Deep Bottleneck Documentation

Deep Bottleneck study project team

Aug 15, 2018

Contents

CHAPTER 1

Big Picture

The new glossary helping you to get the bigger picture.

1.1 Entropy & Mutual Information

including sufficient statistic and data processing inequality

1.2 What is this mysterious information bottleneck?

including distortion theory, lossless and lossy compression, Kullback-Leibler divergence, information plane, bounds

1.3 An Introduction into Neural Networks

including single and multilayer perceptron, Connectivity Matrix, Backpropagation, etc.

1.4 Basic Maths

including Probability Theory, Markov Chains

CHAPTER 2

Contributing

2.1 Extending the framework

There are several possibilities to extend the framework. In the following the structure of the framework is shown to allow an easy extension of the basic modules. There are five types of modules that can be included quite easy, they are listed in the table below: Each module requires a module level `load` method to be defined, that passes the hyperparameters from the sacred configuration to the constructor of the class.

dataset The datasets live in the `deep_bottleneck.dataset` folder and require a load-method returning a training and a test dataset.

model The models live in in the `deep_bottleneck.model` folder and require a `load`-method as well. But in this case the load-method returns a trainable keras-model.

estimator The mutual information estimators live in the `deep_bottleneck.mi_estimator` folder and require a load-method as well. The `load`-method should return an estimator that is able to compute the mutual information based on a dataset and is described in more detailed by a hyperparameter called `discretization_range`.

callback Callbacks can be used for different kinds of tasks. They live in the `deep_bottleneck.callbacks` folder and are used to save the needed information during the training or to influence the training process (e.g. early stopping). They need to inherit from `keras.callbacks.Callback`.

plotter Plotters are using the saved data of the callbacks to create the different plots. They live in the `deep_bottleneck.plotter` folder and need a load method returning a plotter-class inheriting from `deep_bottleneck.plotter.base.BasePlotter`.

To add a new module, it needs to be added into the respective folder. Then the configuration parameter needs to be set to the import path of the module. If the path is correctly defined and the module has a matching interface, it will automatically be imported in `experiment.py` and conduct its tasks. More about the interfaces and the existing methods in the *API-documentation*.

2.2 Git workflow

This workflow describes the process of adding code to the repository.

1. Describe what you want to achieve in an issue.
2. Pull the master to get up to date.
   (a) `git checkout master`
   (b) `git pull`
3. Create a new local branch with `git checkout -b <name-for-your-branch>`. It can make sense to prefix your branch with a description like `feature` or `fix`.
4. Solve the issue, most probably in several commits.
5. In the meantime there might have been changes on the master branch. So you need to merge these changes into your branch.
   (a) `git checkout master`
   (b) `git pull` to get the latest changes.
   (c) `git checkout <name-for-your-branch>`
   (d) `git merge master`. This might lead to conflicts that you have to resolve manually.
6. Push your branch to github with `git push origin <name-for-your-branch>`.
7. Go to github and switch to your branch.
8. Send a pull request from the web UI on github.
9. After you received comments on your code, you can simply update your pull request by pushing to the same branch again.
10. Once your changes are accepted, merge your branch into master. This can also be done by the last reviewer that accepts the pull request.

2.2.1 Git commit messages

Have a look at this guideline.

Most important:

- Single line summary starting with a verb (50 characters)
- Longer summary if necessary (wrapped at 72 characters).

Editors like `vim` enforce these constraints automatically.

2.3 Style Guide

Follow **PEP 8** styleguide. It is worth reading through the entire styleguide, but the most importand points are summarized here.

2.3.1 Naming

- Functions and variables use `snake_case`
- Classes use `CamelCase`
- Constants use `CAPITAL_SNAKE_CASE`

2.3.2 Spacing

Spaces around infix operators and assignment

- `a + b` not `a+b`
- `a = 1` not `a=1`

An exception are keyword arguments

- `some_function(arg1=a, arg2=b)` not `some_function(arg1 = a, arg2 = b)`

Use one space after separating commas

- `some_list = [1, 2, 3]` not `some_list = [1,2,3]`

In general PyCharm's auto format (Ctrl + Alt + l) should be good enough.

2.3.3 Type annotation

Since Python 3.5 type annotation are supported. They make sense for public interfaces, that should be kept consistent.

```
def add(a:  int, b:  int) -> int:
```

2.3.4 Docstrings

Use Google Style for docstrings in everything that has a somewhat public interface.

2.3.5 Clean code

And here our non exhaustive list to guidelines to write cleaner code.

1. Use meaningful variable names
2. Keep your code DRY (Don't repeat yourself) by abstracting into functions and classes.
3. Keep everything at the same level of abstraction
4. Functions without side effects
5. Functions should have a single responsibility
6. Be consistent, stick to conventions, use a styleguide
7. Use comments only for what cannot be described in code
8. Write comments with care, correct grammar and correct punctuation
9. Write tests if you write a module

2.4 Experiment workflow

1. Define a hypothesis
2. Define set of parameters that is going to stay fixed
3. Define parameter to change (including possible values for the parameter)
4. Create a meaningful name for the experiment (group of experiment, name of parameter tested)
5. Make sure you set a seed (Pycharm: in run options append: "with seed=0")
6. Program experiment (set parameters) using our framework
7. Commit your changes locally to obtain commit hash: this is going to be logged by sacredboard
8. Make sure your experiment is logged to the database
9. Start the experiment
10. Interpret and document results in a notebook. Include relevant plots using the artifact viewer. Make sure the notebook is completely executed.
11. Move your notebook to *docs/experiments*, so it will be automatically included in the documentation.
12. Push your local branch to github - to make all commits available to everyone

2.5 Documentation

To build the documentation run:

```
$ cd docs
$ make html
```

A short restructeredText reference. There is also a longer video tutorial

If you added new packages and want to add them to the API documentation use:

```
$ sphinx-apidoc -o docs/api_doc/ deep_bottleneck deep_bottleneck/credentials.py deep_
↪bottleneck/experiment.py deep_bottleneck/demo.py
```

Make sure to change the header of `modules.rst` back to "API Documentation".

CHAPTER 3

Glossary

3.1 Information Theory Basics

This glossary is mainly based on MacKay's *Information Theory, Inference and Learning Algorithms*. If not marked otherwise, all information below can be found there.

Prerequisites: random variable, probability distribution

A major part of information theory is persuing answers to problems like "how to measure information content", "how to compress data" and "how to communicate perfectly over imperfect communcation channels".

At first we will introduce some basic definitions.

3.1.1 The Shannon Information Content

The Shannon information content of an outcome x is defined as

$$h(x) = \log_2 \frac{1}{P(x)}.$$

The unit of this measurement is called "bits", which does not allude to 0s and 1s.

3.1.2 Ensemble

We extend the notion of a random variable to the notion of an **ensemble**. An **ensemble** X is a triplet (x, A_X, P_X), where x is just the variable denoting an outcome of the random variable, A_X is the set of all possible outcomes and P_X is the defining probability distribution.

3.1.3 Entropy

Let X be a random variable and A_X the set of possible outcomes. The entropy is defined as

$$H(X) = \sum_x p(x)\ log_2\left(\frac{1}{p(x)}\right).$$

The **entropy** describes how much we know about the outcome before the experiment. This means

3.1.4 Entropy for two dependent variables

Let X and Y be two dependent random variables.

joint entropy

$$H(X,Y) = \sum_{x,y} p(x,y)\ log_2\left(\frac{1}{p(x,y)}\right)$$

conditional entropy if one variable is observed

$$H(X|y=b) = \sum_x p(x|y=b)\ log_2\left(\frac{1}{p(x|y=b)}\right)$$

conditional entropy in general

$$\begin{aligned} H(X|Y) &= \sum_y p(y)\ H(X|y=y) \\ &= \sum_{x,y} p(x,y)\ log_2\left(\frac{1}{p(x|y)}\right) \end{aligned}$$

chain rule for entropy

$$\begin{aligned} H(X,Y) &= H(X) + H(Y|X) \\ &= H(Y) + H(X|Y) \end{aligned}$$

3.1.5 Mutual Information

Let X and Y be two random variables. The **mutual information** between these variables is then defined as

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \\ &= H(X) + H(Y) - H(X,Y) \end{aligned}$$

The **mutual information** describes how much uncertainty about the one variable remains if we observe the other. It holds that

$$I(X;Y) = I(Y;X) I(X;Y) \geq 0$$

The following figure gives a good overview:

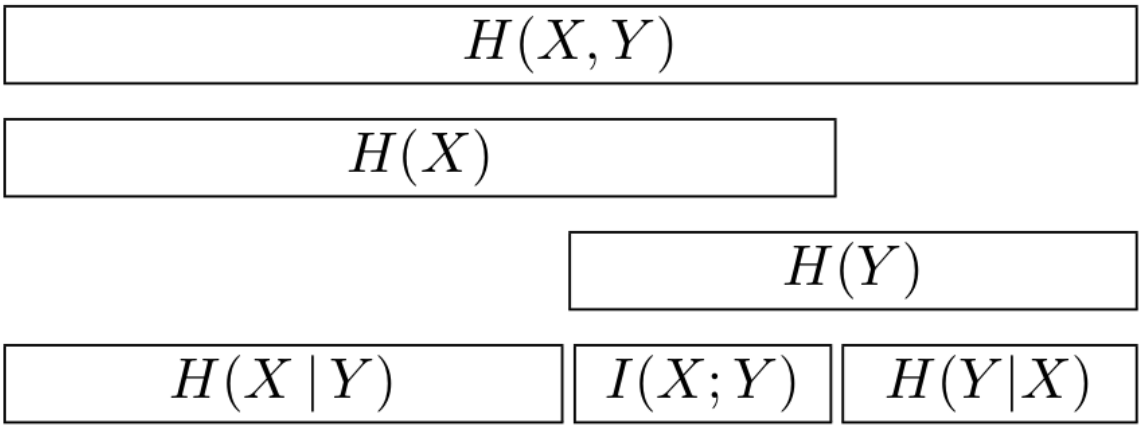


Fig. 1: Mutual information overview.

3.1.6 Kullback-Leibler divergence

Let X be a random variable and $p(x)$ and $q(x)$ two probability distributions over this random variable. The **Kullback-Leibler divergence** is defined as

$$D_{KL}(p||q) = \sum_x p(x)\ log_2\left(\frac{p(x)}{q(x)}\right)$$

The **Kullback-Leibler divergence** is often called *relative entropy* and denotes "something" like a distance between two distributions:

$$D_{KL}(p||q) \geq 0$$
$$D_{KL}(p||q) = 0 \iff p = q$$

Yet it is not a real distance as symmetry is not given.

3.1.7 Typicality

We introduce the "Asymptotic equipartion" principle which can be seen as a *law of large numbers*. This principle denotes that for an ensemble of N independent and identically distributed (i.i.d.) random variables $X^N \equiv (X_1, X_2, \ldots, X_N)$, with N sufficiently large, the outcome $x = (x_1, x_2, \ldots, x_N)$ is almost certain to belong to a subset of $\mathcal{A}_X^N$ with $2^{NH(X)}$ members, each having a probability that is 'close to' $2^{-NH(X)}$.

The typical set is defined as

$$T_{N\beta} \equiv \{x \in \mathcal{A}_X^N : |\frac{1}{N}\log_2\frac{1}{P(x)} - H| < \beta\}.$$

The parameter β sets how close the probability has to be to 2^{-NH} in order to call an element part of the typical set, $\mathcal{A}_X$ is the alphabet for an arbitrary ensemble X.

3.1.8 Shannon's Source Coding Theorem

3.2 Mathematical Terms in Tishby's Experiments

3.2.1 Stochastic Gradient Descent

3.2.2 Spherical Harmonic power spectrum [Tishby (2017) 3.1 Experimental setup]

TODO

3.2.3 O(3) rotations of the sphere [Tishby (2017) 3.1 Experimental setup]

TODO

CHAPTER 4

Literature

This is just to demonstrate a citation. See *[TZ15]* for an introduction to our topic.

CHAPTER 5

Literature Summary

5.1 1. THE INFORMATION BOTTLENECK METHOD (Tishby 1999)

Tishby, N., Pereira, F. C., & Bialek, W. (2000). The information bottleneck method. arXiv preprint physics/0004057.

5.1.1 1.1. Glossary

5.1.2 1.2. Structure

5.1.3 1.3. Criticisim

5.1.4 1.4. Todo List

5.2 2. DEEP LEARNING AND THE INFORMATION BOTTLENECK PRINCIPLE (Tishby 2015)

Tishby, N., & Zaslavsky, N. (2015, April). Deep learning and the information bottleneck principle. In Information Theory Workshop (ITW), 2015 IEEE (pp. 1-5). IEEE.

5.2.1 2.1. Glossary

5.2.2 2.2. Structure

5.2.3 2.3. Criticisim

5.2.4 2.4. Todo List

5.3 3. OPENING THE BLACK BOX OF DEEP NEURAL NETWORKS VIA INFORMATION (Tishby 2017)

Shwartz-Ziv, R., & Tishby, N. (2017). Opening the black box of deep neural networks via information. arXiv preprint arXiv:1703.00810.

5.3.1 3.1. Glossary

5.3.2 3.2. Structure

5.3.3 3.3. Criticisim

5.3.4 3.4. Todo List

5.4 4. ON THE INFORMATION BOTTLENECK THEORY OF DEEP LEARNING (Saxe 2018)

Saxe, A. M., Bansal, Y., Dapello, J., Advani, M., Kolchinsky, A., Tracey, B. D., & Cox, D. D. (2018, May). On the information bottleneck theory of deep learning. In International Conference on Learning Representations.

5.4.1 4.1. Glossary

5.4.2 4.2. Structure

5.4.3 4.3. Criticisim

5.4.4 4.4. Todo List

5.5 5. ON THE INFORMATION BOTTLENECK THEORY OF DEEP LEARNING

[Andrew2017]

5.5.1 Key Points of the paper:

- none of the following claims of Tishby (*[TZ15]*) holds in the general case:

1. deep networks undergo two distinct phases consisting of an initial fitting phase and a subsequent compression phase
2. the compression phase is causally related to the excellent generalization performance of deep networks
3. the compression phase occurs due to the diffusion-like behavior of stochastic gradient descent

- the osberved compression is different based on the activation function: double-sided saturating nonlinearities like tanh yield a compression phase, but linear activation functions and single-sided saturating nonlinearities like ReLU do not.

- there is no evident causal connection between compression and generalization.

- the compression phase, when it exists, does not arise from stochasticity in training.

- when an input domain consists of a subset of task-relevant and task-irrelevant information, the task-irrelevant information compress although the overall information about the input may monotonically increase with training time. This compression happens concurrently with the fitting process rather than during a subsequent compression period.

5.5.2 Most important Experiments:

1. Tishby's experiment reconstructed:
   - 7 fully connected hidden layers of width 12-10-7-5-4-3-2
   - trained with stochastic gradient descent to produce a binary classification from a 12-dimensional input
   - 256 randomly selected samples per batch
   - mutual information is calculated by binning th output activations into 30 equal intervals between -1 and 1
   - trained on Tishby dataset
   - tanh-activation function
2. Tishby's experiment reconstructed with ReLu activation:
   - 7 fully connected hidden layers of width 12-10-7-5-4-3-2
   - trained with stochastic gradient descent to produce a binary classification from a 12-dimensional input

- 256 randomly selected samples per batch
- mutual information is calculated by binning th output activations into 30 equal intervals between -1 and 1
- ReLu-activation function

3. Tanh-activation function on MNIST:
   - 6 fully connected hidden layers of width 784 - 1024 - 20 - 20 - 20 - 10
   - trained with stochastic gradient descent to produce a binary classification from a 12-dimensional input
   - non-parametric kernel density mutual information estimator
   - trained on MNIST dataset
   - tanh-activation function
4. ReLu-activation function on MNIST:
   - 6 fully connected hidden layers of width 784 - 1024 - 20 - 20 - 20 - 10
   - trained with stochastic gradient descent to produce a binary classification from a 12-dimensional input
   - non-parametric kernel density mutual information estimator
   - trained on MNIST dataset
   - ReLu-activation function

5.5.3 Presentation:

Google slides link

CHAPTER 6

User guide

6.1 Installation

6.1.1 Environment

To run the experiment you need to install the required dependencies. We highly recommend that you use a virtual environment as provided by conda or pipenv.

Then in your environment run:

```
$ pip install -r requirements/dev.txt
```

6.1.2 Sacred setup

When running experiments, the hyperparameters, metrics and plots are managed through Sacred and are stored in a mongoDB database. Though you can setup your mongoDB instance however you want, it is most conveniently done through the provided Docker files. This will not only get you started with mongoDB in no time, but will also set up a mongo-express interface to conveniently manage your database and sacredboard to monitor your runs. In order to use them you need to

1. Install Docker Engine.
2. Install Docker Compose.
3. Navigate to the directory with the setup files.

```
$ cd infractructure/sacred_setup
```

4. Edit the `.env` file. This file is hidden by default, but you can still edit it with any text editor, e.g. by `vi .env`. Replace all values in angle brackets with meaningful and secure values.
5. Run docker-compose:

```
docker-compose up -d
```

This will pull the necessary containers from the internet and build them. This may take several minutes. Afterwards mongoDB should be up and running. `mongo-express` should now be available on port `8081`, accessible by the user and password you set in the `.env` file (`ME_CONFIG_BASICAUTH_USERNAME` and `ME_CONFIG_BASICAUTH_PASSWORD`). Sacredboard should be available on port `5000`.

The current setup is optimized for a team that collaboratively stores results on a remote server. When running the experiments locally for yourself, you should change the port mapping in the `docker-compose.yml` file to only map to localhost, such that you do not expose your database to the internet. Simply prefix all port mappings with localhost, e.g. replace:

```
ports:
  - 5000:5000
```

by

```
ports:
  - 127.0.0.1:5000:5000
```

5. In a final step, you need to tell sacred how to connect to the database. Edit file `deep_bottleneck/credentials.py`, again replacing all values in angle brackets by the values you actually set in the `.env` file. Additionally, you have to provide the IP address of the server your database is running on, which is either the address given by your server provider or `127.0.0.1` when running mongo locally.

6. You are ready to run some exciting experiments!

6.1.3 Importing and exporting from mongoDB

The following section is meant to help you migrate your data from one server to another. If you are just starting you can skip this section.

To export data from your mongo container run

```
$ docker run --rm --link <container_id>:mongo --network <network_id> -v /root/dump:/
↪backup mongo bash -c 'mongodump --out /backup --uri mongodb://<username>:<password>
↪@mongo:27017/?authMechanism=SCRAM-SHA-1'
```

make sure you you create the output folder, in this case `/root/dump` beforehand. You also need to look up the id of your current mongo container using `docker ls` and find the id of the network is running is using `docker network ls`. Then replace `<username>` and `<password>` by the values you originally set in your `.env`.

To import data again run following the same steps as above.

```
$ docker run --rm --link <container_id>:mongo --network <network_id> -v /root/dump:/
↪backup mongo bash -c 'mongorestore /backup --uri mongodb://<username>:<password>
↪@mongo:27017/?authMechanism=SCRAM-SHA-1'
```

6.2 How to use the framework

6.2.1 Running experiments

The idea of the project is based on the concepts presented by Tishby. To reproduce the basic setup of the experiments one can simply start `experiment.py`.

If all the required packages are installed properly and the program is started, different things should happen.

1. First the required modules of the framework are imported based on the defined configuration (more about configurations in "Adding new Experiments").
2. A neural network is trained using the defined dataset. The progress of this process is also logged in the console.
3. During the training process the required data is saved in regular time-steps to the local filesystem.
4. Given the saved data (e.g. the activations) it is possible to compute the mutual information of the different layer and the input/output.
5. Using this different plots as e.g. the information plane plot are created and saved simultaneously in the filesystem and in the database. The results of the experiments can be looked up either in the `deep_bottleneck/plots` folder (only the plots of the last runs are saved) or using `eval_tools` as described below.

6.2.2 Evaluation tools

To make the rich results generated by the experiments accessible, we created an *evaluation tool*. It lets you query experiments based on id, name or other configuration parameters and lets you view the generated plots, metrics and videos conveniently in Jupyter notebooks. To get you started have a look at deep_bottleneck/eval_tools_demo.ipynb.

6.2.3 Adding new experiments (config)

Configuration

During the exploration of Tishby's idea already a lot of experiments have been done, but there are still many things one can do using this framework. To define a new experiment a new configuration needs to be added. The existing configurations are saved in the `deep_bottleneck/configs` folder. To add a new configuration a new `JSON` file is required. The currently relevant parts of the configuration and their effects are explained in the following table.

epochs Number of epochs the model is trained for. Most of the experiments for the harmonics dataset used 8000 epochs.

batch_size Batch size used during the training process. Most dominant batch size in our experiments was 256.

architecture Architecture of the trained model. Defined as a list of integers, where every integer defines the number of neurons in one layer. It is important to notify that an additional readout layer is automatically added (with the number of neurons corresponding to the number of classes in the dataset). The basic architecture for the harmonics dataset is [10, 7, 5, 4, 3].

optimizer The optimizer used for the training of the neural network. Possible values are "sgd", or "adam".

learning_rate The learning rate of the optimizer. Default values are 0.0004 for harmonics and 0.001 for mnist.

calculate_mi_for The calculate_mi_for parameter defines the dataset that is used for the mutual information computation. It can be done either for the training data (value: "training"), test data ("test") or the full dataset ("full_dataset").

activation_fn The activation-function used to train the model. The following activation function are implemented: `tanh`, `relu`, `sigmoid`, `softsign`, `softplus`, `leaky_relu`, `hard_sigmoid`, `selu`, `relu6`, `elu` and `linear`.

model The parameter which defines the basic model-choice. Currently only different architectures of feed-foreward-networks can be used. So the possible choices right now are `models.feedforward` and `models.feedforward_batchnorm`, the actual architecture is defined by the architecture parameter.

dataset The parameter which defines the dataset used for training. Currently implemented datasets are `harmonics`, `mnist`, `fashion_mnist` and `mushroom`.

estimator The estimator used for the computation of the mutual information. Because mutual information cannot be computed analytically for more complex networks, it is necessary to estimate it. Possible estimators are `mi_estimator.binning`, `mi_estimator.lower`, `mi_estimator.upper`.

discretization_range The different estimators have a different hyperparameter to add artificial noise to the estimation. This parameter is used as a placeholder for the different hyperparameter. A typical value is 0.07 for `binning` and 0.001 for `upper` and `lower`.

callbacks A list of additional callbacks as for example early stopping. Needs to defined as a list of paths to the callbacks, as e.g. `[callbacks.early_stopping_manual]`.

n_runs Number of runs the experiment is repeated. The results will be averaged over all runs to compensate for outliers.

Executing multiple experiments

Using these parameters one should be able to define experiments as desired. To execute the experiment(s) one could simply start des experiment.py but mainly due to our usage of external hardware resources (Sun grid engine) we had to develop another way to execute experiments. We created two python files: `run_experiment.py` and `run_experiment_local.py`, which can run either a single experiment or a group of experiments. For the local execution of experiments with `run_experiment_local.py` one needs to switch to the deep_bottleneck folder by:

```
$ cd deep_bottleneck
```

and then execute experiments by either pointing to a specific `JSON` file defining the experiment, e.g.:

```
$ python run_experiments_local.py -d configs/basic.json
```

or pointing at a directory containing all the experiments one wants to execute, e.g.:

```
$ python run_experiments_local.py -d configs/mnist
```

In that case all the `JSONs` in the folder and in its sub-folders are recursively executed.

