DeepCpG Documentation

Release 1.0.7

DeepCpG

Dec 03, 2018

Contents

DeepCpG[1] is a deep neural network for predicting the methylation state of CpG dinucleotides in multiple cells. It allows to accurately impute incomplete DNA methylation profiles, to discover predictive sequence motifs, and to quantify the effect of sequence mutations. (Angermueller et al, 2017).

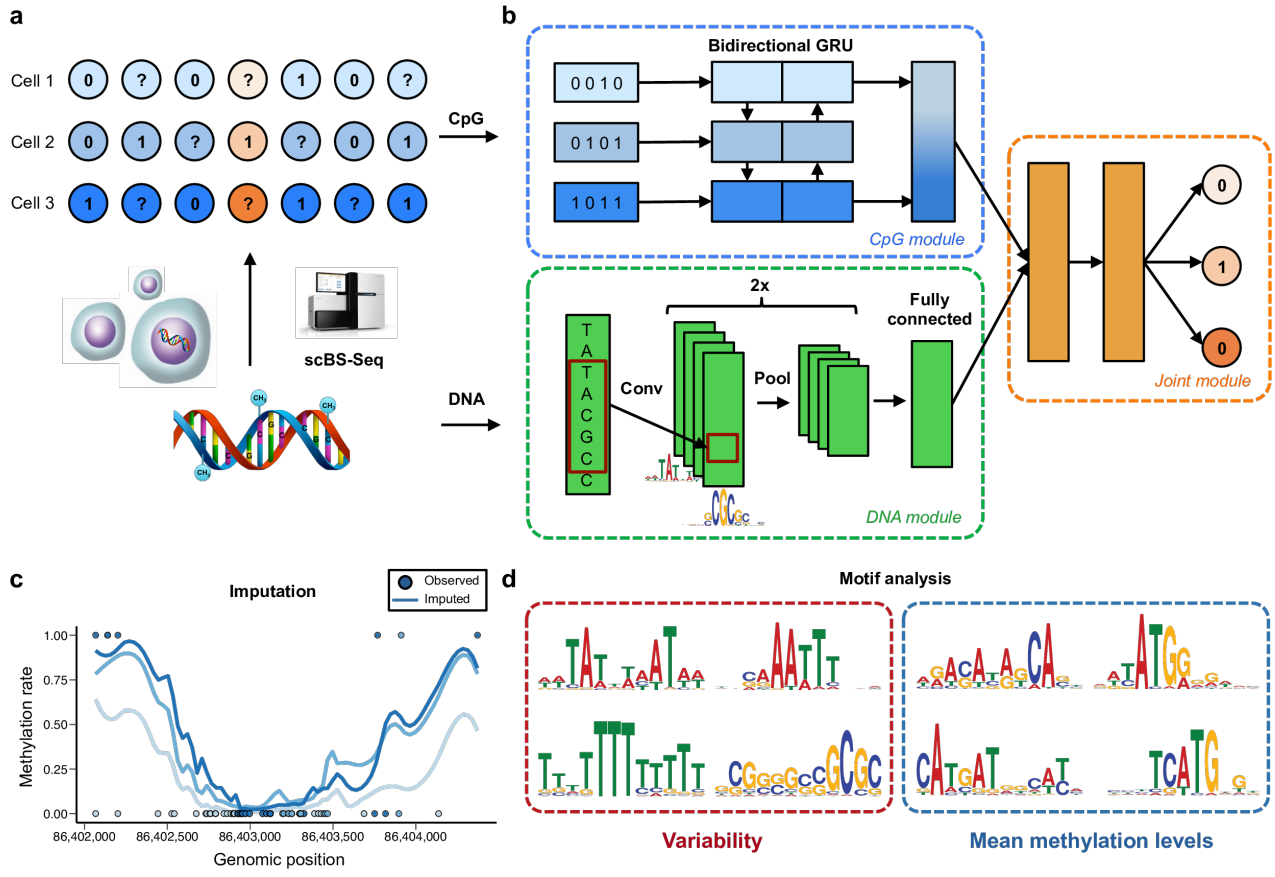

Fig. 1: **DeepCpG model architecture and applications.**
(a) Sparse single-cell CpG profiles as obtained from scBS-seq or scRRBS-seq. Methylated CpG sites are denoted by ones, unmethylated CpG sites by zeros, and question marks denote CpG sites with unknown methylation state (missing data). (b) DeepCpG model architecture. The DNA model consists of two convolutional and pooling layers to identify predictive motifs from the local sequence context, and one fully connected layer to model motif interactions. The CpG model scans the CpG neighborhood of multiple cells (rows in b), using a bidirectional gated recurrent network (GRU), yielding compressed features in a vector of constant size. The Joint model learns interactions between higher-level features derived from the DNA- and CpG model to predict methylation states in all cells. (c, d) The trained DeepCpG model can be used for different downstream analyses, including genome-wide imputation of missing CpG sites (c) and the discovery of DNA sequence motifs that are associated with DNA methylation levels or cell-to-cell variability (d).

[1] Angermueller, Christof, Heather J. Lee, Wolf Reik, and Oliver Stegle. *DeepCpG: Accurate Prediction of Single-Cell DNA Methylation States Using Deep Learning.* Genome Biology 18 (April 11, 2017): 67. doi:10.1186/s13059-017-1189-z.

CHAPTER 1

Installation

The easiest way to install DeepCpG is to use `PyPI`:

```
pip install deepcpg
```

Alternatively, you can checkout the repository

```
git clone https://github.com/cangermueller/deepcpg.git
```

and then install DeepCpG using `setup.py`:

```
python setup.py install
```

CHAPTER 2

Examples

Interactive examples on how to use DeepCpG can be found here.

CHAPTER 3

Documentation

- *Data creation* – Creating and analyzing data.
- *Model training* – Training DeepCpG models.
- *Model architectures* – Description of DeepCpG model architectures.
- *Scripts* – Documentation of DeepCpG scripts.
- *Library* – Documentation of DeepCpG library.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

4.1 Data creation

This tutorial describes how to create and analyze the input data of DeepCpG.

4.1.1 Creating data

`dcpg_data.py` creates the data for model training and evaluation, given multiple methylation profiles:

```
dcpg_data.py
    --cpg_profiles ./cpg/*.tsv
    --dna_files mm10/*.dna.*.fa.gz
    --cpg_wlen 50
    --dna_wlen 1001
    --out_dir ./data
```

`--cpg_profiles` specifies a list of files that store the observed CpG methylation states for either a bulk- or single-cell methylation profile. Supports bedGraph and TSV files.

BedGraph files must start with `track type=bedGraph`. Each following line represents the methylation state of a CpG site, for example:

```
track type=bedGraph
chr1    3007532 3007533 1.0
chr1    3007580 3007581 0.4
chr1    3012096 3012097 1.0
chr1    3017509 3017510 0.1
```

The columns have the following meaning:

- Column 1: the chromosome of the CpG site starting with `chr`.
- Column 2: the location of the C of the CpG site. Positions are enumerated starting at one.
- Column 3: the position of the G of the CpG site.
- Column 4: the observed methylation value ($\in (0; 1)$) of the CpG site. If all values are binary, i.e. either zero or one, they will also be stored by DeepCpG as binary values, which reduces disk usage. Continuous values are required for representing hemi-methylation or bulk methylation profiles.

TSV files do not start with a track column and only contain three columns, for example:

```
chr1    3007532 1.0
chr1    3007580 0.4
chr1    3012096 1.0
chr1    3017509 0.1
```

`--cpg_profiles` files can be gzip-compressed (`*.gz`) to reduce disk usage.

`--dna_files` specifies a list of FASTA files, where each file stores the DNA sequence of a particular chromosome. Files can be downloaded from Ensembl, e.g. mm10 for mouse or hg38 for human, and specified either via a glob pattern, e.g. `--dna_files mm10/*.dna.*fa.gz` or simply by the directory name, e.g. `--dna_files mm10`. The argument `--dna_files` is not required for imputing methylation states from neighboring methylation states without using the DNA sequence.

`--cpg_wlen` specifies the sum of CpG sites to the left and right of the target site that DeepCpG will use for making predictions. For example, DeepCpG will use 25 CpG sites to the left and right of the target CpG site using `--cpg_wlen 50`. A value of about 50 usually covers a wide methylation context and is sufficient to achieve a good performance. If you are dealing with many cells, I recommend using a smaller value to reduce disk usage.

`--dna_wlen` specifies the width of DNA sequence windows in base pairs that are centered on the target CpG site. Wider windows usually improve prediction accuracy but increase compute- and storage costs. I recommend `--dna_wlen 1001`.

These are the most important arguments for imputing methylation profiles. `dcpg_data.py` provides additional arguments for debugging and predicting statistics across profiles, e.g. the mean methylation rate or cell-to-cell variance.

Debugging

For debugging, testing, or reducing compute costs, `--chromos` can be used the select certain chromosomes. `--nb_sample_chromo` randomly samples a certain number of CpG sites from each chromosome, and `--nb_sample` specifies the maximum number of CpG sites in total.

Predicting statistics

For predicting statistics across methylation profiles, `--cpg_stats` and `--win_stats` can be used. These arguments specify a list of statistics that are computed across profiles for either a single CpG site or in windows of length `--win_stats_wlen` that are centered on a CpG site. Following statistics are supported:

- `mean`: the mean methylation rate.
- `mode`: the mode of methylation rates.
- `var`: the cell-to-cell variance.
- `cat_var`: three categories of cell-to-cell variance, i.e. low, medium, or high variance.
- `cat2_var`: two categories of cell-to-cell variance, i.e. low or high variance.

- `entropy`: the entropy across cells.
- `diff`: if a CpG site is differentially methylated, i.e. methylated in one profile but zero in others.
- `cov`: the CpG coverage, i.e. the number of profiles for which the methylation state of the target CpG site is observed.

Per-CpG statistics specified by `--cpg_stats` are computed only for CpG sites that are covered by at least `--cpg_stats_cov` (default 3) cells. Increasing `--cpg_stats_cov` will lead to more robust estimates.

Common issues

Why am I getting warnings 'No CpG site at position X!' when using ''dcpg_data.py''?

This means that some sites in `--cpg_profile` files are not CpG sites, i.e. there is no CG dinucleotide at the given position in the DNA sequence. Make sure that `--dna_files` point to the correct genome and CpG sites are correctly aligned. Since DeepCpG currently does not support allele-specific methylation, data from different alleles must be merged (recommended) or only one allele be used.

