LRP Toolbox for Artificial Neural Networks 1.2.0 – Manual

Sebastian Lapuschkin Alexander Binder Grégoire Montavon
Klaus-Robert Müller Wojciech Samek

February 20, 2020

Contents

1 Overview

This is the manual for the LRP Toolbox for Artificial Neural Networks, an open-source implementation of the Layer-wise Relevance Propagation (LRP)[1] Algorithm for deep learning architectures. LRP is capable of decomposing the decision function $f(\cdot)$ of a given neural network model wrt to an input point $\boldsymbol{x}$ in order to compute *relevance values* $R_d^{(1)}$ at input level explaining *why* and *how* the model predicts the way it does.

The computed relevance values describe whether the state of input component x_d contributes towards the recognition of the classifier's prediction target if $R_d^{(1)} > 0$ — or opposes it if $R_d^{(1)} < 0$ — and can be visualized as a heatmap for visual assessment. A detailed description of LRP can either in the original publication [1] or further below in Section 3.

The toolbox comes with platform-independent stand-alone implementations of LRP in Matlab and python and a plugin for the Caffe [4] open-source ConvNet implementation. Models, Data and Results can be imported and exported as Matlab's `.mat` files, numpy's `.npy` or `.npz` files for python or ASCII-formatted plain text. For details about the supported formats for each implementation see Section 4.

Both the Matlab and python code are fully documented and for each example impelentations demonstrate the application of LRP at hand of the MNIST hand written digit data set [8]. The LRP implementation for Caffe extends the respective classes and files with the needed functionality and operates directly on images and pre-trained model from the command line. The most recent official release of the source code and example data and models are available from

`http://heatmapping.org`

with work-in-progress code and release candidates being obtainable via

`https://github.com/sebastian-lapuschkin/lrp_toolbox`

The following Section 2 will guide through the process of obtaining, setting up and running the LRP Toolbox for all supported programming languages. Section 3 explains the LRP Algorithm in theory and Section 4 the implementation thereof by giving an overview over the provided classes and functions. Section 6 contains additional information, the answers to frequently asked questions and the authors' contact data.

2 Setup and Execution

Getting the latest toolbox version

The latest official release of the LRP Toolbox is available for download from the official homepage:

`http://heatmapping.org`

For using the **Matlab** implementation, download and merge the following archives:
`lrp_toolbox_matlab.zip` *containing the source code*
`models_mnist_mat.zip` *containing Matlab readable model files*
`data_mnist_mat.zip` *containing the MATLAB readable MNIST data*

For using the **python** implementation, download and merge the following archives:
`lrp_toolbox_python.zip` *containing the source code*
`models_mnist_nn.zip` *containing python-readable model files*
`data_mnist_npy.zip` *containing the numpy readable MNIST data*

There are also plain text versions for the MNIST data and pre-trained neural network models available for download, which can be read by both the python and Matlab implementations.

For using LRP within the **Caffe** Framework download
`lrp_toolbox_caffe.zip` *containing the extended caffe source code,* `.cpp`- and `.hpp`-files *-files implementing tools to compute heatmaps from images using Caffe and some example images.*

More work-in-progress versions of the toolbox code and future release candidates may be obtained via `https://github.com/sebastian-lapuschkin/lrp_toolbox`.

Platforms and Requirements

The python and matlab implementations are available for systems running Linux, Windows and OSX due to the platform agnostic nature of python and Matlab. Caffe runs on Linux and OSX, however Caffe ports for Windows became available very recently on github (e.g. `https://github.com/happynear/caffe-windows`). Known and successfully tested minimum package requirements are:

