ELEKTRONN2 Documentation

Marius Killinger

Dec 03, 2018

Contents

ELEKTRONN2 is a Python-based toolkit for training 3D/2D CNNs and general neural networks.

Note: ELEKTRONN2 is being superceded by the more flexible, PyTorch-based elektronn3 library. elektronn3 is actively developed and supported, so we encourage you to use it instead of ELEKTRONN2 (if elektronn3's more experimental status and currently less extensive documentation are acceptable for you).

CHAPTER 1

Introduction

1.1 About ELEKTRONN2

- *License*

1.1.1 Introduction

What is ELEKTRONN2?

ELEKTRONN2 is a flexible and extensible **Python toolkit** that facilitates **design, training and application of neural networks**.

It can be used for general machine learning tasks, but its main focus is on **convolutional neural networks (CNNs) for high-throughput 3D and 2D image analysis**.

ELEKTRONN2 is especially useful for efficiently assessing experimental neural network architectures thanks to its **powerful interactive shell** that can be entered at any time during training, temporarily pausing all calculations.

The shell interface provides shortcuts and autocompletions for frequently used operations (e.g. adjusting the learning rate) and also provides a complete python shell with full read/write access to the network model, the plotting subsystem and all training parameters and hyperparameters. Changes made in the shell take effect immediately, so you can **monitor, analyse and manipulate your training sessions directly during their run time**, without losing any training progress.

Computationally expensive calculations are **automatically compiled and transparently executed as highly-optimized CUDA** binaries on your GPU if a CUDA-compatible graphics card is available[1].

ELEKTRONN2 is written in Python (2.7 / and 3.4+) and is a complete rewrite of the previously published ELEKTRONN library. The largest improvement is the development of a **functional interface that allows easy creation of complex data-flow graphs** with loops between arbitrary points in contrast to simple "chain"-like models. Currently, the only supported platform is Linux (x86_64).

Use cases

Although other high-level libraries are available (Keras, Lasagne), they all lacked desired *features* and flexibility for our work, mostly in terms of an intuitive method to specify complicated computational graphs and utilities for training and data handling, especially in the domain of specialised large-scale 3-dimensional image data analysis for connectomics research (e.g. tracing, mapping and segmenting neurons in in SBEM[2] data sets).

Although the mentioned use cases are ELEKTRONN2's specialty, it can be used and extended for a wide range of other tasks thanks to its modular object-oriented API *design* (for example, new operations can be implemented as subclasses of the Node class and seamlessly integrated into neural network models).

Technology

As the back-end we chose Theano because of good prior experience and competitive performance. With Theano, symbolic computational graphs can be created, in which nodes stand for operations and edges carry values (input and output can be seen as a special operation i.e. a node). Theano is able to compile a graph into a binary executable for different targets (CPU, openMP, GPUs with CUDA support); highly optimised routines from numerical libraries for those targets (e.g. BLAS, cuBLAS, cuDNN) will be used for corresponding operations. Using a symbolic graph facilitates automatic differentiation and optimisation of the graph prior to compilation (e.g. common subexpression elimination, fusion of composed element-wise operations, constant folding). For (convolutional) neural networks the GPU is the most efficient target and Theano can itself be seen as a back-end for low-level CUDA.

[1] You can find out if your graphics card is compatible here. Usage on systems without CUDA is possible but generally not recommended because it is very slow.

[2] Serial Block-Face Scanning Electron Microscopy, a method to generate high resolution 3D images from small samples, such as brain tissue.

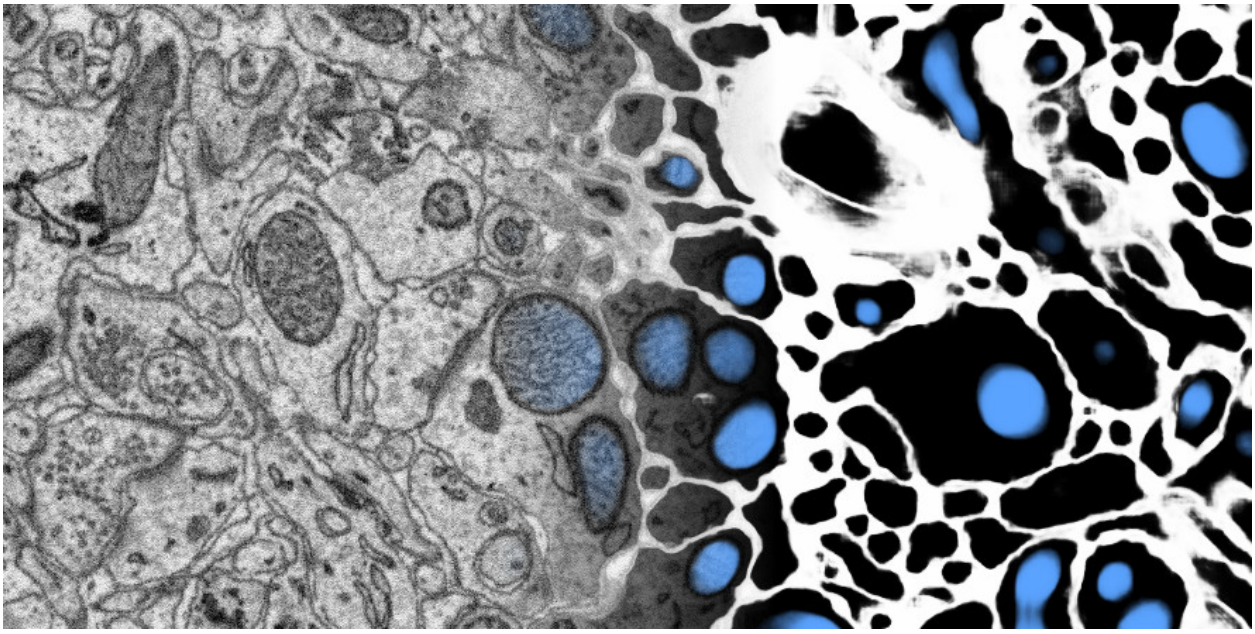

Fig. 1: Example visualisation of ELEKTRONN2's usage on a 3D SBEM data set (blending input to output from left to right).

Left (input: raw data)	Right (output: predictions by ELEKTRONN2, color-coded)
3D electron microscopy images of a zebra finch brain (area X dataset j0126 by Jörgen Kornfeld).	Probability of barriers (union of cell boundaries and extracellular space, marked in **white**) and mitochondria (marked in **blue**) predicted by ELEKTRONN2.

Other dependencies:

- **matplotlib** (plotting of training statistics and prediction previews)
- **h5py** (reading and writing data sets)
- **numpy** (math and data types)
- **scipy** (math and image handling)
- **numba** (accelerating numpy code)
- **future** (maintaining Python 2/3 compatibility)
- **tqdm** (progress bars in the CLI)
- **colorlog** (logging framework)
- **prompt_toolkit** (interactive training shell)
- **jedi** (shell autocompletions)
- **scikit-learn** (cross-validation)
- **scikit-image** (image processing)
- **seaborn** (plot style)
- **pydotplus** (visualizing computation graphs)
- **psutil** (parallelisation)

1.1.2 Design principles

ELEKTRONN2 adds another abstraction layer to Theano. To create a model, the user has to connect different types of node objects and thereby builds a graph as with Theano. But the creation of the raw Theano graph, composed of symbolic variables and trainable model parameters, is hidden and managed through usage of sensible default values and bundling of stereotypical Theano operations into a single ELEKTRONN2 node. For example, creating a convolution layer consists of initialising weights, performing the convolution, adding the bias, applying the activation function and optional operations such as dropout or batch normalisation. Involved parameters might be trainable (e.g. convolution weights) or non-trainable but changeable during training (e.g. dropout rates).

Nodes and layers

Nodes automatically keep track of their parents and children, parameters, computational cost, output shape, spatial field of view, spatial strides etc. Users can call a node object simply like a numpy function. The corresponding Theano compilation is done on demand upon first call; all arguments Theano needs for the compilation process are automatically gathered from the node meta data. Methods for profiling, checking the correct output shape or making dense predictions with a (strided) CNN on arbitrarily shaped input are additionally provided. Shapes are augmented with usage tags e.g. ‘x’, ‘y’, ‘z’ for spatial axes, ‘f’ for the feature axis.

Nodes are mostly generic, e.g. the `Perceptron` node can operate on any input by reading from the input shape tags which axis the dot product should be applied over, irrespective of the total input dimensionality. Likewise there is only one type of convolution node which can handle 1-, 2- and 3-dimensional convolutions and determines the case based on the input shape tags, it does also make replacements of the convolution operation if this makes computation faster: for a 3-dimensional convolution where the filter size is 1 on the z-axis using a 2-dimensional convolution back-end is faster for gradient computation; convolutions where all filter shapes are 1 can be calculated faster using the dot product.

Network models

Whenever a `Node` is created, it is registered internally to a `model` object which also records the exact arguments with which the node was created as node descriptors. The model provides an interface for the trainer by designating nodes as input, target, loss and monitoring outputs. The model also offers functions for plotting the computational graph as image, and showing statistics about gradients, neuron activations and parameters (mean, standard deviation, median).

Furthermore, the `model` offers methods loading and saving from/to disk. Because for this the descriptors are used and not the objects itself, these can programmatically be manipulated before restoration of a saved graph. This is used for: * changing input image size of a CNN (including sanity check of new shape), * inserting Max-Fragment-Pooling (MFP) into a CNN that was trained without MFP, * marking specific parameters as non-trainable for faster training, * changing batch normalisation from training mode to prediction mode * creating a one-step function from a multi-step RNN.

1.1.3 Features

Operations

- Perceptron / fully-connected / dot-product layer, works for arbitrary dimensional input
- Convolution, 1-,2-,3-dimensional
- Max/Average Pooling, 1,2,3-dimensional
- UpConv, 1,2,3-dimensional
- Max Fragment Pooling (MFP), 1,2,3-dimensional
- Gated Recurrent Unit (GRU) and Long Short Term Memory (LSTM) unit
- Recurrence / Scan over arbitrary sub-graph: support of multiple inputs multiple outputs and feed-back of multiple values per iteration
- Batch normalisation with automatic accumulation of whole data set statistics during training
- Gaussian noise layer (for Variational Auto Encoders)
- Activation functions: tanh, sigmoid, relu, prelu, abs, softplus, maxout, softmax-layer
- Local Response Normalisation (LRN), feature-wise or spatially
- Basic operations such as concatenation, slicing, cropping, or element-wise functions

Loss functions

- Bernoulli / Multinoulli negative log likelihood
- Gaussian negative log likelihood
- Squared Deviation Loss, (margin optional)
- Absolute Deviation Loss, (margin optional)
- Weighted sum of losses for multi-task training

Optimisers

- Stochastic Gradient Descent (SGD)
- AdaGrad

- AdaDelta
- Adam

Trainer

- Automatic creation of training directory to which all files (parameters, log files, previews etc.) will be saved
- Frequent printing and logging of current state, iteration speed etc.
- Frequent plotting of monitored states (error samples on training and validation data, classification errors and custom monitoring targets)
- Frequent saving of intermediate parameter states and history of monitored variables
- Frequent preview prediction images for CNN training
- Customisable schedules for non-trainable meta-parameters (e.g. dropout rates, learning rate, momentum)
- Fully functional python command line during training, usable for debugging/inspection (e.g. of inputs, gradient statistics) or for changing meta-parameters

Training Examples for CNNs

- Randomised patch extraction from a list of of input/target image pairs
- Data augmentation trough histogram distortions, rotation, shear, stretch, reflection and perspective distortion
- Real-time data augmentation through a queue with background threads.

Utilities

- Array interface for KNOSSOS data sets with caching, pre-fetching and support for multiple data sets as channel axis.
- Viewer for multichannel 3-dimensional image arrays within the Python runtime
- Function to convert ID images to boundary images
- Utilities needed for skeltonisation agent training and application
- Visualisation of the computational graph
- Class for profiling within loops
- KD Tree that supports append (realised through mixture of KD-Tree and brute-force search and amortised rebuilds)
- Daemon script for the synchronised start of experiments on several hosts, based on resource occupation.

1.1.4 Documentation and usage examples

The documentation is hosted at https://elektronn2.readthedocs.io/ (built automatically from the sources in the `docs/` subdirectory of the code repository).

1.1.5 Contributors

- Marius Killinger (main developer)
- Martin Drawitsch
- Philipp Schubert

ELEKTRONN2 was funded by Winfried Denk's lab at the Max Planck Institute of Neurobiology.

Jörgen Kornfeld was academic advisor to this project.

1.1.6 License

1.2 Practical Introduction to Neural Networks

1.2.1 Fundamentals

Neural Networks (NNs), including Convolutional Neural Networks (CNNs) as a special case, are a class of non-linear, parametrised functions mapping $\mathbb{R}^N \rightarrow \mathbb{R}^M$ - seen from very narrow point of view assumed for the sake of this

practical tutorial. For a more comprehensive description refer to

- Marius' Introduction to Artificial Neural Networks [pdf] (more concise).
- http://neuralnetworksanddeeplearning.com (more detailed and slower-paced)
  - Chapter 1 (about general neural networks).
  - Chapter 6 (about deep convolutional neural networks).

NNs are typically models for *supervised* training (excluding auto encoders, which can be used for *unsupervised* tasks). This means the user must provide input *examples* and corresponding desired outputs, called *labels*. Together they form a *training set*. The deviation between current output and desired output is called *loss*. The parameters of the NN (also called synaptic *weights*) are optimised by gradient descent techniques in order to minimise the loss. Provided the training has been successful in minimising the training loss **and** the training examples cover a representative variety of input data, the NN will *generalise* i.e. it is able to produce good output *predictions* for **unseen** input data. The loss (or classification accuracy) should be evaluated on a hold out *validation* data set to give an estimate of the generalisation capability and to detect over-fitting.

NNs can outperform many other models but they are rather difficult to handle. In particular they demand many training examples and precise labels with little noise. Furthermore they might require a high number of trial runs with different architectures and meta-parameters until a good model is found. The reward is the prospect of state-of-the-art results that often outperform other methods.

1.2.2 Operation Modes in ELEKTRONN2

Generally there are three *modes* of training set ups, supported by the built-in pipeline:

- **img-img**: Input data **and** output data are image-like (i.e. there exists neighbourhood relations between adjacent elements (pixels) in the data arrays. This property is irrespective of dimensionality and can also be seen as neighbourhoods of temporal nature). E.g. for neuron membrane segmentation the inputs are 3D EM-images and the labels images too, that have `1` for pixels in which a membrane is present and `0` for background.

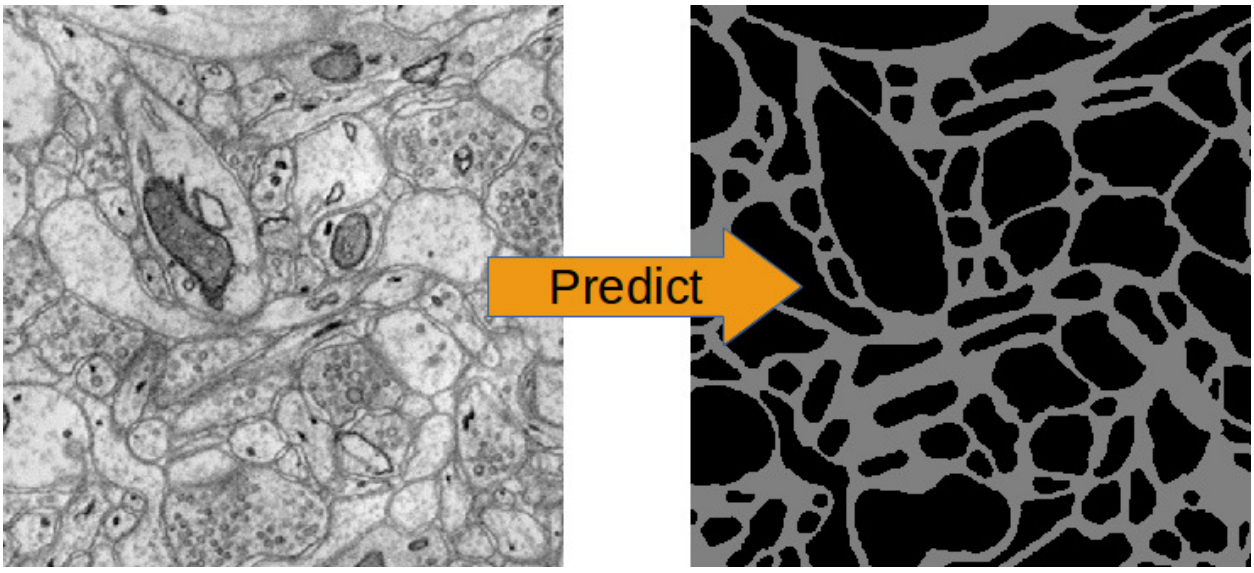

Fig. 2: EM-Images and derived membrane labels from the ISBI 3D data set (the original labels contained neurite *IDs* and not neurite *boundaries*)

- **img-scalar**: The input data is image like, but the labels correspond to the image as a *whole* - not to individual pixels/locations. E.g. "this image shows a cat" or "this is an image of digit 3".

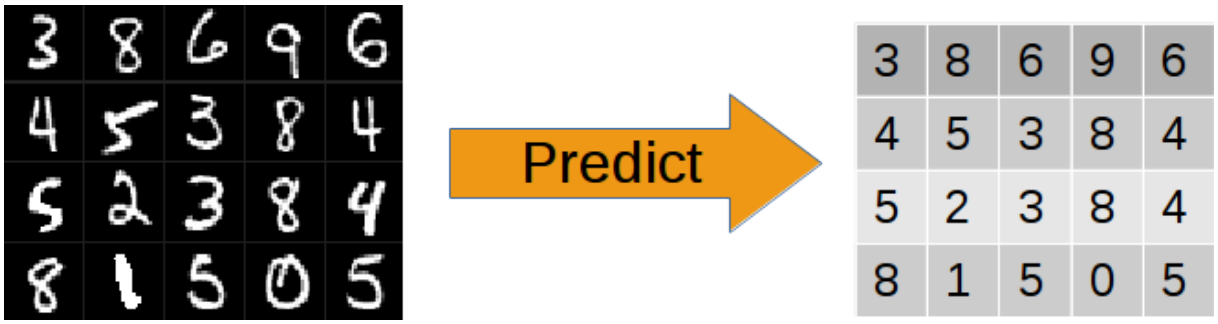

Fig. 3: 20 instances form the MNIST data set and their target labels.

- **vect-scalar**: The input data is a vector of arbitrary features (i.e. there exists no relation between adjacent elements). Common examples of vector data are word counts in a document or demographic properties of persons as in the depicted example:

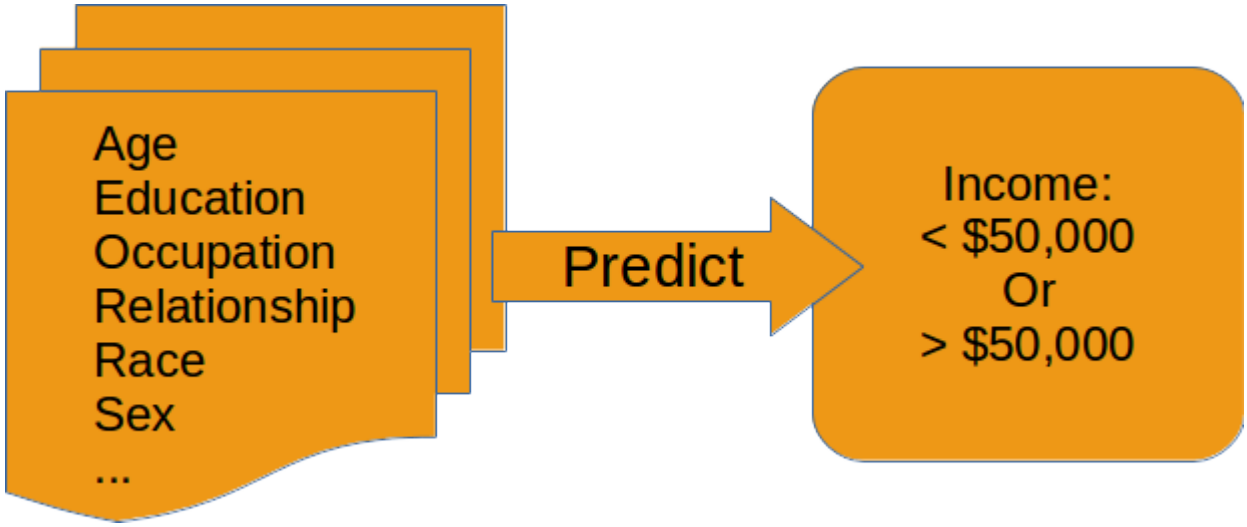

Fig. 4: Exemplary features of the "Adult" data set and the (binary) prediction target.

Convolutions are only applicable for the first and second mode. Note that a neighbourhood relation can also be temporal e.g. a z-*stack* of 2D images (x,y-axes) from a video sequence: the images are ordered by time and a 3D convolution becomes a *spatio-temporal filter*.

1.2.3 Data Considerations

Preprocessing

Training convergence is usually improved a lot by scaling the input data properly. However, there is no single way to do this. Common methods are:

- Normalisation: scale range to `(0,1)`
- Standardisation: subtract mean and scale variance to 1 (where the mean and variances can be computed per pixel, per feature or over all inputs)
- Whitening: like standardisation but including de-correlation of all features

For images, normalisation to `(0,1)` usually works well and is most convenient - images stored as `uint8` (grayscale 0-255) are converted to float and normalised to `(0,1)` automatically by ELEKTRONN2's pipeline.

Label Quality

Training data must in general be densely labelled i.e. for each pixel of a raw image the corresponding pixel in the label image must contain the class index (or a float value for regression). In particular for object detection/classification this means all pixels of that object must carry the index of the class to which the object belongs. E.g. if the task is to detect cats, dogs and birds (their position in an image), every pixel of an animal's silhouette must carry the corresponding class index in the label image. Noise in these annotations is bad (e.g. the labelled area is smaller or larger than the real object extent, the contours do not match). We found that CNNs can tolerate label noise well if there is a large number of training examples, which essentially average out the noise during the training.

The above does not apply to *img-scalar* training, since spatial relations are not considered.

Label Coverage

A CNN can only generalise well if the training data covers a great range/variety of possible inputs.

If you generate ground truth, be aware of the CNN offsets: To make a CNN train at a specific location, an image patch around the location ("context") must be provided as input. So if you have images of a certain size, you cannot make predictions or do training for the pixels which lie in a stripe close to the border. The thickness of this stripe is determined by the offsets, which in turn are determined by the size of the convolution filters. Never label data for in the border stripes in which you cannot make predictions, this is a waste of resources.

1.2.4 Network Architecture

When defining an architecture, several things should be considered:

Note: It should be kept in mind that all training sets and training goals are different and the above instructions are just meant as a general guide. Various architecture versions should be tested against each other to find out what works well for a particular task.

Convolutional Networks

- Filter sizes:
  - Larger filters increase the field of view.
  - Larger filters are slower to compute but do not require significantly more GPU memory.
  - Larger filters introduce more model parameters, but as the number of filters that can be used is limited by speed or GPU memory, the greater "expressiveness" of larger filters might actually not be utilised and smaller filters could be equally good while also being faster.
  - In the very first layer the filter size must be even if pooling by factor 2 is used. Otherwise output neurons lie "between" input pixels.
  - Filter sizes and pooling factors can defined separately for each dimension. This is useful if 3D data has anisotropic resolution or just "a little" information in the z-direction is needed. A useful and fast compromise between a plain 3D and 2D network is a CNN that has e.g. filter shape `(1,4,4)` in the first layers and later `(2,2,2)`: this means the first part is basically a stack of parallel 2D CNNs which are

later concatenated to a 3D CNN. Such "flat" 3D CNNs are faster than their isotropic counterparts. An implementation of this technique can be found in the *"3D Neuro Data" example*.

  - The last layers may have filter sizes `(1,1,1)`, which means no convolution in any dimension and is equivalent to a stack of parallel fully connected layers (where the number of filters corresponds to the neuron count).

