length=False*, *sigmoid_outputs=False*, *exp_outputs=False*, *gauss_outputs=False*, *activation=None*, *prior_scale=0.0*, *log_prior=None*, *use_label_priors=0*, *compute_priors_via_baum_welch=False*, *compute_priors=False*, *compute_priors_exp_average=0*, *compute_priors_accumulate_batches=None*, *compute_distortions=False*, *softmax_smoothing=1.0*, *grad_clip_z=None*, *grad_discard_out_of_bound_z=None*, *normalize_length=False*, *exclude_labels=[]*, *apply_softmax=True*, *substract_prior_from_output=False*, *input_output_similarity=None*, *input_output_similarity_scale=1*, *copy_weights=False*, ***kwargs*)

Parameters

- **index** (*theano.Variable*) – index for batches
- **loss** (*str*) – e.g. 'ce'

layer_class = 'softmax'

create_bias(*n*, *prefix='b'*, *name=''*)

entropy()

Return type theano.Variable

errors()

Return type theano.Variable

class NetworkOutputLayer.**FramewiseOutputLayer**(*loss*, *y*, *dtype=None*, *reshape_target=False*, *copy_input=None*, *copy_output=None*, *time_limit=0*, *use_source_index=False*, *auto_fix_target_length=False*, *sigmoid_outputs=False*, *exp_outputs=False*, *gauss_outputs=False*, *activation=None*, *prior_scale=0.0*, *log_prior=None*, *use_label_priors=0*, *compute_priors_via_baum_welch=False*, *compute_priors=False*, *compute_priors_exp_average=0*, *compute_priors_accumulate_batches=None*, *compute_distortions=False*, *softmax_smoothing=1.0*, *grad_clip_z=None*, *grad_discard_out_of_bound_z=None*, *normalize_length=False*, *exclude_labels=[]*, *apply_softmax=True*, *substract_prior_from_output=False*, *input_output_similarity=None*, *input_output_similarity_scale=1*, *copy_weights=False*, ***kwargs*)

Parameters

- **index** (*theano.Variable*) – index for batches
- **loss** (*str*) – e.g. 'ce'

cost()

Return type (theano.Variable | None, dict[theano.Variable,theano.Variable] | None)

Returns cost, known_grads

class NetworkOutputLayer.**DecoderOutputLayer**(***kwargs*)

cost()

class NetworkOutputLayer.**SequenceOutputLayer**(*ce_smoothing=0.0*, *ce_target_layer_align=None*, *am_scale=1*, *gamma=1*, *bw_norm_class_avg=False*, *fast_bw_opts=None*, *seg_fast_bw_opts=None*, *loss_like_ce=False*, *trained_softmax_prior=False*, *sprint_opts=None*, *warp_ctc_lib=None*, ***kwargs*)

index_for_ctc()

output_index()

cost()

Parameters **y** – shape (time*batch,) -> label

Returns error scalar, known_grads dict

errors()

class NetworkOutputLayer.**UnsupervisedOutputLayer**(*base*, *momentum=0.1*, *oracle=False*, *msteps=100*, *esteps=200*, ***kwargs*)

cost()

errors()

Return type theano.Variable

NetworkRecurrentLayer

class NetworkRecurrentLayer.**Unit**(*n_units*, *n_in*, *n_out*, *n_re*, *n_act*)

Abstract descriptor class for all kinds of recurrent units.

Parameters

- **n_units** – number of cells
- **n_in** – cell fan in
- **n_out** – cell fan out
- **n_re** – recurrent fan in
- **n_act** – number of outputs

set_parent(*parent*)

scan(*x*, *z*, *non_sequences*, *i*, *outputs_info*, *W_re*, *W_in*, *b*, *go_backwards=False*, *truncate_gradient=-1*)
Executes the iteration over the time axis (usually with theano.scan) :param step: python function to be executed :param x: unmapped input tensor in (time,batch,dim) shape :param z: same as x but already transformed to self.n_in :param non_sequences: see theano.scan :param i: index vector in (time, batch) shape :param outputs_info: see theano.scan :param W_re: recurrent weight matrix :param W_in: input weight matrix :param b: input bias :param go_backwards: whether to scan the sequence from 0 to T or from T to 0 :param truncate_gradient: see theano.scan :return:

scan_seg(*x*, *z*, *att*, *non_sequences*, *i*, *outputs_info*, *W_re*, *W_in*, *b*, *go_backwards=False*, *truncate_gradient=-1*)
Executes the iteration over the time axis (usually with theano.scan) :param step: python function to be executed :param x: unmapped input tensor in (time,batch,dim) shape :param z: same as x but already transformed to self.n_in :param non_sequences: see theano.scan :param i: index vector in (time, batch) shape :param outputs_info: see theano.scan :param W_re: recurrent weight matrix :param W_in: input weight matrix :param b: input bias :param go_backwards: whether to scan the sequence from 0 to T or from T to 0 :param truncate_gradient: see theano.scan :return:

class NetworkRecurrentLayer.**VANILLA**(*n_units*, ***kwargs*)
A simple tanh unit

step(*i_t*, *x_t*, *z_t*, *z_p*, *h_p*)
performs one iteration of the recursion :param i_t: index at time step t :param x_t: raw input at time step t :param z_t: mapped input at time step t :param z_p: previous input from time step t-1 :param h_p: previous hidden activation from time step t-1 :return:

class NetworkRecurrentLayer.**LSTME**(*n_units*, ***kwargs*)
A theano based LSTM implementation

step(*i_t*, *x_t*, *z_t*, *y_p*, *c_p*, **other_args*)

class NetworkRecurrentLayer.**LSTMS**(*n_units*, ***kwargs*)
A theano based LSTM implementation

step(*i_t*, *x_t*, *z_t*, *att_p*, *y_p*, *c_p*, **other_args*)

class NetworkRecurrentLayer.**LEAKYLSTM**(*n_units*, ***kwargs*)
A 1D cell proposed in http://jmlr.org/papers/volume17/14-203/14-203.pdf The simplified equations can be seen in Table 7, page 36. Type A with gamma_3==0. This cell has 3 units instead of 4 like LSTM

step(*i_t*, *x_t*, *z_t*, *y_p*, *c_p*, **other_args*)

class NetworkRecurrentLayer.**LEAKYLPLSTM**(*n_units*, ***kwargs*)
A 1D cell proposed in http://jmlr.org/papers/volume17/14-203/14-203.pdf The simplified equations can be seen in Table 7, page 36. Type A. This cell has 4 units like the LSTM

step(*i_t*, *x_t*, *z_t*, *y_p*, *c_p*, **other_args*)

class NetworkRecurrentLayer.**PIDLSTM**(*n_units*, ***kwargs*)
A 1D cell proposed in http://jmlr.org/papers/volume17/14-203/14-203.pdf The simplified equations can be seen in Table 7, page 36. Type E. This cell works as a dynamic PID filter of the input. The forget gate determines if it has PD od PI characteristic, the Proportional gate gates the P/I part, the Difference gate the D/P part. It can have advantages if there is no subsampling in the layer. This cell has 4 units like the LSTM

step(*i_t*, *x_t*, *z_t*, *y_p*, *c_p*, **other_args*)

class NetworkRecurrentLayer.**LSTMP**(*n_units*, ***kwargs*)
Very fast custom LSTM implementation

scan(*x*, *z*, *non_sequences*, *i*, *outputs_info*, *W_re*, *W_in*, *b*, *go_backwards=False*, *truncate_gradient=-1*)

class NetworkRecurrentLayer.**LSTMPS**(*n_units*, ***kwargs*)
Very fast custom LSTM implementation for segment encoding

scan_seg(*x*, *z*, *non_sequences*, *i*, *att*, *outputs_info*, *W_re*, *W_in*, *b*, *go_backwards=False*, *truncate_gradient=-1*)

class NetworkRecurrentLayer.**LSTMB**(*n_units*, ***kwargs*)
Very fast custom BLSTM implementation

scan(*x*, *z*, *non_sequences*, *i*, *outputs_info*, *W_re*, *W_in*, *b*, *go_backwards=False*, *truncate_gradient=-1*)

NetworkRecurrentLayer.**BLSTM**
alias of *LSTMB*

class NetworkRecurrentLayer.**LSTMC**(*n_units*, ***kwargs*)
The same implementation as above, but it executes a theano function (recurrent transform) in each iteration. This allows for additional dependencies in the recursion of the LSTM.

scan(*x*, *z*, *non_sequences*, *i*, *outputs_info*, *W_re*, *W_in*, *b*, *go_backwards=False*, *truncate_gradient=-1*)

class NetworkRecurrentLayer.**LSTMR**(*n_units*, ***kwargs*)
Same as LSTMC but without recurrent matrix multiplication

scan(*x*, *z*, *non_sequences*, *i*, *outputs_info*, *W_re*, *W_in*, *b*, *go_backwards=False*, *truncate_gradient=-1*)

class NetworkRecurrentLayer.**GRU**(*n_units*, ***kwargs*)
Gated recurrent unit as described in http://arxiv.org/abs/1502.02367

step(*i_t*, *x_t*, *z_t*, *z_p*, *h_p*)

class NetworkRecurrentLayer.**SRU**(*n_units*, ***kwargs*)
Same as GRU but without reset weights, which allows for a faster computation on GPUs

step(*i_t*, *x_t*, *z_t*, *z_p*, *h_p*)

class NetworkRecurrentLayer.**RecurrentUnitLayer**(*n_out=None*, *n_units=None*, *direction=1*, *truncation=-1*, *sampling=1*, *encoder=None*, *unit='lstm'*, *n_dec=0*, *attention='none'*, *recurrent_transform='none'*, *recurrent_transform_attribs='{}'*, *attention_template=128*, *attention_distance='l2'*, *attention_step='linear'*, *attention_beam=0*, *attention_norm='exp'*, *attention_momentum='none'*, *attention_sharpening=1.0*, *attention_nbest=0*, *attention_store=False*, *attention_smooth=False*, *attention_glimpse=1*, *attention_filters=1*, *attention_accumulator='sum'*, *attention_loss=0*, *attention_bn=0*, *attention_lm='none'*, *attention_ndec=1*, *attention_memory=0*, *base=None*, *aligner=None*, *lm=False*, *force_lm=False*, *droplm=1.0*, *forward_weights_init=None*, *bias_random_init_forget_shift=0.0*, *copy_weights_from_base=False*, *segment_input=False*, *join_states=False*, *sample_segment=None*, ***kwargs*)
Layer class to execute recurrent units

Parameters

- **n_out** – number of cells
- **n_units** – used when initialized via Network.from_hdf_model_topology

- **direction** – process sequence in forward (1) or backward (-1) direction
- **truncation** – gradient truncation
- **sampling** – scan every nth frame only
- **encoder** – list of encoder layers used as initalization for the hidden state
- **unit** – cell type (one of 'lstm', 'vanilla', 'gru', 'sru')
- **n_dec** – absolute number of steps to unfold the network if integer, else relative number of steps from encoder
- **recurrent_transform** – name of recurrent transform
- **recurrent_transform_attribs** – dictionary containing parameters for a recurrent transform
- **attention_template** –
- **attention_distance** –
- **attention_step** –
- **attention_beam** –
- **attention_norm** –
- **attention_sharpening** –
- **attention_nbest** –
- **attention_store** –
- **attention_align** –
- **attention_glimpse** –
- **attention_lm** –
- **base** – list of layers which outputs are considered as based during attention mechanisms
- **lm** – activate RNNLM
- **force_lm** – expect previous labels to be given during testing
- **droplm** – probability to take the expected output as predecessor instead of the real one when LM=true
- **bias_random_init_forget_shift** – initialize forget gate bias of lstm networks with this value

recurrent = True

layer_class = 'rec'

cost()

Return type (theano.Variable | None, dict[theano.Variable,theano.Variable] | None)

Returns cost, known_grads

create_seg_wise_encoder_output(*att*, *aligner=None*)

class NetworkRecurrentLayer.**RecurrentUpsampleLayer**(*factor*, ***kwargs*)

layer_class = 'recurrent_upsample'

class NetworkRecurrentLayer.**LinearRecurrentLayer**(*n_out*, *direction=1*, ***kwargs*)
Inspired from: http://arxiv.org/abs/1510.02693 Basically a very simple LSTM.

recurrent = True

layer_class = 'linear_recurrent'

NetworkStream

class NetworkStream.**NetworkStream**(*name*, *port*, *cache_size=100*)

class ThreadingServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
Constructor. May be extended, do not override.

NetworkStream.**count**()

NetworkStream.**data**(*start_index*, *count=1*)

NetworkStream.**update**(*task*, *data*, *tags=[]*)

NetworkTwoDLayer

class NetworkTwoDLayer.**TwoDBaseLayer**(*n_out*, ***kwargs*)

create_xavier_weights(*shape*, *name*)

class NetworkTwoDLayer.**OneDToTwoDLayer**(***kwargs*)

layer_class = '1Dto2D'

recurrent = False

class NetworkTwoDLayer.**OneDToTwoDFixedSizeLayer**(*pad_x=0*, *pad_y=0*, *d_row=-1*, ***kwargs*)

layer_class = '1Dto2D_fixed_size'

recurrent = True

class NetworkTwoDLayer.**TwoDToOneDLayer**(*collapse='mean'*, *maxout=False*, *transpose=False*, ***kwargs*)

layer_class = '2Dto1D'

recurrent = False

class NetworkTwoDLayer.**DeepLSTM**(*n_out*, *depth*, ***kwargs*)

layer_class = 'deep_lstm'

recurrent = True

create_and_add_2d_lstm_weights(*n*, *m*, *name_suffix*)

create_and_add_bias(*n_cells*, *name_suffix*)

class NetworkTwoDLayer.**TwoDLSTMLayer**(*n_out*, *collapse_output=False*, *directions=4*, *projection='average'*, *base=None*, ***kwargs*)

layer_class = 'mdlstm'

recurrent = True

create_and_add_2d_lstm_weights(*n*, *m*, *name_suffix*)

create_and_add_bias(*n_cells*, *name_suffix*)

NetworkTwoDLayer.**conv_crop_pool_op**(*X*, *sizes*, *output_sizes*, *W*, *b*, *n_in*, *n_maps*, *filter_height*, *filter_width*, *filter_dilation*, *poolsize*)

class NetworkTwoDLayer.**ConvBaseLayer**(*n_features*, *filter*, *base=None*, *activation='tanh'*, ***kwargs*)

layer_class = 'conv_base'

recurrent = False

create_conv_weights(*n_features*, *n_in*, *filter_height*, *filter_width*, *name_suffix=''*)

create_and_add_bias(*n_out*, *name_suffix=''*)

conv_output_size_from_input_size(*sizes*)

NetworkTwoDLayer.**maybe_print_pad_warning**(*_*, *x*)

class NetworkTwoDLayer.**ConvPoolLayer2**(*pool_size*, *filter_dilation=None*, *padding=False*, ***kwargs*)

layer_class = 'conv2'

recurrent = True

output_size_from_input_size(*sizes*)

class NetworkTwoDLayer.**ConvFMPLayer**(*factor=1.4142135623730951*, *decay=1.0*, *min_factor=None*, *padding=False*, ***kwargs*)

layer_class = 'conv_fmp'

recurrent = False

NormalizationData

class NormalizationData.**NormalizationData**(*normalizationFilePath*)

This class holds normalization data for inputs and outputs. It also contains methods to create the normalization HDF file.

Reads normalization data from the given HDF file and saves it into the member variables.

Parameters **normalizationFilePath** (*str*) – path to the HDF file with normalization data.

GROUP_INPUTS = 'inputs'

GROUP_OUTPUTS = 'outputs'

DATASET_MEAN = 'mean'

DATASET_MEAN_OF_SQUARES = 'meanOfSquares'

DATASET_VARIANCE = 'variance'

DATASET_TOTAL_FRAMES = 'totalNumberOfFrames'

DATASET_TIME_DIMENSION_INDEX = 0

DATASET_FEATURE_DIMENSION_INDEX = 1

SUMMATION_PRECISION = 1e-05

static createNormalizationFile(*bundleFilePath*, *outputFilePath*, *dtype=<type 'numpy.float64'>*, *flag_includeOutputs=True*)

Calculates means over inputs and outputs of datasets in the HDF files described by the given bundle file.

See BundleFile.BundleFile

Each HDF dataset file is expected to have the following groups:

•NormalizationData.GROUP_INPUTS (the group for the input data)

•NormalizationData.GROUP_OUTPUTS (the group for the output data)

Each group may have datasets. Each dataset is expected to have shape (time frames, features). E.g. (267, 513) – 267 time frames each containing a feature vector of dimensionality 513.

The method writes results into the given output file. Availability of means and variances depends on whether the corresponding groups are available in the input dataset HDF files.

!!! IMPORTANT !!! General rule of thumb: if one dataset file has both input and output groups then you should make sure that all the dataset files have them. Otherwise means and variance will not be correct. It is OK if *all* the datasets have only the input group. In this case means and variance only for inputs will be calculated.

Parameters

- **bundleFilePath** (*str*) – path to the bundle file. :see: BundleFile.BundleFile
- **outputFilePath** (*str*) – path to the output HDF normalization file.
- **dtype** (*numpy.dtype*) – type of data to use during calculations.
- **flag_includeOutputs** (bool) – if True then normalization data will be calculated for outputs (targets) as well.

inputMean

Mean of the input data.

Return type numpy.ndarray | None

Returns Mean of the input data if it is available or None otherwise.

inputVariance

Variance of the input data.

Return type numpy.ndarray | None

Returns Variance of the input data if it is available or None otherwise.

outputMean

Mean of the output data.

Return type numpy.ndarray | None

Returns Mean of the output data if it is available or None otherwise.

outputVariance

Variance of the output data.

Return type numpy.ndarray | None

Returns Variance of the output data if it is available or None otherwise.

NumpyDumpDataset

class NumpyDumpDataset.**NumpyDumpDataset**(*prefix*, *postfix='.txt.gz'*, *start_seq=0*, *end_seq=None*, *num_inputs=None*, *num_outputs=None*, ***kwargs*)

file_format_data = '%i.data'

file_format_targets = '%i.targets'

init_seq_order(*epoch=None*, *seq_list=None*)

get_input_data(*seq_idx*)

get_targets(*target*, *seq_idx*)

get_ctc_targets(*seq_idx*)

get_seq_length(*seq_idx*)

num_seqs

len_info()

OpBLSTM

class OpBLSTM.**BLSTMOpGrad**(*inplace*)

make_node(*V_f*, *V_b*, *c_f*, *c_b*, *idx_f*, *idx_b*, *Dd_f*, *Dd_b*, *DY_f*, *DY_b*, *Y_f*, *Y_b*, *H_f*, *H_b*)

infer_shape(*node*, *input_shapes*)

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

c_code_cache_version()

class OpBLSTM.**BLSTMOp**(*inplace*)

make_node(*Z_f*, *Z_b*, *V_f*, *V_b*, *c_f*, *c_b*, *i_f*, *i_b*)

Parameters

- **Z_f** – {input,output,forget} gate + cell state forward. 3d (time,batch,dim*4)
- **Z_b** – {input,output,forget} gate + cell state backward. 3d (time,batch,dim*4)
- **V_f** – forward recurrent matrix. 2d (dim,dim*4)
- **V_b** – backward recurrent matrix. 2d (dim,dim*4)
- **c_f** – initial forward cell state. 2d (batch,dim)
- **c_b** – initial backward cell state. 2d (batch,dim)
- **i** – index. 2d (time,batch) -> 0 or 1

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

c_code_cache_version()

OpInvAlign

class OpInvAlign.**InvAlignOp**(*tdps*, *nstates*)

itypes = [TensorType(int8, matrix), TensorType(int8, matrix), TensorType(float32, 3D), TensorType(int32, matrix)]

otypes = [TensorType(int32, matrix), TensorType(int32, matrix), TensorType(int8, matrix)]

perform(*node*, *inputs_storage*, *output_storage*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

class OpInvAlign.**InvFullAlignOp**(*tdps*, *nstates*)

itypes = [TensorType(int8, matrix), TensorType(int8, matrix), TensorType(float32, 3D), TensorType(int32, matrix)]

otypes = [TensorType(float32, 3D), TensorType(int8, matrix)]

perform(*node*, *inputs_storage*, *output_storage*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

class OpInvAlign.**InvBacktrackOp**(*tdps*, *nstates*, *penalty*)

itypes = [TensorType(int8, matrix), TensorType(float32, 3D), TensorType(float32, 3D)]

otypes = [TensorType(int32, matrix), TensorType(int32, matrix), TensorType(int8, matrix)]

perform(*node*, *inputs_storage*, *output_storage*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

class OpInvAlign.**InvDecodeOp**(*tdps*, *nstates*, *penalty*)

itypes = [TensorType(int8, matrix), TensorType(float32, 3D)]

otypes = [TensorType(int32, matrix), TensorType(int32, matrix), TensorType(int8, matrix)]

perform(*node*, *inputs_storage*, *output_storage*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

OpLSTM

class OpLSTM.**LSTMOpGrad**(*inplace*)

make_node(*V_h*, *c*, *idx*, *Dd*, *DY*, *Y*, *H*)

infer_shape(*node*, *input_shapes*)

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

c_code_cache_version()

class OpLSTM.**LSTMOp**(*inplace*)

make_node(*Z*, *V_h*, *c*, *i*)

Parameters

- **Z** – {input,output,forget} gate + cell state. 3d (time,batch,dim*4)
- **V_h** – recurrent matrix. 2d (dim,dim*4)
- **c** – initial cell state. 2d (batch,dim)
- **i** – index. 2d (time,batch) -> 0 or 1

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

c_code_cache_version()

class OpLSTM.**LSTMSOp**(*inplace*)

make_node(*Z*, *V_h*, *c*, *i*, *att*)

Parameters

- **Z** – {input,output,forget} gate + cell state. 3d (time,batch,dim*4)
- **V_h** – recurrent matrix. 2d (dim,dim*4)
- **c** – initial cell state. 2d (batch,dim)
- **i** – index. 2d (time,batch) -> 0 or 1
- **att** – attention from inverted alignment layer

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

c_code_cache_version()

OpLSTMCell

class OpLSTMCell.**LSTMOpCellGrad**(*inplace*)

make_node(*V_h*, *c*, *idx*, *Dd*, *DY*, *Y*, *H*)

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

class OpLSTMCell.**LSTMOpCell**(*inplace*)

make_node(*Z*, *V_h*, *c*, *i*)

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

OpLSTMCustom

class OpLSTMCustom.**LSTMCustomOpGrad**(*fun_name*, *inplace*, *recurrent_transform*)

make_node(*Y*, *H*, *c*, *y0*, *i*, *freq*, *Dd*, *DY*, *W_re*, **args*)

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

class OpLSTMCustom.**LSTMCustomOp**(*fun_name*, *inplace*, *recurrent_transform*)

make_node(*Z*, *c*, *y0*, *i*, *freq*, *W_re*, **args*)

Parameters

- **Z** – {input,output,forget} gate + cell state. 3d (time,batch,dim*4)
- **c** – initial cell state. 2d (batch,dim)
- **y0** – output of t = -1 (for recursion at t = 0). 2d (batch,dim)
- **i** – index. 2d (time,batch) -> 0 or 1
- **W_re** – recurrent matrix. 2d (dim,dim*4)
- **freq** – call frequency to custom function. int
- **args** – custom_inputs + initial_state_vars: other inputs for the custom function

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

grad(*inputs*, *output_grads*)

OpLSTMCustom.**register_func**(*recurrent_transform*)

OpLSTMRec

class OpLSTMRec.**LSTMRecOpGrad**(*fun_name*, *inplace*, *recurrent_transform*)

make_node(*Y*, *H*, *c*, *y0*, *i*, *Dd*, *DY*, **args*)

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

class OpLSTMRec.**LSTMRecOp**(*fun_name*, *inplace*, *recurrent_transform*)

make_node(*Z*, *c*, *y0*, *i*, **args*)

Parameters

- **Z** – {input,output,forget} gate + cell state. 3d (time,batch,dim*4)
- **c** – initial cell state. 2d (batch,dim)
- **y0** – output of t = -1 (for recursion at t = 0). 2d (batch,dim)
- **i** – index. 2d (time,batch) -> 0 or 1
- **args** – custom_inputs + initial_state_vars: other inputs for the custom function

c_support_code()

c_code(*node*, *name*, *input_names*, *output_names*, *sub*)

grad(*inputs*, *output_grads*)

OpLSTMRec.**register_func**(*recurrent_transform*)

OpNumpyAlign

class OpNumpyAlign.**NumpyAlignOp**(*inverse*)

itypes = [TensorType(int8, matrix), TensorType(int8, matrix), TensorType(float32, 3D), TensorType(int32, matrix)]

otypes = [TensorType(int32, matrix)]

perform(*node*, *inputs_storage*, *output_storage*)

grad(*inputs*, *output_grads*)

infer_shape(*node*, *input_shapes*)

Pretrain

class Pretrain.**Pretrain**(*original_network_json*, *network_init_args*, *copy_output_layer=None*, *greedy=None*, *repetitions=None*, *construction_algo=None*)

Start with 1 hidden layers up to N hidden layers -> N pretrain steps -> N epochs (with repetitions == 1). The first hidden layer is the input layer. This works for generic network constructions. See _construct_epoch().