python	**Matlab**	**Caffe**
python 2.7.5+ numpy 1.7.1+ matplotlib 1.3.0+ skimage 0.8.2+	Matlab 8.3.0.532 (R2014a) optional: Image Processing Toolbox	*Caffe-own requirements:* Boost Libraries (1.55.0) Protocol Buffers: libprotobuf (2.5.0) protobuf-compiler (2.5.0) Google logging: libglogs (0.3.3) Google Flags: libgflags (2.0) HDF5: libhdf5 (1.8.11) Google Snappy: libsnappy (1.1.0) LevelDB: libleveldb (1.15.0) Lightning Mem-mapped DB: liblmdb (0.9.10) ATLAS *OR* OpenBLAS *OR* Intel Math Kernel Library (tested with: libatlas-base-dev 3.10.1) OpenCV$\geq$2.4
		for the LRP heatmap computation binaries: ImageMagick (6.7.7)

where the requirement listing for the LRP implementation on Caffe only lists the additional requirements beyond those for the caffe itself, which can be taken from `http://caffe.berkeleyvision.org/installation.html`

Installing and Running

Out of the box: using *Singularity*

We offer a *Singularity* image which enables a very convenient way to calculate heatmaps for pre-trained *caffe* models on imagenet. The image also offers an example for the installation process and workflow and facilitates getting started with the *caffe* toolbox. *Singularity* [6] is a container platform that enables a convenient way to run, store and share code in a controlled environment while offering good hardware management options. Inside the container, all dependencies are taken care of and experiments can be rerun on exactly the same hardware settings making them easily reprocucible. For more material on *Singularity* , we refer to the authors website (`https://sylabs.io/singularity/`). In order to download and install the *Singularity* platform, follow the instructions in the user guide (`https://sylabs.io/guides/3.5/user-guide/quick_start.html`).

The singularity image can be found in the `lrp_toolbox/singularity` folder and is built by calling

```
singularity build --fakeroot --force caffe-lrp-cpu-u16.04.sif caffe-lrp-cpu-u16.04.def
```

The resulting `.sif` file has all necessary dependencies isntalled and contains demonstration files as well as the needed pre-trained models for them. All material for the demo can be found in the `lrp_demo` folder. Start the heatmapping process with

```
bash lrp_demo.sh
```

and find the results in `lrp_demo/lrp_output/someimages/`. Since *Singularity* containers are read-only, the skript links the `lrp_demo` folder to the equivalent folder on the image and then calls the lrp routine from within the container. The easiest way to add additional examples is to copy image files to the `lrp_demo/someimages` folder and add the corresponding filenames to the `lrp_demo/testfilelist.txt` file. In order to change *LRP* parameters adjust the `lrp_demo/config.txt`. Since the contents of these folders are linked to the image when executing the `lrp_demo.sh` skript, the image itself does not need to be rebuilt.

Matlab and python

The toolbox implementations for Matlab and python can be run out of the box, given all requirements in Section 2 are fulfilled. After downloading the toolbox as `.zip`- or `.tar.gz`-archives and unpacking and merging its contents to disk, the demo applications can be executed (with `[optional]` substrings indicated by brackets) as

```
cd <toolbox_location>/matlab
matlab -nodesktop -r lrp_[cnn_]demo
```

for the Matlab implementation and

```
cd <toolbox_location>/python
python lrp_[cnn_]demo.py
```

for python. On systems running Ubuntu 14.04 LTS (other versions might also be supported), executing

```
cd <toolbox_location>/python
sudo bash install.sh
```

will install all required software packages, download the MNIST [8] data set and several neural network models trained to discriminate between digit classes and then execute `lrp_demo`. Please read and modify `install.sh` carefully to suit your needs.

Caffe open-source package

The LRP implementation for Caffe has been forked from the Caffe repository on October 3rd 2015 (rc2). After fulfilling all requirements listed above, execute

```
cd <toolbox_location>/caffe-master-lrp
make clean
make all
```

Note that the Caffe-based LRP implementation currently only supports execution only on CPUs and already provides a correspondingly condigured `Makefile.config`.

After successfully building Caffe, navigate to the subdirectory `demonstrator` and execute `build.sh` (include and library paths might have to be adjusted) to compile the executables `lrp_demo`, `lrp_parallel_demo`, `lrp_demo_minimaloutput` and `lrp_parallel_demo_minimaloutput` for sequential and parallel heatmap computation respectively. Note that while the binaries ending in `_minimaloutput` will only write out plain text relevance maps and the ten dominantly predicted classes, while the other binaries will also produce images showing the actual resized and padded network input, as well as a heatmap visualization as images.