- Number of Filters:
  - Due to larger feature map sizes in the first layers (before pooling), fewer filters can be used than in later layers.
  - A large number of filters in later layers may be cheap to compute for training as the feature map sizes are small but predictions still become expensive then.
  - Still it is advisable to have a tendency of increasing filter size for later layers. This can be motivated from the view that early layers extract primitives (such as edges) and the number of relevant primitives is rather small compared to the number of relevant combinations of such primitives.
- Max-Pooling:
  - Reduces the feature map size of that layer, so subsequent layers are cheaper to compute.
  - Adds some translational invariance (e.g. it does not matter if an edge-pattern is a little bit more right or left). This is good to some extent, but too many consecutive poolings will reduce localisation.
  - Increases the field of view of a single output neuron.
  - Results in *strided* output/predictions due to the down-sampling. "Strided" means the neurons after pooling correspond (spatially) to every second input neuron. By applying successive poolings this becomes every fourth, eight and so on, the "stepsize" is called stride. Per layer for a given number of input neurons the number of output neurons is reduced by the pooling factor, this is important because too few output neurons give noisier gradients and the training progress might be slower. Another effect is that poolings make prediction more expensive, because the pixels "between the stride" must be predicted in another forward-pass through the CNN. The simple and slow way is iterating over all positions between the strides and accumulating the strided predictions to a dense image. The fast (and computationally optimal) way is to activate *Max Fragment Pooling (MFP)*, which gives dense images directly but requires a lot of GPU memory.
  - The final strides in each dimension is the product of pooling factors in each dimension (e.g. `2**4 = 16`), the number of total prediction positions (or fragments for MFP) is the product of all pooling factors: in 3D, 4 poolings with factor 2 in all dimensions gives the astonishing number of 4096! As mentioned for the filter sizes below, it is possible to create "flat" 3D CNNs that avoid this, by applying the pooling only in `x` and `y`, not `z` with pooling factors written as `(1,2,2)`.
  - It is recommended to use only poolings in the first layers and not more than in 4 layers in total. The value of the pooling factor should be 2.

Note: To get centered field of views (this means label pixels are aligned with output neurons and do not lie "in between") when using pooling factors of 2, the filter size in the first layer must be even. This is at first counter-intuitive because for an even-sized filter there is no "central" pixel, but if followed by a pooling with factor 2, this results in a centered output in total.

Multi Layer Perceptron (MLP)

`Perceptron` layers are only needed for *img-scalar* training.

The image-like feature maps of the last convolutional layer are *flattened* to a vector and given as input to the perceptron layer; thus one or more perceptron layers can be attached. If the image-like extent of the last convolutional layer is large and/or the layer has many filters the flattened vector might be quite large. It is therefore advisable to reduce the image extent by using maxpooling in the layers to a small extent, e.g. `2 x 2 (x 2)`.

The convolutional part of the network can be interpreted as a feature extractor and perceptron layers as a classificator, but in fact this is rather a continuous transition. Each Perceptron layer is characterised by the number of (output) neurons.

Note: Always check the CNN architecture before starting a training by using the *`cnncalculator()`* function. Only the input shapes listed in the attribute `valid_inputs` can be used. This is also applicable for *img-scalar* training, because for pooling by factor 2, the layers must have even sizes; if the desired architecture is not possible for the size of the images, the images must be constant-padded/cropped to change their size or the architecture must be changed.

1.2.5 Tweaks

A number of helpful CNN/NN tweaks is supported by ELEKETRONN2 and presented in this section.

Class weights

Often data sets are unbalanced (e.g. there are more background pixels than object pixels, or much more people earning less than 50 000 $). In such cases the classifier might get stuck predicting the most frequent class with high probability and assigning little probability to the remaining classes - but not actually learning the discrimination. Using class weights, the training errors (i.e. incentives) can be changed to give the less frequent classes greater importance. This prevents the mentioned problem.

`class_weights` can be specified when initializing loss nodes, e.g.:

```
loss  = neuromancer.MultinoulliNLL(probs, target, class_weights=[0.5, 2.0])
```

will weigh class `0` much less than class `1` (given there are two classes).

Data Augmentation, Warping

CNNs are well-performing classifiers, but require a lot of data examples to generalise well. A method to supply this demand is data *augmentation*: from the limited given data set (potentially infinitely) many examples are created by applying transforms under which the labels are expected to be constant. This is especially well suited for images. In almost all cases small translations and changes in brightness and contrast leave the overall content intact. In many cases rotations, mirroring, little scaling and minor warping deformations are possible, too.

For *img-img* training the labels are subjected to the geometric transformations jointly with the images (preserving the spatial correspondence). By applying the transformations with randomly drawn parameters the training set becomes arbitrarily large. But it should be noted that the augmented training inputs are *highly correlated* compared to genuinely new data. It should furthermore be noted, that the warping deformations require on average greater patch sizes (see black regions in image below) and thus the border regions are exposed to the classifier less frequently. This can be mitigated by applying the warps only to a fraction of the examples.

Warping and general augmentations can be enabled and configured in the *data_batch_args* section of a config file.

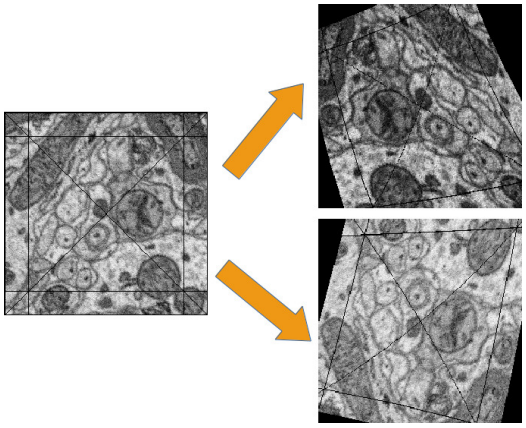

Fig. 5: Two exemplary results of random rotation, flipping, deformation and historgram augmentation. The black regions are only shown for illustration here, internally the data pipeline calculates the required input patch (larger than the CNN input size) such that if cropped to the CNN input size, after the transformation, no missing pixels remain. The labels would be transformed in the same way but are not shown here.

Dropout

Dropout is a major regularisation technique for Neural Networks that improves generalisation. When using dropout for training, a fraction of neurons are turned off - but randomly, changing at every training iteration step.

This can be interpreted as training an *ensemble* of networks (in which the members share common weights) and sampling members randomly every training step. To make a prediction the ensemble average is used, which can be *approximated* by turning all neurons on i.e. setting the dropout rate to 0 (because then the sum of incoming activations at a neuron is larger, the weights are rescaled automatically when changing the rate).

Training with dropout requires more neurons per layer (i.e. more filters for CNNs), larger training times and larger learning rates. We recommend to first narrow down a useful architecture without dropout and from that point start experimenting with dropout.

Dropout rates can be specified by the `dropout_rate` argument when initializing *`Perceptrons`* and any nodes that inherit from it (e.g. *`Conv`*). For example, to make a `Conv` node use a 30% dropout rate, you initialize it with:

```
out = neuromancer.Conv(out, 200, (1,4,4), (1,1,1), dropout_rate=0.3)
```

(Compare this line with the *3D CNN example*, which doesn't use dropout.)

Input Noise

This source of randomisation adds Gaussian noise to the input of a layer (e.g. in the central layer of an auto encoder). Thereby the NN is forced to be invariant and robust against small differences in the input and to generalise better. Input noise is somewhat similar to drop out, but contrast drop out sets certain inputs to 0 randomly.

This feature is provided by the *`GaussianRV`* layer.

1.2.6 Training / Optimisation

Because of the non-linear activation functions, the loss function of a NN is a highly non-convex function of its weights. Analytic solutions do not exist, so we optimize using gradient descent techniques with various heuristics. Convergence is a user-defined state, either determined by good enough results (no progress possible any more) or by the point where

the loss on a held out *validation set* begins to increase, while the loss on the training set still decreases - continuing training in this situation inevitably leads to over-fitting and bad generalisation.

Optimisers

Stochastic Gradient Descent (SGD)

This is the basic principle behind all other optimisers. SGD is the most common optimisation scheme and works in most cases. One advantage of SGD is that it works well with only one example per batch.

In every iteration:

- From the training data one or several examples are drawn. The number of drawn examples is called *batch size*.
- The output of the NN, given the current weights, is calculated
- The **gradient** of the loss (deviation between output and desired output) is calculated w.r.t. the weights
- The weights are *updated* by following down the gradient for a fixed step size - the *learning rate*
- The whole procedure is repeated with a new batch until convergence

The learning rate is usually decreased by schedule over the time of the training (see *schedules*).

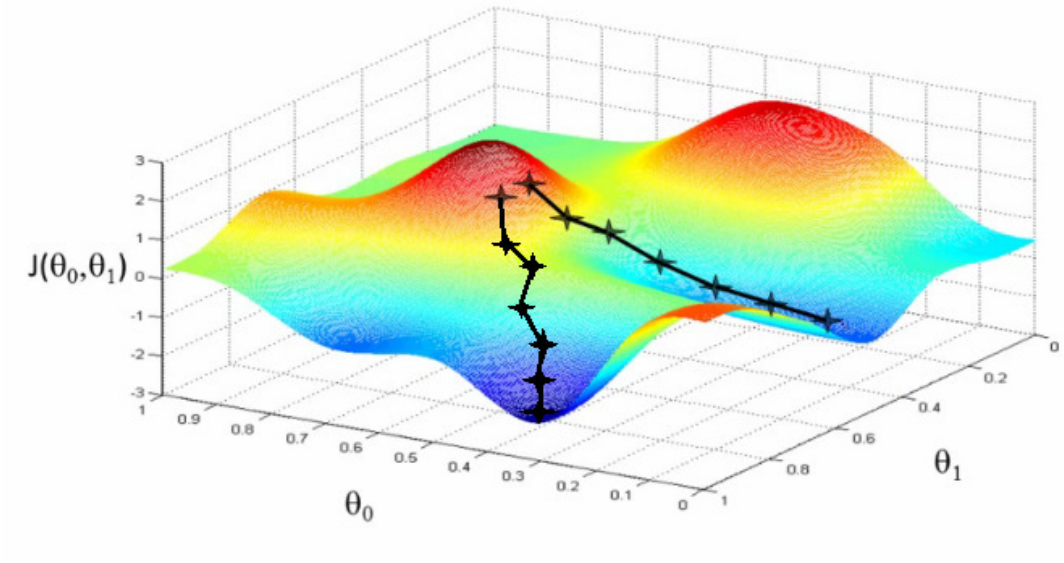

Fig. 6: Illustration of gradient descent on a 2D error surface. This corresponds to a model with just two parameters. As can be seen, the outcome depends on the starting point (a.k.a. *weight initialisation*) and may lead to different *local* optima. For more dimensions the problem of multiple local optima is even more severe. If you train a network twice under same conditions except for the random weight initialisation and the random batch shuffling, you will almost definitely end up in completely different local optima. But empirically the performance is pretty close. In practice, another difficulty is more relevant: saddle-points which may ill-condition the training. [image source]

`elektronn2.neuromancer.optimiser.SGD`

Adam

The Adam optimiser is explained in detail here: https://arxiv.org/abs/1412.6980v9.

We recommend starting training with Adam (especially if you are not sure about your hyperparameter choices, because it is relatively robust).

elektronn2.neuromancer.optimiser.Adam

AdaGrad

elektronn2.neuromancer.optimiser.AdaGrad

AdaDelta

elektronn2.neuromancer.optimiser.AdaDelta

Optimiser hyperparameters

Learning rate

The learning rate should be as large as possible at the beginning of the training and decrease gradually or in steps (optimally always after the loss has plateaued for some time). "As large as possible" here means the following: since the gradient is only a linear approximation, the loss decreases only along a small step size on the gradient and goes up again for larger steps (very quickly). Thus by setting a fixed learning rate, some update steps may in fact lead to an increase of the loss if they are too large. The learning rate should be so large that **most** of the updates decrease the loss but large enough that a few steps lead to increases - because then you know that a greater learning rate would not be possible. The training pipeline creates a plot with the per-step changes of the loss.

The learning rate depends on the NN architecture and the batch size:

- Deeper nets commonly require smaller learning rates.
- Larger batches can go with larger learning rates (there is less noise in the gradients).

Momentum

Momentum replaces the true gradient by an exponential moving average over the previous gradients. This can speed up progress by accumulation of gradients and prevent over-fitting to only the current example by averaging over other examples. Momentum is parametrised by a meta-parameter that determines the mixing rate of the previous gradients and the current gradient. In the picture of the error surface it can be visualised by a massive ball rolling down the hill which, through its mass, can accumulate speed/momentum and also go upwards shortly - across a small ridge for example.

Momentum should be raised towards the end of the training but it can also be kept constant.

A very well written in-depth explanation of momentum can be found in the article Why Momentum Really Works.

Weight Decay

Weight decay is synonymous with a L2 penalty on the weights. This means additional to the loss that comes from the deviation between current output and desired output, large weight values are regarded as loss - the weights are driven

to have smaller magnitudes while at the same time being able to produce good output. This acts as a regulariser (see Tikhonov Regularisation).

More about weight decay can be found in this paper.

You can specify weight decay in the `wd` entry of the *optimiser_params* inside a config.

1.3 Installation

1.3.1 ELEKTRONN2

There are two supported ways to install ELEKTRONN2: With the package managers `conda` or `pip`. We highly recommend using the `conda` install method.

Note: ELEKTRONN2 is supported on Linux (x86_64), with Python versions 2.7, 3.4, 3.5 and 3.6. Everything else is untested, but some other platforms might work as well.

Installing with `conda`

The recommended way to install ELEKTRONN2 is to use the conda package manager, which is included in Anaconda (Miniconda also works). The ELEKTRONN2 package is hosted by the conda-forge channel, so you first need to add it to your local channel list if you haven't yet done this:

```
conda config --add channels conda-forge
```

Then you can either install ELEKTRONN2 directly into your current environment:

```
conda install elektronn2
```

... or create a new conda env just for ELEKTRONN2:

```
conda create -n elektronn2_env elektronn2
```

Optionally run `conda activate elektronn2_env` to activate the new environment and ensure all ELEKTRONN2 executables are on your PATH. The effects of the activation only last for the current shell session, so remember to repeat this step after re-opening your shell.

Installing with `pip`

You can install the current version of ELEKTRONN2 and all of its dependencies with the `pip` package manager. For Python 3, run:

```
python3 -m pip install elektronn2
```

Or if you want to install ELEKTRONN2 for Python 2:

```
python2 -m pip install elektronn2
```

To prevent permission errors and conflicts with other packages, we suggest that you run these `pip install` commands inside a virtualenv or a conda env.

Please **do not** attempt to use `sudo` or the `root` account for `pip install`, because this can overwrite system packages and thus potentially destroy your operating system (this is not specific to ELEKTRONN2, but a general flaw in `pip` and applies to all packages (see this issue). `pip install --user` can be used instead, but this method can also break other Python packages due to the version/precedence conflicts between system and user packages.

1.3.2 CUDA and cuDNN

In order to use Nvidia GPUs for accelleration, you will need to additionally install CUDA. Install Nvidia's CUDA toolkit by following the instructions on the Nvidia website or install it with your system package manager.

Even higher performance for training and inference in deep convolutional neural networks can be enabled by installing the cuDNN library. Once it is installed, ELEKTRONN2 will automatically make use of it.

For example if you use Arch Linux, both libraries can be installed with:

```
sudo pacman -S cuda cudnn
```

If you don't have root rights on your machine, you can install CUDA and cuDNN to a custom path in your `$HOME` directory. If you do that, don't forget to update your environment variables, e.g. by adding these lines to you `~/.bashrc`-file (assuming you have installed both to `~/opt/cuda/`):

```
export PATH=~/opt/cuda/bin:$PATH
export LD_LIBRARY_PATH=~/opt/cuda/lib64:$LD_LIBRARY_PATH
```

1.3.3 Theano configuration

Lastly, the Theano back end needs to be configured. It is responsible for optimizing and compiling ELEKTRONN2's computation graphs. Create the file `~/.theanorc` and put the following lines into it:

```
[global]
floatX = float32
linker = cvm_nogc

[nvcc]
fastmath = True
```

Note: The `cvm_nogc` linker option disables garbage collection. This increases GPU-RAM usage but gives a significant performance boost. If you run out of GPU-RAM, remove this option (or set it to `cvm`).

If your CUDA installation is in a custom location (e.g. `~/opt/cuda/`) and is not found automatically, additionally set the `cuda.root` option in your `~/.theanorc`:

```
[cuda]
root = ~/opt/cuda
```

If you always want to use the same GPU (e.g. GPU number 0) for ELEKTRONN2 and don't want to specify it all the time in your command line, you can also configure it in `~/.theanorc`:

```
[global]
device = gpu0
```

More options to configure Theano can be found in the theano.config documentation.

CHAPTER 2

Tutorial

Code examples for creating, training and deploying neural networks with ELEKTRONN2.

2.1 Examples

This page gives examples for different use cases of ELEKTRONN2. Besides, the examples are intended to give an idea of how custom network architectures could be created and trained without the built-in pipeline. To understand the examples, basic knowledge of neural networks is recommended.

2.1.1 3D CNN architecture and concepts

Here we explain how a simple 3-dimensional CNN and its loss function can be specified in ELEKTRONN2. The model batch size is 10 and the CNN takes an [23, 183, 183] image volume with 3 channels[1] (e.g. RGB colours) as input.

[1] For consistency reasons the axis containing image channels and the axis containing classification targets are also denoted by `'f'` like the feature maps or features of a MLP.

Defining the neural network model

The following code snippet[2] exemplifies how a 3-dimensional CNN model can be built using ELEKTRONN2.

```
# The neuromancer submodule contains the API to define neural network models.
# It is usually accessed via the alias "nm".
from elektronn2 import neuromancer as nm

# Input shape: (batch size, number of features/channels, z, x, y)
image = nm.Input((10, 3, 23, 183, 183), 'b,f,z,x,y', name='image')

# If no node name is given, default names and enumeration are used.
# 3d convolution with 32 filters of size (1,6,6) and max-pool sizes (1,2,2).
# Node constructors always receive their parent node as the first argument
# (except for the root nodes like the 'image' node here).
conv0 = nm.Conv(image, 32, (1,6,6), (1,2,2))
conv1 = nm.Conv(conv0, 64, (4,6,6), (2,2,2))
conv2 = nm.Conv(conv1, 5, (3,3,3), (1,1,1), activation_func='lin')

# Softmax automatically infers from the input's 'f' axis
# that the number of classes is 5 and the axis index is 1.
class_probs = nm.Softmax(conv2)

# This copies shape and strides from class_probs but the feature axis
# is overridden to 1, the target array has only one feature on this axis,
# the class IDs i.e. 'sparse' labels. It is also possible to use 5
# features where each contains the probability for the corresponding class.
target = nm.Input_like(class_probs, override_f=1, name='target', dtype='int16')

# Voxel-wise loss calculation
voxel_loss = nm.MultinoulliNLL(class_probs, target, target_is_sparse=True)
scalar_loss = nm.AggregateLoss(voxel_loss , name='loss')

# Takes class with largest predicted probability and calculates classification␣
↪accuracy.
errors = nm.Errors(class_probs, target, target_is_sparse=True)

# Creation of nodes has been tracked and they were associated to a model object.
model = nm.model_manager.getmodel()

# Tell the model which nodes fulfill which roles.
# Intermediates nodes like conv1 do not need to be explicitly registered
# because they only have to be known by their children.
model.designate_nodes(
    input_node=image,
    target_node=target,
    loss_node=scalar_loss,
    prediction_node=class_probs,
    prediction_ext=[scalar_loss, errors, class_probs]
)
```

`model.designate_nodes()` triggers printing of aggregated model stats and extended shape properties of the `prediction_node`. Executing the above model creation code prints basic information for each node and its output shape and saves it to the log file. Example output:

[2] For complete network config files that you can directly run with little to no modification, see the 3d Neuro Data section below and the "examples" directory in the code repository, especially neuro3d.py.

```
<Input-Node> 'image'
Out:[(10,b), (3,f), (23,z), (183,x), (183,y)]
------------------------------------------------------------------------------------------
↪-
<Conv-Node> 'conv'
#Params=3,488 Comp.Cost=25.2 Giga Ops, Out:[(10,b), (32,f), (23,z), (89,x), (89,y)]
n_f=32, 3d conv, kernel=(1, 6, 6), pool=(1, 2, 2), act='relu',
------------------------------------------------------------------------------------------
↪-
<Conv-Node> 'conv1'
#Params=294,976 Comp.Cost=416.2 Giga Ops, Out:[(10,b), (64,f), (10,z), (42,x), (42,y)]
n_f=64, 3d conv, kernel=(4, 6, 6), pool=(2, 2, 2), act='relu',
------------------------------------------------------------------------------------------
↪-
<Conv-Node> 'conv2'
#Params=8,645 Comp.Cost=1.1 Giga Ops, Out:[(10,b), (5,f), (8,z), (40,x), (40,y)]
n_f=5, 3d conv, kernel=(3, 3, 3), pool=(1, 1, 1), act='lin',
------------------------------------------------------------------------------------------
↪-
<Softmax-Node> 'softmax'
Comp.Cost=640.0 kilo Ops, Out:[(10,b), (5,f), (8,z), (40,x), (40,y)]
------------------------------------------------------------------------------------------
↪-
<Input-Node> 'target'
Out:[(10,b), (1,f), (8,z), (40,x), (40,y)]
85
------------------------------------------------------------------------------------------
↪-
<MultinoulliNLL-Node> 'nll'
Comp.Cost=640.0 kilo Ops, Out:[(10,b), (1,f), (8,z), (40,x), (40,y)]
Order of sources=['image', 'target'],
------------------------------------------------------------------------------------------
↪-
<AggregateLoss-Node> 'loss'
Comp.Cost=128.0 kilo Ops, Out:[(1,f)]
Order of sources=['image', 'target'],
------------------------------------------------------------------------------------------
↪-
<_Errors-Node> 'errors'
Comp.Cost=128.0 kilo Ops, Out:[(1,f)]
Order of sources=['image', 'target'],
Prediction properties:
[(10,b), (5,f), (8,z), (40,x), (40,y)]
fov=[9, 27, 27], offsets=[4, 13, 13], strides=[2 4 4], spatial shape=[8, 40, 40]
Total Computational Cost of Model: 442.5 Giga Ops
Total number of trainable parameters: 307,109.
Computational Cost per pixel: 34.6 Mega Ops
```

Exploring `Model` and `Node` objects

The central concept in ELEKTRONN2 is that a neural network is represented as a Graph of executable `Node` objects that are registered and organised in a `Model`.

In general, we have one `Model` instance that is called `model` by convention (see *elektronn2.neuromancer.model.Model*.

All other variables here are instances of different subclasses of *Node*, which are implemented in the *neuromancer.*

node_basic, *neuromancer.neural*, *neuromancer.loss* and *neuromancer.various* submodules.

For more detailed information about Node and how its subclasses are derived, see the Node *API docs*.

After executing the *above* code (e.g. by %paste-ing into an ipython session or by running the whole file via elektronn2-train and hitting Ctrl+C during training), you can play around with the variables defined there to better understand how they work.

Node objects can be used like functions to calculate their output. The first call triggers compilation and caches the compiled function:

```
>>> import numpy as np
>>> test_input = np.ones(shape=image.shape.shape, dtype=np.float32)
>>> test_output = class_probs(test_input)
Compiling softmax, inputs=[image]
Compiling done - in 21.32 s
>>> np.all(test_output > 0) and np.all(test_output < 1)
True
```

The model object has a dict interface to its Nodes:

```
>>> model
['image', 'conv', 'conv1', 'conv2', 'softmax', 'target', 'nll', 'loss', 'cls for␣
↪errors', 'errors']
>>> model['nll'] == voxel_loss
True
>>> conv2.shape.ext_repr
'[(10,b), (5,f), (8,z), (40,x), (40,y)]\nfov=[9, 27, 27], offsets=[4, 13, 13],
strides=[2 4 4], spatial shape=[8, 40, 40]'
>>> target.measure_exectime(n_samples=5, n_warmup=4)
Compiling target, inputs=[target]
Compiling done - in 0.65 s
86
target samples in ms:
[ 0.019 0.019 0.019 0.019 0.019]
target: median execution time: 0.01903 ms
```

For efficient dense prediction, batch size is changed to 1 and MFP is inserted. To do that, the model must be rebuilt/reloaded. MFP needs a different patch size. The closest possible one is selected:

```
>>> from elektronn2 import neuromancer as nm
>>> model_prediction = nm.model.rebuild_model(model, imposed_batch_size=1,
                                                override_mfp_to_active=True)
patch_size (23) changed to (22) (size not possible)
patch_size (183) changed to (182) (size not possible)
patch_size (183) changed to (182) (size not possible)
------------------------------------------------------------------------------
↪-
<Input-Node> 'image'
Out:[(1,b), (3,f), (22,z), (182,x), (182,y)]
...
```

Dense prediction: test_image can have any spatial shape as long as it is larger than the model patch size:

```
>>> model_prediction.predict_dense(test_image, pad_raw=True)
Compiling softmax, inputs=[image]
Compiling done - in 27.63 s
Predicting img (3, 58, 326, 326) in 16 Blocks: (4, 2, 2)
...
```

The whole model can also be plotted as a graph by using the `elektronn2.utils.d3viz.visualize_model()` method:

```
>>> from elektronn2.utils.d3viz import visualise_model
>>> visualise_model(model, '/tmp/modelgraph')
```

2.1.2 3D Neuro Data (`examples/neuro3d.py`)

Note: This section is under construction and is currently incomplete.