Parameters

- **network_init_args** (*dict[str]*) – additional args we use for LayerNetwork.from_json(). must have n_in, n_out.
- **copy_output_layer** (*bool|str*) – whether to copy the output layer params from last epoch or reinit
- **greedy** (bool) – if True, only train output+last layer, otherwise train all

- **| int | list[int] | dict repetitions** (*None*) – how often to repeat certain pretrain steps. default is one epoch. It can also be a dict, with keys like 'default' and 'final'. See code below.
- **construction_algo** (*str*) – e.g. "from_output"

copy_params_from_old_network(*new_network*, *old_network*)

:returns the remaining hidden layer names which exist only in the new network. :rtype: set[str]

get_final_network_json()

get_network_for_epoch(*epoch*, *mask=None*)

Return type *Network.LayerNetwork*

get_network_json_for_epoch(*epoch*)

Parameters **epoch** (int) – starting at 1

Return type dict[str]

get_train_num_epochs()

get_train_param_args_for_epoch(*epoch*)

:returns the kwargs for LayerNetwork.set_train_params, i.e. which params to train. :rtype: dict[str]

class Pretrain.**WrapEpochValue**(*func*)

Use this wrapper if you want to define some value in your network which depends on the pretrain epoch. This is going to be part in your network description dict.

get_value(*epoch*)

Pretrain.**demo**()

Pretrain.**find_pretrain_wrap_values**(*net_json*)

Pretrain.**pretrainFromConfig**(*config*)

Return type Pretrain | None

RawWavDataset

class RawWavDataset.**RawWavDataset**(*listFile*, *frameLength*, *frameShift*, *num_outputs=None*, ***kwargs*)

This dataset returns the raw waveform information of wav files as sequence input data It uses temporary hdf files to buffer the data, to avoid repeatadly rading the wav files.

constructor

Parameters

- **listFile** (*string*) – path to the file containing a list of wav file pathes (on path per line) each line needs to contain exactly one wav file which is considered a sequence
- **frameLenth** – length of one frame in samples
- **frameShift** (int) – shift length of frame in samples
- **num_outputs** (int) – this needs to be set if the data set is used with only input data (e.g. for the extraction process).

get_data_dim(*key*)
This is copied from CachedDataset2 but the assertion is removed (see CachedDataset2.py)

Return type *int*

Returns number of classes, no matter if sparse or not

init_seq_order(*epoch=None*, *seq_list=None*)

Parameters

- **epoch** (*int|None*) – epoch number
- **seq_list** (*list[str] | None seq_list: In case we want to set a predefined order.*) – only None is currently supported

Initialize lists: self.seq_index # sorted seq idx

num_seqs
returns the number of sequences of the dataset

Return type *int*

RecurrentTransform

class RecurrentTransform.**RecurrentTransformBase**(*force_gpu=False*, *layer=None*, *for_custom=False*)

Parameters **for_custom** (*bool*) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = None

copy_for_custom(*force_gpu=True*)
:returns a new instance of this class for LSTMCustomOp

create_vars_for_custom()
Called via CustomLSTMFunctions.

init_vars()

create_vars()
Called for regular theano.scan().

add_param(*v*, *name=None*, ***kwargs*)

add_input(*v*, *name=None*)

add_state_var(*initial_value*, *name=None*)

add_var(*v*, *name=None*)

get_sorted_non_sequence_inputs()

get_sorted_custom_vars()

get_sorted_state_vars()

get_sorted_state_vars_initial()

set_sorted_state_vars(*state_vars*)

get_state_vars_seq(*state_var*)

step(*y_p*)

Parameters **y_p** (*theano.Variable*) – output of last time-frame. 2d (batch,dim)

Returns z_re, updates

Return type (theano.Variable, dict[theano.Variable, theano.Variable])

cost()

Return type theano.Variable | None

class RecurrentTransform.**AttentionTest**(*force_gpu=False*, *layer=None*, *for_custom=False*)

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'test'

create_vars()

step(*y_p*)

class RecurrentTransform.**DummyTransform**(*force_gpu=False*, *layer=None*, *for_custom=False*)

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'none'

step(*y_p*)

class RecurrentTransform.**DynamicTransform**(*force_gpu=False*, *layer=None*, *for_custom=False*)

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'rnn'

create_vars()

step(*y_p*)

class RecurrentTransform.**BatchNormTransform**(*force_gpu=False*, *layer=None*, *for_custom=False*)

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'batch_norm'

create_vars()

batch_norm(*h*, *use_shift=True*, *use_std=True*, *use_sample=0.0*)

step(*y_p*)

class RecurrentTransform.**LM**(*force_gpu=False*, *layer=None*, *for_custom=False*)

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'lm'

create_vars()

step(*y_p*)

class RecurrentTransform.**AttentionBase**(*force_gpu=False*, *layer=None*, *for_custom=False*)

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

base = None

name = 'attention_base'

attrs

create_vars()

default_updates()

step(*y_p*)

distance(*C*, *H*)

beam(*X*, *beam_idx=None*)

align(*w_i*, *Q*)

softmax(*D*, *I*)

class RecurrentTransform.**AttentionList**(*force_gpu=False*, *layer=None*, *for_custom=False*)
attention over list of bases

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'attention_list'

init(*i*)

create_bias(*n*, *name*, *i=-1*)

create_weights(*n*, *m*, *name*, *i=-1*)

create_vars()

item(*name*, *i*)

get(*y_p*, *i*, *g*)

attend(*y_p*)

cost()

class RecurrentTransform.**AttentionAlign**(*force_gpu=False*, *layer=None*, *for_custom=False*)
alignment controlled attention

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'attention_align'

create_vars()

attend(*y_p*)

class RecurrentTransform.**AttentionInverted**(*force_gpu=False*, *layer=None*, *for_custom=False*)
alignment controlled attention

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'attention_inverted'

create_vars()

attend(*y_p*)

class RecurrentTransform.**AttentionSegment**(*force_gpu=False*, *layer=None*, *for_custom=False*)
alignment controlled attention over segments

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'attention_segment'

create_bias(*n*, *name*, *i=-1*)

create_weights(*n*, *m*, *name*, *i=-1*)

create_vars()

make_index(*inv_att*, *ind*)

attend(*y_p*)

class RecurrentTransform.**AttentionTime**(*force_gpu=False*, *layer=None*, *for_custom=False*)
Concatenate time-aligned base element into single list element

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'attention_time'

make_base()

create_vars()

default_updates()

class RecurrentTransform.**AttentionTree**(*force_gpu=False*, *layer=None*, *for_custom=False*)
attention over hierarchy of bases in different time resolutions

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'attention_tree'

attend(*y_p*)

class RecurrentTransform.**AttentionBin**(*force_gpu=False*, *layer=None*, *for_custom=False*)
pruning of hypotheses in base[0] by attending over versions in time lower resolutions

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'attention_bin'

attend(*y_p*)

class RecurrentTransform.**AttentionTimeGauss**(*force_gpu=False*, *layer=None*, *for_custom=False*)

Parameters **for_custom** (bool) – When used with LSTMC + LSTMCustomOp, there are two instances of this class: One via the network initialization as part of the layer (for_custom == False) and another one via CustomLSTMFunctions (for_custom == True). The symbolic vars will look different. See self.create_vars_for_custom().

name = 'attention_time_gauss'

create_vars()

step(*y_p*)

cost()

RecurrentTransform.**get_dummy_recurrent_transform**(*recurrent_transform_name*, *n_out=5*, *n_batches=2*, *n_input_t=2*, *n_input_dim=2*)

Return type *RecurrentTransformBase*

This function is a useful helper for testing/debugging.

Server

SprintCache

This module is about reading (maybe later also writing) the Sprint archive format.

class SprintCache.**AllophoneLabeling**(*silence_phone*, *allophone_file*, *phoneme_file=None*, *state_tying_file=None*, *verbose_out=None*)

Parameters

- **silence_phone** (*str*) – e.g. "si"
- **allophone_file** (*str*) – list of allophones
- **phoneme_file** (*str|None*) – list of phonemes
- **state_tying_file** (*str|None*) – allophone state tying (e.g. via CART). maps each allophone state to a class label

- **verbose_out** (*file*) – stream to dump log messages

get_label_idx(*allo_idx*, *state_idx*)

get_label_idx_by_allo_state_idx(*allo_state_idx*)

class SprintCache.**FileArchive**(*filename*, *must_exists=True*)

SprintCacheHeader = 'SP_ARC1\x00'

addAttributes(*filename*, *dim*, *duration*)

addFeatureCache(*filename*, *features*, *times*)

end_recovery_tag = 1437226410

file_list()

finalize()

getState(*mix*)

has_entry(*filename*)

Parameters **filename** (*str*) – argument for self.read()

Returns True if we have this entry

read(*filename*, *typ*)

Parameters

- **filename** (*str*) – the entry-name in the archive
- **typ** (*str*) – "str", "feat" or "align"

Returns

depending on typ, "str" -> string, "feat" -> (time, data), "align" -> align, where string is a str, time is list of time-stamp tuples (start-time,end-time) in millisecs,

data is a list of features, each a numpy vector,

align is a list of (time, allophone, state), time is an int from 0 to len of align, allophone is some int, state is e.g. in [0,1,2].

Return type str|(*list*[numpy.ndarray],*list*[numpy.ndarray])|list[(*int*,*int*,*int*)]

readFileInfoTable()

read_U32()

read_bytes(*l*)

read_char()

read_f32()

read_f64()

read_str(*l*, *enc='ascii'*)

read_u32()

read_u64()

read_v(*typ*, *size*)

Parameters

- **typ** (*str*) – "f" for float (float32) or "d" for double (float64)
- **size** (*int*) – number of elements to return

Returns numpy array of shape (size,) of dtype depending on typ

Return type numpy.ndarray

scanArchive()

setAllophones(*f*)

Parameters **f** (*str*) – allophone filename. line-separated. will ignore lines starting with "#"

start_recovery_tag = 2857740885

writeFileInfoTable()

write_U32(*i*)

write_char(*i*)

write_f32(*i*)

write_f64(*i*)

write_str(*s*)

write_u32(*i*)

write_u64(*i*)

class SprintCache.**FileArchiveBundle**(*filename*)

Parameters **filename** (*str*) – .bundle file

file_list()

Return type *list*[str]

Returns list of content-filenames (which can be used for self.read())

has_entry(*filename*)

Parameters **filename** (*str*) – argument for self.read()

Returns True if we have this entry

read(*filename*, *typ*)

Parameters

- **filename** (*str*) – the entry-name in the archive
- **typ** (*str*) – "str", "feat" or "align"

Returns

depending on typ, "str" -> string, "feat" -> (time, data), "align" -> align, where string is a str, time is list of time-stamp tuples (start-time,end-time) in millisecs,

data is a list of features, each a numpy vector,

align is a list of (time, allophone, state), time is an int from 0 to len of align, allophone is some int, state is e.g. in [0,1,2].

Return type str|(*list*[numpy.ndarray],*list*[numpy.ndarray])|list[(*int*,*int*,*int*)]

Uses FileArchive.read().

setAllophones (*filename*)

Parameters **filename** (*str*) – allophone filename

class SprintCache.**FileInfo** (*name*, *pos*, *size*, *compressed*, *index*)

class SprintCache.**MixtureSet** (*filename*)

getCovByIdx (*idx*)

getMeanByIdx (*idx*)

getNumberMixtures ()

read_U32 ()

read_char ()

read_f32 ()

read_f64 ()

read_str (*l*, *enc='ascii'*)

read_u32 ()

read_u64 ()

read_v (*size*, *a*)

write (*filename*)

SprintCache.**is_sprint_cache_file** (*filename*)

Parameters **filename** (*str*) – file to check. must exist

Returns True iff this is a sprint cache (which can be loaded with *open_file_archive()*)

Return type *bool*

SprintCache.**main** ()

SprintCache.**open_file_archive** (*archive_filename*, *must_exists=True*)

Parameters

- **archive_filename** (*str*) –
- **must_exists** (bool) –

Return type FileArchiveBundlelFileArchive

SprintControl

This is the Sprint interface implementation, concretely for Sprint PythonControl, Sprint PythonSegmentOrder and Sprint SprintNnPythonLayer. For reference, in Sprint code, see:

- src/Nn/PythonControl.cc
- src/Tools/NnTrainer/python_control_demo.py

This interface will behave similar as SprintExternInterface. See SprintErrorSignals for the other end. It can also be used as a PythonSegmentOrdering interface. It also supports SprintNnPythonLayer.

class SprintControl.**PythonControl**(*c2p_fd*, *p2c_fd*, ***kwargs*)

This will send data to CRNN over a pipe. We expect that we are child process and the parent process has spawned us,

An instance of this class is also the interface for multiple Sprint interfaces, i.e.:

- PythonControl (standalone via NnTrainer tool)
- PythonControl (via SegmentwiseNnTrainer)
- implicitly PythonSegmentOrder (see code above)

Parameters

- **c2p_fd** (int) – child-to-parent file descriptor
- **p2c_fd** (int) – parent-to-child file descriptor

Version = 1

check_control_loop_running()

close()

classmethod create(***kwargs*)

exit(***kwargs*)

Called by Sprint.

get_current_seg_posteriors(*seg_len*)

Parameters **seg_len** (int) – just for double checking, the length of the current segment

Returns matrix (time,label)

handle_cmd(*cmd*, **args*)

handle_cmd_exit()

handle_cmd_export_allophone_state_fsa_by_segment_name(*segment_name*)

handle_cmd_get_loss_and_error_signal(*seg_name*, *seg_len*, *posteriors*)

Parameters

- **seg_name** (*str*) – seg name
- **seg_len** (int) – the segment length in frames
- **posteriors** (*numpy.ndarray*) – 2d (time,label) float array

See SprintErrorSignals.SprintSubprocessInstance.get_loss_and_error_signal().

handle_cmd_init(*name*, *version*)

handle_next()

Called by self.run_control_loop. We catch some message from our parent process, handle it and send back the result.

init_processing(*input_dim=None*, *output_dim=None*, ***kwargs*)

This is called via Sprint when we use PythonControl to iterate the corpus, i.e. we set –*.action=python-control in Sprint in the NN trainer tool. We expect that we use the Sprint callback to calculate loss and error signal. This is called on the first segment. input_dim/output_dim are set iff we extract features/alignments.

init_segment(*segment_name*)

Called by Sprint PythonControl in FeedForwardTrainer/SegmentwiseNnTrainer.

instance = None

notify_segment_loss(*segment_name*, *loss*)
Called by Sprint PythonControl in FeedForwardTrainer/SegmentwiseNnTrainer.

own_tcb_get_loss_and_error_signal(*seg_name*, *seg_len*, *posteriors*)

own_tcb_version()

own_threaded_callback(*cmd*, **args*)
This is used if we run our own control loop via run_threaded_control_loop.

process_segment(*name*, *orthography*, *features=None*, *alignment=None*, *soft_alignment=None*, ***kwargs*)
This is called via Sprint when we use PythonControl to iterate the corpus. :param str name: segment name :param str orthography: segment orth

run_control_loop(*callback*, ***kwargs*)
Called by Sprint when we are in PythonControl run_control_loop mode. Also called by us via self.run_threaded_control_loop().

run_threaded_control_loop()

segment_list_iterator()

set_current_seg_error_signal(*seg_len*, *error_signal*)

Parameters

- **seg_len** (int) – just for double checking, the length of the current segment
- **error_signal** – matrix (time,label)

set_current_seg_loss(*seg_name*, *loss*)

Parameters

- **seg_name** (*str|None*) – just for double checking, the name of the current segment. might be None
- **loss** (float) – the loss of the current seg

class SprintControl.**SprintNnPythonLayer**(*config*, ***kwargs*)
Sprint will directly call this class, i.e. create an instance of it. It implements the Sprint NN PythonLayer interface.

backpropagate(*errorSignalIn*)

Parameters **errorSignalIn** – matrix of format (output_size,time)

Returns tuple of matrices of format (input_size,time)

finalize()

forward(*input*)

Parameters **input** – tuple of input matrices of format (input_size,time). we ignore them.

Returns single output matrix of format (output_size,time)

getNumberOfFreeParameters()

initializeNetworkParameters()

isTrainable()

loadNetworkParameters(*filename*)

saveNetworkParameters(*filename*)

setInputDimension(*stream*, *size*)

setOutputDimension(*size*)

SprintControl.**getSegmentList**(*corpusName*, *segmentList*, *config*, ***kwargs*)
Sprint will directly call this function.

SprintControl.**init**(*name*, *reference*, *config*, *sprint_unit=None*, *version_number=None*, *callback=None*, ***kwargs*)
This will be called by Sprint PythonControl. But we also call it ourselves e.g. in getSegmentList() and SprintNnPythonLayer. :param str name: this specifies the caller. e.g. "Sprint.PythonControl" :param reference: this is any object to identify the specific instance of the caller, if there are multiple. :param str config: this will be passed over from Sprint. you can configure that via –*.pymod-config. :param str sprint_unit: if this is called by Sprint PythonControl, this will specify which specific part

of Sprint is using this PythonControl, because there can be multiple parts. E.g. there is "FeedForwardTrainer", "SegmentwiseNnTrainer" and "NnTrainer.pythonControl".

Parameters

- **version_number** (*int|None*) – if this is called by Sprint PythonControl, this will set the version number. only newer Sprint versions will set this.
- **callback** (*function|None*) – if this is called by Sprint PythonControl, this might provide a callback. Only newer Sprint versions will provide this to init(). This callback can be used for many different actions. It's supposed to be called like callback(action, **other_args), where action is a string. See Sprint PythonControl code about the possible actions and arguments.
- **kwargs** – all remaining args are specific for each caller.

In this specific module, we expect that there is "c2p_fd" and "p2c_fd" in the config string to communicate with the parent process, which is usually handled by SprintErrorSignals.

SprintControl.**print**(**args*, ***kwargs*)

SprintDataset

Implements the SprintDatasetBase and ExternSprintDataset classes, some Dataset subtypes. Note that from the main RETURNN process, you probably want ExternSprintDataset instead.

class SprintDataset.**ExternSprintDataset**(*sprintTrainerExecPath*, *sprintConfigStr*, *partitionEpoch=1*, ***kwargs*)
This is a Dataset which you can use directly in RETURNN. You can use it to get any type of data from Sprint (RWTH ASR toolkit), e.g. you can use Sprint to do feature extraction and preprocessing.

This class is like SprintDatasetBase, except that we will start an external Sprint instance ourselves which will forward the data to us over a pipe. The Sprint subprocess will use SprintExternInterface to communicate with us.

Parameters

- **sprintTrainerExecPath** (*str|list[str]*) –
- **| list[str] |()->str | list[()->str] |()->list[str] |()->list[()->str] sprintConfigStr** (*str*) – via eval_shell_str

exit_handler()

init_epoch(*epoch=None*, *seq_list=None*)

init_seq_order(*epoch=None*, *seq_list=None*)

num_seqs

reader_thread_proc(*child_pid*, *epoch*)

class SprintDataset.**SprintCacheDataset**(*data*, ***kwargs*)

Can directly read Sprint cache files (and bundle files). Supports both cached features and cached alignments. For alignments, you need to provide all options for the AllophoneLabeling class, such as allophone file, etc.

Parameters **data** (*dict[str,dict[str]]*) – data-key -> dict which keys such as filename, see SprintCacheReader constructor

class **SprintCacheReader**(*data_key*, *filename*, *type=None*, *allophone_labeling=None*)

Parameters

- **data_key** (*str*) – e.g. "data" or "classes"
- **filename** (*str*) – to Sprint cache archive
- **type** (*str|None*) – "feat" or "align"
- **allophone_labeling** (*dict[str]*) – kwargs for AllophoneLabeling

read(*name*)

Parameters **name** (*str*) – content-filename for sprint cache
Returns numpy array of shape (time, [num_labels])
Return type numpy.ndarray

SprintCacheDataset.**get_data_keys**()

Return type *list*[str]

SprintCacheDataset.**get_dataset_seq_for_name**(*name*, *seq_idx=-1*)

SprintCacheDataset.**get_tag**(*sorted_seq_idx*)

Return type str

SprintCacheDataset.**get_target_list**()

Return type *list*[str]

SprintCacheDataset.**init_seq_order**(*epoch=None*, *seq_list=None*)

class SprintDataset.**SprintDatasetBase**(*target_maps=None*, *str_add_final_zero=False*, ***kwargs*)

In Sprint, we use this object for multiple purposes: - Multiple epoch handling via SprintInterface.getSegmentList().

For this, we get the segment list from Sprint and use the Dataset shuffling method.

•Fill in data which we get via SprintInterface.feedInput*(). Note that each such input doesn't necessarily correspond to a single segment. This depends which type of FeatureExtractor is used in Sprint. If we use the BufferedFeatureExtractor in utterance mode, we will get one call for every segment and we get also segmentName as parameter. Otherwise, we will get batches of fixed size - in that case, it doesn't correspond to the segments. In any case, we use this data as-is as a new seq. Because of that, we cannot really know the number of seqs in advance, nor the total number of time frames, etc.

If you want to use this directly in RETURNN, see ExternSprintDataset.

SprintCachedSeqsMax = 200

SprintCachedSeqsMin = 100

addNewData(*features*, *targets=None*, *segmentName=None*)
Adds a new seq. This is called via the Sprint main thread. :param numpy.ndarray features: format (input-feature,time) (via Sprint) :param dict[str,numpy.ndarray|str] targets: format (time) (idx of output-feature) :returns the sorted seq index :rtype: int

finalizeSprint()
Called when SprintInterface.getSegmentList() ends.

finishSprintEpoch()
Called by SprintInterface.getSegmentList(). This is in a state where Sprint asks for the next segment after we just finished an epoch. Thus, any upcoming self.addNewData() call will contain data from a segment in the new epoch. Thus, we finish the current epoch in Sprint.

get_complete_frac(*seq_idx*)

get_ctc_targets(*sorted_seq_idx*)

get_input_data(*sorted_seq_idx*)

get_num_timesteps()

get_seq_length(*sorted_seq_idx*)

get_tag(*sorted_seq_idx*)

get_target_list()

get_targets(*target*, *sorted_seq_idx*)

have_seqs()

initSprintEpoch(*epoch*)

Called by SprintInterface.getSegmentList() when we start a new epoch. We must not call this via self.init_seq_order() because we will already have filled the cache by Sprint before the CRNN train thread starts the epoch.

init_seq_order(*epoch=None*, *seq_list=None*)
Called by CRNN train thread when we enter a new epoch.

is_cached(*start*, *end*)

is_less_than_num_seqs(*n*)

load_seqs(*start*, *end*)

num_seqs

setDimensions(*inputDim*, *outputDim*)

Called via python_train.

set_complete_frac(*frac*)

useMultipleEpochs()
Called via SprintInterface.getSegmentList().

waitForCrnnEpoch(*epoch*)
Called by SprintInterface.

SprintDataset.**demo**()

SprintErrorSignals

This provides the Theano Op *SprintErrorSigOp* to get a loss and error signal which is calculated via Sprint. And there are helper classes to communicate with the Sprint subprocess to transfer the posteriors and get back the loss and error signal. It uses the SprintControl Sprint interface for the communication.

class SprintErrorSignals.**SeqTrainParallelControlDevHost**(*output_layer*, *output_target*, *sprint_opts*, *forward_seq_delay=5*)

Counterpart to Engine.SeqTrainParallelControl. This does all the handling on the Device proc side.

class **CalcLossState**(*forward_data*, *sprint_instance*)

class SeqTrainParallelControlDevHost.**ForwardData**(*seq_idx*, *seq_tag*, *posteriors*)

class SeqTrainParallelControlDevHost.**LossData**(*calc_loss_state*)

SeqTrainParallelControlDevHost.**do_forward**(*batch*)

Called via Engine.SeqTrainParallelControl. We expect that assign_dev_data was called before to set the right data. :param EngineUtil.Batch batch: the current batch, containing one or more seqs

SeqTrainParallelControlDevHost.**get_loss_and_hat_y**(*seq_idx*)

SeqTrainParallelControlDevHost.**have_seqs_loss_data**(*start_seq*, *end_seq*)

SeqTrainParallelControlDevHost.**have_space_in_forward_data_queue**(*num_seqs=0*)

Called via Engine.SeqTrainParallelControl.

SeqTrainParallelControlDevHost.**remove_old_loss_data**(*current_start_seq*)

SeqTrainParallelControlDevHost.**train_check_calc_loss**()

Called via Engine.SeqTrainParallelControl. :returns whether we added something to self.calc_loss_states.

SeqTrainParallelControlDevHost.**train_finish_epoch**()

Called via Engine.SeqTrainParallelControl.

SeqTrainParallelControlDevHost.**train_have_loss_for_cur_batches**()

Returns True iff we can start training for the current batches.

Called via Engine.SeqTrainParallelControl.

SeqTrainParallelControlDevHost.**train_set_cur_batches**(*batches*)

Called via Engine.SeqTrainParallelControl.

SeqTrainParallelControlDevHost.**train_set_loss_vars_for_cur_batches**()

Called via Engine.SeqTrainParallelControl.

SeqTrainParallelControlDevHost.**train_start_epoch**()

Called via Engine.SeqTrainParallelControl.

class SprintErrorSignals.**SprintAlignmentAutomataOp**(*sprint_opts*)

Op: maps segment names (tags) to fsa automata (using sprint) that can be used to compute a BW-alignment

make_node(*tags*)

perform(*node*, *inputs*, *output_storage*, *params=None*)

class SprintErrorSignals.**SprintErrorSigOp**(*sprint_opts*)

Op: log_posteriors, seq_lengths -> loss, error_signal (grad w.r.t. z, i.e. before softmax is applied)

make_node(*log_posteriors*, *seq_lengths*)

perform(*node*, *inputs*, *output_storage*, *params=None*)

class SprintErrorSignals.**SprintInstancePool**(*sprint_opts*)
This is a pool of Sprint instances. First, for each unique sprint_opts, there is a singleton

which can be accessed via get_global_instance.

Then, this can be used in multiple ways.

1. get_batch_loss_and_error_signal.
2. ...

get_automata_for_batch(*tags*)

Parameters **tags** (*list[str]|numpy.ndarray*) – sequence names, used for Sprint (ndarray of shape (batch, max_str_len))

Returns (edges, weights, start_end_states). all together in one automaton. edges are of shape (4, num_edges), each (from, to, emission-idx, seq-idx), of dtype uint32. weights are of shape (num_edges,), of dtype float32. start_end_states are of shape (2, batch), each (start,stop) state idx, batch = len(tags), of dtype uint32.

Return type (numpy.ndarray, numpy.ndarray, numpy.ndarray)

get_batch_loss_and_error_signal(*log_posteriors*, *seq_lengths*, *tags=None*)

Parameters

- **log_posteriors** (*numpy.ndarray*) – 3d (time,batch,label)
- **seq_lengths** (*numpy.ndarray*) – 1d (batch)
- **tags** (*list[str]*) – seq names, length = batch

:rtype (numpy.ndarray, numpy.ndarray) :returns (loss, error_signal). error_signal has the same shape as posteriors. loss is a 1d-array (batch).

Note that this accesses some global references, like global current seg info, via the current Device instance. Thus this is expected to be run from the Device host proc,

inside from SprintErrorSigOp.perform.

This also expects that we don't have chunked seqs.

get_free_instance()

classmethod **get_global_instance**(*sprint_opts*)

global_instances = {}

class SprintErrorSignals.**SprintSubprocessInstance**(*sprintExecPath*, *minPythonControlVersion=2*, *sprintConfigStr=''*, *sprintControlConfig=None*, *usePythonSegmentOrder=True*)
The Sprint instance which is used to calculate the error signal. Communication is over a pipe. We pass the fds via cmd-line to the child proc. Basic protocol with the subprocess (encoded via pickle):

P2C: tuple (cmd, *cmd_args). cmd is any str. C2P: tuple (status, *res_args). status == "ok" if no error.

Commands: "init", name, version -> "ok", child_name, version "exit" -> (exit) "get_loss_and_error_signal", seg_name, seg_len, posteriors -> "ok", loss, error_signal

Numpy arrays encoded via TaskSystem.Pickler (which is optimized for Numpy).

On the Sprint side, we handle this via the SprintControl Sprint interface.

Parameters

- **sprintExecPath** (*str*) – this executable will be called for the sub proc.
- **minPythonControlVersion** (int) – will be checked in the subprocess. via Sprint PythonControl
- **sprintConfigStr** (*str*) – passed to Sprint as command line args. can have "config:" prefix - in that case, looked up in config. handled via eval_shell_str(), can thus have lazy content (if it is callable, will be called).
- **sprintControlConfig** (*dict[str]|None*) – passed to SprintControl.init().

Version = 1

exit_handler()

get_loss_and_error_signal__have_data()

get_loss_and_error_signal__read()
:rtype (str, float, numpy.ndarray) :returns (seg_name, loss, error_signal). error_signal has the same shape as posteriors.

get_loss_and_error_signal__send(*seg_name*, *seg_len*, *log_posteriors*)

Parameters

- **seg_name** (*str*) – the segment name (seq_tag)
- **seg_len** (int) – the segment length in frames
- **log_posteriors** (*numpy.ndarray*) – 2d (time,label) float array, log probs

init()

SprintErrorSignals.**sprint_loss_and_error_signal**(*output_layer*, *target*, *sprint_opts*, *log_posteriors*, *seq_lengths*)

Parameters

- **output_layer** (NetworkOutputLayer.SequenceOutputLayer) – output layer
- **target** (*str*) – e.g. "classes"
- **sprint_opts** (*dict[str]*) – for SprintInstancePool
- **log_posteriors** – 3d ndarray (time,batch,dim)
- **seq_lengths** – 1d ndarray (batch,) -> seq len

Returns loss, error_signal. loss is a 2d ndarray (batch,) -> loss. error_signal has the same shape as log_posteriors. error_signal is the grad w.r.t. z, i.e. before softmax is applied.

SprintExternInterface

This is a Sprint interface implementation. See SprintInterface.py for another Sprint interface. This Sprint interface is to be used for ExternSprintDataset, which should automatically use it.

SprintExternInterface.**getSegmentList**(*corpusName*, *segmentList*, ***kwargs*)
Called by Sprint PythonSegmentOrder. Set python-segment-order = true in Sprint to use this.

If this is used, this gets called really early. If it is used together with the Sprint PythonTrainer, it will get called way earlier before the init() below. It might also get called multiple times, e.g. if Sprint is in interactive mode to calc the seg count. This is optional. You can use the SprintInterface only for the PythonTrainer.

Return type *list*[str]

:returns segment list. Can also be an iterator.

SprintExternInterface.**exchook**(*exc_type*, *exc_obj*, *exc_tb*)

SprintExternInterface.**init**(***kwargs*)

SprintExternInterface.**init_PythonTrainer**(*inputDim*, *outputDim*, *config*, *targetMode*, ***kwargs*)

Called by Sprint when it initializes the PythonTrainer. Set trainer = python-trainer in Sprint to enable. Note that Sprint will call this, i.e. the trainer init lazily quite late, only once it sees the first data.