4.1.2 Computing data statistics

`dcpg_data_stats.py` enables to compute statistics for a list of DeepCpG input files:

```
dcpg_data_stats.py ./data/c1_000000-001000.h5 ./data/c13_000000-001000.h5
```

```
           output  nb_tot  nb_obs  frac_obs      mean       var
0  cpg/BS27_1_SER    2000     391    0.1955  0.790281  0.165737
1  cpg/BS27_3_SER    2000     408    0.2040  0.740196  0.192306
2  cpg/BS27_5_SER    2000     393    0.1965  0.692112  0.213093
3  cpg/BS27_6_SER    2000     402    0.2010  0.666667  0.222222
4  cpg/BS27_8_SER    2000     408    0.2040  0.776961  0.173293
```

The columns have the following meaning:

- `output`: The name of the target cell.
- `nb_tot`: The total number of CpG sites.
- `nb_obs`: The number of CpG sites for which the true label of `output` is observed.
- `frac_obs`: The fraction `nb_obs/nb_tot` of observed CpG sites.
- `mean`: The mean of `output`, e.g. the mean methylation rate.
- `var`: The variance of `output`, e.g. the variance in CpG methylation levels.

`--nb_tot` and `--nb_obs` are particularly useful for deciding how to split the data into a training, test, validation set as explained in the *training tutorial*. Statistics can be written to a TSV file using `--out_tsv` and be visualized using `--out_plot`.

4.1.3 Printing data

`dcpg_data_show.py` enables to selectively print the content of a list of DeepCpG data files. Using `--outputs` prints all DeepCpG model outputs in a selected region:

```
dcpg_data_show.py ./data/c1_000000-001000.h5 --chromo 1 --start  189118909 --end␣
↪189867450 --outputs
```

```
      loc                   outputs
    chromo        pos cpg/BS27_1_SER cpg/BS27_3_SER cpg/BS27_5_SER cpg/BS27_6_SER cpg/
↪BS27_8_SER
950      1  189118909             -1             -1              1             -1
↪         -1
951      1  189314906             -1             -1              1             -1
↪         -1
952      1  189506185              1             -1             -1             -1
↪         -1
953      1  189688256             -1              0             -1             -1
↪         -1
954      1  189688274             -1             -1             -1             -1
↪          0
955      1  189699529             -1             -1             -1              1
↪         -1
956      1  189728263             -1             -1              0             -1
↪         -1
957      1  189741539             -1              1             -1             -1
↪         -1
958      1  189850865             -1             -1             -1              1
↪         -1
959      1  189867450             -1              1             -1             -1
↪         -1
```

`-1` indicates unobserved methylation states. If `--outputs` is followed by a list of output names, only they will be printed. The arguments `--cpg`, `--cpg_wlen`, and `--cpg_dist` control how many (`--cpg_wlen`) neighboring methylation states (`--cpg`) and corresponding distances (`--cpg_dist`) are printed. For example, the following commands prints the state and distance of four neighboring CpG sites of cell *BS27_1_SER*:

```
dcpg_data_show.py ./data/c1_000000-001000.h5 --chromo 1 --start  189118909 --end␣
↪189867450 --outputs cpg/BS27_1_SER --cpg BS27_1_SER --cpg_wlen 4 --cpg_dist
```

```
      loc                   outputs BS27_1_SER/state           BS27_1_SER/dist
    chromo        pos cpg/BS27_1_SER              -2 -1 +1 +2              -2
↪-1        +1        +2
950      1  189118909             -1               1  1  1  1         84023.0
↪65557.0  114153.0  373437.0
951      1  189314906             -1               1  1  1  1        261554.0
↪81844.0  177440.0  191279.0
952      1  189506185              1               1  1  1  0        273123.0
↪13839.0  162360.0  662239.0
953      1  189688256             -1               1  1  0  1        182071.0
↪19711.0  480168.0  705968.0
954      1  189688274             -1               1  1  0  1        182089.0
↪19729.0  480150.0  705950.0
955      1  189699529             -1               1  1  0  1        193344.0
↪30984.0  468895.0  694695.0
956      1  189728263             -1               1  1  0  1        222078.0
↪59718.0  440161.0  665961.0
957      1  189741539             -1               1  1  0  1        235354.0
↪72994.0  426885.0  652685.0
958      1  189850865             -1               1  1  0  1        344680.0
↪182320.0  317559.0  543359.0
959      1  189867450             -1               1  1  0  1        361265.0
↪198905.0  300974.0  526774.0
```

Analogously, `--dna_wlen` prints the DNA sequence window of that length centered on the target CpG sites:

```
dcpg_data_show.py ./data/c1_000000-001000.h5 --chromo 1 --start  189118909 --end␣
↪189867450 --outputs cpg/BS27_1_SER --dna_wlen 11
```

```
      loc                   outputs dna
    chromo        pos cpg/BS27_1_SER  -5 -4 -3 -2 -1  0 +1 +2 +3 +4 +5
950      1  189118909             -1   2  1  0  0  0  3  2  2  0  0  3
951      1  189314906             -1   3  1  3  3  2  3  2  3  0  1  3
952      1  189506185              1   0  3  3  3  0  3  2  2  2  0  1
953      1  189688256             -1   2  3  3  2  2  3  2  2  3  2  2
954      1  189688274             -1   2  3  0  2  0  3  2  1  3  2  2
955      1  189699529             -1   2  3  2  2  0  3  2  3  1  1  1
956      1  189728263             -1   3  1  3  3  3  3  2  2  3  3  2
957      1  189741539             -1   2  0  2  1  2  3  2  1  2  2  3
958      1  189850865             -1   2  2  3  2  2  3  2  2  3  2  2
959      1  189867450             -1   3  1  3  0  3  3  2  1  2  3  0
```

With `--out_hdf`, the selected data can be stored as Pandas data frame to a HDF5 file.

4.2 Model training

Here you can find information about how to train DeepCpG models.

4.2.1 Splitting data into training, validation, and test set

For comparing different models, it is necessary to train, select hyper-parameters, and test models on distinct data. In holdout validation, the dataset is split into a training set (~60% of the data), validation set (~20% of the data), and test set (~20% of the data). Models are trained on the training set, hyper-parameters selected on the validation set, and the selected models compared on the test set. For example, you could use chromosome 1-5, 7, 9, 11, 13 as training set, chromosome 14-19 as validation set, and chromosome 6, 8, 10, 12, 14 as test set:

```
train_files="$data_dir/c{1,2,3,4,5,7,9,11,13}_*.h5
val_files="$data_dir/c{14,15,16,17,18,19}_*.h5"
test_files="$data_dir/c{6,8,10,12,14}_*.h5"

dcpg_train.py
    $train_files
    --val_file $val_files
    ...
```

As you can see, DeepCpG allows to easily split the data by glob patterns. You do not have to split the dataset by chromosomes. For example, you could use `train_files=$data_Dir/c*_[01].h5` to select all data files starting with index 0 or 1 for training, and use the remaining files for validation.

If you are not concerned about comparing DeepCpG with other models, you do not need a test set. In this case, you could, for example, leave out chromosome 14-19 as validation set, and use the remaining chromosomes for training.

If your data were generated using whole-genome scBS-seq, then the number of CpG sites on few chromosomes is usually already sufficient for training. For example, chromosome 1, 3, and 5 from *Smallwood et al (2014)* cover already more than 3 million CpG sites. I found about 3 million CpG sites as sufficient for training models without overfitting. However, if you are working with scRRBS-seq data, you probably need more chromosomes for training. To check how many CpG sites are stored in a set of DeepCpG data files, you can use the `dcpg_data_stats.py`. The following command will compute different statistics for the training set, including the number number of CpG sites:

```
dcpg_data_stats.py $data_dir/$train_files
```

```
################################
dcpg_data_stats.py ./data/c19_000000-032768.h5 ./data/c19_032768-050000.h5
################################
           output  nb_tot  nb_obs  frac_obs      mean       var
0  cpg/BS27_1_SER   50000   20621   0.41242  0.665972  0.222453
1  cpg/BS27_3_SER   50000   13488   0.26976  0.573102  0.244656
2  cpg/BS27_5_SER   50000   25748   0.51496  0.529633  0.249122
3  cpg/BS27_6_SER   50000   17618   0.35236  0.508117  0.249934
4  cpg/BS27_8_SER   50000   16998   0.33996  0.661019  0.224073
```

For each output cell, `nb_tot` is the total number of CpG sites, `nb_obs` the number of CpG sites with known methylation state, `frac_obs` the ratio between `nb_obs` and `nb_tot`, `mean` the mean methylation rate, and `var` the variance of the methylation rate.

4.2.2 Training DeepCpG models jointly

As described in Angermueller et al (2017), DeepCpG consists of a DNA, CpG, and Joint model. The DNA model recognizes features in the DNA sequence window that is centered on a target site, the CpG model recognizes features in observed neighboring methylation states of multiple cells, and the Joint model integrates features from the DNA and CpG model and predicts the methylation state of all cells.

The easiest way is to train all models jointly:

```
dcpg_train.py
    $train_files
    --val_files $val_files
    --dna_model CnnL2h128
    --cpg_model RnnL1
    --out_dir $models_dir/joint
    --nb_epoch 30
```

`--dna_model`, `--cpg_model`, and `--joint_model` specify the architecture of the DNA, CpG, and Joint model, respectively, which are described in *here <./models.rst>*_.

4.2.3 Training DeepCpG models separately

Although it is convenient to train all models jointly by running only a single command as described above, I suggest to train models separately. First, because it enables to train the DNA and CpG model in parallel on separate machines and thereby to reduce the training time. Second, it enables to compare how predictive the DNA model is relative to CpG model. If you think the CpG model is already accurate enough alone, you might not need the DNA model. Thirdly, I obtained better results by training the models separately. However, this may not be true for your particular dataset.