In the following concrete example, ELEKTRONN2 is used for detecting neuron cell boundaries in 3D electron microscopy image volumes. The more general goal is to find a volume segmentation by assigning a cell ID to each voxel. Predicting boundaries is a surrogate target for which a CNN can be trained. The actual segmentation would be made by e.g. running a watershed transformation on the predicted boundary map. This is a typical *img-img* task.

For demonstration purposes, a relatively small CNN with only 3M parameters and 7 layers is used. It trains fast but is obviously limited in accuracy. To solve this task well, more training data would be required in addition.

The full configuration file on which this section is based can be found in ELEKTRONN2's examples folder as neuro3d.py. If your GPU is slow or you want to try ELEKTRONN2 on your CPU, we recommend you use the neuro3d_lite.py config instead. It uses the same data and has the same output format, but it runs significantly faster (at the cost of accuracy).

Getting Started

1. Download and unpack the neuro_data_zxy test data (98 MiB):

   ```
   wget http://elektronn.org/downloads/neuro_data_zxy.zip
   unzip neuro_data_zxy.zip -d ~/neuro_data_zxy
   ```

2. `cd` to the `examples` directory or download the example file to your working directory:

   ```
   wget https://raw.githubusercontent.com/ELEKTRONN/ELEKTRONN2/master/examples/
   ↪neuro3d.py
   ```

4. Run:

   ```
   elektronn2-train neuro3d.py --gpu=auto
   ```

   During training, you can pause the neural network and enter the interactive shell interface by pressing `Ctrl+C`. There you can directly inspect and modify all (hyper-)parameters and options of the training session.

4. Inspect the printed output and the plots in the save directory

5. You can start experimenting with changes in the config file (for example by inserting a new `Conv` layer) and validate your model by directly running the config file through your Python interpreter before trying to train it:

   ```
   python neuro3d.py
   ```

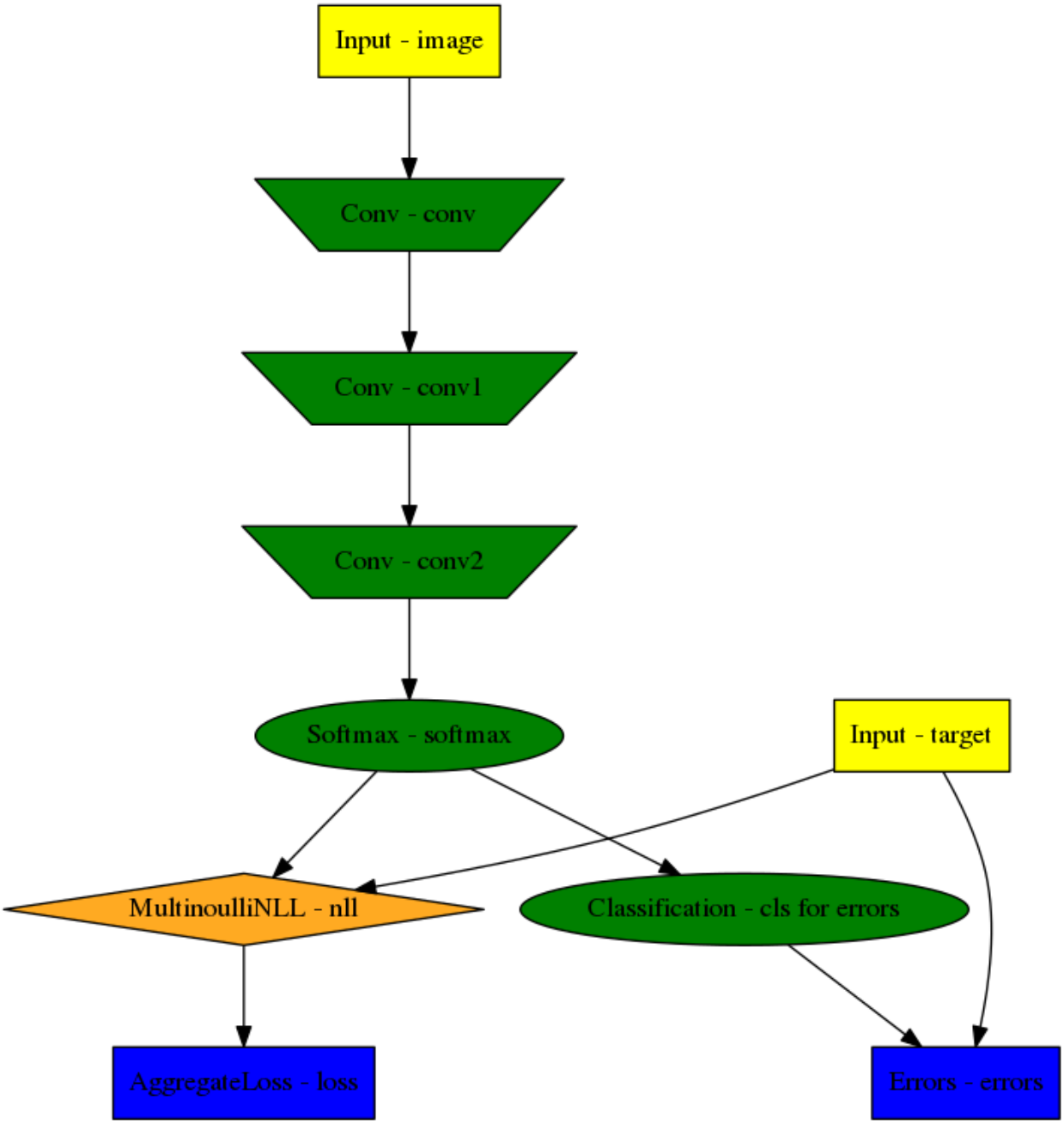

Fig. 1: Model graph of the example CNN. Inputs are yellow and outputs are blue. Some node classes are represented by special shapes, the default shape is oval.

Data Set

This data set is a subset of the zebra finch area X dataset j0126 by Jörgen Kornfeld. There are 3 volumes which contain "barrier" labels (union of cell boundaries and extra cellular space) of shape `(150,150,150)` in `(z,x,y)` axis order. Correspondingly, there are 3 volumes which contain raw electron microscopy images. Because a CNN can only make predictions within some offset from the input image extent, the size of the image cubes is larger `(250, 350,350)` in order to be able to make predictions and to train for every labelled voxel. The margin in this examples allows to make predictions for the labelled region with a maximal field of view of `201` in `x, y` and `101` in `z`.

There is a difference in the lateral dimensions and in `z` - direction because this data set is anisotropic: lateral voxels have a spacing of $10\mu m$ in contrast to $20\mu m$ vertically. Snapshots of images and labels are depicted below.

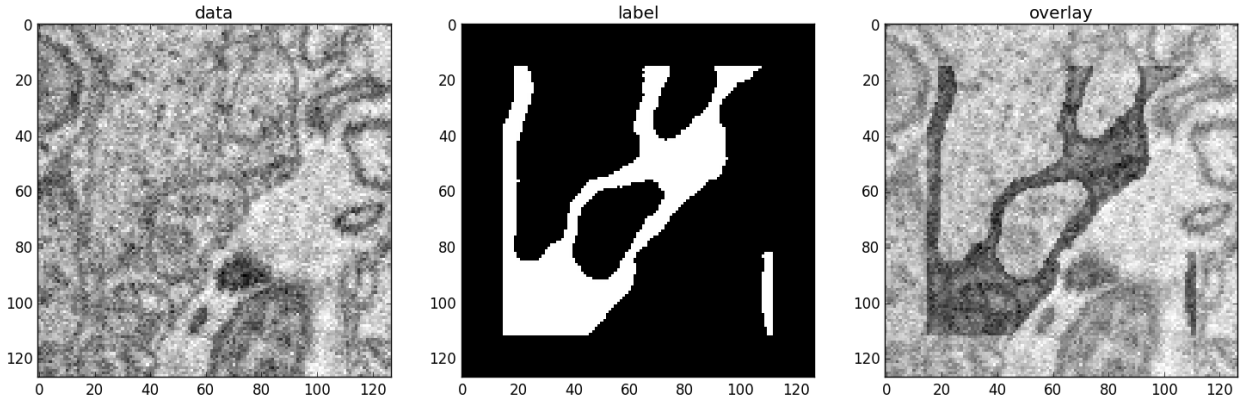

Fig. 2: An example slice of the neuro_data_zxy data set

Left	Center	Right
Raw input data	Barrier labels	Labels overlayed on top of input data.

During training, the pipeline cuts image and target patches from the loaded data cubes at randomly sampled locations and feeds them to the CNN. Therefore the CNN input size should be smaller than the size of the data cubes, to leave enough space to cut from many different positions. Otherwise it will always use the same patch (more or less) and soon over-fit to that one.

During training initialisation a debug plot of a randomly sampled batch is made to check whether the training data is presented to the CNN in the intended way and to find errors (e.g. image and label cubes are not matching or labels are shifted w.r.t. images). Once the training loop has started, more such plots can be made from the ELEKTRONN2 command line (`Ctrl+C`)

```
>>> mfk@ELEKTRONN2: self.debug_getcnnbatch()
```

Note: Implementation details: When the cubes are read into the pipeline, it is implicitly assumed that the smaller label cube is spatially centered w.r.t the larger image cube (hence the size surplus of the image cube must be even). Furthermore, for performance reasons the cubes are internally zero-padded to the same size and cropped such that only the area in which labels and images are both available after considering the CNN offset is used. If labels cannot be effectively used for training (because either the image surplus is too small or your FOV is too large) a note will be printed.

Additionally to the 3 pairs of images and labels, 2 small image cubes for live previews are included. Note that preview data must be a **list** of one or several cubes stored in a `h5`-file.

Data and training options

In this section we explain selected training options in the neuro3d.py config. Training options are usually specified at the top of the config file, before the network model definition. They are parsed by the `elektronn2-train` application and processed and stored in the global `ExperimentConfig` object

preview_kwargs

`preview_kwargs` specifies how preview predictions are generated:

```
preview_kwargs = {
    'export_class': '1',
    'max_z_pred': 3
}
```

- `export_class` is the list of class indices (channels) of predictions that are exported in the preview images. In the case of the *neuro_data_zxy* data set that we use here, `1` is the index of the "barrier" class, so the exported preview images should show a probability map of cell barriers. If you set it to `'all'`, all predicted classes are exported in each preview process.
- `max_z_pred` defines how many subsequent z slices (i.e. images of the `x, y` plane) should be written to disk per preview prediction step). Limiting the values of these options can be useful to reduce clutter in your `save_path` directory.

Note: Internally, `preview_kwargs` specifies non-default arguments for *`elektronn2.training.trainer.Trainer.preview_slice()`*

---

data_init_kwargs

`data_init_kwargs` sets up where the training data is located and how it is interpreted:

```
data_init_kwargs = {
    'd_path' : '~/neuro_data_zxy/',
    'l_path': '~/neuro_data_zxy/',
    'd_files': [('raw_%i.h5' %i, 'raw') for i in range(3)],
    'l_files': [('barrier_int16_%i.h5' %i, 'lab') for i in range(3)],
    'aniso_factor': 2,
    'valid_cubes': [2],
}
```

- `d_path`/`l_path`: Directory paths from which the input images/labels are read.
- `d_files`: A list of tuples each consisting of a file name inside `d_path` and the name of the hdf5 data set that should be read from it. (here the data sets are all named 'raw' and contain grayscale images of brain tissue).
- `l_files`: A list of tuples each consisting of a file name inside `l_path` and the hdf5 data set name. (here: label arrays named 'lab' that contain ground truth for cell barriers in the respective `d_files`).
- `aniso_factor`: Describes anisotropy in the first (`z`) axis of the given data. The data set used here demands an `aniso_factor` of 2 because lateral voxels biologically correspond to a spacing of $10\mu m$, whereas in `z` direction the spacing is $20\mu m$.
- `valid_cubes`: Indices of training data sets that are reserved for validation and never used for training. Here, of the three training data cubes the last one (index `2`) is used as validation data. All training data is stored inside the hdf5 files as 3-dimensional numpy arrays.

Note: Internally, `data_init_kwargs` specifies non-default arguments for the constructor of *elektronn2.data.cnndata.BatchCreatorImage*

data_batch_args

`data_batch_args` determines how batches are prepared and how augmentation is applied:

```
data_batch_args = {
    'grey_augment_channels': [0],
    'warp': 0.5,
    'warp_args': {
        'sample_aniso': True,
        'perspective': True
    }
}
```

- `grey_augment_channels`: List of channels for which grey-value *augmentation* should be applied. Our input images are grey-valued, i.e. they have only 1 channel (with index `0`). For this channel grey value augmentations (randomised histogram distortions) are applied when sampling batches during training. This helps to achieve invariance against varying contrast and brightness gradients.
- `warp`: Fraction of image samples to which *warping transformations* are applied (see also *elektronn2.data.transformations.get_warped_slice()*)
- `warp_args`: Non-default arguments passed to *elektronn2.data.cnndata.BatchCreatorImage.warp_cut()*
  - `sample_aniso`: Scale `z` coordinates by the `aniso_factor`-warp-arg (which defaults to 2, as needed for the neuro_data_xzy data set)
  - `perspective`: Apply random perspective transformations while warping (in extension to affine transformations, which are used default).

Note: Internally, `data_batch_args` specifies non-default arguments for *elektronn2.data.cnndata.BatchCreatorImage.getbatch()*

optimiser, optimiser_params

```
optimiser = 'Adam'
optimiser_params = {
    'lr': 0.0005,
    'mom': 0.9,
    'beta2': 0.999,
    'wd': 0.5e-4
}
```

- `optimiser`: We choose the Adam optimiser because it is known to work well with our data set. Alternative optimisers are `'AdaDelta'`, `'AdaGrad'` and `'SGD'` (see implementations in *elektronn2.neuromancer.optimiser* and the documentation section about *Training / Optimisation*).
- `optimiser_params`:
  - `lr`: *Learning rate* (α).

- `mom`: *Momentum*.
- `beta2`: β_2, i.e. exponential decay rate for *Adam*'s moment estimates. (only applicable to the *Adam* optimiser!)
- `wd`: *Weight decay*

schedules

You can specify `schedules` for hyperparameters (i.e. non-trainable parameters) like learning rates, momentum etc.

`schedules` is a dict whose keys are hyperparameter names and whose values describe the their respective update schedules:

```
schedules = {
  'lr': {'dec': 0.995},
}
```

In this case, we have specified that the `lr` (learning rate) variable should decay by factor 0.995 (meaning a decrease by 0.5%) every 1000 training steps (the step size of 1000 for `'dec'` schedules is currently a global constant).

For other schedule types (linear decay, explicit step-value mappings), see the *`Schedule`* class documentation for reference.

CNN design

The architecture of the CNN is determined by the body of the `create_model` function inside the network config file:

```
from elektronn2 import neuromancer as nm
in_sh = (None,1,23,185,185)
inp = nm.Input(in_sh, 'b,f,z,x,y', name='raw')

out    = nm.Conv(inp, 20,  (1,6,6), (1,2,2))
out    = nm.Conv(out, 30,  (1,5,5), (1,2,2))
out    = nm.Conv(out, 40,  (1,5,5))
out    = nm.Conv(out, 80,  (4,4,4))

out    = nm.Conv(out, 100, (3,4,4))
out    = nm.Conv(out, 100, (3,4,4))
out    = nm.Conv(out, 150, (2,4,4))
out    = nm.Conv(out, 200, (1,4,4))
out    = nm.Conv(out, 200, (1,4,4)))

out    = nm.Conv(out, 200, (1,1,1))
out    = nm.Conv(out,   2, (1,1,1), activation_func='lin')
probs = nm.Softmax(out)

target = nm.Input_like(probs, override_f=1, name='target')
loss_pix  = nm.MultinoulliNLL(probs, target, target_is_sparse=True)

loss = nm.AggregateLoss(loss_pix , name='loss')
errors = nm.Errors(probs, target, target_is_sparse=True)

model = nm.model_manager.getmodel()
model.designate_nodes(
```

(continues on next page)

(continued from previous page)

```
    input_node=inp,
    target_node=target,
    loss_node=loss,
    prediction_node=probs,
    prediction_ext=[loss, errors, probs]
)
return model
```

- Because the data is anisotropic the lateral (`x, y`) FOV is chosen to be larger. This reduces the computational complexity compared to a naive isotropic CNN. Even for genuinely isotropic data this might be a useful strategy if it is plausible that seeing a large lateral context is sufficient to solve the task.
- As an extreme, the presented CNN is partially actually 2D: Only the middle layers (4. - 7.) perform a true 3D aggregation of the features along the z axis. In all other layers the filter kernels have the extent `1` in `z`.
- The resulting FOV is `[15, 105, 105]` (to solve this task well, more than `105` lateral FOV is beneficial, but this would be too much for this simple example...)
- Using this input size gives an output shape of `[5, 21, 21]` i.e. `2205` prediction neurons. For training, this is a good compromise between computational cost and sufficiently many prediction neurons to average the gradient over. Too few output pixel result in so noisy gradients that convergence might be impossible. For making predictions, it is more efficient to re-create the CNN with a larger input size.
- If there are several `100-1000` output neurons, a `batch_size` of `1` (specified directly above the `create_model` method in the config) is commonly sufficient and it is not necessary to compute an average gradient over several images.
- The output shape has strides of `[2, 4, 4]` due to one pooling by 2 in `z` direction and 2 lateral poolings by 2. This means that the predicted `[5, 21, 21]` voxels do not lie laterally adjacent if projected back to the space of the input image: for every lateral output voxel there are `3` voxels separating it from the next output voxel (`1` separating voxel in `z` direction, accordingly) - for those no prediction is available. To obtain dense predictions (e.g. when making the live previews) the method `elektronn2.neuromancer.node_basic.predict_dense()` is used, which moves along the missing locations and stitches the results. For making large scale predictions after training, this can be done more efficiently using MFP.
- To solve this task well, a larger architecture and more training data are needed.

2.2 Making Predictions

2.2.1 Predicting with a trained model

Once you have found a good neural network architecture for your task and you have already trained it with `elektronn2-train`, you can load the saved model from its training directory and make predictions on arbitrary data in the correct format. In general, use *`elektronn2.neuromancer.model.modelload()`* to load a trained model file and *`elektronn2.neuromancer.model.Model.predict_dense()`* to create *dense* predictions for images/volumes of arbitrary sizes (the input image must however be larger than the input patch size). Normally, predictions can only be made with some offset w.r.t. the input image extent (due to the convolutions) but this method provides an option to mirror the raw data such that the returned prediction covers the full extent of the input image (this might however introduce some artifacts because mirroring is not a natural continuation of the image).

For making predictions, you can write a custom prediction script. Here is a small example:

Prediction example for `neuro3d`

(Optional) If you want to predict on a GPU that is not already assigned in `~/.theanorc` or `~/.elektronn2rc`, you need to initialise it **before** the other imports:

```
from elektronn2.utils.gpu import initgpu
initgpu('auto')  # or "initgpu('0')" for the first GPU etc.
```

Find the save directory and choose a model file (you probably want the one called `<save_name>-FINAL.mdl`, or alternatively `<save_name>-LAST.mdl` if the training process is not finished yet) and a file that contains the raw images on which you want to execute the neural network, e.g.:

```
model_path = '~/elektronn2_training/neuro3d/neuro3d-FINAL.mdl'
raw_path = '~/neuro_data_zxy/raw_2.h5'  # raw cube for validation
```

For loading data from (reasonably small) hdf5 files, you can use the *`h5load()`* utility function. Here we load the 3D numpy array called *'raw''* from the input file:

```
from elektronn2.utils import h5load
raw3d = h5load(raw_path, 'raw')
```

Now we load the neural network model:

```
from elektronn2 import neuromancer as nm
model = nm.model.modelload(model_path)
```

Input sizes should be at least as large as the spatial input shape of the model's input node (which you can query by `model.input_node.shape.spatial_shape`). Smaller inputs are automatically padded. Here we take an arbitrary `32x160x160` subvolume of the raw data to demonstrate the predictions:

```
raw3d = raw3d[:32, :160, :160]
```

To match the input node's expected input shape, we need to prepend an empty axis for the single input channel. An empty axis is sufficient because we trained with only 1 input channel here (the uint8 pixel intensities):

```
raw4d = raw3d[None, :, :, :]  # shape: (f=1, z=32, x=160, y=160)

pred = model.predict_dense(raw4d)
```

The numpy array `pred` now contains the predicted output in the shape `(f=2, z=18, x=56, y=56)` (same axis order but different sizes than the input `raw4d` due to convolution padding etc.).

2.2.2 Optimal patch sizes

Prediction speed benefits greatly from larger input patch sizes and MFP (see below). It is recommended to impose a larger patch size when making predictions by loading an already trained model with with the `imposed_patch_size` argmument:

```
from elektronn2 import neuromancer as nm
ps = (103,201,201)
model = nm.model.modelload(model_path, imposed_patch_size=ps)
```

During the network initialization that is launched by calling *`modelload()`*, invalid values of `imposed_patch_size` will be rejected and the first dimension which needs to be changed will be shown. If one of the dimensions does not fit the model, you should be able to find a valid one by by trial and error (either in an IPython session or with a script that loops over possible values until the model compiles successfully),

To find an optimal patch size that works on your hardware, you can use the `elektronn2-profile` command, which varies the input size of a given network model until the RAM limit is reached. The script creates a CSV table of the respective speeds. You can find the fastest input size that just fits in your RAM in that table and use it to make predictions.

Theoretically, predicting the whole image in a single patch, instead of several tiles, would be fastest. For each tile some calculations have to be repeated and the larger the tiles, the more intermediate results can be shared. But this is obviously impossible due to limited GPU-RAM.

Note: GPU-RAM usage can be lowered by enabling garbage collection (set `linker = cvm` in the `[global]` section of `.theanorc`) and by using cuDNN.

2.2.3 Max Fragment Pooling (MFP)

MFP is the computationally optimal way to avoid redundant calculations when making predictions with strided output (as arises from pooling). It requires more GPU RAM (you may need to adjust the input size) but it can speed up predictions by a factor of 2 - 10. The larger the patch size (i.e. the more RAM you have) the faster. Compilation time is significantly longer.

CHAPTER 3

API documentation

Auto-generated documentation from docstrings and signatures.

ELEKTRONN2 consists of 4 main sub-modules:

- `neuromancer`: Classes and functions for designing neural network models
- `training`: Training neural networks
- `data`: Reading and processing data sets
- `utils`: Utility functions and data structures

3.1 elektronn2.neuromancer package

3.1.1 Submodules

elektronn2.neuromancer.computations module

elektronn2.neuromancer.computations.**apply_activation**(*x*, *activation_func*, *b1=None*)

Return an activation function callable matching the name Allowed names: 'relu', 'tanh','prelu', 'sigmoid', 'maxout <i>', 'lin','abs','soft+', 'elu', 'selu'.

Parameters

- **x** (*T.Tensor*) – Input tensor.
- **activation_func** (*str*) – Name of the activation function.
- **b1** – Optional b1 parameter for the activation function. If this is None, no parameter is passed.

Returns Activation function applied to `x`.

Return type T.Tensor

elektronn2.neuromancer.computations.**apply_except_axis**(*x*, *axis*, *func*)

Apply a contraction function on all but one axis.

Parameters

- **x** (*T.Tensor*) – Input tensor.
- **axis** (*int*) – Axis to exclude on application.
- **func** (*function*) – A function with signature `func(x, axis=)` eg T.mean, T.std ...

Returns Contraction of `x`, but of the same dimensionality.

Return type T.Tensor

elektronn2.neuromancer.computations.**conv**(*x*, *w*, *axis_order=None*, *conv_dim=None*, *x_shape=None*, *w_shape=None*, *border_mode='valid'*, *stride=None*)

Apply appropriate convolution depending on input and filter dimensionality. If input `w_shape` is known, conv might be replaced by tensordot

There are static assumptions which axes are spatial.

Parameters

- **x** (*T.Tensor*) –

  Input data (mini-batch).
  Tensor of shape `(b, f, x)`, `(b, f, x, y)`, `(b, z, f, x, y)` or `(b,f,x,y,z)`.

- **w** (*T.Tensor*) –

  Set of convolution filter weights.
  Tensor of shape `(f_out, f_in, x)`, `(f_out, f_in, x, y)`, `(f_out, z, f_in, x, y)` or `(f_out, f_in, x, y, z)`.

- **axis_order** (*str*) –

  (only relevant for 3d)
  `'dnn'` `(b,f,x,y(,z))` or `'theano'` `(b, z, f, x, y)`.