Parameters

- **config** (*str*) – config string, passed by Sprint. assumed to be ","-separated
- **targetMode** (*str*) – "target-alignment" or "criterion-by-sprint" or so

SprintExternInterface.**exit**()

SprintExternInterface.**feedInput**(*features*, *weights=None*, *segmentName=None*)

SprintExternInterface.**feedInputAndTargetAlignment**(*features*, *targetAlignment*, *weights=None*, *segmentName=None*)

SprintExternInterface.**feedInputAndTargetSegmentOrth**(*features*, *targetSegmentOrth*, *weights=None*, *segmentName=None*)

SprintExternInterface.**feedInputUnsupervised**(*features*, *weights=None*, *segmentName=None*)

SprintExternInterface.**feedInputAndTarget**(*features*, *weights=None*, *segmentName=None*, *orthography=None*, *alignment=None*, *speaker_name=None*, *speaker_gender=None*, ***kwargs*)

class SprintExternInterface.**PythonControl**(*config*, ***kwargs*)

instance = None

classmethod init(***kwargs*)

init_processing(*input_dim*, *output_dim*, ***kwargs*)

process_segment(*name*, *orthography*, *features*, *alignment*, *soft_alignment*, ***kwargs*)

exit(***kwargs*)

class SprintExternInterface.**ExternSprintDatasetSource**(*c2p_fd*, *p2c_fd*, *inputDim*, *outputDim*, *numSegments*)

This will send data to ExternSprintDataset over a pipe. We expect that we are child process and the parent process has spawned us via ExternSprintDataset and is waiting for our data.

Parameters

- **c2p_fd** (*int*) – child-to-parent file descriptor
- **p2c_fd** (*int*) – parent-to-child file descriptor
- **numSegments** (*int | None*) – can be None if not known in advance

addNewData(*segmentName*, *features*, *targets*)

Parameters

- **features** (*numpy.ndarray*) – 2D array, (feature,time)

- **targets** (*dict[str,numpy.ndarray]*) – each target is either 1D (time->idx) or 2D (time,class)

close()

SprintInterface

This is a Sprint interface implementation, i.e. you would specify this module in your Sprint config. (Sprint = the RWTH ASR toolkit.) Note that there are multiple Sprint interface implementations provided. This one would be used explicitly, e.g. for forwarding in recognition or wherever else Sprint needs posteriors (a FeatureScorer). Most of the other Sprint interfaces will be used automatically, e.g. via ExternSprintDataset, when it spawns its Sprint subprocess.

class SprintInterface.**Criterion**

error = None

errorSignal = None

gotErrorSignal = <threading._Event object>

gotPosteriors = <threading._Event object>

make_node(*posteriors*, *seq_lengths*)

perform(*node*, *inputs*, *output_storage*, *params=None*)

posteriors = None

SprintInterface.**demo**()

SprintInterface.**dumpFlags**()

SprintInterface.**exit**()

SprintInterface.**feedInput**(*features*, *weights=None*, *segmentName=None*)

SprintInterface.**feedInputAndTarget**(*features*, *weights=None*, *segmentName=None*, *orthography=None*, *alignment=None*, *speaker_name=None*, *speaker_gender=None*, ***kwargs*)

SprintInterface.**feedInputAndTargetAlignment**(*features*, *targetAlignment*, *weights=None*, *segmentName=None*)

SprintInterface.**feedInputAndTargetSegmentOrth**(*features*, *targetSegmentOrth*, *weights=None*, *segmentName=None*)

SprintInterface.**feedInputForwarding**(*features*, *weights=None*, *segmentName=None*)

SprintInterface.**feedInputUnsupervised**(*features*, *weights=None*, *segmentName=None*)

SprintInterface.**finishDiscard**()

SprintInterface.**finishError**(*error*, *errorSignal*, *naturalPairingType=None*)

SprintInterface.**forward**(*segmentName*, *features*)

Parameters **features** (*numpy.ndarray*) – format (input-feature,time) (via Sprint)

:return numpy.ndarray, format (output-dim,time)

SprintInterface.**getFinalEpoch**()

SprintInterface.**getSegmentList**(*corpusName*, *segmentList*, ***kwargs*)
Called by Sprint PythonSegmentOrder. Set python-segment-order = true in Sprint to use this.

If this is used, this gets called really early. If it is used together with the Sprint PythonTrainer, it will get called way earlier before the init() below. It might also get called multiple times, e.g. if Sprint is in interactive mode to calc the seg count. This is optional. You can use the SprintInterface only for the PythonTrainer.

Return type *list*[str]

:returns segment list. Can also be an iterator.

SprintInterface.**init**(*inputDim*, *outputDim*, *config*, *targetMode*, ***kwargs*)

Called by Sprint when it initializes the PythonTrainer. Set trainer = python-trainer in Sprint to enable. Note that Sprint will call this, i.e. the trainer init lazily quite late, only once it sees the first data.

Parameters

- **config** (*str*) – config string, passed by Sprint. assumed to be ","-separated
- **targetMode** (*str*) – "target-alignment" or "criterion-by-sprint" or so

SprintInterface.**initBase**(*configfile=None*, *targetMode=None*, *epoch=None*)

SprintInterface.**initDataset**()

SprintInterface.**prepareForwarding**()

SprintInterface.**setTargetMode**(*mode*)

Parameters **mode** (*str*) – target mode

SprintInterface.**startTrainThread**(*epoch=None*)

SprintInterface.**train**(*segmentName*, *features*, *targets=None*)

Parameters

- **segmentName** (*str|None*) – full name
- **features** (*numpy.ndarray*) – 2d array
- **targets** (*numpy.ndarray|dict[str,numpy.ndarray]|None*) – 2d or 1d array

StereoDataset

This file contains dataset implementations to have an easy to use interface for using RETURNN for regression. Applications are for example speech enhancement or mask estimations

class StereoDataset.**StereoDataset**(***kwargs*)

The purpose of this dataset is to be a base dataset for datasets which have an easy to use interface for using RETURNN as a regression tool

constructor

num_seqs

returns the number of sequences of the dataset

Return type *int*

init_seq_order(*epoch=None*, *seq_list=None*)

Parameters

- **epoch** (*int|None*) – epoch number

- **seq_list** (*list[str] | None seq_list: In case we want to set a predefined order.*) – only None is currently supported

Initialize lists: self.seq_index # sorted seq idx

class StereoDataset.**StereoHdfDataset**(*hdfFile*, *num_outputs=None*, *normalizationFile=None*, *flag_normalizeInputs=True*, *flag_normalizeTargets=True*, ***kwargs*)

A stereo dataset which needs an hdf file as input. The hdf file is supposed to always have group 'inputs' and for the training data it also needs to contain the group 'outputs'. Each group is supposed to contain one dataset per sequence. The names of the datasets are supposed to be consecutive numbers starting at 0.

The datasets are 2D numpy arrays, where dimension 0 is the time axis and dimension 1 is the feature axis. Therefore dimension 0 of the 'input' dataset and the respective 'output' dataset need to be the same.

Constructor

Parameters

- **hdfFile** (*str*) – path to the hdf file. if a bundle file is given (*.bundle) all hdf files listed in the bundle file will be used for the dataset. :see: BundleFile.BundleFile
- **num_outputs** (int) – this needs to be set if the stereo data hdf file only contains 'inputs' data (e.g. for the extraction process). Only if no 'outputs' data exists in the hdf file num_outputs is used.
- **normalizationFile** (*str | None*) – path to a HDF file with normalization data. The file is optional: if it is not provided then no normalization is performed. :see: NormalizationData.NormalizationData
- **flag_normalizeInputs** (bool) – if True then inputs will be normalized provided that the normalization HDF file has necessary datasets (i.e. mean and variance)
- **flag_normalizeTargets** (bool) – if True then targets will be normalized provided that the normalization HDF file has necessary datasets (i.e. mean and variance)

get_data_dim(*key*)

This is copied from CachedDataset2 but the assertion is removed (see CachedDataset2.py)

Return type *int*

Returns number of classes, no matter if sparse or not

num_seqs

Returns the number of sequences of the dataset

Return type *int*

Returns the number of sequences of the dataset.

class StereoDataset.**DatasetWithTimeContext**(*hdfFile*, *tau=1*, ***kwargs*)

This dataset composes a context feature by stacking together time frames.

Constructor

Parameters

- **hdfFile** (*string*) – see the StereoHdfDataset
- **tau** (int) – how many time frames should be on the left and on the right. E.g. if tau = 2 then the context feature will be created by stacking two neighboring time frames from left and two neighboring time frames from right: newInputFeature = [x_{t-2}, x_{t-1}, x_t, x_{t+1}, x_{t+2}]. In general new feature will have shape (2 * tau + 1) * originalFeatureDimensionality Output features are not changed.

- **kwargs** (*dictionary*) – the rest of the arguments passed to the StereoHdfDataset

TFDataPipeline

TensorFlow data pipeline

This module covers all related code to handle the data loading, preprocessing, chunking, batching and related things in TensorFlow, i.e. the TensorFlow data pipeline from the Dataset.

Some related documents:

https://www.tensorflow.org/programmers_guide/reading_data https://www.tensorflow.org/programmers_guide/threading_and_queues https://www.tensorflow.org/performance/performance_models https://www.tensorflow.org/api_guides/python/io_ops https://www.tensorflow.org/versions/r1.2/api_docs/python/tf/contrib/data https://github.com/tensorflow/tensorflow/issues/4535 https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/data/README.md https://stackoverflow.com/questions/41187745/tensorflow-how-can-i-evaluate-a-validation-data-queue

Data shuffling

1. The sequence shuffling is implemented as part of the Dataset, although we could also use a tf.RandomShuffleQueue on sequence level for training.
2. Chunk shuffling can be done with another tf.RandomShuffleQueue for training.
3. Frame shuffling only makes sense for non-recurrent networks. It also only makes sense if we already did any windowing beforehand. It also only makes sense in training. In that case, we could do chunking just with chunk size 1 and chunk step 1, or maybe chunk size = context window size, and then the frame shuffling is just chunk shuffling, thus we do not need any separate frame shuffling logic.

Generic pipeline

1. We initialize the Dataset for some epoch. The Dataset could shuffle the sequence order based on the epoch. Then we iterate over the sequences/entries of the dataset (seq_idx), starting with seq_idx = 0, checking Dataset.is_less_than_num_seqs. We get the data for each data key (e.g. "data" or "classes") as Numpy arrays with any shape. They don't need to have the same number of time frames, although in the common case where we have a frame-wise alignment of class labels, they would have. In any case, we basically have dict[str,numpy.ndarray], the seq_idx and also a seq_tag.
2. We could implement another sequence shuffling with tf.RandomShuffleQueue in training.
3. We do chunking of the data of each sequence, i.e. selecting chunks of chunk size frames, iterating over the frames of a sequence with stride = chunk step. While doing chunking, we could add more context around each chunk (zero padding at the borders) if needed, e.g. if we use windowing or convolution with padding="valid", such that the output of the net will match that of the targets. However, this depends on where the data is used in the network; maybe it is used at multiple points?
4. We can do chunk shuffling with another tf.RandomShuffleQueue in training.
5. We build up batches from the chunks.

First the simple method via feed_dict and placeholders

The input data which (and optionally the targets) can be represented with tf.placeholder and feed via feed_dict from tf.Session.run which does one train/eval/forward step. In this case, any preprocessing such as chunking and batching must be done beforehand via Numpy. This was the initial implementation and is also the standard implementation for the Theano backend.

This is not optimal because the tf.Session.run first has to copy the data from CPU to GPU and then can do whatever it is supposed to do (e.g. one train step). So copying and then the calculation is done in serial but it could be done in parallel with some other method which we will discuss below. Also the preprocessing could involve some more complex operations which could be slow with Python + Numpy. Also the chunk shuffling is more difficult to implement and would be slower compared to a pure TF solution.

Some use case

Conv net training. For every sequence, window around every frame for context. Window must belong together, no unnecessary zero padding should be done introduced by chunking. Thus, windowing must be done before chunking, or additional zero-padding must be added before chunking. Then formulated differently: Chunking with step 1, output for a chunk is a single frame. It also means that the windowing can not be part of the network because we will get only chunks there, or the windowing makes only sense with padding="valid", otherwise we would get way too much zero-padding also at the border of every chunk. The same is true for convolution, pooling and others. I.e. padding in time should always be in "valid" mode. If we feed in a whole sequence, must return the whole sequence, in recog, forwarding, search or eval. With padding="valid", the output has less time frames, exactly context-size less frames. Conv should use padding="valid" anyway to save computation time, and only explicitly pad zeros where needed. In recog, the input-format is (batch, time + context_size, ...) which is zero-padded by context_size additional frames. So, have context_size as an additional global option for the network (could be set explicitly in config, or calculated automatically at construction). When chunking for such case, we also should have chunks with such zero-paddings so that recog matches. So, basically, a preprocessing step, before chunking, in both training and recog, is to add zero-padding of size context_size to the input, then we get the output of the expected size.

Pipeline implementation

1. One thread which goes over the Dataset. No need for different training/eval queue, no random-shuffle-queue, seq-shuffling is done by the Dataset. Here we can also handle the logic to add the context_size padding to the input. Maybe use Dataset.iterate_seqs which gets us the offsets for each chunk. We can then just add the context_size to each. After that, chunking can be done (can be done in the same thread just at the final step).
2. Another thread TFBatchingQueue, which collects seqs or chunks and prepares batches.

It depends on whether the full network is recurrent or not.

class `TFDataPipeline.`**`PipeBase`**

`have_data_for_dequeue`()

Returns if we can dequeue from us now without blocking

Return type *bool*

`have_incoming_data`(*dep_pipe_connector*)

Parameters **`dep_pipe_connector`** (`PipeConnectorBase`) – will queue data to us

Returns whether we have now or in the future data ready for dequeue

Return type *bool*

class TFDataPipeline.**PipeConnectorBase**

is_running()

E.g. for pipe_in/pipe_out model: If the pipe_in has data, we increase our counter by 1, then dequeue from pipe_in, do sth and queue to pipe_out, and only then decrease the counter again. Thus, if we return False, we have ensured that the pipe_out already has the data, or there is no data anymore. If we return True, we will ensure that we will push more data to pipe_out at some point.

Returns counter > 0

Return type *bool*

class TFDataPipeline.**DatasetReader**(*extern_data*, *dataset*, *coord*, *feed_callback*, *with_seq_tag=False*, *with_seq_idx=False*, *with_epoch_end=False*)

Reads from Dataset into a queue.

Parameters

- **extern_data** (ExternData) –
- **dataset** (Dataset) –
- **coord** (*tf.train.Coordinator*) –
- **feed_callback** (*(dict[str,numpy.ndarray|str|int])->None*) –
- **with_seq_tag** (bool) –
- **with_seq_idx** (bool) –
- **with_epoch_end** (bool) –

SpecialKeys = ('seq_tag', 'seq_idx', 'epoch_end')

get_dtype_for_key(*key*)

Parameters **key** (*str*) –

Return type str

get_shape_for_key(*key*)

Parameters **key** (*str*) –

Returns shape without batch-dim

Return type tuple[int | None]

get_queue_kwargs()

loop()

is_running()

class TFDataPipeline.**MakePlaceholders**(*data_keys*, *extern_data*, *with_batch*)

Parameters

- **data_keys** (list[str]) –
- **extern_data** (ExternData) –
- **with_batch** (bool) –

data_placeholders()

feed_dict(*d*)

Parameters **d** (*dict[str,numpy.ndarray|str|int]*) –

Returns keys replaced by placeholders

Return type dict[tf.placeholder,numpy.ndarray|str|int]

class TFDataPipeline.**TFDataQueues**(*extern_data*, *capacity=100*, *seed=1*, *with_batch=False*, *enqueue_data=None*)

Generic queues which differ between train/eval queues.

Parameters

- **extern_data** (ExternData) – this specifies the data keys
- **capacity** (int) –
- **seed** (int) –
- **with_batch** (bool) – whether we have the batch-dim in input/output
- **enqueue_data** (*dict[str,tf.Tensor]*) – if provided, will be the input

make_dequeue_op()

have_more(*tf_session*)

Parameters **tf_session** (*tf.Session*) –

have_data_for_dequeue()

one_more_enqueue_is_enough()

enqueue(*tf_session*, *data=None*)

Parameters

- **tf_session** (*tf.Session*) –
- **data** (*dict[str,numpy.ndarray]|None*) – needed iff self.with_feed_input

class TFDataPipeline.**TFChunkingQueueRunner**(*extern_data*, *make_dequeue_op*, *target_queue*, *chunk_size=None*, *chunk_step=None*, *context_window=None*, *source_has_epoch_end_signal=False*)

Implements chunking in pure TF. I.e. we get full sequences of varying lengths as input (from a queue), and we go over it with stride = chunk step, and extract a window of chunk size at each position, which we feed into the target queue. Optionally, for each chunk, we can add more frames (context window) around the chunk.

Parameters

- **extern_data** (ExternData) –
- **make_dequeue_op** (*()->dict[str,tf.Tensor]*) –
- **target_queue** (*tf.QueueBase*) –
- **chunk_size** (*int|None*) –
- **chunk_step** (*int|None*) –
- **context_window** (*int|NumbersDict|None*) –
- **source_has_epoch_end_signal** (bool) –

is_running()

class TFDataPipeline.**TFBatchingQueue**(*data_queues*, *batch_size*, *max_seqs*, *capacity=10*)
Wrapper around tf.PaddingFIFOQueue with more control. Gets in data via TFDataQueues without batch-dim, and adds the batch-dim, according to batch_size and max_seqs. Output can be accessed via self.output_as_extern_data. This will represent the final output used by the network, controlled by QueueDataProvider.

Parameters

- **data_queues** (TFDataQueues) –
- **batch_size** (int) –
- **max_seqs** (int) –
- **capacity** (int) –

loop(*tf_session*, *coord*)

Parameters

- **tf_session** (*tf.Session*) –
- **coord** (*tf.train.Coordinator*) –

class TFDataPipeline.**QueueOutput**

get_data()

Return type dict[str,tf.Tensor]

have_data()

class TFDataPipeline.**CpuToDefaultDevStage**(*input_data*, *names*, *dtypes*, *extern_data*, *data_keys*)

Parameters

- **input_data** (*dict[str,tf.Tensor]*) –
- **names** (list*[str]*) – data_keys + extra info
- **dtypes** (list*[tf.DType|str]*) – corresponds to names
- **extern_data** (ExternData) –
- **data_keys** (list*[str]*) –

loop(*parent*, *coord*, *session*)

Parameters

- **parent** (QueueDataProvider) –
- **coord** (*tf.train.Coordinator*) –
- **session** (*tf.Session*) –

class TFDataPipeline.**DataProviderBase**(*extern_data*, *data_keys*)
Base class which wraps up the logic in this class. See derived classes.

Parameters

- **extern_data** (ExternData) –
- **data_keys** (set*(str)|None*) –

start_threads()

stop_threads()

have_more_data(*session*)

It is supposed to return True as long as we want to continue with the current epoch in the current dataset (train or eval). This is called from the same thread which runs the main computation graph (e.g. train steps).

Parameters **session** (*tf.Session*) –

Returns whether the next session.run() can run in the current epoch & dataset

Return type *bool*

get_feed_dict(*single_threaded=False*)

Gets the feed dict for TF session run(). Note that this will block if there is nothing in the queue. The queue gets filled by the other thread, via self.thread_main().

Parameters **single_threaded** (bool) – whether to not use the queue

Returns we dequeue one batch from the queue and provide it for all placeholders of our external data

Return type dict[tf.Tensor,tf.Tensor]

have_reached_end()

Returns whether the current dataset says that we reached the end.

Return type *bool*

get_dataset_name()

Returns current dataset name, e.g. "train" or "dev"

Return type str

get_complete_frac()

Returns by how much we are through the current dataset, number between 0 and 1, for visual feedback

Return type *float*

get_num_frames()

Return type *NumbersDict*

class TFDataPipeline.**FeedDictDataProvider**(*tf_session*, *dataset*, *batches*, *capacity=10*, *tf_queue=None*, ***kwargs*)

This class will fill all the placeholders used for training or forwarding or evaluation etc. of a *TFNetwork.Network*. It will run a background thread which reads the data from a dataset and puts it into a queue.

Parameters

- **tf_session** (*tf.Session|tf.InteractiveSession*) –
- **dataset** (Dataset) –
- **batches** (BatchSetGenerator) –
- **extern_data** (ExternData) –
- **data_keys** (set *(str)|None*) –
- **capacity** (int) –
- **tf_queue** (*TFDataQueues|None*) –

start_threads()

stop_threads()

get_next_batch()

thread_main()

have_more_data(*session*)

Returns when we go through an epoch and finished reading, this will return False

If this returns True, you can definitely read another item from the queue. Threading safety: This assumes that there is no other consumer thread for the queue.

get_feed_dict(*single_threaded=False*)

Gets the feed dict for TF session run(). Note that this will block if there is nothing in the queue. The queue gets filled by the other thread, via self.thread_main().

Parameters **single_threaded** (bool) – whether to not use the queue

Returns we dequeue one batch from the queue and provide it for all placeholders of our external data

Return type dict[tf.Tensor,numpy.ndarray]

get_dataset_name()

have_reached_end()

get_complete_frac()

get_num_frames()

class TFDataPipeline.**QueueDataProvider**(*shuffle_train_seqs=False*, ***kwargs*)

This class is supposed to encapsulate all the logic of this module and to be used by the TF engine. It gets the train and dev dataset instances.

High-level (not differentiating between train/eval) queues: 1. sequence queue (filled by the data from Dataset) 2. chunk queue (filled by chunking, and maybe context window) 3. batch queue (constructed batches from the chunks) 4. staging area (e.g. copy to GPU)

Creates the queues and connector instances (which will be the queue runners). Thus this will be created in the current TF graph, and you need to create a new instance of this class for a new TF graph. This is also only intended to be recreated when we create a new TF graph, so all other things must be created while it exists.

get_feed_dict(*single_threaded=False*)

have_more_data(*session*)

It is supposed to return True as long as we want to continue with the current epoch in the current dataset (train or eval). We want to continue if we still can do a next *self.final_stage.dequeue* op with the current dataset. This is called from the same thread which runs the main computation graph (e.g. train steps), as well as from the final stage thread.

Parameters **session** (*tf.Session*) –

Returns whether the next session.run() can run in the current epoch & dataset

Return type *bool*

have_reached_end()

get_complete_frac()

get_num_frames()

init_dataset(*session*, *dataset*, *is_train_dataset*)

Parameters

- **session** (*tf.Session*) –

- **dataset** (Dataset) –
- **is_train_dataset** (bool) –

get_dataset_name()

get_threads()

start_threads()

stop_threads()

TFEngine

TensorFlow engine

The basic engine for the TensorFlow backend is implemented here, i.e. the high-level logic to train, i.e. looping over epochs, holding the network instance, creating the TensorFlow session, managing the data pipeline, etc.

See *Technological overview* for an overview how it fits all together.

class TFEngine.**Engine**(*config=None*)

Parameters **config** (*Config.Config|None*) –

analyze(*data*, *statistics*)

Parameters

- **data** (Dataset.Dataset) –
- **statistics** (list[str]|None) – ignored at the moment

Returns nothing, will print everything to log.v1

check_last_epoch()

check_uninitialized_vars()

All vars in TF which are controlled by us should also have been initialized by us. We also take care about the optimizer slot variables. However, TF can still create other vars which we do not know about. E.g. the Adam optimizer creates the beta1_power/beta2_power vars (which are no slot vars). Here, we find all remaining uninitialized vars, report about them and initialize them.

config_get_final_epoch(*config*)

epoch_model_filename(*model_filename*, *epoch*, *is_pretrain*)

Return type str

eval_model()

eval_single(*dataset*, *seq_idx*, *output_dict*)

Parameters

- **dataset** (Dataset.Dataset) –
- **seq_idx** (int) –
- **output_dict** (dict[str,tf.Tensor]) – key -> tf.Tensor

Returns output_dict but values evaluated

Return type dict[str,numpy.ndarray]

finalize()

format_score(*score*)

forward_single(*dataset*, *seq_idx*, *output_layer_name=None*)

Parameters

- **dataset** (Dataset.Dataset) –
- **seq_idx** (int) –
- **output_layer_name** (*str|None*) – e.g. "output". if not set, will read from config "forward_output_layer"

Returns numpy array, output in time major format (time,dim)

Return type numpy.ndarray

forward_to_hdf(*data*, *output_file*, *combine_labels=''*, *batch_size=0*)

Is aiming at recreating the same interface and output as Engine.forward_to_hdf.

get_all_merged_summaries()

Returns merged summaries, serialized string

Return type tf.Tensor

get_const_tensor(*key*, *value*)

get_epoch_model(*config*)

:returns (epoch, modelFilename) :rtype: (int|None, str|None)

get_epoch_model_filename(*epoch=None*)

get_epoch_str()

get_eval_datasets()

get_specific_feed_dict(*dataset*, *seq_idx*)

Parameters

- **dataset** (Dataset.Dataset) –
- **seq_idx** (int) –

Returns feed_dict for self.tf_session.run()

Return type dict[str,numpy.ndarray]

get_train_start_epoch_batch(*config*)

We will always automatically determine the best start (epoch,batch) tuple based on existing model files. This ensures that the files are present and enforces that there are no old outdated files which should be ignored. Note that epochs start at idx 1 and batches at idx 0. :type config: Config.Config :returns (epoch,batch) :rtype (int,int)

init_network_from_config(*config*)

Parameters **config** (Config.Config) –

init_train_epoch()

init_train_from_config(*config*, *train_data*, *dev_data=None*, *eval_data=None*)

is_first_epoch_after_pretrain()

is_pretrain_epoch(*epoch=None*)

is_requesting_for_gpu()

load_model(*epoch=None*, *filename=None*)

Parameters

- **epoch** (int) –
- **filename** (*str*) –

maybe_init_new_network(*net_desc*)

save_model(*filename=None*)

Parameters **filename** (*str*) – full filename for model

search(*dataset*, *do_eval=True*, *output_layer_name='output'*)

Parameters

- **dataset** (Dataset.Dataset) –
- **do_eval** (bool) – calculate errors. can only be done if we have the reference target
- **output_layer_name** (*str*) –

train()

train_epoch()

class TFEngine.**Runner**(*engine*, *dataset*, *batches*, *train*, *eval=True*, *extra_fetches=None*, *extra_fetches_callback=None*)

Parameters

- **engine** (Engine) –
- **dataset** (Dataset.Dataset) –
- **batches** (BatchSetGenerator) –
- **train** (bool) – whether to do updates on the model
- **eval** (bool) – whether to evaluate (i.e. calculate loss/error)
- **extra_fetches** (*dict[str,tf.Tensor|TFUtil.Data|TFNetworkLayer.LayerBase]|None*) – additional fetches per step. *extra_fetches_callback* will be called with these. In case of Data/LayerBase, it will return a list, where each item corresponds to the batch-seq. It might also be useful to add *network.get_extern_data("seq_idx")* and *network.get_extern_data("seq_tag")*.
- **extra_fetches_callback** (*(**dict[str,numpy.ndarray|str|list[numpy.ndarray|str]) -> None*) – called if extra_fetches

run(*report_prefix*)

Parameters **report_prefix** (*str*) – prefix for logging

TFNativeOp

class TFNativeOp.**OpDescription**(*in_info*, *out_info*, *c_fw_code*, *c_bw_code=None*, *c_extra_support_code=None*, *code_version=None*, *cpu_support=True*, *grad_input_map=None*, *name=None*)

Parameters

- **in_info** (list[dict(str)]) – each dict describes one input var. attribs in the dict:

  int ndim: the ndim. tuple shape: tuple and can contain None for specific dimensions.

  optional attribs: str dtype: "float32" by default. bool need_contiguous: false by default. int want_inplace: -1 by default. try to optimize to destroy input, on output-index.

  "dummy_out" is a special value which will add another output.

  bool is_inplace: false by default. whether the optimization was applied. str gradient: can be "disconnected". see grad(). bool bw_input: True by default. add this param to the bw input.

  other attribs are just ignored.

- **out_info** (list[dict(str)]) – like in_info. slightly different behavior for:

  shape: we also allow refs to the in_info in the form (in-idx,dim). see infer_shape(). need_contiguous/want_inplace: used for bw, in case for bw_input == True.