You can train the CpG model separately by only using the `--cpg_model` argument, but not `--dna_model`:

```
dcpg_train.py
    $train_files
    --val_files $val_files
    --dna_model CnnL2h128
    --out_dir $models_dir/dna
    --nb_epoch 30
```

You can train the DNA model separately by only using `--dna_model`:

```
dcpg_train.py
    $train_files
    --val_files $val_files
    --cpg_model RnnL1
    --out_dir $models_dir/cpg
    --nb_epoch 30
```

After training the CpG and DNA model, we are joining them by specifying the name of the Joint model with `--joint_model`:

```
dcpg_train.py
    $train_files
    --val_files $val_files
    --dna_model $models_dir/dna
    --cpg_model $models_dir/cpg
    --joint_model JointL2h512
    --train_models joint
    --out_dir $models_dir/joint
    --nb_epoch 10
```

`--dna_model` and `--cpg_model` point to the output training directory of the DNA and CpG model, respectively, which contains their specification and weights:

```
ls $models_dir/dna
```

```
events.out.tfevents.1488213772.lawrence model.json
lc_train.csv                            model_weights_train.h5
lc_val.csv                              model_weights_val.h5
model.h5
```

`model.json` is the specification of the trained model, `model_weights_train.h5` the weights with the best performance on the training set, and `model_weights_val.h5` the weights with the best performance on the validation set. `--dna_model ./dna` is equivalent to using `--dna_model ./dna/model.json ./dna/model_weights_val.h5`, i.e. the validation weights will be used. The training weights can be used by `--dna_model ./dna/model.json ./dna/model_weights_train.h5`

In the command above, we used `--train_models joint` to only train the parameters of the Joint model without training the pre-trained DNA and CpG model. Although this reduces training time, you might obtain better results by also fine-tuning the parameters of the DNA and CpG model without using `--train_models`.

4.2.4 Monitoring training progress

To check if your model is training correctly, you should monitor the training and validation loss. DeepCpG prints the loss and performance metrics for each output to the console as you can see from the previous commands. `loss` is the loss on the training set, `val_loss` the loss on the validation set, and `cpg/X_acc`, is, for example, the accuracy for output cell X. DeepCpG also stores these metrics in `X.csv` in the training output directory.

Both the training loss and validation loss should continually decrease until saturation. If at some point the validation loss starts to increase while the training loss is still decreasing, your model is overfitting the training set and you should stop training. DeepCpG will automatically stop training if the validation loss does not increase over the number of epochs that is specified by `--early_stopping` (by default 5). If your model is overfitting already after few epochs, your training set might be to small, and you could try to regularize your model model by choosing a higher value for `--dropout` or `--l2_decay`.

If your training loss fluctuates or increases, then you should decrease the learning rate. For more information on interpreting learning curves I recommend this tutorial.

To stop training before reaching the number of epochs specified by `--nb_epoch`, you can create a *stop file* (default name `STOP`) in the training output directory with `touch STOP`.

Watching numeric console outputs is not particular user friendly. TensorBoard provides a more convenient and visually appealing way to mointor training. You can use TensorBoard provided that you are using the *Tensorflow backend*. Simply go to the training output directory and run `tensorboard --logdir .`.

4.2.5 Deciding how long to train

The arguments `--nb_epoch` and `--early_stopping` control how long models are trained.

`--nb_epoch` defines the maximum number of training epochs (default 30). After one epoch, the model has seen the entire training set once. The time per epoch hence depends on the size of the training set, but also on the complexity of the model that you are training and the hardware of your machine. On a large dataset, you have to train for fewer epochs than on a small dataset, since your model will have seen already a lot of training samples after one epoch. For training on about 3,000,000 samples, good default values are 20 for the DNA and CpG model, and 10 for the Joint model.

Early stopping stops training if the loss on the validation set did not improve after the number of epochs that is specified by `--early_stopping` (default 5). If you are training without specifying a validation set with `--val_files`, early stopping will be deactivated.

`--max_time` sets the maximum training time in hours. This guarantees that training terminates after a certain amount of time regardless of the `--nb_epoch` or `--early_stopping` argument.

`--stop_file` defines the path of a file that, if it exists, stop training after the end of the current epoch. This is useful if you are monitoring training and want to terminate training manually as soon as the training loss starts to saturate regardless of `--nb_epoch` or `--early_stopping`. For example, when using `--stop_file ./train/STOP`, you can create an empty file with `touch ./train/STOP` to stop training at the end of the current epoch.

4.2.6 Optimizing hyper-parameters

DeepCpG has different hyper-parameters, such as the learning rate, dropout rate, or model architectures. Although the performance of DeepCpG is relatively robust to different hyper-parameters, you can tweak performances by trying out different parameter combinations. For example, you could train different models with different parameters on a subset of your data, select the parameters with the highest performance on the validation set, and then train the full model.

The following hyper-parameters are most important (default values shown): 1. Learning rate: `--learning_rate 0.0001` 2. Dropout rate: `--dropout 0.0` 3. DNA model architecture: `--dna_model CnnL2h128` 4. Joint model architecture: `--joint_model JointL2h512` 5. CpG model architecture: `--cpg_model RnnL1` 6. L2 weight decay: `--l2_decay 0.0001`

The learning rate defines how aggressively model parameters are updated during training. If the training loss *changes only slowly*, you could try increasing the learning rate. If your model is overfitting of if the training loss fluctuates, you should decrease the learning rate. Reasonable values are 0.001, 0.0005, 0.0001, 0.00001, or values in between.

The dropout rate defines how strongly your model is regularized. If you have only few data and your model is overfitting, then you should increase the dropout rate. Reasonable values are, e.g., 0.0, 0.2, 0.4.

DeepCpG provides different architectures for the DNA, CpG, and joint model. Architectures are more or less complex, depending on how many layers and neurons say have. More complex model might yield better performances, but take longer to train and might overfit your data. You can find more information about available model architecture *here*.

L2 weight decay is an alternative to dropout for regularizing model training. If your model is overfitting, you might try 0.001, or 0.005.

4.2.7 Testing training

`dcpg_train.py` provides different arguments that allow to briefly test training before training the full model for a about a day.

`--nb_train_sample` and `--nb_val_sample` specify the number of training and validation samples. When using `--nb_train_sample 500`, the training loss should briefly decay and your model should start overfitting.

`--nb_output` and `--output_names` define the maximum number and the name of model outputs. For example, `--nb_output 3` will train only on the first three outputs, and `--output_names cpg/.*SER.*` only on outputs that include 'SER' in their name.

Analogously, `--nb_replicate` and `--replicate_name` define the number and name of cells that are used as input to the CpG model. `--nb_replicate 3` will only use observed methylation states from the first three cells, and allows to briefly test the CpG model.

`--dna_wlen` specifies the size of DNA sequence windows that will be used as input to the DNA model. For example, `--dna_wlen 101` will train only on windows of size 101, instead of using the full window length that was specified when creating data files with `dcpg_data.py`.

Analogously, `--cpg_wlen` specifies the sum of the number of observed CpG sites to the left and the right of the target CpG site for training the CpG model. For example, `--cpg_wlen 10` will use 5 observed CpG sites to the left and to the right.

4.2.8 Fine-tuning and training selected components

`dcpg_train.py` provides different arguments that allow to selectively train only some components of a model.

With `--fine_tune`, only the output layer will be trained. As the name implies, this argument is useful for fine-tuning a pre-trained model.

`--train_models` specifies which models are trained. For example, `--train_models joint` will train the Joint model, but not the DNA and CpG model. `--train_models cpg joint` will train the CpG and Joint model, but not the DNA model.

`--trainable` and `--not_trainable` allow including and excluding certain layers. For example, `--not_trainable '.*' --trainable 'dna/.*_2'` will only train the second layers of the DNA model.

`--freeze_filter` excludes the first convolutional layer of the DNA model from training.

4.2.9 Configuring the Keras backend

DeepCpG use the Keras deep learning library, which supports Theano or Tensorflow as backend. While Theano has long been the dominant deep learning library, Tensorflow is more suited for parallelizing computations on multiple GPUs and CPUs, and provides TensorBoard to interactively monitor training.

You can configure the backend by setting the `backend` attribute in `~/.keras/keras.json` to `tensorflow` or `theano`. Alternatively you can set the environemnt variable `KERAS_BACKEND='tensorflow'` to use Tensorflow, or `KERAS_BACKEND='theano'` to use Theano.

You can find more information about Keras backends here.

4.3 Model architectures

DeepCpG consists of a DNA model to recognize features in the DNA sequence, a CpG model to recognize features in the methylation neighborhood of multiple cells, and a Joint model to combine the features from the DNA and CpG

model.

DeepCpG provides different architectures for the DNA, CpG, and joint model. Architectures differ in the number of layers and neurons, and are hence more or less complex. More complex models are usually more accurate, but more expensive to train. You can select a certain architecture using the `--dna_model`, `--cpg_model`, and `--joint_model` argument of `dcpg_train.py`, for example:

```
dcpg_train.py
    --dna_model CnnL2h128
    --cpg_model RnnL1
    --joint_model JointL2h512
```

In the following, the following layer specifications will be used:

Specification	Description
conv[x@y]	Convolutional layer with x filters of size y
mp[x]	Max-pooling layer with size x
fc[x]	Full-connected layer with x units
do	Dropout layer
bgru[x]	Bidirectional GRU with x units
gap	Global average pooling layer
resb[x,y,z]	Residual network with three bottleneck residual units of size x, y, z
resc[x,y,z]	Residual network with three convolutional residual units of size x, y, z
resa[x,y,z]	Residual network with three Atrous residual units of size x, y, z

4.3.1 DNA model architectures

Name	Parameters	Specification
CnnL1h128	4,100,000	conv[128@11]_mp[4]_fc[128]_do
CnnL1h256	8,100,000	conv[128@11]_mp[4]_fc[256]_do
CnnL2h128	4,100,000	conv[128@11]_mp[4]_conv[256@3]_mp[2]_fc[128]_do
CnnL2h256	8,100,000	conv[128@11]_mp[4]_conv[256@3]_mp[2]_fc[256]_do
CnnL3h128	4,400,000	conv[128@11]_mp[4]_conv[256@3]_mp[2]_conv[512@3]_mp[2]_fc[128]_do
CnnL3h256	8,300,000	conv[128@11]_mp[4]_conv[256@3]_mp[2]_conv[512@3]_mp[2]_fc[128]_do
CnnRnn01	1,100,000	conv[128@11]_pool[4]_conv[256@7]_pool[4]_bgru[256]_do
ResNet01	1,700,000	conv[128@11]_mp[2]_resb[2x128|2x256|2x512|1x1024]_gap_do
ResNet02	2,000,000	conv[128@11]_mp[2]_resb[3x128|3x256|3x512|1x1024]_gap_do
ResConv01	2,800,000	conv[128@11]_mp[2]_resc[2x128|1x256|1x256|1x512]_gap_do
ResAtrous01	2,000,000	conv[128@11]_mp[2]_resa[3x128|3x256|3x512|1x1024]_gap_do

Th prefixes `Cnn`, `CnnRnn`, `ResNet`, `ResConv`, and `ResAtrous` denote the class of the DNA model.