The provided script `download_model.sh` may be used to download and extract the BVLC reference Caffe network model. To use your own caffe files, simply adapt the configuration files.

```
cd <toolbox_location>/caffe-master-lrp/demonstrator
./build.sh
./download_model.sh
```

The caffe heatmap computation binary then called as e.g.

```
./lrp_demo ./config_sequential.txt ./testfilelist.txt ./
```

Where `config_sequential.txt` is the config file for the sequentially operating binary. `testfilelist.txt` lists image names to compute heatmaps for, with an optional prediction target. The contents of both files are discussed further below in Section Caffe specifics. The last parameter — here `./` — as a path prefix added to each image file in `testfilelist.txt`. Should the paths in `testfilelist.txt` be full paths already, `/` should be the working choice. The first part of above command `./lrp_demo ./config_sequential.txt [...]` may be exchanged to `./lrp_parallel_demo ./config.txt [...]` to process all images listed in `./testfilelist.txt` in parallel at the cost of increased memory consumption.

On systems running Ubuntu 14.04 LTS (other versions might also be supported), executing

```
cd <toolbox_location>/caffe-master-lrp
bash install.sh
```

will install all required software packages, build Caffe and the demonstrator application and compute a small set of heatmaps for chosen example images.

3 LRP for Artificial Neural Networks Summarized

Refreshing neural network prediction

Artificial neural networks are commonly built from layers of computation

$$z_{ij} = x_i w_{ij} \tag{1}$$

$$z_j = \sum_i z_{ij} + b_j \tag{2}$$

$$x_j = g(z_j) \tag{3}$$

where x_i is the ith index of the input $\boldsymbol{x}$ of a (hidden) layer l, with w_{ij} connecting the ith unit from layer l to the jth unit of the succeeding layer $l+1$. The variables z_{ij} describe the *forward messages* from the input neurons i to the output neurons j, which are then aggregated by summation together with a bias term b_j. An optional non-linear activation function is described by g. Commonly used non-linearities are the hyperbolic tangent $g(z_j) = \tanh(z_j)$, the rectifying linear unit (ReLU) $g(z_j) = \max(0, z_j)$ or a softmax function $g(z_j) = \exp(z_j)/\sum_{j'} \exp(z_{j'})$.

Multilayer neural networks stack several of these layers, each of them composed of a greater number of neurons. This formulation of neural networks is also sufficient to cover a wide range of architectures such as the simple multilayer perceptron or convolutional neural networks.

LRP for Artificial Neural Networks

While the forward pass through an artificial neural network sends messages from the nodes from one layer to the succeeding ones, computing a final layer prediction starting at the input step-by-step, LRP moves over the layers in opposite direction to resolve the classifier's output as *relevance messages* $R_{i\leftarrow j}^{(l,l+1)}$ sent from a layer $l+1$ to its predecessor l:

Knowing the relevance $R_j^{(l+1)}$ of an upper layer neuron j at layer $l+1$, the goal is to obtain a decomposition into relevance messages which can then be aggregated again at the receiving nodes i in layer l. As expressed in [1], a set of constraints

$$R_j^{(l+1)} = \sum_i R_{i\leftarrow j}^{(l,l+1)} \tag{4}$$

$$R_i^{(l)} = \sum_j R_{i\leftarrow j}^{(l,l+1)} \tag{5}$$

must hold for the total amount of relevance to remain constant, e.g.

$$f(\boldsymbol{x}) = \dots = \sum_{j\in(l+1)} R_j^{(l+1)} = \sum_{i\in(l)} R_i^{(l)} = \dots = \sum_{d=1}^{\dim(\boldsymbol{x})} R_d^{(1)} \tag{6}$$

ensuring the relevance conservation principle of LRP holds.