Running experiment on the Sun grid engine

In case one uses a sun grid engine to execute the experiments it is possible to start `run_experiments.py` on the engine in the same way with as described above. The experiments will get submitted to the engine using `qsub`. In that case it is important to make sure that an /output/-folder exists on the directory-level of the `experiment.sge` file.

Additionally it might be important to run experiments that are repeatable and will return the same results in every run. Because the basic step of the framework is to train a neural network, including some kind of randomness the results of two runs might be different even though they are based on the same configuration. To avoid misconceptions it is possible to set a seed for each experiment, simply by using:

```
$ python experiment.py with seed=0
```

(the exact seed is arbitrary, it just needs to be consistent). In case that one of the `run_experiment` files is used this step is done for you, but even in the other cases some IDEs allow to set script-parameters for normal executions of a specific file, such that it is not required to start the `experiment.py` out of the command-line.

CHAPTER 7

Experiments

7.1 Description of cohorts

The experiments are structured in different cohort, containing one specific variation of parameters. To show the aim of the cohorts and to simplify the access of the saved artifacts using the artifact-viewer the following table offers a simple description for each cohort.

Cohort	Description
cohort_1	Comparison of upper, lower and binning as different estimator. Additionally the hyperparameter of the estimators are varied. All experiments are done for relu and tanh using sgd as optimizer.
cohort_2	Comparison of training-, test- and full-dataset as base for the mi-computation. All experiments are done for relu and tanh using sgd as optimizer.
cohort_3	Comparison of upper, lower and binning as different estimator. Additionally the hyperparameter of the estimators are varied. All experiments are done for relu and tanh using adam as optimizer.
cohort_4	Comparison of training-, test- and full-dataset as base for the mi-computation. All experiments are done for relu and tanh using Adam as optimizer.
cohort_5	Comparison of different standard activation functions. All experiments are done using adam as optimizer.
cohort_6	Comparison of basic architectures. All experiments are done for relu and tanh using adam as optimizer.
cohort_7	Comparison of different hyperparameter for max-norm regularization. All experiments are done for relu and tanh using adam as optimizer.
cohort_8	Comparison of architecture with batchnorm and without batchnorm. All experiments are done for relu and tanh using adam as optimizer.

7.2 Comparing activation functions for a minimal model

The opposing paper argues that input compression is merely an artifact of double saturated activation functions. We test this assumption for the minimal model in a numeric simulation. We look at the development of mututal information

with the input in a one neuron model with growing weights and compare different activation functions.

```
In [1]: import numpy as np
        np.random.seed(0)
        from scipy import stats
        import matplotlib.pyplot as plt
        import seaborn as sns
        import tensorflow as tf
        import tensorflow.contrib.eager as tfe
        tf.enable_eager_execution()
```

```
/home/jarno/.miniconda/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of
  from ._conv import register_converters as _register_converters
```

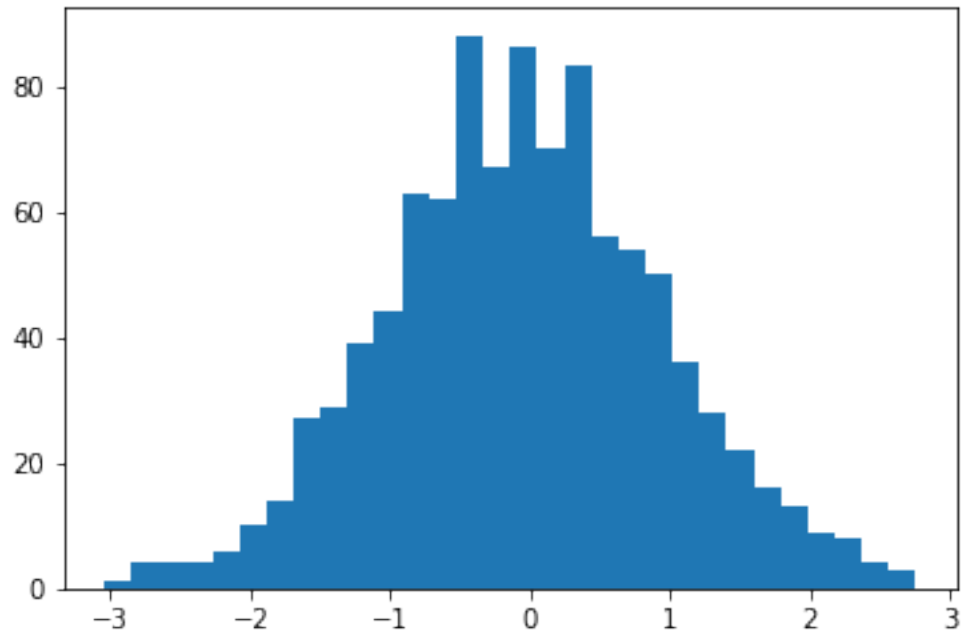
```
In [2]: weights = np.arange(0.1, 8, 0.1)
```

```
In [3]: weights
```

```
Out[3]: array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2, 1.3,
               1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6,
               2.7, 2.8, 2.9, 3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9,
               4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5. , 5.1, 5.2,
               5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6. , 6.1, 6.2, 6.3, 6.4, 6.5,
               6.6, 6.7, 6.8, 6.9, 7. , 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8,
               7.9])
```

The input is sampled from a standard normal distribution.

```
In [4]: input_distribution = stats.norm()
        input_ = input_distribution.rvs(1000)
        plt.hist(input_, bins=30);
```



The input is multiplied by the different weights. This is the pass from the input neuron to the hidden neuron.

```
In [5]: net_input = np.outer(weights, input_)
```

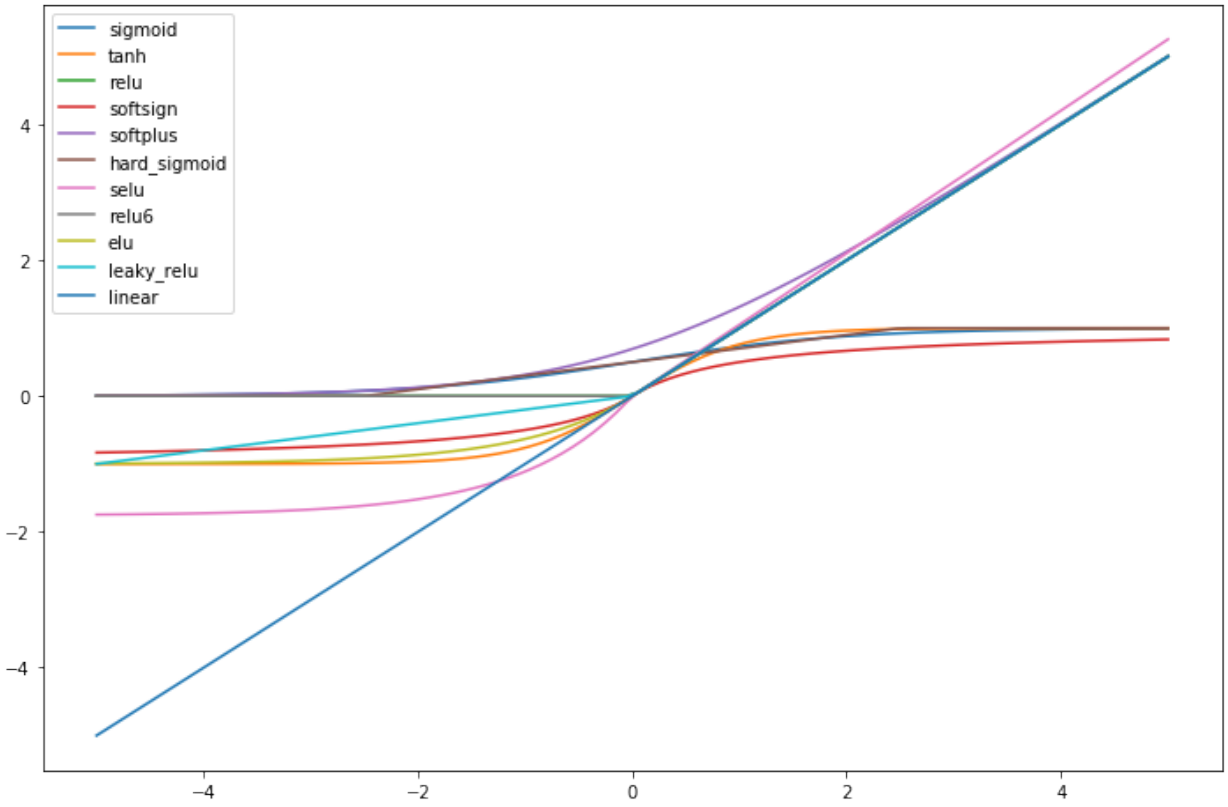
The activation functions we want to test.

```
In [6]: def hard_sigmoid(x):
            lower_bound = -2.5
```

```
    upper_bound = 2.5
    linear = 0.2 * x + 0.5
    linear[x < lower_bound] = 0
    linear[x > upper_bound] = 1
    return linear

def linear(x):
    return x

activation_functions = [tf.nn.sigmoid, tf.nn.tanh, tf.nn.relu, tf.nn.softsign, tf.nn.softplus
                        tf.nn.selu, tf.nn.relu6, tf.nn.elu, tf.nn.leaky_relu, linear]
```

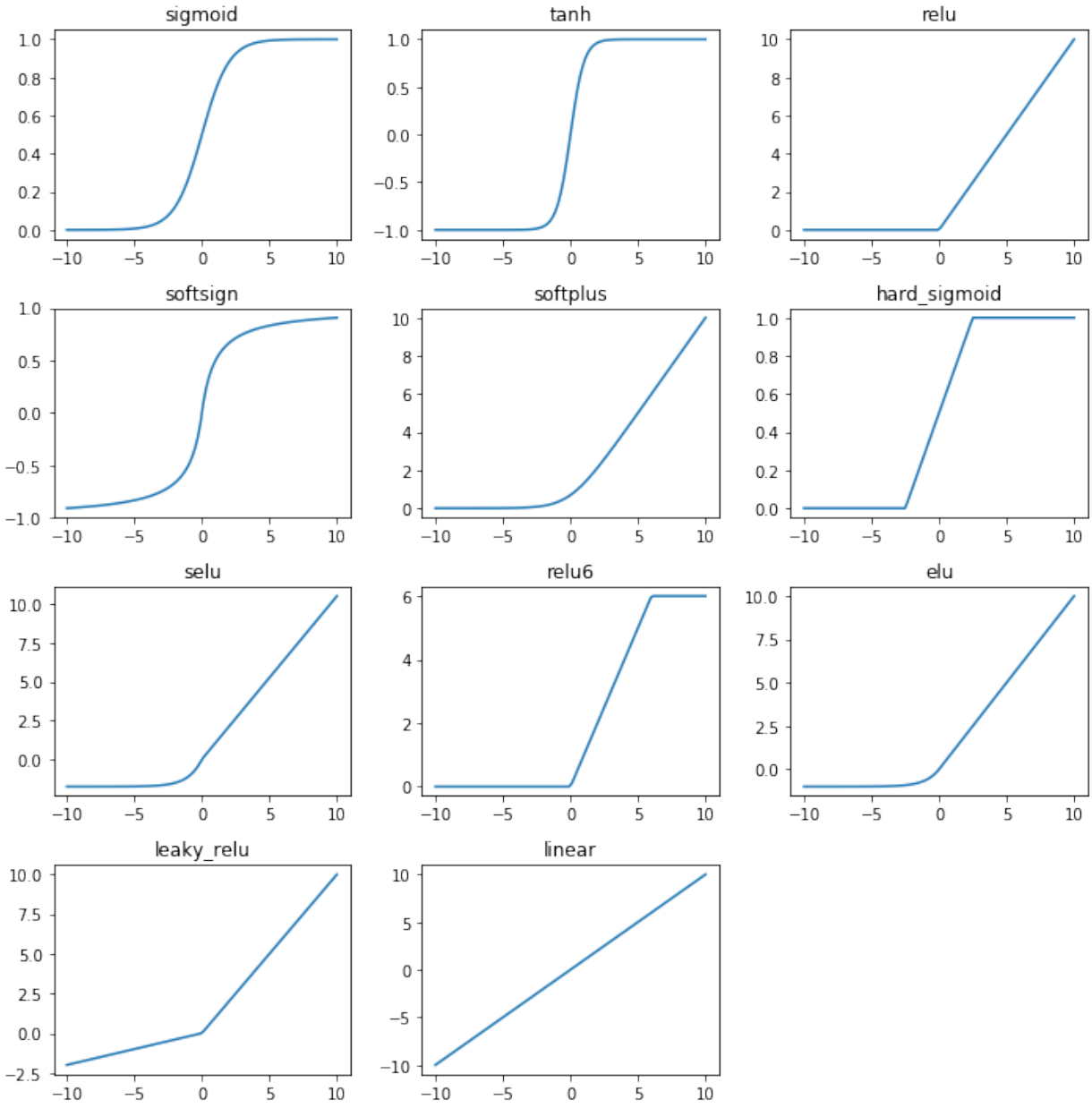
First we look at the shape of the different activation functions. We see that some are double saturated like `tanh` and some are not like `relu`.

```
In [7]: fig, ax = plt.subplots(figsize=(12, 8))
        for actvation_function in activation_functions:
            x = np.linspace(-5,5,100)
            ax.plot(x, actvation_function(x), label=actvation_function.__name__)
        plt.legend()
        plt.show()
```




```
In [8]: fig, axes = plt.subplots(nrows=int(len(activation_functions)/3)+1, ncols=3, figsize=(10, 10))
        axes = axes.flat

        for i, actvation_function in enumerate(activation_functions):
            x = np.linspace(-10,10,100)
            axes[i].plot(x, actvation_function(x))
            axes[i].set(title=actvation_function.__name__)

        # Remove unused plots.
```

```
        for ax in axes:
            if not ax.lines:
                ax.axis('off')


        plt.tight_layout()
        plt.show()
```



Apply the activation functions to the weighted inputs.

```
In [9]: outputs = {}
        for actvation_function in activation_functions:
            try:
                outputs[actvation_function.__name__] = actvation_function(net_input).numpy()
            except AttributeError:
                outputs[actvation_function.__name__] = actvation_function(net_input)
```

We now estimate the discrete mututal information between the input X and the activity of the hidden neuron Y, which is in this case also the output. $H(Y|X) = 0$, since Y is a deterministic function of X. Therefore

$$I(X;Y) = H(Y) - H(Y|X) \tag{7.1}$$
$$= H(Y) \tag{7.2}$$
$$\tag{7.3}$$

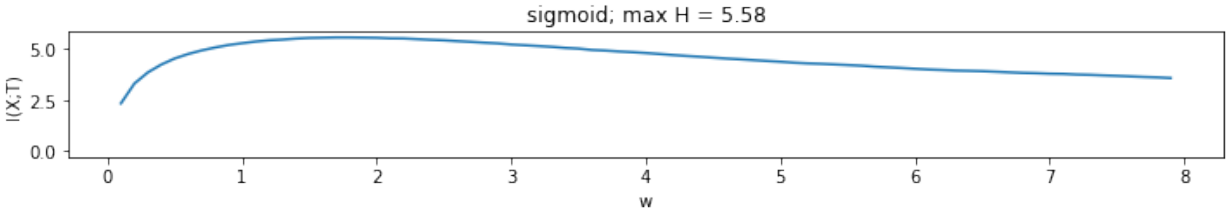
The entropy of the input is

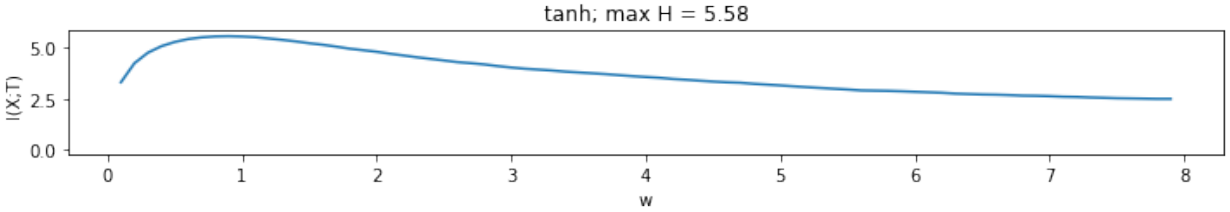
```
In [10]: dig, _ = np.histogram(input_, 50)
         print(f'{stats.entropy(dig, base=2):.2f} bits')
```

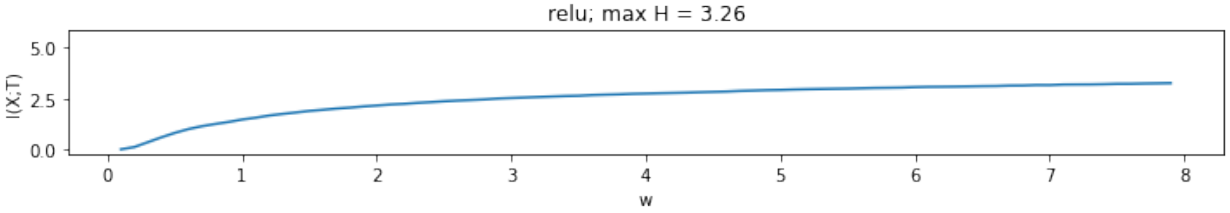
```
5.11 bits
```

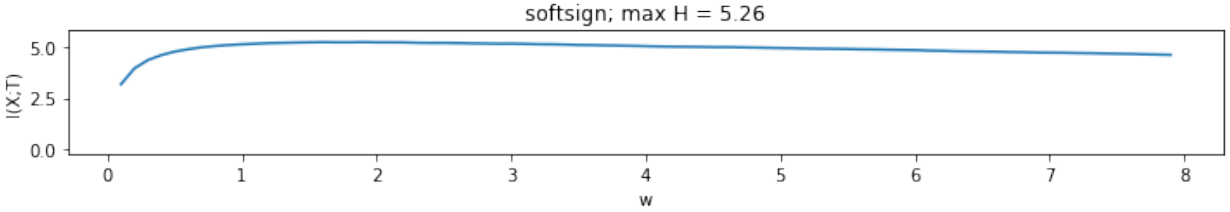
In the paper a fixed number of bins is evenly distributed between the minimum and the maximum activation over all weight values. The result below is indeed comparable to the paper and shows that mutual information decreases only in double saturated activation functions, while it increases otherwise.

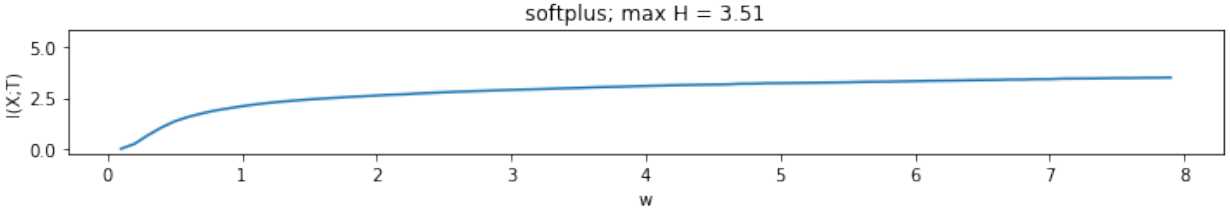
```
In [11]: fig, ax = plt.subplots(nrows=len(outputs), figsize=(10, 20), sharey=True)
         ax = ax.flat
         for ax_idx, (activation_function, Y) in enumerate(outputs.items()):
             min_acitivity = Y.min()
             max_acitivity = Y.max()

             mi = np.zeros(len(weights))
             for i in range(len(weights)):
                 bins = np.linspace(min_acitivity, max_acitivity, 50)
                 digitized, _ = np.histogram(Y[i], bins=bins)

                 mi[i] = stats.entropy(digitized, base=2)

             ax[ax_idx].plot(weights,  mi)
             ax[ax_idx].set(title=f'{activation_function}; max H = {mi.max():.2f}', xlabel='w', ylabe
         plt.tight_layout()
```