Models starting with `Cnn` are convolutional neural networks (CNNs). DeepCpG CNN architectures consist of a series of convolutional and max-pooling layers, which are followed by one fully-connected layer. Model `CnnLxhy` has `x` convolutional-pooling layers, and one fully-connected layer with `y` units. For example, `CnnL2h128` has two convolutional layers, and one fully-connected layer with 128 units. `CnnL3h256` has three convolutional layers and one fully-connected layer with 256 units. `CnnL1h128` is the fastest model, but models with more layers and neurons usually perform better. In my experiments, `CnnL2h128` provided a good trade-off between performance and runtime, which I recommend as default.

`CnnRnn01` is a convolutional-recurrent neural network. It consists of two convolutional-pooling layers, which are followed by a bidirectional recurrent neural network (RNN) with one layer and gated recurrent units (GRUs). `CnnRnn01` is slower than `Cnn` architectures and did not perform better in my experiments.

Models starting with `ResNet` are residual neural networks. ResNets are very deep networks with skip connections to improve the gradient flow and to allow learning how many layers to use. A residual network consists of multiple residual blocks, and each residual block consists of multiple residual units. Residual units have a bottleneck architecture with three convolutional layers to speed up computations. `ResNet01` and `ResNet02` have three residual blocks with two and three residual units, respectively. ResNets are slower than CNNs, but can perform better on large datasets.

Models starting with `ResConv` are ResNets with modified residual units that have two convolutional layers instead of a bottleneck architecture. `ResConv` models performed worse than `ResNet` models in my experiments.

Models starting with `ResAtrous` are ResNets with modified residual units that use Atrous convolutional layers instead of normal convolutional layers. Atrous convolutional layers have dilated filters, i.e. filters with 'holes', which allow scanning wider regions in the inputs sequence and thereby better capturing distant patters in the DNA sequence. However, `ResAtrous` models performed worse than `ResNet` models in my experiments

4.3.2 CpG model architectures

Name	Parameters	Specification
FcAvg	54,000	fc[512]_gap
RnnL1	810,000	fc[256]_bgru[256]_do
RnnL2	1,100,000	fc[256]_bgru[128]_bgru[256]_do

`FcAvg` is a lightweight model with only 54000 parameters, which first transforms observed neighboring CpG sites of all cells independently, and than averages the transformed features across cells. `FcAvg` is very fast, but performs worse than RNN models.

`Rnn` models consists of bidirectional recurrent neural networks (RNNs) with gated recurrent units (GRUs) to summarize the methylation neighborhood of cells in a more clever way than averaging. `RnnL1` consists of one fully-connected layer with 256 units to transform the methylation neighborhood of each cell independently, and one bidirectional GRU with 2x256 units to summarize the transformed methylation neighborhood of cells. `RnnL2` has two instead of one GRU layer. `RnnL1` is faster and performed as good as `RnnL2` in my experiments.

4.3.3 Joint model architectures

Name	Parameters	Specification
JointL0	0	
JointL1h512	524,000	fc[512]
JointL2h512	786,000	fc[512]_fc[512]
JointL3h512	1,000,000	fc[512]_fc[512]_fc[512]

Joint models join the feature from the DNA and CpG model. `JointL0` simply concatenates the features and has no learnable parameters (ultra fast). `JointLXh512` has `X` fully-connect layers with 512 neurons. Models with more layers usually perform better, at the cost of a higher runtime. I recommend using `JointL2h512` or `JointL3h12`.

4.4 Scripts

Documentation of DeepCpG scripts.

4.4.1 dcpg_data.py

Create DeepCpG input data from incomplete methylation profiles.

Takes as input incomplete CpG methylation profiles of multiple cells, extracts neighboring CpG sites and/or DNA sequences windows, and writes data chunk files to output directory. Output data can than be used for model training using `dcpg_train.py` model evaluation using `dcpg_eval.py`.

Examples

Create data files for training a CpG and DNA model, using 50 neighboring methylation states and DNA sequence windows of 1001 bp from the mm10 genome build:

```
dcpg_data.py
    --cpg_profiles ./cpg/*.tsv
    --cpg_wlen 50
    --dna_files ./mm10
    --dna_wlen 1001
    --out_dir ./data
```

Create data files from gzip-compressed bedGraph files for predicting the mean methylation rate and cell-to-cell variance from the DNA sequence:

```
dcpg_data.py
    --cpg_profiles ./cpg/*.bedGraph.gz
    --dna_files ./mm10
    --dna_wlen 1001
    --win_stats mean var
    --win_stats_wlen 1001 2001 3001 4001 5001
    --out_dir ./data
```

See Also

- `dcpg_data_stats.py`: For computing statistics of data files.
- `dcpg_data_show.py`: For showing the content of data files.
- `dcpg_train.py`: For training a model.

`scripts.dcpg_data.`**`extract_seq_windows`**(*seq*, *pos*, *wlen*, *seq_index=1*, *assert_cpg=False*)

Extracts DNA sequence windows at positions.

Parameters

seq: str DNA sequence.

pos: list Positions at which windows are extracted.

wlen: int Window length.

seq_index: int Offset at which positions start.

assert_cpg: bool If *True*, check if positions in *pos* point to CpG sites.

Returns

np.array Array with integer-encoded sequence windows.

scripts.dcpg_data.**map_cpg_tables**(*cpg_tables*, *chromo*, *chromo_pos*)
Maps values from cpg_tables to *chromo_pos*.

Positions in *cpg_tables* for *chromo* must be a subset of *chromo_pos*. Inserts *dat.CPG_NAN* for uncovered positions.

scripts.dcpg_data.**map_values**(*values*, *pos*, *target_pos*, *dtype=None*, *nan=-1*)
Maps *values* array at positions *pos* to *target_pos*.

Inserts *nan* for uncovered positions.

scripts.dcpg_data.**prepro_pos_table**(*pos_tables*)
Extracts unique positions and sorts them.

scripts.dcpg_data.**read_cpg_profiles**(*filenames*, *log=None*, **args*, ***kwargs*)
Read methylation profiles.

Input files can be gzip compressed.

Returns

dict *dict (key, value)*, where *key* is the output name and *value* the CpG table.

scripts.dcpg_data.**split_ext**(*filename*)
Remove file extension from *filename*.

4.4.2 dcpg_data_show.py

Show the content of DeepCpG data files.

Shows the content of `dcpg_data.py` output files for a selected region, for example the methylation state of the target CpG site, neighboring CpG sites, or the DNA sequence.

Examples

Show the output methylation state of CpG sites on on chromosome 19 between position 3028955 and 3079682:

```
dcpg_data_show.py
    ./data/*.h5
    --chromo 1
    --start 3028955
    --end 3079682
    --outputs
```

Show output methylation states and the state as well as the distance of 10 neighboring CpG sites of cell BS27_1_SER:

```
dcpg_data_show.py
    ./data/*.h5
    --chromo 1
    --start 3028955
    --end 3079682
    --outputs cpg/BS27_1_SER
    --cpg BS27_1_SER
    --cpg_wlen 10
    --cpg_dist
```

Show output methylation states and DNA sequence windows of length 11 and store the results in HDF5 file `selected.h5`:

```
dcpg_data_show.py
    ./data/*.h5
    --chromo 1
    --start 3028955
    --end 3079682
    --outputs
    --dna_wlen 11
    --out_hdf selected.h5
```

4.4.3 dcpg_data_stats.py

Compute summary statistics of data files.

Computes summary statistics of data files such as the number of samples or the mean and variance of output variables.

Examples

```
dcpg_data_stats.py
    ./data/*.h5
```

4.4.4 dcpg_download.py

Download a pre-trained model from DeepCpG model zoo.

Downloads a pre-trained model from the DeepCpG model zoo by its identifier. Model descriptions can be found on online.

Examples

Show available models:

```
dcpg_download --show
```

Download DNA model trained on serum cells from Smallwood et al:

```
dcpg_download.py
    Smallwood2014_serum_dna
    -o ./model
```

4.4.5 dcpg_eval.py

Evaluate the prediction performance of a DeepCpG model.

Imputes missing methylation states and evaluates model on observed states. `--out_report` will write evaluation metrics to a TSV file using. `--out_data` will write predicted and observed methylation state to a HDF5 file with following structure:

- `chromo`: The chromosome of the CpG site.
- `pos`: The position of the CpG site on the chromosome.
- `outputs`: The input methylation state of each cell and CpG site, which can either observed or missing (-1).

- `preds`: The predicted methylation state of each cell and CpG site.

Examples

```
dcpg_eval.py
    ./data/*.h5
    --model_files ./model
    --out_data ./eval/data.h5
    --out_report ./eval/report.tsv
```

4.4.6 dcpg_eval_export.py

Export imputed methylation profiles.

Exports imputed methylation profiles from *dcpg_eval.py* output file to different data formats. Outputs for each CpG site and cell either the experimentally observed or predicted methylation state depending on whether or not the methylation state was observed in the input file or not, respectively. Creates for each methylation profile one file in the output directory.

Examples

Export profiles of all cells as HDF5 files to *./eval*:

```
dcpg_eval_export.py
    ./eval/data.h5
    --out_dir ./eval
```

Export the profile of cell Ca01 for chromosomes 4 and 5 to a bedGraph file:

```
dcpg_eval_export.py
    ./eval/data.h5
    --output cpg/Ca01
    --chromo 4 5
    --format bedGraph
    --out_dir ./eval
```

4.4.7 dcpg_eval_perf.py

Evaluate prediction performance.

Evaluates prediction performances globally and genomic annotations.

Examples

Evaluate prediction performance globally and in genomic contexts annotated as CGI, TSS, or gene body. Also compute precision recall and ROC curve of individual outputs:

```
dcpg_eval_perf.py
    ./eval/data.h5
    --out_dir ./eval
    --curves pr roc
    --annos_files ./bed/CGI.bed ./bed/TSS.bed ./bed/gene_body.bed
```

scripts.dcpg_eval_perf.**annotate**(*chromos*, *pos*, *anno*)
Annotate genomic locations.