Deep neural networks are sequences of linear or convolution layers alternating with non-linear activations and/or pooling steps and the decomposition process of LRP operates in a layer-by-layer manner. In the case of a linear fully connected layer

$$z_{ij} = x_i w_{ij} \quad z_j = \sum_i z_{ij} + b_j \tag{7}$$

such a decomposition is immediately given by $R_{i\leftarrow j} = z_{ij}$. Therefore, a first possible choice of relevance decomposition is given as

$$R_{i\leftarrow j}^{(l,l+1)} = \frac{z_{ij}}{z_j} \cdot R_j^{(l+1)} \tag{8}$$

easily showing to approximate the conservation properties of Equation 6. Specifically,

$$\sum_i R_{i\leftarrow j}^{(l,l+1)} = R_j^{(l+1)} \cdot (1 - \frac{b_j}{z_j}) \tag{9}$$

since the bias term b_j does inject information during the forward pass as an always active input node and absorbs (or injects) relevance during the application of LRP proportionately. Note that a treatment of the bias term as a weight connected and constantly activated input node x_0 with $x_0 w_{0j} = b_j$, above formula uniformly yields a decomposition into relevance messages proportionally to the forward messages $R_{(i,j)}^{(l,l+1)} \propto z_{ij}$. While the treatment of convolutional and pooling layers is identical to that of fully connected layers, activation layers operate in a component-wise manner, i.e. $z_j = g(x_i) \Leftrightarrow z_{ij} = g(x_i)\delta_{ij}$, where $g(\cdot)$ is a non-linear activation function and δ_{ij} describes the kronecker delta. Consequently, the relevance messages $R_{i\leftarrow j}^{(l,l+1)}$ for activation layers describe an identity function $R_i^{(l)} = R_j^{(l+1)}$, making LRP a piece-wise linear decomposition process for each layer. However, the integration of the forward-passed and potentially non-linear activations x_i into the z_{ij} of Equation 8 (and the following decompositions below), renders LRP an in overall non-linear and unsupervised per-sample explanation method.

Once a message decomposition formula has been selected from Equations 8 to 15 — all of which are part of the toolbox implementation — the lower layer relevances $R_i^{(l)}$ are computed as a sum aggregation of incoming messages in consistency with the constraints in Equation 4 and5:

$$R_i^{(l)} = \sum_j = R_{i\leftarrow j}^{(l,l+1)} \tag{10}$$

Decomposition Variants and Parameters

The ϵ-decomposition formula. The disadvantage of the decomposition in Equation 8 is that relevance messages $R_{i\leftarrow j}^{(l,l+1)}$ may become unbounded for very small z_j, which can be circumvented by introducing a sign-dependant numerical stabilizer ϵ in the denominator.

$$R_{i\leftarrow j}^{(l,l+1)} = \frac{z_{ij}}{z_j + \epsilon \cdot sign(z_j)} \cdot R_j^{(l+1)} \tag{11}$$

This approach, although being an easy to understand and straight-forward solution to the issue of numerical instability, does leak relevance into ϵ and even may fully absorb relevance if ϵ becomes very large. The ϵ-decomposition formula explains the predictor output "as-is", while control over noisy explanations may be exerted via setting ϵ to appropriate values.

The $\alpha-\beta$-decomposition formula. An alternative and numerically conservative stabilizing method is the $\alpha-\beta$-method (or shorter, just α-decomposition). This decomposition treats the activating and inhibiting activation factors of a layer separately, e.g. let $z_j^+ = \sum_i z_{ij}^+ + b_j^+$ and $z_j^- = \sum_i z_{ij}^- + b_j^-$ with

$$z_{ij}^+ = \begin{cases} z_{ij} & ; \ z_{ij} > 0 \\ 0 & ; \ else \end{cases} \text{ and } z_{ij}^- = \begin{cases} z_{ij} & ; \ z_{ij} < 0 \\ 0 & ; \ else \end{cases} \tag{12}$$

and b^+ and b^- being subject to the same rule. The relevance decomposition into messages $R_{i\leftarrow j}^{(l,l+1)}$ as a weighted combination of positive (activating) and negative (inhibiting) contributions is then calculated as

$$R_{i\leftarrow j}^{(l,l+1)} = \left(\alpha \cdot \frac{z_{ij}^+}{z_j^+} + \beta \cdot \frac{z_{ij}^-}{z_j^-}\right) \cdot R_j^{(l+1)} \tag{13}$$

Adding the constraint $\alpha+\beta = 1$ ensures the relevance conservation principle to hold and reduces the number of added free parameters to one.