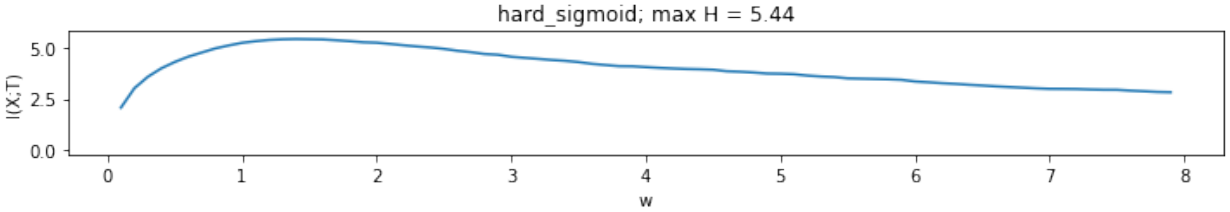

sigmoid; max H = 5.58
I(X;T)
5.0
2.5
0.0
0
1
2
3
4
5
6
7
8
w

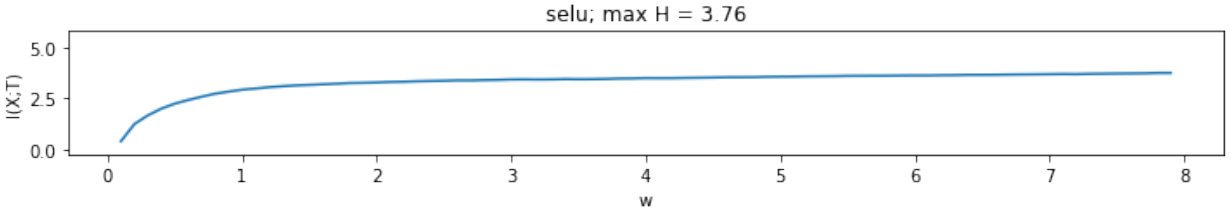

tanh; max H = 5.58
I(X;T)
5.0
2.5
0.0
0
1
2
3
4
5
6
7
8
w

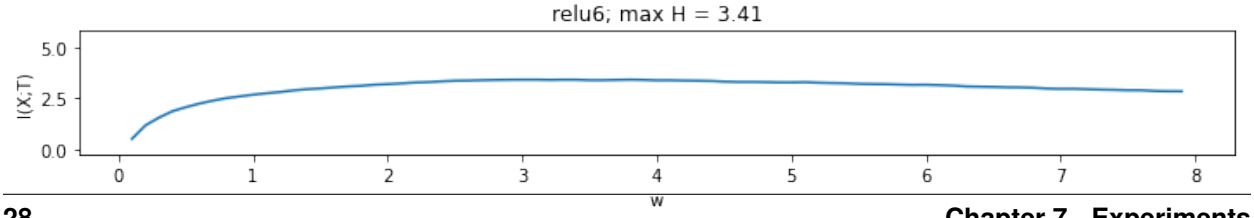

relu; max H = 3.26
I(X;T)
5.0
2.5
0.0
0
1
2
3
4
5
6
7
8
w

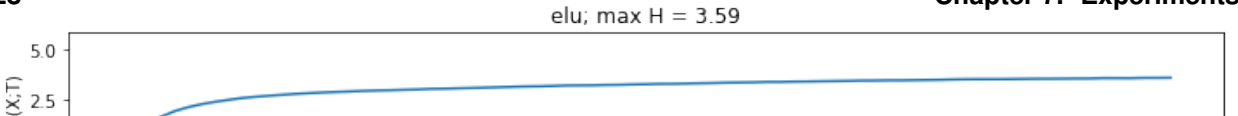

softsign; max H = 5.26
I(X;T)
5.0
2.5
0.0
0
1
2
3
4
5
6
7
8
w


softplus; max H = 3.51
I(X;T)
5.0
2.5
0.0
0
1
2
3
4
5
6
7
8
w


hard_sigmoid; max H = 5.44
I(X;T)
5.0
2.5
0.0
0
1
2
3
4
5
6
7
8
w

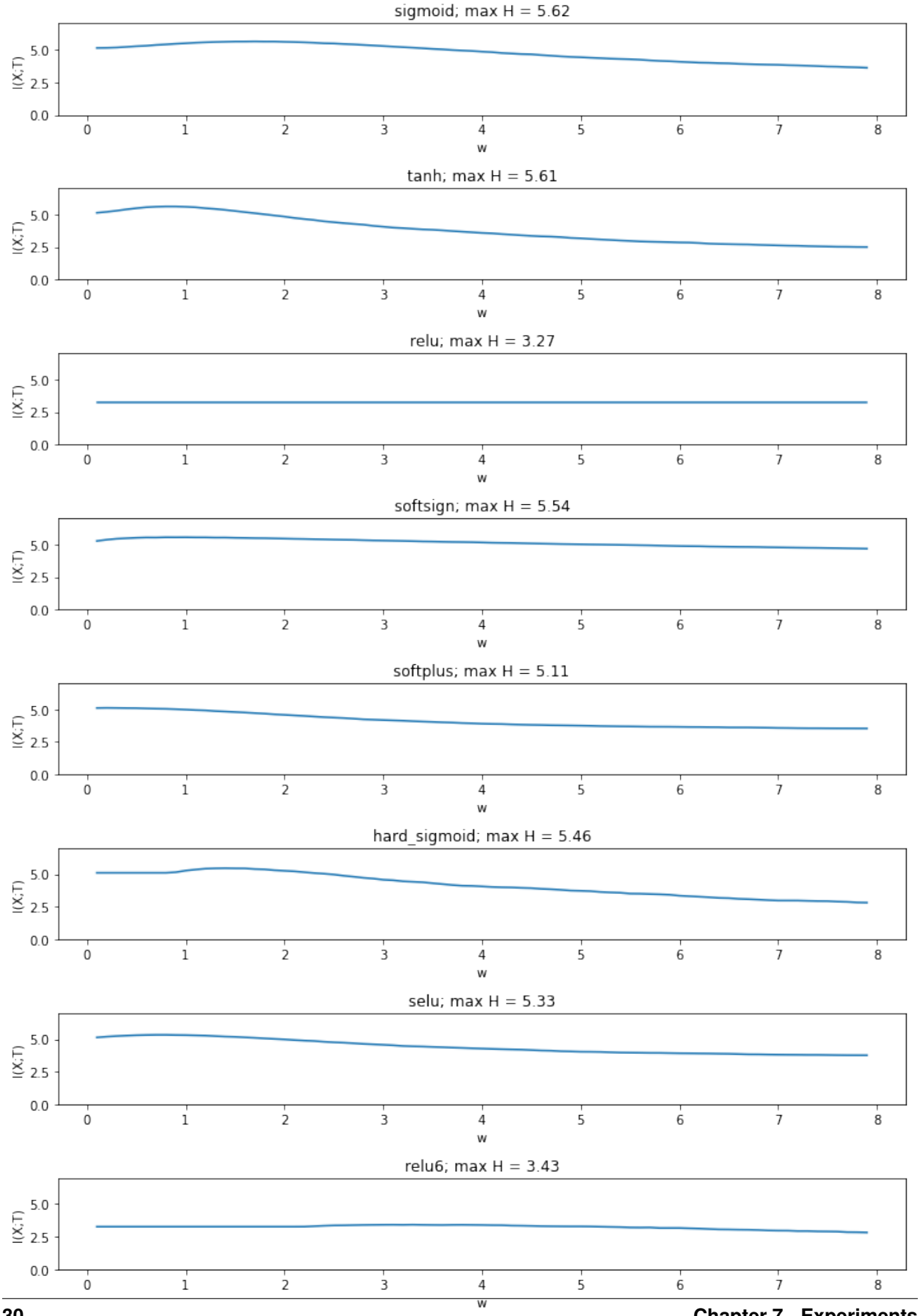

selu; max H = 3.76
I(X;T)
5.0
2.5
0.0
0
1
2
3
4
5
6
7
8
w


relu6; max H = 3.41
I(X;T)
5.0
2.5
0.0
0
1
2
3
4
5
6
7
8
w


elu; max H = 3.59
I(X;T)
5.0
2.5

Yet the fact that mutual information increases even in the linear case is a result of binning between the **minimum and the maximum activation of all neurons**. It raises the question whether this approch is sensible at all or whether binning boundaries should be determined for each simulated weight value seperately. We compare the approach with creating a fixed number of bins between the **minimum and the maximum activity for each weight**.

```
In [12]: fig, ax = plt.subplots(nrows=len(outputs), figsize=(10, 20), sharey=True)
         ax = ax.flat
         for ax_idx, (activation_function, Y) in enumerate(outputs.items()):

             mi = np.zeros(len(weights))
             for i in range(len(weights)):
                 digitized, _ = np.histogram(Y[i], bins=50)
                 mi[i] = stats.entropy(digitized, base=2)

             ax[ax_idx].plot(weights,  mi)
             ax[ax_idx].set(title=f'{activation_function}; max H = {mi.max():.2f}', xlabel='w', ylabe

         plt.tight_layout()
         plt.show()
```


sigmoid; max H = 5.62
tanh; max H = 5.61
relu; max H = 3.27
softsign; max H = 5.54
softplus; max H = 5.11
hard_sigmoid; max H = 5.46
selu; max H = 5.33
relu6; max H = 3.43
elu; max H = 5.22
I(X;T)
0.0
2.5
5.0
0
1
2
3
4
5
6
7
8
w

This gives a more sensible result. The mutual information is now constant in the linear case and has the same value as the entropy of the input. We now see that mutual information stays constant for many non saturated activation functions, while it still decreases for double saturated functions. Yet it also decreases for some non-double saturated functions such as `elu` and `softplus`.

Moreover, we see that some activation functions produce distributions with a higher maximum entropy than the input distribution. While it is known that the data processing inequality does no longer hold after the addition noise through binning, it should be investigated whether this is a systematic effect.

It also needs to be determined which way of binning (over the global range or over the range for each weight) is valid.

7.3 Calculation of mutual information for different parts of the dataset

In the this experiment we show the influence of calculating mutual information over different parts of the dataset. Mutual information can be calculated either over the training, the testing or the full dataset. Moreover, we look at the influcence of varying the activation between `tanh` and `ReLU` under these different settings.

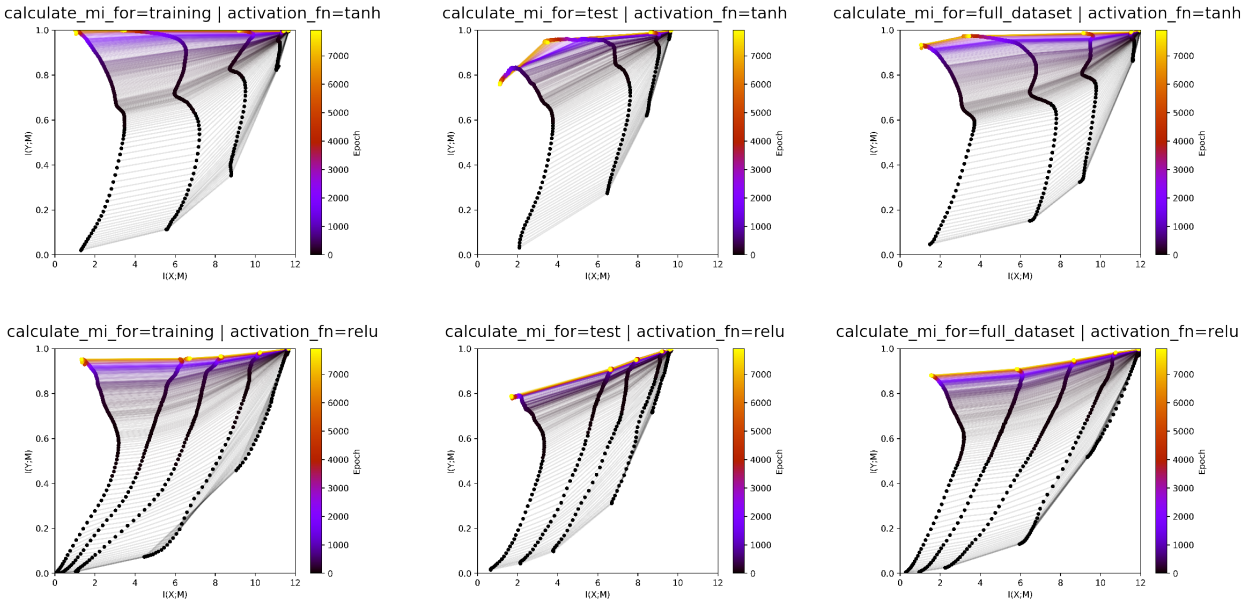
These experiments belong to cohort 4.

```
In [1]: import sys
        sys.path.append('../..')
        from deep_bottleneck.eval_tools.experiment_loader import ExperimentLoader
        from deep_bottleneck.eval_tools.utils import format_config, find_differing_config_keys
        import matplotlib.pyplot as plt
        from io import BytesIO
```

```
In [2]: loader = ExperimentLoader()
```

We look at the different infoplane plots.

```
In [3]: fig, ax = plt.subplots(2,3, figsize=(40, 20))
        ax = ax.flat

        experiment_ids = [209, 206, 208, 207, 204, 205]
        experiments = loader.find_by_ids(experiment_ids)
        differing_config_keys = find_differing_config_keys(experiments)

        for i, experiment in enumerate(experiments):
            img = plt.imread(BytesIO(experiment.artifacts['infoplane'].content))
            ax[i].axis('off')
            ax[i].imshow(img)
            ax[i].set_title(format_config(experiment.config, *differing_config_keys),
                            fontsize=30)
```

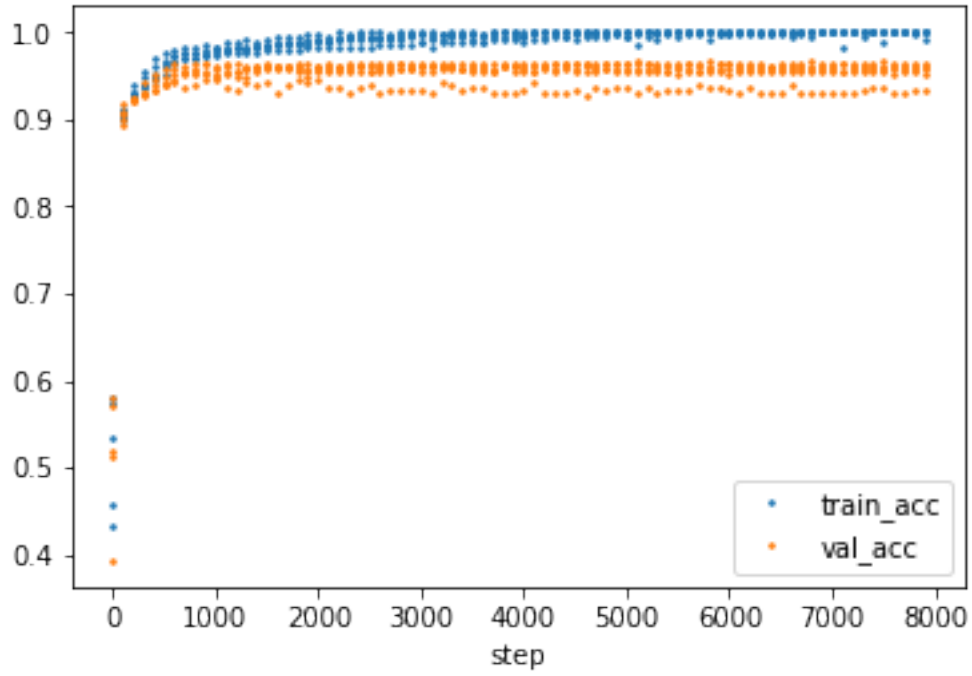


We can see in test data that `tanh` is overfitting at the end. We also see that `ReLU` has lower training than test accuracy as it has less mutual information with the test data than with the train data. These details get lost when estmating mutual information on the full dataset. It is more a smoothed version of both plots, which is less interpretable. Therefore we conclude that it makes most sense to look at the infoplanes for both test and train data.

The infoplane for test data should give more insights into the generalization dynamics. The infoplane on the training data should give insights into the training dynamics.

The overfitting of the tanh can also be seen in the develepment of training and test accuracy.

```
In [4]: import pandas as pd
        import numpy as np

        experiment = loader.find_by_id(206)
        df = pd.DataFrame(data=np.array([experiment.metrics['training.accuracy'].values, experiment.m
                          index=experiment.metrics['validation.accuracy'].index,
                          columns=['train_acc', 'val_acc'])

        df[::100].plot(linestyle='', marker='.', markersize=3)

Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7f60b825eba8>
```



The general configuration for the experiments.

```
In [5]: variable_config_dict = {k: '<var>' for k in differing_config_keys}
        config = experiment.config
        config.update(variable_config_dict)
        config
```

```
Out[5]: {'activation_fn': '<var>',
         'architecture': [10, 7, 5, 4, 3],
         'batch_size': 256,
         'calculate_mi_for': '<var>',
         'callbacks': [],
         'dataset': 'datasets.harmonics',
         'discretization_range': 0.001,
         'epochs': 8000,
         'estimator': 'mi_estimator.upper',
         'learning_rate': 0.0004,
         'model': 'models.feedforward',
         'n_runs': 5,
         'optimizer': 'adam',
         'plotters': [['plotter.informationplane', []],
          ['plotter.snr', []],
          ['plotter.informationplane_movie', []],
          ['plotter.activations', []]],
         'regularization': False,
         'seed': 0}
```

7.4 Experiment Evaluation of Activation Functions

In order to replicate and validate the experiment of Tishby, we tried different activation functions. We decided to test the same activation functions used by the opposing paper "On the Information Bottleneck Theory" by Saxe, Bansal, Dapello, Advani, Kolchinsky, Tracey and Cox. These included ReLU, tanh, Softplus and Softsign. Moreover, we

added SELU, Leaky ReLU, ReLU6, ELU, Sigmoid, Hard-Sigmoid and a simple linear activation function.

7.4.1 1. The Hypothesis

The opposing paper states that "the information plane trajectory is predominantly a function of the neural nonlinearity employed: **double-sided saturating nonlinearities** like tanh yield a **compression phase** as neural activations enter the saturation regime, but **linear activation functions and single-sided saturating nonlinearities** like the widely used ReLU in fact **do no not**" (Saxe et al., 2018, p.1).

Keep in mind that we have alreday seen before that this assumption holds true in our minimal modal analysis in a numeric simulation.

7.4.2 2. Experimental Setting

In the following experiment we tested the eleven activation functions mentioned above with the following **parameter settings**:

- Dataset: Harmonics
- Architecture: [10, 7, 5, 4, 3]
- Batchsize: 256
- Calculate MI for: full dataset
- Discretization range: 0.001
- Epochs: 8000
- Estimator: mi_upper
- Learning rate: 0.0004
- Model: models.feedforward
- n_runs: 5
- Optimizer: Adam
- Regularization: False
- Seed: 0

Note that we sticked to the standard architecture used in Tishby's experiments, used the full dataset and the mutual information upper estimator. The reasons for using these settings can be found in previous experiments (see cohort_1 to cohort_5).

7.4.3 3. Results

Import ArtifactLoader and instantiate it.

```
In [1]: import sys
        sys.path.append('../..')
        from iclr_wrap_up.artifact_viewer import ArtifactLoader
        import matplotlib.pyplot as plt
        from io import BytesIO

In [2]: loader = ArtifactLoader()
```

We varied the 11 activation functions. The experiments are named e.g. **cohort_5_activationfn_tanh**. The experiment ID's are the following:

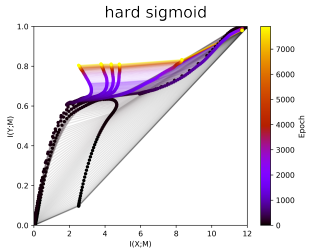
- 210 (hard sigmoid, double-saturated), - 211 (softplus, single-saturated), - 212 (tanh, double-saturated), - 213 (selu, single-saturated), - 214 (sigmoid, double-saturated), - 215 (relu6, double-saturated), - 216 (elu, single-saturated), - 217 (softsign, double-saturated), - 218 (leaky relu, single-saturated), - 219 (relu, single-saturated), - 220 (linear, single-saturated).

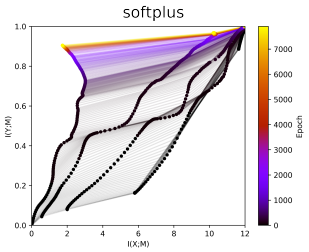
3.1 Infoplane Plots

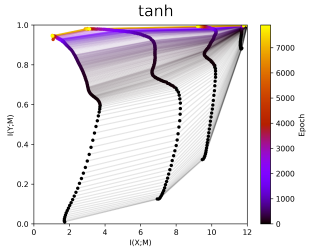
Taking a look at the infoplane plots, one can see that the assumption that only double-saturated activation functions have a compression phase does **not** hold true. With 98.78% accuracy ELU works best, whereas the linear function produces the worst results with an accuracy of 88.4%.

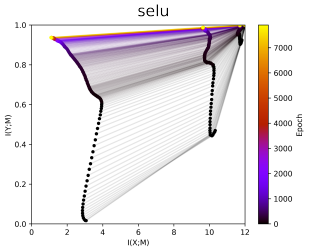
Only the double-saturated **tanh** activation function clearly shows a compression phase - especially in layer 3 and 4.

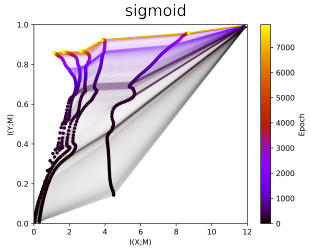
```
In [4]: fig, ax = plt.subplots(6, 2, figsize=(200, 200))
        ax = ax.flat
        activationfunctions = ['hard sigmoid','softplus', 'tanh', 'selu', 'sigmoid', 'relu6',
                               'elu', 'softsign', 'leaky relu', 'relu', 'linear']

        for i,n in enumerate([210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220]):
            artifacts = loader.load(experiment_id=n)
            byte = artifacts['infoplane'].content
            img = plt.imread(BytesIO(byte))
            ax[i].axis('off')
            ax[i].imshow(img)
            ax[i].set_title(activationfunctions[i], fontsize=150)

        plt.tight_layout()
```

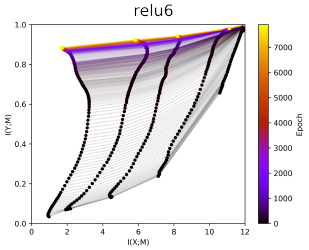


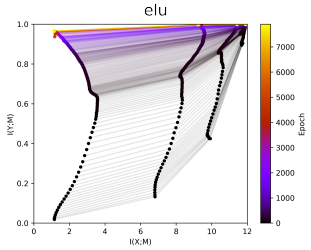


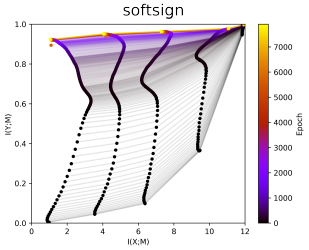


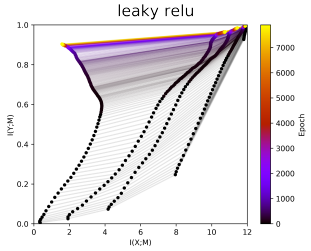


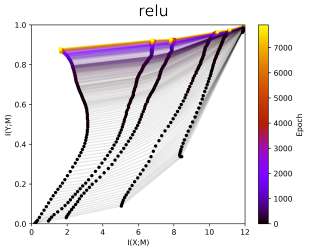


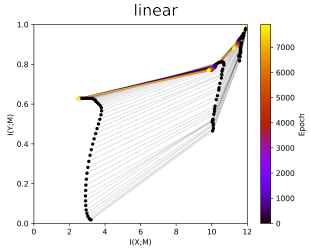












3.2 Compression of Each Layer for ReLU and tanh

In order to get a better understanding how compression happens, we plot the mutual information of the input per layer over the epochs for our n=5 runs. A decreasing graph therefore indicates compression.

Note that the last layer - here layer 5 - is the softmax layer.

Layer 1: no compression, neither for ReLU, nor for tanh is happening. The mutual information of the input stays the same at around 12.

Layer 2: One can clearly see that ReLU4 initially starts below all other activation functions (at around 8 after 8000 epochs. The mutual information of tanh is generally higher or equal than ReLU. No compression is visible.

Layer 3: It displays the same situation as in layer 2. It becomes more obvious that the mutual information of tanh is above ReLU.