Tests if sites specified by *chromos* and *pos* are annotated by *anno*.

Parameters

chromos: :class:'numpy.ndarray' numpy.ndarray with chromosome of sites.

pos: :class:'numpy.ndarray' numpy.ndarray with position on chromosome of sites.

anno: :class:'pandas.DataFrame' pandas.DataFrame with columns *chromo*, *start*, *end* that specify annotated regions.

Returns

:class:'numpy.ndarray' Binary numpy.ndarray of same length as *chromos* indicating if positions are annotated.

scripts.dcpg_eval_perf.**get_curve_fun**(*name*)
Return performance curve function by its name.

scripts.dcpg_eval_perf.**read_anno_file**(*anno_file*, *chromos=None*, *nb_sample=None*)
Read annotations from BED file.

Reads annotations from BED file merges overlapping annotations.

Parameters

anno_file: str File name.

chromos: list List of chromosomes for filtering annotations.

nb_sample: int Maximum number of annotated regions.

Returns

:class:'pandas.DataFrame' pandas.DataFrame with columns *chromo*, *start*, *end*.

4.4.8 dcpg_filter_act.py

Compute filter activations of a DeepCpG model.

Computes the activation of the filters of the first convolutional layer for a given DNA model. The resulting activations can be used to visualize and cluster motifs, or correlated with model outputs.

Examples

Compute activations in 25000 sequence windows and also store DNA sequences. For example to visualize motifs.

```
dcpg_filter_act.py
    ./data/*.h5
    --model_files ./models/dna
    --out_file ./activations.h5
    --nb_sample 25000
    --store_inputs
```

Compute the weighted mean activation in each sequence window and also store model predictions. For example to cluster motifs or to correlated mean motif activations with model predictions.

```
dcpg_filter_act.py
    ./data/*.h5
    --model_files ./models/dna
    --out_file ./activations.h5
    --act_fun wmean
```

See Also

- `dcpg_filter_motifs.py`: For motif visualization and analysis.

4.4.9 dcpg_filter_motifs.py

Visualizes and analyzes filter motifs.

Enables to visualize motifs as sequence logos, compare motifs to annotated motifs, cluster motifs, and compute motif summary statistics. Requires Weblogo3 for visualization, and Tomtom for motif comparison.

Copyright (c) 2015 David Kelley since since parts of the code are based on the Basset script `basset_motifs.py` from David Kelley.

Examples

Compute filter activations and also store input DNA sequence windows:

```
dcpg_filter_act.py
    ./data/*.h5
    --out_file ./activations.h5
    --store_inputs
    --nb_sample 100000
```

Visualize and analyze motifs:

```
dcpg_filter_motifs.py
    ./activations.h5
    --out_dir ./motifs
    --motif_db ./motif_databases/CIS-BP/Mus_musculus.meme
    --plot_heat
    --plot_dens
    --plot_pca
```

4.4.10 dcpg_snp.py

Compute the effect of DNA mutations on methylation.

Computes the effect of DNA mutation on the mean methylation rate or cell-to-cell variance using gradient backpropagation.

Examples

Compute the effect on mean methylation rates and cell-to-cell variance:

```
dcpg_snp.py
    ./data/*.h5
    --model_files ./model/dna
    --out_file ./effects.h5
    --targets mean var
```

Compute weighted mean effects in DNA sequence windows of length 101:

```
dcpg_snp.py
    ./data/*.h5
    --model_files ./model/dna
    --out_file ./effects.h5
    --targets mean var
    --dna_wlen 101
    --agg_effects wmean
```

4.4.11 dcpg_train.py

Train a DeepCpG model to predict DNA methylation.

Trains a DeepCpG model on DNA (DNA model), neighboring methylation states (CpG model), or both (Joint model) to predict CpG methylation of multiple cells. Allows to fine-tune individual models or to train them from scratch.

Examples

Train a DNA model on chromosome 1, 3, and 5, and use chromosome 13, 14, and 15 for validation:

```
dcpg_train.py
    ./data/c{1,3,5}_*.h5
    --val_files ./data/c{13,14,15}_*.h5
    --dna_model CnnL2h128
    --out_dir ./models/dna
```

Train a CpG model:

```
dcpg_train.py
    ./data/c{1,3,5}_*.h5
    --val_files ./data/c{13,14,15}_*.h5
    --cpg_model RnnL1
    --out_dir ./models/cpg
```

Train a Joint model using a pre-trained DNA and CpG model:

```
dcpg_train.py
    ./data/c{1,3,5}_*.h5
    --val_files ./data/c{13,14,15}_*.h5
    --dna_model ./models/dna
    --cpg_model ./models/cpg
    --joint_model JointL2h512
    --train_models joint
    --out_dir ./models/joint
```

See Also

- **`dcpg_eval.py`: For evaluating a trained model and imputing methylation** profiles.

4.4.12 dcpg_train_viz.py

Visualizes learning curves of `dcpg_train.py`.

Visualizes training and validation learning from *dcpg_train.py*. Tensorboard is recommended for advanced visualization.

Examples

```
dcpg_train_viz.py
    ./model/lc_train.tsv ./model/lc_val.tsv
    --out_file ./lc.pdf
```

4.5 Library

Documentation of DeepCpG library.

4.5.1 `callbacks`

Keras callback classes used by *dcpg_train.py*.

class deepcpg.callbacks.**PerformanceLogger**(*metrics=['loss', 'acc'], log_freq=0.1, precision=4, callbacks=[], verbose=<class 'bool'>, logger=<built-in function print>*)

Logs performance metrics during training.

Stores and prints performance metrics for each batch, epoch, and output.

Parameters

metrics: list Name of metrics to be logged.

log_freq: float Logging frequency as the percentage of training samples per epoch.

precision: int Floating point precision.

callbacks: list List of functions with parameters *epoch*, *epoch_logs*, and *val_epoch_logs* that are called at the end of each epoch.

verbose: bool If *True*, log performance metrics of individual outputs.

logger: function Logging function.

class deepcpg.callbacks.**TrainingStopper**(*max_time=None, stop_file=None, verbose=1, logger=<built-in function print>*)

Stop training after certain time or when file is detected.

Parameters

max_time: int Maximum training time in seconds.

stop_file: str Name of stop file that triggers the end of training when existing.

verbose: bool If *True*, log message when training is stopped.

4.5.2 evaluation

Functions for evaluating prediction performance.

deepcpg.evaluation.**acc**(*y*, *z*, *round=True*)
Compute accuracy.

deepcpg.evaluation.**auc**(*y*, *z*, *round=True*)
Compute area under the ROC curve.

deepcpg.evaluation.**cat_acc**(*y*, *z*)
Compute categorical accuracy given one-hot matrices.

deepcpg.evaluation.**cor**(*y*, *z*)
Compute Pearson's correlation coefficient.

deepcpg.evaluation.**evaluate**(*y, z, mask=-1, metrics=[<function auc>, <function acc>, <function tpr>, <function tnr>, <function f1>, <function mcc>]*)
Compute multiple performance metrics.

Computes evaluation metrics using functions in *metrics*.

Parameters

y: :class:'numpy.ndarray' `numpy.ndarray` vector with labels.

z: :class:'numpy.ndarray' `numpy.ndarray` vector with predictions.

mask: scalar Value to mask unobserved labels in *y*.

metrics: list List of evaluation functions to be used.

Returns

Ordered dict Ordered dict with name of evaluation functions as keys and evaluation metrics as values.

deepcpg.evaluation.**evaluate_cat**(*y, z, metrics=[<function cat_acc>], binary_metrics=None*)
Compute multiple performance metrics for categorical outputs.

Computes evaluation metrics for categorical (one-hot encoded labels) using functions in *metrics*.

Parameters

y: :class:'numpy.ndarray' `numpy.ndarray` matrix with one-hot encoded labels.

z: :class:'numpy.ndarray' `numpy.ndarray` matrix with class probabilities in rows.

metrics: list List of evaluation functions to be used.

binary_metrics: list List of binary evaluation metrics to be computed for each category, e.g. class, separately. Will be encoded as *name_i* in the output dictionary, where *name* is the name of the evaluation metrics and *i* the index of the category.

Returns

Ordered dict Ordered dict with name of evaluation functions as keys and evaluation metrics as values.

deepcpg.evaluation.**evaluate_curve**(*outputs*, *preds*, *fun=<function roc_curve>*, *mask=-1*, *nb_point=None*)
Evaluate performance curves of multiple outputs.

Given the labels and predictions of multiple outputs, computes a performance a curve, e.g. ROC or PR curve, for each output.

Parameters

outputs: dict *dict* with the name of outputs as keys and a `numpy.ndarray` vector with labels as value.

preds: dict *dict* with the name of outputs as keys and a `numpy.ndarray` vector with predictions as value.

fun: function Function to compute the performance curves.

mask: scalar Value to mask unobserved labels in *y*.

nb_point: int Maximum number of points to curve to reduce memory.

Returns

:class:'pandas.DataFrame' `pandas.DataFrame` with columns *output*, *x*, *y*, *thr*.

`deepcpg.evaluation.`**`evaluate_outputs`**(*outputs*, *preds*)
Evaluate performance metrics of multiple outputs.

Given the labels and predictions of multiple outputs, chooses and computes performance metrics of each output depending on its name.

Parameters

outputs: dict *dict* with the name of outputs as keys and a `numpy.ndarray` vector with labels as value.

preds: dict *dict* with the name of outputs as keys and a `numpy.ndarray` vector with predictions as value.

Returns

:class:'pandas.DataFrame' `pandas.DataFrame` with columns *metric*, *output*, *value*.

`deepcpg.evaluation.`**`f1`**(*y*, *z*, *round=True*)
Compute F1 score.

`deepcpg.evaluation.`**`get`**(*name*)
Return object from module by its name.

`deepcpg.evaluation.`**`get_output_metrics`**(*output_name*)
Return list of evaluation metrics for model output name.

`deepcpg.evaluation.`**`is_binary_output`**(*output_name*)
Return *True* if *output_name* is binary.

`deepcpg.evaluation.`**`kendall`**(*y*, *z*, *nb_sample=100000*)
Compute Kendall's correlation coefficient.

