ELEKTRONN2 Documentation

Release

Marius Killinger

Mar 21, 2017

Contents

CHAPTER 1

User guide

Examples

Important: This page is heavily outdated and under construction. There are many sections that have to be rewritten from ELEKTRONN (legacy) specifics to fit to ELEKTRONN2.

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 (e.g. from training) is recommended. The details of the configuration parameters are described here.

A simple 3D CNN

The first example demonstrates how a 3-dimensional CNN and its loss function are specified. 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.

Defining the neural network model

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

```
from elektronn2 import neuromancer

image = neuromancer.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).
conv0 = neuromancer.Conv(image, 32, (1,6,6), (1,2,2))
conv1 = neuromancer.Conv(conv0, 64, (4,6,6), (2,2,2))
conv2 = neuromancer.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 = neuromancer.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 = neuromancer.Input_like(class_probs, override_f=1, name='target', dtype='int16
↪')

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

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

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

[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.

[2] For complete network config files that you can directly run with little to no modification, see the "examples" directory in the code repository.

```
<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
```

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')
```

Interactive usage of the `Model` and `Node` objects

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

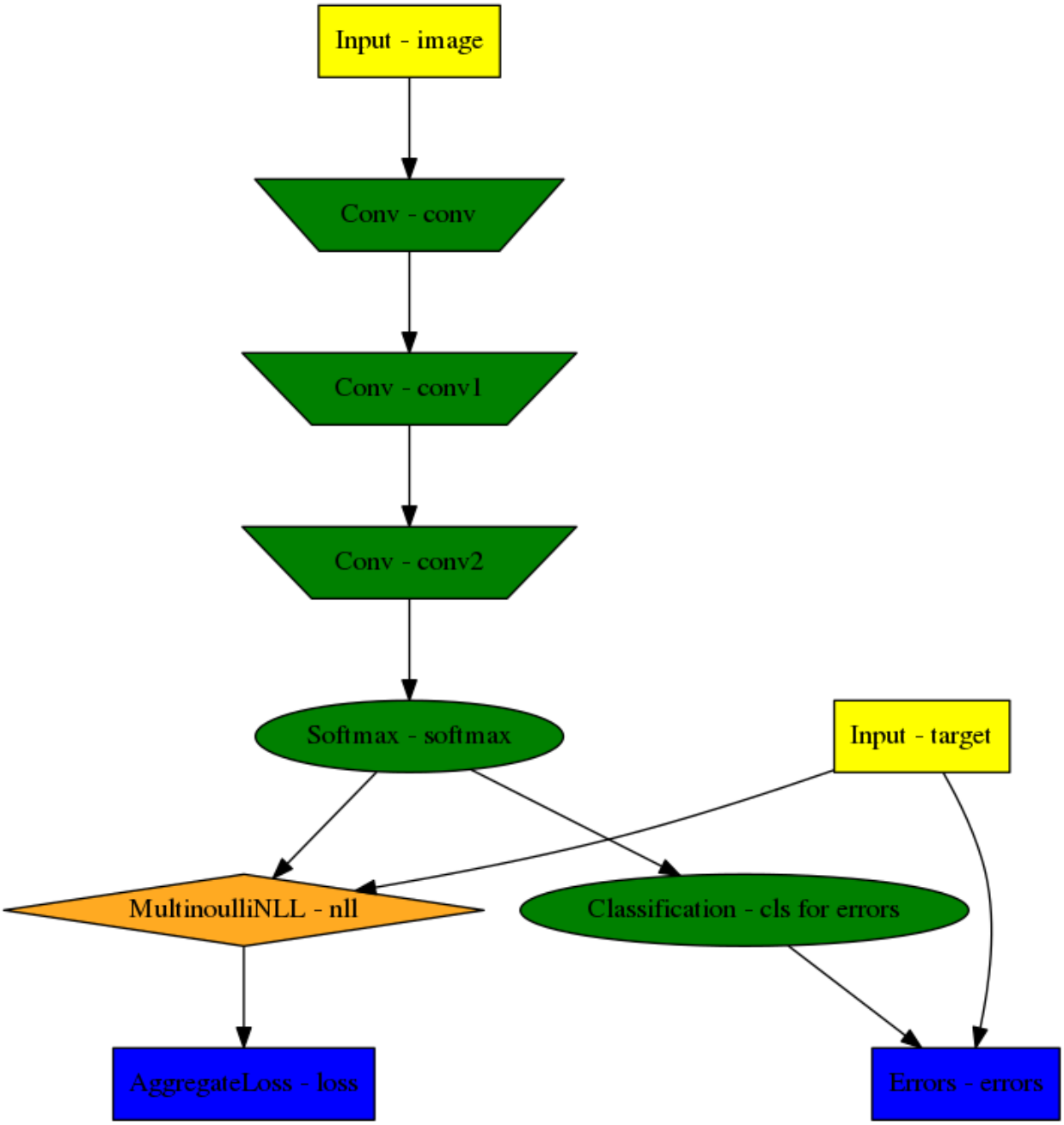

Fig. 1.1: Model graph of the example CNN. Inputs are yellow and outputs are blue. Some node classes are represented by special shapes, the default shape is oval.

compiled function:

```
>>> test_output = class_probs(test_image)
Compiling softmax, inputs=[image]
Compiling done - in 21.32 s
>>> import numpy as np
>>> np.allclose(test_output, reference_output)
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:

```
>>> model_prediction = neuromancer.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)
...
```

Plotting the `model` graph:

```
>>> utils.d3viz.visualise_model(model, '/tmp/model')
```

3D Neuro Data

Important: This section is out of date and has to be revised for ELEKTRONN2

This task is about detecting neuron cell boundaries in 3D electron microscopy image volumes. The more general goal is to find a volume segmentation by assigning each voxel a cell ID. Predicting boundaries is a surrogate target for which a CNN can be trained (see also the note about target formulation here) - the actual segmentation would be made by e.g. running a watershed on the predicted boundary map. This is a typical *img-img* task.

For demonstration purpose, a very small CNN with only 70k parameters and 5 layers is used. This 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 can be found in ELEKTRONN2's `examples` folder as `neuro_3d_config.py`. Here only selected settings will be mentioned.

Getting Started

Important: This section is out of date and has to be revised for ELEKTRONN2

1. Download example training data (~100MB):

```
wget http://elektronn.org/downloads/neuro_data.zip
unzip neuro_data.zip
```

2. Edit `save_path`, `data_path`, `label_path`, `preview_data_path` in the config file `neuro_3d_config.py` in ELEKTRONN2's `examples` folder

3. Run:

```
elektronn2-train </path/to_config_file> [ --gpu={Auto|False|<int>}]
```

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

Data Set

Important: This section is out of date and has to be revised for ELEKTRONN2

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 `(x,y,z)` 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 `(350, 350,250)` 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.

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 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.

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, the cubes are for performance reasons internally axis swapped to `(z, (ch,) x, y)` order, 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. 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.

CNN design

Important: This section is out of date and has to be revised for ELEKTRONN2

The architecture of the CNN is determined by:

```
n_dim = 3
filters = [[4,4,1],[3,3,1],[3,3,3],[3,3,3],[2,2,1]]
pool    = [[2,2,1],[2,2,1],[1,1,1],[1,1,1],[1,1,1]]
nof_filters = [10,20,40,40,40]
desired_input = [127,127,7]
batch_size = 1
```

- Because the data is anisotropic the lateral 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: in the first two and in the last layer the filter kernels have extent `1` in `z`. Only two middle layers perform a truly 3D aggregation of the features along the third axis.
- The resulting FOV is `[31,31,7]` (to solve this task well, more than `100` lateral FOV is beneficial...)
- Using this input size gives an output shape of `[25,25,3]` i.e. 1875 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-created the CNN with a larger input size (see here).
- If there are several `100-1000` output neurons, a batch size of `1` is commonly sufficient and is not necessary to compute an average gradient over several images.
- The output shape has strides of `[4,4,1]` due to 2 times lateral pooling by 2. This means that the predicted `[25,25,3]` voxels do not lie laterally adjacent, if projected back to the space of the input image: for every lateral output voxel there are `3` voxel separating it from the next output voxel - for those no prediction is available. To obtain dense predictions (e.g. when making the live previews) the method `elektronn2.net.convnet.MixedConvNN.predictDense()` 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 (see here).
- To solve this task well, about twice the number of layers, several million parameters and more training data are needed.

Training Data Options

Important: This section is out of date and has to be revised for ELEKTRONN2

Config:

```
valid_cubes = [2,]
grey_augment_channels = [0]
flip_data = True
anisotropic_data = True
warp_on = 0.7
```

- Of the three training data cubes the last one is used as validation data.
- The input images are grey-valued i.e. they have only 1 channel. For this channel "grey value augmentaion" (randomised histogram distortions) are applied when sampling batches during training. This helps to achieve invariance against varying contrast and brightness gradients.
- During patch cutting the axes are flipped and transposed as a means of data augmentation.
- If the data is anisotropic, the pipeline assumes that the singled-out axis is `z`. For anisotropic data axes are not transposed in a way that axes of different resolution get mixed up.
- For 70% of the batches the image and labels are randomly warped.

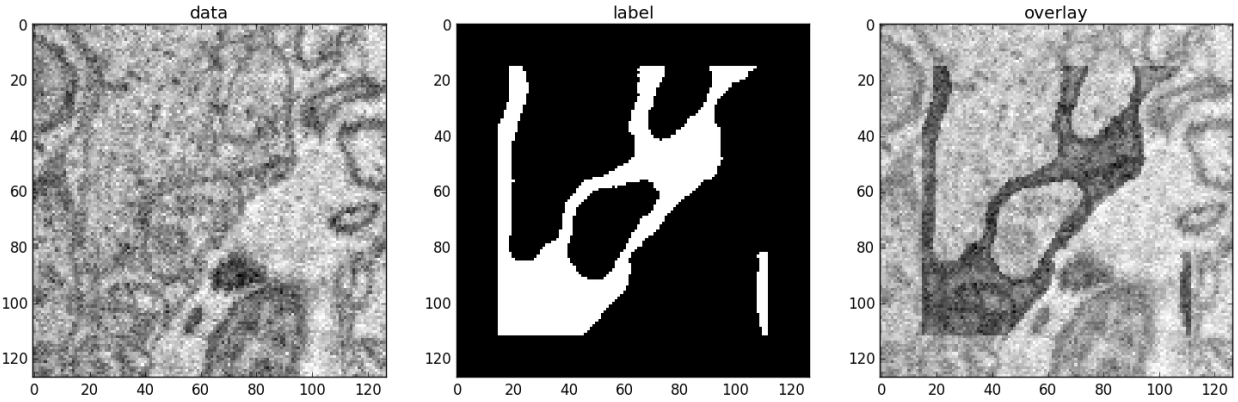

Fig. 1.2: Left: The input data. Centre: The labels, note the offset. Right: Overlay of data with labels, here you can check whether they are properly registered.

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 to images). Once the training loop has started, more such plots can be made from the ELEKTRONN2 command line (`ctrl+c`)

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

Note: **Training with 2D images**: The shown setup works likewise for training a 2D CNN on this task. Just the CNN configuration parameters must be adjusted. Then 2D training patches are cut from the cubes. If `anisotropic_data = True` these are cut only from the `x,y`-plane; otherwise transposed, too. Therefore, this setup can be used for actual 2D images if they are stacked to form a cube along a new "`z`"-axis. If the 2D images have different shapes they cannot be stacked but, the 2D arrays can be augmented with a third dummy-axis to be of shape `(x,y,1)` and each put in a separate `h5`-file, which is slightly more intricate.

Results & Comments

Important: This section is out of date and has to be revised for ELEKTRONN2

- When running this example, commonly the NLL-loss stagnates for about `15k` iterations around `0.7`. After that you should observe a clear decrease. On a desktop with a high-end GPU, with latest theano and cuDNN versions and using background processes for the batch creation the training should run `at 15-20 it/s`.
- Because of the (too) small training data size the validation error should stagnate soon and even go up later.
- Because the model has too few parameters, predictions are typically not smooth and exhibit grating-like patterns - using a more complex model mitigates this effect.
- Because the model has a small FOV (which for this task should rather be increase by more layers than more maxpooling) predictions contain a lot of "clutter" within wide cell bodies: there the CNN does not see the the cell outline which is apparently an important clue to solve this task.

MNIST Examples

Important: This section is out of date and has to be revised for ELEKTRONN2

MNIST is a benchmark data set for handwritten digit recognition/classification. State of the art benchmarks for comparison can be found here.

Note: The data will be automatically downloaded but can also be downloaded manually from here.

CNN with built-in Pipeline

Important: This section is out of date and has to be revised for ELEKTRONN2

In ELEKTRONN2's `examples` folder is a file `MNIST_CNN_warp_config.py`. This is a configuration for *img-scalar* training and it uses a different data class than the "big" pipeline for neuro data. When using an alternative data pipeline, the options for data loading and batch creation are given given by keyword argument dictionaries in the `Data Alternative` section of the config file:

```
data_class_name      = 'MNISTData'
data_load_kwargs     = dict(path=None, convert2image=True, warp_on=True, shift_
↪augment=True)
data_batch_kwargs    = dict()
```

This configuration results in:

- Initialising a data class adapted for MNIST from *`elektronn2.data.traindata`*
- Downloading the MNIST data automatically if path is `None` (otherwise the given path is used)
- Reshaping the "flat" training examples (they are stored as vectors of length 784) to `28 x 28` matrices i.e. images

- Data augmentation through warping (see warping): for each batch in a training iteration random deformation parameters are sampled and the corresponding transformations are applied to the images in a background process.
- Data augmentation through translation: `shift_augment` crops the `28 x 28` images to `26 x 26` (you may notice this in the printed output). The cropping leaves choice of the origin (like applying small translations), in this example the data set size is inflated by factor `4`.
- For the function `getbatch` no additional kwargs are required (the warping and so on was specified already with the initialisation).

The architecture of the NN is determined by:

```
n_dim           = 2             # MNIST are 2D images
desired_input   = 26
filters         = [3,3]         # two conv layers with each 3x3 filters
pool            = [2,2]         # for each conv layer maxpooling by 2x2
nof_filters     = [16,32]       # number of feature maps per layer
MLP_layers       = [300,300]    # numbers of filters for perceptron layers (after conv␣
↪layers)
```

This is 2D CNN with two conv layers and two fully connected layers each with 300 neurons. As MNIST has 10 classes, an output layer with 10 neurons is automatically added, and not specified here.

To run the example, make a copy of the config file and adjust the paths. Then run the `elektronn2-train` script, and pass the path of your config file:

```
elektronn2-train </path/to_config_file> [ --gpu={Auto|False|<int>}]
```

The output should read like this:

```
Reading config-file ../elektronn2/examples/MNIST_CNN_warp_config.py
WARNING: Receptive Fields are not centered with even field of view (10)
WARNING: Receptive Fields are not centered with even field of view (10)
Selected patch-size for CNN input: Input: [26, 26]
Layer/Fragment sizes:        [[12, 5], [12, 5]]
Unpooled Layer sizes:        [[24, 10], [24, 10]]
Receptive fields:    [[4, 10], [4, 10]]
Strides:             [[2, 4], [2, 4]]
Overlap:             [[2, 6], [2, 6]]
Offset:              [5.0, 5.0].
If offset is non-int: output neurons lie centered on input neurons,they have an odd␣
↪FOV

Overwriting existing save directory: /home/mfk/CNN_Training/2D/MNIST_example_warp/
Using gpu device 0: GeForce GTX TITAN
Load ELEKTRONN2 Core
10-class Data Set: #training examples: 200000 and #validing: 10000
MNIST data is converted/augmented to shape (1, 26, 26)
------------------------------------------------------------
Input shape   =  (50, 1, 26, 26) ; This is a 2 dimensional NN
---
2DConv: input= (50, 1, 26, 26)       filter= (16, 1, 3, 3)
Output = (50, 16, 12, 12) Dropout OFF, Act: relu pool: max
Computational Cost: 4.1 Mega Ops
---
2DConv: input= (50, 16, 12, 12)      filter= (32, 16, 3, 3)
Output = (50, 32, 5, 5) Dropout OFF, Act: relu pool: max
Computational Cost: 23.0 Mega Ops
```