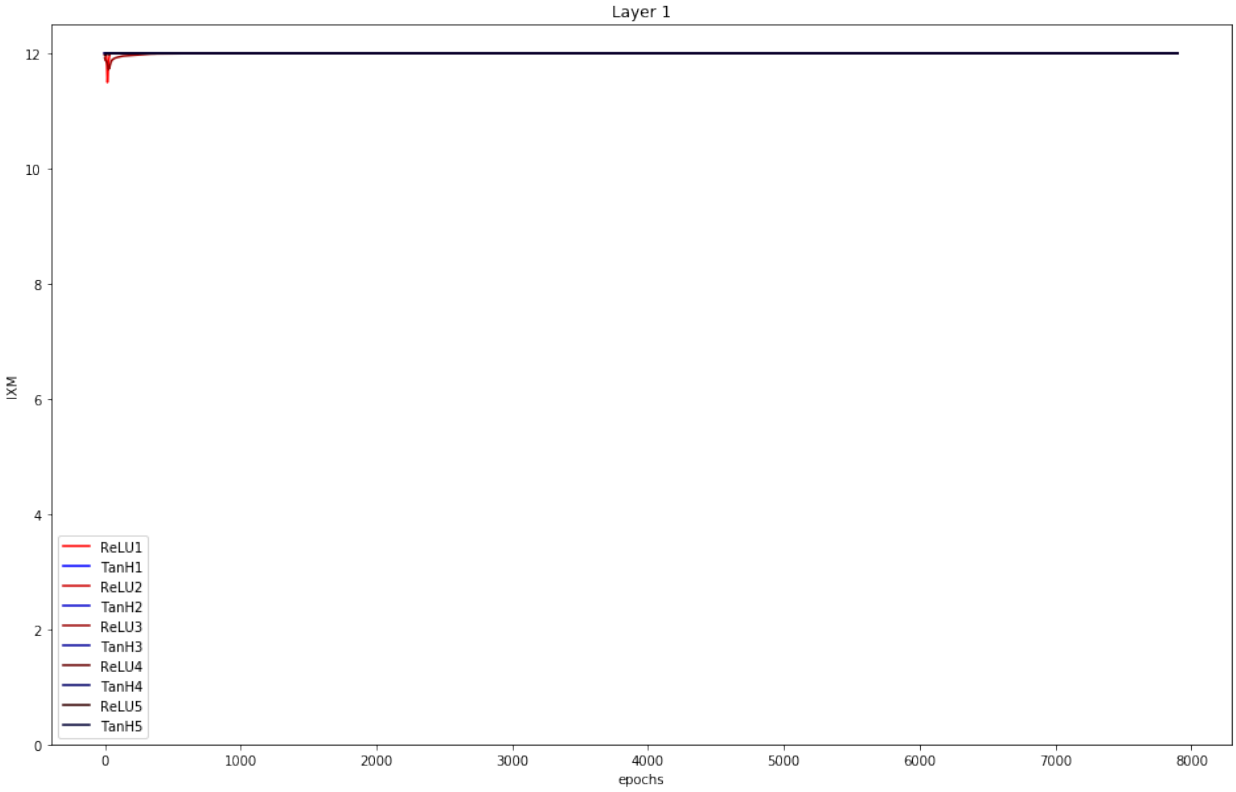
Layer 4: Same situation as in layer 3 but ReLU4 shows a little bit of compression.

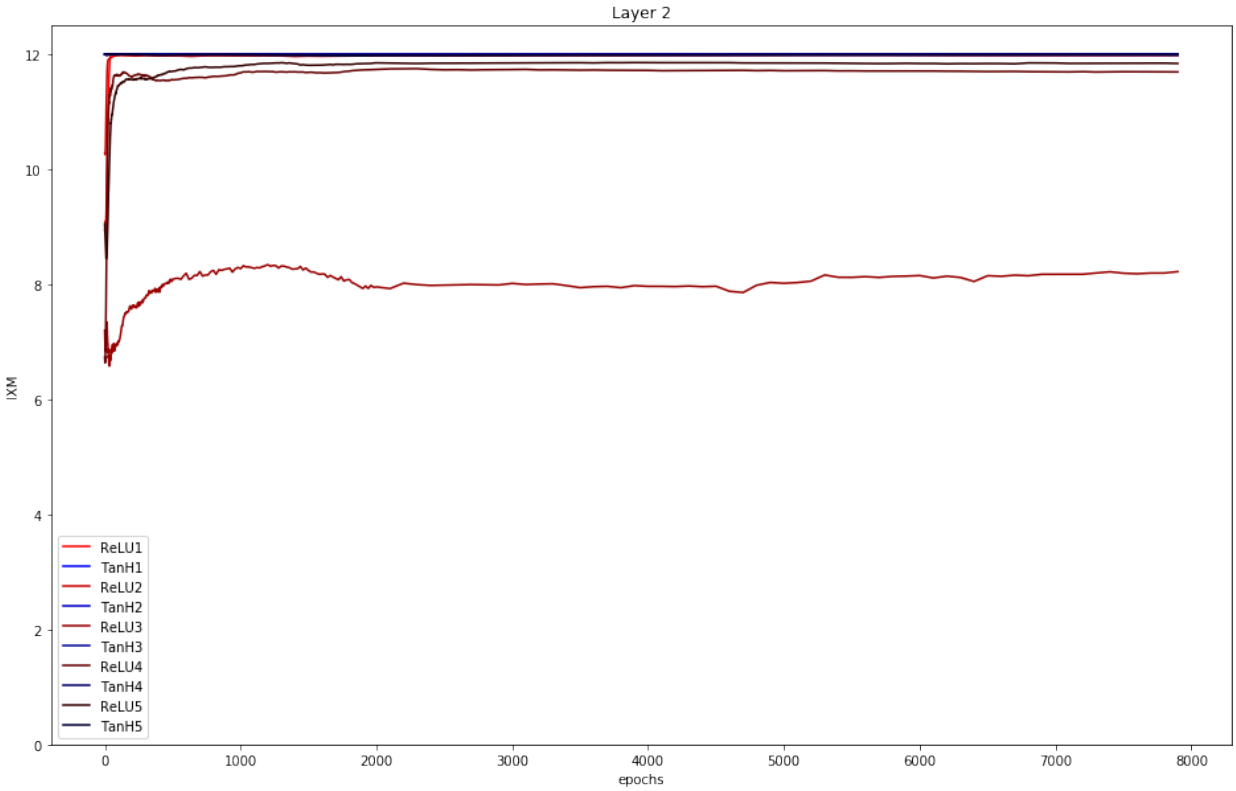
Layer 5: This layer clearly shows compression for all 5 runs of tanh. Moreover, ReLU has generally a higher mutual information than tanh.

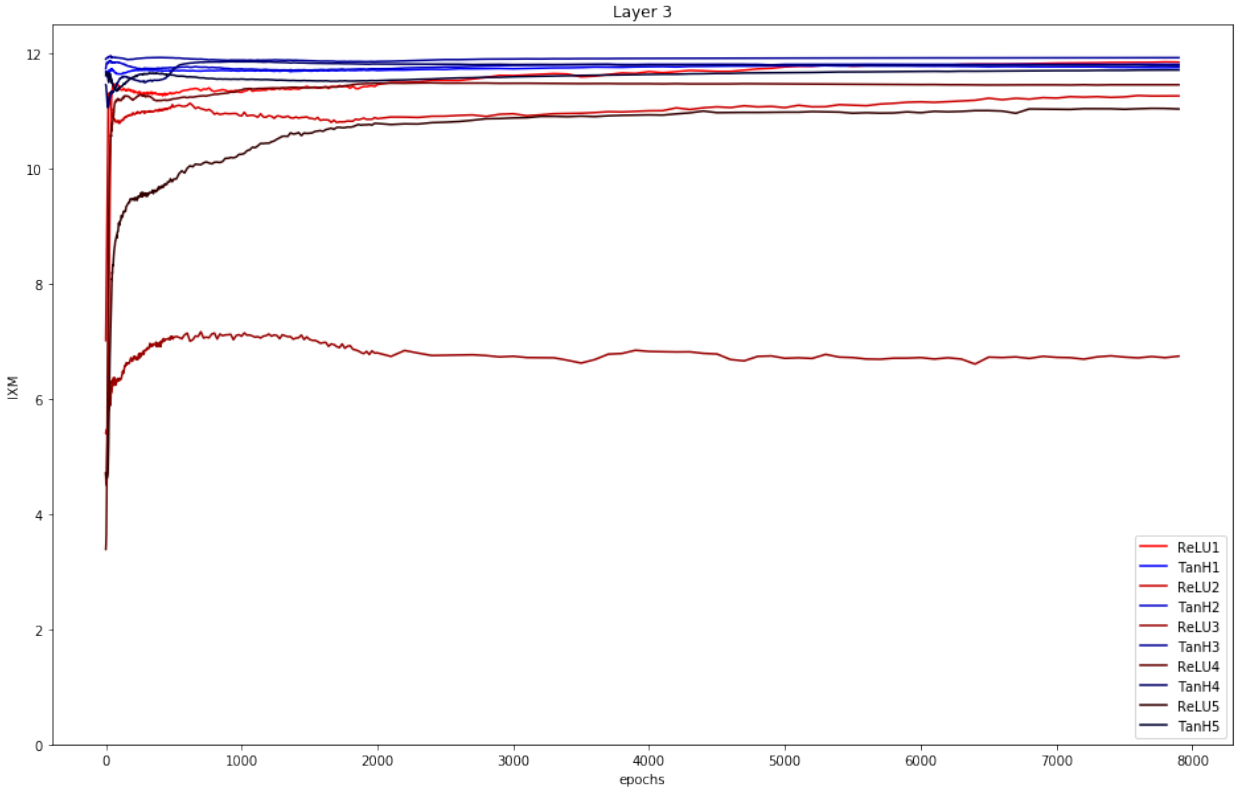
Layer 6: Here the softmax function is applied, i.e. the outputs are in the range (0, 1], and all the entries add up to 1. The mutual information of tanh stays underneath ReLU and therefore performs better.

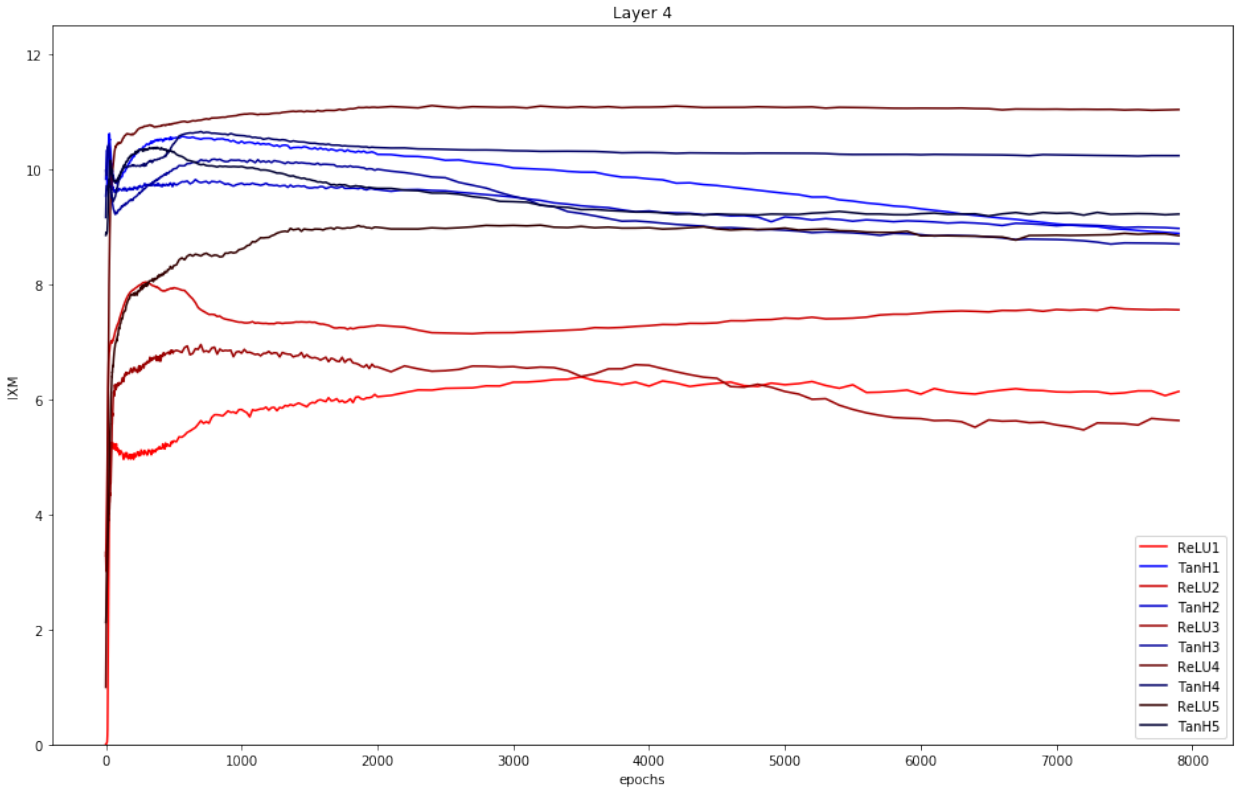
To conclude: Only tanh really shows compression (decreasing graph) in layer 4.

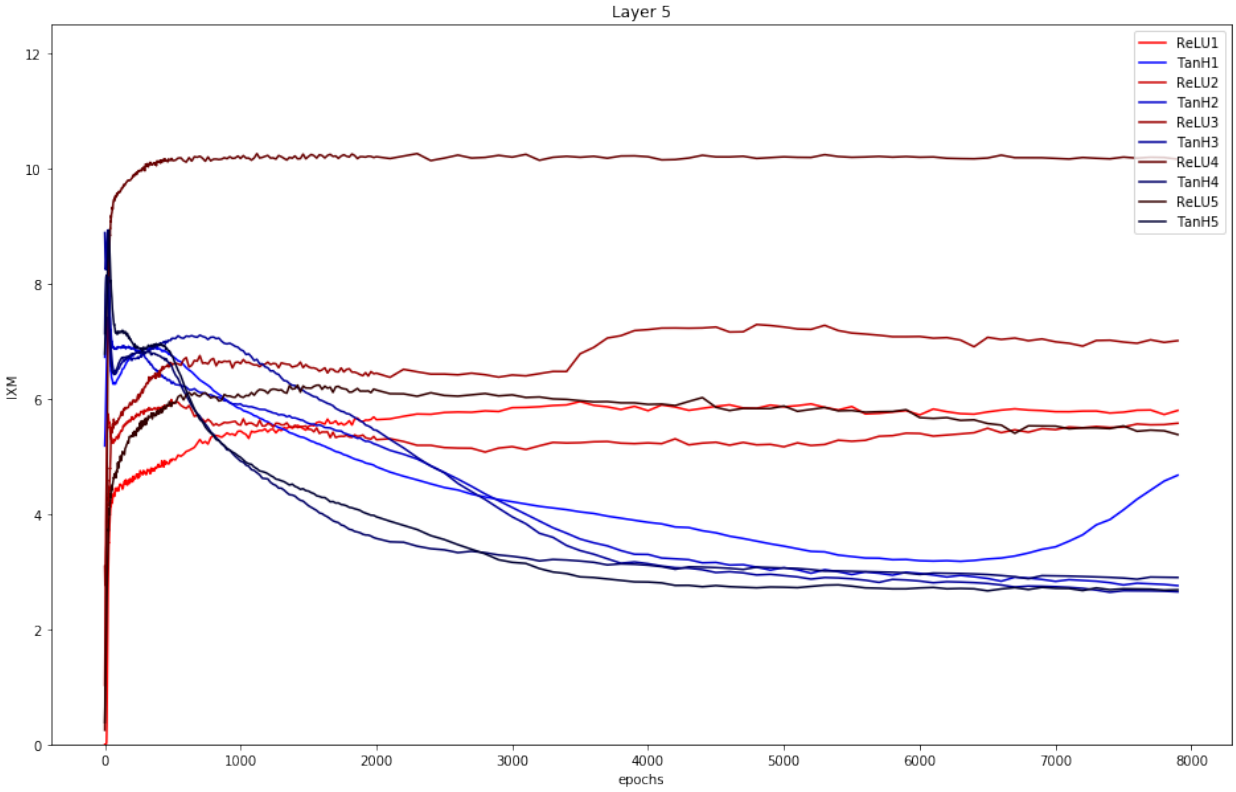
```
In [10]: artifacts_relu = loader.load(experiment_id=219)
         relucsv = artifacts_relu['information_measures'].show()
         artifacts_tanh = loader.load(experiment_id=212)
         tanhcsv = artifacts_tanh['information_measures'].show()

         for layer in range(0,6):
             plt.figure(figsize=(16,10))
             for run in range(0,5):
                 epochs = relucsv[(relucsv['run'] == run) & (relucsv['layer'] == layer)]['epoch'].val
                 mixm_relu = relucsv[(relucsv['run'] == run) & (relucsv['layer'] == layer)]['MI_XM']
                 mixm_tanh = tanhcsv[(tanhcsv['run'] == run) & (tanhcsv['layer'] == layer)]['MI_XM']
                 cr = [1-0.2*run,0,0]
                 cb = [0,0,1-0.2*run]
                 plt.plot(epochs, mixm_relu, color=cr,label="ReLU{}".format(run+1))
                 plt.plot(epochs, mixm_tanh, color=cb, label="TanH{}".format(run+1))
                 plt.ylim([0,12.5])
                 plt.xlabel("epochs")
                 plt.ylabel("IXM")
                 plt.title("Layer {}".format(layer+1))
                 plt.legend()
             plt.show()
```

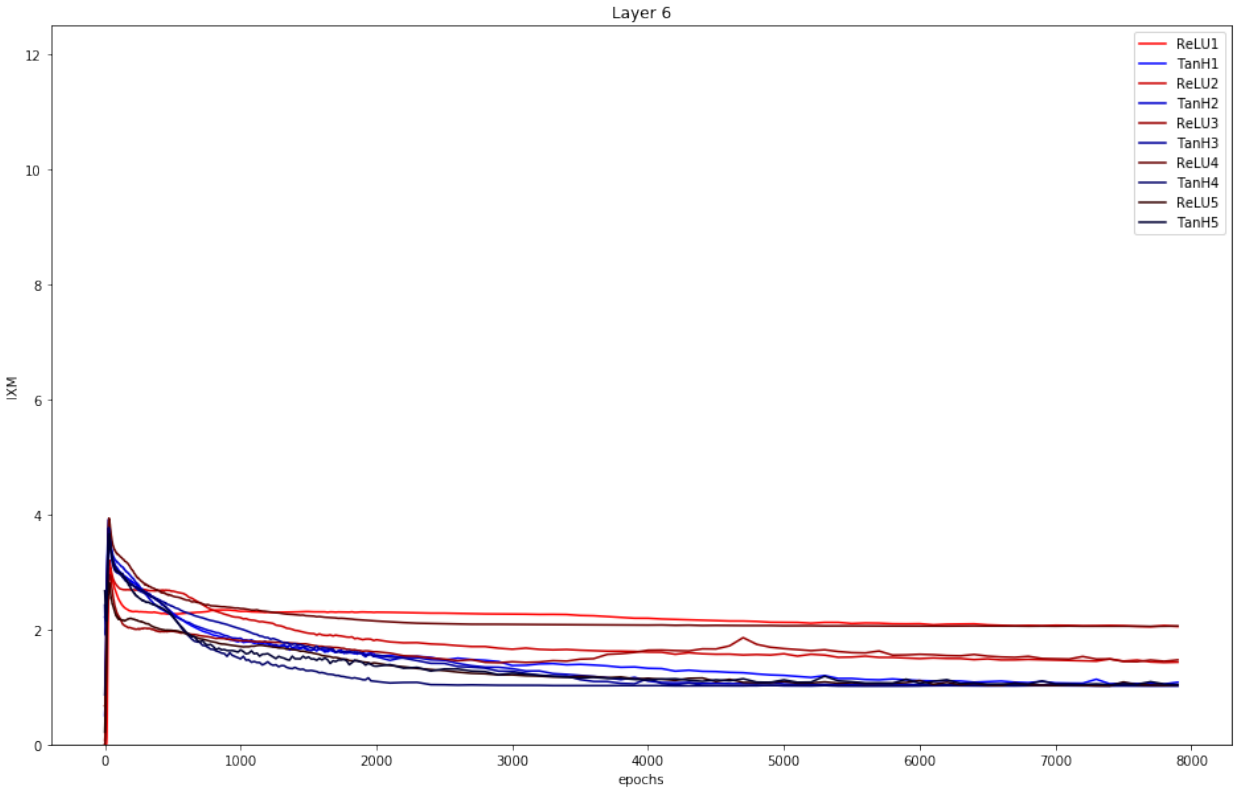

Layer 1
IXM
epochs
ReLU1
TanH1
ReLU2
TanH2
ReLU3
TanH3
ReLU4
TanH4
ReLU5
TanH5


Layer 2
IXM
epochs
ReLU1
TanH1
ReLU2
TanH2
ReLU3
TanH3
ReLU4
TanH4
ReLU5
TanH5


Layer 3
I(X;M)
epochs
ReLU1
TanH1
ReLU2
TanH2
ReLU3
TanH3
ReLU4
TanH4
ReLU5
TanH5


Layer 4
I(X;M)
epochs
ReLU1
TanH1
ReLU2
TanH2
ReLU3
TanH3
ReLU4
TanH4
ReLU5
TanH5





Summary: only tanh shows compression with the given parameter setting. This does neither support the hypothesis of the opposing paper that only double-saturated activation functions show compression, nor supports Tishby.

7.5 Standard vs. Weighted Binning

```
In [1]: import numpy as np
        from scipy.stats import entropy
        from scipy.stats import ortho_group
        import matplotlib.pyplot as plt
        np.random.seed(0)
        import tensorflow as tf
        import tensorflow.contrib.eager as tfe
        tf.enable_eager_execution()
```

7.5.1 In this notebook we try to show that the normal binning approach is not useful and try to derive a new one.

As before we simplify the calculation of the mutual information between the input and a representation, by just calculating the entropy of the representation (as the representation is determined by the input).

We use a very simplistic neural network model of 3 input, 3 hidden and 3 output neurons. The first weights matrix is an orthogonal matrix, such that the transposed matrix (after scaling) is the inverse matrix. We use linear activation function.

```
In [62]: w1 = (1/3) * ortho_group.rvs(dim=3)
         w2 = 9 * w1.T
         print(w1@w2)
```

```
[[ 1.00000000e+00 -2.23155048e-17 -2.80566172e-18]
 [-1.74057426e-17  1.00000000e+00 -6.79414319e-17]
 [ 2.19160108e-19 -1.25252028e-16  1.00000000e+00]]
```

```
In [174]: def act_fn(x):
              return x
```

The datapoints are randomly sampled from a normal distribution.