- **conv_dim** (*int*) – Dimensionality of the applied convolution (not the absolute dim of the inputs).
- **x_shape** (*tuple*) – shape tuple (`TaggedShape` supported).
- **w_shape** (*tuple*) – shape tuple, see `w`.
- **border_mode** (*str*) –
  - `'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.
- **stride** (*tuple*) –

  (tuple of len 2)
  Factor by which to subsample the output.

Returns Set of feature maps generated by convolution.

Return type T.Tensor

elektronn2.neuromancer.computations.**dot**(*x*, *W*, *axis=1*)
Calculate a tensordot between 1 axis of x and the first axis of W.

Requires x.shape[axis]==W.shape[0]. Identical to dot if x, W 2d and axis==1.

Parameters

- **x** (*T.Tensor*) – Input tensor.
- **W** (*T.Tensor*) – Weight tensor, (f_in, f_out).
- **axis** (*int*) – Axis on x to apply dot.

Returns x with dot applied. The shape of x changes on axis to n_out.

Return type T.Tensor

elektronn2.neuromancer.computations.**fragmentpool**(*conv_out*, *pool*, *offsets*, *strides*, *spatial_axes*, *mode='max'*)

elektronn2.neuromancer.computations.**fragments2dense**(*fragments*, *offsets*, *strides*, *spatial_axes*)

elektronn2.neuromancer.computations.**maxout**(*x*, *factor=2*, *axis=None*)
Maxpooling along the feature axis.

The feature count is reduces by factor.

Parameters

- **x** (*T.Tensor*) – Input tensor (b, f, x, y), (b, z, f, x, y).
- **factor** (*int*) – Pooling factor.
- **axis** (*int or None*) – Feature axis of x (1 or 2). If None, 5d tensors get axis 2 and all others axis 1.

Returns x with pooling applied.

Return type T.Tensor

elektronn2.neuromancer.computations.**pooling**(*x*, *pool*, *spatial_axes*, *mode='max'*, *stride=None*)
Pooling along spatial axes of 3d and 2d tensors.

There are static assumptions which axes are spatial. The spatial axes must be divisible by the corresponding pooling factor, otherwise the computation might crash later.

Parameters

- **x** (*T.Tensor*) – Input tensor (b, f, x, y), (b, z, f, x, y).
- **pool** (*tuple*) – 2/3-tuple of pooling factors. They refer to the spatial axes of x (x,y)/(z,x,y).
- **spatial_axes** (*tuple*) –
- **mode** (*str*) – Can be any of the modes supported by Theano's dnn_pool(): ('max', 'average_inc_pad', 'average_exc_pad', 'sum').

  'max' (default): max-pooling 'average' or 'average_inc_pad': average-pooling 'sum': sum-pooling
- **stride** (*tuple*) –

Returns x with maxpooling applied. The spatial axes are decreased by the corresponding pooling factors

Return type T.Tensor

elektronn2.neuromancer.computations.**softmax**(*x*, *axis=1*, *force_builtin=False*)
Calculate softmax (pseudo probabilities).

Parameters

- **x** (*T.Tensor*) – Input tensor.
- **axis** (*int*) – Axis on which to apply softmax.
- **force_builtin** (*bool*) – force usage of `theano.tensor.nnet.softmax` (more stable).

Returns `x` with softmax applied, same shape.

Return type T.Tensor

elektronn2.neuromancer.computations.**unpooling**(*x*, *pool*, *spatial_axes*)
Symmetric unpooling with border: s_new = s*pool + pool-1.

Insert values strided, e.g for pool=3: 00x00x00x...00x00.

Parameters

- **x** (*T.Tensor*) – Input tensor.
- **pool** (*int*) – Unpooling factor.
- **spatial_axes** (*list*) – List of axes on which to perform unpooling.

Returns `x` with unpooling applied.

Return type T.Tensor

elektronn2.neuromancer.computations.**unpooling_nd**(*x*, *pool*)

elektronn2.neuromancer.computations.**upconv**(*x*, *w*, *stride*, *x_shape=None*, *w_shape=None*, *axis_order='dnn'*)

elektronn2.neuromancer.computations.**upsampling**(*x*, *pool*, *spatial_axes*)
Upsamling through repetition: s_new = s*p.

e.g for pool=3: aaabbbccc...

Parameters

- **x** (*T.Tensor*) – Input tensor.
- **pool** (*int*) – Upsampling factor.
- **spatial_axes** (*list*) – List of axes on which to perform upsampling.

Returns `x` with upsampling applied.

Return type T.Tensor

elektronn2.neuromancer.computations.**upsampling_nd**(*x*, *pool*)

elektronn2.neuromancer.graphmanager module

class elektronn2.neuromancer.graphmanager.**GraphManager**(*name=''*)
Bases: `object`

function_code()

static get_lock_names(*node_descriptors*, *count*)

node_count

plot()

register_node(*node*, *name*, *args*, *kwargs*)

register_split(*node*, *func*, *name*, *args*, *kwargs*)

reset()

restore(*descriptors*, *override_mfp_to_active=False*, *make_weights_constant=False*, *replace_bn=False*, *lock_trainable_params=False*, *unlock_all_params=False*)

serialise()

sinks

sources

elektronn2.neuromancer.graphutils module

class elektronn2.neuromancer.graphutils.**TaggedShape**(*shape*, *tags*, *strides=None*, *mfp_offsets=None*, *fov=None*)

Bases: object

Object to manage shape and associated tags uniformly. The []-operator can be used get shape values by either index (int) or tag (str)

Parameters

- **shape** (*list/tuple of int*) – shape of array, unspecified shapes are None
- **tags** (*list/tuple of strings or comma-separated string*) – tags indicate which purpose the dimensions of the tensor serve. They are sometimes used to decide about reshapes. The maximal tensor has tags: "r, b, s, f, z, y, x, s" which denote: * r: perform recurrence along this axis * b: batch size * s: samples of the same instance (over which expectations are calculated) * f: features, filters, channels * z: convolution no. 3 (slower than 1,2) * y: convolution no. 1 * x: convolution no. 2
- **strides** – list of strides, only for spatial dimensions, so it is 1-3 long
- **mfp_offsets** –

addaxis(*axis*, *size*, *tag*)

Create new TaggedShape with new axis inserted at axis of size size tagged tag. If axis is a tag, the new axis is **right** of that tag

copy()

delaxis(*axis*)

Create new TaggedShape with new axis inserted at axis of size size tagged tag. If axis is a tag, the new axis is **right** of that tag

ext_repr

fov

fov_all_centered

hastag(*tag*)

mfp_offsets

ndim

offsets

shape

spatial_axes

spatial_shape

spatial_size

strides

stripbatch_prod
Calculate product excluding batch dimension

stripnone
Return the shape but with all None elements removed (e.g. if batch size is unspecified)

stripnone_prod
Return the product of the shape but with all None elements removed (e.g. if batch size is unspecified)

tag2index(*target_tag*)
Finds the index of the desired tag

tags

updatefov(*axis*, *new_fov*)
Create new TaggedShape with `new_fov` on `axis`. Axis is given as index of the spatial axes (not matching the absolute index of sh).

updatemfp_offsets(*mfp_offsets*)

updateshape(*axis*, *new_size*, *mode=None*)
Create new TaggedShape with `new_size` on `axis` . Modes for updating: `None` (override), `'add'`, `'mult'`

updatestrides(*strides*)

elektronn2.neuromancer.graphutils.**as_floatX**(*x*)

class elektronn2.neuromancer.graphutils.**make_func**(*tt_input*, *tt_output*, *updates=None*, *name='Unnamed Function'*, *borrow_inp=False*, *borrow_out=False*, *profile_execution=False*)

Bases: `object`

Wrapper for compiled theano functions. Features:

- The function is compiled on demand (i.e. no wait at initialisation)
- Singleton return values are returned directly, multiple values as list
- The last execution time can inspected in the attribute `last_exec_time`
- Functions can be timed: `profile_execution` is an `int` that specifies the number of runs to average. The average time is printed then.
- In/Out values can have a `borrow` flag which might overwrite the numpy arrays but might speed up execution (see theano doc)

compile(*profile=False*)

elektronn2.neuromancer.graphutils.**getinput_for_multioutput**(*outputs*)
For list of several output layers return a list of required input tensors (without duplicates) to compute these outputs.

elektronn2.neuromancer.loss module

class elektronn2.neuromancer.loss.**GaussianNLL**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Similar to squared loss but "modulated" in scale by the variance.

Parameters

- **target** (Node) – True value (target), usually directly an input node
- **mu** (Node) – Mean of the predictive Gaussian density
- **sig** (Node) – Sigma of the predictive Gaussian density
- **sig_is_log** (*bool*) – Whether `sig` is actually the ln(sig), then it is exponentiated internally

Computes element-wise:

$$0.5 \cdot (ln(2\pi\sigma)) + (target - \mu)^2/\sigma^2)$$

class elektronn2.neuromancer.loss.**BinaryNLL**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Binary NLL node. Identical to cross entropy.

Parameters

- **pred** (Node) – Predictive Bernoulli probability.
- **target** (Node) – True value (target), usually directly an input node.

Computes element-wise:

$$-(target ln(pred) + (1 - target) ln(1 - pred))$$

class elektronn2.neuromancer.loss.**AggregateLoss**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

This node is used to average the individual losses over a batch (and possibly, spatial/temporal dimensions). Several losses can be mixed for multi-target training.

Parameters

- **parent_nodes** (*list/tuple of graph or single node*) – each component is some (possibly element-wise) loss array
- **mixing_weights** (*list/None*) – Weights for the individual costs. If none, then all are weighted equally. If mixing weights are used, they can be changed during training by manipulating the attribute `params['mixing_weights']`.
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.
- **The following is all wrong, mixing_weights are directly used** (*#*) –
- **losses are first summed per component, and then the component sums** (*The*) –

- **summed using the relative weights. The resulting scalar is finally** (*are*) –
- **such that** (*normalised*) –
  - The cost does not grow with the number of mixed components
  - Components which consist of more individual losses have more weight e.g. If there is a constraint on some hidden representation with 20 features and a constraint the reconstruction of 100 features, the reconstruction constraint has 5x more impact on the overall loss than the constraint on the hidden state (provided those two loss are initially on the same scale). If they are intended to have equal impact, the weights should be used to upscale the constraint against the reconstruction.

class elektronn2.neuromancer.loss.**SquaredLoss**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Squared loss node.

Parameters

- **pred** (Node) – Prediction node.
- **target** (*T.Tensor*) – corresponds to a vector that gives the correct label for each example. Labels < 0 are ignored (e.g. can be used for label propagation).
- **margin** (*float or None*) –
- **scale_correction** (*float or None*) – Downweights absolute deviations for large target scale. The value specifies the target value at which the square deviation has half weight compared to target=0 If the target is twice as large as this value the downweight is 1/3 and so on. Note: the smaller this value the stronger the effect. No effect would be +inf
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.loss.**AbsLoss**(***kwargs*)
Bases: *elektronn2.neuromancer.loss.SquaredLoss*

AbsLoss node.

Parameters

- **pred** (Node) – Prediction node.
- **target** (*T.Tensor*) – corresponds to a vector that gives the correct label for each example. Labels < 0 are ignored (e.g. can be used for label propagation).
- **margin** (*float or None*) –
- **scale_correction** (*float or None*) – Boosts loss for large target values: if target=1 the error is multiplied by this value (and linearly for other targets)
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.loss.**Softmax**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Softmax node.

Parameters

- **parent** (Node) – Input node.

- **n_class** –
- **n_indep** –
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.loss.**MultinoulliNLL**(***kwargs*)

Bases: *elektronn2.neuromancer.node_basic.Node*

Returns the symbolic mean and instance-wise negative log-likelihood of the prediction of this model under a given target distribution.

Parameters

- **pred** (Node) – Prediction node.
- **target** (*T.Tensor*) – corresponds to a vector that gives the correct label for each example. Labels < 0 are ignored (e.g. can be used for label propagation).
- **target_is_sparse** (*bool*) – If the target is sparse.
- **class_weights** (*T.Tensor*) – weight vector of float32 of length `n_lab`. Values: `1.0` (default), `w < 1.0` (less important), `w > 1.0` (more important class).
- **example_weights** (*T.Tensor*) – weight vector of float32 of shape `(bs, z, x, y)` that can give the individual examples (i.e. labels for output pixels) different weights. Values: `1.0` (default), `w < 1.0` (less important), `w > 1.0` (more important example). Note: if this is not normalised/bounded it may result in a effectively modified learning rate!
- **weakness** (*float*) – Should be a number between 0 and 1. 0 (default) disables weak learning. If > 0, mix real targets with network outputs to achieve less accurate but softer training labels. Can improve convergence.
- **following refers to lazy labels, the masks are always on a per patch basis, depending on the** (*The*) –
- **cube of the patch. The masks are properties of the individual image cubes and must be loaded** (*origin*) –
- **CNNData.** (*into*) –
- **mask_class_labeled** (*T.Tensor*) – shape = (batchsize, num_classes). Binary masks indicating whether a class is properly labeled in `y`. If a class `k` is (in general) present in the image patches **and** `mask_class_labeled[k]==1`, then the labels **must** obey `y==k` for all pixels where the class is present. If a class `k` is present in the image, but was not labeled (-> cheaper labels), set `mask_class_labeled[k]=0`. Then all pixels for which the `y==k` will be ignored. Alternative: set `y=-1` to ignore those pixels. Limit case: `mask_class_labeled[:]==1` will result in the ordinary NLL.
- **mask_class_not_present** (*T.Tensor*) – shape = (batchsize, num_classes). Binary mask indicating whether a class is present in the image patches. `mask_class_not_present[k]==1` means that the image does **not** contain examples of class `k`. Then for all pixels in the patch, class `k` predictive probabilities are trained towards `0`. Limit case: `mask_class_not_present[:]==0` will result in the ordinary NLL.
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

Examples

- A cube contains no class `k`. Instead of labelling the remaining classes they can be marked as unlabelled by the first mask (`mask_class_labeled[:]==0`, whether `mask_class_labeled[k]` is `0` or `1` is actually indifferent because the labels should not be `y==k` anyway in this case). Additionally `mask_class_not_present[k]==1` (otherwise `0`) to suppress predictions of `k` in in this patch. The actual value of the labels is indifferent, it can either be `-1` or it could be the background class, if the background is marked as unlabelled (i.e. then those labels are ignored).
- Only part of the cube is densely labelled. Set `mask_class_labeled[:]=1` for all classes, but set the label values in the unlabelled part to `-1` to ignore this part.
- Only a particular class `k` is labelled in the cube. Either set all other label pixels to `-1` or the corresponding flags in `mask_class_labeled` for the unlabelled classes.

Note: Using `-1` labels or telling that a class is not labelled, is somewhat redundant and just supported for convenience.

---

class elektronn2.neuromancer.loss.**MalisNLL**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Malis NLL node. (See https://github.com/TuragaLab/malis)

Parameters

- **pred** (Node) – Prediction node.
- **aff_gt** (*T.Tensor*) –
- **seg_gt** (*T.Tensor*) –
- **nhood** (*np.ndarray*) –
- **unrestrict_neg** (*bool*) –
- **class_weights** (*T.Tensor*) – weight vector of float32 of length `n_lab`. Values: `1.0` (default), `w < 1.0` (less important), `w > 1.0` (more important class).
- **example_weights** (*T.Tensor*) – weight vector of float32 of shape `(bs, z, x, y)` that can give the individual examples (i.e. labels for output pixels) different weights. Values: `1.0` (default), `w < 1.0` (less important), `w > 1.0` (more important example). Note: if this is not normalised/bounded it may result in a effectively modified learning rate!
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

elektronn2.neuromancer.loss.**Errors**(*pred*, *target*, *target_is_sparse=False*, *n_class='auto'*, *n_indep='auto'*, *name='errors'*, *print_repr=True*)

class elektronn2.neuromancer.loss.**BetaNLL**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Similar to BinaryNLL loss but "modulated" in scale by the variance.

Parameters

- **target** (Node) – True value (target), usually directly an input node, must be in range [0,1]
- **mode** (Node) – Mode of the predictive Beta density, must come from linear activation function (will be transformed by exp(.) + 2)
- **concentration** (*node*) – concentration of the predictive Beta density

Computes element-wise:

$$0.5 \cdot 2$$

elektronn2.neuromancer.loss.**SobelizedLoss**(*pred*, *target*, *loss_type='abs'*, *loss_kwargs=None*)

SobelizedLoss node.

Parameters

- **pred** (Node) – Prediction node.
- **target** (*T.Tensor*) – corresponds to a vector that gives the correct label for each example. Labels < 0 are ignored (e.g. can be used for label propagation).
- **loss_type** (*str*) – Only "abs" is supported.
- **loss_kwargs** (*dict*) – kwargs for the AbsLoss constructor.

Returns The loss node.

Return type *Node*

class elektronn2.neuromancer.loss.**BlockedMultinoulliNLL**(**kwargs*)

Bases: *elektronn2.neuromancer.node_basic.Node*

Returns the symbolic mean and instance-wise negative log-likelihood of the prediction of this model under a given target distribution.

Parameters

- **pred** (Node) – Prediction node.
- **target** (*T.Tensor*) – corresponds to a vector that gives the correct label for each example. Labels < 0 are ignored (e.g. can be used for label propagation).
- **blocking_factor** (*float*) – Blocking factor.
- **target_is_sparse** (*bool*) – If the target is sparse.
- **class_weights** (*T.Tensor*) – weight vector of float32 of length `n_lab`. Values: `1.0` (default), `w < 1.0` (less important), `w > 1.0` (more important class).
- **example_weights** (*T.Tensor*) – weight vector of float32 of shape `(bs, z, x, y)` that can give the individual examples (i.e. labels for output pixels) different weights. Values: `1.0` (default), `w < 1.0` (less important), `w > 1.0` (more important example). Note: if this is not normalised/bounded it may result in a effectively modified learning rate!
- **following refers to lazy labels, the masks are always on a per patch basis, depending on the** (*The*) –
- **cube of the patch. The masks are properties of the individual image cubes and must be loaded** (*origin*) –
- **CNNData.** (*into*) –
- **mask_class_labeled** (*T.Tensor*) – shape = (batchsize, num_classes). Binary masks indicating whether a class is properly labeled in `y`. If a class `k` is (in general) present in the image patches **and** `mask_class_labeled[k]==1`, then the labels **must** obey `y==k` for all pixels where the class is present. If a class `k` is present in the image, but was not labeled (-> cheaper labels), set `mask_class_labeled[k]=0`. Then all pixels for which the `y==k` will be ignored. Alternative: set `y=-1` to ignore those pixels. Limit case: `mask_class_labeled[:]==1` will result in the ordinary NLL.

- **mask_class_not_present** (*T.Tensor*) – shape = (batchsize, num_classes). Binary mask indicating whether a class is present in the image patches. `mask_class_not_present[k]==1` means that the image does **not** contain examples of class `k`. Then for all pixels in the patch, class `k` predictive probabilities are trained towards `0`. Limit case: `mask_class_not_present[:]==0` will result in the ordinary NLL.
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

Examples

- A cube contains no class `k`. Instead of labelling the remaining classes they can be marked as unlabelled by the first mask (`mask_class_labeled[:]==0`, whether `mask_class_labeled[k]` is `0` or `1` is actually indifferent because the labels should not be `y==k` anyway in this case). Additionally `mask_class_not_present[k]==1` (otherwise `0`) to suppress predictions of `k` in in this patch. The actual value of the labels is indifferent, it can either be `-1` or it could be the background class, if the background is marked as unlabelled (i.e. then those labels are ignored).
- Only part of the cube is densely labelled. Set `mask_class_labeled[:]=1` for all classes, but set the label values in the unlabelled part to `-1` to ignore this part.
- Only a particular class `k` is labelled in the cube. Either set all other label pixels to `-1` or the corresponding flags in `mask_class_labeled` for the unlabelled classes.

Note: Using `-1` labels or telling that a class is not labelled, is somewhat redundant and just supported for convenience.

---

class elektronn2.neuromancer.loss.**OneHot**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Onehot node.

Parameters

- **target** (*T.Tensor*) – Target tensor.
- **n_class** (*int*) –
- **axis** –
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.loss.**EuclideanDistance**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Euclidean distance node.

Parameters

- **pred** (Node) – Prediction node.
- **target** (*T.Tensor*) – corresponds to a vector that gives the correct label for each example. Labels < 0 are ignored (e.g. can be used for label propagation).
- **margin** (*float/None*) –

- **scale_correction** (*float/None*) – Downweights absolute deviations for large target scale. The value specifies the target value at which the square deviation has half weight compared to target=0 If the target is twice as large as this value the downweight is 1/3 and so on. Note: the smaller this value the stronger the effect. No effect would be +inf
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.loss.**RampLoss**(***kwargs*)

Bases: *elektronn2.neuromancer.node_basic.Node*

RampLoss node.

Parameters

- **pred** (Node) – Prediction node.
- **target** (*T.Tensor*) – corresponds to a vector that gives the correct label for each example. Labels < 0 are ignored (e.g. can be used for label propagation).
- **margin** (*float/None*) –
- **scale_correction** (*float/None*) – downweights absolute deviations for large target scale. The value specifies the target value at which the square deviation has half weight compared to target=0 If the target is twice as large as this value the downweight is 1/3 and so on. Note: the smaller this value the stronger the effect. No effect would be +inf
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

elektronn2.neuromancer.model module

class elektronn2.neuromancer.model.**Model**(*name=''*)

Bases: *elektronn2.neuromancer.graphmanager.GraphManager*

Represents a neural network model and its current training state.

The Model is defined and checked by running Model.designate_nodes() with appropriate Nodes as arguments (see example in examples/numa_mnist.py).

Permanently saving a Model with its respective training state is possible with the Model.save() function. Loading a Model from a file is done by elektronn2.neuromancer.model.modelload().

During training of a neural network, you can access the current Model via the interactive training shell as the variable "model" (see elektronn2.training.trainutils.user_input()). There are several statistics and hyperparameters of the Model that you can inspect and set directly in the shell, e.g. entering >>> model.lr = 1e-3 and exiting the prompt again effectively sets the learning rate to 1e-3 for further training. (This can also be done with the shortcut "setlr 1e-3".)

activations(**args*, ***kwargs*)

actstats(**args*, ***kwargs*)

batch_normalisation_active

Check if batch normalisation is active in any Node in the Model.

debug_output_names

If debug_outputs is set, a list of all debug output names is returned.

designate_nodes(*input_node='input'*, *target_node=None*, *loss_node=None*, *prediction_node=None*, *prediction_ext=None*, *error_node=None*, *debug_outputs=None*)
Register input, target and other special Nodes in the Model.

Most of the Model's attributes are derived from the Nodes that are given as arguments here.

dropout_rates
Get dropout rates.

get_param_values(*skip_const=False*, *as_list=False*)
Only use this to save/load parameters!

Returns a dict of mapping the values of the params (such that they can be saved to disk) :param skip_const: whether to exclude constant parameters

gradients(**args*, ***kwargs*)

gradnet_rates
Get gradnet rates.

Description: https://arxiv.org/abs/1511.06827

gradstats(**args*, ***kwargs*)

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

loss_input_shapes
Get shape(s) of loss nodes' input node(s).

The return value is either a shape (if one input) or a list of shapes (if multiple inputs).

loss_smooth
Get average loss during the last training steps.

The average is calculated over the last n steps, where n is defined by the config variable time_per_step_smoothing_length (default: 50).

lr
Get learning rate.

measure_exectimes(*n_samples=5*, *n_warmup=4*, *print_info=True*)
Return an OrderedDict that maps node names to their estimated execution times in milliseconds.

Parameters are the same as in elektronn2.neuromancer.node_basic.Node.measure_exectime()

mixing
Get mixing weights.

mom
Get momentum.

paramstats()

predict(**args*, ***kwargs*)

predict_dense(*raw_img*, *as_uint8=False*, *pad_raw=False*)

predict_ext(**args*, ***kwargs*)

prediction_feature_names
If a prediction node is set, return its feature names.

save(*file_name*)
Save a Model (including its training state) to a pickle file. :param file_name: File name to save the Model in.

set_opt_meta_params(*opt_name*, *value_dict*)

set_param_values(*value_dict*, *skip_const=False*)
Only use this to save/load parameters!

Sets new values for non constant parameters :param value_dict: dict mapping values by parameter name / or file name thereof :param skip_const: if dict also maps values for constants, these can be skipped, otherwise an exception is raised.

test_run_prediction()
Execute test run on the prediction node.

time_per_step
Get average run time per training step.

The average is calculated over the last n steps, where n is defined by the config variable time_per_step_smoothing_length (default: 50).

trainingstep(**args*, ***kwargs*)
Perform one optimiser iteration. Optimisers can be chosen by the kwarg `optimiser`.