`deepcpg.evaluation.`**`mad`**(*y*, *z*)
Compute mean absolute deviation.

`deepcpg.evaluation.`**`mcc`**(*y*, *z*, *round=True*)
Compute Matthew's correlation coefficient.

`deepcpg.evaluation.`**`mse`**(*y*, *z*)
Compute mean squared error.

`deepcpg.evaluation.`**`rmse`**(*y*, *z*)
Compute root mean squared error.

deepcpg.evaluation.**tnr**(*y*, *z*, *round=True*)
Compute true negative rate.

deepcpg.evaluation.**tpr**(*y*, *z*, *round=True*)
Compute true positive rate.

deepcpg.evaluation.**unstack_report**(*report*)
Unstack performance report.

Reshapes a pandas.DataFrame of *evaluate_outputs()* such that performance metrics are listed as columns.

Parameters

report: :class:'pandas.DataFrame' pandas.DataFrame from *evaluate_outputs()*.

Returns

:class:'pandas.DataFrame' pandas.DataFrame with performance metrics as columns.

4.5.3 motifs

Motif analysis.

deepcpg.motifs.**get_report**(*filter_stats_file*, *tomtom_file*, *meme_motifs*)
Read and join *filter_stats_file* and *tomtom_file*.

Used by *dcpg_filter_motifs.py* to read and join output files.

Returns

:class:'pandas.DataFrame' pandas.DataFrame with columns from Tomtom and statistic file.

deepcpg.motifs.**read_meme_db**(*meme_db_file*)
Read MEME database as Pandas DataFrame.

Parameters

meme_db_file: str File name of MEME database.

Returns

:class:'pandas.DataFrame' pandas.DataFrame with columns 'id', 'protein', 'url'.

deepcpg.motifs.**read_tomtom**(*path*)
Read Tomtom output file.

4.5.4 utils

General-purpose functions.

class deepcpg.utils.**ProgressBar**(*nb_tot*, *logger=<built-in function print>*, *interval=0.1*)
Vertical progress bar.

Unlike the progressbar2 package, logs progress as multiple lines instead of single line, which enables printing to a file. Used, for example, in

Parameters

nb_tot: int Maximum value

logger: function Function that takes a *str* and prints it.

interval: float Logging frequency as fraction of one. For example, 0.1 logs every tenth value.

See also:

`dcpg_eval.py`

`deepcpg.utils.`**`filter_regex`**(*values*, *regexs*)
Filters list of *values* by list of *regexs*.

Returns

list Sorted *list* of values in *values* that match any regex in *regexs*.

`deepcpg.utils.`**`fold_dict`**(*data*, *nb_level=100000*)
Fold dict *data*.

Turns dictionary keys, e.g. 'level1/level2/level3', into sub-dicts, e.g. data['level1']['level2']['level3'].

Parameters

data: dict dict to be folded.

nb_level: int Maximum recursion depth.

Returns

dict Folded dict.

`deepcpg.utils.`**`format_table`**(*table*, *colwidth=None*, *precision=2*, *header=True*, *sep=' | '*)
Format a table of values as string.

Formats a table represented as a *dict* with keys as column headers and values as a lists of values in each column.

Parameters

table: 'dict' or 'OrderedDict' *dict* or *OrderedDict* with keys as column headers and values as lists of values in each column.

precision: int or list of ints Precision of floating point values in each column. If *int*, uses same precision for all columns, otherwise formats columns with different precisions.

header: bool If *True*, print column names.

sep: str Column separator.

Returns

str String of formatted table values.

`deepcpg.utils.`**`format_table_row`**(*values*, *widths=None*, *sep=' | '*)
Format a row with *values* of a table.

`deepcpg.utils.`**`get_from_module`**(*identifier*, *module_params*, *ignore_case=True*)
Return object from module.

Return object with name *identifier* from module with items *module_params*.

Parameters

identifier: str Name of object, e.g. a function, in module.

module_params: dict *dict* of items in module, e.g. *globals()*

ignore_case: bool If *True*, ignore case of *identifier*.

Returns

object Object with name *identifier* in module, e.g. a function or class.

deepcpg.utils.**linear_weights**(*length*, *start=0.1*)
Create linear-triangle weights.

Create array *x* of length *length* with linear weights, where the weight is highest (one) for the center x[length//2] and lowest (*start*) at the ends x[0] and x[-1].

Parameters

length: int Length of the weight array.

start: float Minimum weights.

Returns

:class:'np.ndarray' Array of length *length* with weight.

deepcpg.utils.**make_dir**(*dirname*)
Create directory *dirname* if non-existing.

Parameters

dirname: str Path of directory to be created.

Returns

bool *True*, if directory did not exist and was created.

deepcpg.utils.**move_columns_front**(*frame*, *columns*)
Move *columns* of Pandas DataFrame to the front.

deepcpg.utils.**slice_dict**(*data*, *idx*)
Slice elements in dict *data* by *idx*.

Slices array-like objects in *data* by index *idx*. *data* can be tree-like with sub-dicts, where the leafs must be sliceable by *idx*.

Parameters

data: dict dict to be sliced.

idx: slice Slice index.

Returns

dict dict with same elements as in *data* with sliced by *idx*.

deepcpg.utils.**to_list**(*value*)
Convert *value* to a list.

4.5.5 data

Package for reading, writing, and transforming data.

data.annotations

Functions for reading and matching annotations.

deepcpg.data.annotations.**distance**(*pos*, *start*, *end*)
Return shortest distance between a position and a list of intervals.

Parameters

pos: list List of integer positions.

start: list Start position of intervals.

end: list End position of intervals.

Returns

:class:'numpy.ndarray' `numpy.ndarray` of same length as *pos* with shortest distance between each *pos[i]* and any interval.

`deepcpg.data.annotations.`**`extend_len`**(*start*, *end*, *min_len*, *min_pos=1*)

Extend intervals to minimum length.

Extends intervals *start-end* with length smaller than *min_len* to length *min_len* by equally decreasing *start* and increasing *end*.

Parameters

start: list List of start position of intervals.

end: list List of end position of intervals.

min_len: int Minimum interval length.

min_pos: int Minimum position.

Returns

tuple *tuple* with start end end position of extended intervals.

`deepcpg.data.annotations.`**`extend_len_frame`**(*d*, *min_len*)

Extend length of intervals in Pandas DataFrame using *extend_len*.

`deepcpg.data.annotations.`**`group_overlapping`**(*s*, *e*)

Assign group index of indices.

Assigns group index to intervals. Overlapping intervals will be assigned to the same group.

Parameters

s [list] list with start of interval sorted in ascending order.

e [list] list with end of interval.

Returns

:class:'numpy.ndarray' `numpy.ndarray` with group indices.

`deepcpg.data.annotations.`**`in_which`**(*x*, *ys*, *ye*)

Return index of positions in intervals.

Returns for positions *x[i]* index *j*, s.t. *ys[j]* <= *x[i]* <= *ye[j]*, or -1 if *x[i]* is not overlapped by any interval. Intervals must be non-overlapping!

Parameters

x [list] list of positions.

ys: list list with start of interval sorted in ascending order.

ye: list list with end of interval.

Returns

:class:'numpy.ndarray' n:class:*numpy.ndarray* with indices of overlapping intervals or -1.

`deepcpg.data.annotations.`**`is_in`**(*pos*, *start*, *end*)

Test if position is overlapped by at least one interval.

deepcpg.data.annotations.**join_overlapping**(*s*, *e*)
Join overlapping intervals.

Transforms a list of possible overlapping intervals into non-overlapping intervals.

Parameters

s [list] List with start of interval sorted in ascending order

e [list] List with end of interval.

Returns

tuple *tuple* (s, e) of non-overlapping intervals.

deepcpg.data.annotations.**join_overlapping_frame**(*d*)
Join overlapping intervals of Pandas DataFrame.

Uses *join_overlapping* to join overlapping intervals of `pandas.DataFrame` *d*.

deepcpg.data.annotations.**read_bed**(*filename, sort=False, usecols=[0, 1, 2], *args, **kwargs*)
Read chromo,start,end from BED file without formatting chromo.

data.dna

Functions for representing DNA sequences.

deepcpg.data.dna.**char_to_int**(*seq*)
Translate chars of single sequence *seq* to ints.

Parameters

seq: str DNA sequence.

Returns

list Integer-encoded *seq*.

deepcpg.data.dna.**get_alphabet**(*special=False*, *reverse=False*)
Return char->int alphabet.

Parameters

special: bool If *True*, remove special 'N' character.

reverse: bool If *True*, return int->char instead of char->int alphabet.

Returns

OrderedDict DNA alphabet.

deepcpg.data.dna.**int_to_char**(*seq*, *join=True*)
Translate ints of single sequence *seq* to chars.

Parameters

seq: list Integers of sequences

join: bool If *True* joint characters to *str*.

Returns

If 'join=True', 'str', otherwise list of chars.

deepcpg.data.dna.**int_to_onehot**(*seqs*, *dim=4*)
One-hot encodes array of integer sequences.

Takes array [nb_seq, seq_len] of integer sequence end encodes them one-hot. Special nucleotides (int > 4) will be encoded as [0, 0, 0, 0].

Returns

:class:'numpy.ndarray' [nb_seq, seq_len, dim] numpy.ndarray of one-hot encoded sequences.

deepcpg.data.dna.**onehot_to_int**(*seqs*, *axis=-1*)
Translates one-hot sequences to integer sequences.

data.fasta

Functions reading FASTA files.

class deepcpg.data.fasta.**FastaSeq**(*head*, *seq*)
FASTA sequence.

deepcpg.data.fasta.**parse_lines**(*lines*)
Parse FASTA sequences from list of strings.

Parameters

lines: list List of lines from FASTA file.

Returns

list List of *FastaSeq* objects.

deepcpg.data.fasta.**read_chromo**(*filenames*, *chromo*)
Read DNA sequence of chromosome *chromo*.

Parameters

filenames: list List of FASTA files.

chromo: str Chromosome that is read.

Returns

str DNA sequence of chromosome *chromo*.

deepcpg.data.fasta.**read_file**(*filename*, *gzip=None*)
Read FASTA file and return sequences.

Parameters

filename: str File name.

gzip: bool If *True*, file is gzip compressed. If *None*, suffix is used to determine if file is compressed.