- **c_fw_code** (*str*) – C code for forward pass
- **c_extra_support_code** (*str|dict[str]*) – C support code (for c_support_code)
- **c_bw_code** (*str|None*) – C code for backward pass (for gradient)
- **code_version** (*tuple[int]*) – will be returned by c_code_cache_version.
- **cpu_support** (bool) –
- **grad_input_map** (*tuple[int]|callable*) – selection of grad inputs. by default, we get all inputs + all outputs + all grad outputs.
- **name** (*str*) – name

classmethod from_gen_base(*gen_base*)

Parameters **gen_base** (NativeOp.NativeOpGenBase) –

Return type *OpDescription*

is_grad_defined

grad()

Return type OpDescription|None

class TFNativeOp.**OpMaker**(*description*, *compiler_opts=None*)
https://www.tensorflow.org/versions/master/how_tos/adding_an_op/

Parameters

- **description** (OpDescription) –
- **compiler_opts** (*dict[str]|None*) – passed on to OpCodeCompiler as kwargs

with_cuda = None

mod_cache = {}

op_cache = {}

op_name

cache_key

support_native_op_cpp_filename

make_op()

TFNativeOp.**make_lstm_op**(***kwargs*)

See *NativeLstmCell* for usage.

Returns op

Return type (tf.Tensor) -> tuple[tf.Tensor]

class TFNativeOp.**RecSeqCellOp**(*n_hidden*)

class TFNativeOp.**NativeLstmCell**(*n_hidden*)

classmethod map_layer_inputs_to_op(*Z*, *V_h*, *i*, *initial_state=None*)

Just like NativeOp.LstmGenericBase.map_layer_inputs_to_op(). :param tf.Tensor Z: inputs: shape (time,batch,n_hidden*4) :param tf.Tensor V_h: W_re: shape (n_hidden,n_hidden*4) :param tf.Tensor i: index: shape (time,batch) :param tf.Tensor|None initial_state: shape (batch,n_hidden) :rtype: (tf.Tensor,tf.Tensor,tf.Tensor,tf.Tensor)

TFNativeOp.**make_fast_baum_welch_op**(***kwargs*)

Returns op

Return type (tf.Tensor) -> tuple[tf.Tensor]

TFNativeOp.**fast_baum_welch**(*am_scores*, *edges*, *weights*, *start_end_states*, *float_idx*, *state_buffer=None*)

Parameters

- **am_scores** (*tf.Tensor*) – (time, batch, dim), in -log space
- **edges** (*tf.Tensor*) – (4,num_edges), edges of the graph (from,to,emission_idx,sequence_idx)
- **weights** (*tf.Tensor*) – (num_edges,), weights of the edges
- **start_end_states** (*tf.Tensor*) – (2, batch), (start,end) state idx in automaton. there is only one single automaton.
- **float_idx** (*tf.Tensor*) – (time, batch) -> 0 or 1 (index mask, via seq lens)
- **state_buffer** (*tf.Tensor*) – (2, num_states)

Returns (fwdbwd, obs_scores), fwdbwd is (time, batch, dim), obs_scores is (time, batch), in -log space

Return type (tf.Tensor, tf.Tensor)

TFNativeOp.**fast_baum_welch_by_sprint_automata**(*am_scores*, *float_idx*, *tags*, *sprint_opts*)

Parameters

- **am_scores** (*tf.Tensor*) – (time, batch, dim), in -log space
- **float_idx** (*tf.Tensor*) – (time, batch) -> 0 or 1 (index mask, via seq lens)
- **tags** (*tf.Tensor*) – (batch,) -> seq name (str)
- **sprint_opts** (*dict[str]*) –

Returns (fwdbwd, obs_scores), fwdbwd is (time, batch, dim), obs_scores is (time, batch), in -log space

Return type (tf.Tensor, tf.Tensor)

`TFNativeOp.`**demo**()

TFNetwork

class `TFNetwork.`**ExternData**(*data=None*, *default_input='data'*, *default_target='classes'*)

This holds *Data* instances for every data-key of external data from the dataset, i.e. the description such as shape and sparsity, etc.

Parameters **data** (*None|dict[str,dict[str]]*) – optional init kwargs for Data

get_data(*name*)

get_data_description()

get_default_input_data()

get_default_target_data()

get_queue_args(*with_batch_dim*, *fixed_batch_dim=None*)

Parameters

- **with_batch_dim** (bool) –
- **fixed_batch_dim** (*int|None*) –

Returns kwargs for tf.Queue.__init__

Return type dict[str,*list*]

has_data(*name*)

init_from_config(*config*)

Parameters **config** (Config.Config) –

init_from_dataset(*dataset*)

Parameters **dataset** (Dataset.Dataset) –

register_data(*data*)

Parameters **data** (Data) – will use data.name as the key

register_data_from_dict(*data*)

Parameters **data** (*dict[str,dict[str]]*) – init kwargs for Data

class `TFNetwork.`**TFNetwork**(*config=None*, *extern_data=None*, *rnd_seed=42*, *train_flag=False*, *search_flag=False*, *parent_layer=None*, *parent_net=None*, *name=None*)

Parameters

- **config** (Config.Config) – only needed to init extern_data if not specified explicitly
- **extern_data** (*ExternData|None*) –
- **rnd_seed** (int) –
- **train_flag** (*bool|tf.Tensor*) – True if we want to use this model in training, False if in eval, or dynamic
- **parent_layer** (*TFNetworkLayer.LayerBase|None*) –

- **parent_net** (TFNetwork) –
- **name** (*str*) – only for debugging

add_layer(*name*, *layer_class*, ***layer_desc*)

Parameters

- **name** (*str*) –
- **layer_class** (*(()->LayerBase)|LayerBase*) –

cond_on_train(*fn_train*, *fn_eval*)

Uses fn_train() or fn_eval() base on self.train_flag. It will be a branched evaluation.

Parameters

- **fn_train** (*()->tf.Tensor*) –
- **fn_eval** (*()->tf.Tensor*) –

Returns fn_train() if self.train_flag else fn_eval()

Return type tf.Tensor

construct_from(*list_or_dict*)

Parameters **| dict[str,dict[str]] list_or_dict** (list*[dict[str]]*) –

construct_from_dict(*net_dict*)

Parameters **net_dict** (*dict[str,dict[str]]*) –

construct_from_list(*net_list*)

Parameters **net_list** (list*[dict[str]]*) – list of layer descriptions

construct_objective()

declare_train_params(*hidden_layer_selection=None*, *with_output=None*)

get_absolute_name_scope_prefix()

get_all_errors()

Return type dict[str|tf.Tensor]

Returns layer-name -> error dict. contains only the layers which have some error value

get_all_losses()

get_auxiliary_params()

get_batch_dim()

Get the batch-dim size, i.e. amount of sequences in the current batch. Consider that the data tensor is usually of shape [batch, time, dim], this would return shape(data)[0].

The code currently assumes that the batch-dim can be taken from the extern data. If it does not have that available for some reason (e.g. some subnetwork), it will try some alternative sources and assumes that they have the correct batch-dim.

Note that the batch-dim usually stays always the same across the whole network and also every individual batch sequence will stay related. One notable exception of this is the choice layer, where the batch-dim will get expanded by the beam search if search is used, as well as in all following layers, until there is a decide layer.

Returns int scalar tensor which states the batch-dim

Return type int|tf.Tensor

get_default_output_layer(*must_exist=True*)

Parameters **must_exist** (bool) – if it does not exist, will raise an exception

Return type LayerBase|None

Returns the default output layer

get_default_output_layer_name()

Return type str|None

Returns default output layer name if there is one, or None

get_default_target()

Returns e.g. "classes"

Return type str

get_extern_data(*key*, *mark_data_key_as_used=True*)

Returns Data and add the key to self.used_data_keys if mark_data_key_as_used. :param str key: :param bool mark_data_key_as_used: :rtype: Data

get_global_train_step(*session*)

Parameters **session** (*tf.Session*) –

Return type *int*

get_num_params()

Returns number of model parameters, i.e. total dimension

Return type *int*

get_objective()

get_output_layers()

Return type *list*[*LayerBase*]

get_param_values_dict(*session*)

Parameters **session** (*tf.Session*) –

Returns dict: layer_name -> param_name -> variable numpy array

Return type dict[str,dict[str,numpy.ndarray]]

Note that this excludes auxiliary params.

get_params_list()

Returns list of model variables, i.e. from all the layers, excluding auxiliary vars like global_step

Return type *list*[tf.Variable]

get_params_nested_dict()

Returns dict: layer_name -> param_name -> variable

Return type dict[str,dict[str,tf.Variable]]

get_params_serialized(*session*)

Parameters **session** (*tf.Session*) –

Return type *TFNetworkParamsSerialized*

get_saveable_params_list()

Returns list of model variables or SaveableObject, to save/restore

Return type *list*[tf.Variableltensorflow.python.training.saver.BaseSaverBuilder.SaveableObject]

get_search_choices(*sources=None*, *src=None*, *_visited=None*)
Recursively searches through all sources, and if there is a ChoiceLayer, returns it. Could also go to the parent network.

Parameters

- **src** (*LayerBase|None*) –
- **sources** (*list[LayerBase]|None*) –
- **_visited** (*set[LayerBase]|None*) – keep track about visited layers in case there are circular deps

Returns (direct or indirect) source LayerBase which has search_choices, or None

Return type LayerBaselNone

get_seq_tags(*mark_data_key_as_used=True*)

get_total_constraints()

get_total_loss()

Return type intltf.Tensor

Returns 0 if no loss, or tf.Tensor

get_trainable_params()

Returns list of variables

Return type *list*[tf.Variable]

get_used_targets()

Returns sorted list of targets

Return type *list*[str]

get_var_assigner(*var*)

Parameters **var** (*tf.Variable*) –

initialize_params(*session*)

Parameters **session** (*tf.Session*) –

Note: This will create a new node to the graph for each call! And it will overwrite also the already initialized variables. So you should call this only once after network construction and before you maybe load some of the params from external sources. If you know that you will load all params explicitly, you would not need to call this function.

load_params_from_file(*filename*, *session*)
Will save the model parameters to the filename. Note that the model parameters live inside the current TF session. :param str filename: :param tf.Session session:

maybe_construct_objective()

print_network_info(*name='Network'*)

reset_saver()
Resets the `tf.train.Saver` object which will be used for *load_params_from_file()* and *save_params_to_file()*. Warning: Don't repeat that too often as it will always create new ops in the computation graph.

save_params_to_file(*filename*, *session*)
Will save the model parameters to the filename. Note that the model parameters live inside the current TF session. :param str filename: :param tf.Session session:

set_global_train_step(*step*, *session*)

Parameters

- **step** (int) –
- **session** (*tf.Session*) –

set_param_values_by_dict(*values_dict*, *session*)

Parameters

- **values_dict** (*dict[str,dict[str,numpy.ndarray]]*) –
- **session** (*tf.Session*) –

Note that this excludes auxiliary params.

set_params_by_serialized(*serialized*, *session*)

Parameters

- **serialized** (TFNetworkParamsSerialized) –
- **session** (*tf.Session*) –

class TFNetwork.**TFNetworkParamsSerialized**(*values_dict*, *global_train_step*)
Holds all the params as numpy arrays, including auxiliary params.

Parameters

- **values_dict** (*dict[str,dict[str,numpy.ndarray]]*) –
- **global_train_step** (int) –

TFNetworkLayer

class TFNetworkLayer.**AccumulateMeanLayer**(*exp_average*, *axes='bt'*, *initial_value=None*, *is_prob_distribution=None*, ***kwargs*)
Accumulates the mean of the input (in training). It's similar to *ReduceLayer*

Parameters

- **exp_average** (float) – momentum in exponential average calculation
- **axes** (*int|list[str]|str*) – the axes to reduce. must contain batch and time.
- **initial_value** (float) – how to initialize the variable which accumulates the mean
- **is_prob_distribution** (bool) – if provided, better default for initial_value

classmethod get_out_data_from_opts(*axes='bt'*, ***kwargs*)

layer_class = 'accumulate_mean'

class TFNetworkLayer.**ActivationLayer**(*activation*, ***kwargs*)
This layer just applies an activation function.

Parameters **activation** (*str*) – e.g. "relu", "tanh", etc

layer_class = 'activation'

class TFNetworkLayer.**BatchNormLayer**(***kwargs*)
Implements batch-normalization (http://arxiv.org/abs/1502.03167) as a separate layer.

All kwargs which are present in our base class are passed to our base class. All remaining kwargs are used for self.batch_norm().

layer_class = 'batch_norm'

class TFNetworkLayer.**BinaryCrossEntropy**(*base_network*)

Parameters **base_network** (TFNetwork.TFNetwork) –

class_name = 'bin_ce'

get_value()

class TFNetworkLayer.**CombineDimsLayer**(*axes*, ***kwargs*)
Combines multiple dimensions. See also MergeDimsLayer.

Parameters **axis** (*int|list[int]|str*) – one axis or multiple axis to reduce. this is counted with batch-dim, which by default is axis 0 (see enforce_batch_dim_axis). it also accepts the special tokens "B"|"batch", "spatial", "spatial_except_time", or "F"|"feature"

classmethod get_out_data_from_opts(*axes*, *sources*, ***kwargs*)

layer_class = 'combine_dims'

class TFNetworkLayer.**CombineLayer**(*kind*, *sources*, *activation=None*, *with_bias=False*, ***kwargs*)
Applies some binary operation on all sources, such as addition.

Parameters

- **kind** (*str*) – e.g. "average" or "add"
- **sources** (list[LayerBase]) –
- **activation** (*str|None*) – if provided, activation function to apply, e.g. "tanh" or "relu"
- **with_bias** (bool) – if given , will add a bias

classmethod get_out_data_from_opts(*n_out=None*, *out_type=None*, *sources=()*, ***kwargs*)

layer_class = 'combine'

class TFNetworkLayer.**CompareLayer**(*kind='equal'*, *value=None*, ***kwargs*)
Compares (e.g. equality check) all the sources element-wise.

Parameters

- **kind** (*str*) – e.g. "equal"
- **value** (*float|int|None*) – if specified, will also compare to this

classmethod get_out_data_from_opts(*n_out=None*, *out_type=None*, *sources=()*, ***kwargs*)

layer_class = 'compare'

class TFNetworkLayer.**ConstantLayer**(*sources*, *value=0*, *dtype=None*, ***kwargs*)
Output is a constant value.

classmethod get_out_data_from_opts(*name*, *dtype='float32'*, ***kwargs*)

layer_class = 'constant'

class TFNetworkLayer.**ConvLayer**(*n_out*, *filter_size*, *padding*, *strides=1*, *dilation_rate=1*, *input_expand_dims=0*, *input_add_feature_dim=False*, *input_split_feature_dim=None*, *with_bias=False*, *activation=None*, *forward_weights_init='glorot_uniform'*, *bias_init=0.0*, ***kwargs*)

A generic convolution layer which supports 1D, 2D and 3D convolution. Pooling can be done in the separate "pool" layer.

Parameters

- **n_out** (*int*) – number of outgoing features
- **filter_size** (*tuple[int]*) – (width,), (height,width) or (depth,height,width) for 1D/2D/3D conv. the input data ndim must match, or you can add dimensions via input_expand_dims or input_add_feature_dim. it will automatically swap the batch-dim to the first axis of the input data.
- **padding** (*str*) – "same" or "valid"
- **strides** (*int|tuple[int]*) – strides for the spatial dims, i.e. length of this tuple should be the same as filter_size, or a single int.
- **input_expand_dims** (*int*) – number of dynamic dims to add to the input
- **input_add_feature_dim** (*bool*) – will add a dim at the end and use input-feature-dim == 1, and use the original input feature-dim as a spatial dim.
- **input_split_feature_dim** (*None|int*) – if set, like input_add_feature_dim it will add a new feature dim which is of value input_split_feature_dim, and the original input feature dim will be divided by input_split_feature_dim, thus it must be a multiple of that value.
- **with_bias** (*bool*) – if True, will add a bias to the output features
- **activation** (*None|str*) – if set, will apply this function at the end

classmethod calc_out_dim(*in_dim*, *filter_size*, *stride*, *padding*, *dilation_rate=1*)

Parameters

- **in_dim** (*int|tf.Tensor*) – dimension in some axis
- **filter_size** (*int*) – e.g. 2, for the corresponding axis
- **stride** (*int*) – e.g. 1, for the corresponding axis
- **dilation_rate** (*int*) – e.g. 1
- **padding** (*str*) – "valid" or "same"

Returns the output dimension

Return type *int*

classmethod get_out_data_from_opts(***kwargs*)

layer_class = 'conv'

recurrent = True

class TFNetworkLayer.**CopyLayer**(***kwargs*)

This layer does nothing, it copies its input. If multiple sources are provided, they are concatenated in the feature-dim.

classmethod get_out_data_from_opts(*name*, *sources=()*, *out_type=None*, *n_out=None*, ***kwargs*)

layer_class = 'copy'

class TFNetworkLayer.**CrossEntropyLoss**(*base_network*)
Cross-Entropy loss. Basically sum(target * log(output)).

Parameters **base_network** (TFNetwork.TFNetwork) –

class_name = 'ce'

get_value()

class TFNetworkLayer.**CtcLoss**(*target_collapse_repeated=False*, *auto_clip_target_len=False*, ***kwargs*)
Connectionist Temporal Classification (CTC) loss. Basically a wrapper around tf.nn.ctc_loss.

Parameters

- **target_collapse_repeated** (bool) – like preprocess_collapse_repeated option for CTC. used for sparse_labels().
- **auto_clip_target_len** (bool) – see self._get_target_sparse_labels().

class_name = 'ctc'

get_auto_output_layer_dim(*target_dim*)

get_error()

get_value()

init(***kwargs*)

recurrent = True

class TFNetworkLayer.**EditDistanceLoss**(*debug_print=False*, ***kwargs*)
Note that this loss is not differentiable, thus it's only for keeping statistics.

class_name = 'edit_distance'

get_error()

get_value()

init(*output*, *output_with_activation=None*, *target=None*)

Parameters

- **output** (Data) – generated output
- **output_with_activation** (*OutputWithActivation|None*) –
- **target** (Data) – reference target from dataset

recurrent = True

class TFNetworkLayer.**ElemwiseProdLayer**(*axes*, *size=None*, ***kwargs*)
Element-wise product in some axes. Microsoft calls this "static attention", in Deep Conv. NN with Layer-wise Context Expansion and Attention (LACE).

Parameters

- **axes** (*str|list[str]*) – e.g. "spatial", but all those axes must be of fixed dimension
- **size** (*tuple[int]*) – for double-checking, you can explicitly provide the size

classmethod get_out_data_from_opts(*name*, *sources*, ***kwargs*)

layer_class = 'elemwise_prod'

class TFNetworkLayer.**ExpandDimsLayer**(*axis*, *dim=1*, ***kwargs*)
Adds some axis.

Parameters

- **axis** (*str|int*) – axis to add, e.g. "F"|"feature" or "spatial". if this is an integer, the input data is first converted into batch-major mode, and then this is counted with batch-dim.
- **dim** (int) – dimension of new axis (1 by default)

classmethod get_out_data_from_opts(*name*, *axis*, *dim=1*, *sources=()*, ***kwargs*)

layer_class = 'expand_dims'

class TFNetworkLayer.**ExternSprintLoss**(*sprint_opts*, ***kwargs*)
The loss is calculated by an extern Sprint instance.

Parameters **sprint_opts** (*dict[str]*) –

class_name = 'sprint'

get_error()

get_value()

recurrent = True

class TFNetworkLayer.**FastBaumWelchLoss**(*sprint_opts*, ***kwargs*)
The loss is calculated via fast_baum_welch(). The automata are created by an extern Sprint instance.

Parameters **sprint_opts** (*dict[str]*) –

class_name = 'fast_bw'

get_error()

get_value()

recurrent = True

class TFNetworkLayer.**FramewiseStatisticsLayer**(*sil_label_idx*, *histogram_num_bins=20*, ***kwargs*)
Collects various statistics (such as FER, etc) on the sources. The tensors will get stored in self.stats which will be collected by TFEngine.

classmethod get_out_data_from_opts(***kwargs*)

layer_class = 'framewise_statistics'

class TFNetworkLayer.**FsaLayer**(***kwargs*)

layer_class = 'fsa'

class TFNetworkLayer.**GatingLayer**(*activation*, *gate_activation='sigmoid'*, ***kwargs*)
Splits the output into two equal parts, applies the gate_activation (sigmoid by default) on the one part, some other activation (e.g. tanh) on the other part and then element-wise multiplies them. Thus, the output dimension is input-dimension / 2.

classmethod get_out_data_from_opts(*name*, *sources*, *n_out=None*, ***kwargs*)

layer_class = 'gating'

class TFNetworkLayer.**GenericCELoss**(***kwargs*)

class_name = 'generic_ce'

get_value()

class TFNetworkLayer.**InternalLayer**(*name*, *network*, *output=None*, *n_out=None*, *out_type=None*, *sources=()*, *target=None*, *loss=None*, *loss_scale=1.0*, *size_target=None*, *L2=None*, *is_output_layer=None*, *batch_norm=False*, *spatial_smoothing=0.0*, *initial_output=None*, *rec_previous_layer=None*, *trainable=True*)

This is not supposed to be used by the user. It is used by some code to construct a wrapper layer or so.

Parameters

- **name** (*str*) –
- **network** (TFNetwork.TFNetwork) –
- **output** (Data) –
- **n_out** (*None|int*) – output dim
- **out_type** (*dict[str]*) – kwargs for Data class. more explicit than n_out.
- **sources** (list[LayerBase]) – via self.transform_config_dict()
- **target** (*str|None*) – if some loss is set, this is the target data-key, i.e. network.extern_data.get_data(target) alternatively, this also can be a layer name.
- **size_target** (*str|None*) – like target but this is only used to set our output size in case of training
- **loss** (*Loss|None*) – via self.transform_config_dict()
- **loss_scale** (float) – scale factor for loss (1.0 by default)
- **L2** (*float|None*) – for constraints
- **is_output_layer** (*bool|None*) –
- **batch_norm** (*bool|dict*) – see self.batch_norm()
- **initial_output** (*str|float*) – used for recurrent layer, see self.get_rec_initial_output()
- **rec_previous_layer** (*LayerBase|None*) – via the recurrent layer, layer (template) which represents the past of us
- **trainable** (bool) – whether the parameters of this layer will be trained

class TFNetworkLayer.**L1Loss**(*base_network*)

L1-distance loss. sum(target - output).

Parameters **base_network** (TFNetwork.TFNetwork) –

class_name = 'l1'

get_value()

class TFNetworkLayer.**LayerBase**(*name*, *network*, *output=None*, *n_out=None*, *out_type=None*, *sources=()*, *target=None*, *loss=None*, *loss_scale=1.0*, *size_target=None*, *L2=None*, *is_output_layer=None*, *batch_norm=False*, *spatial_smoothing=0.0*, *initial_output=None*, *rec_previous_layer=None*, *trainable=True*)

This is the base class for all layers. Every layer by default has a list of source layers *sources* and defines *self.output* which is of type Data. It shares some common functionality across all layers, such as explicitly defining the output format, some parameter regularization, and more.

If you want to implement your own layer:

```
class YourOwnLayer(_ConcatInputLayer):  # e.g. either _ConcatInputLayer or␣
↪LayerBase as a base
    " some docstring "
    layer_class = "your_layer_name"

    def __init__(self, your_kwarg1, your_opt_kwarg2=None, **kwargs):
        " docstring, document the args! "
        super(YourOwnLayer, self).__init__(**kwargs)
        # Now we need to set self.output, which must be of type :class:`Data`.
        # It is set at this point to whatever we got from `selfget_out_data_from_
↪opts()`,
        # so it is enough if we set self.output.placeholder and self.output.size_
↪placeholder,
        # but we could also reset self.output.
        self.output.placeholder = self.input_data.placeholder + 42  # whatever␣
↪you want to do
        # If you don't modify the sizes (e.g. sequence-length), just copy the␣
↪input sizes.
        self.output.size_placeholder = self.input_data.size_placeholder.copy()

    @classmethod
    def get_out_data_from_opts(cls, **kwargs):
        " This is supposed to return a :class:`Data` instance as a template,␣
↪given the arguments. "
        # example, just the same as the input:
        return get_concat_sources_data_template(kwargs["sources"], name="%s_output
↪" % kwargs["name"])
```

Parameters

- **name** (*str*) –
- **network** (TFNetwork.TFNetwork) –
- **output** (Data) –
- **n_out** (*None|int*) – output dim
- **out_type** (*dict[str]*) – kwargs for Data class. more explicit than n_out.
- **sources** (list[LayerBase]) – via self.transform_config_dict()
- **target** (*str|None*) – if some loss is set, this is the target data-key, i.e. network.extern_data.get_data(target) alternatively, this also can be a layer name.
- **size_target** (*str|None*) – like target but this is only used to set our output size in case of training
- **loss** (*Loss|None*) – via self.transform_config_dict()
- **loss_scale** (float) – scale factor for loss (1.0 by default)
- **L2** (*float|None*) – for constraints
- **is_output_layer** (*bool|None*) –
- **batch_norm** (*bool|dict*) – see self.batch_norm()
- **initial_output** (*str|float*) – used for recurrent layer, see self.get_rec_initial_output()
- **rec_previous_layer** (*LayerBase|None*) – via the recurrent layer, layer (template) which represents the past of us

- **trainable** (bool) – whether the parameters of this layer will be trained

add_param(*param*, *custom_update=None*)

Parameters

- **param** (*tf.Variable*) –
- **custom_update** (*None|CustomUpdate*) – will be applied in training, instead of taking the gradient

Returns param

:rtype tf.Variable

batch_norm(*data*, *use_shift=True*, *use_std=True*, *use_sample=0.0*, *force_sample=False*, *momentum=0.99*, *epsilon=0.001*, *sample_mean=None*, *sample_variance=None*, *gamma=None*, *beta=None*)

Parameters

- **data** (Data) –
- **use_shift** (bool) –
- **use_std** (bool) –
- **use_sample** (float) – defaults to 0.0 which is used in training
- **force_sample** (bool) – even in eval, use the use_sample factor
- **momentum** (float) – for the running average of sample_mean and sample_std
- **epsilon** (float) –
- **sample_mean** (*tf.Tensor*) –
- **sample_variance** (*tf.Tensor*) –
- **gamma** (*tf.Tensor*) –
- **beta** (*tf.Tensor*) –

Return type tf.Tensor

http://arxiv.org/abs/1502.03167

Also see: tf.nn.batch_normalization() https://github.com/deepmind/sonnet/blob/master/sonnet/python/modules/batch_norm.py

classmethod **cls_get_tf_scope_name**(*name*)

Parameters **name** (*str*) – layer name

Returns scope name, might be just name

get_absolute_name_scope_prefix()

Returns e.g. "output/", always with "/" at end

Return type str

get_batch_dim()

The batch dim by this layer, not taken from our output but calculated. Normally it is self.network.get_batch_dim() but if we do search and there was a choice layer, it it multiplied by the beam size. :return: batch dim * beam size :rtype: tf.Tensor

get_constraints_value()

get_dep_layers()

Returns list of layers this layer depends on. normally this is just self.sources but e.g. the attention layer in addition has a base, etc.

Return type *list*[*LayerBase*]

get_error_value()

Returns usually the frame error rate, or None if not defined

Return type tf.Tensor | None

get_hidden_state()
If this is a recurrent layer, this would return the hidden state. This is used e.g. for the RnnCellLayer class. :rtype: tf.Tensor | list[tf.Tensor] | None :return: optional tensor(s) with shape (time, batch, dim)

get_last_hidden_state()
If this is a recurrent layer, this would return the last hidden state. If not, as a fallback, we recursively check our sources. :rtype: tf.Tensor | None :return: optional tensor with shape (batch, dim)

get_loss_normalization_factor()

get_loss_value()

Returns the loss, a scalar value, or None if not set. not multiplied by loss_scale

Return type tf.Tensor | None

classmethod get_out_data_from_opts(***kwargs*)
Gets a Data template (i.e. shape etc is set but not the placeholder) for our __init__ args. The purpose of having this as a separate classmethod is to be able to infer the shape information without having to construct the layer. This function should not create any nodes in the computation graph.