```
---
PerceptronLayer( #Inputs = 800 #Outputs = 300 )
Computational Cost: 12.0 Mega Ops
---
PerceptronLayer( #Inputs = 300 #Outputs = 300 )
Computational Cost: 4.5 Mega Ops
---
PerceptronLayer( #Inputs = 300 #Outputs = 10 )
Computational Cost: 150.0 kilo Ops
---
GLOBAL
Computational Cost: 43.8 Mega Ops
Total Count of trainable Parameters: 338410
Building Computational Graph took 0.030 s
Compiling output functions for nll target:
    using no class_weights
    using no example_weights
    using no lazy_labels
    label propagation inactive
```

A few comments on the expected output before training:

- There will be a warning that receptive fields are not centered (the neurons in the last conv layer lie spatially "between" the neurons of the input layer). This is ok because this training task does require localisation of objects. All local information is discarded anyway when the fully connected layers are put after the conv layers.
- The information of `elektronn2.net.netutils.CNNCalculator()` is printed first, i.e. the layer sizes, receptive fields etc.
- Although MNIST contains only 50000 training examples, it will print 200000 because of the shift augmentation, which is done when loading the data
- For image training, an auxiliary dimension for the (colour) channel is introduced.
- The input shape `(50, 1, 26, 26)` indicates that the batch size is 50, the number of channels is just 1 and the image extent is `26 x 26`.
- You can observe that the first layer outputs an image of size is `12 x 12`: the convolution with filter size 3 reduces 26 to 24, then the maxpooling by factor 2 reduces 24 to 12.
- After the last conv layer everything except the batch dimension is flattened to be feed into a fully connected layer: `32 x 5 x 5 == 800`. If the image extent is not sufficiently small before doing this (e.g. `10 x 10 == 100`) this will be a bottleneck and introduce **huge** weight matrices for the fully connected layer; more poolings must be used then.

Results & Comments

Important: This section is out of date and has to be revised for ELEKTRONN2

The values in the example file should give a good result after about 10-15 minutes on a recent GPU, but you are invited to play around with the network architecture and meta-parameters such as the learning rate. To watch the progress (in a nicer way than the reading the printed numbers on the console) go to the save directory and have a look at the plots. Every time a new line is printed in the console, the plot gets updated as well.

If you had not used warping the progress of the training would look like this:

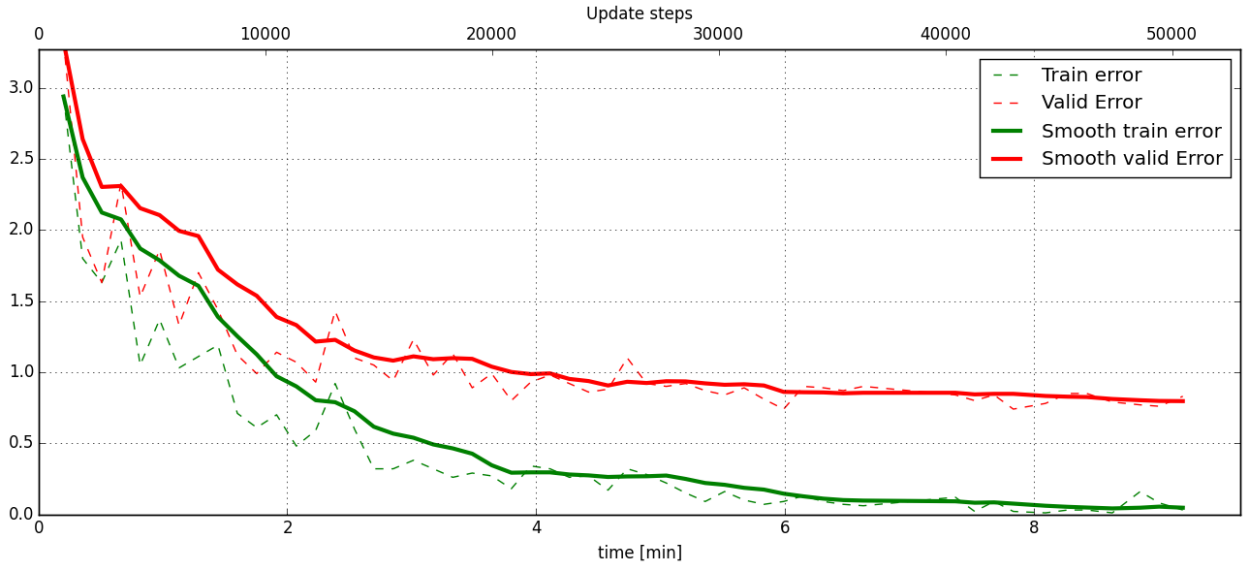

Fig. 1.3: Withing a few minutes the *training* error goes to 0 whereas the *validation* error stays on a higher level.

The spread between training and validation set (a partition of the data not presented as training examples) indicates a kind of over-fitting. But actually the over-fitting observed here is not as bad as it could be: because the training error is 0 the gradients are close to 0 - no weight updates are made for 0 gradient, so the training stops "automatically" at this point. For different data sets the training error might not reach 0 and weight updates are made all the time resulting in a validation error that goes **up** after some time - this would be real over-fitting.

A common regularisation technique to prevent over-fitting is drop out which is also implemented in ELEKETRONN. But since MNIST data are images, we want to demonstrate the use of warping instead in this example.

Warping makes the training goal more difficult, therefore the CNN has to learn its task "more thoroughly". This greatly reduces the spread between training and validation set. Training also takes slightly more time. And because the task is more difficult the training error will not reach 0 anymore. The validation error is also high during training, since the CNN is devoting resources to solving the difficult (warped) training set at the expense of generalization to "normal" data of the validation set.

The actual boost in (validation) performance comes when the warping is turned off and the training is fine-tuned with a smaller learning rate. Wait until the validation error approximately plateaus, then interrupt the training using `Ctrl+C`:

```
>>> data.warp_on = False # Turn off warping
>>> setlr 0.002          # Lower learning rate
>>> q                    # quit console to continue training
```

This stops the warping for further training and lowers the learning rate. The resulting training progress would look like this:

Because our decisions on the best learning rate and the best point to stop warping have been influenced by the validation set (we could somehow over-fit to the validation set), the actual performance is evaluated on a separate, third set, the *test* set (we should really only ever look at the test error when we have decided on a training setup/schedule, the test set is not meant to influence training at all).

Stop the training using `ctrl+c`:

```
>>> print self.testModel('test')
(<NLL>, <Errors>)
```

The result should be competitive - around 0.5% error, i.e. 99.5% accuracy.

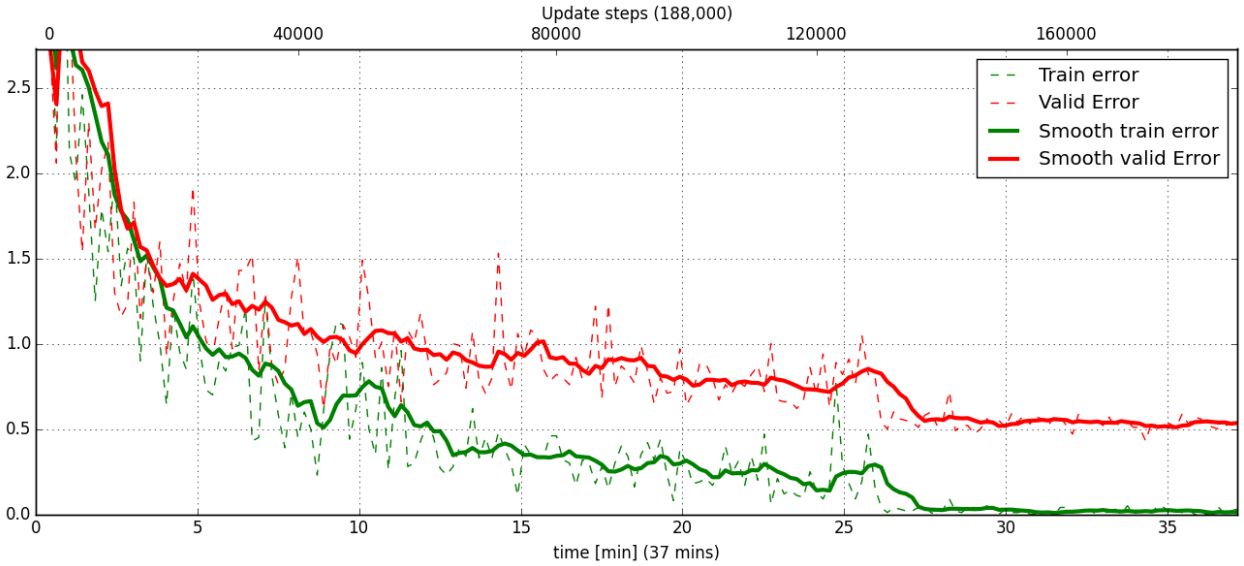

Fig. 1.4: The training was interrupted after ca. 130000 iterations. Turning off warping reduced both errors to their final level (after the gradient is 0 again, no progress can be made).

MLP with built-in Pipeline

Important: This section is out of date and has to be revised for ELEKTRONN2

In the spirit of the above example, MNIST can also be trained with a pure multi layer perceptron (MLP) without convolutions. The images are then just flattened vectors (–> *vect-scalar* mode). There is a config file `MNIST_MLP_config.py` in the `Examples` folder. This method can also be applied for any other non-image data, e.g. predicting income from demographic features.

Standalone CNN

Important: This section is out of date and has to be revised for ELEKTRONN2

If you think the big pipeline and long configuration file is a bit of an overkill for good old MNIST we have an alternative lightweight example in the file `MNIST_CNN_standalone.py` of the `Examples` folder. This example illustrates what (in a slightly more elaborate way) happens under the hood of the big pipeline.

First we import the required classes and initialise a training data object from `elektronn2.training.traindata` (which we actually used above, too). It does not more than loading the training, validation and testing data and sample batches randomly - all further options e.g. for augmentation are not used here:

```
from elektronn2.training.traindata import MNISTData
from elektronn2.net.convnet import MixedConvNN

data = MNISTData(path='~/devel/ELEKTRONN2/Examples/mnist.pkl',convert2image=True,␣
↪shift_augment=False)
```

Next we set up the Neural Network. Each method of `cnn` has much more options which are explained in the API doc. Start with similar code if you want to create customised NNs:

```
batch_size = 100
cnn = MixedConvNN((28,28),input_depth=1) # input_depth: only 1 gray channel (no RGB
↪or depth)
cnn.addConvLayer(10,5, pool_shape=2, activation_func="abs") # (nof, filtersize)
cnn.addConvLayer(8, 5, pool_shape=2, activation_func="abs")
cnn.addPerceptronLayer(100, activation_func="abs")
cnn.addPerceptronLayer(80, activation_func="abs")
cnn.addPerceptronLayer(10, activation_func="abs") # need 10 outputs as there are 10
↪classes in the data set
cnn.compileOutputFunctions()
cnn.setOptimizerParams(SGD={'LR': 1e-2, 'momentum': 0.9}, weight_decay=0) # LR:
↪learning rate
```

Finally, the training loop which applies weight updates in every iteration:

```
for i in range(5000):
    d, l = data.getbatch(batch_size)
    loss, loss_instance, time_per_step = cnn.trainingStep(d, l, mode="SGD")

    if i%100==0:
        valid_loss, valid_error, valid_predictions = cnn.get_error(data.valid_d, data.
↪valid_l)
        print("update:",i,"; Validation loss:",valid_loss, "Validation error:",valid_
↪error*100.,"%")

loss, error, test_predictions = cnn.get_error(data.test_d, data.test_l)
print "Test loss:",loss, "Test error:",error*100.,"%"
```

Of course the performance of this setup is not as good of the model above, but feel free tweak - how about dropout? Simply add `enable_dropout=True` to the cnn initialisation: all layers have by default a dropout rate of 0.5 - unless it is suppressed with `force_no_dropout=True` when adding a particular layer (it should not be used in the last layer). Don't forget to set the dropout rates to 0 while estimating the performance and to their old value afterwards (the methods `cnn.getDropoutRates` and `cnn.setDropoutRates` might be useful). Hint: for dropout, a different activation function than `abs`, more neurons per layer and more training iterations might perform better... you can try adapting it yourself or find a ready setup with drop out in the `examples` folder.

Auto encoder Example

Important: This section is out of date and has to be revised for ELEKTRONN2

This examples also uses MNIST data, but this time the task is not classification but compression. The input images have shape `28 x 28` but we will regard them as 784 dimensional vectors. The NN is shaped like an hourglass: the number of neurons decreases from 784 input neurons to 50 internal neurons in the central layer. Then the number increases symmetrically to 784 for the output. The training target is to reproduce the input in the output layer (i.e. the labels are identical to the data). Because the inputs are float numbers, so is the output and this is a regression problem. The first part of the auto encoder compresses the information and the second part decompresses it. The weights of both parts are shared, i.e. the weight matrix of each decompression layer is the transposed weight matrix of the corresponding compression layer, and updates are made simultaneously in both layers. For constructing an auto encoder the method `cnn.addTiedAutoencoderChain` is used.

```
import matplotlib.pyplot as plt

from elektronn2.training.traindata import MNISTData
from elektronn2.net.convnet import MixedConvNN
```

```
from elektronn2.net.introspection import embedMatricesInGray


# Load Data #
data = MNISTData(path='/docs/devel/ELEKTRONN2/elektronn2/examples/mnist.pkl',
↪convert2image=False, shift_augment=False)


# Load Data #
data = MNISTData(path='~/devel/ELEKTRONN2/Examples/mnist.pkl',convert2image=False,␣
↪shift_augment=False)

# Create Autoencoder #
batch_size = 100
cnn = MixedConvNN((28**2),input_depth=None)
cnn.addPerceptronLayer( n_outputs = 300, activation_func="tanh")
cnn.addPerceptronLayer( n_outputs = 200, activation_func="tanh")
cnn.addPerceptronLayer( n_outputs = 50, activation_func="tanh")
cnn.addTiedAutoencoderChain(n_layers=None, activation_func="tanh",input_noise=0.3,␣
↪add_layers_to_network=True)
cnn.compileOutputFunctions(target="regression")  #compiles the cnn.get_error function␣
↪as well
cnn.setOptimizerParams(SGD={'LR': 5e-1, 'momentum': 0.9}, weight_decay=0)

for i in range(10000):
    d, l = data.getbatch(batch_size)
    loss, loss_instance, time_per_step = cnn.trainingStep(d, d, mode="SGD")

    if i%100==0:
        print("update:",i,"; Training error:", loss)

loss,  test_predictions = cnn.get_error(data.valid_d, data.valid_d)

plt.figure(figsize=(14,6))
plt.subplot(121)
images = embedMatricesInGray(data.valid_d[:200].reshape((200,28,28)),1)
plt.imshow(images, interpolation='none', cmap='gray')
plt.title('Data')
plt.subplot(122)
recon = embedMatricesInGray(test_predictions[:200].reshape((200,28,28)),1)
plt.imshow(recon, interpolation='none', cmap='gray')
plt.title('Reconstruction')