Signature: `trainingstep(data, target(, *aux)(, **kwargs**))`

Parameters

- ***args** –
  - **data: floatX array** input [bs, ch (, z, y, x)]
  - **targets: int16 array** [bs,((z,)y,x)] (optional)
  - **(optional) other inputs: np.ndarray** depending in model
- ****kwargs** –
  - **optimiser: str** Name of the chosen optimiser class in *elektronn2.neuromancer.optimiser*
  - **update_loss: Bool** determine current loss *after* update step (e.g. needed for queue, but `get_loss` can also be called explicitly)

Returns

- **loss** (*floatX*) – Loss (nll, squared error etc. . .)
- **t** (*float*) – Time spent on the GPU per step

wd
Get weight decay.

elektronn2.neuromancer.model.**modelload**(*file_name*, *override_mfp_to_active=False*, *imposed_patch_size=None*, *imposed_batch_size=None*, *name=None*, ***model_load_kwargs*)
Load a Model from a pickle file (created by Model.save()).

model_load_kwargs: remove_bn, make_weights_constant (True/False)

elektronn2.neuromancer.model.**kernel_lists_from_node_descr**(*model_descr*)
Extract the tuple (filter_shapes, pool_shapes, mfp) from a model description.

Parameters **model_descr** – Model description OrderedDict.

Returns Tuple (filter_shapes, pool_shapes, mfp).

elektronn2.neuromancer.model.**params_from_model_file**(*file_name*)
Load parameters from a model file.

Parameters **file_name** – File name of the pickled Model.

Returns OrderedDict of model parameters.

elektronn2.neuromancer.model.**rebuild_model**(*model*, *override_mfp_to_active=False*, *imposed_patch_size=None*, *name=None*, ***model_load_kwargs*)

Rebuild a Model by saving it to a file and reloading it from there.

Parameters

- **model** – Model object.
- **override_mfp_to_active** – (See elektronn2.neuromancer.model.modelload()).
- **imposed_patch_size** – (See elektronn2.neuromancer.model.modelload()).
- **name** – New model name.
- **model_load_kwargs** – Additional kwargs for restoring Model (see elektronn2.neuromancer.graphmanager.GraphManager.restore()).

Returns Rebuilt Model.

elektronn2.neuromancer.model.**simple_cnn**(*batch_size*, *n_ch*, *n_lab*, *desired_input*, *filters*, *nof_filters*, *activation_func*, *pools*, *mfp=False*, *tags=None*, *name=None*)

Create a simple Model of a convolutional neural network. :param batch_size: Batch size (how many data samples are used in one

update step).

Parameters

- **n_ch** – Number of channels.
- **n_lab** – Number of distinct labels (classes).
- **desired_input** – Desired input image size. (Must be smaller than the size of the training images).
- **filters** – List of filter sizes in each layer.
- **nof_filters** – List of number of filters for each layer.
- **activation_func** – Activation function.
- **pools** – List of maxpooling factors for each layer.
- **mfp** – List of bools that tell if max fragment pooling should be used in each layer (only intended for prediction).
- **tags** – Tuple of tags for Input node (see docs of elektronn2.neuromancer.node_basic.Input).
- **name** – Name of the model.

Returns Network Model.

elektronn2.neuromancer.neural module

class elektronn2.neuromancer.neural.**Perceptron**(***kwargs*)

Bases: elektronn2.neuromancer.neural.NeuralLayer

Perceptron Layer.

Parameters

- **parent** (*Node or list of Node*) – The input node(s).
- **n_f** (*int*) – Number of filters (nodes) in layer.
- **activation_func** (*str*) – Activation function name.
- **flatten** (*bool*) –
- **batch_normalisation** (*str or None*) – Batch normalisation mode. Can be False (inactive), "train" or "fadeout".
- **dropout_rate** (*float*) – Dropout rate (probability that a node drops out in a training step).
- **name** (*str*) – Perceptron name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.
- **w** (*np.ndarray or T.TensorVariable*) – Weight matrix. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **b** (*np.ndarray or T.TensorVariable*) – Bias vector. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **gamma** – (For batch normalisation) Initialises gamma parameter.
- **mean** – (For batch normalisation) Initialises mean parameter.
- **std** – (For batch normalisation) Initialises std parameter.
- **gradnet_mode** –

make_dual (*parent*, *share_w=False*, ***kwargs*)

Create the inverse of this `Perceptron`.

Most options are the same as for the layer itself. If `kwargs` are not specified, the values of the primal layers are re-used and new parameters are created.

Parameters

- **parent** (`Node`) – The input node.
- **share_w** (*bool*) – If the weights (`w`) should be shared from the primal layer.
- **kwargs** (*dict*) – kwargs that are passed through to the constructor of the inverted Perceptron (see signature of `Perceptron`). `n_f` is copied from the existing node on which `make_dual` is called. Every other parameter can be changed from the original `Perceptron`'s defaults by specifying it in `kwargs`.

Returns The inverted perceptron layer.

Return type *Perceptron*

class elektronn2.neuromancer.neural.**Conv**(***kwargs*)

Bases: *elektronn2.neuromancer.neural.Perceptron*

Convolutional layer with subsequent pooling.

Examples

Examples for constructing convolutional neural networks can be found in examples/neuro3d.py and examples/mnist.py.

Parameters

- **parent** (Node) – The input node.
- **n_f** (*int*) – Number of features.
- **filter_shape** (*tuple*) – Shape of the convolution filter kernels.
- **pool_shape** (*tuple*) – Shape of max-pooling to be applied after the convolution. None (default) disables pooling along all axes.
- **conv_mode** (*str*) – Possible values: * “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”: Zero-pads input image so that the output shape is equal to the input shape (Only supported for odd filter sizes).
- **activation_func** (*str*) – Activation function name.
- **mfp** (*bool*) – Whether to apply Max-Fragment-Pooling in this Layer.
- **batch_normalisation** (*str or False*) – Batch normalisation mode. Can be False (inactive), “train” or “fadeout”.
- **dropout_rate** (*float*) – Dropout rate (probability that a node drops out in a training step).
- **name** (*str*) – Layer name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.
- **W** (*np.ndarray or T.TensorVariable*) – Weight matrix. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **b** (*np.ndarray or T.TensorVariable*) – Bias vector. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **gamma** – (For batch normalisation) Initialises gamma parameter.
- **mean** – (For batch normalisation) Initialises mean parameter.
- **std** – (For batch normalisation) Initialises std parameter.
- **gradnet_mode** –
- **invalidate_fov** (*bool*) – Overrides the computed fov with an invalid value to force later recalculation (experimental).

make_dual(*parent*, *share_w=False*, *mfp=False*, ***kwargs*)

Create the inverse (UpConv) of this Conv node.

Most options are the same as for the layer itself. If kwargs are not specified, the values of the primal layers are re-used and new parameters are created.

Parameters

- **parent** (Node) – The input node.
- **share_w** (*bool*) – If the weights (w) should be shared from the primal layer.
- **mfp** (*bool*) – If max-fragment-pooling is used.
- **kwargs** (*dict*) – kwargs that are passed through to the new UpConv node (see signature of UpConv). n_f and pool_shape are copied from the existing node on which make_dual is called. Every other parameter can be changed from the original Conv's defaults by specifying it in kwargs.

Returns The inverted conv layer (as an UpConv node).

Return type *UpConv*

class elektronn2.neuromancer.neural.**UpConv**(***kwargs*)

Bases: *elektronn2.neuromancer.neural.Conv*

Upconvolution layer. Also known as transposed convolution.

See http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html#transposed-convolution-arithmetic

E.g. pooling + upconv with pool_shape = 3:

```
  x x x x x x x x x    before pooling (not in this layer)
   \|/   \|/   \|/     pooling (not in this layer)
    x     x     x      input to this layer
0 0 x 0 0 x 0 0 x 0 0  unpooling + padding (done in this layer)
   /|\   /|\   /|\     conv on unpooled (done in this layer)
  y y y y y y y y y    result of this layer
```

Parameters

- **parent** (Node) – The input node.
- **n_f** (*int*) – Number of filters (nodes) in layer.
- **pool_shape** (*tuple*) – Size of the UpConvolution.
- **activation_func** (*str*) – Activation function name.
- **identity_init** (*bool*) – Initialise weights to result in pixel repetition upsampling
- **batch_normalisation** (*str or False*) – Batch normalisation mode. Can be False (inactive), "train" or "fadeout".
- **dropout_rate** (*float*) – Dropout rate (probability that a node drops out in a training step).
- **name** (*str*) – Layer name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.
- **w** (*np.ndarray or T.TensorVariable*) – Weight matrix. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **b** (*np.ndarray or T.TensorVariable*) – Bias vector. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **gamma** – (For batch normalisation) Initialises gamma parameter.
- **mean** – (For batch normalisation) Initialises mean parameter.

- **std** – (For batch normalisation) Initialises std parameter.
- **gradnet_mode** –

make_dual(**args*, ***kwargs*)

Create the inverse (UpConv) of this Conv node.

Most options are the same as for the layer itself. If kwargs are not specified, the values of the primal layers are re-used and new parameters are created.

Parameters

- **parent** (Node) – The input node.
- **share_w** (*bool*) – If the weights (w) should be shared from the primal layer.
- **mfp** (*bool*) – If max-fragment-pooling is used.
- **kwargs** (*dict*) – kwargs that are passed through to the new UpConv node (see signature of UpConv). n_f and pool_shape are copied from the existing node on which make_dual is called. Every other parameter can be changed from the original Conv's defaults by specifying it in kwargs.

Returns

The inverted conv layer (as an UpConv node). NOTE: docstring was inherited

Return type *UpConv*

class elektronn2.neuromancer.neural.**Crop**(***kwargs*)

Bases: *elektronn2.neuromancer.node_basic.Node*

This node type crops the output of its parent.

Parameters

- **parent** (Node) – The input node whose output should be cropped.
- **crop** (*tuple or list of ints*) – Crop each spatial axis from either side by this number.
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.neural.**LSTM**(***kwargs*)

Bases: elektronn2.neuromancer.neural.NeuralLayer

Long short term memory layer.

Using an implementation without peepholes in f, i, o, i.e. weights cell state is not taken into account for weights. See http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Parameters

- **parent** (Node) – The input node.
- **memory_states** (Node) – Concatenated (initial) feed-back and cell state (one Node!).
- **n_f** (*int*) – Number of features.
- **activation_func** (*str*) – Activation function name.
- **flatten** –
- **batch_normalisation** (*str or None*) – Batch normalisation mode. Can be False (inactive), "train" or "fadeout".

- **dropout_rate** (*float*) – Dropout rate (probability that a node drops out in a training step).
- **name** (*str*) – Layer name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.
- **w** (*np.ndarray or T.TensorVariable*) – Weight matrix. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **b** (*np.ndarray or T.TensorVariable*) – Bias vector. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **gamma** – (For batch normalisation) Initialises gamma parameter.
- **mean** – (For batch normalisation) Initialises mean parameter.
- **std** – (For batch normalisation) Initialises std parameter.
- **gradnet_mode** –

class elektronn2.neuromancer.neural.**FragmentsToDense**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

class elektronn2.neuromancer.neural.**Pool**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Pooling layer.

Reduces the count of training parameters by reducing the spatial size of its input by the factors given in `pool_shape`.

Pooling modes other than max-pooling can only be selected if cuDNN is available.

Parameters

- **parent** (Node) – The input node.
- **pool_shape** (*tuple*) – Tuple of pooling factors (per dimension) by which the input is downsampled.
- **stride** (*tuple*) – Stride sizes (per dimension).
- **mfp** (*bool*) – If max-fragment-pooling should be used.
- **mode** (*str*) – (only if cuDNN is available) Mode can be any of the modes supported by Theano's dnn_pool(): ('max', 'average_inc_pad', 'average_exc_pad', 'sum').

  'max' (default): max-pooling 'average' or 'average_inc_pad': average-pooling 'sum': sum-pooling
- **name** (*str*) – Name of the pooling layer.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

elektronn2.neuromancer.neural.**Dot**
alias of *elektronn2.neuromancer.neural.Perceptron*

class elektronn2.neuromancer.neural.**FaithlessMerge**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

FaithlessMerge node.

Parameters

- **hard_features** (Node) –
- **easy_features** (Node) –
- **axis** –
- **failing_prob** (*float*) – The higher the more often merge is unreliable
- **hardeasy_ratio** (*float*) – The higher the more often the harder features fail instead of the easy ones
- **name** (*str*) – Name of the pooling layer. print_repr: bool

  Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.neural.**GRU**(***kwargs*)

Bases: elektronn2.neuromancer.neural.NeuralLayer

Gated Recurrent Unit Layer.

Parameters

- **parent** (Node) – The input node.
- **memory_state** (Node) – Memory node.
- **n_f** (*int*) – Number of features.
- **activation_func** (*str*) – Activation function name.
- **flatten** (*bool*) – (Unsupported).
- **batch_normalisation** (*str or None*) – Batch normalisation mode. Can be False (inactive), "train" or "fadeout".
- **dropout_rate** (*float*) – Dropout rate (probability that a node drops out in a training step).
- **name** (*str*) – Layer name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.
- **w** (*np.ndarray or T.TensorVariable*) – (Unsupported). Weight matrix. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **b** (*np.ndarray or T.TensorVariable*) – (Unsupported). Bias vector. If this is a np.ndarray, its values are used to initialise a shared variable for this layer. If it is a T.TensorVariable, it is directly used (weight sharing with the layer which this variable comes from).
- **gamma** – (For batch normalisation) Initialises gamma parameter.
- **mean** – (For batch normalisation) Initialises mean parameter.
- **std** – (For batch normalisation) Initialises std parameter.
- **gradnet_mode** –

class elektronn2.neuromancer.neural.**LRN**(***kwargs*)

Bases: *elektronn2.neuromancer.node_basic.Node*

LRN (Local Response Normalization) layer.

Parameters

- **parent** (Node) – The input node.

- **filter_shape** (*tuple*) –
- **mode** (*str*) – Can be “spatial” or “channel”.
- **alpha** (*float*) –
- **k** (*float*) –
- **beta** (*float*) –
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

elektronn2.neuromancer.neural.**AutoMerge**(*parent1*, *parent2*, *upconv_n_f=None*, *merge_mode='concat'*, *disable_upconv=False*, *upconv_kwargs=None*, *name='merge'*, *print_repr=True*)

Merges two network branches by automatic cropping and upconvolutions.

(Wrapper for *UpConv*, *Crop*, *Concat* and *Add*.)

Tries to automatically align and merge a high-res and a low-res (convolution) output of two branches of a CNN by applying UpConv and Crop to make their shapes and strides compatible. UpConv is used if the low-res parent's strides are at least twice as large as the strides of the high-res parent in any dimension.

The parents are automatically identified as high-res and low-res by their strides. If both parents have the same strides, the concept of high-res and low-res is ignored and this function just crops the larger parent's output until the parents' spatial shapes match and then merges them.

This function can be used to simplify creation of e.g. architectures similar to U-Net (see https://arxiv.org/abs/1505.04597) or skip-connections.

If a ValueError that the shapes cannot be aligned is thrown, you can try changing the filter shapes and pooling factors of the (grand-)parent nodes or add/remove Convolutions and Crops in the preceding branches until the error disappears (of course you should try to keep those changes as minimal as possible).

(This function is an alias for UpConvMerge.)

Parameters

- **parent1** (Node) – First parent to be merged.
- **parent2** (Node) – Second parent to be merged.
- **upconv_n_f** (*int*) – Number of filters for the aligning UpConv for the low-res parent.
- **merge_mode** (*str*) – How the merging should be performed. Available options: ‘concat’ (default): Merge with a Concat node. ‘add’: Merge with an Add node.
- **disable_upconv** (*bool*) – If True, no automatic upconvolutions are performed to match strides.
- **upconv_kwargs** (*dict*) – Additional keyword arguments that are passed to the UpConv constructor if upconvolution is applied.
- **name** (*str*) – Name of the final merge node.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

Returns Concat or Add node (depending on merge_mode) that merges the aligned high-res and low-res outputs.

Return type *Concat* or *Add*

elektronn2.neuromancer.neural.**UpConvMerge**(*parent1*, *parent2*, *upconv_n_f=None*, *merge_mode='concat'*, *disable_upconv=False*, *upconv_kwargs=None*, *name='merge'*, *print_repr=True*)

Merges two network branches by automatic cropping and upconvolutions.

(Wrapper for *UpConv*, *Crop*, *Concat* and *Add*.)

Tries to automatically align and merge a high-res and a low-res (convolution) output of two branches of a CNN by applying UpConv and Crop to make their shapes and strides compatible. UpConv is used if the low-res parent's strides are at least twice as large as the strides of the high-res parent in any dimension.

The parents are automatically identified as high-res and low-res by their strides. If both parents have the same strides, the concept of high-res and low-res is ignored and this function just crops the larger parent's output until the parents' spatial shapes match and then merges them.

This function can be used to simplify creation of e.g. architectures similar to U-Net (see https://arxiv.org/abs/1505.04597) or skip-connections.

If a ValueError that the shapes cannot be aligned is thrown, you can try changing the filter shapes and pooling factors of the (grand-)parent nodes or add/remove Convolutions and Crops in the preceding branches until the error disappears (of course you should try to keep those changes as minimal as possible).

(This function is an alias for UpConvMerge.)

Parameters

- **parent1** (*Node*) – First parent to be merged.
- **parent2** (*Node*) – Second parent to be merged.
- **upconv_n_f** (*int*) – Number of filters for the aligning `UpConv` for the low-res parent.
- **merge_mode** (*str*) – How the merging should be performed. Available options: 'concat' (default): Merge with a `Concat` node. 'add': Merge with an `Add` node.
- **disable_upconv** (*bool*) – If `True`, no automatic upconvolutions are performed to match strides.
- **upconv_kwargs** (*dict*) – Additional keyword arguments that are passed to the `UpConv` constructor if upconvolution is applied.
- **name** (*str*) – Name of the final merge node.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

Returns `Concat` or `Add` node (depending on `merge_mode`) that merges the aligned high-res and low-res outputs.

Return type *Concat* or *Add*

class elektronn2.neuromancer.neural.**Pad**(***kwargs*)

Bases: *elektronn2.neuromancer.node_basic.Node*

Pads the spatial axes of its parent's output.

Parameters

- **parent** (*Node*) – The input node whose output should be padded.
- **pad** (*tuple or list of ints*) – The padding length from either side for each spatial axis
- **value** (*float*) – Value of the padding elements (default: 0.0)
- **name** (*str*) – Node name.

- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

elektronn2.neuromancer.node_basic module

class elektronn2.neuromancer.node_basic.**Node**(*parent*, *name=''*, *print_repr=False*)
Bases: object

Basic node class. All neural network nodes should inherit from Node.

Parameters

- **parent** (*Node or list[Node]*) – The input node(s).
- **name** (*str*) – Given name of the Node, may be an empty string.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

Models are built from the interplay of *Nodes* to form a (directed, acyclic) computational graph.

The **ELEKTRONN2** framework can be seen as an intelligent abstraction level that hides the raw theano-graph and manages the involved symbolic variables. The overall goal is the intuitive, flexible and **easy** creation of complicated graphs.

A Node has one or several inputs, called parent, (unless it is a *source*, i.e. a node where external data is feed into the graph). The inputs are node objects themselves.

Layers automatically keep track of their previous inputs, parameters, computational cost etc. This allows to compile the theano-functions without manually specifying the inputs, outputs and parameters. In the most simple case, any node, which might be part of a more complicated graph, can be called as a function (passing suitable numpy arrays):

```
>>> import elektronn2.neuromancer.utils
>>> inp = neuromancer.Input((batch_size, in_dim))
>>> test_data = elektronn2.neuromancer.utils.as_floatX(np.random.rand(batch_size,␣
↪in_dim))
>>> out = inp(test_data)
>>> np.allclose(out, test_data)
True
```

At the first time the theano function is compiled and cached for re-use in future calls.

Several properties (with respect to the sub-graph the node depends on, or only from the of the node itself) These can also be looked up externally (e.g. required sources, parameter count, computational count).

The theano variable that represents the output of a node is kept in the attribute output. Subsequent Nodes must use this attribute of their inputs to perform their calculation and write the result to their own output (this happens in the method _calc_output, which is hidden because it must be called only internally at initialisation).

A divergence in the computational graph is created by passing the *parent* to several *children* as input:

```
>>> inp = neuromancer.Input((1,10), name='Input_1')
>>> node1 = neuromancer.ApplyFunc(inp, func1)
>>> node2 = neuromancer.ApplyFunc(inp, func2)
```

A convergence in the graph is created by passing several inputs to a node that performs a reduction:

```
>>> out = neuromancer.Concat([node1, node2])
```

Although the node "out" has two immediate inputs, it is detected that the required sources is only a single object:

```
>>> print(out.input_nodes)
Input_1
```

Computations that result in more than a single output for a Node must be broken apart using divergence and individual nodes for the several outputs. Alternatively the function `split` can be used to create two dummy nodes of the output of a previous Node by splitting along the specified axis. Note that possible redundant computations in Nodes are most likely eliminated by the theano graph optimiser.

Instructions for subclassing:

Overriding `__init__`: At the very first the base class' initialiser must be called, which just assigns the names and emtpy default values for attributes. Then node specific initialisations are made e.g. initialisation of shared parameters / weights. Finally the `_finialise_init` method of the base class is automatically called: This evokes the execution of the methods: `_make_output`, `_calc_shape` and `self._calc_comp_cost`. Each of those updates the corresponding attributes. NOTE: if a Node (except for the base `Node`) is subclassed and the derived calls `__init__` of the base Node, this will also call `_finialise_init` exactly right the call to the superclass' `__init__`.

For the graph serialisation and restoration to work, the following conditions must additionally be met:

- The name of of a node's trainable parameter in the parameter dict must be the same as the (optional) keyword used to initialise this parameter in `__init__`; moreover, parameters must not be initialised/shared from positional arguments.
- When serialising only the current state of parameters is kept, parameter value arrays given for initialisation are never kept.

Depending on the purpose of the node, the latter methods and others (e.g. `__repr__`) must be overridden. The default behaviour of the base class is: output = input, outputs shape = input shape, computational cost = tensor size (!) ...

`all_children`

`all_computational_cost`

`all_extra_updates`
List of the parameters updates of all parent nodes. They are tuples.

`all_nontrainable_params`
Dict of the trainable parameters (weights) of all parent nodes. They are theano shared variables.

`all_params`
Dict of the all parameters of all parent nodes. They are theano variable

`all_params_count`
Count of all trainable parameters in the entire sub-graph used to compute the output of this node

`all_parents`
List all nodes that are involved in the computation of the output of this node (incl. `self`). The list contains no duplicates. The return is a dict, the keys of which are the layers, the values are just all `True`

`all_trainable_params`
Dict of the trainable parameters (weights) of all parent nodes. They are theano shared variables.

`feature_names`

`get_debug_outputs`(**args*)

`get_param_values`(*skip_const=False*)
Returns a dict that maps the values of the params. (such that they can be saved to disk)

Parameters **`skip_const`** (*bool*) – whether to exclude constant parameters.

Returns Dict that maps the values of the params.

Return type dict

input_nodes
Contains the all parent nodes that are sources, i.e. inputs that are required to compute the result of this node.

input_tensors
The same as `input_nodes` but contains the theano tensor variables instead of the node objects. May be used as input to compile theano functions.

last_exec_time
Last function execution time in seconds.

local_exec_time

measure_exectime(*n_samples=5*, *n_warmup=4*, *print_info=True*, *local=True*, *nonegative=True*)
Measure how much time the node needs for its calculation (in milliseconds).

Parameters

- **n_samples** (*int*) – Number of independent measurements of which the median is taken.
- **n_warmup** (*int*) – Number of warm-up runs before each measurement (not taken into account for median calculation).
- **print_info** (*bool*) – If True, print detailed info about measurements while running.
- **local** (*bool*) – Only compute exec time for this node by subtracting its parents' times.
- **nonegative** (*bool*) – Do not return exec times smaller than zero.

Returns median of execution time measurements.

Return type np.float

param_count
Count of trainable parameters in this node

plot_theano_graph(*outfile=None*, *compiled=True*, ***kwargs*)
Plot the execution graph of this Node's Theano function to a file.

If "outfile" is not specified, the plot is saved in "/tmp/<user>_<name>.png"

Parameters

- **outfile** (*str or None*) – File name for saving the plot.
- **compiled** (*bool*) – If True, the function is compiled before plotting.
- **kwargs** – kwargs (plotting options) that get directly passed to theano.printing.pydotprint().

predict_dense(*raw_img*, *as_uint8=False*, *pad_raw=False*)
Core function that performs the inference

Parameters

- **raw_img** (*np.ndarray*) – raw data in the format (ch, (z,) y, x)
- **as_uint8** (*Bool*) – Return class probabilites as uint8 image (scaled between 0 and 255!)