Parameters **kwargs** – all the same kwargs as for self.__init__()

Returns Data template (placeholder not set)

Return type *Data*

get_output_spatial_smoothing_energy()

get_param_values_dict(*session*)

Parameters **session** (*tf.Session*) –

Returns dict name -> values

Return type dict[str,numpy.ndarray]

get_params_l2_norm()

classmethod get_rec_initial_extra_outputs(*batch_dim*, ***kwargs*)

classmethod get_rec_initial_output(*batch_dim*, *name*, *output*, *initial_output=None*, ***kwargs*)

get_saveable_params_dict()

Returns params and saveable_param_replace resolved

Return type dict[str,tf.Variable|tensorflow.python.training.saver.BaseSaverBuilder.SaveableObject]

get_search_beam_size()

Returns beam size if there was a choice layer and we do search

Return type int|None

is_output_layer()
Some code differs between an output layer and other layers. It is a bit arbitrary what we define as output layer. :rtype: bool

layer_class = None

post_init()
This gets called right after self.__init__().

recurrent = False

set_param_values_by_dict(*values_dict*, *session*)

Parameters

- **values_dict** (*dict[str,numpy.ndarray]*) –
- **session** (*tf.Session*) –

tf_scope_name

classmethod transform_config_dict(*d*, *network*, *get_layer*)

Parameters

- **d** (*dict[str]*) – will modify inplace
- **network** (TFNetwork.TFNetwork) –
- **-> LayerBase) get_layer** (*((str)*) – function to get or construct another layer

Will modify *d* such that it becomes the kwargs for *self.__init__()*. Mostly leaves *d* as-is. This is used by TFNetwork.construct_from_dict().

class TFNetworkLayer.**LinearLayer**(*activation*, *with_bias=True*, *grad_filter=None*, *forward_weights_init='glorot_uniform'*, *bias_init=0.0*, ***kwargs*)
Linear/forward/fully-connected/1x1-conv layer. Does a linear transformation on the feature-dimension of the input with an optional bias term and an optional activation function.

Parameters

- **activation** (*str|None*) – e.g. "relu", or None
- **with_bias** (bool) –
- **grad_filter** (*float|None*) – if grad norm is higher than this threshold (before activation), the grad is removed
- **forward_weights_init** (*str*) – see *TFUtil.get_initializer()*
- **recurrent_weights_init** (*str*) – see *TFUtil.get_initializer()*
- **bias_init** (*str|float*) – see *TFUtil.get_initializer()*

layer_class = 'linear'

class TFNetworkLayer.**Loss**(*base_network*)
Base class for all losses.

Parameters **base_network** (TFNetwork.TFNetwork) –

class_name = None

get_auto_output_layer_dim(*target_dim*)

Parameters **target_dim** (int) –

Returns normally just the same as target_dim. e.g. for CTC, we would add 1 for the blank label

Return type *int*

get_error()

Returns frame error rate as a scalar value

Return type tf.Tensor

get_normalization_factor()

Returns factor as a float scalar, usually 1.0 / num_frames. see self.reduce_func.

Return type tf.Tensor

get_value()

Returns loss as a scalar value

Return type tf.Tensor|None

init(*output*, *output_with_activation=None*, *target=None*)

Parameters

- **output** (Data) – generated output
- **output_with_activation** (*OutputWithActivation|None*) –
- **target** (Data) – reference target from dataset

recurrent = False

class TFNetworkLayer.**MeanSquaredError**(*base_network*)
The generic mean squared error loss function

Parameters **base_network** (TFNetwork.TFNetwork) –

class_name = 'mse'

get_value()

class TFNetworkLayer.**MergeDimsLayer**(*axes*, *n_out=None*, ***kwargs*)
Merges a list of axes into a single one. E.g. input is (batch, width, height, dim) and axes=(1,2), then we get (batch, width*height, dim). Or input is (batch, time, height, dim) and axes="except_time", then we get (batch, time, height*dim). See also CombineDimsLayer.

Parameters

- **axes** (*str|list[str]|list[int]*) – see Data.get_axes_from_description(), e.g. "except_time"
- **n_out** (*int|None*) –

classmethod get_out_data_from_opts(*name*, *axes*, *sources=()*, *n_out=None*, *out_type=None*, ***kwargs*)

layer_class = 'merge_dims'

class TFNetworkLayer.**PadLayer**(*axes*, *padding*, *value=None*, *mode='constant'*, ***kwargs*)
Adds (e.g. zero) padding in some axis or axes.

Parameters

- **axes** (*str|list[str]*) – e.g. "F" etc. see Dataset.get_axes_from_description().
- **padding** (list[(int,int)]|(int,int)|*int*) – how much to pad left/right in each axis

- **value** (*int | float*) – what constant value to pad, with mode==”constant”
- **mode** (*str*) – “constant”, “reflect” or “symmetric”

classmethod get_out_data_from_opts(*name*, *axes*, *padding*, *sources=()*, ***kwargs*)

layer_class = ‘pad’

class TFNetworkLayer.**PoolLayer**(*mode*, *pool_size*, *padding='VALID'*, *dilation_rate=1*, *strides=None*, ***kwargs*)

A generic N-D pooling layer. This would usually be done after a convolution for down-sampling.

Parameters

- **mode** (*str*) – “max” or “avg”
- **pool_size** (*tuple[int]*) – shape of the window of each reduce
- **padding** (*str*) – “valid” or “same”
- **dilation_rate** (*tuple[int] | int*) –
- **strides** (*tuple[int] | int | None*) – in contrast to tf.nn.pool, the default (if it is None) will be set to pool_size

classmethod get_out_data_from_opts(*name*, *pool_size*, *strides=None*, *dilation_rate=1*, *sources=()*, *padding='VALID'*, ***kwargs*)

layer_class = ‘pool’

recurrent = True

class TFNetworkLayer.**PrefixInTimeLayer**(*prefix=0.0*, *repeat=1*, ***kwargs*)

Parameters

- **prefix** (*float | str*) – either some constant or another layer
- **repeat** (*int*) – how often to repeat the prefix

layer_class = ‘prefix_in_time’

class TFNetworkLayer.**ReduceLayer**(*mode*, *axes=None*, *axis=None*, *keep_dims=False*, *enforce_batch_dim_axis=None*, ***kwargs*)

This reduces some axis by using “sum” or “max”. It’s basically a wrapper around tf.reduce_sum or tf.reduce_max.

Parameters

- **mode** (*str*) – “sum” or “max” or “mean”
- **axes** (*int | list[int] | str*) – one axis or multiple axis to reduce. this is counted with batch-dim, which by default is axis 0 (see enforce_batch_dim_axis). it also accepts the special tokens “B”|”batch”, “spatial”, “spatial_except_time”, or “F”|”feature”
- **axis** (*int | list[int] | str*) – for compatibility, can be used instead of axes
- **keep_dims** (*bool*) – if dimensions should be kept (will be 1)
- **enforce_batch_dim_axis** (*int*) – will swap the batch-dim-axis of the input with the given axis. e.g. 0: will convert the input into batch-major format if not already like that.

classmethod get_axes(*axis*, *input_data*)

Parameters

- **axis** – see self.__init__()
- **input_data** (*Data*) –

Returns list of axes

Return type *list*[*int*]

classmethod get_out_data_from_opts(*name*, *sources*, *axes=None*, *axis=None*, *keep_dims=False*, *enforce_batch_dim_axis=None*, ***kwargs*)

layer_class = 'reduce'

classmethod need_enforce_batch_dim_axis(*axes*)

Parameters **axes** (*int|list[int]|str*) –

Returns if any integer is in axes, thus we should have a fixed dimension layout

Return type *bool*

class TFNetworkLayer.**ResizeLayer**(*factor*, *axis*, *kind='nn'*, ***kwargs*)

Resizes the input, i.e. upsampling or downsampling. Supports different kinds, such as linear interpolation or nearest-neighbor.

Parameters

- **factor** (int) –
- **axis** (*str|int*) – the axis to resize, counted with batch-dim. can also be "T" for time
- **kind** (*str*) – "linear", "nn"/"nearest_neighbor", "cubic"

classmethod get_out_data_from_opts(*factor*, *axis*, *sources*, ***kwargs*)

layer_class = 'resize'

class TFNetworkLayer.**SearchChoices**(*owner*, *src_beams=None*, *beam_size=None*, *is_decided=False*)

Parameters

- **owner** (LayerBase) –
- **src_beams** (*tf.Tensor|None*) – (batch, beam) -> src beam index
- **beam_size** (*int|None*) –
- **is_decided** (bool) – by decide layer

filter_seqs(*seq_filter*)

Parameters **seq_filter** (*tf.Tensor*) – (batch, beam) of type bool

set_beam_scores(*scores*)

Parameters **scores** (*tf.Tensor*) – (batch, beam) -> log score

set_beam_scores_from_own_rec()

set_beam_scores_from_rec(*rev_vars_outputs*)

Parameters **rev_vars_outputs** (*dict[str,tf.Tensor]*) –

src_layer

Return type *LayerBase*

class TFNetworkLayer.**SliceLayer**(*axis*, *slice_start=None*, *slice_end=None*, *slice_step=None*, ***kwargs*)

Slicing on the input, i.e. x[start:end:step] in some axis.

Parameters

- **axis** (*int | str*) –
- **axis_kind** (*str | None*) – "T" for time, "B" for batch, "F" for feature
- **slice_start** (*int | None*) –
- **slice_end** (*int | None*) –
- **slice_step** (*int | None*) –

classmethod get_out_data_from_opts(*name*, *axis*, *sources=()*, *slice_start=None*, *slice_end=None*, *slice_step=None*, ***kwargs*)

layer_class = 'slice'

class TFNetworkLayer.**SoftmaxLayer**(*activation='softmax'*, ***kwargs*)
Just a LinearLayer with activation="softmax" by default.

layer_class = 'softmax'

class TFNetworkLayer.**SourceLayer**(*network*, *data_key=None*, *sources=()*, ***kwargs*)

Parameters

- **network** (TFNetwork.TFNetwork) –
- **data_key** (*str | None*) –
- **sources** (*tuple*) –

classmethod get_out_data_from_opts(*network*, *data_key=None*, ***kwargs*)

Parameters

- **network** (TFNetwork.TFNetwork) –
- **data_key** (*str | None*) –

Return type *Data*

layer_class = 'source'

class TFNetworkLayer.**SplitBatchTimeLayer**(*base*, ***kwargs*)
A very specific layer which expects to get input of shape (batch * time, ...) and converts it into (batch, time, ...), where it recovers the seq-lens from some other layer.

Parameters **base** (LayerBase) –

classmethod get_out_data_from_opts(*name*, *base*, *sources=()*, ***kwargs*)

layer_class = 'split_batch_time'

classmethod transform_config_dict(*d*, *network*, *get_layer*)

class TFNetworkLayer.**SplitDimsLayer**(*axis*, *dims*, ***kwargs*)
Splits one axis into multiple axes. E.g. if you know that your feature-dim is composed by a window, i.e. the input is (batch, time, window * feature), you can set axis="F", dims=(window, -1), and you will get the output (batch, time, window, feature).

Parameters

- **axis** (*str*) – e.g. "F"
- **dims** (*tuple[int]*) – what the axis should be split into. e.g. (window, -1)

classmethod get_out_data_from_opts(*name*, *axis*, *dims*, *sources=()*, ***kwargs*)

layer_class = 'split_dims'

class TFNetworkLayer.**SqueezeLayer**(*axis*, *enforce_batch_dim_axis=0*, ***kwargs*)
Removes an axis with dimension 1. This is basically a wrapper around tf.squeeze.

Parameters **axis** (*int|list[int]|str*) – one axis or multiple axis to squeeze. this is counted with batch-dim, which by default is axis 0 (see enforce_batch_dim_axis). it also accepts the special tokens "B"|"batch", "spatial", "spatial_except_time", or "F"|"feature"

classmethod get_out_data_from_opts(***kwargs*)

layer_class = 'squeeze'

class TFNetworkLayer.**SubnetworkLayer**(*subnetwork*, *concat_sources=True*, ***kwargs*)
You can define a whole subnetwork as a single layer by this class.

Parameters

- **network** (*dict[str,dict]*) – subnetwork as dict (JSON content). must have an "output" layer
- **concat_sources** (bool) – if we concatenate all sources into one, like it is standard for most other layers

get_constraints_value()

get_error_value()

get_last_hidden_state()

get_loss_value()

classmethod get_out_data_from_opts(*subnetwork*, *n_out=None*, *out_type=None*, ***kwargs*)

Parameters

- **subnetwork** (*dict[str,dict[str]]*) –
- **n_out** (*int|None*) –
- **out_type** (*dict[str]|None*) –

Return type *Data*

layer_class = 'subnetwork'

recurrent = True

class TFNetworkLayer.**WeightedSumLayer**(*axes*, *padding=None*, *size=None*, *keep_dims=None*, ***kwargs*)
Calculates a weighted sum, either over a complete axis of fixed dimension, or over some window. Can also do that for multiple axes.

Parameters

- **axes** (*str|list[str]*) – the axes to do the weighted-sum over
- **padding** (*str*) – "valid" or "same", in case of keep_dims=True
- **size** (*None|tuple[int]*) – the kernel-size. if left away, the axes must be of fixed dimension, and we will use keep_dims=False, padding="valid" by default. Otherwise, if given, you must also provide padding and keep_dims=True by default.
- **keep_dims** (bool) – if False, the axes will be squeezed away. see also *size*.

classmethod get_out_data_from_opts(*name*, *sources*, *axes*, *padding=None*, *size=None*, *keep_dims=None*, ***kwargs*)

layer_class = 'weighted_sum'

class TFNetworkLayer.**WindowLayer**(*window_size*, *axis='T'*, *padding='same'*, ***kwargs*)

Adds a window dimension. By default, uses the time axis and goes over it with a sliding window. The new axis for the window is created right after the time axis. Will always return as batch major mode. E.g. if the input is (batch, time, dim), the output is (batch, time, window_size, dim).

Parameters

- **window_size** (*int*) –
- **axis** (*str | int*) – see Data.get_axis_from_description()
- **padding** (*str*) – "same" or "valid"
- **kwargs** –

classmethod get_out_data_from_opts(*axis='T'*, *sources=()*, ***kwargs*)

layer_class = 'window'

recurrent = True

TFNetworkLayer.**concat_sources**(*src_layers*)

Parameters **src_layers** (*list[LayerBase]*) –

Returns data with placeholders set

Return type *Data*

TFNetworkLayer.**concat_sources_with_opt_dropout**(*src_layers*, *dropout=0*)

Parameters

- **src_layers** (*list[LayerBase]*) –
- **dropout** (*float*) – will be applied if train_flag is set

Returns data with placeholders set

Return type *Data*

TFNetworkLayer.**get_concat_sources_data_template**(*src_layers*, *name='concat_sources'*)

Parameters

- **src_layers** (*list[LayerBase]*) –
- **name** (*str*) – name of the Data

Returns data with no placeholders set

Return type *Data*

TFNetworkLayer.**get_layer_class**(*name*)

Parameters **name** (*str*) – matches layer_class

Return type (() -> LayerBase) | type[*LayerBase*] | LayerBase

TFNetworkLayer.**get_loss_class**(*loss*)

Parameters **loss** (*str*) – loss type such as "ce"

Return type () -> Loss | type[*Loss*] | Loss

TFNetworkRecLayer

class TFNetworkRecLayer.**AttentionBaseLayer**(*base*, ***kwargs*)

This is the base class for attention. This layer would get constructed in the context of one single decoder step. We get the whole encoder output over all encoder frames (the base), e.g. (batch,enc_time,enc_dim), and some current decoder context, e.g. (batch,dec_att_dim), and we are supposed to return the attention output, e.g. (batch,att_dim).

Some sources: * Bahdanau, Bengio, Montreal, Neural Machine Translation by Jointly Learning to Align and Translate, 2015, https://arxiv.org/abs/1409.0473 * Luong, Stanford, Effective Approaches to Attention-based Neural Machine Translation, 2015, https://arxiv.org/abs/1508.04025

-> dot, general, concat, location attention; comparison to Bahdanau

- https://github.com/ufal/neuralmonkey/blob/master/neuralmonkey/decoders/decoder.py
- https://google.github.io/seq2seq/ https://github.com/google/seq2seq/blob/master/seq2seq/contrib/seq2seq/decoder.py https://github.com/google/seq2seq/blob/master/seq2seq/decoders/attention_decoder.py
- https://github.com/deepmind/sonnet/blob/master/sonnet/python/modules/attention.py

Parameters **base** (LayerBase) – encoder output to attend on

get_base_weight_last_frame()

From the base weights (see self.get_base_weights(), must return not None) takes the weighting of the last frame in the time-axis (according to sequence lengths).

Returns shape (batch,) -> float (number 0..1)

Return type tf.Tensor

get_base_weights()

We can formulate most attentions as some weighted sum over the base time-axis.

Returns the weighting of shape (batch, base_time), in case it is defined

Return type tf.Tensor|None

get_dep_layers()

classmethod **get_out_data_from_opts**(*base*, *n_out=None*, ***kwargs*)

Parameters **base** (LayerBase) –

Return type *Data*

classmethod **transform_config_dict**(*d*, *network*, *get_layer*)

class TFNetworkRecLayer.**ChoiceLayer**(*beam_size*, ***kwargs*)

This layer represents a choice to be made in search during inference, such as choosing the top-k outputs from a log-softmax for beam search. During training, this layer can return the true label. This is supposed to be used inside the rec layer. This can be extended in various ways.

Assume that we get input (batch,dim) from a log-softmax. Assume that each batch is already a choice via search. In search with a beam size of N, we would output sparse (batch=N,) and scores for each.

classmethod **get_out_data_from_opts**(*target*, *network*, *beam_size*, ***kwargs*)

classmethod **get_rec_initial_extra_outputs**(*network*, *beam_size*, ***kwargs*)

Parameters

- **network** (TFNetwork.TFNetwork) –

- **beam_size** (int) –

Return type dict[str,tf.Tensor]

layer_class = 'choice'

class TFNetworkRecLayer.**DecideLayer**(***kwargs*)

This is kind of the counter-part to the choice layer. This only has an effect in search mode. E.g. assume that the input is of shape (batch * beam, time, dim) and has search_sources set. Then this will output (batch, time, dim) where the beam with the highest score is selected. Thus, this will do a decision based on the scores. In will convert the data to batch-major mode.

classmethod decide(*src*, *output=None*, *name=None*)

Parameters

- **src** (LayerBase) – with search_choices set. e.g. input of shape (batch * beam, time, dim)
- **output** (*Data|None*) –
- **name** (*str|None*) –

Returns best beam selected from input, e.g. shape (batch, time, dim)

Return type *Data*

classmethod get_out_data_from_opts(*name*, *sources*, *network*, ***kwargs*)

Parameters

- **name** (*str*) –
- **sources** (list*[LayerBase]*) –
- **network** (TFNetwork.TFNetwork) –

Return type *Data*

layer_class = 'decide'

class TFNetworkRecLayer.**DotAttentionLayer**(***kwargs*)

Classic global attention: Dot-product as similarity measure between base_ctx and source.

layer_class = 'dot_attention'

class TFNetworkRecLayer.**GaussWindowAttentionLayer**(*window_size*, *std=1.0*, *inner_size=None*, *inner_size_step=0.5*, ***kwargs*)

Interprets the incoming source as the location (float32, shape (batch,)) and returns a gauss-window-weighting of the base around the location. The window size is fixed (TODO: but the variance can optionally be dynamic).

Parameters

- **window_size** (int) – the window size where the Gaussian window will be applied on the base
- **std** (float) – standard deviation for Gauss
- **inner_size** (*int|None*) – if given, the output will have an additional dimension of this size, where t is shifted by +/- inner_size_step around. e.g. [t-1,t-0.5,t,t+0.5,t+1] would be the locations with inner_size=5 and inner_size_step=0.5.
- **inner_size_step** (float) – see inner_size above

classmethod get_out_data_from_opts(*inner_size=None*, ***kwargs*)

layer_class = 'gauss_window_attention'

class TFNetworkRecLayer.**GetLastHiddenStateLayer**(*n_out*, *combine='concat'*, ***kwargs*)

Will combine (concat or add or so) all the last hidden states from all sources.

Parameters

- **n_out** (int) – dimension. output will be of shape (batch, n_out)
- **combine** (*str*) – "concat" or "add"

get_last_hidden_state()

classmethod get_out_data_from_opts(*n_out*, ***kwargs*)

layer_class = 'get_last_hidden_state'

class TFNetworkRecLayer.**GlobalAttentionContextBaseLayer**(*base_ctx*, ***kwargs*)

Parameters **base_ctx** (LayerBase) – encoder output used to calculate the attention weights

get_dep_layers()

classmethod transform_config_dict(*d*, *network*, *get_layer*)

class TFNetworkRecLayer.**RecLayer**(*unit='lstm'*, *direction=None*, *input_projection=True*, *initial_state=None*, *max_seq_len=None*, *forward_weights_init=None*, *recurrent_weights_init=None*, *bias_init=None*, ***kwargs*)

Recurrent layer, has support for several implementations of LSTMs (via `unit` argument), see *TensorFlow LSTM benchmark* (http://returnn.readthedocs.io/en/latest/tf_lstm_benchmark.html), and also GRU, or simple RNN.

A subnetwork can also be given which will be evaluated step-by-step, which can use attention over some separate input, which can be used to implement a decoder in a sequence-to-sequence scenario.

Parameters

- **unit** (*str|dict[str,dict[str]]*) – the RNNCell/etc name, e.g. "nativelstm". see comment below. alternatively a whole subnetwork, which will be executed step by step, and which can include "prev" in addition to "from" to refer to previous steps.
- **direction** (*int|None*) – None|1 -> forward, -1 -> backward
- **input_projection** (bool) – True -> input is multiplied with matrix. False only works if same input dim
- **initial_state** (*LayerBase|None*) –
- **max_seq_len** (int) – if unit is a subnetwork
- **forward_weights_init** (*str*) – see *TFUtil.get_initializer()*
- **recurrent_weights_init** (*str*) – see *TFUtil.get_initializer()*
- **bias_init** (*str*) – see *TFUtil.get_initializer()*

static convert_cudnn_canonical_to_lstm_block(*reader*, *prefix*, *target='lstm_block_wrapper/'*)

This assumes CudnnLSTM currently, with num_layers=1, input_mode="linear_input", direction='unidirectional'!

Parameters

- **reader** (*tf.train.CheckpointReader*) –
- **prefix** (*str*) – e.g. "layer2/rec/"
- **target** (*str*) – e.g. "lstm_block_wrapper/" or "rnn/lstm_cell/"

Returns dict key -> value, {".../kernel": ..., ".../bias": ...} with prefix

Return type dict[str,numpy.ndarray]

get_absolute_name_scope_prefix()

get_constraints_value()

get_dep_layers()

get_error_value()

get_last_hidden_state()

get_loss_normalization_factor()

get_loss_value()

classmethod get_out_data_from_opts(*unit*, *sources=()*, *initial_state=None*, ***kwargs*)

classmethod get_rnn_cell_class(*name*)

Parameters **name** (*str*) – cell name, minus the "Cell" at the end

Return type () -> tensorflow.contrib.rnn.RNNCell

layer_class = 'rec'

recurrent = True

classmethod transform_config_dict(*d*, *network*, *get_layer*)

Parameters

- **d** (*dict[str]*) – will modify inplace
- **network** (TFNetwork.TFNetwork) –
- **-> LayerBase) get_layer** (*((str)*) – function to get or construct another layer

class TFNetworkRecLayer.**RnnCellLayer**(*n_out*, *unit*, *initial_state=None*, *unit_opts=None*, ***kwargs*)

Wrapper around tf.contrib.rnn.RNNCell. This will operate a single step, i.e. there is no time dimension, i.e. we expect a (batch,n_in) input, and our output is (batch,n_out).

Parameters

- **n_out** (int) – so far, only output shape (batch,n_out) supported
- **unit** (*str|tf.contrib.rnn.RNNCell*) – e.g. "BasicLSTM" or "LSTMBlock"
- **initial_state** (*str|float|LayerBase*) – see self.get_rec_initial_state()
- **unit_opts** (*dict[str]|None*) – passed to the cell.__init__

get_dep_layers()

get_hidden_state()

classmethod get_hidden_state_size(*n_out*, *unit*, *unit_opts=None*, ***kwargs*)

get_last_hidden_state()

classmethod get_out_data_from_opts(*n_out*, *name*, *sources=()*, ***kwargs*)

classmethod get_rec_initial_extra_outputs(***kwargs*)

layer_class = 'rnn_cell'

TFSprint

Like SprintErrorSignals.py but for TensorFlow.

TFSprint.**py_get_sprint_automata_for_batch**(*sprint_opts*, *tags*)

Parameters

- **sprint_opts** (*dict[str]*) –
- **tags** (*list[str]*) –

Returns (edges, weights, start_end_states)

Return type (numpy.ndarray, numpy.ndarray, numpy.ndarray)

TFSprint.**get_sprint_automata_for_batch_op**(*sprint_opts*, *tags*)

Parameters

- **sprint_opts** (*dict[str]*) –
- **tags** (*tf.Tensor*) – shape (batch,), of dtype string

Returns (edges, weights, start_end_states). all together in one automaton. edges are of shape (4, num_edges), each (from, to, emission-idx, seq-idx), of dtype int32. weights are of shape (num_edges,), of dtype float32. start_end_states are of shape (2, batch), each (start,stop) state idx, batch = len(tags), of dtype int32.

Return type (tf.Tensor, tf.Tensor, tf.Tensor)

TFSprint.**py_get_sprint_loss_and_error_signal**(*sprint_opts*, *log_posteriors*, *seq_lengths*, *seq_tags*)

Parameters

- **sprint_opts** (*dict[str]*) –
- **log_posteriors** (*numpy.ndarray*) – 3d (time,batch,label)
- **seq_lengths** (*numpy.ndarray*) – 1d (batch)
- **seq_tags** (*list[str]*) – seq names

Returns (loss, error_signal), error_signal has the same shape as posteriors. loss is a 1d-array (batch).

Return type (numpy.ndarray, numpy.ndarray)

TFSprint.**get_sprint_loss_and_error_signal**(*sprint_opts*, *log_posteriors*, *seq_lengths*, *seq_tags*)

Parameters

- **sprint_opts** (*dict[str]*) –
- **log_posteriors** (*tf.Tensor*) – 3d (time,batch,label)
- **seq_lengths** (*tf.Tensor*) – 1d (batch,)
- **seq_tags** (*tf.Tensor*) – 1d (batch,), seq names

Returns (loss, error_signal), error_signal has the same shape as posteriors. loss is a 1d-array (batch).

Return type (tf.Tensor, tf.Tensor)

TFUpdater

class TFUpdater.**Updater**(*config*, *tf_session*, *network*)

This will create the tf.train.Optimizer instance given the config and the update-op for all trainable vars. See the code of *Updater.create_optimizer()* for valid config options.

Note: Vincent Vanhoucke says, in case you get nans, consider increasing the epsilon (for Adam, Nadam and similar). This is the config option optimizer_epsilon. In some places in our Theano code, 1e-16 is our default epsilon, in some other parts, 1e-8 is. 1e-8 might be more stable. Or even 1e-6. Note that when the gradient is suddenly zero in one step, the update can be proportional to lr / eps.

From the tf.train.AdamOptimizer documentation:

> The default value of 1e-8 for epsilon might not be a good default in general. For example, when training an Inception network on ImageNet a current good choice is 1.0 or 0.1. Note that since AdamOptimizer uses the formulation just before Section 2.1 of the Kingma and Ba paper rather than the formulation in Algorithm 1, the "epsilon" referred to here is "epsilon hat" in the paper.

More from Vincent Vanhoucke:

> One thing you can do is run with a tiny learning rate, or even zero learning rate. If you still have divergence then, you have a bug in your setup. If not, increase your rate slowly and see if there is a regime in which things train without diverging. It's completely possible to have weights that are in a good range, but activations or gradients going to infinity because of the shape of the loss, or too high a learning rate. It's obviously always a possibility that there is a bug in the optimizers, but in my experience, every single instance of this kind of problem could be traced back to a weirdly wired model, learning rate issues, bad randomization of the input examples, or - in the case of Adam or RMSProp - issues with the epsilon value.

In addition, you might also want to try gradient_nan_inf_filter or maybe set beta1=0.5.