cnn.saveParameters('AE-pretraining.param')
```

The above NN learns to compress the 784 pixels of an image to a 50 dimensional code (ca. 15x). The quality of the reconstruction can be inspected from plotting the images and comparing them to the original input:

The compression part of the auto encoder can be used to reduce the dimension of a data vector, while still preserving the information necessary to reconstruct the original data.

Often training data (e.g. lots of images of digits) are vastly available but nobody has taken the effort to create training labels for all of them. This is when auto encoders can be useful: train an auto encoder on the unlabelled data and use the learnt weights to initialise a NN for classification (aka pre-training).The classifcation NN does not have to learn a good internal data representation from scratch. To fine-tune the weights for classification (mainly in the additional output layer), only a small fraction of the examples must be labelled. To construct a pre-trained NN:

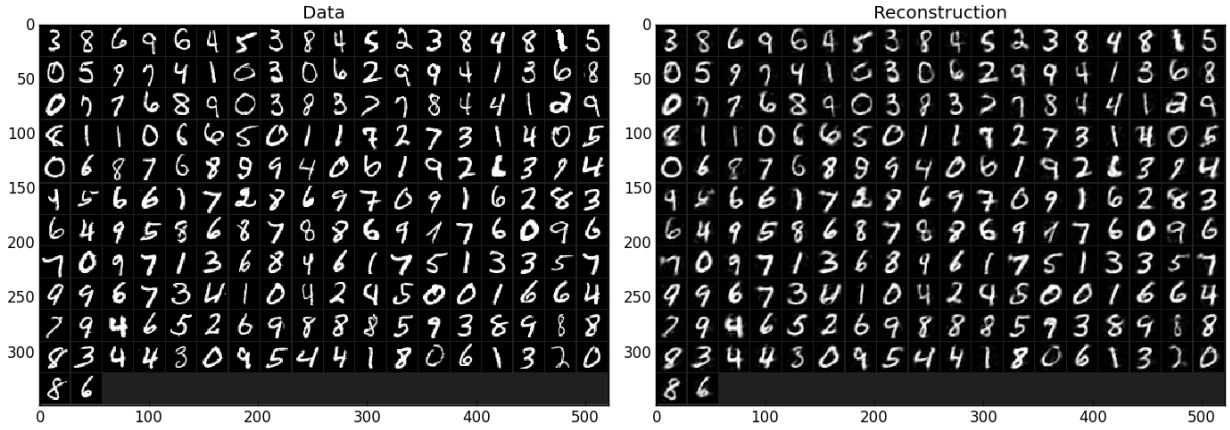

Fig. 1.5: Left input data (from validation set) and right reconstruction. The reconstruction values have been slightly rescaled for better visualisation.

```
cnn.saveParameters('AE-pretraining.param', layers=cnn.layers[0:3]) # save the␣
↪parameters for the compression part
cnn2 = MixedConvNN((28**2),input_depth=None) # Create a new NN
cnn2.addPerceptronLayer( n_outputs = 300, activation_func="tanh")
cnn2.addPerceptronLayer( n_outputs = 200, activation_func="tanh")
cnn2.addPerceptronLayer( n_outputs = 50, activation_func="tanh")
cnn2.addPerceptronLayer( n_outputs = 10, activation_func="tanh") # Add a layer for 10-
↪class classificaion
cnn2.compileOutputFunctions(target="nll")  #compiles the cnn.get_error function as␣
↪well # target function nll for classification
cnn2.setOptimizerParams(SGD={'LR': 0.005, 'momentum': 0.9}, weight_decay=0)
cnn2.loadParameters('AE-pretraining.param') # This overloads only the first 3 layers,
↪because the file contains only params for 3 layers