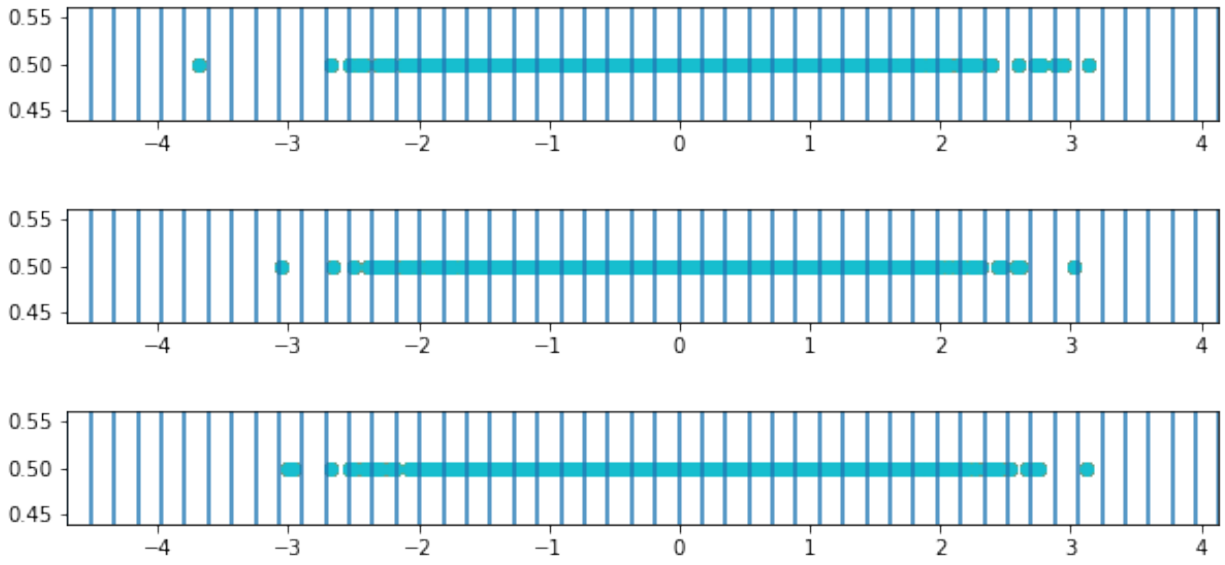
```
In [189]: N = 1000 # number of datapoints
          data = np.random.randn(3,N)
          y = np.ones(N)*0.5
```

The bins are created with linspace between the minimum of all datavalues minus 1 and the maximum of all datavalues + 1.

```
In [190]: n = 50 # number of bins
          a = np.min(data)-1
          b = np.max(data)+1
          bins = np.tile(np.linspace(a,b,n),(3,1))
```

Here you can see the data points and the bins.

```
In [191]: for i in range(bins.shape[0]):
              fig, ax = plt.subplots(1,1, figsize=(10,1), sharex=True, )
              for border in bins[i,:]:
                  ax.axvline(border)
                  ax.set_xlim([a,b])
                  ax.scatter(data[i],y)
```



This function computes the entropy for certain bins. In the case that some data points will land completely outside of the bins I add one bin from the lower border to -infinity and one from the upper border to +infinity.

```
In [192]: def compute_entropy(data, bins):
              digitized = []
              bins_c = np.sort(bins)
              bins_c = np.hstack([np.reshape(np.ones(3)*-np.inf,(3,1)),bins_c])
              bins_c = np.hstack([bins_c,np.reshape(np.ones(3)*np.inf,(3,1))])
              for i in range(data.shape[0]):
                  dig = np.digitize(data[i,:],bins_c[i,:])
                  digitized.append(dig)
              digitized = np.array(digitized)
              uniques, unique_counts = np.unique(digitized, return_counts=True, axis=1)
              return entropy(unique_counts, base=2)
```

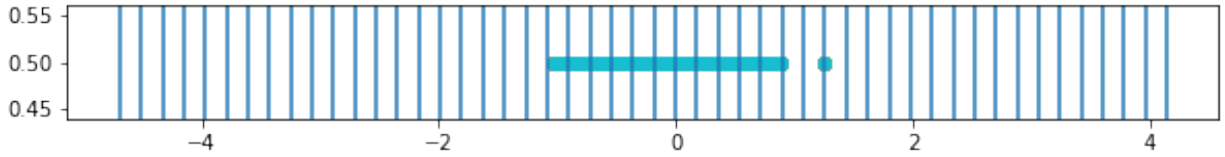
```
In [193]: print("The entropy of the dataset is : {}".format(compute_entropy(data,bins)))
```

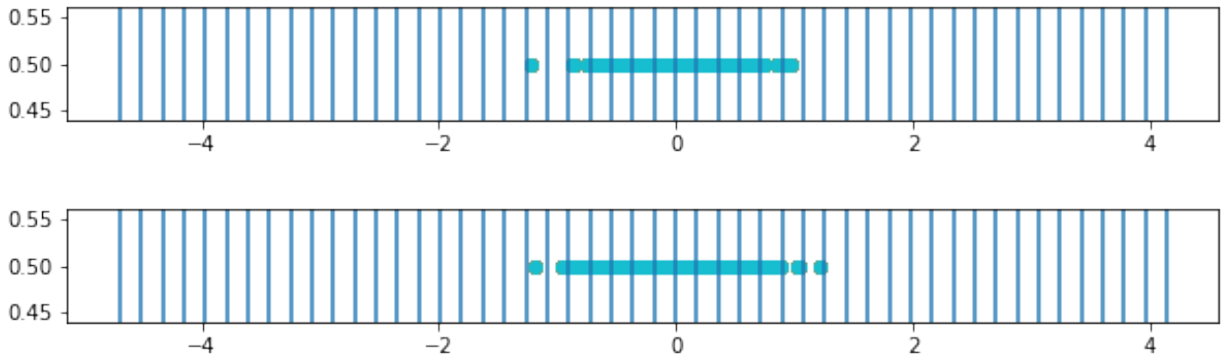
```
The entropy of the dataset is : 9.845254959649102
```

```
In [194]: o1 = act_fn(w1 @ data)
```

```
          for i in range(bins_ours_1.shape[0]):
              fig, ax = plt.subplots(1,1, figsize=(10,1), sharex=True, )
              for border in bins[i,:]:
                  ax.axvline(border)
                  ax.scatter(o1[i,:],y)

          print("Entropy with standard binning of the hidden layer is: {}".format(compute_entropy(o1,
```

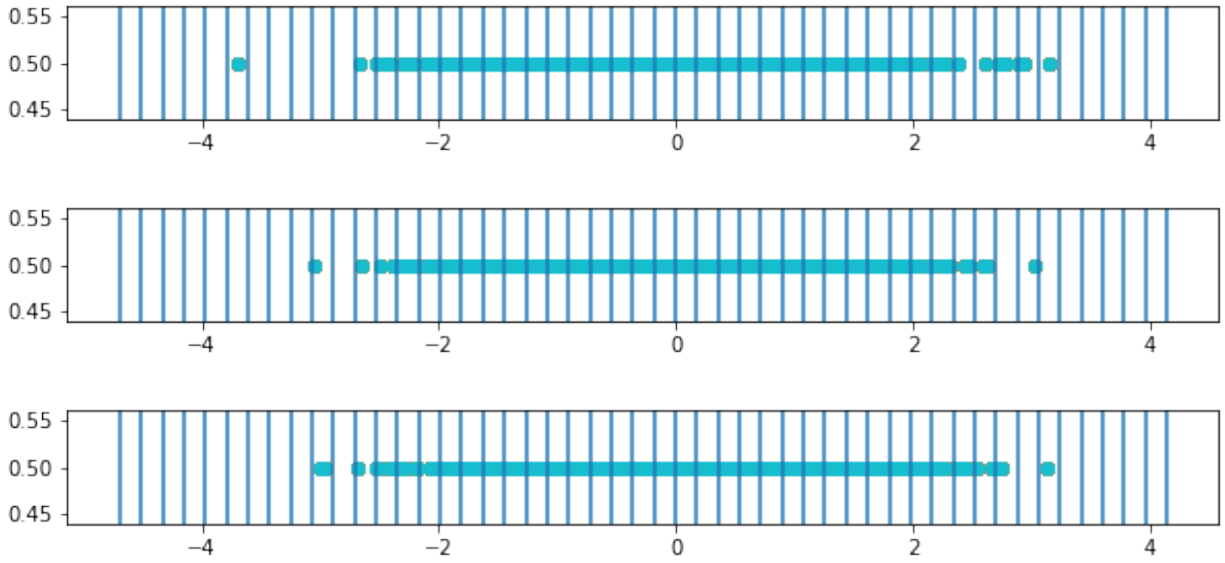
```
Entropy with standard binning of the hidden layer is: 8.03183943622959
```





In [195]:
```
o2 = act_fn(w2 @ o1)

        for i in range(bins_ours_1.shape[0]):
            fig, ax = plt.subplots(1,1, figsize=(10,1), sharex=True, )
            for border in bins[i,:]:
                ax.axvline(border)
                ax.scatter(o2[i,:],y)

        print("Entropy with standard binning of the output layer is: {}".format(compute_entropy(o2,
```

```
Entropy with standard binning of the output layer is: 9.845254959649102
```



We can see that the data processing inequality does not hold with this kind of binning. But what is wrong with it?

I the example above we constructed a network that does not lose information (transformation with invertable matrix and back). The normal binning approach cannot capture this though, because it bins always on the same scale. The scale obviously does not matter to the network as it just can rescale the values in the next layer.

This shows us that the information that is stored in a representation can not be calculated independently but is heavily dependent on what the layer afterwards can get out of it.

So let's think about what a good binning in layer K, with regard to the binning in layer K+1, should look like. First activations that are in the same bin in layer K should not lead to activations which are binned differently in layer K+1. Second activations that are in different bins in layer K should only in few cases lead to the same bins in layer K+1. Meaning this should not happen randomly, but only when the network "wants it on purpose".

Now assume that we could find a meaningfull binning in the last layer which actually describes the few different representations that the network encodes in this layer. Then we could from there compute a "fitting" binning in the layer before and so on. T This meaningful binning cannot be found, so we have to choose one and then propagate this through the network. In this way we make only one "false" choice for binning but do it then the same in every layer. To make it even easier we can do the binning on the input already and from there propagate it forward.

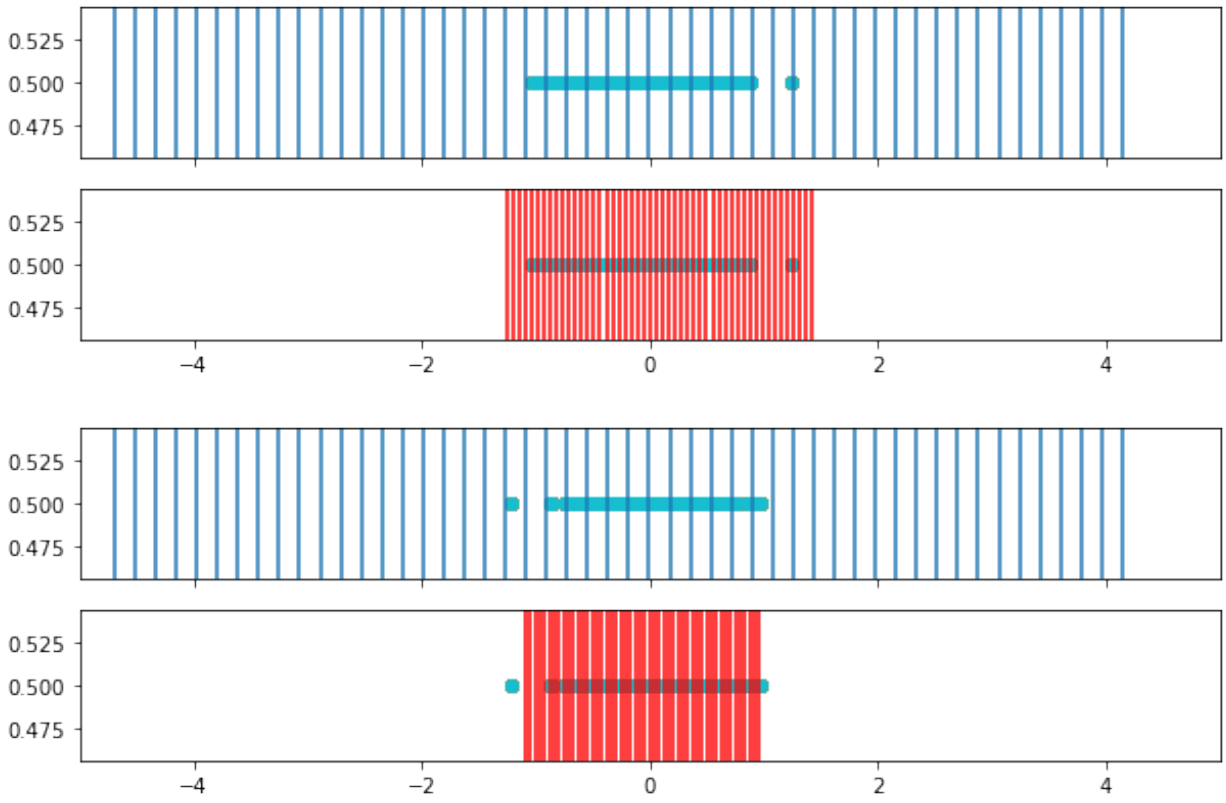
As this binning is dependent on the weights we will call it "weighted binning". So lets see how this works out.

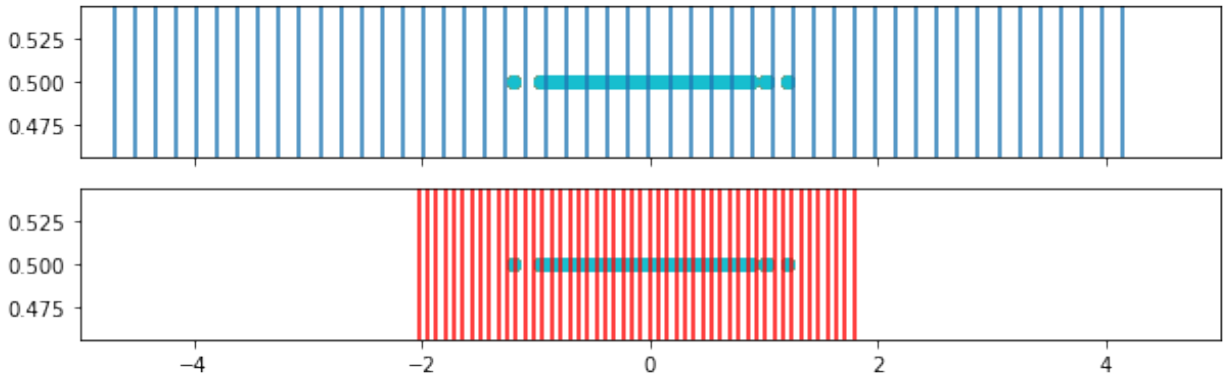
```
In [196]: # First forwardstep of data.
          o1 = act_fn(w1 @ data)
          # First forwardstep of bins. Gotta sort it as the bins might be mirrored.
          #weighted_bins_1 = np.sort(act_fn( w1 @ bins ))
          weighted_bins_1 = (act_fn( w1 @ bins ))

          print("Entropy with weighted binning of hidden layer: {}".format(compute_entropy(o1, weight
```

```
Entropy with weighted binning of hidden layer: 9.854274509657758
```

In this plot we see the weighted bins (red) in comparison to the original bins (blue).

```
In [197]: for i in range(weighted_bins_1.shape[0]):
              fig, ax = plt.subplots(2,1, figsize=(10,3), sharex=True, )
              for border in bins[i,:]:
                  ax[0].axvline(border)
                  ax[0].set_xlim([-5,5])
                  ax[0].scatter(o1[i,:],y)
              for border in weighted_bins_1[i,:]:
                  ax[1].axvline(border, color='r')
                  ax[1].scatter(o1[i,:],y)
```

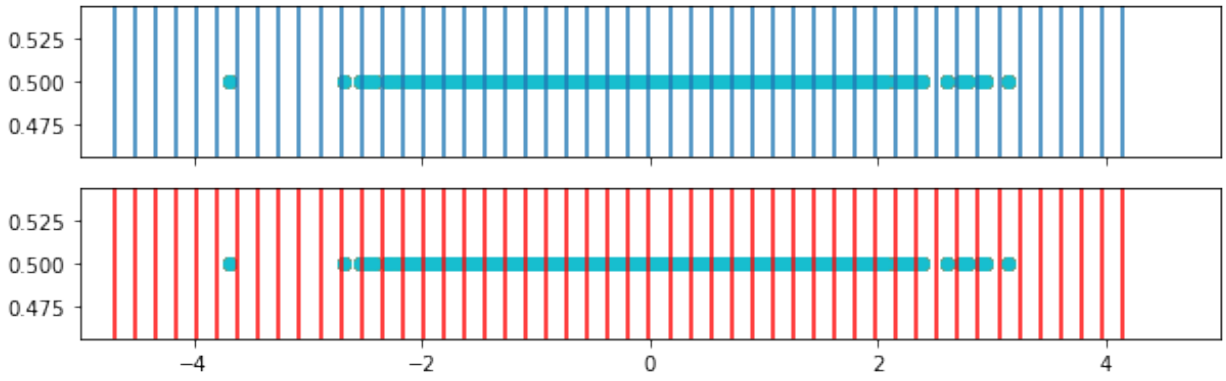


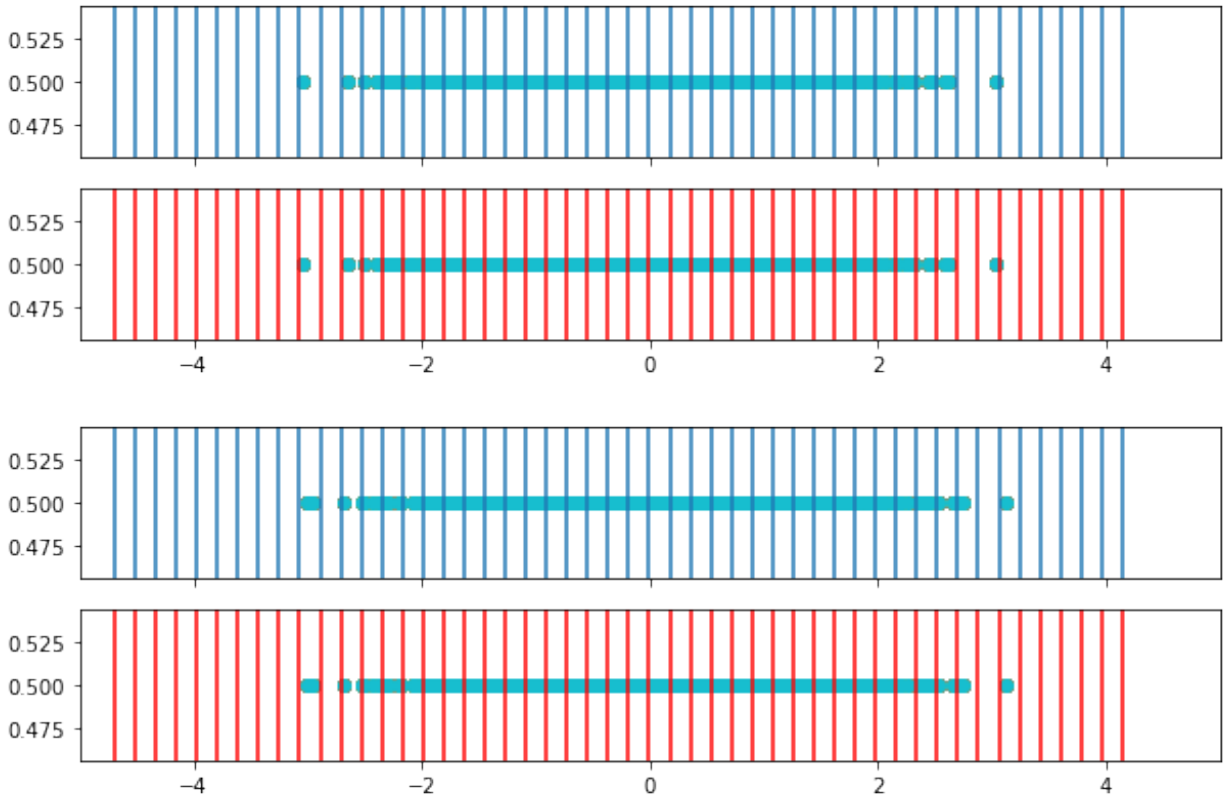



```
In [198]: # Second forwardstep of data.
          o2 = act_fn(w2 @ o1)
          # Second forwardstep of bins.
          #weighted_bins_2 = np.sort(act_fn( w2 @ weighted_bins_1 ))
          weighted_bins_2 = (act_fn( w2 @ weighted_bins_1 ))

          print("Entropy with weighted binning of output layer: {}".format(compute_entropy(o2, weight
```

```
Entropy with weighted binning of output layer: 9.845254959649102
```

```
In [199]: for i in range(weighted_bins_2.shape[0]):
              fig, ax = plt.subplots(2,1, figsize=(10,3), sharex=True, )
              for border in bins[i,:]:
                  ax[0].axvline(border)
                  ax[0].set_xlim([-5,5])
                  ax[0].scatter(o2[i,:],y)
              for border in weighted_bins_2[i,:]:
                  ax[1].axvline(border, color='r')
                  ax[1].scatter(o2[i,:],y)
```






```
In [200]: print("Entropy with standard binning is:")
          print(compute_entropy(data, bins))
          print(compute_entropy(o1, bins))
          print(compute_entropy(o2, bins))


          print("Entropy with weighted binning is:")
          print(compute_entropy(data, bins))
          print(compute_entropy(o1, weighted_bins_1))
          print(compute_entropy(o2, weighted_bins_2))
```

```
Entropy with standard binning is:
9.845254959649102
8.03183943622959
9.845254959649102
Entropy with weighted binning is:
9.845254959649102
9.854274509657758
9.845254959649102
```

We see that the data processing inequality also does not hold for weighted binning but the error being made is much smaller.

Next it would be interesting to find a way to implement this into the model and see what we find there.

7.6 Effect of weight renormalization on activity patterns

In this experiment we show the influence of weight renormalization on the structure of activations in different layers.

Over the course of training the weights in a neural network usually get larger. For `tanh` activated neurons as an example this means that on average the preactivations will be larger in magnitude and therefore the output of the neurons be close to either -1 or 1.

It was argued in the opposing paper that in general double-sided saturating nonlinearities like `tanh` yield a compression phase as neural activations enter the saturation regime. For `relu`, the process of ever growing weights yields bigger activations over time for the positive side of the activation spectrum. It is argued, that as `relu` is not bounded for positive preactivations, this does not lead to compression in later stages of the training.

7.6.1 Experiments with max_weight_norm=0.8

Here, we are picking up on these ideas by introducing rescaling of the weights after each epoch, such that the norm of every neuron's weight vector does not exceed a specific threshold. We observe activation patterns that appear under these constraint during training and interpret these with respect to the phenomenon of "compression" in the infoplane plot.