This decomposition method is especially usefull when paired with a networks making use of ReLU activation layers, since hidden layer inputs will always be ≥ 0 and their pre-activation outputs also need to be ≥ 0 to "fire". This means that positively weighted connections w_{ij} — which all factor into z_{ij}^+ — propagate the inputs in an activatingly, while negative weights create deactivating messages z_{ij}^-. So by choosing α (and therefore β) appropriately, allows to choose what is backwards-propagated in terms of relevance. For example setting $\alpha = 1$, causes $\beta = 0$ will result in relevance maps which yield insight into which features cause neuron activations, while $\alpha = 2, \beta = -1$ yields a more balanced explanation with still more focus on the neuron-exciting inputs. Be careful, however, with setting a variable to 0: Sum-pooling after a ReLu activation layer will only yield positive activations. Here, setting $\alpha = 0, \beta = 1$ will result in no heatmap at all, since there are no inhibiting layer inputs to propagate the relevance to.

The w^2-decomposition formula The w^2-rule was first introduced in [9] and is obtained by choosing the closest root to $\boldsymbol{x}$ in direction of maximal descent when using Deep Taylor Decomposition. Although it might seem that this decomposition is independent to the input data, the resulting heatmap explanations are unique for each data point through the influence of $R^{(l)}$. The relevance messages for this decomposition compute as

$$R_{i\leftarrow j}^{(l,l+1)} = \frac{w_{ij}^2}{\sum\limits_{i'} w_{i'j}^2} R_j^{(l)} \tag{14}$$

This decomposition might come in handy when the resulting relevance map when using Equations 11 or 13 are too sparse for the intended purposes, e.g. because the input level activations include zero valued variables which can not absorb any relevance ($z_{ij} = 0 w_{ij} = 0 \Rightarrow R_{i\leftarrow j}^{(l,l+1)} = 0$), or it is in general desirable to reduce the resolution of the explanatory heatmap.

Flat weight decomposition The flat weight decomposition removes any model- and data-dependant influence for the decomposed layer altogether (except model- and data-specific relevance input from $R^{(l+1)}$), by simply projecting the upper layer relevance values $R_{i\leftarrow j}^{(l+1)}$ uniformly towards their receptive fields.

$$R_{i\leftarrow j}^{(l,l+1)} = \frac{1}{\sum\limits_{i'} 1} R_j^{(l)} \tag{15}$$

The resulting relevance maps will yield explanations of input representations in higher network layers, potentially changing the semantics of the heatmaps. This decomposition method is further discussed in [2]. Note that for fully connected layers, the application of Equation 15 will yield completely uniform and structure-less relevance maps. Using Equation 14 instead is advised.

4 Implementation and Examples

This section will provide detailed information about the implementation of the LRP Toolbox and demonstrates the basic workflow.

How to use the toolbox: Parameters and Results

The use of the LRP toolbox requires in general

- A pre-trained neural network model
- Data for performing a neural network prediction and LRP
- Parameters for the LRP algorithm (optional)

with the usual workflow consisting of the following steps:

1. Obtain trained classifier $f(\cdot)$ and data compatible with classifier
2. Forward-pass data $\boldsymbol{x}$ through classifier, obtain $y_{\text{pred}} = f(\boldsymbol{x})$. The forward pass is not only necessary to obtain a prediction y_{pred} but more importantly to populate the classifier's internal variables with information specific to $\boldsymbol{x}$.
3. Apply LRP to obtain input layer relevances $R_d^{(1)}$ explaining the decision of the classifier for $\boldsymbol{x}$ ...
    - wrt to the prediction y_{pred}
    - or: wrt to a chosen prediction target y_{choice}

   by setting the desired final layer relevance as input to LRP. See [1], `mini.{py,m}`, `lrp_demo.{py,m}` and `lrp_cnn_demo.{py,m}` for details and examples.
4. Visualize input-layer relevances $R_d^{(1)}$ as a heatmap (optional)