- **pad_raw** (*Bool*) – Whether to apply padding (by mirroring) to the raw input image in order to get predictions on the full image domain.

Returns Predictions.

Return type np.ndarray

set_param_values(*value_dict*, *skip_const=False*)
Sets new values for non constant parameters.

Parameters

- **value_dict** (*dict*) – A dict that maps values by parameter name.
- **skip_const** (*bool*) – if dict also maps values for constants, these can be skipped, otherwise an exception is raised.

test_run(*on_shape_mismatch='warn'*, *debug_outputs=False*)
Test execution of this node with random (but correctly shaped) data.

Parameters **on_shape_mismatch** (*str*) – If this is "warn", a warning is emitted if there is a mismatch between expected and calculated output shapes.

Returns

Return type Debug output of the Theano function.

total_exec_time

class elektronn2.neuromancer.node_basic.**Input**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Input node

Parameters

- **shape** (*list/tuple of int*) – shape of input array, unspecified shapes are `None`
- **tags** (*list/tuple of strings or comma-separated string*) – tags indicate which purpose the dimensions of the tensor serve. They are sometimes used to decide about reshapes. The maximal tensor has tags: "r, b, f, z, y, x, s" which denote: * r: perform recurrence along this axis * b: batch size * f: features, filters, channels * z: convolution no. 3 (slower than 1,2) * y: convolution no. 1 * x: convolution no. 2 * s: samples of the same instance (over which expectations are calculated) Unused axes are to be removed from this list, but `b` and `f` must always remain. To avoid bad memory layout, the order must not be changed. For less than 3 convolutions conv1,conv2 are preferred for performance reasons. Note that CNNs can mix nodes with 2d and 3d convolutions as 2d is a special case of 3d with filter size 1 on the respective axis. In this case conv3 should be used for the axis with smallest filter size.
- **strides** –
- **fov** –
- **dtype** (*str*) – corresponding to numpy dtype (e.g., 'int64'). Default is floatX from theano config
- **hardcoded_shape** –
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

elektronn2.neuromancer.node_basic.**Input_like**(*ref*, *dtype=None*, *name='input'*, *print_repr=True*, *override_f=False*, *hardcoded_shape=False*)

class elektronn2.neuromancer.node_basic.**Concat**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Node to concatenate the inputs. The inputs must have the same shape, except in the dimension corresponding to `axis`. This is not checked as shapes might be unspecified prior to compilation!

Parameters

- **parent_nodes** (*list of Node*) – Inputs to be concatenated.
- **axis** (*int*) – Join axis.
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.node_basic.**ApplyFunc**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Apply function to the input. If the function changes the output shape, this node should not be used.

Parameters

- **parent** (Node) – Input (single).
- **functor** (*function*) – Function that acts on theano variables (e.g. `theano.tensor.tanh`).
- **args** (*tuple*) – Arguments passed to `functor` **after** the input.
- **kwargs** (*dict*) – kwargs for `functor`.

class elektronn2.neuromancer.node_basic.**FromTensor**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Dummy node to be used in the split-function.

Parameters

- **tensor** (*T.Tensor*) –
- **tensor_shape** –
- **tensor_parent** (*T.Tensor*) –
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

elektronn2.neuromancer.node_basic.**split**(*node*, *axis='f'*, *index=None*, *n_out=None*, *strip_singleton_dims=False*, *name='split'*)

class elektronn2.neuromancer.node_basic.**GenericInput**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Input node for arbitrary oject.

Parameters

- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.node_basic.**ValueNode**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

(Optionally) trainable value node

Parameters

- **shape** (*list/tuple of int*) – shape of input array, unspecified shapes are `None`
- **tags** (*list/tuple of strings or comma-separated string*) – tags indicate which purpose the dimensions of the tensor serve. They are sometimes used to decide about reshapes. The maximal tensor has tags: "r, b, f, z, y, x, s" which denote:
  - r: perform recurrence along this axis
  - b: batch size
  - f: features, filters, channels
  - z: convolution no. 3 (slower than 1,2)
  - y: convolution no. 1
  - x: convolution no. 2
  - s: samples of the same instance (over which expectations are calculated)

  Unused axes are to be removed from this list, but `b` and `f` must always remain. To avoid bad memory layout, the order must not be changed. For less than 3 convolutions conv1,conv2 are preferred for performance reasons. Note that CNNs can mix nodes with 2d and 3d convolutions as 2d is a special case of 3d with filter size 1 on the respective axis. In this case conv3 should be used for the axis with smallest filter size.
- **strides** –
- **fov** –
- **dtype** (*str*) – corresponding to numpy dtype (e.g., 'int64'). Default is floatX from theano config
- **apply_train** (*bool*) –
- **value** –
- **init_kwargs** (*dict*) –
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

get_value()

class elektronn2.neuromancer.node_basic.**MultMerge**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Node to concatenate the inputs. The inputs must have the same shape, except in the dimension corresponding to `axis`. This is not checked as shapes might be unspecified prior to compilation!

Parameters

- **n1** (Node) – First input node.
- **n2** (Node) – Second input node.
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.node_basic.**InitialState_like**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Parameters

- **parent** –

- **override_f** –
- **dtype** –
- **name** –
- **print_repr** –
- **init_kwargs** –

get_value()

class elektronn2.neuromancer.node_basic.**Add**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Add two nodes using `theano.tensor.add`.

Parameters

- **n1** (Node) – First input node.
- **n2** (Node) – Second input node.
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

elektronn2.neuromancer.optimiser module

class elektronn2.neuromancer.optimiser.**AdaDelta**(*inputs*, *loss*, *grads*, *params*, *extra_updates*, *additional_outputs=None*)
Bases: *elektronn2.neuromancer.optimiser.Optimiser*

AdaDelta optimiser (See https://arxiv.org/abs/1212.5701).

Like AdaGrad, but accumulate squared only over window The delta part is some diagonal hessian approximation. Claims to be robust against sudden large gradients because then the denominator explodes, but this explosion is persistent for a while... (and this argumentation is true for any method accumulating squared grads).

repair()

class elektronn2.neuromancer.optimiser.**AdaGrad**(*inputs*, *loss*, *grads*, *params*, *extra_updates*, *additional_outputs=None*)
Bases: *elektronn2.neuromancer.optimiser.Optimiser*

AdaGrad optimiser (See http://jmlr.org/papers/v12/duchi11a.html).

Tries to favor making faster progress on parameters with usually small gradients (but does somehow ignore their actual direction, i.e. a parameter which has a lot of small gradients in the same direction and one that has many small gradients in opposite directions have both a high LR !

repair()

class elektronn2.neuromancer.optimiser.**Adam**(*inputs*, *loss*, *grads*, *params*, *extra_updates*, *additional_outputs=None*)
Bases: *elektronn2.neuromancer.optimiser.Optimiser*

Adam optimiser (See https://arxiv.org/abs/1412.6980v9).

Like AdaGrad with windowed squared_accum and with momentum and a bias for the initial phase (t). The normalisation of Adam and AdaGrad (and RMSProp) does not damp but exaggerates sudden steep gradients (their squared_accum is small and their current grad is large).

repair()

class elektronn2.neuromancer.optimiser.**CG**(*inputs*, *loss*, *grads*, *params*, *additional_outputs*)
Bases: *elektronn2.neuromancer.optimiser.Optimiser*

class elektronn2.neuromancer.optimiser.**Optimiser**(*inputs*, *loss*, *grads*, *params*, *additional_outputs*)
Bases: object

alloc_shared_grads(*name_suffix='_lg'*, *init_val=0.0*)
Returns new shared variables matching the shape of params/gradients

clear_last_dir(*last_dir=None*)

get_rotational_updates()

global_lr = lr

global_mom = mom

global_weight_decay = weight_decay

repair()

set_opt_meta_params(*value_dict*)
Update the meta-parameters via value dictionary

classmethod setlr(*val*)
Set learning rate (global to all optimisers)

classmethod setmom(*val*)
Set momentum parameter (global to all optimisers)

classmethod setwd(*val*)
Set weight decay parameter (global to all optimisers)

class elektronn2.neuromancer.optimiser.**SGD**(*inputs*, *loss*, *grads*, *params*, *extra_updates*, *additional_outputs=None*)
Bases: *elektronn2.neuromancer.optimiser.Optimiser*

SGD optimiser (See https://en.wikipedia.org/wiki/Stochastic_gradient_descent).

elektronn2.neuromancer.variables module

class elektronn2.neuromancer.variables.**VariableParam**(*value=None*, *name=None*, *apply_train=True*, *apply_reg=True*, *dtype=None*, *strict=False*, *allow_downcast=None*, *borrow=False*, *broadcastable=None*)
Bases: theano.tensor.sharedvar.TensorSharedVariable

Extension of theano TensorSharedVariable. Additional features are described by the parameters, otherwise identical

Parameters

- **value** –
- **name** (*str*) –
- **flag** (*apply_train*) – whether to apply regularisation (e.g. L2) on this param

- **flag** – whether to train this parameter (as opposed to a meta-parameter or a parameter that is kept const. during a training phase)
- **dtype** –
- **strict** (*bool*) –
- **allow_downcast** (*bool*) –
- **borrow** (*bool*) –
- **broadcastable** –

clone()

Return a new Variable like self.

Returns A new Variable instance (or subclass instance) with no owner or index.

Return type Variable instance

Notes

Tags are copied to the returned instance.

Name is copied to the returned instance.

updates

class elektronn2.neuromancer.variables.**VariableWeight**(*shape=None*, *init_kwargs=None*, *value=None*, *name=None*, *apply_train=True*, *apply_reg=True*, *dtype=None*, *strict=False*, *allow_downcast=None*, *borrow=False*, *broadcastable=None*)

Bases: *elektronn2.neuromancer.variables.VariableParam*

set_value(*new_value*, *borrow=False*)

Set the non-symbolic value associated with this SharedVariable.

Parameters

- **borrow** (*bool*) – True to use the new_value directly, potentially creating problems related to aliased memory.
- **to this value will be visible to all functions using** (*Changes*) –
- **SharedVariable.** (*this*) –

class elektronn2.neuromancer.variables.**ConstantParam**(*value*, *name=None*, *dtype=None*, *make_singletons_broadcastable=True*)

Bases: theano.tensor.var.TensorConstant

Identical to theano VariableParam except that there are two two addition attributes apply_train and *apply_reg'*, which are both false. This is just to tell ELEKTRONN2 that this parameter is to be exempted from training. Obviously the set_value method raises an exception because this is a real constant. Constants are faster in the theano graph.

clone()
We clone this object, but we don't clone the data to lower memory requirement. We suppose that the data will never change.

get_value(*borrow=False*)

set_value(*new_value*, *borrow=False*)

updates

elektronn2.neuromancer.variables.**initweights**(*shape*, *dtype='float64'*, *scale='glorot'*, *mode='normal'*, *pool=None*, *spatial_axes=None*)

elektronn2.neuromancer.various module

class elektronn2.neuromancer.various.**GaussianRV**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Parameters

- **mu** (*node*) – Mean of the Gaussian density
- **sig** (*node*) – Sigma of the Gaussian density
- **n_samples** (*int*) – Number of samples to be drawn per instance. Special case '0': draw 1 sample but don't' increase rank of tensor!
- **output is a sample from separable Gaussians of given mean and** (*The*) –
- **(but this operation is still differentiable, due to the** (*sigma*) –
- **trick")** (*"re-parameterisation*) –
- **output dimension mu.ndim+1 because the samples are accumulated along** (*The*) –
- **new axis right of 'b' (batch)** (*a*) –

make_priorlayer()
Creates a new Layer that calculates the Auto-Encoding-Variation-Bayes (AEVB) prior corresponding to this Layer.

elektronn2.neuromancer.various.**SkelLoss**(*pred*, *loss_kwargs*, *skel=None*, *name='skel_loss'*, *print_repr=True*)

class elektronn2.neuromancer.various.**SkelPrior**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

pred must be a vector of shape [(1,b),(3,f)] or [(3,f)] i.e. only batch_size=1 is supported.

Parameters

- **pred** –
- **target_length** –
- **prior_n** –
- **prior_posz** –
- **prior_z** –

- **prior_xy** –
- **name** –
- **print_repr** –

elektronn2.neuromancer.various.**Scan**(*step_result*, *in_memory*, *out_memory=None*, *in_iterate=None*, *in_iterate_0=None*, *n_steps=None*, *unroll_scan=True*, *last_only=False*, *name='scan'*, *print_repr=True*)

Parameters

- **step_result** (*node/list(nodes)*) – nodes that represent results of step function
- **in_memory** (*node/list(nodes)*) – nodes that indicate at which place in the computational graph the memory is feed back into the step function. If `out_memory` is not specified this must contain a node for *every* node in `step_result` because then the whole result will be fed back.
- **out_memory** (*node/list(nodes)*) – (optional) must be subset of `step_result` and of same length as `in_memory`, tells which nodes of the result are fed back to `in_memory`. If `None`, all are fed back.
- **in_iterate** (*node/list(nodes)*) – nodes with a leading `'r'` axis to be iterated over (e.g. time series of shape [(30,r),(100,b),(50,f)]). In every step a slice from the first axis is consumed.
- **in_iterate_0** (*node/list(nodes)*) – nodes that consume a single slice of the `in_iterate` nodes. Part of "the inner function" of the scan loop in contrast to `in_iterate`
- **n_steps** (*int*) –
- **unroll_scan** (*bool*) –
- **last_only** (*bool*) –
- **name** (*str*) –
- **print_repr** (*bool*) –

Returns

- A node for every node in `step_result` which either contains the last
- state or the series of states - then it has a leading `'r'` axis.

elektronn2.neuromancer.various.**SkelGetBatch**(*skel*, *aux*, *img_sh*, *t_img_sh*, *t_grid_sh*, *t_node_sh*, *get_batch_kwargs*, *scale_strenght=None*, *name='skel_batch'*)

class elektronn2.neuromancer.various.**SkelLossRec**(***kwargs*)

Bases: *elektronn2.neuromancer.node_basic.Node*

pred must be a vector of shape [(1,b),(3,f)] or [(3,f)] i.e. only batch_size=1 is supported.

Parameters

- **pred** –
- **skel** –
- **loss_kwargs** –
- **name** –

- **print_repr** –

class elektronn2.neuromancer.various.**Reshape**(***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Reshape node.

Parameters

- **parent** –
- **shape** –
- **tags** –
- **strides** –
- **fov** –
- **name** –
- **print_repr** –

elektronn2.neuromancer.various.**SkelGridUpdate**(*grid*, *skel*, *radius*, *bio*, *name='skelgridupdate'*, *print_repr=True*)

3.2 elektronn2.training package

3.2.1 Submodules

elektronn2.training.parallelisation module

class elektronn2.training.parallelisation.**BackgroundProc**(*target*, *dtypes=None*, *shapes=None*, *n_proc=1*, *target_args=()*, *target_kwargs={}*, *profile=False*)
Bases: elektronn2.training.parallelisation.SharedMem

Data structure to manage repeated background tasks by reusing a fixed number of *initially* created background process with the same arguments at every time. (E.g. retrieving an augmented batch) Remember to call BackgroundProc.shutdown after use to avoid zombie process and RAM clutter.

Parameters

- **dtypes** – list of dtypes of the target return values
- **shapes** – list of shapes of the target return values
- **n_proc** (*int*) – number of background procs to use
- **target** (*callable*) – target function for background proc. Can even be a method of an object, if object data is read-only (then data will not be copied in RAM and the new process is lean). If several procs use random modules, new seeds must be created inside target because they have the same random state at the beginning.
- **target_args** (*tuple*) – Proc args (constant)
- **target_kwargs** (*dict*) – Proc kwargs (constant)
- **profile** (*Bool*) – Whether to print timing results in to stdout

Examples

Use case to retrieve batches from a data structure `D`:

```
>>> data, label = D.getbatch(2, strided=False, flip=True, grey_augment_
↪channels=[0])
>>> kwargs = {'strided': False, 'flip': True, 'grey_augment_channels': [0]}
>>> bg = BackgroundProc([np.float32, np.int16], [data.shape,label.shape],
↪                    D.getbatch, n_proc=2, target_args=(2,),
↪        target_kwargs=kwargs, profile=False)
>>> for i in range(100):
>>>     data, label = bg.get()
```

get(*timeout=False*)
This gets the next result from a background process and blocks until the corresponding proc has finished.

reset()
Should be called after an exception (e.g. by pressing ctrl+c) was raised.

shutdown()
Must be called to free memory if the background tasks are no longer needed

class elektronn2.training.parallelisation.**SharedQ**(*n_proc=0*, *profile=False*)
Bases: elektronn2.training.parallelisation.SharedMem

FIFO Queue to process np.ndarrays in the background (also pre-loading of data from disk)

procs must accept list of `mp.Array` and make items `np.ndarray` using `SharedQ.shm2ndarray`, for this the shapes are required as too. The target requires the signature:

```
>>> target(mp_arrays, shapes, *args, **kwargs)
```

Whereas mp_array and shape are *automatically* added internally

All parameters are optional:

Parameters

- **n_proc** (*int*) – If larger than 0, a message is printed if to few processes are running
- **profile** (*Bool*) – Whether to print timing results in terminal

Examples

Automatic use:

```
>>> Q = SharedQ(n_proc=2)
>>> Q.startproc(target=, shape= args=, kwargs=)
>>> Q.startproc(target=, shape= args=, kwargs=)
>>> for i in range(5):
>>>     Q.startproc(target=, shape= args=, kwargs=)
>>>     item = Q.get() # starts as many new jobs as to maintain n_proc
>>>     dosomethingelse(item) # processes work in background to pre-fetch data␣
↪for next iteration
```

get()
This gets the first results in the queue and blocks until the corresponding proc has finished. If a n_proc value is defined this then new procs must be started *before* to avoid a warning message.

startproc(*dtypes*, *shapes*, *target*, *target_args=()*, *target_kwargs={}*)
Starts a new process

procs must accept list of `mp.Array` and make items `np.ndarray` using `SharedQ.shm2ndarray`, or this the shapes are required as too. The target requires the signature:

```
target(mp_arrays, shapes, *args, **kwargs)
```

Whereas mp_array and shape are *automatically* added internally

exception elektronn2.training.parallelisation.**TimeoutError**(**args*, ***kwargs*)
Bases: `exceptions.RuntimeError`

elektronn2.training.trainer module

class elektronn2.training.trainer.**Trainer**(*exp_config*)
Bases: `object`

debug_getcnnbatch()
Executes `getbatch` but with un-strided labels and always returning info. The first batch example is plotted and the whole batch is returned for inspection.

predict_and_write(*pred_node*, *raw_img*, *number=0*, *export_class='all'*, *block_name=''*, *z_thick=5*)
Predict and and save a slice as preview image

Parameters

- **raw_img** (*np.ndarray*) – raw data in the format (ch, x, y, z)
- **number** (*int/float*) – consecutive number for the save name (i.e. hours, iterations etc.)
- **export_class** (*str or int*) – 'all' writes images of all classes, otherwise only the class with index `export_class` (int) is saved.
- **block_name** (*str*) – Name/number to distinguish different raw_imges

preview_slice(*number=0*, *export_class='all'*, *max_z_pred=5*)
Predict and and save a data from a separately loaded file as preview

Parameters

- **number** (*int/float*) – consecutive number for the save name (i.e. hours, iterations etc.)
- **export_class** (*str or int*) – 'all' writes images of all classes, otherwise only the class with index `export_class` (int) is saved.
- **max_z_pred** (*int*) – approximate maximal number of z-slices to produce (depends on CNN architecture)

preview_slice_from_traindata(*cube_i=0*, *off=(0, 0, 0)*, *sh=(10, 400, 400)*, *number=0*, *export_class='all'*)
Predict and and save a selected slice from the training data as preview

Parameters

- **cube_i** (*int*) – index of source cube in CNNData
- **off** (*3-tuple of int*) – start index of slice to cut from cube (z,y,x)
- **sh** (*3-tuple of int*) – shape of cube to cut (z,y,x)

- **number** (*int*) – consecutive number for the save name (i.e. hours, iterations etc.)
- **export_class** (*str or int*) – 'all' writes images of all classes, otherwise only the class with index `export_class` (int) is saved.

run()

test_model(*data_source*)
Computes Loss and error/accuracy on batch with `monitor_batch_size`

Parameters **data_source** (*string*) – 'train' or 'valid'

Returns

Return type Loss, error

class elektronn2.training.trainer.**TracingTrainer**(*exp_config*)
Bases: *elektronn2.training.trainer.Trainer*

debug_getcnnbatch(*extended=False*)
Executes `getbatch` but with un-strided labels and always returning info. The first batch example is plotted and the whole batch is returned for inspection.

run()

static save_batch(*img*, *lab*, *k*, *lab_img=None*)

test_model(*data_source*)
Computes Loss and error/accuracy on batch with `monitor_batch_size`

Parameters **data_source** (*string*) – 'train' or 'valid'

Returns

Return type Loss, error

class elektronn2.training.trainer.**TracingTrainerRNN**(*exp_config*)
Bases: *elektronn2.training.trainer.TracingTrainer*

run()

test_model(*data_source*)
Computes Loss and error/accuracy on batch with `monitor_batch_size`

Parameters **data_source** (*string*) – 'train' or 'valid'

Returns

Return type Loss, error

elektronn2.training.trainutils module

class elektronn2.training.trainutils.**ExperimentConfig**(*exp_file*, *host_script_file=None*, *use_existing_dir=False*)
Bases: `object`

check_config()

classmethod levenshtein(*s1*, *s2*)
Computes Levenshtein-distance between `s1` and `s2` strings Taken from: http://en.wikibooks.org/wiki/Algorithm_Implementation/ Strings/Levenshtein_distance#Python

make_dir()
Saves all python files into the folder specified by `self.save_path` Also changes working directory to the `save_path` directory

read_user_config()

class elektronn2.training.trainutils.**HistoryTracker**
Bases: object

load(*file_name*)

plot(*save_name=None*, *autoscale=True*, *close=True*)

register_debug_output_names(*names*)

save(*save_name*)

update_debug_outputs(*vals*)

update_history(*vals*)

update_regression(*pred*, *target*)

update_timeline(*vals*)

class elektronn2.training.trainutils.**Schedule**(***kwargs*)
Bases: object

Create a schedule for parameter or property

Examples

```
>>> lr_schedule = Schedule(dec=0.95) # decay by factor 0.95 every 1000 steps (i.e.
↪ decreasing by 5%)
>>> wd_schedule = Schedule(lindec=[4000, 0.001]) # from 0.001 to 0 in 400 steps
>>> mom_schedule = Schedule(updates=[(500,0.8), (1000,0.7), (1500,0.9), (2000, 0.
↪2)])
>>> dropout_schedule = Schedule(updates=[(1000,[0.2, 0.2])]) # set rates per Layer
```

bind_variable(*variable_param=None*, *obj=None*, *prop_name=None*)

update(*iteration*)

elektronn2.training.trainutils.**binary_nll**(*pred*, *gt*)

elektronn2.training.trainutils.**confusion_table**(*labs*, *preds*)

Gives all counts of binary classifications situations:

labs correct labels (-1 for ignore)

preds 0 for negative 1 for positive (class probabilities must be thresholded first)

Returns count of: (true positive, true negative, false positive, false negative)

elektronn2.training.trainutils.**error_hist**(*gt*, *preds*, *save_name*, *thresh=0.42*)
preds: predicted probability of class '1' Saves plot to file

elektronn2.training.trainutils.**eval_thresh**(*args*)
Calculates various performance measures at certain threshold :param args: thresh, labs, preds :return: tpr, fpr, precision, recall, bal_accur, accur, f1

elektronn2.training.trainutils.**evaluate**(*gt*, *preds*, *save_name*, *thresh=None*, *n_proc=None*)

Evaluate prediction w.r.t. GT Saves plot to file :param save_name: :param gt: :param preds: from 0.0 to 1.0 :param thresh: if thresh is given (e.g. from tuning on validation set) some performance measures are shown at this threshold :return: perf, roc-area, threshs

elektronn2.training.trainutils.**evaluate_model_binary**(*model*, *name*, *data=None*, *valid_d=None*, *valid_l=None*, *train_d=None*, *train_l=None*, *n_proc=2*, *betaloss=False*, *fudgeysoft=False*)

elektronn2.training.trainutils.**find_nearest**(*array*, *value*)

elektronn2.training.trainutils.**loadhistorytracker**(*file_name*)

elektronn2.training.trainutils.**performance_measure**(*tp*, *tn*, *fp*, *fn*)