# Do training steps with the labels like
for i in range(10000):
    d, l = data.getbatch(batch_size)
    cnn2.trainingStep(d, l, mode="SGD")
```

RNN Example

Note: Coming soon

CHAPTER 2

API documentation

elektronn2.neuromancer package

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+'.

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*) – 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.

- **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*)

Maxpooling 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*) –
- **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**(**args*, ***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 actaully 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**(**args*, ***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:

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

class elektronn2.neuromancer.loss.**AggregateLoss**(**args*, ***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**(**args*, ***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**(**args*, ***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 linearliy for other targets)
- **name** (*str*) – Node name.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

class elektronn2.neuromancer.loss.**Softmax**(**args*, ***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**(**args*, ***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!
- **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**(**args*, ***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**(**args*, ***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**(**args*, ***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**(**args*, ***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**(**args*, ***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Euclidian 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**(**args*, ***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. Optimizers can be chosen by the kwarg `mode`. They are complied on demand (which may take a while) and cached

Signature: cnn.trainingStep(data, target(, *aux)(,**kwargs))

Parameters

- **data** (*floatX array*) – input [bs, ch (, z, y, x)]
- **targets** (*int16 array*) – [bs,((z,)y,x)] (optional)
- **other inputs** (*(optional)*) – depending in model
- **kwargs** –
  - **mode: string** ['SGD']: (default) Good if data set is big and redundant

    'RPROP': which does neither uses a fix learning rate nor the momentum-value. It is faster than SGD if you do full-batch Training and use NO dropout. Any source of noise leads to failure of convergence (at all).

    'CG': Good generalisation but requires large batches. Returns current loss always

    'LBFGS': http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin_l_bfgs_b.html

  - **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...)
- **time_per_step** (*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**(**args*, ***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) Initializes gamma parameter.
- **mean** – (For batch normalisation) Initializes mean parameter.
- **std** – (For batch normalisation) Initializes 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**(**args*, ***kwargs*)

Bases: *elektronn2.neuromancer.neural.Perceptron*

Convolutional layer with subsequent pooling.

Examples

Examples for constructing convolutional neural networks can be found in examples/3d_cnn.py and examples/numa_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*) – Size/shape of pooling after the convolution.

- **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.

- **activation_func** (*str*) – Activation function name.
- **mfp** (*bool*) – Whether to apply Max-Fragment-Pooling in this Layer.
- **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) Initializes gamma parameter.
- **mean** – (For batch normalisation) Initializes mean parameter.
- **std** – (For batch normalisation) Initializes std parameter.
- **gradnet_mode** –

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

Bases: *elektronn2.neuromancer.neural.Conv*

Upconvolution layer.

E.g. pooling + upconv with p=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/shape of pooling.
- **activation_func** (*str*) – Activation function name.
- **identity_init** (*bool*) – Initialise weights to result in pixel repetition upsampling
- **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) Initializes gamma parameter.
- **mean** – (For batch normalisation) Initializes mean parameter.
- **std** – (For batch normalisation) Initializes 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**(**args*, ***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**(**args*, ***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) Initializes gamma parameter.

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

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

class elektronn2.neuromancer.neural.**Pool**(**args*, ***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').
- **name** (*str*) – Name of the pooling layer.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.

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

class elektronn2.neuromancer.neural.**FaithlessMerge**(**args*, ***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**(**args*, ***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) Initializes gamma parameter.
- **mean** – (For batch normalisation) Initializes mean parameter.
- **std** – (For batch normalisation) Initializes std parameter.
- **gradnet_mode** –

class elektronn2.neuromancer.neural.**LRN**(**args*, ***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.**ImageAlign**(*hi_res*, *lo_res*, *hig_res_n_f*, *activation_func='relu'*, *identity_init=True*, *batch_normalisation=False*, *dropout_rate=0*, *name='upconv'*, *print_repr=True*, *w=None*, *b=None*, *gamma=None*, *mean=None*, *std=None*, *gradnet_mode=None*)

Try to automatically align and concatenate 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 Node's strides are at least twice as large as the strides of the high-res Node in any dimension.

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

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

- **hi_res** (*Node*) – Parent Node with high resolution output.
- **lo_res** (*Node*) – Parent Node with low resolution output.
- **hig_res_n_f** (*int*) – Number of filters for the aligning UpConv.
- **activation_func** (*str*) – (passed to new UpConv if required).
- **identity_init** (*bool*) – (passed to new UpConv if required).
- **batch_normalisation** (*bool*) – (passed to new UpConv if required).
- **dropout_rate** (*float*) – (passed to new UpConv if required).
- **name** (*str*) – Name of the intermediate UpConv node if required.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.
- **w** – (passed to new UpConv if required).
- **b** – (passed to new UpConv if required).
- **gamma** – (passed to new UpConv if required).
- **mean** – (passed to new UpConv if required).
- **std** – (passed to new UpConv if required).
- **gradnet_mode** – (passed to new UpConv if required).

Returns Concat Node that merges the aligned high-res and low-res outputs.

Return type *Concat*

elektronn2.neuromancer.neural.**UpConvMerge**(*hi_res*, *lo_res*, *hig_res_n_f*, *activation_func='relu'*, *identity_init=True*, *batch_normalisation=False*, *dropout_rate=0*, *name='upconv'*, *print_repr=True*, *w=None*, *b=None*, *gamma=None*, *mean=None*, *std=None*, *gradnet_mode=None*)

Try to automatically align and concatenate 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 Node's strides are at least twice as large as the strides of the high-res Node in any dimension.

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

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

- **hi_res** (Node) – Parent Node with high resolution output.
- **lo_res** (Node) – Parent Node with low resolution output.
- **hig_res_n_f** (*int*) – Number of filters for the aligning UpConv.
- **activation_func** (*str*) – (passed to new UpConv if required).
- **identity_init** (*bool*) – (passed to new UpConv if required).
- **batch_normalisation** (*bool*) – (passed to new UpConv if required).
- **dropout_rate** (*float*) – (passed to new UpConv if required).
- **name** (*str*) – Name of the intermediate UpConv node if required.
- **print_repr** (*bool*) – Whether to print the node representation upon initialisation.
- **w** – (passed to new UpConv if required).
- **b** – (passed to new UpConv if required).
- **gamma** – (passed to new UpConv if required).
- **mean** – (passed to new UpConv if required).
- **std** – (passed to new UpConv if required).
- **gradnet_mode** – (passed to new UpConv if required).

Returns Concat Node that merges the aligned high-res and low-res outputs.

Return type *Concat*

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

Models are built from the interplay of *node* 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 an one or several inputs, called *parent*, (unless it is an *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 like as 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 reuse in future calls.

Several properties (with respect to the sub-graph the node depends on, or only from the of the node itself) this 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 node must use this attribute of their inputs to perform their calculation and write the result 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 specified axis. Note that possible redundant computations in nodes are most likely eliminated by the theano graph optimiser.

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/<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 proabilites 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 imgae 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**(**args*, ***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**(**args*, ***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**(**args*, ***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**(**args*, ***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**(**args*, ***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**(**args*, ***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

(Optinally) 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**(**args*, ***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**(**args*, ***kwargs*)
Bases: *elektronn2.neuromancer.node_basic.Node*

Parameters

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

get_value()

elektronn2.neuromancer.optimiser module

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

repair_fuckup()

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

repair_fuckup()

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

repair_fuckup()

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_fuckup()

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*

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()

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*)

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()

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**(**args*, ***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**(**args*, ***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 inidcate 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`**(**args*, ***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`**(**args*, ***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*)

Module contents

elektronn2.training package

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
>>>     dosomehtingelse(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 0.95 every 1000 steps
>>> 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 to 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*)

Module contents

elektronn2.data package

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=[]*, *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=[]*, *r_max_scale=0.9*, *tracing_dir_prior_c=0.5*, *force_dense=False*, *flatfield_p=0.001*)

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*)

Bases: object

getbatch(*batch_size=1*, *source='train'*, *grey_augment_channels=[]*, *warp=False*, *warp_args=None*, *ignore_thresh=False*, *force_dense=False*, *affinities=False*, *nhood_targets=False*, *ret_ll_mask=False*)

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** (*string*) – Data set to draw data from: 'train'/'valid'
- **flip** (*Bool*) – If True examples are mirrored and rotated by 90 deg randomly
- **grey_augment_channels** (*list*) – List of channel indices to apply grey-value augmentation to
- **ret_ll_mask** (*Bool*) – If True additional information for reach batch example is returned. Currently implemented are two ll_mask arrays to indicate the targetling mode. The first dimension of those arrays is the batch_size!
- **warp_on** (*Bool/Float(0,1)*) – 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
- **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!

Returns

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

sample_warp_params = dict(sample_aniso = True, lock_z = False, no_x_flip = False, warp_amount=1.0, perspective=True)

warp_stats

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

getbatch(***get_batch_kwargs*)

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*)

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 terminatons (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(**args0*, ***kwargs0*)

get_hull_branch_dist_cutoff(**args0*, ***kwargs0*)

get_hull_points_inner(**args0*, ***kwargs0*)

get_hull_skel_direc_rel(**args0*, ***kwargs0*)

get_kdtree(**args0*, ***kwargs0*)

get_knn(**args0*, ***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 neigbours, 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 shit

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*, *cutoff_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*)

Parameters

- **img** – (f, z, x, y)
- **ps** – (spatial only) patch_size (z,x,y)
- **M** – forward tansform, must contain translations in source and target array!
- **target** – optional target array to be extracted in the same way
- **target_ps** –
- **target_vec_ix** – list of triples that denote vector value parts in the target array e.g. [(0,1,2),(4,5,6)] denotes two vectorfields separated by a scalar field in channel 3

Returns

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*)

Module contents

elektronn2.utils package

Subpackages

elektronn2.utils.d3viz package

Submodules

elektronn2.utils.d3viz.formatting module

Visualisiation 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
dict – Color table of node types.

apply_colors
dict – Color table of apply nodes.

shapes
dict – Shape table of node types.

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`.

Module contents

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 of int*) – Desired patch_size size(s). If `None` a range of suggestions can be found in the attribute `valid_patch_sizes`
- **mfp** (*list of int/{0,1}*) – 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** (*int or list of 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   = [211, 211, 20]
>>> filters          = [[6,6,1], [4,4,4], [2,2,2], [1,1,1]]
>>> pool             = [[2,2,1], [2,2,2], [2,2,2], [1,1,1]]
>>> mfp              = [1,        1,       0,       0,   ]
>>> ndim=3
>>> d = cnncalculator(filters, pool, desired_patch_size, mfp=mfp, force_
↪center=True, desired_output=None, ndim=ndim)
Info: patch_size (211) changed to (210) (size not possible)
Info: patch_size (211) changed to (210) (size not possible)
Info: patch_size (20) changed to (22) (size too small)
>>> print(d)
patch_size: [210, 210, 22]
Layer/Fragment sizes:     [[102, 49, 24, 24], [102, 49, 24, 24], [22, 9, 4, 4]]
Unpooled Layer sizes:     [[205, 99, 48, 24], [205, 99, 48, 24], [22, 19, 8, 4]]
Receptive fields: [[7, 15, 23, 23], [7, 15, 23, 23], [1, 5, 9, 9]]
Strides:          [[2, 4, 8, 8], [2, 4, 8, 8], [1, 2, 4, 4]]
Overlap:          [[5, 11, 15, 15], [5, 11, 15, 15], [0, 3, 5, 5]]
Offset:           [11.5, 11.5, 4.5].
    If offset is non-int: floor(offset).
    Select labels from within img[offset-x:offset+x]
    (non-int means, output neurons lie centered on patch_size neurons,
    i.e. they have an odd field of view)
```