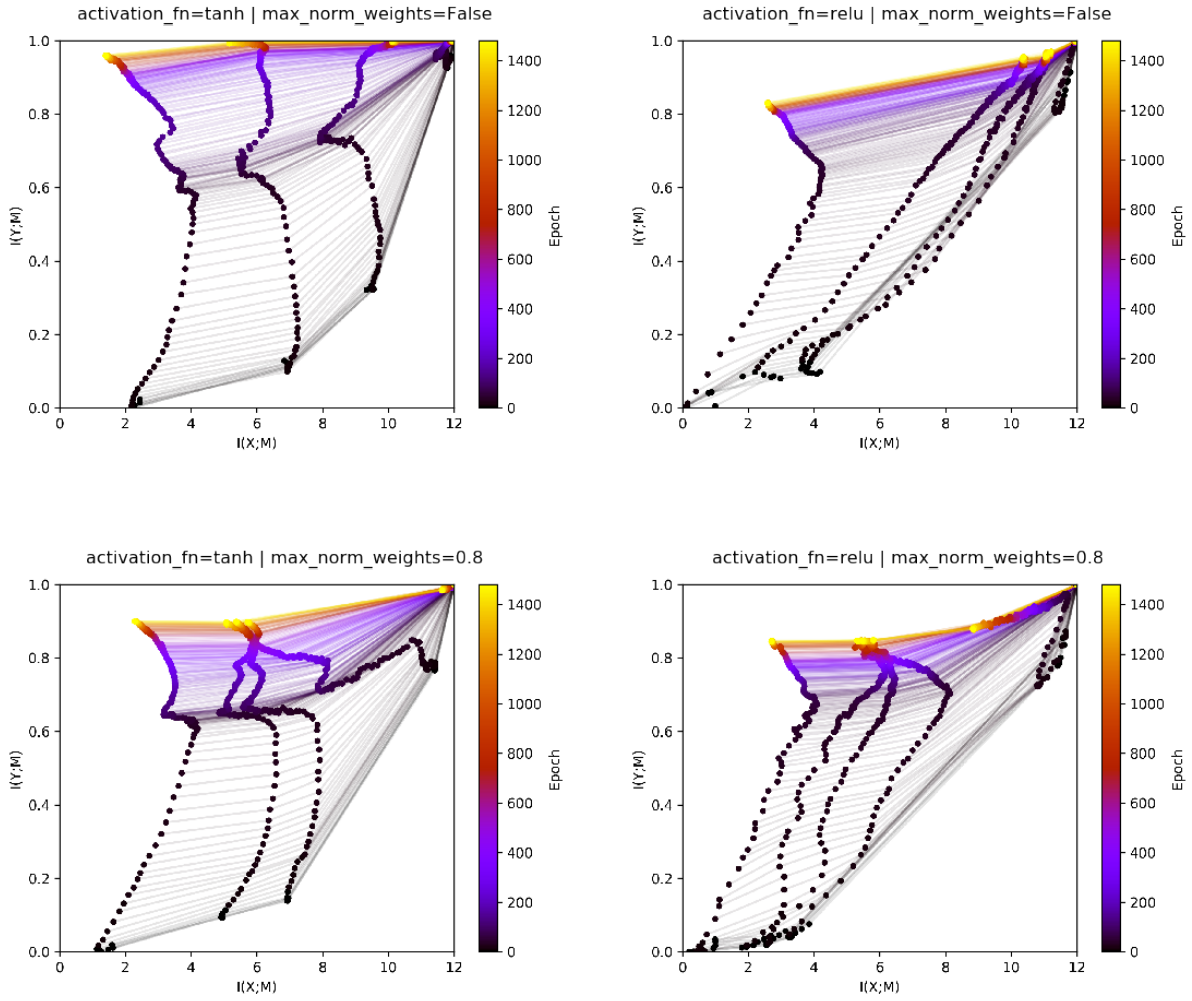
```
In [1]: import sys
        sys.path.append('../..')
        from deep_bottleneck.eval_tools.experiment_loader import ExperimentLoader
        from deep_bottleneck.eval_tools.utils import format_config, find_differing_config_keys
        import matplotlib.pyplot as plt
        from io import BytesIO

        import pandas as pd
        import numpy as np
```

```
In [2]: loader = ExperimentLoader()
```

```
In [3]: experiment_ids = [599, 600, 601, 602]
        experiments = loader.find_by_ids(experiment_ids)
        differing_config_keys = find_differing_config_keys(experiments)
```

We first look at the informationplane plot for 4 different experiments. We varied the activation function between `tanh` and `relu` as well as fixing the maximum magnitude of the weight vector per neuron to 0.8 (`max_norm_weights=0.8`) or leaving the weight magnitude unconstrained (`max_norm_weight=False`). The corresponding informationplane plots for one run are displayed below.

```
In [4]: fig, ax = plt.subplots(2,2, figsize=(16, 14))
        ax = ax.flat

        for i, experiment in enumerate(experiments):
            img = plt.imread(BytesIO(experiment.artifacts['infoplane'].content))
            ax[i].axis('off')
            ax[i].imshow(img)
            ax[i].set_title(format_config(experiment.config, *differing_config_keys),
                            fontsize=16)
        plt.tight_layout()
        plt.show()
```

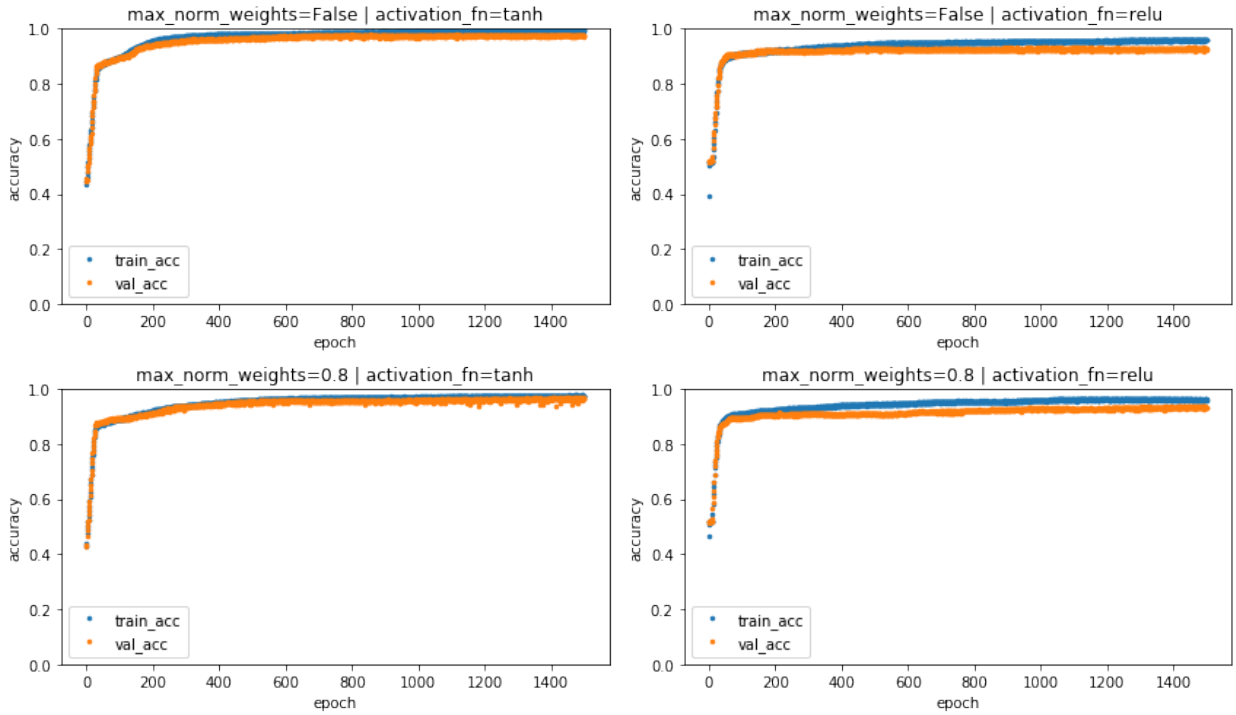


With no weight regularzation, for `relu` we see a pattern in the infoplane as it has been reported before by the opposing paper. Except for the last layer, which always has a `softmax` activation function, no distict compression phase is visible. For `tanh`, the 5th layer starts to compress, while all previous layers do not show distinct compression. In a previously run experiment with several runs it was comfirmed that on average over several runs the earlier layers do show compression as well. Still, we keep in mind that the phenomenon is not as stable as it might be suggested by reports of Tishby et. al. Additionally, the prominent "dip" to the left after 60-100 epochs is missing interpretation by previous works of Tishby and the opposing paper. Also, it seems to be to consistent in some experimental settings as to be only a result of random fluctuations.

With weight normalization, several layers of `tanh` start to compress. Furthermore and noteably, with resticted weight norm, **"relu" compresses.** The layers of both `relu` and `tanh` do not reach information with the output as high as they did without constrained weights. Below we plot training and validation accuracies to confirm that the weight regularization does not come at a cost of a signitficant loss of accuracy.

```
In [9]: fig, ax = plt.subplots(2,2, figsize=(12, 7))
        ax = ax.flat

        for i, experiment in enumerate(experiments):
            df = pd.DataFrame(data=np.array([experiment.metrics['training.accuracy'].values, experime
                              index=experiment.metrics['validation.accuracy'].index,
                              columns=['train_acc', 'val_acc'])

            df.plot(linestyle='', marker='.', markersize=5, ax=ax[i])
```

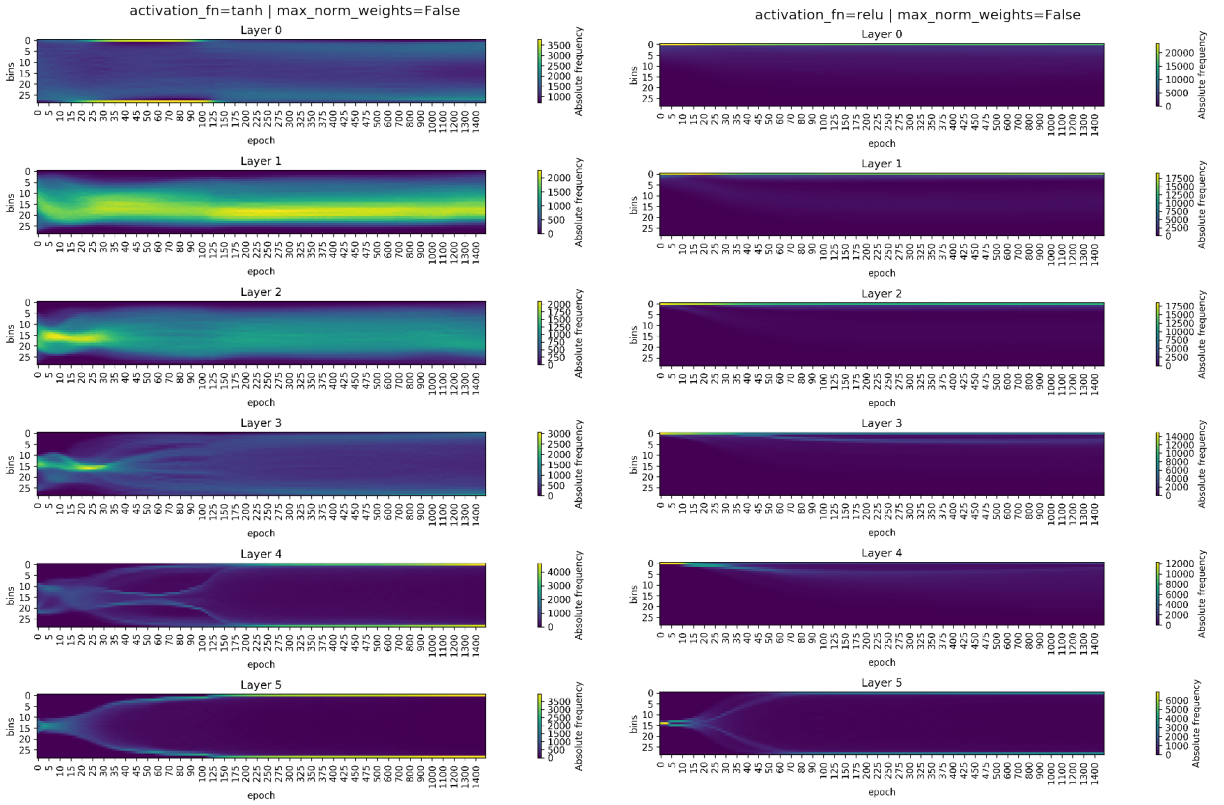
```
        ax[i].set_title(format_config(experiment.config, *differing_config_keys),
                        fontsize=12)
        ax[i].set_ylim([0,1])
        ax[i].set(xlabel='epoch', ylabel='accuracy')

    plt.tight_layout()
    plt.show()
```

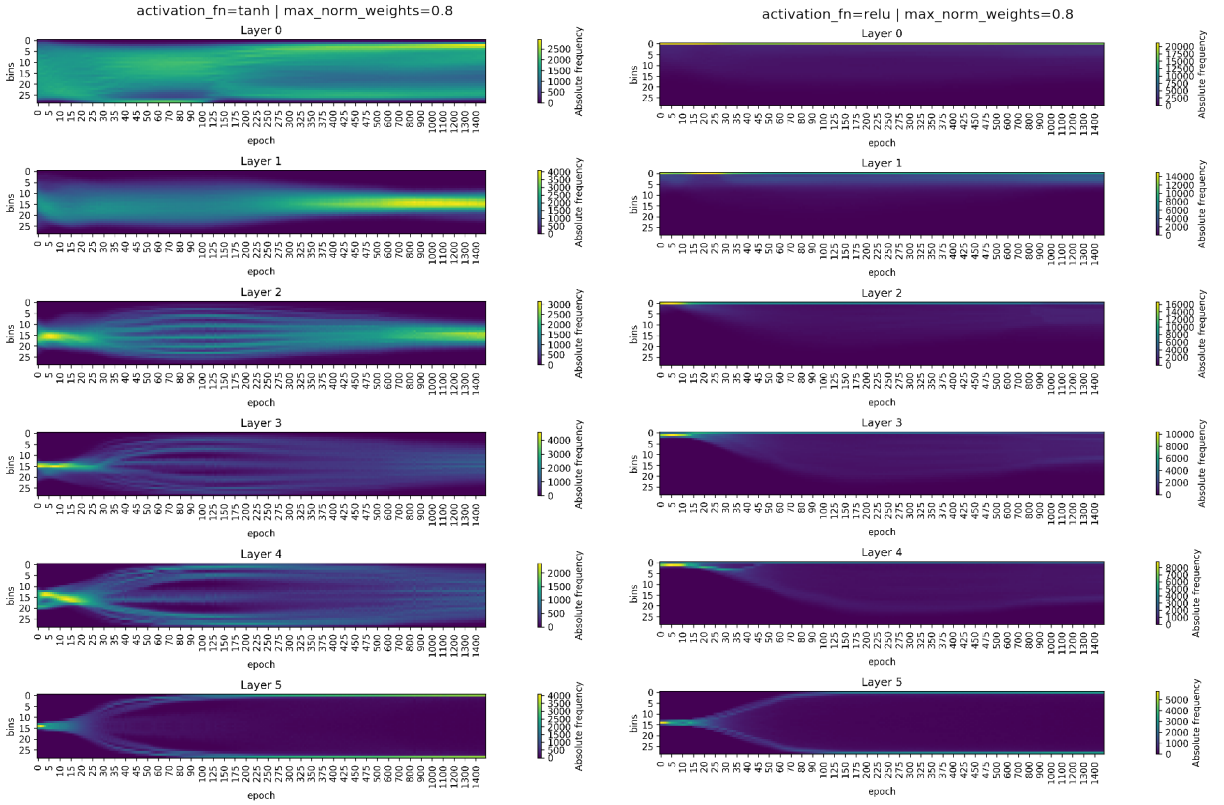


Mutual information with the input for deterministic networks is currently calculated as the entropy of the representation. The representation in this context is the (histogram of the) activation pattern resulting from display of all input samples to the network in a specific epoch. A process towards a lower entropy activity distribution is therefore termed "compression".

We now look at the average activations over epochs for each layer. Each column for the plots is a non-normalized histogram with 30 bins of the activations that were recorded during training and testing of the network.

```
In [6]: fig, ax = plt.subplots(2,2, figsize=(25, 35))
        ax = ax.flat

        for i, experiment in enumerate(experiments):
            img = plt.imread(BytesIO(experiment.artifacts['activations'].content))
            ax[i].axis('off')
            ax[i].imshow(img)
            ax[i].set_title(format_config(experiment.config, *differing_config_keys),
                            fontsize=20)

        plt.tight_layout()
        plt.show()
```


activation_fn=tanh | max_norm_weights=False
activation_fn=relu | max_norm_weights=False
Layer 0
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
bins
epoch
Absolute frequency


activation_fn=tanh | max_norm_weights=0.8
activation_fn=relu | max_norm_weights=0.8
Layer 0
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
bins
epoch
Absolute frequency

In the activation plots for `tanh` we can identify several very prominent peaks of activations, especially during epochs 30-120 for normal `tanh` and 30-250 for weight-restricted `tanh`. These activity patterns conincide with the early phase of compression in the `tanh` informationplane plots. Towards the end of training, higher layers saturate and have peaks at activations of -1 and 1 in the histograms. This phase also displays as compression in the informationplane plot.

With regards to `relu`, the unconstrained network exhibits a peak of activations at 0. In tendency, the remaining nonzero activations grow over the course of training and spread a broader range. This distribution has little structure (except for the prominent peak at 0) and congruent with this also does not show compression in the informationplane. The bias towards the entropy of the prominent peak at 0 due to the `relu` activation function will be discussed in more detail in another notebook.

In the experiment with `relu` and with constrained weight vector, the nonzero activations in the higher layers, especially in layer 3 and 4, show several relatively pronounced peaks. They do not seem to be very prominent, because there is still an big portion of all activations at 0. The pattern of several equidistant peaks of activation is similar to the one observed in weight constrained tanh plots. Again, this is a process towards a lower entropy distribution, which is reflected by the observed "compression" in the information plane.

7.6.2 Experiment with max_weight_norm=0.4

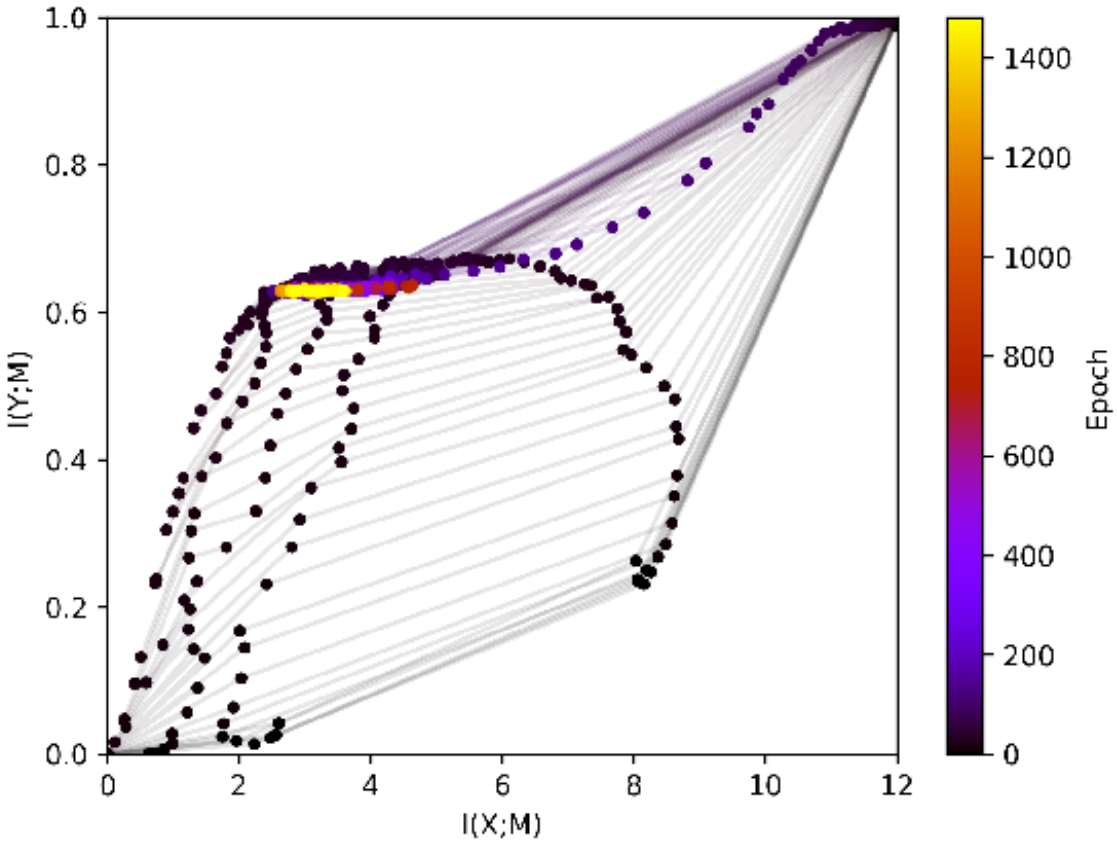
In the following we present an example with `relu` and the norm of the weight vector for each layer restricted to 0.4 This is a significantly stronger regularization which this time will also have an effect on the performance of the network.

```
In [12]: relu04 = loader.find_by_id(603)
         relu04.config
```

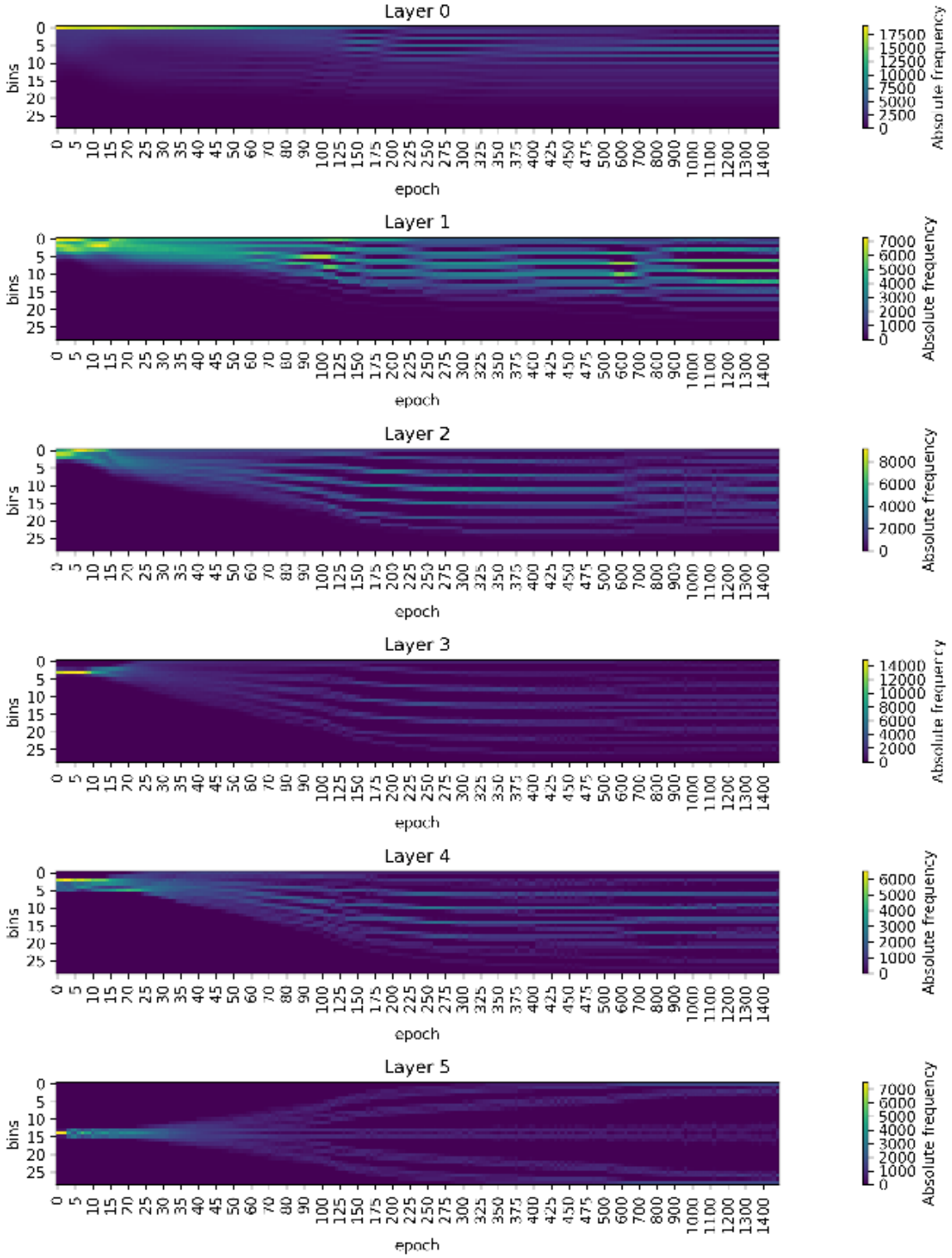
```
Out[12]: {'activation_fn': 'relu',
          'architecture': [10, 7, 5, 4, 3],
          'batch_size': 256,
          'calculate_mi_for': 'full_dataset',
          'callbacks': [],
          'dataset': 'datasets.harmonics',
          'discretization_range': 0.07,
          'epochs': 1500,
          'estimator': 'mi_estimator.binning',
          'learning_rate': 0.0004,
          'max_norm_weights': 0.4,
          'model': 'models.feedforward',
          'n_runs': 1,
          'optimizer': 'adam',
          'plotters': [['plotter.informationplane', []],
           ['plotter.snr', []],
           ['plotter.informationplane_movie', []],
           ['plotter.activations', []],
           ['plotter.activations_single_neuron', []]],
          'seed': 42}
```