For further debugging, see tf.add_check_numerics_ops() or add_check_numerics_ops_and_debug_print(), which is config option debug_add_check_numerics_ops.

Parameters

- **config** (Config.Config) –
- **tf_session** (*tf.Session*) –
- **network** (TFNetwork) –

create_optim_op()

create_optimizer()

get_optim_op(*callback_on_new=None*)

Parameters **callback_on_new** (*None | () ->None*) –

Return type tf.Operation

init_optimizer_vars()

reset_optim_op()

Call this if sth is changed which the optim_op depends on. See self.create_optim_op().

set_learning_rate(*value*)

Parameters **value** (float) –

set_trainable_vars(*trainable_vars*)

Parameters **trainable_vars** (list*[tf.Variable]*) –

TFUpdater.**add_check_numerics_ops**(*fetches=None*, *ignore_ops=None*, *use_check_numerics=True*, *debug_print_added_checks=True*, *name='add_check_numerics_ops'*)

This is similar to `tf.add_check_numerics_ops()` and based on similar code. It adds some more logic and options.

Parameters

- **fetches** (list[*tf.Operation|tf.Tensor]|None*) – in case this is given, will only look at these and dependent ops
- **ignore_ops** (list[*str*]) – e.g. “”
- **use_check_numerics** (bool) – if False, instead of `tf.check_numerics()`, it does the check manually (via `tf.is_finite()`) and in case there is inf/nan, it will also print the tensor (while *tf.check_numerics* does not print the tensor). Note that this can be about 50 times slower.
- **debug_print_added_checks** (bool) – prints info about each added check
- **name** (*str*) – op-name for the final tf.group

Returns operation which performs all the checks

Return type tf.Operation

TFUpdater.**get_optimizer_class**(*class_name*)

Parameters **class_name** (*str*) – e.g. “adam”

Returns

TFUtil

class TFUtil.**Condition**(*lock=None*, *name='Condition'*)

A pure TensorFlow implementation of a condition.

init()

signal()

Must be called with the lock held. Emits one signal.

signal_all()

Must be called with the lock held. Emits as many signals as they are waiters.

wait()

Must be called with the lock held, will unlock while waiting for a signal.

wait_counter()

class TFUtil.**CudaEnv**

get_compiler_bin()

get_compiler_opts()

classmethod **get_instance**()

Return type *CudaEnv*

is_available()

verbose_find_cuda = **False**

class TFUtil.**CustomGradient**

Defun(**input_types*, ***kwargs*)

Parameters

- **tf.Tensor) -> tf.Tensor grad_op** (*(tf.Operation,)*) –
- **input_types** (*list[tf.DType]*) –
- **kwargs** (*dict[str]*) – passed to self.register()

Returns function decorator

Return type ((tf.Tensor) -> tf.Tensor) -> ((tf.Tensor) -> tf.Tensor)

register(*input_types*, *op*, *grad_op*, *name=None*)

Parameters

- **input_types** (*list[tf.DType]*) –
- **-> tf.Tensor op** (*(tf.Tensor)*) –
- **tf.Tensor) -> tf.Tensor grad_op** (*(tf.Operation,)* – args are (op, out_grad) and it must return in_grad
- **name** (*str*) – optional func_name

Returns op

Return type (tf.Tensor) -> tf.Tensor

register_loss_and_error_signal(*loss*, *x*, *grad_x*, *name=None*)

Wrapper around self.register(). Expects that loss = loss(x), and grad_x = partial loss / partial x.

Parameters

- **loss** (*tf.Tensor*) –
- **x** (*tf.Tensor*) –
- **grad_x** (*tf.Tensor*) –
- **name** (*str*) – optional func_name

Returns loss but with the gradient for x

Return type tf.Tensor

class TFUtil.**CustomUpdate**

set_on_var(*var*)

Parameters **var** (*tf.Variable*) – variable to update. this will be recognized by *TFUpdater.Updater*

update_var(*var*)

Parameters **var** (*tf.Variable*) – variable to update

Returns operation which updates the variable, e.g. tf.assign_add(var, something)

Return type tf.Operation

class TFUtil.**CustomUpdateExpAverage**(*average*, *alpha*)

exponential moving average

Parameters

- **average** (*tf.Tensor*) –
- **alpha** (*float*) –

update_var(*var*)

class TFUtil.**Data**(*name*, *shape=None*, *dtype=None*, *placeholder=None*, *sparse=None*, *dim=None*, *size_placeholder=None*, *batch_dim_axis=0*, *time_dim_axis=<class 'Util.NotSpecified'>*, *available_for_inference=True*, *auto_create_placeholders=False*, *beam_size=None*)

This class is to describe a tensor, i.e. it's shape and properties like whether we should consider it as sparse data (i.e. it represents indices). This is used in TFNetwork to describe the dataset external data as well as for every layer output.

Parameters

- **name** (*str*) –
- **shape** (*tuple[int|None]|list[int|None]*) – including time-dim (can be None). excluding batch-dim. e.g. (time,feat)=(None,128)
- **dtype** (*str*) – e.g. "float32" or "int64"
- **placeholder** (*tf.Tensor|None*) – with added batch-dim
- **sparse** (*bool*) – whether to treat the value as an index. do not confuse with tf.SparseTensor
- **dim** (*None|int*) – feature dimension, shape[-1] if not sparse, otherwise like num_classes
- **batch_dim_axis** (*int|None*) – where we add the batch-dim. e.g. shape=(time,...), 0 -> (batch,time,...), 1 -> (time,batch,...). This is normally always set, and a lot of code expects this. However, you can set it to None if this Data does not have a batch-dim.
- **time_dim_axis** (*int|None*) – where we have the time dim axis, after we added the batch-dim. this is often 1. however, can be None if there is no time-dim.
- **tf.Tensor size_placeholder** (*dict[int,tf.Tensor]*) – for every None in shape, this will describe the size. The size is always a tensor of shape (batch,), i.e. the size can be different for each sequence in a batch.
- **available_for_inference** (*bool*) – e.g. the extern data "classes" is usually not available for inference
- **beam_size** (*int|None*) – the batch-dim could be extended by a beam-size, such that it represents the merged dims [batch, beam_size].

SpecialAxesNames = ('batch_dim_axis', 'time_dim_axis', 'feature_dim_axis')

batch_ndim

Return type *int*

Returns ndim counted with batch-dim

batch_shape

Returns shape with added batch-dim. e.g. (batch,time,feat) = (None,None,128)

Return type tuple[int|None]

copy()

Returns copy of myself, using self.get_kwargs(), and with placeholder and size_placeholder

Return type *Data*

copy_as_batch_major()

Returns copy of myself with batch_dim_axis == 0

Return type *Data*

copy_as_time_major()

Returns copy of myself with time_dim_axis == 0

Return type *Data*

copy_extend_with_beam(*beam_size*)

Parameters **beam_size** (int) –

Returns copy of myself where the batch-dim is extended/multiplied by beam_size, using tile_transposed

Return type *Data*

copy_template(*name=None*)

Returns copy of myself, using self.get_kwargs(), without placeholder

Return type *Data*

copy_template_adding_time_dim(*name=None*, *time_dim_axis=0*)

Parameters

- **name** (*str|None*) – if set, this will be the new name
- **time_dim_axis** (int) – the new time-dim-axis index

Returns copy of myself adding the time-dimension without placeholder

Return type *Data*

copy_template_excluding_time_dim(*name=None*)

Parameters **name** (*str|None*) – if set, this will be the new name

Returns copy of myself excluding the time-dimension without placeholder

Return type *Data*

copy_time_flattened()

Returns copy of myself where the time-axis is flattened away into the batch-dim-axis. See *get_placeholder_time_flattened()* and :func:‘flatten_with_seq_len_mask for more details.

Return type *Data*

copy_with_batch_dim_axis(*batch_dim_axis*)

Parameters **batch_dim_axis** (int) –

Returns copy of myself with specific batch_dim_axis

Return type *Data*

feature_dim_axis

get_axes(*exclude_time=False*, *exclude_batch=False*)

Parameters

- **exclude_time** (bool) – will filter out the time-axis
- **exclude_batch** (bool) – will filter out the batch-axis

Returns list of axes, like *range(len(self.shape))*, calculated with batch dim.

Return type *list*[*int*]

get_axes_from_description(*axes*)

Parameters **axes** (*int | list[int] | str | list[str]*) – one axis or multiple axis. This is counted with batch-dim, which by default is axis 0 (see enforce_batch_dim_axis). It also accepts the special tokens "B"|"batch", "spatial", "spatial_except_time", or "F"|"feature", and more (see the code).

Returns list of axes, counted with batch-dim

Return type *list*[*int*]

get_axes_with_size()

Returns list of axes which can vary in size for each entry of the batch-dim, e.g. the time-dim-axis. The axis index is counted without the batch-dim.

Return type *list*[*int*]

get_axis_from_description(*axis*)

Parameters **axis** (*int | str*) –

Returns axis, counted with batch-dim

Return type *int*

get_batch_axis(*axis*)

Parameters **axis** (int) – counted without batch-dim

Returns axis counted with batch-dim

Return type *int*

get_batch_axis_excluding_batch(*axis*)

Parameters **axis** (int) – counted with batch-dim

Returns axis counted without batch-dim

Return type *int*

get_bc_spatial_batch_shape()

Returns shape which will broadcast along all spatial dimensions and time/batch dim

Return type tuple[*int*]

get_description(*with_name=True*, *with_placeholder=False*)

get_feature_axes()

Return type *list*[*int*]

Returns list of axes which are feature axes, counted without batch-dim.

get_feature_batch_axes()

Return type *list*[*int*]

Returns list of axes which are feature axes, counted with batch-dim. currently there is only one or zero such axis.

get_kwargs()

get_placeholder_as_batch_major()

get_placeholder_as_time_major()

get_placeholder_flattened(*keep_dims=False*)

Parameters **keep_dims** (bool) – if set, it will add broadcast dimensions after the flattening behind the first axis

Return type tf.Tensor

Returns placeholder where all dynamic axes are flattened into a single axis. e.g. for the usual case (batch, time, dim), it becomes (batch'|time', dim), or (batch, time, height, dim) will also become (batch'|time', dim). with keep_dims, (batch, time, height, dim) will become (batch'|time', 1, 1, dim).

get_placeholder_kwargs(*with_batch=True*)

get_placeholder_time_flattened()

get_placeholder_with_specific_batch_dim_axis(*batch_dim_axis*)

get_sequence_lengths()

Returns seq lens tensor of shape (batch,) of dtype int32

Return type tf.Tensor

get_size_placeholder_kwargs(*axis*, *with_batch=True*)

get_spatial_axes()

Return type *list*[*int*]

Returns list of axes which are not feature and batch axes, counted without batch-dim.

get_spatial_batch_axes()

Return type *list*[*int*]

Returns list of axes which are not feature and batch axes, counted with batch-dim.

get_special_axes_dict(*counted_with_batch_dim=True*, *include_batch_dim_axis=False*, *only_available=False*)

Parameters

- **counted_with_batch_dim** (bool) –
- **include_batch_dim_axis** (bool) –
- **only_available** (bool) –

Returns dict axis-name -> axis

Return type dict[str,*int*]

have_batch_axis()

have_time_axis()

is_batch_major

Returns whether this is in batch-major format, i.e. (batch,...)

Return type *bool*

is_time_major

Returns whether this is in time-major format, i.e. (time,batch,...)

Return type *bool*

matches_var_dim_pattern(*other*)

Parameters **other** (Data) –

Returns whether the variable-dims pattern matches, i.e. same variable dims (get_variable_dim_pattern), same time dim, excluding batch-dim. i.e. the size_placeholder should be compatible.

Return type *bool*

ndim

Return type *int*

Returns ndim counted without batch-dim

ndim_dense

Return type *int*

Returns ndim counted without batch-dim, added by 1 if we are sparse

shape_dense

size_dtype = 'int32'

time_dim_axis_excluding_batch

time_dimension()

Returns shape(placeholder)[time_dim_axis], int scalar

Return type tf.Tensor

class TFUtil.**ExplicitRandomShuffleQueue**(*capacity*, *min_after_dequeue=0*, *dtypes=None*, *shapes=None*, *names=None*, *seed=None*, *shared_name=None*, *name='explicit_random_shuffle_queue'*)

This is intended to behave very much like tf.RandomShuffleQueue, except that it's implemented by other TF native ops / data structures, and you can change min_after_dequeue at runtime. This means that if you have your own logic about when to end, you can set min_after_dequeue=0 and dequeue all the remaining entries from the queue, and then later increase min_after_dequeue again. You can also start with a small min_after_dequeue and increase the number steadily. The original tf.RandomShuffleQueue had the effect of a reset min_after_dequeue=0 after you closed the queue. However, there was no way to reopen the queue. That is the whole reason this implementation exists.

One difference of this implementation is that you must call the init() op once before usage.

One way to implement this is in pure TF. We need some TF container type which supports having entries of different shapes (where the shape can differ where-ever we specified None). We also need some TF container which we can access by index. tf.TensorArray can handle that.

Another way to implement this is by multiple stateful tf.py_func which all reference this instance.

Parameters

- **capacity** (int) –
- **min_after_dequeue** (*int|tf.Tensor*) –
- **dtypes** (list[*str|tf.DType*]) –
- **shapes** (list[*tuple[int|tf.Tensor|None]*]) –

- **names** (list[str]|None) –
- **seed** (int) –
- **shared_name** (str|None) –
- **name** (str) –

dequeue()

enqueue(*v*)

Parameters **v** (list[tf.Tensor]|dict[str,tf.Tensor]|tf.Tensor) –

Return type tf.Operation

init()

Return type tf.Operation

min_after_dequeue_assign(*min_after_dequeue*)

Parameters **min_after_dequeue** (tf.Tensor) –

Return type tf.Operation

min_after_dequeue_read()

size()

Return type tf.Tensor

class TFUtil.**FlipGradientBuilder**
Gradient Reversal Layer. Discussion:

https://github.com/fchollet/keras/issues/3119 https://github.com/tensorflow/tensorflow/issues/4342

Code from here: https://github.com/pumpikano/tf-dann/blob/master/flip_gradient.py

class TFUtil.**GlobalTensorArrayOpMaker**
Creates a TensorArray which does not use the per-run ("per-step") resource manager container but uses the standard container which persists across runs. This TensorArray resource handle is then just a standard TensorArray resource handle which can be used with all TensorArray related functions/ops.

Note: This whole implementation currently does not work because tensor_array.h is not available. See https://github.com/tensorflow/tensorflow/issues/10527 and test_GlobalTensorArray().

An alternative to this might be the MapStagingArea (https://github.com/tensorflow/tensorflow/pull/9686), which should get into TF 1.2.2.

code = '\n #include "tensorflow/core/framework/op_kernel.h"\n #include "tensorflow/core/framework/register_types.h"

get_op()

class TFUtil.**Lock**(*name='Lock'*)
A pure TensorFlow implementation of a mutex / lock.

init()

lock()
On first call, just returns. Any further call will block, unless there is an unlock() call.

unlock()
Must be called after lock().

class TFUtil.**OpCodeCompiler**(*base_name*, *code_version*, *code*, *c_macro_defines=None*, *ld_flags=None*, *include_deps=None*, *static_version_name=None*, *should_cleanup_old_all=True*, *should_cleanup_old_mydir=False*, *use_cuda_if_available=True*, *verbose=False*)

Helper class to compile TF ops on-the-fly, similar as Theano. https://www.tensorflow.org/versions/master/how_tos/adding_an_op/

Parameters

- **base_name** (*str*) – base name for the module, e.g. "zero_out"
- **code_version** (*int|tuple[int]*) – check for the cache whether to reuse
- **code** (*str*) – the source code itself
- **c_macro_defines** (*dict[str,str|int]|None*) – e.g. {"TENSORFLOW": 1}
- **ld_flags** (list*[str]|None*) – e.g. ["-lblas"]
- **include_deps** (list*[str]|None*) – if provided and an existing lib file, we will check if any dependency is newer and we need to recompile. we could also do it automatically via -MD but that seems overkill and too slow.
- **static_version_name** (*str|None*) – normally, we use .../base_name/hash as the dir but this would use .../base_name/static_version_name.
- **should_cleanup_old_all** (bool) – whether we should look in the cache dir and check all ops if we can delete some old ones which are older than some limit (self._cleanup_time_limit_days)
- **should_cleanup_old_mydir** (bool) – whether we should delete our op dir before we compile there.
- **verbose** (bool) – be slightly more verbose

load_module()

class TFUtil.**OutputWithActivation**(*x*, *act_func=None*)

Parameters

- **x** (*tf.Tensor*) –
- **act_func** (*None|(tf.Tensor)->tf.Tensor*) –

get_logits()

Return type tf.Tensor

Returns logits. logits are (not necessarily normalized) log probabilities, i.e. the input of softmax.

This call assumes that self.y is in probability space.

is_softmax_act_func()

class TFUtil.**TFArrayContainer**(*dtype*, *handle=None*, *container=None*, *shared_name=None*, *name='array_container'*)

Array container, like std::vector, with random index access.

Currently does not work. See https://github.com/tensorflow/tensorflow/issues/10950, and test_TFArrayContainer(). Bug #10950 is fixed upstream, should be in TF 1.2.2.

An alternative to this could be *GlobalTensorArrayOpMaker* and MapStagingArea, which should get into TF 1.2.2.

Parameters

- **dtype** (*tf.DType*) –
- **container** (*str*) –
- **shared_name** (*str*) –
- **name** (*str*) –
- **handle** (*tf.resource*) – existing handle to reuse. otherwise we will create a new one

code = '\n #include <vector>\n\n // For Eigen::ThreadPoolDevice.\n #define EIGEN_USE_THREADS 1\n\n #include "ter

get(*index*)

Parameters **index** (*tf.Tensor*) – >= 0 and < size

Returns tensor at that index

Return type tf.Tensor

get_size()

Returns size int32 scalar

Return type tf.Tensor

set(*index*, *value*)

Parameters

- **index** (*tf.Tensor*) – >= 0 and < size
- **value** (*tf.Tensor*) –

Returns operation

Return type tf.Operation

set_size(*size*)

Parameters **size** (*tf.Tensor*) –

Returns operation

Return type tf.Operation

class TFUtil.**VariableAssigner**(*var*)

Parameters **var** (*tf.Variable*) –

assign(*value*, *session*)

Parameters

- **value** (*numpy.ndarray|int|float*) –
- **session** (*tf.Session*) –

TFUtil.**add_scaled_noise_to_gradients**(*grads_and_vars*, *gradient_noise_scale*)

Adds scaled noise from a 0-mean normal distribution to gradients. Adapted from tf.contrib.layers.optimizers.

Parameters

- **tf.Variable)] grads_and_vars** (*list[(tf.Tensor,*) –
- **gradient_noise_scale** (*float*) – used as stddev for tf.truncated_normal().

Returns adapted grads_and_vars

Return type *list*[(tf.Tensor, tf.Variable)]

TFUtil.**assert_min_tf_version**(*version*, *reason*)

Parameters

- **version** (*tuple[int]*) – e.g. (1,2,0) or (1,2)
- **reason** (*str*) –

TFUtil.**auto_init_var**(*v*)

Parameters **v** (*tf.Variable*) –

Returns a reference to the var via tf.identity

Return type tf.Tensor

TFUtil.**batched_uniq**(*x*, *seq_lens*)

Parameters

- **x** (*tf.Tensor*) – shape (batch,time) -> index, some int type
- **seq_lens** (*tf.Tensor|None*) – shape (batch,) of int32|int64

Returns tuple (z, new_seq_lens), where z is of shape (batch, max_new_time), max_new_time = max(new_seq_lens), seq_lens is of shape (batch,).

Return type (tf.Tensor, tf.Tensor)

TFUtil.**check_dim_equal**(*x*, *x_axis*, *y*, *y_axis*)

Parameters

- **x** (*tf.Tensor*) –
- **x_axis** (*int*) – which axis to check
- **y** (*tf.Tensor*) –
- **y_axis** (*int*) – which axis to check

Returns x with check added that shape(x)[x_axis] == shape(y)[y_axis]

Return type tf.Tensor

TFUtil.**check_initial_tf_thread_pool_init**()

TFUtil.**check_input_dim**(*x*, *axis*, *dim*)

Parameters

- **x** (*tf.Tensor*) –
- **axis** (*int*) – which axis to check
- **dim** (*int|tf.Tensor*) –

Returns x with check added

Return type tf.Tensor

TFUtil.**check_input_ndim**(*x*, *ndim*)

Parameters

- **x** (*tf.Tensor*) –
- **ndim** (*int*) –

Returns x with check added

Return type tf.Tensor

TFUtil.**check_input_ndim_equal_offset**(*x*, *y*, *y_ndim_offset=0*)

Parameters

- **x** (*tf.Tensor*) –
- **y** (*tf.Tensor*) –
- **y_ndim_offset** (int) –

Returns x with check added such that ndim(x) == ndim(y) + y_ndim_offset

Return type tf.Tensor

TFUtil.**check_shape_equal**(*x*, *y*)

Parameters

- **x** (*tf.Tensor*) –
- **y** (*tf.Tensor*) –

Returns x with check added that shape(x) == shape(y)

Return type tf.Tensor

TFUtil.**circular_pad**(*x*, *paddings*, *axes=None*)

Parameters

- **x** (*tf.Tensor*) – shape (..., height, width)
- **(int,int))|tf.Tensor paddings** (*int|((int,int),)*) – how much to add ((top,bottom),(left,right))

Returns tensor with shape (..., top + height + bottom, left + width + right)

Return type tf.Tensor

TFUtil.**cond**(*pred*, *fn1*, *fn2*, *name=None*)

This is a wrapper around tf.control_flow_ops.cond(). This will be a branched execution, i.e. either fn1() or fn2() will be executed, or at least the resulting graph will be evaluated. If pred can is constant at the call, only the corresponding fn will be called. This is similar as the TF internal _smart_cond().

Parameters

- **pred** (*tf.Tensor|bool*) –
- **fn1** (*()->(tf.Tensor|list[tf.Tensor])*) –
- **fn2** (*()->(tf.Tensor|list[tf.Tensor])*) –
- **name** (*str*) –

Returns fn1() if pred else fn2()

Return type tf.Tensor|list[tf.Tensor]

TFUtil.**constant_with_shape**(*x*, *shape*, *dtype=None*, *name='constant_with_shape'*)

Parameters

- **x** (*tf.Tensor|float|int|bool*) – scalar
- **shape** (list[*tf.Tensor|int*]|tuple[*tf.Tensor|int*]|*tf.Tensor*) –
- **dtype** (*tf.DType*) –

- **name** (*str*) –

Returns x of the specified shape

Return type tf.Tensor

TFUtil.**debugRegisterBetterRepr**()
Some types don't have good __repr__ implementations by default (for the current TF version). For debugging, it can be helpful to give some more info. This monkey-patches clazz.__repr__ of some TF classes if they are object.__repr__.

TFUtil.**dimshuffle**(*x*, *axes*, *name='dimshuffle'*)
Like Theanos dimshuffle. Combines tf.transpose, tf.expand_dims and tf.squeeze.

Parameters

- **x** (*tf.Tensor*) –
- **axes** (list[int|str]|tuple[int|str]) –
- **name** (*str*) – scope name

Return type tf.Tensor

TFUtil.**directed**(*x*, *direction*)
If direction == 1 or direction is None, returns just x. If direction == -1, returns reversed(x).

Parameters

- **x** (*tf.Tensor*) –
- **direction** (*int|None*) – -1 or 1 (or None)

Return type tf.Tensor

TFUtil.**dot**(*a*, *b*)

Parameters

- **a** (*tf.Tensor*) – shape [...da...,d]
- **b** (*tf.Tensor*) – shape [d,...db...]

Returns tensor of shape [...da...,d,...db...]

Return type tf.Tensor

TFUtil.**dropout**(*x*, *keep_prob*, *noise_shape=None*, *seed=None*, *name=None*)
Computes dropout. Like `tf.nn.dropout()` but avoid `tf.div()` if possible.

Parameters

- **x** (*tf.Tensor*) –
- **keep_prop** (*float|tf.Tensor*) –
- **noise_shape** (*tf.Tensor|tuple[int]*) –
- **seed** (int) –
- **name** (*str*) –

TFUtil.**encode_raw**(*x*, *axis=-1*, *seq_lens=None*)
The inverse function of tf.decode_raw(). Also see: https://stackoverflow.com/questions/43403147/how-to-create-a-encode-raw-tensorflow-function

Parameters

- **x** (*tf.Tensor*) – of integer types [0,255], will get casted to uint8

- **axis** (*int*) – the axis to reduce-join the string. decode_raw has added it at the end
- **seq_lens** (*tf.Tensor|None*) – must have same shape as x after reduce-joining. Note that using seq_lens will make our output not compatible with tf.decode_raw() anymore because tf.decode_raw() requires all strings to be of the same length.

Returns string tensor

Return type tf.Tensor

TFUtil.**enforce_copy**(*x*)

Parameters **x** (*tf.Tensor|tf.Variable*) –

Returns copy of input, i.e. enforces that this is not a ref

TFUtil.**expand_dims_unbroadcast**(*x*, *axis*, *dim*, *name='expand_dims_unbroadcast'*)

Parameters

- **x** (*tf.Tensor*) –
- **axis** (*int|tf.Tensor*) – new axis
- **dim** (*int|tf.Tensor*) – dimension for axis
- **name** (*str*) – scope name

Returns if x is of shape (a,b,c) and axis=0, then we return (dim,a,b,c)

Return type tf.Tensor

TFUtil.**expand_multiple_dims**(*x*, *axes*, *name='expand_multiple_dims'*)

Parameters

- **x** (*tf.Tensor*) –
- **axes** (*list[int]|tuple[int]*) – after completion, tf.shape(y)[axis] == 1 for axis in axes
- **name** (*str*) – scope name

Returns y where we have a new broadcast axis for each axis in axes

Return type tf.Tensor

TFUtil.**filter_grad**(*x*, *threshold*, *axis*)

Parameters

- **x** (*tf.Tensor*) –
- **threshold** (*float*) – all grads going through *x* which max(grad**2) is over the threshold are removed
- **axis** (*int|list[int]*) – max(grad**2) will be reduced over this axis

Returns identity(x) with custom gradient

Return type tf.Tensor

TFUtil.**flatten_with_seq_len_mask**(*x*, *seq_lens*, *time_major=False*)

Parameters

- **x** (*tf.Tensor*) – shape (batch,time,...s...) with time_major=False or otherwise shape (time,batch,...s....)
- **seq_lens** (*tf.Tensor*) – shape (batch,) of int32

- **time_major** (bool) – if the time-dim is the first dimension in x

Returns tensor of shape (time', ...s...) where time' = sum(seq_len) <= batch*time

Return type tf.Tensor

TFUtil.**get_activation_function**(*s*)

Parameters **s** (*str|None*) –

Return type (tf.Tensor) -> tf.Tensor

TFUtil.**get_base_name**(*x*)

Parameters **x** (*tf.Tensor*) – has name e.g. "layer0/rec/W:0"

Returns return the base name, e.g. "W", without the output index

TFUtil.**get_current_name_scope**()

Returns current absolute name scope, via tf.name_scope

Return type str

http://stackoverflow.com/questions/40907769/how-to-get-current-tensorflow-name-scope

Note that this is a private member and might break at some point. Note also that this does not need to be the same as get_current_var_scope_name().