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.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 interation 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.**my_jit**(**args*, ***kwargs*)
Bases: elektronn2.utils.utils_basic.DecoratorBase

This mock decorator is used as a pure-Python fallback for `numba.jit` if numba is not availabe.

If numba is available, the decorator is later replaced by the real numba code.

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*)

Module contents

elektronn2.malis package

Submodules

elektronn2.malis.malis_utils module

elektronn2.malis.malis_utils.**compute_V_rand_N2**(*seg_true*, *seg_pred*)
Computes Rand index of `seg_pred` w.r.t `seg_true`. Small is better!!! The input arrays both contain label IDs and may be of arbitrary, but equal, shape.

Pixels which are have ID in the true segmentation are not counted!

Parameters:

seg_true: np.ndarray True segmentation, IDs

seg_pred: np.ndarray Predicted segmentation

Returns **ri**

Return type

???

elektronn2.malis.malis_utils.**mknhood2d**(*radius=1*)
Makes nhood structures for some most used dense graphs

elektronn2.malis.malis_utils.**mknhood3d**(*radius=1*)
Makes nhood structures for some most used dense graphs. The neighborhood reference for the dense graph representation we use nhood(1,:) is a 3 vector that describe the node that conn(:,:,:,1) connects to so to use it: conn(23,12,42,3) is the edge between node [23 12 42] and [23 12 42]+nhood(3,:) See? It's simple! nhood is just the offset vector that the edge corresponds to.

elektronn2.malis.malis_utils.**mknhood3d_aniso**(*radiusxy=1*, *radiusxy_zminus1=1.8*)
Makes nhood structures for some most used dense graphs.

elektronn2.malis.malis_utils.**bmap_to_affgraph**(*bmap*, *nhood*)
Construct an affinity graph from a boundary map

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.

Parameters

- **bmap** (*3d np.ndarray, int*) – Volume of boundaries 0: object interior, 1: boundaries / ECS
- **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

Returns **aff** – Affinity graph of shape (#edges, x, y, z) 1: connected, 0: disconnected

Return type 4d np.ndarray int32

elektronn2.malis.malis_utils.**seg_to_affgraph**(*seg_gt*, *nhood*)
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.

Parameters

- **seg_gt** (*3d np.ndarray, int (any precision)*) – Volume of segmentation IDs
- **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

Returns **aff** – Affinity graph of shape (#edges, x, y, z) 1: connected, 0: disconnected

Return type 4d np.ndarray int16

elektronn2.malis.malis_utils.**watershed_from_affgraph**(*aff*, *seeds*, *nhood*)

elektronn2.malis.malis_utils.**bmappred_to_affgraph**(*pred*, *nhood*)
Construct an affinity graph from boundary predictions

Parameters

- **pred** (*3d np.ndarray*) – Volume of boundary predictions
- **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

Returns **aff** – Affinity graph of shape (#edges, x, y, z) 1: connected, 0: disconnected

Return type 4d np.ndarray int16

elektronn2.malis.malisop module

elektronn2.malis.malisop.**malis_weights**(*affinity_pred*, *affinity_gt*, *seg_gt*, *nhood*, *unrestrict_neg=False*)
Computes MALIS loss weights

Roughly speaking the malis weights quantify the impact of an edge in the predicted affinity graph on the resulting segmentation.

Parameters

- **affinity_pred** (*4d np.ndarray float32*) – Affinity graph of shape (#edges, x, y, z) 1: connected, 0: disconnected
- **affinity_gt** (*4d np.ndarray int16*) – Affinity graph of shape (#edges, x, y, z) 1: connected, 0: disconnected
- **seg_gt** (*3d np.ndarray, int (any precision)*) – Volume of segmentation IDs
- **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
- **unrestrict_neg** (*Bool*) – Use this to relax the restriction on neg_counts. The restriction modifies the edge weights for before calculating the negative counts as:

`edge_weights_neg = np.maximum(affinity_pred, affinity_gt)` If unrestricted the predictions are used directly.

Returns

- **pos_counts** (*4d np.ndarray int32*) – Impact counts for edges that should be 1 (connect)
- **neg_counts** (*4d np.ndarray int32*) – Impact counts for edges that should be 0 (disconnect)

•Computes for all pixel-pairs the MaxiMin-Affinity

•Separately for pixel-pairs that should/should not be connected

•Every time an affinity prediction is a MaxiMin-Affinity its weight is incremented by one in the output matrix (in different slices depending on whether that that pair should/should not be connected)

Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex

Python Module Index

e

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z