In the infoplane plot below it can be seen that training is impaired for the choice of such strict weight regularization.

```
In [8]: relu04.artifacts['infoplane'].show()
```



In [9]: relu04.artifacts['activations'].show(figsize=(12,16))

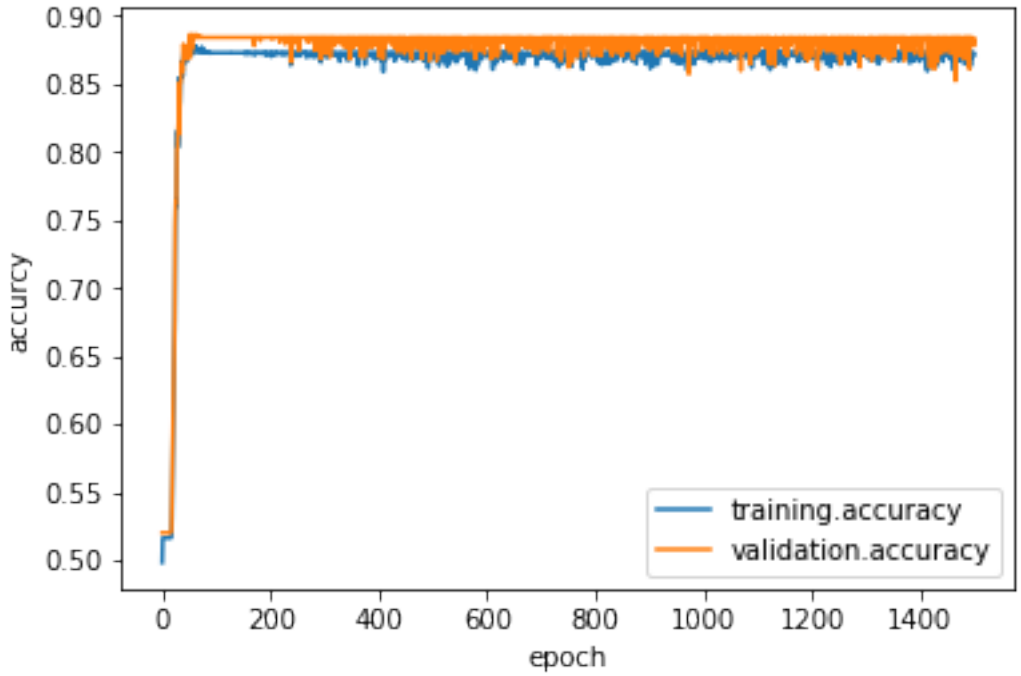


The activation pattern of several peaks is even more pronounced with stronger restiction on the size of the weights.

The performance of the network is worse than with higher weightnorm. But the training dynamics still look ok. The network learns the task up to a certain accurcy without overfitting.

```
In [15]: relu04.metrics['training.accuracy'].plot()
         relu04.metrics['validation.accuracy'].plot()
         plt.ylabel('accurcy')
         plt.xlabel('epoch')
         plt.legend()
```

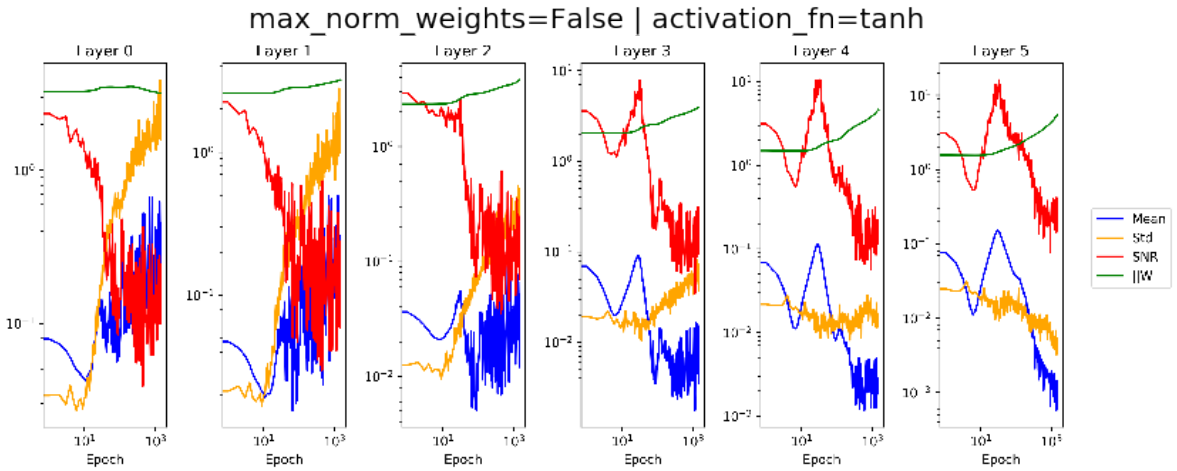
```
Out[15]: <matplotlib.legend.Legend at 0x7f9c33efb978>
```

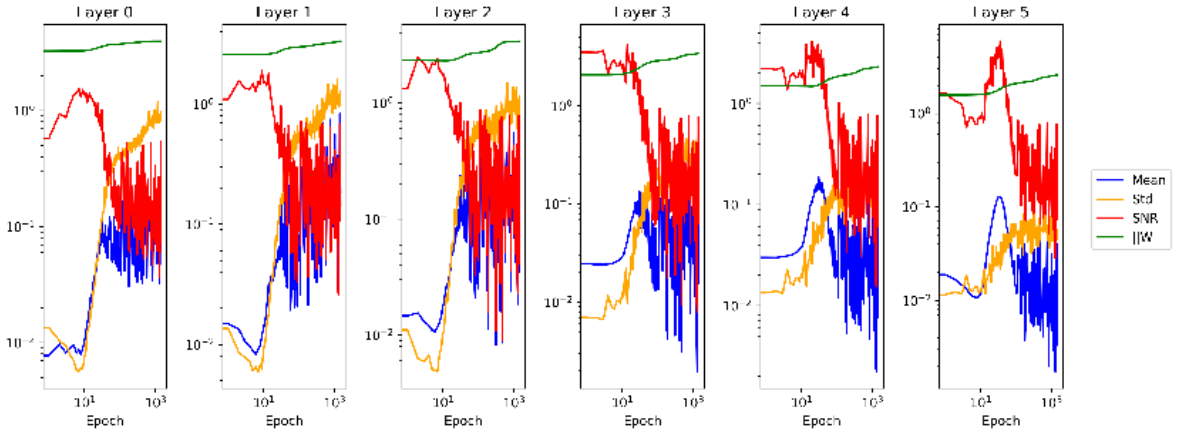


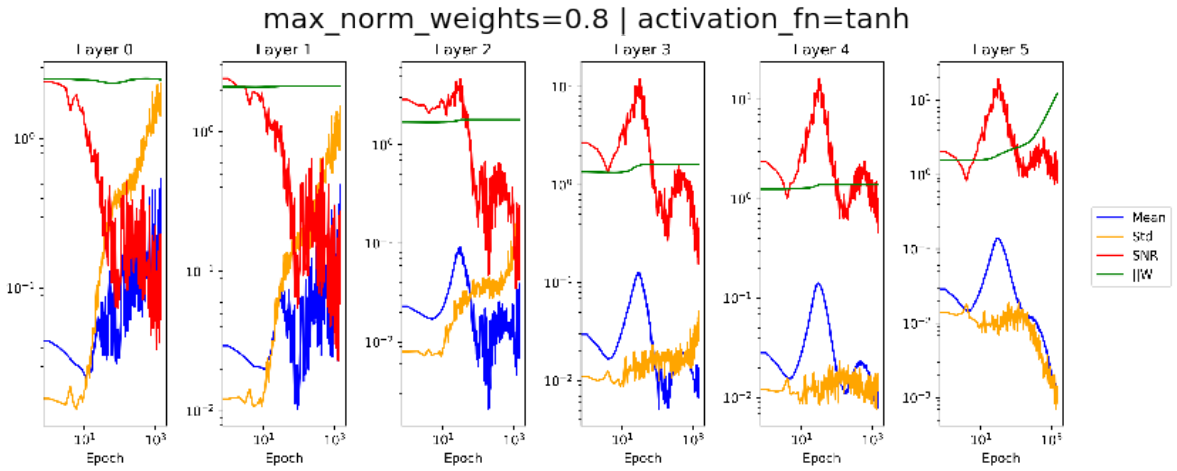
7.6.3 Supplementary material

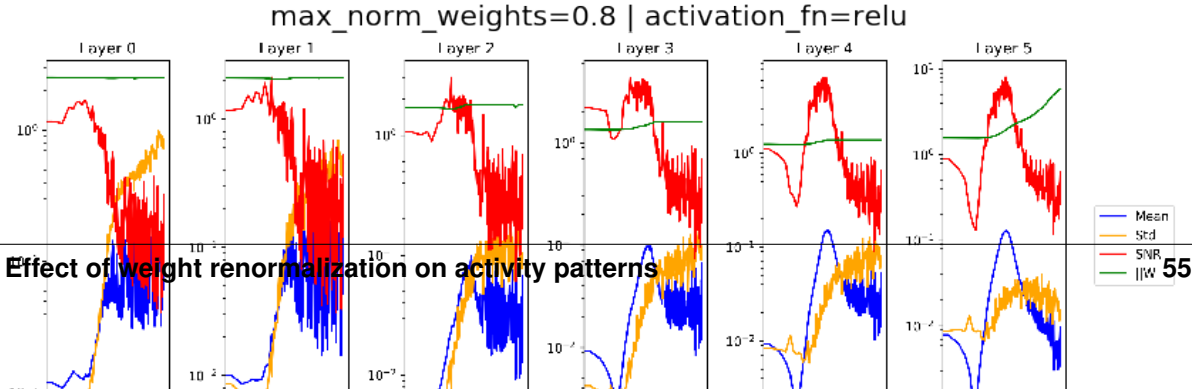
Below we find plots indicating the development of means and standard deviation of the gradient, its signal to noise ratio as well as the norm of the weight vector for all layers over the course of training. Comparing plots for unconstrained vs. constrained weight vector, we can reassure ourselves that rescaling the weights worked as we expected.

```
In [10]: fig, ax = plt.subplots(4,1, figsize=(16, 20))
         ax = ax.flat

         for i, experiment in enumerate(experiments):
             img = plt.imread(BytesIO(experiment.artifacts['snr'].content))
             ax[i].axis('off')
             ax[i].imshow(img)
             ax[i].set_title(format_config(experiment.config, *differing_config_keys),
                             fontsize=20)

         plt.tight_layout()
         plt.show()
```


max_norm_weights=False | activation_fn=tanh
Layer 0
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Epoch
Mean
Std
SNR
||W


max_norm_weights=False | activation_fn=relu
Layer 0
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Epoch
Mean
Std
SNR
||W


max_norm_weights=0.8 | activation_fn=tanh
Layer 0
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Epoch
Mean
Std
SNR
||W


max_norm_weights=0.8 | activation_fn=relu
Layer 0
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Epoch
Mean
Std
SNR
||W

Below we find the configuration of all non-varied parameters that we used for the experiments above.

```
In [10]: variable_config_dict = {k: '<var>' for k in differing_config_keys}
         config = experiment.config
         config.update(variable_config_dict)
         config
```

```
Out[10]: {'activation_fn': '<var>',
          'architecture': [10, 7, 5, 4, 3],
          'batch_size': 256,
          'calculate_mi_for': 'full_dataset',
          'callbacks': [],
          'dataset': 'datasets.harmonics',
          'discretization_range': 0.07,
          'epochs': 1500,
          'estimator': 'mi_estimator.binning',
          'learning_rate': 0.0004,
          'max_norm_weights': '<var>',
          'model': 'models.feedforward',
          'n_runs': 1,
          'optimizer': 'adam',
          'plotters': [['plotter.informationplane', []],
           ['plotter.snr', []],
           ['plotter.informationplane_movie', []],
           ['plotter.activations', []],
           ['plotter.activations_single_neuron', []]],
          'seed': 42}
```

7.7 The data set provided by Tishby

First, we load the data set provided by Tishby.

```
In [1]: import os
        import sys
        nb_dir = os.path.split(os.path.split(os.getcwd())[0])[0]
        sys.path.append(nb_dir)
        from iclr_wrap_up.datasets import harmonics
        import numpy as np

        train_data, test_data = harmonics.load(nb_dir = nb_dir + '/iclr_wrap_up/')
        X = np.concatenate([train_data[0], test_data[0]])
        Y = np.concatenate([train_data[2], test_data[2]])
```

Next, we analyze the loaded data set. In this data set, X corresponds to the 12 binary inputs that represent 12 uniformly distributed points on a 2-sphere. X is realized as one of the 4096 possible combinations of the 12 binary inputs. Y corresponds to $\Theta\left(f\left(X\right)-\theta\right)$, where f is a spherically symmetric real-valued function, θ is a threshold, Θ a step function that outputs either 0 or 1.

We sum over the 12 binary inputs of X and determine the share of $Y=1$ per each possible sum value.

```
In [2]: X_sum_Y = []

        for i in range(13):
            X_sum_Y.append([])

        for i in range(len(X)):
            n_pos_inputs = np.sum(X[i])
            X_sum_Y[n_pos_inputs].append(Y[i])
```

```
        print('Share of Y=1 per each value of the sum of the binary inputs of X:')
        for i in range(13):
            proportion = 100 * np.sum(np.array(X_sum_Y[i]))/len(X_sum_Y[i])
            print(f'{i}: {proportion}%')
```

```
Share of Y=1 per each value of the sum of the binary inputs of X:
0: 0.0%
1: 0.0%
2: 0.0%
3: 0.0%
4: 0.0%
5: 7.575757575757576%
6: 57.57575757575758%
7: 92.42424242424242%
8: 100.0%
9: 100.0%
10: 100.0%
11: 100.0%
12: 100.0%
```

7.8 Our attempt to generate the data set above

After analyzing the provided data set, we try to generate a similar one ourselves since the explicit algorithm for doing so was not provided in the paper.

The first task is to sample 12 uniformly distributed points on a unit 2-sphere. For that purpose, we randomly sample θ from $[0, 2\pi]$ using the uniform distribution. Then, we randomly sample a value for the cosine of ϕ from $[-1, 1]$ again using the uniform distribution. Finally, we apply arccosine to the sampled cosine of ϕ to get ϕ itself. Contrary to the convention, we have assigned θ to the aziumthal angle and ϕ to the polar angle to keep the naming consistent with SciPy's spherical harmonics implementation.

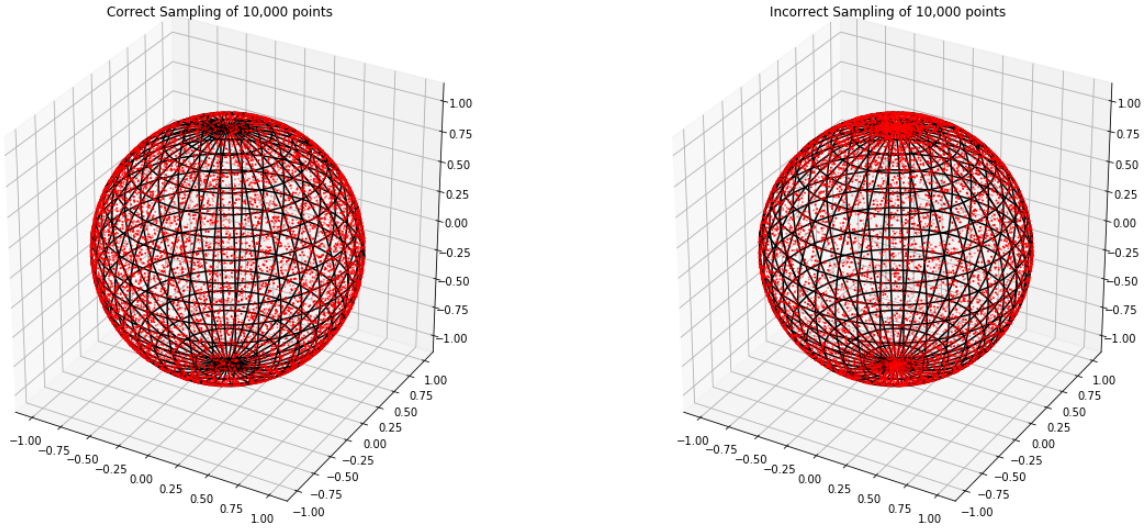
Alternatively, one can randomly sample a 3-dimensional vector from $\mathbb{R}^3$ using standard normal distribution, normalize it, and convert Cartesian coordinates to spherical coordinates.

In [3]:
```
import random
from scipy.special import sph_harm

def sample_spherical(npoints):
    """Sample npoints uniformly distributed
    points on a unit 2-sphere
    """
    thetas = np.zeros(npoints)
    phis = np.zeros(npoints)
    for i in range(npoints):
        thetas[i] = random.uniform(0, 2*np.pi)
        cos_phi = random.uniform(-1, 1)
        phis[i] = np.arccos(cos_phi)
    return thetas, phis

def alternative_sample_spherical(npoints):
    """Sample npoints uniformly distributed
    points on a unit 2-sphere
    """
    vec = np.random.randn(3, npoints)
    vec /= np.linalg.norm(vec, axis=0)
    thetas = np.arctan2(vec[1], vec[0]) + np. pi
    phis = np.arccos(vec[2])
    return thetas, phis
```

```
        def wrong_sample_spherical(npoints):
            """Sample npoints points on a unit 2-sphere
            """
            thetas = np.zeros(npoints)
            phis = np.zeros(npoints)
            for i in range(npoints):
                thetas[i] = random.uniform(0, 2*np.pi)
                phis[i] = random.uniform(0, np.pi)
            return thetas, phis

        np.random.seed(2)
        random.seed(0)
        thetas, phis = sample_spherical(12)
```

Directly sampling ϕ from $[0, \pi]$ using uniform distribution results in concentration of points on the polar caps of the unit sphere (neighborhoods of $z = 1$ and $z = -1$). Note that to express the spherical coordinates $(1, \phi, \theta)$ of a point on a unit sphere in terms of Cartesian coordinates (x, y, z), we use the formulas $x = sin(\phi) \cdot cos(\theta)$, $y = sin(\phi) \cdot sin(\theta)$, $z = cos(\phi)$. Because $cos(\phi)$ is flat near $\phi = 0$ ($z = cos(\phi) \approx 1$) and $\phi = \pi$ ($z = cos(\phi) \approx -1$) and $sin(\phi)$ is almost zero near $\phi = 0$ and $\phi = \pi$ ($x = sin(\phi) \cdot cos(\theta) \approx 0$ and $y = sin(\phi) \cdot sin(\theta) \approx 0$), higher than the expected share (under uniform distribution) of the sampled points will be concentrated on the northern ($(x, y, z) \approx (0, 0, 1)$) and southern ($(x, y, z) \approx (0, 0, -1)$ polar caps of the unit sphere.

```
In [4]: %matplotlib inline
        from matplotlib import pyplot as plt
        from mpl_toolkits.mplot3d import axes3d

        correct_thetas, correct_phis = sample_spherical(10000)
        wrong_thetas, wrong_phis = wrong_sample_spherical(10000)

        phi = np.linspace(0, np.pi, 20)
        theta = np.linspace(0, 2 * np.pi, 40)
        x = np.outer(np.sin(theta), np.cos(phi))
        y = np.outer(np.sin(theta), np.sin(phi))
        z = np.outer(np.cos(theta), np.ones_like(phi))

        correct_x = np.sin(correct_phis) * np.cos(correct_thetas)
        correct_y = np.sin(correct_phis) * np.sin(correct_thetas)
        correct_z = np.cos(correct_phis) * np.ones_like(correct_thetas)

        wrong_x = np.sin(wrong_phis) * np.cos(wrong_thetas)
        wrong_y = np.sin(wrong_phis) * np.sin(wrong_thetas)
        wrong_z = np.cos(wrong_phis) * np.ones_like(wrong_thetas)

        fig, ax = plt.subplots(1, 2, subplot_kw={'projection':'3d', 'aspect':'equal'}, figsize=(20,2
        ax[0].plot_wireframe(x, y, z, color='k', rstride=1, cstride=1)
        ax[0].scatter(correct_x, correct_y, correct_z, s=1, c='r', zorder=1)
        ax[0].set_title('Correct Sampling of 10,000 points')
        ax[1].plot_wireframe(x, y, z, color='k', rstride=1, cstride=1)
        ax[1].scatter(wrong_x, wrong_y, wrong_z, s=1, c='r', zorder=1)
        ax[1].set_title('Incorrect Sampling of 10,000 points')
        plt.show()
```



Next, we have to generate expansion coefficients a_{nm} for the spherical harmonic decomposition of the spherical function that underlies our spherically symmetric real-valued function f.

```
In [5]: def generate_coeffs(l):
            """Generate random expansion coefficients
            for the spherical harmonic decomposition
            up to l-th degree
            """
            coeffs = []
            # 0-th coefficient is 1.
            coeffs.append([1])
            for n in range(1, l+1):
                coeff = []
                for m in np.linspace(-n, n, 2*n + 1, dtype=int):
                    # The rest of the coefficients are randomly
                    # sampled from a standard normal distribution.
                    coeff.append(np.random.randn())
                coeffs.append(coeff)
            return coeffs

        # We generate coefficients up to 85th degree,
        # since starting from 86, certain orders of the
        # degree result in undefined coefficient values.
        a_nm = generate_coeffs(85)
```

Now, we simulate the spherically symmetric real-valued function f using the generated coefficients a_{nm}, the upper limit on the degree of the spherical harmonic decomposition l, and inputs θ and ϕ. For that purpose, we use Kazhdan's rotation invariant descriptors, i.e., the energies of a spherical function g, $SH(g) = \{\|g_0(\theta,\phi)\|, \|g_1(\theta,\phi)\|, ...\}$ where g_n are the frequency components of g:

$$g_n(\theta,\phi) = \pi_n(g) = \sum_{m=n}^{-n} a_{nm} Y_n^m(\theta,\phi).$$

$$In the equation above, : math : `\pi_n` is the projection onto the$$

subspace $V_n = Span\left(Y_n^{-n}, Y_n^{-n+1}, ..., Y_n^{n-1}, Y_n^n\right)$, and Y_n^m is the spherical harmonic of degree $n \geq 0$ and order