Returns

List of :class:'FastaSeq' objects.

deepcpg.data.fasta.**select_file_by_chromo**(*filenames*, *chromo*)
Select file of chromosome *chromo*.

Parameters

filenames: list List of file names or directory with FASTA files.

chromo: str Chromosome that is selected.

Returns

str Filename in *filenames* that contains chromosome *chromo*.

data.feature_extractor

Feature extraction.

class deepcpg.data.feature_extractor.**IntervalFeatureExtractor**

Check if positions are in a list of intervals (start, end).

static index_intervals(*x*, *ys*, *ye*)

Return for positions x[i] index j, s.t. ys[j] <= x[i] <= ye[j] or -1. Intervals must be non-overlapping!

Parameters

x [list] List of positions.

ys: list List with start of interval sorted in ascending order.

ye: list List with end of interval.

Returns

:class:'numpy.ndarray' numpy.ndarray of same length than x with index or -1.

static join_intervals(*s*, *e*)

Transform a list of possible overlapping intervals into non-overlapping intervals.

Parameters

s: list List with start of interval sorted in ascending order.

e: list List with end of interval.

Returns

tuple Tuple (s, e) of non-overlapping intervals.

class deepcpg.data.feature_extractor.**KnnCpgFeatureExtractor**(*k=1*)

Extract k CpG sites next to target sites. Exclude CpG sites at the same position.

extract(*x*, *y*, *ys*)

Extract state and distance of k CpG sites next to target sites. Target site is excluded.

Parameters

x: :class:'numpy.ndarray' numpy.ndarray with target positions sorted in ascending order.

y: :class:'numpy.ndarray' numpy.ndarray with source positions sorted in ascending order.

ys: :class:'numpy.ndarray' numpy.ndarray with source CpG states.

Returns

tuple Tuple (cpg, dist) with numpy arrays of dimension (len(x), 2k): cpg: CpG states to the left (0:k) and right (k:2k) dist: Distances to the left (0:k) and right (k:2k)

data.hdf

Functions for accessing HDF5 files.

deepcpg.data.hdf.**hnames_to_names**(*hnames*)
Flattens *dict hnames* of hierarchical names.

Converts hierarchical *dict*, e.g. hnames={'a': ['a1', 'a2'], 'b'}, to flat list of keys for accessing HDF5 file, e.g. ['a/a1', 'a/a2', 'b']

deepcpg.data.hdf.**ls**(*filename*, *group='/'*, *recursive=False*, *groups=False*, *regex=None*, *nb_key=None*, *must_exist=True*)
List name of records HDF5 file.

Parameters

filename: Path of HDF5 file.

group: HDF5 group to be explored.

recursive: bool If *True*, list records recursively.

groups: bool If *True*, only list group names but not name of datasets.

regex: str Regex to filter listed records.

nb_key: int Maximum number of records to be listed.

must_exist: bool If *False*, return *None* if file or group does not exist.

Returns

list *list* with name of records in *filename*.

deepcpg.data.hdf.**write_data**(*data*, *filename*)
Write data in dict *data* to HDF5 file.

data.stats

Functions for computing statistic about binary CpG matrix.

CpG matrix x assumed to have shape

- [sites, cells] for per CpG statistics
- [sites, cells, context] for window-based statistics

deepcpg.data.stats.**cat2_var**(**args*, ***kwargs*)
Binary variance between cells.

deepcpg.data.stats.**cat_var**(*x*, *nb_bin=3*, **args*, ***kwargs*)
Categorical variance between cells.

Discretizes variance from *var()* into *nb_bin* equally-spaced bins.

deepcpg.data.stats.**diff**(*x*)
Test if CpG site is differentially methylated.

deepcpg.data.stats.**entropy**(*x*)
Entropy of single CpG sites between cells.

deepcpg.data.stats.**get**(*name*)
Return object from module by its name.

deepcpg.data.stats.**mean**(*x*)
Mean methylation rate.

deepcpg.data.stats.**mode**(*x*)
Mode of methylation rate.

deepcpg.data.stats.**var**(*x*, **args*, ***kwargs*)
Variance between cells.

data.utils

General purpose IO functions.

class deepcpg.data.utils.**GzipFile**(*filename*, *mode='r'*, **args*, ***kwargs*)
Wrapper to read and write gzip-compressed files.

If *filename* ends with *gz*, opens file with gzip package, otherwise builtin *open* function.

Parameters

filename: str Path of file

mode: str File access mode

***args: list** Unnamed arguments passed to open function.

****kwargs: dict** Named arguments passed to open function.

deepcpg.data.utils.**add_to_dict**(*src*, *dst*)
Add *dict 'src* to *dict dst*

Adds values in *dict src* to *dict dst* with same keys but values are lists of added values. lists of values in *dst* can be stacked with `stack_dict()`. Used for example in *dpcg_eval.py* to stack dicts from different batches.

deepcpg.data.utils.**format_chromo**(*chromo*)
Format chromosome name.

Makes name upper case, e.g. 'mt' -> 'MT' and removes 'chr', e.g. 'chr1' -> '1'.

deepcpg.data.utils.**get_anno_names**(*data_file*, **args*, ***kwargs*)
Return name of annotations stored in *data_file*.

deepcpg.data.utils.**get_cpg_wlen**(*data_file*, *max_len=None*)
Return number of CpG neighbors stored in *data_file*.

deepcpg.data.utils.**get_dna_wlen**(*data_file*, *max_len=None*)
Return length of DNA sequence windows stored in *data_file*.

deepcpg.data.utils.**get_nb_sample**(*data_files*, *nb_max=None*, *batch_size=None*)
Count number of samples in all *data_files*.

Parameters

data_files: list *list* with file name of DeepCpG data files.

nb_max: int If defined, stop counting if that number is reached.

batch_size: int If defined, return the largest multiple of *batch_size* that is smaller or equal than the actual number of samples.

Returns

int Number of samples in *data_files*.

deepcpg.data.utils.**get_output_names**(*data_file*, **args*, ***kwargs*)
Return name of outputs stored in *data_file*.

deepcpg.data.utils.**get_replicate_names**(*data_file*, **args*, ***kwargs*)
Return name of replicates stored in *data_file*.

deepcpg.data.utils.**is_bedgraph**(*filename*)
Test if *filename* is a bedGraph file.

bedGraph files are assumed to start with 'track type=bedGraph'

deepcpg.data.utils.**is_binary**(*values*)
Check if values in array *values* are binary, i.e. zero or one.

deepcpg.data.utils.**read_cpg_profile**(*filename*, *chromos=None*, *nb_sample=None*, *round=False*, *sort=True*, *nb_sample_chromo=None*)
Read CpG profile from TSV or bedGraph file.

Reads CpG profile from either tab delimited file with columns *chromo*, *pos*, *value*. *value* or bedGraph file. *value* columns contains methylation states, which can be binary or continuous.

Parameters

filenamne: str Path of file.

chromos: list List of formatted chromosomes to be read, e.g. ['1', 'X'].

nb_sample: int Maximum number of sample in total.

round: bool If *True*, round methylation states in column 'value' to zero or one.

sort: bool If *True*, sort by rows by chromosome and position.

nb_sample_chromo: int Maximum number of sample per chromosome.

Returns

:class:'pandas.DataFrame' `pandas.DataFrame` with columns *chromo*, *pos*, *value*.

deepcpg.data.utils.**sample_from_chromo**(*frame*, *nb_sample*)
Randomly sample *nb_sample* samples from each chromosome.

Samples *nb_sample* records from `pandas.DataFrame` which must contain a column with name 'chromo'.

deepcpg.data.utils.**stack_dict**(*data*)
Stacks lists of numpy arrays in *dict data*.

deepcpg.data.utils.**threadsafe_generator**(*f*)
A decorator that takes a generator function and makes it thread-safe.

class deepcpg.data.utils.**threadsafe_iter**(*it*)
Takes an iterator/generator and makes it thread-safe by serializing call to the *next* method of given iterator/generator.

4.5.6 model

Package for building and training DeepCpG modules.

model.utils

Functions for building, training, and loading models.

class deepcpg.models.utils.**DataReader**(*output_names=None*, *use_dna=True*, *dna_wlen=None*, *replicate_names=None*, *cpg_wlen=None*, *cpg_max_dist=25000*, *encode_replicates=False*)

Read data from *dcpg_data.py* output files.

Generator to read data batches from *dcpg_data.py* output files. Reads data using `hdf.reader()` and preprocesses data.

Parameters

output_names: list Names of outputs to be read.

use_dna: bool If *True*, read DNA sequence windows.

dna_wlen: int Maximum length of DNA sequence windows.

replicate_names: list Name of cells (profiles) whose neighboring CpG sites are read.

cpg_wlen: int Maximum number of neighboring CpG sites.

cpg_max_dist: int Value to threshold the distance of neighboring CpG sites.

encode_replicates: bool If *True*, encode replicated names in key of returned dict. This option is deprecated and will be removed in the future.

Returns

tuple *dict* (*inputs*, *outputs*, *weights*), where *inputs*, *outputs*, *weights* is a *dict* of model inputs, outputs, and output weights. *outputs* and *weights* are not returned if *output_names* is undefined.

class deepcpg.models.utils.**Model**(*dropout=0.0*, *l1_decay=0.0*, *l2_decay=0.0*, *init='glorot_uniform'*)

Abstract model call.

Abstract class of DNA, CpG, and Joint models.

Parameters

dropout: float Dropout rate.

l1_decay: float L1 weight decay.

l2_decay: float L2 weight decay.

init: str Name of Keras initialization.

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

Return list of Keras model inputs.

class deepcpg.models.utils.**ScaledSigmoid**(*scaling=1.0*, ***kwargs*)

Scaled sigmoid activation function.

Scales the maximum of the sigmoid function from one to the provided value.

Parameters

scaling: float Maximum of sigmoid function.

call(*x*, *mask=None*)

This is where the layer's logic lives.

 # Arguments inputs: Input tensor, or list/tuple of input tensors. **kwargs: Additional keyword arguments.

 # Returns A tensor or list/tuple of tensors.

get_config()
Returns the config of the layer.

A layer config is a Python dictionary (serializable) containing the configuration of a layer. The same layer can be reinstantiated later (without its trained weights) from this configuration.

The config of a layer does not include connectivity information, nor the layer class name. These are handled by *Network* (one layer of abstraction above).