For output of confusion table gives various perfomance performance_measures:

return tpr, fpr, precision, recall, balanced accuracy, accuracy, f1-score

elektronn2.training.trainutils.**rescale_fudge**(*pred*, *fudge=0.15*)

elektronn2.training.trainutils.**roc_area**(*tpr*, *fpr*)

Integrate ROC curve:

data (tpr, fpr)

return area

elektronn2.training.trainutils.**user_input**(*local_vars*)

3.3 elektronn2.data package

3.3.1 Submodules

elektronn2.data.cnndata module

class elektronn2.data.cnndata.**AgentData**(*input_node*, *side_target_node*, *path_prefix=None*, *raw_files=None*, *skel_files=None*, *vec_files=None*, *valid_skels=None*, *target_vec_ix=None*, *target_discrete_ix=None*, *abs_offset=None*, *aniso_factor=2*)

Bases: *elektronn2.data.cnndata.BatchCreatorImage*

Load raw_cube, vec_prob_obj_cube and skelfiles + rel.offset

get_newslice(*position_l*, *direction_il*, *batch_size=1*, *source='train'*, *aniso=True*, *z_shift=0*, *gamma=0*, *grey_augment_channels=None*, *r_max_scale=0.9*, *tracing_dir_prior_c=0.5*, *force_dense=False*, *flatfield_p=0.001*, *scale=1.0*, *last_ch_max_interp=False*)

getbatch(*batch_size=1*, *source='train'*, *aniso=True*, *z_shift=0*, *gamma=0*, *grey_augment_channels=None*, *r_max_scale=0.9*, *tracing_dir_prior_c=0.5*, *force_dense=False*, *flatfield_p=0.001*)

Prepares a batch by randomly sampling, shifting and augmenting patches from the data

Parameters

- **batch_size** (*int*) – Number of examples in batch (for CNNs often just 1)
- **source** (*str*) – Data set to draw data from: 'train'/'valid'
- **grey_augment_channels** (*list*) – List of channel indices to apply grey-value augmentation to
- **warp** (*bool or float*) – Whether warping/distortion augmentations are applied to examples (slow –> use multiprocessing). If this is a float number, warping is applied to this fraction of examples e.g. 0.5 –> every other example.
- **warp_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.transformations.get_warped_slice()
- **ignore_thresh** (*float*) – If the fraction of negative targets in an example patch exceeds this threshold, this example is discarded (Negative targets are ignored for training [but could be used for unsupervised target propagation]).
- **force_dense** (*bool*) – If True the targets are *not* sub-sampled according to the CNN output strides. Dense targets requires MFP in the CNN!
- **affinities** –
- **nhood_targets** –
- **ret_ll_mask** (*bool*) – If True additional information for reach batch example is returned. Currently implemented are two ll_mask arrays to indicate the targeting mode. The first dimension of those arrays is the batch_size!
- **nga_blur_noise_probability** (*bool or float*) – Probability of applying a Gaussian filter and noise to the input data. The value must be a bool or be within the range [0.0, 1.0] Default: False (disabled)
- **blurry_blobs_probability** (*bool or float*) – Probability of augmenting the input data with blurry "blobs". "Blobs" mean random cubes across the input data. The region within a cube is blurred with Gaussian smoothing. The level of smoothing and the position of each cube are random. The value must be a bool or be within the range [0.0, 1.0] Default: False (disabled)
- **blurry_blobs_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.blurry_blobs(). Effective if setting blurry_blobs_probability > 0.
- **noisy_random_erasing_probability** (*float*) – Probability of augmenting the input data with noise blobs. The blob region is filled with random noise values between 0 and 255.
- **noisy_random_erasing_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.noisy_random_erasing(). Effective if setting noisy_random_erasing_probability > 0.
- **uniform_random_erasing_probability** (*float*) – Probability of augmenting the input data with uniform blobs. The blob region is filled with one value.
- **uniform_random_erasing_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.uniform_random_erasing(). Effective if uniform_random_erasing_probability > 0.

Returns

- **data** (*np.ndarray*) – [bs, ch, x, y] or [bs, ch, z, y, x] for 2D and 3D CNNS

- **target** (*np.ndarray*) – [bs, ch, x, y] or [bs, ch, z, y, x]
- **ll_mask1** (*np.ndarray*) – (optional) [bs, n_target]
- **ll_mask2** (*np.ndarray*) – (optional) [bs, n_target]

getskel(*source*)
Draw an example skeleton according to sampling weight on training data, or randomly on valid data

load_data()

Parameters

- **d_path/l_path** (*string*) – Directories to load data from
- **d_files/l_files** (*list*) – List of data/target files in <path> directory (must be in the same order!). Each list element is a tuple in the form **(<Name of h5-file>, <Key of h5-dataset>)**
- **cube_prios** (*list*) – (not normalised) list of sampling weights to draw examples from the respective cubes. If None the cube sizes are taken as priorities.
- **valid_cubes** (*list*) – List of indices for cubes (from the file-lists) to use as validation data and exclude from training, may be empty list to skip performance estimation on validation data.

read_files()
Image files on disk are expected to be in order (ch,x,y,z) or (x,y,z) But image stacks are returned as (z,ch,x,y) and target as (z,x,y,) irrespective of the order in the file. If the image files have no channel this dimension is extended to a singleton dimension.

class elektronn2.data.cnndata.**BatchCreatorImage**(*input_node*, *target_node=None*, *d_path=None*, *l_path=None*, *d_files=None*, *l_files=None*, *cube_prios=None*, *valid_cubes=None*, *border_mode='crop'*, *aniso_factor=2*, *target_vec_ix=None*, *target_discrete_ix=None*, *h5stream=False*, *zxy=True*)

Bases: object

check_files()
Check if file paths in the network config are available.

getbatch(*batch_size=1*, *source='train'*, *grey_augment_channels=None*, *warp=False*, *warp_args=None*, *ignore_thresh=False*, *force_dense=False*, *affinities=False*, *nhood_targets=False*, *ret_ll_mask=False*, *nga_blur_noise_probability=False*, *blurry_blobs_probability=False*, *blurry_blobs_args=None*, *noisy_random_erasing_probability=0*, *noisy_random_erasing_args=None*, *uniform_random_erasing_probability=0*, *uniform_random_erasing_args=None*)
Prepares a batch by randomly sampling, shifting and augmenting patches from the data

Parameters

- **batch_size** (*int*) – Number of examples in batch (for CNNs often just 1)
- **source** (*str*) – Data set to draw data from: 'train'/'valid'
- **grey_augment_channels** (*list*) – List of channel indices to apply grey-value augmentation to

- **warp** (*bool or float*) – Whether warping/distortion augmentations are applied to examples (slow –> use multiprocessing). If this is a float number, warping is applied to this fraction of examples e.g. 0.5 –> every other example.
- **warp_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.transformations.get_warped_slice()
- **ignore_thresh** (*float*) – If the fraction of negative targets in an example patch exceeds this threshold, this example is discarded (Negative targets are ignored for training [but could be used for unsupervised target propagation]).
- **force_dense** (*bool*) – If True the targets are *not* sub-sampled according to the CNN output strides. Dense targets requires MFP in the CNN!
- **affinities** –
- **nhood_targets** –
- **ret_ll_mask** (*bool*) – If True additional information for reach batch example is returned. Currently implemented are two ll_mask arrays to indicate the targeting mode. The first dimension of those arrays is the batch_size!
- **nga_blur_noise_probability** (*bool or float*) – Probability of applying a Gaussian filter and noise to the input data. The value must be a bool or be within the range [0.0, 1.0] Default: False (disabled)
- **blurry_blobs_probability** (*bool or float*) – Probability of augmenting the input data with blurry "blobs". "Blobs" mean random cubes across the input data. The region within a cube is blurred with Gaussian smoothing. The level of smoothing and the position of each cube are random. The value must be a bool or be within the range [0.0, 1.0] Default: False (disabled)
- **blurry_blobs_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.blurry_blobs(). Effective if setting blurry_blobs_probability > 0.
- **noisy_random_erasing_probability** (*float*) – Probability of augmenting the input data with noise blobs. The blob region is filled with random noise values between 0 and 255.
- **noisy_random_erasing_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.noisy_random_erasing(). Effective if setting noisy_random_erasing_probability > 0.
- **uniform_random_erasing_probability** (*float*) – Probability of augmenting the input data with uniform blobs. The blob region is filled with one value.
- **uniform_random_erasing_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.uniform_random_erasing(). Effective if uniform_random_erasing_probability > 0.

Returns

- **data** (*np.ndarray*) – [bs, ch, x, y] or [bs, ch, z, y, x] for 2D and 3D CNNS
- **target** (*np.ndarray*) – [bs, ch, x, y] or [bs, ch, z, y, x]
- **ll_mask1** (*np.ndarray*) – (optional) [bs, n_target]
- **ll_mask2** (*np.ndarray*) – (optional) [bs, n_target]

load_data()

Parameters

- **d_path/l_path** (*string*) – Directories to load data from
- **d_files/l_files** (*list*) – List of data/target files in <path> directory (must be in the same order!). Each list element is a tuple in the form **(<Name of h5-file>, <Key of h5-dataset>)**
- **cube_prios** (*list*) – (not normalised) list of sampling weights to draw examples from the respective cubes. If None the cube sizes are taken as priorities.
- **valid_cubes** (*list*) – List of indices for cubes (from the file-lists) to use as validation data and exclude from training, may be empty list to skip performance estimation on validation data.

read_files()

Image files on disk are expected to be in order (ch,x,y,z) or (x,y,z) But image stacks are returned as (z,ch,x,y) and target as (z,x,y,) irrespective of the order in the file. If the image files have no channel this dimension is extended to a singleton dimension.

warp_cut(*img*, *target*, *warp*, *warp_params*)

(Wraps *elektronn2.data.transformations.get_warped_slice()*)

Cuts a warped slice out of the input and target arrays. The same random warping transformation is each applied to both input and target.

Warping is randomly applied with the probability defined by the `warp` parameter (see below).

Parameters

- **img** (*np.ndarray*) – Input image
- **target** (*np.ndarray*) – Target image
- **warp** (*float or bool*) – False/True disable/enable warping completely. If `warp` is a float, it is used as the ratio of inputs that should be warped. E.g. 0.5 means approx. every second call to this function actually applies warping to the image-target pair.
- **warp_params** (*dict*) – kwargs that are passed through to *elektronn2.data.transformations.get_warped_slice()*. Can be empty.

Returns

- **d** (*np.ndarray*) – (Warped) input image slice
- **t** (*np.ndarray*) – (Warped) target slice

warp_stats

class elektronn2.data.cnndata.**GridData**(**args*, ***kwargs*)

Bases: *elektronn2.data.cnndata.AgentData*

getbatch(***get_batch_kwargs*)

Prepares a batch by randomly sampling, shifting and augmenting patches from the data

Parameters

- **batch_size** (*int*) – Number of examples in batch (for CNNs often just 1)
- **source** (*str*) – Data set to draw data from: 'train'/'valid'
- **grey_augment_channels** (*list*) – List of channel indices to apply grey-value augmentation to

- **warp** (*bool or float*) – Whether warping/distortion augmentations are applied to examples (slow –> use multiprocessing). If this is a float number, warping is applied to this fraction of examples e.g. 0.5 –> every other example.
- **warp_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.transformations.get_warped_slice()
- **ignore_thresh** (*float*) – If the fraction of negative targets in an example patch exceeds this threshold, this example is discarded (Negative targets are ignored for training [but could be used for unsupervised target propagation]).
- **force_dense** (*bool*) – If True the targets are *not* sub-sampled according to the CNN output strides. Dense targets requires MFP in the CNN!
- **affinities** –
- **nhood_targets** –
- **ret_ll_mask** (*bool*) – If True additional information for reach batch example is returned. Currently implemented are two ll_mask arrays to indicate the targeting mode. The first dimension of those arrays is the batch_size!
- **nga_blur_noise_probability** (*bool or float*) – Probability of applying a Gaussian filter and noise to the input data. The value must be a bool or be within the range [0.0, 1.0] Default: False (disabled)
- **blurry_blobs_probability** (*bool or float*) – Probability of augmenting the input data with blurry "blobs". "Blobs" mean random cubes across the input data. The region within a cube is blurred with Gaussian smoothing. The level of smoothing and the position of each cube are random. The value must be a bool or be within the range [0.0, 1.0] Default: False (disabled)
- **blurry_blobs_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.blurry_blobs(). Effective if setting blurry_blobs_probability > 0.
- **noisy_random_erasing_probability** (*float*) – Probability of augmenting the input data with noise blobs. The blob region is filled with random noise values between 0 and 255.
- **noisy_random_erasing_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.noisy_random_erasing(). Effective if setting noisy_random_erasing_probability > 0.
- **uniform_random_erasing_probability** (*float*) – Probability of augmenting the input data with uniform blobs. The blob region is filled with one value.
- **uniform_random_erasing_args** (*dict*) – Additional keyword arguments that get passed through to elektronn2.data.non_geometric_augmentation.uniform_random_erasing(). Effective if uniform_random_erasing_probability > 0.

Returns

- **data** (*np.ndarray*) – [bs, ch, x, y] or [bs, ch, z, y, x] for 2D and 3D CNNS
- **target** (*np.ndarray*) – [bs, ch, x, y] or [bs, ch, z, y, x]
- **ll_mask1** (*np.ndarray*) – (optional) [bs, n_target]
- **ll_mask2** (*np.ndarray*) – (optional) [bs, n_target]

elektronn2.data.image module

elektronn2.data.image.**make_affinities**(*labels*, *nhood=None*, *size_thresh=1*)
Construct an affinity graph from a segmentation (IDs)

Segments with ID 0 are regarded as disconnected The spatial shape of the affinity graph is the same as of seg_gt. This means that some edges are are undefined and therefore treated as disconnected. If the offsets in nhood are positive, the edges with largest spatial index are undefined.

Connected components is run on the affgraph to relabel the IDs locally.

Parameters

- **labels** (*4d np.ndarray, int (any precision)*) – Volumes of segmentation IDs (bs, z, y, x)
- **nhood** (*2d np.ndarray, int*) – Neighbourhood pattern specifying the edges in the affinity graph Shape: (#edges, ndim) nhood[i] contains the displacement coordinates of edge i The number and order of edges is arbitrary
- **size_thresh** (*int*) – Size filters for connected components, smaller objects are mapped to BG

Returns

- **aff** (*5d np.ndarray int16*) – Affinity graph of shape (bs, #edges, x, y, z) 1: connected, 0: disconnected
- *seg_gt* – 4d np.ndarray int16 Affinity graph of shape (bs, x, y, z) Relabelling of components

elektronn2.data.image.**downsample_xy**(*d*, *l*, *factor*)
Downsample by averaging :param d: data :param l: label :param factor: :return:

elektronn2.data.image.**ids2barriers**(*ids, dilute=[True, True, True], connectivity=[True, True, True], ecs_as_barr=True, smoothen=False*)

elektronn2.data.image.**smearbarriers**(*barriers*, *kernel=None*)
barriers: 3d volume (z,x,y)

elektronn2.data.image.**center_cubes**(*cube1*, *cube2*, *crop=True*)
shapes (ch,x,y,z) or (x,y,z)

elektronn2.data.knossos_array module

class elektronn2.data.knossos_array.**KnossosArray**(*path*, *max_ram=1000*, *n_preload=2*, *fixed_mag=1*)
Bases: object

Interfaces with knossos cubes, all axes are in zxy order!

cut_slice(*shape*, *offset*, *out=None*)

n_f

preload(*position*, *start_end=None*, *sync=False*)
preloads around position preload distance but at least to cover start-end

shape

class elektronn2.data.knossos_array.**KnossosArrayMulti**(*path_prefix*, *feature_paths*, *max_ram=3000*, *n_preload=2*, *fixed_mag=1*)

Bases: *elektronn2.data.knossos_array.KnossosArray*

cut_slice(*shape*, *offset*, *out=None*)

preload(*position*, *sync=True*)

preloads around position preload distance but at least to cover start-end

elektronn2.data.non_geometric_augmentation module

exception elektronn2.data.non_geometric_augmentation.**InvalidNonGeomAugmentParameters**

Bases: exceptions.Exception

elektronn2.data.non_geometric_augmentation.**add_blobs**(*data*, *num_blob_range*, *blob_size_range*, *data_overwrite*, *blob_operator*, *blob_operator_args=()*)

elektronn2.data.non_geometric_augmentation.**blur_augment**(*data*, *level=1*, *data_overwrite=False*)

The function performs Gaussian smoothing on the original data.

By default the raw data will be copied in order to avoid overwriting the original data. However, the user can disable that and allow the function to make changes to the passed data. The function doesn't require to change the corresponding labels of the raw data.

Parameters

- **data** (*np.ndarray*) – Current field of view. Has to have the following format: [num_channels, z, x, y]
- **level** (*int*) – Strength of the gaussian smoothing.
- **data_overwrite** (*bool*) – Determines whether the input data may be modified. If data_overwrite is true the original data passed to the function will be overwritten.

Returns **data** – The array has the following format: [num_channels, z, x, y]

Return type np.ndarray

elektronn2.data.non_geometric_augmentation.**blur_blob**(*blob*, *diffuseness=None*)

elektronn2.data.non_geometric_augmentation.**blurry_blobs**(*data*, *diffuseness*, *num_blob_range=(5, 15)*, *blob_size_range=(10, 32)*, *data_overwrite=False*)

Generates random blobs across the given (raw) input data.

A blob is a cube of the size that will be randomly drawn from the blob_size_range. The area within the blob will be affected by Gausssian smoothing. Depending on the diffuseness level the blob can stay almost transparent (in case of low diffuseness value) or be filled with the mean color value of the blob region (in case of high diffuseness value)

By default the raw data will be copied in order to avoid overwriting the original data. However, the user can disable that and allow the function to make changes to the passed data. The function doesn't require to change the corresponding labels of the raw data

If the size of a blob exceeds the length of any dimension of the give volume the blob size will be assigned to the length of the corresponding dimension.

Parameters

- **data** (*np.ndarray*) – Current field of view. It has to have the following format: [num_channels, z, x, y]
- **diffuseness** (*int*) – The standard deviation of the applied Gaussian kernel.
- **num_blob_range** (*(int, int)*) – Range of possible numbers of blobs generated
- **blob_size_range** (*(int, int)*) – Range of possible blob sizes
- **data_overwrite** (*bool*) – Determines whether the input data may be modified. If `data_overwrite` is true the original data passed to the function will be overwritten.

Returns **data** – Augmented data of the following format: [num_channels, z, x, y]

Return type np.ndarray

`elektronn2.data.non_geometric_augmentation.`**make_blob**(*data*, *blob_operator*, *blob_operator_args*, *depth*, *width*, *height*, *blob_size*)

Generates a random blob within the given field of view.

The user can control the level of diffuseness or it can be generated automatically drawing from a distribution.

Parameters

- **data** (*np.ndarray*) – Represents the current field of view. It has to have the following format: [num_channels, z, x, y]
- **blob_operator** (*Function to apply on the blob, i.e.*) – blur_blob, random_noise_blob or uniform_blob
- **blob_operator_args** (*tuple*) – Arguments that are passed to the blob operator
- **depth** (*int*) – The depth of the field of view
- **width** (*int*) – The width of the field of view
- **height** (*int*) – The height of the field of view
- **blob_size** (*int*) – A particular size of a blob

`elektronn2.data.non_geometric_augmentation.`**noise_augment**(*data*, *level=0.15*, *data_overwrite=False*)

Adds random noise to the original raw data passed to the function.

By default the raw data will be copied in order to avoid of overwriting of the original data. However, the user can disable that and allow the function to make changes on the passed data. The function doesn't require to change the corresponding labels of the raw data

If the noise level is too high (more than 1.0) or too low (less than 0.0) the function throws the corresponding error

Parameters

- **data** (*np.ndarray*) – Current field of view. Has to have the following format: [num_channels, z, x, y]
- **level** (*float*) – Strength of the noise. The maximum value is 1.0.
- **data_overwrite** – Determines whether the input data may be modified. If `data_overwrite` is true the original data passed to the function will be overwritten.

data: np.ndarray The array has the following format: [num_channels, z, x, y]

elektronn2.data.non_geometric_augmentation.**noisy_random_erasing**(*data*, *noise_range=(0, 255)*, *num_blob_range=(5, 15)*, *blob_size_range=(10, 32)*, *data_overwrite=False*)

Like blurry_blobs, but the blob area is filled with random noise.

elektronn2.data.non_geometric_augmentation.**random_noise_blob**(*blob*, *noise_range=(0, 255)*)

elektronn2.data.non_geometric_augmentation.**uniform_blob**(*blob*, *value*)

elektronn2.data.non_geometric_augmentation.**uniform_random_erasing**(*data*, *value*, *num_blob_range=(5, 15)*, *blob_size_range=(10, 32)*, *data_overwrite=False*)

Like blurry blobs, but the blob area is filled with a uniform value.

elektronn2.data.skeleton module

elektronn2.data.skeleton.**trace_to_kzip**(*trace_xyz*, *fname*)

class elektronn2.data.skeleton.**SkeletonMFK**(*aniso_scale=2*, *name=None*, *skel_num=None*)

Bases: object

Joints: all branches and end points / node terminations (nodes not of deg 2) Branches: Joints of degree >= 3

calc_max_dist_to_skels()

static find_joints(*node_list*)

get_closest_node(*position_s*)

get_hull_branch_direc_cutoff(***kwargs0*)

get_hull_branch_dist_cutoff(***kwargs0*)

get_hull_points_inner(***kwargs0*)

get_hull_skel_direc_rel(***kwargs0*)

get_kdtree(***kwargs0*)

get_knn(***kwargs0*)

get_loss_and_gradient(*new_position_s*, *cutoff_inner=0.3333333333333333*, *rise_factor=0.1*)

prediction_c (zxy) Zoned error surface: flat in inner hull (selected at cutoff_inner) constant gradient in "outer" hull towards nearest inner hull voxel gradient increasing with distance (scaled by rise_factor) for predictions outside hull

static get_scale_factor(*radius*, *old_factor*, *scale_strenght*)

Parameters

- **radius** (*predicted radius (not the true radius)*) –

- **old_factor** (*factor by which the radius prediction and the image was scaled*) –
- **scale_strenght** (*limits the maximal scale factor*) –

Returns

Return type new_factor

getbatch(*prediction*, *scale_strenght*, ***get_batch_kwargs*)

Parameters

- **prediction** (*[[new_position_c, radius,]]*) –
- **scale_strenght** (*limits the maximal scale factor for zoom*) –
- **get_batch_kwargs** –

Returns batch

Return type img, target_img, target_grid, target_node

init_from_annotation(*skeleton_annotatation*, *min_radius=None*, *interpolation_resolution=0.5*, *interpolation_order=1*)

interpolate_bone(*bone*, *max_k=1*, *resolution=0.5*)

interpolate_prop(*old_bone*, *old_prop*, *new_bone*, *discrete=False*)

make_grid = <elektronn2.utils.utils_basic.cache object>

map_hull(*hull_points*)
Distances take already into account the anisotropy in z (i.e. they are true distances) But all coordinates for hulls and vectors are still pixel coordinates

plot_debug_traces(*grads=True*, *fig=None*)

plot_hull(*fig=None*)

plot_hull_inner(*cutoff*, *fig=None*)

plot_radii(*fig=None*)

plot_skel(*fig=None*)

plot_vec(*substep=15*, *dict_name='skel'*, *key='direc'*, *vec=None*, *fig=None*)

static point_potential(*r*, *margin_scale*, *size*, *repulsion=None*)

sample_local_direction_iso(*point*, *n_neighbors=6*)
For a point gives the local skeleton direction/orientation by fitting a line through the nearest neighbours, sign is randomly assigned

sample_skel_point(*rng*, *joint_ratio=None*)

sample_tracing_direction_iso(*rng*, *local_direction_iso*, *c=0.5*)
Sample a direction close to the local direction there is a prior so that the normalised (0,1) angle of deviation a has this distribution: p(a) = 1/N * (1-c*a), where N= 1 - c/2, tmp is the inverse cdf of this.

sample_tube_point(*rng*, *r_max_scale=0.9*, *joint_ratio=None*)
This is skeleton node based sampling: Go to a random node, sample a random orthogonal direction go a random distance into direction (uniform over the [0, r_max_scale * local maximal radius])

save(*fname*)

step_feedback(*new_position_s*, *new_direction_is*, *pred_c*, *pred_features*, *cut-off_inner=0.3333333333333333*, *rise_factor=0.1*)

step_grid_update(*grid*, *radius*, *bio*)

class elektronn2.data.skeleton.**Trace**(*linked_skel=None*, *aniso_scale=2*, *max_cutoff=200*, *uturn_detection_k=40*, *uturn_detection_thresh=0.45*, *uturn_detection_hold=10*, *feature_count=7*)