$|m| \leq n$. To process the pattern X, we define f_l to be:

$$f_l\left(X = (x_1, x_2, ..., x_{12})\right) = \sum_{n\,=\,0}^{l} \left\| \sum_{i\,=\,1}^{12} x_i \cdot g_n\left(\theta_i, \phi_i\right) \right\|$$
$$where : math : `\forall\, i :\ x_i \in \{0, 1\}`.$$

```
In [6]: def func(a_nm, l, thetas, phis):
            """Apply f_l as defined above to the
            inputs thetas and phis using the
            decomposition coefficients a_nm
            """
            result = 0
            # the first summation
            for n in range(l+1):
                result_n = 0
                # the second summation
                for (theta, phi) in zip(thetas, phis):
                    # the third summation corresponding to the freqency
                    # component of the function
                    for m in np.linspace(-n, n, 2*n + 1, dtype=int):
                        result_n += a_nm[n][m] * sph_harm(m, n, theta, phi)
                # L2 norm
                result += np.linalg.norm(result_n)
            return result
```

Now, we assume $f = f_{85}$ which implies that the expansion coefficients of the 86th degree and on are all zero. To find the appropriate θ value for $\Theta\left(f\left(X\right) - \theta\right)$, we look at our analysis of the provided data above. We see that the sum values between 0 and 4 involving the 12 binary inputs of X correspond to $Y = 0$, and the sum values between 8 and 12 to $Y = 1$. We also see that roughly 57.58% of the data points corresponding to the sum value of 6 result in $Y = 1$. Hence, we assign the 42.42nd percentile (roughly 468.67) of $f_{85}(X_6)$ (X_6 corresponds to the data points with sum value of 6) to θ. We get binary $\hat{Y}$ values through $\hat{Y} = \Theta\left(f_{85}\left(X\right) - 468.67\right)$. In the end, we calculate the share of $\hat{Y} = 1$ per each value of the sum of the binary inputs of X. As you can see below, the resulting numbers roughly mirror the ones we got above using the provided data set.

```
In [7]: X_sum_Y_hat = []

        for i in range(13):
            X_sum_Y_hat.append([])

        for i in range(len(X)):
            n_pos_inputs = np.sum(X[i])
            X_sum_Y_hat[n_pos_inputs].append(func(a_nm, 85, thetas[X[i].astype(bool)], phis[X[i].asty

        threshold = np.percentile(X_sum_Y_hat[6], 42.42)
        print('\u03B8 \u2248 ' + str(np.around(threshold, decimals = 2)))

        print('')

        print('Share of \u0398(f(X)-\u03B8)=1 per each value of the sum of the binary inputs of X:')
        for i in range(13):
            proportion = 100 * np.sum(np.array(X_sum_Y_hat[i]) > threshold)/len(X_sum_Y_hat[i])
            print(f'{i}: {proportion}%')
```

```
  468.67

Share of (f(X)-)=1 per each value of the sum of the binary inputs of X:
0: 0.0%
1: 0.0%
2: 0.0%
```

```
3: 0.0%
4: 0.0%
5: 9.848484848484848%
6: 57.57575757575758%
7: 96.08585858585859%
8: 100.0%
9: 100.0%
10: 100.0%
11: 100.0%
12: 100.0%
```

In the next step, we soften $\hat{Y} = \Theta(f(X) - \theta)$ to $p\left(\hat{Y} = 1 \mid X\right) = \psi(f(X) - \theta)$ where $\psi(u) = \frac{1}{1+e^{-\gamma \cdot u}}$. Here, γ is the sigmoidal gain and it should be high enough to keep the mutual information $I\left(X\,;\,\hat{Y}\right) \approx 0.99$ bits. We set γ to 1. We also assume that the selected $\theta \approx 468.67$ results in $p\left(\hat{Y} = 1\right) = \sum_X p\left(\hat{Y} = 1 \mid X\right) \cdot p(X) \approx 0.5$ with uniform $p(X)$. As you can see below, the values we get for $I\left(X\,;\,\hat{Y}\right)$ and $p\left(\hat{Y} = 1\right)$ do meet the requirements.

```
In [8]: def sigmoidal_func(u, gamma=1):
            """Sigmoidal function with
            the sigmoidal gain gamma
            """
            return 1/(1 + np.exp(-gamma*u))

        # Here we calculate p(Y^hat = 1)
        p_Y_hat = 0
        p_X = 1/4096
        for i in range(13):
            p_Y_hat += np.sum(sigmoidal_func(np.array(X_sum_Y_hat[i]) - threshold))
        p_Y_hat *= p_X
        print('p(\u0176=1) \u2248 ' + str(np.around(p_Y_hat, decimals = 1)))

        # Here we calculate H(Y^hat)
        H_Y_hat = -p_Y_hat*np.log2(p_Y_hat)-(1-p_Y_hat)*np.log2(1-p_Y_hat)
        print('H(\u0176) \u2248 ' + str(np.around(H_Y_hat, decimals = 2)))

        # Here we calculate H(Y^hat|X)
        H_Y_hat_given_X = 0
        p_X = 1/4096
        for i in range(13):
            p_Y_hat_given_X = sigmoidal_func(np.array(X_sum_Y_hat[i]) - threshold)
            H_Y_hat_given_X -= np.sum(p_Y_hat_given_X*np.log2(p_Y_hat_given_X))
        H_Y_hat_given_X *= p_X
        print('H(\u0176|X) \u2248 ' + str(np.around(H_Y_hat_given_X, decimals = 2)))

        # Here we calculate I(X;Y^hat) = H(Y^hat) - H(Y^hat|X)
        I_X_Y_hat = H_Y_hat - H_Y_hat_given_X
        print('I(X;\u0176) = H(\u0176) - H(\u0176|X) \u2248 '
              + str(np.around(H_Y_hat, decimals = 2)) + ' - '
              + str(np.around(H_Y_hat_given_X, decimals = 2)) + ' = '
              + str(np.around(I_X_Y_hat, decimals = 2)))
```

```
p(Ŷ=1)  0.5
H(Ŷ)  1.0
H(Ŷ|X)  0.01
I(X;Ŷ) = H(Ŷ) - H(Ŷ|X)  1.0 - 0.01 = 0.99
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

CHAPTER 9

API Documentation

9.1 deep_bottleneck package

9.1.1 Subpackages

deep_bottleneck.callbacks package

Submodules

deep_bottleneck.callbacks.activityprojector module

class deep_bottleneck.callbacks.activityprojector.**ActivityProjector**(*train*, *test*, *log_dir='./logs'*, *embeddings_freq=10*)

Bases: sphinx.ext.autodoc.importer._MockObject

Read activity from layers of a Keras model and log is for TensorBoard

This callback reads activity from the hidden layers of a Keras model and logs it as Model Checkpoint files. The network activity patterns can then be explored in TensorBoard with its Embeddings Projector

on_epoch_end(*epoch*, *logs=None*)

Write layer activations to file :param epoch: Number of the current epoch :param logs: Quantities such as acc, loss which are passed by Sequential.fit()

Returns None

on_train_end(*logs=None*)

Close files :param logs: Quantities such as acc, loss which are passed by Sequential.fit()

Returns None

set_model(*model*)
Prepare for logging the activities of the layers and set up the TensorBoard projector :param model: The Keras model

Returns None

deep_bottleneck.callbacks.earlystopping_manual module

class deep_bottleneck.callbacks.earlystopping_manual.**EarlyStoppingAtSpecificAccuracy**(*monitor*, *value*)

Bases: sphinx.ext.autodoc.importer._MockObject

classmethod load(*monitor='val_acc'*, *value=0.94*)

on_epoch_end(*epoch*, *logs*)

deep_bottleneck.callbacks.earlystopping_manual.**load**(*monitor='val_acc'*, *value=0.94*)

deep_bottleneck.callbacks.loggingreporter module

class deep_bottleneck.callbacks.loggingreporter.**LoggingReporter**(*trn*, *tst*, *calculate_mi_for*, *batch_size*, *activation_fn*, *file_all_activations*, *do_save_func=None*, **args*, ***kwargs*)

Bases: sphinx.ext.autodoc.importer._MockObject

on_batch_begin(*batch*, *logs={}*)

on_epoch_begin(*epoch*, *logs={}*)

on_epoch_end(*epoch*, *logs={}*)

on_train_begin(*logs={}*)

deep_bottleneck.callbacks.metrics_logger module

class deep_bottleneck.callbacks.metrics_logger.**MetricsLogger**(*run*)

Bases: sphinx.ext.autodoc.importer._MockObject

Callback to log loss and accuracy to sacred database.

on_epoch_end(*epoch*, *logs=None*)

Module contents

deep_bottleneck.datasets package

Submodules

deep_bottleneck.datasets.fashion_mnist module

deep_bottleneck.datasets.fashion_mnist.**load**()
Load the Fashion-MNIST dataset

The output follows the following naming convention:

- X is the data
- y is class, with numbers from 0 to 9
- Y is class, but coded as a 10-dim vector with one entry set to 1 at the column index corresponding to the class

Returns Returns two namedtuples, the first one containing training and the second one containing test data respectively. Both come with fields X, y and Y:

deep_bottleneck.datasets.harmonics module

deep_bottleneck.datasets.harmonics.**import_IB_data_from_mat**(*name_ID*, *nb_dir=''*)
Writes a .npy file to disk containing the harmonics dataset used by Tishby

Parameters **name_ID** – Identifier which is going to be part of the output filename

Returns None

deep_bottleneck.datasets.harmonics.**load**(*nb_dir=''*)
Load the Information Bottleneck harmonics dataset

The output follows the following naming convention:

- X is the data
- y is class, with numbers from 0 to 9
- Y is class, but coded as a 10-dim vector with one entry set to 1 at the column index corresponding to the class

Returns Returns two namedtuples, the first one containing training and the second one containing test data respectively. Both come with fields X, y and Y:

deep_bottleneck.datasets.mnist module

deep_bottleneck.datasets.mnist.**load**()
Load the MNIST handwritten digits dataset

The output follows the following naming convention:

- X is the data
- y is class, with numbers from 0 to 9

- Y is class, but coded as a 10-dim vector with one entry set to 1 at the column index corresponding to the class

Returns Returns two namedtuples, the first one containing training and the second one containing test data respectively. Both come with fields X, y and Y:

deep_bottleneck.datasets.mushroom module

deep_bottleneck.datasets.mushroom.**load**()
Load the mushroom dataset.

Mushrooms are to be classified as either edible or poisonous. The output follows the following naming convention:

- X is the data
- y is class, with numbers from 0 to 9
- Y is class, but coded as a 10-dim vector with one entry set to 1 at the column index corresponding to the class

Returns Returns two namedtuples, the first one containing training and the second one containing test data respectively. Both come with fields X, y and Y:

Module contents

deep_bottleneck.eval_tools package

Submodules

deep_bottleneck.eval_tools.artifact module

class deep_bottleneck.eval_tools.artifact.**Artifact**(*name*, *file*)
Bases: object

Displays or saves an artifact.

content

extension = ''

save()

class deep_bottleneck.eval_tools.artifact.**CSVArtifact**(*name*, *file*)
Bases: *deep_bottleneck.eval_tools.artifact.Artifact*

Displays and saves a CSV artifact

extension = 'csv'

show()

class deep_bottleneck.eval_tools.artifact.**MP4Artifact**(*name*, *file*)
Bases: *deep_bottleneck.eval_tools.artifact.Artifact*

Displays or saves a MP4 artifact

extension = 'mp4'

show()

class deep_bottleneck.eval_tools.artifact.**PNGArtifact**(*name*, *file*)
Bases: *deep_bottleneck.eval_tools.artifact.Artifact*

Displays or saves a PNG artifact.

extension = 'png'

img

show(*figsize=(10, 10)*)

deep_bottleneck.eval_tools.experiment module

class deep_bottleneck.eval_tools.experiment.**Experiment**(*id_*, *database*, *grid_filesystem*, *config*, *artifact_links*, *metric_links*)
Bases: object

artifact_name_to_cls = {'activations': <class 'deep_bottleneck.eval_tools.artifact.PN

artifacts
The artifacts belonging to the experiment.

Returns A mapping from artifact names to artifact objects, that belong to the experiment.

classmethod from_db_object(*database*, *grid_filesystem*, *experiment_data: dict*)

metrics
The metrics belonging to the experiment.

Returns A mapping from metric names to pandas Series objects, that belong to the experiment.

deep_bottleneck.eval_tools.experiment_loader module

class deep_bottleneck.eval_tools.experiment_loader.**ExperimentLoader**(*mongo_uri='mongodb://<MONG*
SHA-1',
db_name='<MONGO_DATABAS
Bases: object

Loads artifacts related to experiments.

find_by_config_key
Find experiments based on regex search against an configuration value.

A partial match between configuration value and regex is enough to find the experiment.

Parameters

- **key** – Configuration key to search on.
- **value** – Regex that is matched against the experiment's configuration.

Returns The matched experiments.

find_by_id
Find experiment based on its id.

Parameters **experiment_id** – The id of the experiment.

Returns The experiment corresponing to the id.

find_by_ids(*experiment_ids: Iterable[int]*) → List[deep_bottleneck.eval_tools.experiment.Experiment]
Find experiments based on a collection of ids.

Parameters **experiment_ids** – Iterable of experiment ids.

Returns The experiments corresponding to the ids.

find_by_name
Find experiments based on regex search against its name.

A partial match between experiment name and regex is enough to find the experiment.

Parameters **name** – Regex that is matched against the experiment name.

Returns The matched experiments.

deep_bottleneck.eval_tools.utils module

deep_bottleneck.eval_tools.utils.**find_differing_config_keys**(*experiments: Iterable[deep_bottleneck.eval_tools.experiment.E*
Find the config keys that were assigned to different values in a cohort of experiments..

deep_bottleneck.eval_tools.utils.**format_config**(*config*, **config_keys*)

Module contents

deep_bottleneck.mi_estimator package

Submodules

deep_bottleneck.mi_estimator.base module

class deep_bottleneck.mi_estimator.base.**MutualInformationEstimator**(*discretization_range*, *training_data*, *test_data*, *architecture*, *calculate_mi_for*)
Bases: object

compute_mi(*file_all_activations*) → pandas.core.frame.DataFrame

nats2bits = 1.4426950408889634
Nats to bits conversion factor.

deep_bottleneck.mi_estimator.binning module

class deep_bottleneck.mi_estimator.binning.**BinningMutualInformationEstimator**(*discretization_range*, *training_data*, *test_data*, *architecture*, *calculate_mi_for*)

Bases: *deep_bottleneck.mi_estimator.base.MutualInformationEstimator*

deep_bottleneck.mi_estimator.binning.**load**(*discretization_range*, *training_data*, *test_data*, *architecture*, *calculate_mi_for*)

deep_bottleneck.mi_estimator.bounded module

class deep_bottleneck.mi_estimator.bounded.**BoundedMutualInformationEstimator**(*discretization_range*, *training_data*, *test_data*, *architecture*, *calculate_mi_for*)

Bases: *deep_bottleneck.mi_estimator.base.MutualInformationEstimator*

deep_bottleneck.mi_estimator.kde module

deep_bottleneck.mi_estimator.kde.**Kget_dists**(*X*)

Keras code to compute the pairwise distance matrix for a set of vectors specifie by the matrix X.

deep_bottleneck.mi_estimator.kde.**entropy_estimator_bd**(*x*, *var*)

deep_bottleneck.mi_estimator.kde.**entropy_estimator_kl**(*x*, *var*)

deep_bottleneck.mi_estimator.kde.**get_shape**(*x*)

deep_bottleneck.mi_estimator.kde.**kde_condentropy**(*output*, *var*)

deep_bottleneck.mi_estimator.lower module

class deep_bottleneck.mi_estimator.lower.**LowerBoundMutualInformationEstimator**(*discretization_ran training_data, test_data, architecture, calculate_mi_for*)

Bases: *deep_bottleneck.mi_estimator.bounded.BoundedMutualInformationEstimator*

deep_bottleneck.mi_estimator.lower.**load**(*discretization_range*, *training_data*, *test_data*, *architecture*, *calculate_mi_for*)

deep_bottleneck.mi_estimator.upper module

class deep_bottleneck.mi_estimator.upper.**UpperBoundMutualInformationEstimator**(*discretization_ran training_data, test_data, architecture, calculate_mi_for*)

Bases: *deep_bottleneck.mi_estimator.bounded.BoundedMutualInformationEstimator*

deep_bottleneck.mi_estimator.upper.**load**(*discretization_range*, *training_data*, *test_data*, *architecture*, *calculate_mi_for*)

Module contents

deep_bottleneck.models package

Submodules

deep_bottleneck.models.feedforward module

deep_bottleneck.models.feedforward.**load**(*architecture*, *activation_fn*, *optimizer*, *learning_rate*, *input_size*, *output_size*, *max_norm_weights=False*)

deep_bottleneck.models.feedforward_batchnorm module

deep_bottleneck.models.feedforward_batchnorm.**load**(*architecture*, *activation_fn*, *optimizer*, *learning_rate*, *input_size*, *output_size*, *max_norm_weights=False*)

Module contents

deep_bottleneck.plotter package

Submodules

deep_bottleneck.plotter.activations module

class deep_bottleneck.plotter.activations.**ActivityPlotter**(*run*, *dataset*)

Bases: *deep_bottleneck.plotter.base.BasePlotter*

plot(*measures_summary*)

plotname = 'activations'

deep_bottleneck.plotter.activations.**load**(*run*, *dataset*)

deep_bottleneck.plotter.activations_single_neuron module

class deep_bottleneck.plotter.activations_single_neuron.**SingleNeuronActivityPlotter**(*run*, *dataset*)

Bases: *deep_bottleneck.plotter.base.BasePlotter*

plot(*measures_summary*)

plotname = 'single_neuron_activations'

deep_bottleneck.plotter.activations_single_neuron.**load**(*run*, *dataset*)

deep_bottleneck.plotter.base module

class deep_bottleneck.plotter.base.**BasePlotter**

Bases: object

Base class for plotters.

generate(*measures_summary*)

plot(*measures_summary*) → matplotlib.figure.Figure

plotname = ''

deep_bottleneck.plotter.informationplane module

class deep_bottleneck.plotter.informationplane.**InformationPlanePlotter**(*run*, *dataset*)

Bases: *deep_bottleneck.plotter.base.BasePlotter*

Plot the infoplane for average MI estimates.

plot(*measures_summary*)

plotname = 'infoplane'

deep_bottleneck.plotter.informationplane.**load**(*run*, *dataset*)

deep_bottleneck.plotter.informationplane_movie module

class deep_bottleneck.plotter.informationplane_movie.**InformationPlaneMoviePlotter**(*run*, *dataset*)

Bases: *deep_bottleneck.plotter.base.BasePlotter*

Plot the infoplane movie for several runs of the same network.

filename = 'plots/infoplane_movie.mp4'

generate(*measures_summary*)

plot(*measures_summary*)

plotname = 'infoplane_movie'

deep_bottleneck.plotter.informationplane_movie.**load**(*run*, *dataset*)

deep_bottleneck.plotter.snr module

class deep_bottleneck.plotter.snr.**SignalToNoiseRatioPlotter**(*run*, *dataset*)

Bases: *deep_bottleneck.plotter.base.BasePlotter*

plot(*measures_summary*)

plotname = 'snr'

deep_bottleneck.plotter.snr.**load**(*run*, *dataset*)

Module contents

9.1.2 Submodules

9.1.3 deep_bottleneck.artifact_viewer module

class deep_bottleneck.artifact_viewer.**Artifact**(*name*, *file*)

Bases: object

Displays or saves an artifact.

content

extension = ''

save()

class deep_bottleneck.artifact_viewer.**ArtifactLoader**(*mongo_uri='mongodb://<MONGO_INITDB_ROOT_U*
SHA-1',
db_name='<MONGO_DATABASE>')

Bases: object

Loads artifacts related to experiments.

load

class deep_bottleneck.artifact_viewer.**CSVArtifact**(*name*, *file*)
Bases: *deep_bottleneck.artifact_viewer.Artifact*

Displays and saves a CSV artifact

extension = 'csv'

show()

class deep_bottleneck.artifact_viewer.**MP4Artifact**(*name*, *file*)
Bases: *deep_bottleneck.artifact_viewer.Artifact*

Displays or saves a MP4 artifact

extension = 'mp4'

show()

class deep_bottleneck.artifact_viewer.**PNGArtifact**(*name*, *file*)
Bases: *deep_bottleneck.artifact_viewer.Artifact*

Displays or saves a PNG artifact.

extension = 'png'

show(*figsize=(10, 10)*)

9.1.4 deep_bottleneck.run_experiments module

This script can be used to submit several experiments to the grid.

All the experiments need to be specified as seperate JSON or yml files.

deep_bottleneck.run_experiments.**create_output_directory**()

deep_bottleneck.run_experiments.**main**()

deep_bottleneck.run_experiments.**parse_command_line_args**()

deep_bottleneck.run_experiments.**start_experiments**(*config_dir*)
Recursively walk through the config dir and submit all experiment configurations in there to the grid.

9.1.5 deep_bottleneck.utils module

deep_bottleneck.utils.**construct_full_dataset**(*training*, *test*)
Concatenates training and test data splits to obtain the full dataset.

The input arguments use the following naming convention:

- X is the training data
- y is training class, with numbers from 0 to 1
- Y is training class, but coded as a 2-dim vector with one entry set to 1 at the column index corresponding to the class

Parameters

- **training** – Namedtuple with fields X, y and Y:
- **test** – Namedtuple with fields X, y and Y:

Returns A new Namedtuple with fields X, y and Y containing the concatenation of training and test data

deep_bottleneck.utils.**data_shuffle**(*data_sets_org*, *percent_of_train*, *min_test_data=80*, *shuffle_data=False*)

Divided the data to train and test and shuffle it

deep_bottleneck.utils.**get_min_max**(*activations_summary*, *layer_number*, *neuron_number=None*)

Get minimum and maximum of activations of a specific layer or a specific neuron over all epochs :param activations_summary: numpy ndarray :param layer_number: Index of the layer :param neuron_number: Index of the neuron. If None, activations of the whole layer serve as a basis

Returns Minimum and maximum value of activations over all epochs

deep_bottleneck.utils.**is_dense_like**(*layer*)

Check whether a layer has attribute 'kernel', which is true for dense-like layers :param layer: Keras layer to check for attribute 'kernel'

Returns True if layer has attribute 'kernel', False otherwise

deep_bottleneck.utils.**shuffle_in_unison_inplace**(*a*, *b*)

Shuffles both array a and b randomly in unison :param a: An Array, for example containing data samples :param b: An Array, fpor example containing labels

Returns Both arrays shuffled in the same way

9.1.6 Module contents

Bibliography

[TZ15] Naftali Tishby and Noga Zaslavsky. Deep Learning and the Information Bottleneck Principle. *Ieee*, pages 1–5, 2015. URL: https://arxiv.org/pdf/1503.02406.pdf, arXiv:1503.02406, doi:10.1109/ITW.2015.7133169.

Python Module Index

d

Index

E

F

G

I

K

L

M

N

O

P

S

U