TFUtil.**get_current_var_scope_name**()

Returns current absolute variable scope name, via tf.variable_scope

Return type str

TFUtil.**get_global_train_flag_placeholder**()

Returns bool scalar tensor

Return type tf.Tensor

TFUtil.**get_initializer**(*s*, *seed=None*, *eval_local_ns=None*)

Parameters

- **s** (*str|dict[str]|float*) – e.g. "glorot_uniform" or "truncated_normal" or "orthogonal", or config dict with "class", or string to be 'eval'ed if it contains "(". constant if a float is given.
- **seed** (*int|tf.Tensor*) –
- **eval_local_ns** (*dict[str]|None*) –

Returns (function (shape) -> tf.Tensor) | tf.Initializer

Return type ((tuple[*int*]) -> tf.Tensor) | tf.Initializer

TFUtil.**get_name_scope_of_tensor**(*x*)

Parameters **x** (*tf.Tensor*) – has name e.g. "layer0/rec/W:0"

Returns the name scope of x, e.g. "layer0/rec"

Return type str

TFUtil.**get_ndim**(*x*)

Parameters **x** (*tf.Tensor*) –

Returns x.ndim either as a static int or otherwise as an expression

Return type int|tf.Tensor

TFUtil.**get_range**(*start*, *stop=<class 'Util.NotSpecified'>*)

Parameters

- **start** (*int|tf.Tensor|None*) –
- **stop** (*int|tf.Tensor|None*) –

Returns either tuple(range(start, stop)) or the same as a symbolic expression

Return type tuple[*int*]|tf.Tensor

TFUtil.**get_shape**(*x*)

Parameters **x** (*tf.Tensor*) –

Returns list of scalars, which are either int if known statically, or otherwise expressions

Return type *list*[int|tf.Tensor]

TFUtil.**get_shape_dim**(*x*, *axis*, *name='shape_dim'*)

Parameters

- **x** (*tf.Tensor*) –
- **axis** (*int*) – which axis
- **name** (*str*) –

Returns x.shape[axis] either as a static int or otherwise as an expression

Return type int|tf.Tensor

TFUtil.**global_queue**(*name*, *queue_type*, *capacity*, *dtypes*, *shapes=None*, *names=None*)

Parameters

- **queue_type** (*(args)->tf.QueueBase*) – some function which creates a queue
- **name** (*str*) – global name
- **dtypes** (*list[tf.DType|str]*) –
- **shapes** (*list[tf.TensorShape|tuple[int|None]]|None*) –
- **names** (*list[str]|None*) –

Return type tf.QueueBase

TFUtil.**global_tensor**(*f*, *name*)

This creates a global accessible tensor in the graph to be reused later, i.e. on the second call given a unique name, it will not create a new tensor but return the previously created tensor. This is for the current graph, i.e. if there is a new graph, it will recreate the tensor.

Parameters

- **-> tf.Tensor f** (*()*) – callable which creates the tensor
- **name** (*str*) – global reference name for the tensor

Returns the tensor

Return type tf.Tensor

TFUtil.**identity**(*x*)

Parameters **x** (*tf.Tensor*) –

Return type tf.Tensor

TFUtil.**identity_op_nested**(*x*, *name='identity'*)

Parameters

- **x** (*tf.Tensor|list[tf.Tensor]|dict[str,tf.Tensor]*) –
- **name** (*str*) –

:rtype tf.Tensor|list[tf.Tensor]|dict[str,tf.Tensor]

TFUtil.**identity_with_ops**(*x*, *ops*)

Parameters

- **x** (*tf.Tensor*) –
- **-> list[tf.Operation|tf.Tensor] ops** (*()*) –

Returns x with all ops executed

Return type tf.Tensor

TFUtil.**init_variable_if_needed**(*v*)

Parameters **v** (*tf.Variable*) –

Return type tf.Operation

TFUtil.**is_gpu_available**()
Returns whether TensorFlow can access a GPU.

TFUtil.**make_var_tuple**(*v*)

Parameters **v** (*tf.Tensor|list[tf.Tensor]|tuple[tf.Tensor]*) –

Returns tuple of tensors

Return type tuple[tf.Tensor]

TFUtil.**move_axis**(*x*, *old_axis*, *new_axis*)

Parameters

- **x** (*tf.Tensor*) –
- **old_axis** (*int*) –
- **new_axis** (*int*) –

TFUtil.**nan_to_num**(*x*, *nan_num=0*, *inf_num=1e+30*)
Like numpy.nan_to_num().

Parameters

- **x** (*tf.Tensor*) –
- **nan_num** (*float|tf.Tensor*) –
- **inf_num** (*float|tf.Tensor*) –

Returns x with replaced nan and inf

TFUtil.**nd_indices**(*indices*, *batch_axis=0*)

Parameters **indices** (*tf.Tensor*) – e.g. (batch, ...) -> index

Returns extended indices with batch-idx which can be used for tf.gather_nd, i.e. in the example of shape (batch, ..., 2) where the 2-tuple represents (batch_idx, index).

Return type tf.Tensor

TFUtil.**optional_add**(**args*)

Parameters **args** (list[*tf.Tensor|None]|tf.Tensor*) –

Return type tf.Tensor|None

Returns sums all non-None values, or returns None if there are none

TFUtil.**pad_zeros_in_axis**(*x*, *before=0*, *after=0*, *axis=0*)

Parameters

- **x** (*tf.Tensor*) –
- **before** (*int|tf.Tensor*) –
- **after** (*int|tf.Tensor*) –
- **axis** (int) –

Returns

TFUtil.**post_control_dependencies**(*x*, *updates*)

Parameters

- **x** (*tf.Tensor|list[tf.Tensor]|dict[str,tf.Tensor]*) –
- **updates** (list*[tf.Operation]*) –

Returns identity(x) with control_dependencies(updates)

Return type tf.Tensor|list[tf.Tensor]|dict[str,tf.Tensor]

TFUtil.**print_available_devices**()

TFUtil.**raise_OutOfRangeError**()

Returns an op which raises an OutOfRangeError

Return type tf.Operation

TFUtil.**random_uniform_abs_initializer**(*limit*, ***kwargs*)

TFUtil.**reuse_name_scope**(**args*, ***kwds*)

Parameters

- **name** (*str|tf.VariableScope*) – relative name scope (absolute if absolute=True or if tf.VariableScope)
- **absolute** (bool) – if True it will be absolute

We try to both set the variable scope and the name scope.

TFUtil.**reuse_name_scope_of_tensor**(**args*, ***kwds*)

Parameters

- **x** (*tf.Tensor*) – has name e.g. “layer0/rec/W:0”
- **prefix** (*str*) –
- **postfix** (*str*) –

Returns reuse the name scope of x, e.g. “layer0/rec”, yields scope

TFUtil.**reversed**(*x*)

Just returns x[::-1]. It will cache the value inside the passed object so that we don’t recompute it multiple times.

Parameters **x** (*tf.Tensor*) –

Return type tf.Tensor

TFUtil.**sequence_mask**(*lengths*, ***kwargs*)
Wraps around tf.sequence_mask(). It will cache the value inside the passed object so that we don't recompute it multiple times.

Parameters

- **lengths** (*tf.Tensor*) – shape (batch,)
- **kwargs** (*dict[str]*) – passed on to tf.sequence_mask

Returns tensor mask of shape (batch,maxlen/time). default dtype is bool unless you specify something else

Return type tf.Tensor

TFUtil.**sequence_mask_time_major**(*lengths*, ***kwargs*)
Wraps around tf.transpose(tf.sequence_mask(), (1,0)). It will cache the value inside the passed object so that we don't recompute it multiple times.

Parameters

- **lengths** (*tf.Tensor*) – shape (batch,)
- **kwargs** (*dict[str]*) – passed on to tf.sequence_mask

Returns mask of shape (maxlen/time,batch)

TFUtil.**sequential_control_dependencies**(**args*, ***kwds*)
tf.control_dependencies but each operation will be created such that it is executed after the ones coming before in the list, i.e. l[0] is executed first, l[-1] is executed last.

Parameters **l** (*list[() -> (tf.Operation|tf.Tensor)]*) –

TFUtil.**setup_tf_thread_pools**(*num_threads=None*, *log_file=None*)
See here for documentation of intra_op_parallelism_threads and inter_op_parallelism_threads: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto

intra_op_parallelism_threads is used for the LocalDevice::EigenThreadPoolInfo, which is always global. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/common_runtime/local_device.cc

inter_op_parallelism_threads is used for the (global if not use_per_session_threads) session thread pool. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/common_runtime/direct_session.cc

TF will setup the thread pools on first usage. That can happen quite early, esp for intra_op_parallelism_threads. E.g. list_local_devices() will trigger this, i.e. any call to is_gpu_available() or print_available_devices(). For debugging, you can set the env-var TF_CPP_MIN_VLOG_LEVEL=1 and then check for these message:

```
Local device intra op parallelism threads: 4
Direct session inter op parallelism threads: 4
```

Thus, call this function as early as possible with your preferred number of threads, used for both thread pools. It will create a dummy session and directly close it again, but if you use the global thread pools, those settings will remain for further sessions. This function will only execute on the first call.

Parameters

- **num_threads** (*int*) – used for both intra and inter parallelism thread pools
- **log_file** (*stream|None*) –

TFUtil.**single_strided_slice**(*x*, *axis*, *begin=None*, *end=None*, *step=None*)

Parameters

- **x** (*tf.Tensor*) –
- **axis** (*int|tf.Tensor*) –
- **begin** (*int|tf.Tensor|None*) –
- **end** (*int|tf.Tensor|None*) –
- **step** (*int|tf.Tensor|None*) –

Returns e.g. if axis == 0, returns x[begin:end:step], if axis == 1, returns x[:, begin:end:step], etc.

Return type tf.Tensor

TFUtil.**slice_pad_zeros**(*x*, *begin*, *end*, *axis=0*)

Parameters

- **x** (*tf.Tensor*) – of shape (..., time, ...)
- **begin** (*int|tf.Tensor*) –
- **end** (*int|tf.Tensor*) –
- **axis** (*int*) –

Returns basically x[begin:end] (with axis==0) but if begin < 0 or end > x.shape[0], it will not discard these frames but pad zeros, such that the resulting shape[0] == end - begin.

Return type tf.Tensor

TFUtil.**sparse_labels**(*x*, *seq_lens*, *dtype=tf.int32*, *collapse_repeated=False*)

Parameters

- **x** (*tf.Tensor*) – shape (batch,time) -> index, some int type
- **seq_lens** (*tf.Tensor|None*) – shape (batch,) of int32|int64
- **dtype** (*tf.DType|None*) – if given, will cast the *x* values to this type. ctc_loss() wants int32
- **collapse_repeated** (*bool*) – like uniq() behavior

Returns SparseTensor, e.g. input for tf.nn.ctc_loss()

Return type tf.SparseTensor

TFUtil.**sparse_labels_with_seq_lens**(*x*, *seq_lens*, *dtype=tf.int32*, *collapse_repeated=False*)

Parameters

- **x** (*tf.Tensor*) – shape (batch,time) -> index, some int type
- **seq_lens** (*tf.Tensor|None*) – shape (batch,) of int32|int64
- **dtype** (*tf.DType|None*) – if given, will cast the *x* values to this type. ctc_loss() wants int32
- **collapse_repeated** (*bool*) – like uniq() behavior

Returns SparseTensor, e.g. input for tf.nn.ctc_loss(), and seq_lens of shape (batch,)

Return type (tf.SparseTensor, tf.Tensor)

TFUtil.**spatial_smoothing_energy**(*x*, *dim*, *use_circular_conv=True*)

Parameters

- **x** (*tf.Tensor*) – shape (..., dim)
- **dim** (int) – last dimension of x
- **use_circular_conv** (bool) – whether to use circular convolution, via circular_pad

Return type tf.Tensor

Returns energy of shape (...)

Via Achieving Human Parity in Conversational Speech Recognition, Microsoft, 2017. Interpret the last dimension as 2D (w, h) and apply some high-pass filter on it.

TFUtil.**stop_event_writer_thread**(*event_writer*)

There is a bug in TensorFlow (at least 1.1.0) (https://github.com/tensorflow/tensorflow/issues/4820) that the event writer thread is never stopped. This will try to stop it. Only do it if you don't use the event writer anymore.

Parameters **event_writer** (*tensorflow.python.summary.writer.event_file_writer.EventFileWriter*) –

TFUtil.**swapaxes**(*x*, *axis1*, *axis2*)

Parameters

- **x** (*tf.Tensor*) –
- **axis1** (*tf.Tensor|int*) –
- **axis2** (*tf.Tensor|int*) –

Returns tensor with swapped axes, like numpy.swapaxes

Return type tf.Tensor

TFUtil.**tf_version_tuple**()

Returns version tuple, e.g. (1, 1, 0), parsed from tf.__version__

Return type tuple[*int*]

TFUtil.**tile_transposed**(*x*, *axis*, *multiples*)

Example: x with shape (D,), tf.tile(x, [N]) can be reshaped into (N,D), while tile_transposed(x, axis=0, multiples=N) can be reshaped into (D,N).

Parameters

- **x** (*tf.Tensor*) –
- **axis** (int) –
- **multiples** (*int|tf.Tensor*) –

Returns tensor with shape[axis] == x.shape[axis] * multiples

Return type tf.Tensor

TFUtil.**true_once**()

Returns tensor which will be True once and then always False Internally, this creates a non-trainable variable as a helper.

Return type tf.Tensor

TFUtil.**uniq**(*x*)

Parameters **x** (*tf.Tensor*) – 1D shape (time,) -> index, some int type

Returns like numpy.uniq. unlike tf.unique which will never repeat entries.

Example: uniq([0, 0, 1, 1, 0, 0]) == [0, 1, 0], tf.unique([0, 0, 1, 1, 0, 0]) == [0, 1]. For a batched variant, see batched_uniq, or sparse_labels() with option collapse_repeated.

TFUtil.**var_creation_scope**(**args*, ***kwds*)

If you create a variable inside of a while-loop, you might get the following error: InvalidArgumentError: The node 'while/w/Assign' has inputs from different frames. The input 'while/j' is in frame 'while/while/'. The input 'while/w' is in frame ''.

Also see tests/test_TFUtil.py:test_loop_var_creation(). Related TF bugs:

https://github.com/tensorflow/tensorflow/issues/3114 https://github.com/tensorflow/tensorflow/issues/4478 https://github.com/tensorflow/tensorflow/issues/8604

The solution is to reset the current frame. Resetting all control dependencies has this effect.

TFUtil.**variable_scalar_summaries_dict**(*x*, *name=None*)

Collects all interesting information about *x*, such as min/max/mean, etc. (all scalars). This is used by *variable_summaries()*.

Parameters

- **x** (*tf.Tensor|tf.Variable*) –
- **name** (*str*) –

Returns dicth with key -> scalar info, e.g. with "%s_mean" % name -> tf.reduce_mean(x)

Return type dict[str,tf.Tensor]

TFUtil.**variable_summaries**(*var*, *name=None*, *with_histogram=False*)

Attach a lot of summaries to a Tensor (for TensorBoard visualization). Also see *variable_scalar_summaries_dict()*.

Parameters

- **var** (*tf.Tensor|tf.Variable*) –
- **name** (*str*) –
- **with_histogram** (bool) – adds histogram. note that this can add noticeable overhead

Returns nothing, use `tf.summary.merge_all()` to collect the summaries

TFUtil.**view_as**(*x*, *dtype*)

Does the numpy.view equivalent. Note that the current implementation is inefficient (uses tf.py_func) and CPU-only. :param tf.Tensor x: :param tf.DType dtype: :return: x.view(dtype) equivalent (see numpy.view)

TFUtil.**windowed_nd**(*source*, *window*, *padding='same'*, *time_axis=1*, *new_window_axis=2*)

Parameters

- **source** (*tf.Tensor*) – N-D tensor of shape (..., n_time, ...)
- **window** (*int|tf.Tensor*) – window size
- **padding** (*str*) – "same" or "valid"
- **time_axis** (int) –
- **new_window_axis** (int) –

Returns tensor of shape (..., n_time, ..., window, ...)

Return type tf.Tensor

TFUtil.**xavier_initializer**(*uniform=True*, *seed=None*, *dtype=tf.float32*)

Alias for tf.glorot_uniform_initializer or tf.glorot_normal_initializer.

Parameters

- **uniform** (bool) – uniform or normal distribution
- **seed** (int) –
- **dtype** (*tf.DType*) –

Returns ((tuple[int]) -> tf.Tensor) | tf.Initializer

TaskSystem

Here are all subprocess, threading etc related utilities, most of them quite low level.

TaskSystem.**execInMainProc**(*func*)

TaskSystem.**ExecInMainProcDecorator**(*func*)

exception TaskSystem.**AsyncInterrupt**

exception TaskSystem.**ForwardedKeyboardInterrupt**

TaskSystem.**asyncCall**(*func*, *name=None*, *mustExec=False*)

This executes func() in another process and waits/blocks until it is finished. The returned value is passed back to this process and returned. Exceptions are passed back as well and will be reraised here.

If *mustExec* is set, the other process must *exec()* after the *fork()*. If it is not set, it might omit the *exec()*, depending on the platform.

class TaskSystem.**SharedMem**(*size*, *shmid=None*)

exception ShmException

exception SharedMem.**CCallException**

SharedMem.**libc_so = 'libc.so.6'**

SharedMem.**libc = <CDLL 'libc.so.6', handle 7fd1e9c309b0 at 7fd1d4966610>**

SharedMem.**shm_key_t**

alias of c_int

SharedMem.**IPC_PRIVATE = 0**

SharedMem.**IPC_RMID = 0**

SharedMem.**shmget = <_FuncPtr object>**

SharedMem.**shmat = <_FuncPtr object>**

SharedMem.**shmdt = <_FuncPtr object>**

SharedMem.**shmctl = <_FuncPtr object>**

SharedMem.**memcpy = <_FuncPtr object>**

classmethod SharedMem.**check_ccall_error**(*check*, *f*)

classmethod SharedMem.**is_shmget_functioning**()

SharedMem.**remove**()

SharedMem.**ctypes = <module 'ctypes' from '/usr/lib/python2.7/ctypes/__init__.pyc'>**

TaskSystem.**next_power_of_two**(*n*)

class TaskSystem.**SharedNumpyArray**(*shape*, *strides*, *typestr*, *mem=None*, *array_id=None*)

This class provides a way to create Numpy arrays in shared memory. It adds some logic to mark whether some shared memory segment can be reused - that is when the client marks it as unused.

Note that there are a few similar Python modules: https://pypi.python.org/pypi/SharedArray http://parad0x.org/git/python/shared-array/about https://bitbucket.org/cleemesser/numpy-sharedmem/src http://stackoverflow.com/questions/5033799/how-do-i-pass-large-numpy-arrays http://stackoverflow.com/questions/7894791/use-numpy-array-in-shared-memory

ServerLock = <thread.lock object>

ServerInstances = set([])

ServerArrayId = 0

exception TooMuchInstances

SharedNumpyArray.**ExtraSpaceBytes = 4096**

static SharedNumpyArray.**numpy_strides_for_fortran**(*shape*, *typestr*)

static SharedNumpyArray.**numpy_strides_for_c_contiguous**(*shape*, *typestr*)

classmethod SharedNumpyArray.**needed_mem_size**(*shape*, *typestr*)

classmethod SharedNumpyArray.**as_shared**(*array*)

classmethod SharedNumpyArray.**create_copy**(*array*)

classmethod SharedNumpyArray.**create_new**(*shape*, *strides*, *typestr*)

SharedNumpyArray.**is_server = False**

SharedNumpyArray.**mem = None**

SharedNumpyArray.**get_numpy_array_data_ptr**()

SharedNumpyArray.**create_numpy_array**()

SharedNumpyArray.**is_in_use**()

SharedNumpyArray.**set_unused**()

SharedNumpyArray.**shape = None**

SharedNumpyArray.**strides = None**

SharedNumpyArray.**typestr = None**

TaskSystem.**attrChain**(*base*, **attribs*, ***kwargs*)

TaskSystem.**funcCall**(*attrChainArgs*, *args=()*)

TaskSystem.**get_func_closure**(*f*)

TaskSystem.**get_func_tuple**(*f*)

TaskSystem.**bytes**(*x*, **args*)

TaskSystem.**makeFuncCell**(*value*)

TaskSystem.**getModuleDict**(*modname*, *path=None*)

Parameters

- **modname** (*str*) – such that “import <modname>” would work
- **path** (list *[str]*) – sys.path

Returns the dict of the mod

Return type dict[str]

TaskSystem.**getModNameForModDict**(*obj*)

Return type str | None

:returns The module name or None. It will not return '__main__' in any case because that likely will not be the same in the unpickling environment.

TaskSystem.**getNormalDict**(*d*)

Return type dict[str]

It also removes getset_descriptor. New-style classes have those.

TaskSystem.**make_numpy_ndarray_fromstring**(*s*, *dtype*, *shape*)

TaskSystem.**use_shared_mem_for_numpy_array**(*obj*)

TaskSystem.**numpy_set_unused**(*v*)

Parameters **v** (*numpy.ndarray*) – array which will be marked as not-used-anymore

This will tell mechanisms like SharedNumpyArray that it can reuse the memory. On the client side, this will even unmap the memory, so any further access to it will cause a SEGFAULT.

TaskSystem.**numpy_copy_and_set_unused**(*v*)

Parameters **| numpy.ndarray | object v** (*dict[str,numpy.ndarray|object]*) – object to be handled

If v is a dict, we will return a new copied dict where every value is mapped through numpy_copy_and_set_unused. If v is a numpy.ndarray and its base is a SharedNumpyArray, we will copy it and

call numpy_set_unused on the old value.

If v is a numpy.ndarray and its base is not a SharedNumpyArray, we will just return it as it is and do nothing. In all other cases, we will also just return the object as it is and do nothing.

TaskSystem.**numpy_alloc**(*shape*, *dtype*, *fortran_for_shared=False*)

If EnableAutoNumpySharedMemPickling is True, this will allocate a Numpy array in shared memory so we avoid a copy later on when this Numpy array would be transferred to another process via pickling.

class TaskSystem.**Pickler**(**args*, ***kwargs*)

We extend the standard Pickler to be able to pickle some more types, such as lambdas and functions, code, func cells, buffer and more.

dispatch = {<type '_io.BufferedWriter'>: <function save_iobuffer_dummy>, <type 'float'>: <function save_float>, <typ

save_func(*obj*)

save_method(*obj*)

save_code(*obj*)

save_cell(*obj*)

intellisave_dict(*obj*)

save_module(*obj*)

save_buffer(*obj*)

save_string(*obj*, *pack=<built-in function pack>*)

save_ndarray(*obj*)

save_iobuffer_dummy(*obj*)

save_global(*obj*, *name=None*)

save_type(*obj*)

save_class(*cls*)

class TaskSystem.**ExecingProcess**(*target*, *args*, *name*, *env_update*)
This is a replacement for multiprocessing.Process which always uses fork+exec, not just fork. This ensures that you have a separate independent process. This can avoid many types of bugs, such as:

http://stackoverflow.com/questions/24509650 http://bugs.python.org/issue6721 http://stackoverflow.com/questions/8110920 http://stackoverflow.com/questions/23963997 https://github.com/numpy/numpy/issues/654 http://comments.gmane.org/gmane.comp.python.numeric.general/60204

start()

is_alive()

join(*timeout=None*)

Verbose = False

static **checkExec**()

exception TaskSystem.**ProcConnectionDied**

class TaskSystem.**ExecingProcess_ConnectionWrapper**(*fd=None*, *conn=None*)
Wrapper around multiprocessing.connection.Connection. This is needed to use our own Pickler.

poll(**args*, ***kwargs*)

send_bytes(*value*)

send(*value*)

recv_bytes()

recv()

TaskSystem.**ExecingProcess_Pipe**()
This is like multiprocessing.Pipe(duplex=True). It uses our own ExecingProcess_ConnectionWrapper.

TaskSystem.**Pipe_ConnectionWrapper**(**args*, ***kwargs*)

class TaskSystem.**AsyncTask**(*func*, *name=None*, *mustExec=False*, *env_update=None*)
This uses multiprocessing.Process or ExecingProcess to execute some function. In addition, it provides a duplex pipe for communication. This is either multiprocessing.Pipe or ExecingProcess_Pipe.

Parameters

- **func** – a function which gets a single parameter, which will be a reference to our instance in the fork, so that it can use our communication methods put/get.
- **mustExec** (bool) – if True, we do fork+exec, not just fork
- **env_update** (*dict[str,str]*) – for mustExec, also update these env vars

put(*value*)

get()

isParent

isChild

setCancel()

terminate()

join(*timeout=None*)

is_alive()

TaskSystem.**WarnMustNotBeInForkDecorator**(*func*)

class TaskSystem.**ReadWriteLock**

Classic implementation of ReadWriteLock. Note that this partly supports recursive lock usage: - Inside a readlock, a writelock will always block! - Inside a readlock, another readlock is fine. - Inside a writelock, any other writelock or readlock is fine.

readlock

writelock

TaskSystem_example

TaskSystem_example.**start**()

TaskSystem_example.**print_action**()

TaskSystem_example.**process**(*asyncTask*)

TheanoUtil

TheanoUtil.**time_batch_make_flat**(*val*)

Rtype val theano.Variable

Return type theano.Variable

Will flatten the first two dimensions and leave the others as is.

TheanoUtil.**class_idx_seq_to_1_of_k**(*seq*, *num_classes*, *dtype='float32'*)

Parameters

- **seq** (*theano.Variable*) – ndarray with indices
- **| theano.Variable num_classes** (int) – number of classes
- **dtype** (*str*) – eg "float32"

Return type theano.Variable

:returns ndarray with one added dimension of size num_classes. That is the one-hot-encoding. This function is like theano.tensor.extra_ops.to_one_hot but we can handle multiple dimensions.

TheanoUtil.**tiled_eye**(*n1*, *n2*, *dtype='float32'*)

TheanoUtil.**opt_contiguous_on_gpu**(*x*)

TheanoUtil.**windowed_batch**(*source*, *window*)

Parameters

- **source** (*theano.TensorVariable*) – 3d tensor of shape (n_time, n_batch, n_dim)
- **window** (*int|theano.Variable*) – window size

Returns tensor of shape (n_time, n_batch, window * n_dim)

TheanoUtil.**delta_batch**(*source*, *window*)

Parameters

- **source** (*theano.TensorVariable*) – 3d tensor of shape (n_time, n_batch, n_dim)
- **window** (*int|theano.Variable*) – window size

Returns tensor of shape (n_time, n_batch, window * n_dim)

Similar as numpy.diff. Also called delta. TODO with conv op

TheanoUtil.**context_batched**(*source*, *window*)

same as windowed_batch but with window center at the end of the window :param theano.TensorVariable source: 3d tensor of shape (n_time, n_batch, n_dim) :param int|theano.Variable window: window size :return: tensor of shape (n_time, n_batch, window * n_dim)

TheanoUtil.**window_batch_timewise**(*t*, *b*, *w*, *full_index*)

TheanoUtil.**slice_for_axis**(*axis*, *s*)

TheanoUtil.**downsample**(*source*, *axis*, *factor*, *method='average'*)

TheanoUtil.**upsample**(*source*, *axis*, *factor*, *method='nearest-neighbor'*, *target_axis_len=None*)

TheanoUtil.**pad**(*source*, *axis*, *target_axis_len*, *pad_value=None*)

TheanoUtil.**chunked_time_reverse**(*source*, *chunk_size*)

Parameters

- **source** – >=1d array (time,...)
- **chunk_size** – int

Returns like source

Will not reverse the whole time-dim, but only every time-chunk. E.g. source=[0 1 2 3 4 5 6], chunk_size=3, returns [2 1 0 5 4 3 0]. (Padded with 0, recovers original size.)

TheanoUtil.**try_register_canonicalize**(*f*)

TheanoUtil.**try_register_gpu_opt**(*base_op_class*)

class TheanoUtil.**GradDiscardOutOfBound**(*lower_bound*, *upper_bound*)

grad(*args*, *g_outs*)

TheanoUtil.**grad_discard_out_of_bound**(*x*, *lower_bound*, *upper_bound*)

TheanoUtil.**gaussian_filter_1d**(*x*, *sigma*, *axis*, *window_radius=40*)

Filter 1d input with a Gaussian using mode *nearest*. x is expected to be 2D/3D of type (time,batch,...). Adapted via: https://github.com/Theano/Theano/issues/3793 Original Author: https://github.com/matthias-k

TheanoUtil.**log_sum_exp**(*x*, *axis*)

TheanoUtil.**max_filtered**(*x*, *axis*, *index*)

TheanoUtil.**log_sum_exp_index**(*x*, *axis*, *index*)

TheanoUtil.**global_softmax**(*z*, *index*, *mode*)

Parameters

- **z** (*theano.Variable*) – 3D array. time*batch*feature
- **index** (*theano.Variable*) – 2D array, 0 or 1, time*batch

Return type theano.Variable

:returns 3D array. exp(z) / Z, where Z = sum(exp(z),axis=[0,2]) / z.shape[0].