Bases: object

Unless otherwise state all coordinates are in skeleton system (xyz) with z-axis anisotrope and all distances are in pixels (conversion to mu: 1/100)

add_offset(*off*)

append(*coord*, *coord_cnn=None*, *grad=None*, *features=None*)

append_serial(**args*)

avg_dist_self

avg_dist_skel

avg_seg_length

max_dist_skel

min_dist_self

min_normed_dist_self

new_cut_trace(*start*, *stop*)

new_reverted_trace()

plot(*grads=True*, *skel=True*, *rand_color=False*, *fig=None*)

runlength

save(*fname*)

save_to_kzip(*fname*)

split_uturns(*return_accum_pathlength=False*, *print_stat=False*)

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

elektronn2.data.tracing_utils module

class elektronn2.data.tracing_utils.**Tracer**(*model*, *z_shift=0*, *data_source=None*, *bounding_box_zyx=None*, *trace_kwargs={'aniso_scale': 2}*, *modus='m'*, *shotgun_registry=None*, *registry_interval=None*, *reference_radius=18.0*)

Bases: object

get_scale_factor(*radius*, *old_factor*, *scale_strenght*)

static perturb_direciton(*direc*, *azimuth*, *polar*)

static plot_vectors(*cv*, *vectors*, *fig=None*)

trace(*position_l*, *direction_il*, *count*, *gamma=0*, *trace_xyz=None*, *linked_skel=None*, *check_for_lost_track=True*, *check_for_uturn=False*, *check_bb=True*, *profile=False*, *info_str=None*, *reject_obb_traces=False*, *initial_scale=None*)

Although psoition_l is in zyx order, the returned trace_obj is in xyz order

static zeropad(*a*, *length*)

class elektronn2.data.tracing_utils.**CubeShape**(*shape*, *offset=None*, *center=None*, *input_excess=None*, *bbox_reduction=None*)

Bases: object

bbox_off_sh_cent(*bbox_reduction=None*)

bbox_wrt_input()

bbox_wrt_self()

input_off_sh_cent(*input_excess=None*)

shrink_off_sh_cent(*amount*)

class elektronn2.data.tracing_utils.**ShotgunRegistry**(*seeds_zyx*, *registry_extent*, *directions=None*, *debug=False*, *radius_discout=0.5*, *check_w=3*, *occupied_thresh=0.6*, *candidate_max_rel=0.75*, *candidate_max_min_margin=1.5*)

Bases: object

check(*trace*)

Check if trace goes into masked volume. If so, find out to which trace tree this belongs and merge. Return False to stop tracing Mask seeds and volume mask by current trace's log

W: window length to do check on

find_nearest_trace(*coords_xyz*)

Find all other tracks that are at least as close as 1.5 minimal (relative!) distance. (compare to closest point of each track)

get_next_seed()

new_trace(*trace*)

plot_mask_vol(*figure=None*, *adjust_tfs=False*)

update_mask(*coords_xyz*, *radii*, *index=None*)

elektronn2.data.traindata module

class elektronn2.data.traindata.**Data**(*n_lab=None*)

Bases: object

Load and prepare data, Base-Obj

createCVSplit(*data*, *label*, *n_folds=3*, *use_fold=2*, *shuffle=False*, *random_state=None*)

getbatch(*batch_size*, *source='train'*)

class elektronn2.data.traindata.**MNISTData**(*input_node*, *target_node*, *path=None*, *convert2image=True*, *warp_on=False*, *shift_augment=True*, *center=True*)

Bases: *elektronn2.data.traindata.Data*

convert_to_image()

For MNIST / flattened 2d, single-Layer, square images

static download()

getbatch(*batch_size*, *source='train'*)

class elektronn2.data.traindata.**PianoData**(*input_node*, *target_node*, *path='/home/mkilling/devel/data/PianoRoll/Nottingham_enc.pkl'*, *n_tap=20*, *n_lab=58*)

Bases: *elektronn2.data.traindata.Data*

getbatch(*batch_size*, *source='train'*)

class elektronn2.data.traindata.**PianoData_perc**(*input_node*, *target_node*, *path='/home/mkilling/devel/data/PianoRoll/Nottingham_enc.pkl*
n_tap=20, *n_lab=58*)

Bases: *elektronn2.data.traindata.PianoData*

getbatch(*batch_size*, *source='train'*)

elektronn2.data.transformations module

elektronn2.data.transformations.**warp_slice**(*img*, *ps*, *M*, *target=None*, *target_ps=None*, *target_vec_ix=None*, *target_discrete_ix=None*, *last_ch_max_interp=False*, *ksize=0.5*)

Cuts a warped slice out of the input image and out of the target image. Warping is applied by multiplying the original source coordinates with the inverse of the homogeneous (forward) transformation matrix `M`.

"Source coordinates" (`src_coords`) signify the coordinates of voxels in `img` and `target` that are used to compose their respective warped versions. The idea here is that not the images themselves, but the coordinates from where they are read are warped. his allows for much higher efficiency for large image volumes because we don't have to calculate the expensive warping transform for the whole image, but only for the voxels that we eventually want to use for the new warped image. The transformed coordinates usually don't align to the discrete voxel grids of the original images (meaning they are not integers), so the new voxel values are obtained by linear interpolation.

Parameters

- **img** (*np.ndarray*) – Image array in shape `(f, z, x, y)`
- **ps** (*tuple*) – (spatial only) Patch size `(z, x, y)` (spatial shape of the neural network's input node)
- **M** (*np.ndarray*) – Forward warping tansformation matrix (4x4). Must contain translations in source and target array.
- **target** (*np.ndarray or None*) – Optional target array to be extracted in the same way.
- **target_ps** (*tuple*) – Patch size for the `target` array.
- **target_vec_ix** (*list*) – List of triples that denote vector value parts in the target array. E.g. [(0,1,2), (4,5,6)] denotes two vector fields, separated by a scalar field in channel 3.
- **last_ch_max_interp** (*bool*) –
- **ksize** (*float*) –

Returns

- **img_new** (*np.ndarray*) – Warped input image slice
- **target_new** (*np.ndarray or None*) – Warped target image slice or `None`, if `target is None`.

elektronn2.data.transformations.**get_tracing_slice**(*img*, *ps*, *pos*, *z_shift=0*, *aniso_factor=2*, *sample_aniso=True*, *gamma=0*, *scale_factor=1.0*, *direction_iso=None*, *target=None*, *target_ps=None*, *target_vec_ix=None*, *target_discrete_ix=None*, *rng=None*, *last_ch_max_interp=False*)

exception elektronn2.data.transformations.**WarpingOOBError**(**args*, ***kwargs*)
Bases: exceptions.ValueError

class elektronn2.data.transformations.**Transform**(*M*, *position_l=None*, *aniso_factor=2*)
Bases: object

M_lin

M_lin_inv

cnn_coord2lab_coord(*vec_c*, *add_offset_l=False*)

cnn_pred2lab_position(*prediction_c*)

lab_coord2cnn_coord(*vec_l*)

to_array()

elektronn2.data.transformations.**trafo_from_array**(*a*)

elektronn2.data.transformations.**get_warped_slice**(*img*, *ps*, *aniso_factor=2*, *sample_aniso=True*, *warp_amount=1.0*, *lock_z=True*, *no_x_flip=False*, *perspective=False*, *target=None*, *target_ps=None*, *target_vec_ix=None*, *target_discrete_ix=None*, *rng=None*)
(Wraps *elektronn2.data.transformations.warp_slice()*)

Generates the warping transformation parameters and composes them into a single 4D homogeneous transformation matrix `M`. Then this transformation is applied to `img` and `target` in the `warp_slice()` function and the transformed input and target image are returned.

Parameters

- **img** (*np.array*) – Input image
- **ps** (*np.array*) – Patch size (spatial shape of the neural network's input node)
- **aniso_factor** (*float*) – Anisotropy factor that determines an additional scaling in `z` direction.
- **sample_aniso** (*bool*) – Scale coordinates by `1 / aniso_factor` while warping.
- **warp_amount** (*float*) – Strength of the random warping transformation. A lower `warp_amount` will lead to less distorted images.
- **lock_z** (*bool*) – Exclude `z` coordinates from the random warping transformations.
- **no_x_flip** (*bool*) – Don't flip `x` axis during random warping.
- **perspective** (*bool*) – Apply perspective transformations (in addition to affine ones).
- **target** (*np.array*) – Target image

- **target_ps** (*np.array*) – Target patch size
- **target_vec_ix** –
- **target_discrete_ix** –
- **rng** (*np.random.mtrand.RandomState*) – Random number generator state (obtainable by `np.random.RandomState()`). Passing a known state makes the random transformations reproducible.

Returns

- **img_new** (*np.ndarray*) – (Warped) input image slice
- **target_new** (*np.ndarray*) – (Warped) target slice

3.4 elektronn2.utils package

3.4.1 Subpackages

elektronn2.utils.d3viz package

Submodules

elektronn2.utils.d3viz.formatting module

Visualisation code taken form Theano Original Author: Christof Angermueller <cangermueller@gmail.com> Adapted with permission for the ELEKTRONN2 Toolkit by Marius Killinger 2016 Note that this code is licensed under the original terms of Theano (see license containing directory).

class elektronn2.utils.d3viz.formatting.**PyDotFormatter2**(*compact=True*)
Bases: object

Create *pydot* graph object from Theano function.

Parameters **compact** (*bool*) – if True, will remove intermediate variables without name.

node_colors
Color table of node types.

Type dict

apply_colors
Color table of apply nodes.

Type dict

shapes
Shape table of node types.

Type dict

add_scan_edges(*scan*, *graph*, *nodes*)

get_node_props(*node*)

elektronn2.utils.d3viz.formatting.**dict_to_pdnode**(*d*)
Create pydot node from dict.

elektronn2.utils.d3viz.formatting.**escape_quotes**(*s*)
Escape quotes in string.

Parameters **s** (*str*) – String on which function is applied

elektronn2.utils.d3viz.formatting.**replace_patterns**(*x*, *replace*)
Replace *replace* in string *x*.

Parameters

- **s** (*str*) – String on which function is applied
- **replace** (*dict*) – *key*, *value* pairs where key is a regular expression and *value* a string by which *key* is replaced

elektronn2.utils.d3viz.formatting.**sort**(*model*, *select_outputs*)

elektronn2.utils.d3viz.formatting.**visualise_model**(*model*, *outfile*, *copy_deps=True*, *select_outputs=None*, *image_format='png'*, **args*, ***kwargs*)

Parameters

- **model** (*model object*) –
- **outfile** (*str*) – Path to output HTML file.
- **copy_deps** (*bool, optional*) – Copy javascript and CSS dependencies to output directory.

Notes

This function accepts extra parameters which will be forwarded to `theano.d3viz.formatting.PyDotFormatter`.

3.4.2 Submodules

elektronn2.utils.cnncalculator module

elektronn2.utils.cnncalculator.**cnncalculator**(*filters*, *poolings*, *desired_patch_size=None*, *mfp=False*, *force_center=False*, *desired_output=None*, *ndim=1*)
Helper to calculate CNN architectures

This is a *function*, but it returns an *object* that has various architecture values as attributes. Useful is also to simply print 'd' as in the example.

Parameters

- **filters** (*list*) – Filter shapes (for anisotropic filters the shapes are again a list)
- **poolings** (*list*) – Pooling factors
- **desired_patch_size** (*int or list[int]*) – Desired patch_size size(s). If `None` a range of suggestions can be found in the attribute `valid_patch_sizes`
- **mfp** (*list[bool] or bool*) – Whether to apply Max-Fragment-Pooling in this Layer and check compliance with max-fragment-pooling (requires other patch_size sizes than normal pooling)
- **force_center** (*Bool*) – Check if output neurons/pixel lie at center of patch_size neurons/pixel (and not in between)

- **desired_output** (*None or int or list[int]*) – Alternative to desired_patch_size
- **ndim** (*int*) – Dimensionality of CNN

Examples

Calculation for anisotropic "flat" 3d CNN with mfp in the first layers only:

```
>>> desired_patch_size   = [8, 211, 211]
>>> filters         = [[1,6,6], [4,4,4], [2,2,2], [1,1,1]]
>>> pool            = [[1,2,2], [2,2,2], [2,2,2], [1,1,1]]
>>> mfp             = [True,    True,    False,    False ]
>>> ndim=3
>>> d = cnncalculator(filters, pool, desired_patch_size, mfp=mfp, force_
↪center=True, desired_output=None, ndim=ndim)
patch_size (8) changed to (10) (size too small)
patch_size (211) changed to (210) (size not possible)
patch_size (211) changed to (210) (size not possible)
>>> print(d)
patch_size: [10, 210, 210]
Layer/Fragment sizes:   [(1, 24, 24), (1, 24, 24), (3, 49, 49), (10, 102, 102)]
Unpooled Layer sizes:   [(1, 24, 24), (2, 48, 48), (7, 99, 99), (10, 205, 205)]
Receptive fields:       [(9, 23, 23), (9, 23, 23), (5, 15, 15), (1, 7, 7)]
Strides:                [(4, 8, 8), (4, 8, 8), (2, 4, 4), (1, 2, 2)]
Overlap:                [(5, 15, 15), (5, 15, 15), (3, 11, 11), (0, 5, 5)]
Offset:         [4.5, 11.5, 11.5]
If offset is non-int: output neurons lie centered on patch_size neurons,they have␣
↪an odd FOV
```

elektronn2.utils.cnncalculator.**get_cloesest_valid_patch_size**(*filters*, *poolings*, *desired_patch_size=100*, *mfp=False*, *ndim=1*)

elektronn2.utils.cnncalculator.**get_valid_patch_sizes**(*filters*, *poolings*, *desired_patch_size=100*, *mfp=False*, *ndim=1*)

elektronn2.utils.gpu module

elektronn2.utils.gpu.**get_free_gpu**(*wait=0*, *nb_gpus=-1*)

elektronn2.utils.gpu.**initgpu**(*gpu*)

elektronn2.utils.legacy module

elektronn2.utils.legacy.**create_cnn**(*config_file*, *n_ch*, *param_file=None*, *mfp=False*, *axis_order='theano'*, *constant_weights=False*, *imposed_input_size=None*)

elektronn2.utils.legacy.**load_params_into_model**(*param_file*, *model*)

Loads parameters directly from save file into a graph manager (this requires that the graph is identical to the cnn from the param file :param param_file: :param gm: :return:

elektronn2.utils.locking module

Implementation of a simple cross-platform file locking mechanism. This is a modified version of code retrieved on 2013-01-01 from http://www.evanfosmark.com/2009/01/cross-platform-file-locking-support-in-python. (The original code was released under the BSD License. See below for details.)

Modifications in this version:

- Tweak docstrings for sphinx.
- Accept an absolute path for the protected file (instead of a file name relative to cwd).
- Allow timeout to be None.
- Fixed a bug that caused the original code to be NON-threadsafe when the same FileLock instance was shared by multiple threads in one process. (The original was safe for multiple processes, but not multiple threads in a single process. This version is safe for both cases.)
- Added `purge()` function.
- Added `available()` function.
- Expanded API to mimic `threading.Lock interface`: - `__enter__` always calls `acquire()`, and therefore blocks if `acquire()` was called previously. - `__exit__` always calls `release()`. It is therefore a bug to call `release()` from within a context manager. - Added `locked()` function. - Added blocking parameter to `acquire()` method

Warning:

- The locking mechanism used here may need to be changed to support old NFS filesystems: http://lwn.net/Articles/251004 (Newer versions of NFS should be okay, e.g. NFSv3 with Linux kernel 2.6. Check the open(2) man page for details about O_EXCL.)
- This code has not been thoroughly tested on Windows, and there has been one report of incorrect results on Windows XP and Windows 7. The locking mechanism used in this class should (in theory) be cross-platform, but use at your own risk.

ORIGINAL LICENSE:

The original code did not properly include license text. (It merely said "License: BSD".) Therefore, we'll attach the following generic BSD License terms to this file. Those who extract this file from the lazyflow code base (LGPL) for their own use are therefore bound by the terms of both the Simplified BSD License below AND the LGPL.

Copyright (c) 2013, Evan Fosmark and others. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED

TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FreeBSD Project.

class elektronn2.utils.locking.**FileLock**(*protected_file_path*, *timeout=120*, *delay=1*, *lock_file_contents=None*)

Bases: object

A file locking mechanism that has context-manager support so you can use it in a with statement. This should be relatively cross compatible as it doesn't rely on msvcrt or fcntl for the locking.

exception FileLockException

Bases: exceptions.Exception

acquire(*blocking=True*)

Acquire the lock, if possible. If the lock is in use, and *blocking* is False, return False. Otherwise, check again every *self.delay* seconds until it either gets the lock or exceeds *timeout* number of seconds, in which case it raises an exception.

available()

Returns True iff the file is currently available to be locked.

locked()

Returns True iff the file is owned by THIS FileLock instance. (Even if this returns false, the file could be owned by another FileLock instance, possibly in a different thread or process).

purge()

For debug purposes only. Removes the lock file from the hard disk.

release()

Get rid of the lock by deleting the lockfile. When working in a *with* statement, this gets automatically called at the end.

elektronn2.utils.plotting module

class elektronn2.utils.plotting.**Scroller**(*axes*, *images*, *names*, *init_z=None*)

Bases: object

onscroll(*event*)

update()

elektronn2.utils.plotting.**add_timeticks**(*ax*, *times*, *steps*, *time_str='mins'*, *num=5*)

elektronn2.utils.plotting.**embedfilters**(*filters*, *border_width=1*, *normalize=False*, *output_ratio=1.0*, *rgb_axis=None*)

Embed an nd array into an 2d matrix by tiling. The last two dimensions of a are assumed to be spatial, the others are tiled recursively.

elektronn2.utils.plotting.**my_quiver**(*x*, *y*, *img=None*, *c=None*)

first dim of x,y changes along vertical axis second dim changes along horizontal axis x: vertical vector component y: horizontal vector component

elektronn2.utils.plotting.**plot_debug**(*var*, *debug_output_names*, *save_name*)

elektronn2.utils.plotting.**plot_exectimes**(*exectimes*, *save_path='~/exectimes.png'*, *max_items=32*)
Plot model execution time dict obtained from elektronn2.neuromancer.model.Model.measure_exectimes()

Parameters

- **exectimes** – OrderedDict of execution times (output of Model.measure_exectimes())
- **save_path** – Where to save the plot
- **max_items** – Only the max_items largest execution times are given names and are plotted independently. Everything else is grouped under '(other nodes)'.

elektronn2.utils.plotting.**plot_hist**(*timeline*, *history*, *save_name*, *loss_smoothing_length=200*, *autoscale=True*)
Plot graphical info during Training

elektronn2.utils.plotting.**plot_kde**(*pred*, *target*, *save_name*, *limit=90*, *scale='same'*, *grid=50*, *take_last=4000*)

elektronn2.utils.plotting.**plot_regression**(*pred*, *target*, *save_name*, *loss_smoothing_length=200*, *autoscale=True*)
Plot graphical info during Training

elektronn2.utils.plotting.**plot_trainingtarget**(*img*, *lab*, *stride=1*)
Plots raw image vs target to check if valid batches are produced. Raw data is also shown overlaid with targets

Parameters

- **img** (*2d array*) – raw image from batch
- **lab** (*2d array*) – targets
- **stride** (*int*) – strides of targets

elektronn2.utils.plotting.**plot_var**(*var*, *save_name*)

elektronn2.utils.plotting.**scroll_plot**(*images*, *names=None*, *init_z=None*)
Creates a plot 1x2 image plot of 3d volume images Scrolling changes the displayed slices

Parameters

- **images** (*list of arrays (or single)*) – Each array of shape (z,y,x) or (z,y,x,RGB)
- **names** (*list of strings (or single)*) – Names for each image
- **Usage** –
- **-----** –
- **the scroll interaction to work, the "scroller" object** (*For*) –
- **be returned to the calling scope** (*must*) –
- **fig, scroller = _scroll_plot4(images, names)** (*>>>*) –
- **fig.show()** (*>>>*) –

elektronn2.utils.plotting.**sma**(*c*, *n*)
Returns box-SMA of c with box length n, the returned array has the same length as c and is const-padded at the beginning

elektronn2.utils.ptk_completions module

Provides completions for the Python language, file sytem paths and a custom list of words for the ELEKTRONN2/Elektronn prompt_toolkit shell.

This module is mostly based on - https://github.com/jonathanslenders/ptpython/blob/master/ptpython/completer.py - https://github.com/jonathanslenders/ptpython/blob/master/ptpython/utils.py (at git revision 32827385cca65eabefccb06b56e4cf9d2c1e0120), which both are available under the following license (thanks, Jonathan and contributors!):

class elektronn2.utils.ptk_completions.**NumaCompleter**(*get_globals*, *get_locals*, *words=None*, *words_metastring=u"*)

Bases: prompt_toolkit.completion.Completer

Completer for Python, file system paths and custom words

get_completions(*document*, *complete_event*)
Get completions.

elektronn2.utils.utils_basic module

elektronn2.utils.utils_basic.**get_free_cpu_count**()

elektronn2.utils.utils_basic.**parallel_accum**(*func*, *n_ret*, *var_args*, *const_args*, *proc=-1*, *debug=False*)

class elektronn2.utils.utils_basic.**timeit**(**args*, ***kwargs*)
Bases: elektronn2.utils.utils_basic.DecoratorBase

class elektronn2.utils.utils_basic.**cache**(**args*, ***kwargs*)
Bases: elektronn2.utils.utils_basic.DecoratorBase

static hash_args(*args*)

class elektronn2.utils.utils_basic.**CircularBuffer**(*buffer_len*)

Bases: object

append(*data*)

data

mean()

setvals(*val*)

class elektronn2.utils.utils_basic.**AccumulationArray**(*right_shape=()*, *dtype=<type 'numpy.float32'>*, *n_init=100*, *data=None*, *ema_factor=0.95*)

Bases: object

add_offset(*off*)

append(*data*)

clear()

data

ema

max()

mean()

min()

sum()

class elektronn2.utils.utils_basic.**KDT**(*n_neighbors=5*, *radius=1.0*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *n_jobs=1*, ***kwargs*)

Bases: sklearn.neighbors.unsupervised.NearestNeighbors

warning_shown = False

class elektronn2.utils.utils_basic.**DynamicKDT**(*points=None*, *k=1*, *n_jobs=-1*, *rebuild_thresh=100*, *aniso_scale=None*)

Bases: object

append(*point*)

get_knn(*query_points*, *k=None*)

get_radius_nn(*query_points*, *radius*)

elektronn2.utils.utils_basic.**import_variable_from_file**(*file_path*, *class_name*)

elektronn2.utils.utils_basic.**pickleload**(*file_name*)

Loads all object that are saved in the pickle file. Multiple objects are returned as list.

elektronn2.utils.utils_basic.**picklesave**(*data*, *file_name*)

Writes one or many objects to pickle file

data: single objects to save or iterable of objects to save. For iterable, all objects are written in this order to the file.

file_name: string path/name of destination file

elektronn2.utils.utils_basic.**h5save**(*data*, *file_name*, *keys=None*, *compress=True*)

Writes one or many arrays to h5 file

data: single array to save or iterable of arrays to save. For iterable all arrays are written to the file.

file_name: string path/name of destination file

keys: string / list thereof For single arrays this is a single string which is used as a name for the data set. For multiple arrays each dataset is named by the corresponding key. If keys is `None`, the dataset names created by enumeration: `data%i`

compress: Bool Whether to use lzf compression, defaults to `True`. Most useful for label arrays.

`elektronn2.utils.utils_basic.`**`h5load`**(*file_name*, *keys=None*)
Loads data sets from h5 file

file_name: string destination file

keys: string / list thereof Load only data sets specified in keys and return as list in the order of `keys` For a single key the data is returned directly - not as list If keys is `None` all datasets that are listed in the keys-attribute of the h5 file are loaded.

`elektronn2.utils.utils_basic.`**`pretty_string_ops`**(*n*)
Return a humanized string representation of a large number.

`elektronn2.utils.utils_basic.`**`pretty_string_time`**(*t*)
Custom printing of elapsed time

`elektronn2.utils.utils_basic.`**`makeversiondir`**(*path*, *dir_name=None*, *cd=False*)

class `elektronn2.utils.utils_basic.`**`Timer`**(*silent_all=False*)
Bases: `object`

`check`(*name=None*, *silent=False*)

`plot`(*accum=False*)

`summary`(*silent=False*, *print_func=None*)

`elektronn2.utils.utils_basic.`**`unique_rows`**(*a*)

`elektronn2.utils.utils_basic.`**`as_list`**(*var*)

CHAPTER 4

Indices and tables

- genindex
- modindex

Python Module Index

e

Index

A

B

C

D

E

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z