 # Returns Python dictionary.

deepcpg.models.utils.**add_output_layers**(*stem*, *output_names*, *init='glorot_uniform'*)
Add and return outputs to a given layer.

Adds output layer for each output in *output_names* to layer *stem*.

Parameters

stem: Keras layer Keras layer to which output layers are added.

output_names: list List of output names.

Returns

list Output layers added to *stem*.

deepcpg.models.utils.**copy_weights**(*src_model*, *dst_model*, *must_exist=True*)
Copy weights from *src_model* to *dst_model*.

Parameters

src_model Keras source model.

dst_model Keras destination model.

must_exist: bool If *True*, raises *ValueError* if a layer in *dst_model* does not exist in *src_model*.

Returns

list Names of layers that were copied.

deepcpg.models.utils.**data_reader_from_model**(*model*, *outputs=True*, *replicate_names=None*)
Return *DataReader* from *model*.

Builds a *DataReader* for reading data for *model*.

Parameters

model: :class:'Model'. *Model*.

outputs: bool If *True*, return output labels.

replicate_names: list Name of input cells of *model*.

Returns

:class:'DataReader' Instance of *DataReader*.

deepcpg.models.utils.**decode_replicate_names**(*replicate_names*)
Decode string of replicate names and return names as list.

Note: Deprecated This function is used to support legacy models and will be removed in the future.

---

deepcpg.models.utils.**encode_replicate_names**(*replicate_names*)
Encode list of replicate names as single string.

Note: Deprecated This function is used to support legacy models and will be removed in the future.

deepcpg.models.utils.**evaluate_generator**(*model*, *generator*, *return_data=False*, **args*, ***kwargs*)
Evaluate model on generator.

Uses *predict_generator* to obtain predictions and *ev.evaluate* to evaluate predictions.

Parameters

model Model to be evaluated.

generator Data generator.

return_rate: bool Return predictions and labels.

***args: list** Unnamed arguments passed to *predict_generator*.

***kwargs: dict** Named arguments passed to *predict_generator*.

Returns

If 'return_data=False', pandas data frame with performance metrics. If 'return_data=True', tuple ('perf', 'data') with performance metrics 'perf' and 'data'.

deepcpg.models.utils.**get_first_conv_layer**(*layers*, *get_act=False*)
Return the first convolutional layers in a stack of layer.

Parameters

layers: list List of Keras layers.

get_act: bool Return the activation layer after the convolutional weight layer.

Returns

Keras layer Convolutional layer or tuple of convolutional layer and activation layer if *get_act=True*.

deepcpg.models.utils.**get_objectives**(*output_names*)
Return training objectives for a list of output names.

Returns

dict dict with *output_names* as keys and the name of the assigned Keras objective as values.

deepcpg.models.utils.**get_sample_weights**(*y*, *class_weights=None*)
Compute sample weights for model training.

Computes sample weights given a vector of output labels *y*. Sets weights of samples without label (*CPG_NAN*) to zero.

Parameters

y: :class:'numpy.ndarray' 1d numpy array of output labels.

class_weights: dict Weight of output classes, e.g. methylation states.

Returns

:class:'numpy.ndarray' Sample weights of size *y*.

deepcpg.models.utils.**is_input_layer**(*layer*)
Test if *layer* is an input layer.

deepcpg.models.utils.**is_output_layer**(*layer*, *model*)
Test if *layer* is an output layer.

deepcpg.models.utils.**load_model**(*model_files*, *custom_objects={'ScaledSigmoid': <class 'deepcpg.models.utils.ScaledSigmoid'>}*, *log=None*)
Load Keras model from a list of model files.

Loads Keras model from list of filenames, e.g. from *search_model_files*. *model_files* can be single HDF5 file, or JSON and weights file.

Parameters

model_file: list Input model file names.

custom_object: dict Custom objects for loading models that were trained with custom objects, e.g. *ScaledSigmoid*.

Returns

Keras model.

deepcpg.models.utils.**predict_generator**(*model*, *generator*, *nb_sample=None*)
Predict model outputs using generator.

Calls *model.predict* for at most *nb_sample* samples from *generator*.

Parameters

model: Keras model Model to be evaluated.

generator: generator Data generator.

nb_sample: int Maximum number of samples.

Returns

list list [*inputs*, *outputs*, *predictions*].

deepcpg.models.utils.**read_from**(*reader*, *nb_sample=None*)
Read *nb_sample* samples from *reader*.

deepcpg.models.utils.**save_model**(*model*, *model_file*, *weights_file=None*)
Save Keras model to file.

If *model_file* ends with '.h5', saves model description and model weights in HDF5 file. Otherwise, saves JSON model description in *model_file* and model weights in *weights_file* if provided.

Parameters

model Keras model.

model_file: str Output file.

weights_file: str Weights file.

deepcpg.models.utils.**search_model_files**(*dirname*)
Search model files in given directory.

Parameters

dirname: str Directory name

Returns

Model JSON file and weights if existing, otherwise HDF5 file. None if no

model files could be found.

model.cpg

CpG models.

Provides models trained with observed neighboring methylation states of multiple cells.

class deepcpg.models.cpg.**CpgModel**(**args*, ***kwargs*)
Abstract class of a CpG model.

inputs(*cpg_wlen*, *replicate_names*)
Return list of Keras model inputs.

class deepcpg.models.cpg.**FcAvg**(**args*, ***kwargs*)
Fully-connected layer followed by global average layer.

```
Parameters: 54,000
Specification: fc[512]_gap
```

class deepcpg.models.cpg.**RnnL1**(*act_replicate='relu'*, **args*, ***kwargs*)
Bidirectional GRU with one layer.

```
Parameters: 810,000
Specification: fc[256]_bgru[256]_do
```

class deepcpg.models.cpg.**RnnL2**(*act_replicate='relu'*, **args*, ***kwargs*)
Bidirectional GRU with two layers.

```
Parameters: 1,100,000
Specification: fc[256]_bgru[128]_bgru[256]_do
```

deepcpg.models.cpg.**get**(*name*)
Return object from module by its name.

deepcpg.models.cpg.**list_models**()
Return the name of models in the module.

model.dna

DNA models.

Provides models trained with DNA sequence windows.

class deepcpg.models.dna.**CnnL1h128**(*nb_hidden=128*, **args*, ***kwargs*)
CNN with one convolutional and one fully-connected layer with 128 units.

```
Parameters: 4,100,000
Specification: conv[128@11]_mp[4]_fc[128]_do
```

class deepcpg.models.dna.**CnnL1h256**(**args*, ***kwargs*)
CNN with one convolutional and one fully-connected layer with 256 units.

```
Parameters: 8,100,000
Specification: conv[128@11]_mp[4]_fc[256]_do
```

class deepcpg.models.dna.**CnnL2h128**(*nb_hidden=128*, **args*, ***kwargs*)
CNN with two convolutional and one fully-connected layer with 128 units.

```
Parameters: 4,100,000
Specification: conv[128@11]_mp[4]_conv[256@3]_mp[2]_fc[128]_do
```

class deepcpg.models.dna.**CnnL2h256**(**args*, ***kwargs*)
CNN with two convolutional and one fully-connected layer with 256 units.

```
Parameters: 8,100,000
Specification: conv[128@11]_mp[4]_conv[256@3]_mp[2]_fc[256]_do
```

class deepcpg.models.dna.**CnnL3h128**(*nb_hidden=128*, **args*, ***kwargs*)
CNN with three convolutional and one fully-connected layer with 128 units.

```
Parameters: 4,400,000
Specification: conv[128@11]_mp[4]_conv[256@3]_mp[2]_conv[512@3]_mp[2]_
               fc[128]_do
```

class deepcpg.models.dna.**CnnL3h256**(**args*, ***kwargs*)
CNN with three convolutional and one fully-connected layer with 256 units.

```
Parameters: 8,300,000
Specification: conv[128@11]_mp[4]_conv[256@3]_mp[2]_conv[512@3]_mp[2]_
               fc[256]_do
```

class deepcpg.models.dna.**CnnRnn01**(**args*, ***kwargs*)
Convolutional-recurrent model.

Convolutional-recurrent model with two convolutional layers followed by a bidirectional GRU layer.

```
Parameters: 1,100,000
Specification: conv[128@11]_pool[4]_conv[256@7]_pool[4]_bgru[256]_do
```

class deepcpg.models.dna.**DnaModel**(**args*, ***kwargs*)
Abstract class of a DNA model.

inputs(*dna_wlen*)
Return list of Keras model inputs.

class deepcpg.models.dna.**ResAtrous01**(**args*, ***kwargs*)
Residual network with Atrous (dilated) convolutional layers.

Residual network with Atrous (dilated) convolutional layer in bottleneck units. Atrous convolutional layers allow to increase the receptive field and hence better model long-range dependencies.

```
Parameters: 2,000,000
Specification: conv[128@11]_mp[2]_resa[3x128|3x256|3x512|1x1024]_gap_do
```

He et al., 'Identity Mappings in Deep Residual Networks.' Yu and Koltun, 'Multi-Scale Context Aggregation by Dilated Convolutions.'

class deepcpg.models.dna.**ResConv01**(**args*, ***kwargs*)
Residual network with two convolutional layers in each residual unit.

```
Parameters: 2,800,000
Specification: conv[128@11]_mp[2]_resc[2x128|1x256|1x256|1x512]_gap_do
```

He et al., 'Identity Mappings in Deep Residual Networks.'

class deepcpg.models.dna.**ResNet01**(**args*, ***kwargs*)
Residual network with bottleneck residual units.

```
Parameters: 1,700,000
Specification: conv[128@11]_mp[2]_resb[2x128|2x256|2x512|1x1024]_gap_do
```

He et al., 'Identity Mappings in Deep Residual Networks.'

class deepcpg.models.dna.**ResNet02**(**args*, ***kwargs*)
Residual network with bottleneck residual units.

```
Parameters: 2,000,000
Specification: conv[128@11]_mp[2]_resb[3x128|3x256|3x512|1x1024]_gap_do
```

He et al., 'Identity Mappings in Deep Residual Networks.'

deepcpg.models.dna.**get**(*name*)
Return object from module by its name.

deepcpg.models.dna.**list_models**()
Return the name of models in the module.

Python Module Index

d

s

Index

H

I

J

K

L

M

O

P

R

S

T

U

V

W