TheanoUtil.**show_global_softmax_stats**(*z*)

Parameters **z** – numpy.ndarray or Theano Var (eval-able), 2D time*features

TheanoUtil.**complex_elemwise_mult**(*a*, *b*, *axis=-1*)

TheanoUtil.**complex_bound**(*a*, *axis=-1*)

TheanoUtil.**complex_dot**(*a*, *b*)

TheanoUtil.**indices_in_flatten_array**(*ndim*, *shape*, **args*)

We expect that all args can be broadcasted together. So, if we have some array A with ndim&shape as given, A[args] would give us a subtensor. We return the indices so that A[args].flatten() and A.flatten()[indices] are the same.

TheanoUtil.**circular_convolution**(*a*, *b*)

TheanoUtil.**unroll_scan**(*fn*, *sequences=()*, *outputs_info=()*, *non_sequences=()*, *n_steps=None*, *go_backwards=False*)

Helper function to unroll for loops. Can be used to unroll theano.scan. The parameter names are identical to theano.scan, please refer to here for more information.

Note that this function does not support the truncate_gradient setting from theano.scan.

Code adapted from https://github.com/Lasagne/Lasagne. Thank you!

Parameters **fn** : function

Function that defines calculations at each step.

sequences : TensorVariable or list of TensorVariables

List of TensorVariable with sequence data. The function iterates over the first dimension of each TensorVariable.

outputs_info : list of TensorVariables

List of tensors specifying the initial values for each recurrent value.

non_sequences: list of TensorVariables

List of theano.shared variables that are used in the step function.

n_steps: int

Number of steps to unroll.

go_backwards: bool

If true the recursion starts at sequences[-1] and iterates backwards.

Returns Tuple of the form (outputs, updates).

outputs is a list of TensorVariables. Each element in the list gives the recurrent

values at each time step.

updates is an empty dict for now.

TheanoUtil.**make_var_tuple**(*v*)

class TheanoUtil.**Contiguous**

grad(*inputs*, *output_grads*)

class TheanoUtil.**DumpOp**(*filename*, *container=None*, *with_grad=True*, *parent=None*, *step=1*)

view_map = {0: [0]}

make_node(*x*)

perform(*node*, *inputs*, *output_storage*)

grad(*inputs*, *output_grads*)

dump(*x*)

get_full_filename()

get_counter()

inc_counter()

TheanoUtil.**softmax**(*z*)

TheanoUtil.**layer_normalization**(*x*, *bias=None*, *scale=None*, *eps=1e-05*)
Layer Normalization, https://arxiv.org/abs/1607.06450 x is mean and variance normalized along its feature dimension. After that, we allow a bias and a rescale. This is supposed to be trainable. :param x: 3d tensor (time,batch,dim) (or any ndim, last dim is expected to be dim) :param bias: 1d tensor (dim) or None :param scale: 1d tensor (dim) or None

TheanoUtil.**print_to_file**(*filename*, *x*, *argmax=None*, *sum=None*, *shape=False*)

TheanoUtil.**self_similarity_cosine**(*x*)

Parameters **x** – shape (T,D)

:returns cosine similarity matrix, shape (T,T), zeroed at diagonal and upper triangle

TorchWrapper

class TorchWrapper.**TorchWrapperOp**(*in_info*, *out_info*, *lua_fw_func*, *lua_bw_func=None*, *lua_file=None*, *name=None*)

make_node(**args*)

infer_shape(*node*, *input_shapes*)

perform(*node*, *inputs*, *output_storage*)

c_header_dirs()

c_lib_dirs()

c_libraries()

c_compile_args()

c_support_code()

c_support_code_struct(*node*, *name*)

c_init_code_struct(*node*, *name*, *sub*)

c_cleanup_code_struct(*node*, *name*)

c_code(*node*, *name*, *inputs*, *outputs*, *sub*)

grad(*inputs*, *output_grads*)

connection_pattern(*node*)

class TorchWrapper.**GpuTorchWrapperOp**(*in_info*, *out_info*, *lua_fw_func*, *lua_bw_func=None*, *lua_file=None*, *name=None*)

c_support_code()

c_libraries()

TwoStateBestPathDecoder

class TwoStateBestPathDecoder.**TwoStateBestPathDecodeOp**

make_node(*x*, *y*, *seq_lengths*)

c_code(*node*, *name*, *inp*, *out*, *sub*)

c_compile_args()

c_code_cache_version()

c_support_code()

TwoStateHMMOp

class TwoStateHMMOp.**TwoStateHMMOp**

make_node(*x*, *y*, *seq_lengths*, *tdp_loop=TensorConstant{0.0}*, *tdp_fwd=TensorConstant{0.0}*)

c_support_code()

c_compile_args()

c_code(*node*, *name*, *inp*, *out*, *sub*)

c_code_cache_version()

Updater

class Updater.**Updater**(*momentum=0.0*, *nesterov_momentum=0.0*, *momentum2=0.0*, *gradient_clip=-1.0*, *update_clip=-1.0*, *weight_clip=0.0*, *adagrad=False*, *adadelta=False*, *adadelta_decay=0.9*, *adadelta_offset=1e-06*, *max_norm=0.0*, *adasecant=False*, *adam=False*, *adamdelta=False*, *adam_fit_learning_rate=True*, *adamax=False*, *nadam=False*, *nadam_decay=0.004*, *eve=False*, *gradient_l2_norm=False*, *mean_normalized_sgd=False*, *mean_normalized_sgd_average_interpolation=0.5*, *rmsprop=0.0*, *smorms3=False*, *update_multiple_models=0*, *update_multiple_models_average_step=0*, *update_multiple_models_average_step_i=0*, *update_multiple_models_averaging=True*, *update_multiple_models_param_is_cur_model=False*, *multi_batch_update=0*, *variance_reduction=False*, *enforce_triangular_matrix_zero=False*, *gradient_noise=0.0*, *gradient_noise_decay=0.55*, *grad_noise_rel_grad_norm=0.0*, *reset_update_params=False*)

This defines how to update the model parameters per mini-batch. All kind of gradient-based optimization methods are implemented here, such as Adam etc.

Initializes the Updater class. All the params determine the specific optimization variants and their hyper params. Normally this is constructed by *Updater.initFromConfig()*.

Parameters

- **momentum** –
- **nesterov_momentum** –
- **momentum2** –
- **gradient_clip** –
- **update_clip** –
- **adagrad** –
- **adadelta** –
- **adadelta_decay** –
- **adadelta_offset** –
- **max_norm** –
- **adasecant** –
- **adam** –
- **adamdelta** –
- **adam_fit_learning_rate** –
- **adamax** –
- **eve** – Eve optimizer - Adam with a feedback from loss
- **adamvr** – Adam with adasecant variance reduction
- **nadam** – Adam with nag part momentum
- **nadam_decay** –
- **mean_normalized_sgd** –
- **mean_normalized_sgd_average_interpolation** –
- **rmsprop** –
- **smorms3** –
- **update_multiple_models** –
- **update_multiple_models_average_step** –
- **update_multiple_models_average_step_i** –
- **update_multiple_models_averaging** –
- **update_multiple_models_param_is_cur_model** –
- **multi_batch_update** –
- **enforce_triangular_matrix_zero** –
- **gradient_noise** –
- **gradient_noise_decay** –
- **grad_noise_rel_grad_norm** –
- **reset_update_params** –

getUpdateList()

classmethod **initFromConfig**(*config*)
Will construct a *Updater* instance where all params are automatically determined by the given config.

Parameters **config** (Config.Config) –

Return type *Updater*

classmethod **initRule**(*rule*, ***kwargs*)

initVars(*network*, *net_param_deltas*)
Initializes the Theano shared variables. This should be called in the process where you want to do the updating. All further calls must be from the same process. The network.gparams must be created in the same process.

isInitialized

norm_constraint(*tensor_var*, *max_norm*, *norm_axes=None*, *epsilon=1e-12*)

reset()

setLearningRate(*learning_rate*)

setNetParamDeltas(*net_param_deltas*)

update()

var(*value*, *name=''*, *broadcastable=None*, *dtype='float32'*, *zero=False*)

Util

class Util.**NotSpecified**
This is just a placeholder, to be used as default argument to mark that it is not specified.

Util.**is_64bit_platform**()

Returns True if we run on 64bit, False for 32bit

Return type *bool*

http://stackoverflow.com/questions/1405913/how-do-i-determine-if-my-python-shell-is-executing-in-32bit-or-64bit-mode-on-os

class Util.**BackendEngine**

Theano = 0

Default = 0

TensorFlow = 1

selectedEngine = None

classmethod **select_engine**(*engine=None*, *config=None*)

Parameters

- **engine** (int) –
- **config** (Config.Config) –

classmethod **get_selected_engine**()

classmethod **is_theano_selected**()

classmethod **is_tensorflow_selected**()

Util.**cmd**(*s*)

Return type *list*[str]

:returns all stdout splitted by newline. Does not cover stderr. Raises CalledProcessError on error.

Util.**sysexecOut**(**args*, ***kwargs*)

Util.**sysexecRetCode**(**args*, ***kwargs*)

Util.**git_commitRev**(*commit='HEAD'*, *gitdir='.'*)

Util.**git_isDirty**(*gitdir='.'*)

Util.**git_commitDate**(*commit='HEAD'*, *gitdir='.'*)

Util.**git_describeHeadVersion**(*gitdir='.'*)

Util.**describe_crnn_version**()

Util.**describe_theano_version**()

Util.**describe_tensorflow_version**()

Util.**get_tensorflow_version_tuple**()

Returns tuple of ints, first entry is the major version

Return type tuple[*int*]

Util.**eval_shell_env**(*token*)

Util.**eval_shell_str**(*s*)

Return type *list*[str]

Parses *s* as shell like arguments (via shlex.split) and evaluates shell environment variables (eval_shell_env). *s* or its elements can also be callable. In those cases, they will be called and the returned value is used.

Util.**hdf5_dimension**(*filename*, *dimension*)

Util.**hdf5_group**(*filename*, *dimension*)

Util.**hdf5_shape**(*filename*, *dimension*)

Util.**hdf5_strings**(*handle*, *name*, *data*)

Util.**model_epoch_from_filename**(*filename*)

Util.**terminal_size**()

Util.**hms**(*s*)

Parameters **s** (*float | int*) – seconds

Returns e.g. "1:23:45" (hs:ms:secs). see hms_fraction if you want to get fractional seconds

Return type str

Util.**hms_fraction**(*s*, *decimals=4*)

Parameters

- **s** (float) – seconds
- **decimals** (int) – how much decimals to print

Returns e.g. "1:23:45.6789" (hs:ms:secs)

Return type str

Util.**human_size**(*n*, *factor=1000*, *frac=0.8*, *prec=1*)

Util.**progress_bar**(*complete=1.0*, *prefix=''*, *suffix=''*)

Util.**progress_bar_with_time**(*complete=1.0*, *prefix=''*, ***kwargs*)

Util.**availablePhysicalMemoryInBytes**()

Util.**defaultCacheSizeInGBytes**(*factor=0.7*)

Util.**betterRepr**(*o*)
The main difference: this one is deterministic. The orig dict.__repr__ has the order undefined for dict or set. For big dicts/sets/lists, add "," at the end to make textual diffs nicer.

Util.**simpleObjRepr**(*obj*)
All self.__init__ args.

class Util.**ObjAsDict**(*obj*)

items()

class Util.**DictAsObj**(*dikt*)

Util.**dict_joined**(**ds*)

Util.**obj_diff_str**(*self*, *other*)

Util.**dict_diff_str**(*self*, *other*)

Util.**find_ranges**(*l*)

:returns list of ranges (start,end) where end is exclusive such that the union of range(start,end) matches l. :rtype: list[(int,int)] We expect that the incoming list is sorted and strongly monotonic increasing.

Util.**initThreadJoinHack**()

Util.**start_daemon_thread**(*target*, *args=()*)

Util.**is_quitting**()

Util.**interrupt_main**()

Util.**try_run**(*func*, *args=()*, *catch_exc=<type 'exceptions.Exception'>*, *default=None*)

Util.**class_idx_seq_to_1_of_k**(*seq*, *num_classes*)

Util.**uniq**(*seq*)
Like Unix tool uniq. Removes repeated entries. :param seq: numpy.array :return: seq

Util.**slice_pad_zeros**(*x*, *begin*, *end*, *axis=0*)

Parameters

- **x** (*numpy.ndarray*) – of shape (..., time, ...)
- **begin** (int) –
- **end** (int) –
- **axis** (int) –

Returns basically x[begin:end] (with axis==0) but if begin < 0 or end > x.shape[0], it will not discard these frames but pad zeros, such that the resulting shape[0] == end - begin.

Return type numpy.ndarray

Util.**random_orthogonal**(*shape*, *gain=1.0*, *seed=None*)

Returns a random orthogonal matrix of the given shape. Code borrowed and adapted from Keras: https://github.com/fchollet/keras/blob/master/keras/initializers.py Reference: Saxe et al., http://arxiv.org/abs/1312.6120 Related: Unitary Evolution Recurrent Neural Networks, https://arxiv.org/abs/1511.06464

Parameters

- **shape** (*tuple[int]*) –
- **gain** (float) –
- **seed** (int) – for Numpy random generator

Returns random orthogonal matrix

Return type numpy.ndarray

Util.**inplace_increment**(*x*, *idx*, *y*)

This basically does *x[idx] += y*. The difference to the Numpy version is that in case some index is there multiple times, it will only be incremented once (and it is not specified which one). See also theano.tensor.subtensor.AdvancedIncSubtensor documentation.

Util.**parse_orthography_into_symbols**(*orthography*, *upper_case_special=True*, *word_based=False*)

For Speech. Example:

orthography = "hello [HESITATION] there " with word_based == False: returns list("hello ") + ["[HESITATION]"] + list(" there "). with word_based == True: returns ["hello", "[HESITATION]", "there"]

No pre/post-processing such as: Spaces are kept as-is. No stripping at begin/end. (E.g. trailing spaces are not removed.) No tolower/toupper. Doesn't add [BEGIN]/[END] symbols or so. Any such operations should be done explicitly in an additional function. Anything in []-brackets are meant as special-symbols. Also see parse_orthography() which includes some preprocessing.

Parameters

- **orthography** (*str*) – example: "hello [HESITATION] there "
- **upper_case_special** (bool) – whether the special symbols are always made upper case
- **word_based** (bool) – whether we split on space and return full words

Return type *list*[str]

Util.**parse_orthography**(*orthography*, *prefix=()*, *postfix=('[END]',)*, *remove_chars='(){}'*, *collapse_spaces=True*, *final_strip=True*, ***kwargs*)

For Speech. Full processing. Example:

orthography = "hello [HESITATION] there " with word_based == False: returns list("hello ") + ["[HESITATION]"] + list(" there") + ["[END]"] with word_based == True: returns ["hello", "[HESITATION]", "there", "[END]"]

Does some preprocessing on orthography and then passes it on to parse_orthography_into_symbols().

Parameters

- **orthography** (*str*) – e.g. "hello [HESITATION] there "
- **prefix** (list*[str]*) – will add this prefix
- **postfix** (list*[str]*) – will add this postfix

- **remove_chars** (*str*) – those chars will just be removed at the beginning
- **collapse_spaces** (bool) – whether multiple spaces and tabs are collapsed into a single space
- **final_strip** (bool) – whether we strip left and right
- **kwargs** (*dict[str]*) – passed on to parse_orthography_into_symbols()

Return type *list*[str]

Util.**json_remove_comments**(*string*, *strip_space=True*)

Return type str

via https://github.com/getify/JSON.minify/blob/master/minify_json.py, by Gerald Storer, Pradyun S. Gedam, modified by us.

Util.**unicode_to_str_recursive**(*s*)

Util.**load_json**(*filename=None*, *content=None*)

class Util.**NumbersDict**(*auto_convert=None*, *numbers_dict=None*, *broadcast_value=None*)

It's mostly like dict[str,float|int] & some optional broadcast default value. It implements the standard math bin ops in a straight-forward way.

copy()

constant_like(*number*)

keys_set

get(*key*, *default=None*)

pop(*key*, **args*)

keys()

values()

has_values()

unary_op(*op*)

classmethod bin_op_scalar_optional(*self*, *other*, *zero*, *op*)

classmethod bin_op(*self*, *other*, *op*, *zero*, *result=None*)

elem_eq(*other*, *result_with_default=True*)

Element-wise equality check with other. Note about broadcast default value: Consider some key which is neither in self nor in other.

This means that self[key] == self.default, other[key] == other.default. Thus, in case that self.default != other.default, we get res.default == False. Then, all(res.values()) == False, even when all other values are True. This is sometimes not what we want. You can control the behavior via result_with_default.

classmethod max(*items*)

Element-wise maximum for item in items. :param list[NumbersDict|int|float] items: :rtype: NumbersDict

classmethod min(*items*)

Element-wise minimum for item in items. :param list[NumbersDict|int|float] items: :rtype: NumbersDict

max_value()

Maximum of our values.

Util.**collect_class_init_kwargs**(*cls*)

Util.**custom_exec**(*source*, *source_filename*, *user_ns*, *user_global_ns*)

class Util.**FrozenDict**

Util.**make_hashable**(*obj*)

Theano needs hashable objects in some cases, e.g. the properties of Ops. This converts all objects as such, i.e. into immutable frozen types.

Util.**make_dll_name**(*basename*)

Util.**escape_c_str**(*s*)

Util.**attr_chain**(*base*, *attribs*)

Util.**to_bool**(*v*)

Util.**as_str**(*s*)

Util.**load_txt_vector**(*filename*)

Expect line-based text encoding in file. We also support Sprint XML format, which has some additional xml header and footer, which we will just strip away.

class Util.**CollectionReadCheckCovered**(*collection*)

get(*item*, *default=None*)

assert_all_read()

Util.**which**(*program*)

Util.**overwrite_os_exec**(*prefix_args*)

Parameters **prefix_args** (list[str]) –

Util.**get_lsb_release**()

Util.**get_ubuntu_major_version**()

Return type intlNone

Util.**auto_prefix_os_exec_prefix_ubuntu**(*prefix_args*, *ubuntu_min_version=16*)

Parameters

- **prefix_args** (list[str]) –
- **ubuntu_min_version** (int) –

Example usage: auto_prefix_os_exec_prefix_ubuntu([”/u/zeyer/tools/glibc217/ld-linux-x86-64.so.2”])

Util.**cleanup_env_var_path**(*env_var*, *path_prefix*)

Parameters

- **env_var** (str) – e.g. “LD_LIBRARY_PATH”
- **path_prefix** (str) –

Will remove all paths in os.environ[env_var] which are prefixed with path_prefix.

Util.**get_login_username**()

Return type str

Returns the username of the current user.

Use this as a replacement for os.getlogin().

Util.**get_temp_dir**()

Return type str

Returns e.g. "/tmp/$USERNAME"

class Util.**LockFile**(*directory*, *name='lock_file'*, *lock_timeout=3600*)

Parameters

- **directory** (*str*) –
- **lock_timeout** (*int|float*) – in seconds

is_old_lockfile()

maybe_remove_old_lockfile()

is_locked()

lock()

unlock()

Util.**str_is_number**(*s*)

Parameters **s** (*str*) – e.g. "1", ".3" or "x"

Returns whether s can be casted to float or int

Return type *bool*

Util.**sorted_values_from_dict**(*d*)

Util.**dict_zip**(*keys*, *values*)

Util.**parse_ld_conf_file**(*fn*)
Via https://github.com/albertz/system-tools/blob/master/bin/find-lib-in-path.py. :param str fn: e.g. "/etc/ld.so.conf" :return: list of paths for libs :rtype: list[str]

Util.**get_ld_paths**()
To be very correct, see man-page of ld.so. And here: http://unix.stackexchange.com/questions/354295/what-is-the-default-value-of-ld-library-path/354296 Short version, not specific to an executable, in this order: - LD_LIBRARY_PATH - /etc/ld.so.cache (instead we will parse /etc/ld.so.conf) - /lib, /usr/lib (or maybe /lib64, /usr/lib64) Via https://github.com/albertz/system-tools/blob/master/bin/find-lib-in-path.py.

Return type *list*[str]

Returns list of paths to search for libs (*.so files)

Util.**find_lib**(*lib_name*)

Parameters **lib_name** (*str*) – without postfix/prefix, e.g. "cudart" or "blas"

Returns returns full path to lib or None

Return type str|None

Util.**read_sge_num_procs**(*job_id=None*)
From the Sun Grid Engine (SGE), reads the num_proc setting for a particular job. If job_id is not provided and the JOB_ID env is set, it will use that instead (i.e. it uses the current job). This calls qstat to figure out this setting. There are multiple ways this can go wrong, so better catch any exception.

Parameters **job_id** (*int|None*) –

Returns num_proc

Return type int|None

Util.**try_and_ignore_exception**(*f*)

Util.**try_get_caller_name**(*depth=1*, *fallback=None*)

Parameters

- **depth** (int) –
- **fallback** (*str|None*) – this is returned if we fail for some reason

Return type strlNone

Returns caller function name. this is just for debugging

Util.**camel_case_to_snake_case**(*name*)

Parameters **name** (*str*) – e.g. "CamelCase"

Returns e.g. "camel_case"

Return type str

better_exchook

class better_exchook.**Color**(*enable=None*)

Parameters **enable** (*bool|None*) –

ColorIdxTable = {'blue': 4, 'yellow': 3, 'green': 2, 'cyan': 6, 'black': 0, 'magenta': 5, 'white': 7, 'red': 1}

color(*s*, *color=None*, *bold=False*)

Parameters

- **s** (*str*) –
- **color** (*str|None*) – e.g. "blue"
- **bold** (bool) –

Returns s optionally wrapped with ansi escape codes

Return type str

classmethod get_global_color_enabled()

py_syntax_highlight(*s*)

class better_exchook.**DummyFrame**(*filename*, *lineno*, *name*, *f_locals=None*, *f_globals=None*, *f_builtins=None*)

This class has the same attributes as a code and a frame object and is intended to be used as a dummy replacement.

classmethod from_frame_summary(*f*)

Parameters **f** (FrameSummary) –

Return type *DummyFrame*

class better_exchook.**ExtendedFrameSummary**(*frame*, ***kwargs*)

better_exchook.**FrameSummary**
alias of _Dummy

better_exchook.**StackSummary**
alias of _Dummy

better_exchook.**add_indent_lines**(*prefix*, *s*)

better_exchook.**better_exchook**(*etype*, *value*, *tb*, *debugshell=False*, *autodebugshell=True*, *file=None*, *with_color=None*)

better_exchook.**debug_shell**(*user_ns*, *user_global_ns*, *traceback=None*, *execWrapper=None*)

better_exchook.**dump_all_thread_tracebacks**(*exclude_thread_ids=set([])*, *file=None*)

better_exchook.**fallback_findfile**(*filename*)

better_exchook.**format_tb**(*tb=None*, *limit=None*, *allLocals=None*, *allGlobals=None*, *withTitle=False*, *with_color=None*)

better_exchook.**get_indent_prefix**(*s*)

better_exchook.**get_same_indent_prefix**(*lines*)

better_exchook.**get_source_code**(*filename*, *lineno*, *module_globals*)

better_exchook.**grep_full_py_identifiers**(*tokens*)

better_exchook.**install**()

better_exchook.**is_source_code_missing_open_brackets**(*source_code*)

better_exchook.**output_limit**()

better_exchook.**parse_py_statement**(*line*)

better_exchook.**parse_py_statements**(*source_code*)

better_exchook.**pp_extra_info**(*obj*, *depthlimit=3*)

better_exchook.**pretty_print**(*obj*)

better_exchook.**print_tb**(*tb*, *file=None*, ***kwargs*)

better_exchook.**remove_indent_lines**(*s*)

better_exchook.**replace_tab_indent**(*s*, *replace=' '*)

better_exchook.**replace_tab_indents**(*s*, *replace=' '*)

better_exchook.**replace_traceback_format_tb**()

better_exchook.**set_linecache**(*filename*, *source*)

better_exchook.**simple_debug_shell**(*globals*, *locals*)

better_exchook.**str_visible_len**(*s*)

Parameters **s** (*str*) –

Returns len without escape chars

Return type *int*

better_exchook.**test_add_indent_lines**()

better_exchook.**test_get_same_indent_prefix**()

better_exchook.**test_is_source_code_missing_open_brackets**()

better_exchook.**test_remove_indent_lines**()

better_exchook.**to_bool**(*s*, *fallback=None*)

Parameters

- **s** (*str*) – str to be converted to bool, e.g. "1", "0", "true", "false"
- **fallback** (*T*) – if s is not recognized as a bool

Returns boolean value, or fallback

Return type bool|T

collect-orth-symbols

dump-dataset

dump-forward

dump-network-json

extract_state_tying_from_dataset

class extract_state_tying_from_dataset.**OrthHandler**(*lexicon*, *si_label=None*, *allo_num_states=3*, *allo_context_len=1*)

all_allophone_variations(*phon*, *states=None*)

allo_add_all = False

expected_num_labels_for_monophone_state_tying()

iter_orth(*orth*)

orth_to_allophone_states(*orth*)

Parameters **orth** (*str*) – orthography as a str. orth.split() should give words in the lexicon

Return type *list*[*AllophoneState*]

:returns allophone state list. those will have repetitions etc

extract_state_tying_from_dataset.**get_segment_name**(*tree*)

extract_state_tying_from_dataset.**iter_bliss_orth**(*filename*)

extract_state_tying_from_dataset.**iter_dataset_targets**(*dataset*)

extract_state_tying_from_dataset.**main**()

hdf_dump

hdf_dump.**hdf_close**(*hdf_dataset*)

Parameters **hdf_dataset** (*h5py._hl.files.File*) – to close

hdf_dump.**hdf_dataset_init**(*file_name*)

Parameters **file_name** (*str*) – filename of hdf dataset file in the filesystem

Return type h5py._hl.files.File

hdf_dump.**hdf_dump_from_dataset**(*dataset*, *hdf_dataset*, *parser_args*)

Parameters

- **dataset** (Dataset) – could be any dataset implemented as child of Dataset

- **parser_args** – argparse object from main()

Returns

hdf_dump.**init**(*config_filename*, *cmd_line_opts*, *dataset_config_str*)

Parameters

- **config_filename** (*str*) – global config for CRNN
- **cmd_line_opts** (*list[str]*) – options for initConfig method
- **dataset_config_str** (*str*) – dataset via init_dataset_via_str()

hdf_dump.**main**(*argv*)

import-sprint-nn

rnn

Main entry point

This is the main entry point. You can execute this file. See *rnn.initConfig()* for some arguments, or just run `./rnn.py --help`. See *Technological overview* for a technical overview.

rnn.**analyze_data**(*config*)

rnn.**crnnGreeting**(*configFilename=None*, *commandLineOptions=None*)

rnn.**executeMainTask**()

rnn.**finalize**()

rnn.**getCacheByteSizes**()

Return type (*int*,*int*,*int*)

:returns cache size in bytes for (train,dev,eval)

rnn.**init**(*configFilename=None*, *commandLineOptions=()*, *config_updates=None*, *extra_greeting=None*)

Parameters

- **configFilename** (*str|None*) –
- **commandLineOptions** (*tuple[str]|list[str]|None*) –
- **config_updates** (*dict[str]|None*) –
- **extra_greeting** (*str|None*) –

rnn.**initBackendEngine**()

rnn.**initConfig**(*configFilename=None*, *commandLineOptions=()*)

Initializes the global config.

rnn.**initConfigJsonNetwork**()

rnn.**initData**()
Initializes the globals train,dev,eval of type Dataset.

rnn.**initDevices**()

Return type *list*[*Device*]

rnn.**initEngine**(*devices*)

Initializes global engine.

rnn.**initLog**()

rnn.**load_data**(*config*, *cache_byte_size*, *files_config_key*, ***kwargs*)

Parameters

- **config** (Config) –
- **cache_byte_size** (int) –
- **files_config_key** (*str*) – such as “train” or “dev”
- **kwargs** – passed on to init_dataset() or init_dataset_via_str()

Return type (*Dataset*,*int*)

:returns the dataset, and the cache byte size left over if we cache the whole dataset.

rnn.**main**(*argv*)

rnn.**needData**()

rnn.**printTaskProperties**(*devices=None*)

CHAPTER 3

Refs

- genindex
- modindex
- search

Python Module Index

p

r

s

t

u

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X