Examples and Interpretation of Results

The LRP Toolbox provides working examples on pre-trained models for the MNIST [8] data sets for the python and Matlab implementations and an example implementation of an LRP application for arbitrary photographic image data for Caffe. Data and models for the Matlab and python examples in respective binary formats and a shared plain text format are provided accordingly. For Caffe, exemplary image data is provided within the download archives and a reference model and corresponding meta information can be downloaded by following above installation steps. In the following, minimal working examples for both Matlab and python demonstrate the basic workflow with Figure 1 presenting the resulting images. Both minimal working examples are included as `<toolbox_path>/matlab/mini.m` and `<toolbox_path>/python/mini.py`

Matlab

```
% imports
import model_io.*
import data_io.*
import render.*
% end of imports




% read model and MNIST test data
nn = model_io.read(<model_path>);
X = data_io.read(<data_path>);
% pick first image, normalize to [-1 1]
X = X(1,:) / 127.5 - 1;

% forward pass through network
Ypred = nn.forward(X);
% apply lrp to explain prediction of X
R = nn.lrp(Ypred);

% render rgb images and save as image
digit = render.digit_to_rgb(X);
% render heatmap R, use X as outline
hm = render.hm_to_rgb(R,X);
render.save_image({digit,hm},<i_path>);
```

python

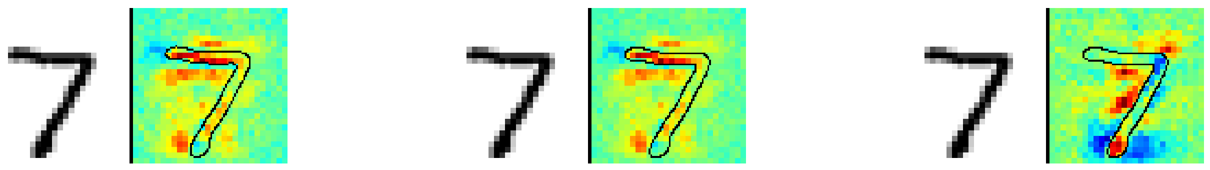
```
# imports
import model_io
import data_io
import render

import numpy as np
na = np.newaxis
# end of imports

# read model and first MNIST test image
nn = model_io.read(<model_path>)
X = data_io.read(<data_path>)[na,0,:]
# normalized data to range [-1 1]
X = X / 127.5 - 1

# forward pass through network
Ypred = nn.forward(X)
# lrp to explain prediction of X
R = nn.lrp(Ypred)

# render rgb images and save as image
digit = render.digit_to_rgb(X)
# render heatmap R, use X as outline
hm = render.hm_to_rgb(R,X)
render.save_image([digit,hm],<i_path>)
```



(a) Matlab : true class (b) python : true class (c) python : class 2

Figure 1: (a) and (b) : Heatmap visualizations computed as a result of the execution of both above listed code samples. (c) : The input digit explained as a 2 by setting input for `nn.lrp()` to `np.array([0,0,1,0,0,0,0,0,0,0])[na,:]`. The image is a result of the python implementation. For both languages, the input parameters in angle brackets specifying the loaded model and data locations have been set to
`<model_path> = '<toolbox_path>/models/MNIST/long-rect.txt'` and
`<data_path> = '<toolbox_path>/data/MNIST/test_images.txt'`
The resulting rgb-images have been written to a location specified by `<i_path>`. Slight differences in digit outlines and color hue are the result of internal differences of the used color maps and edge detection functions provided by Matlab and the respective python modules. The heatmaps themselves can be expected to be identical for both implementations, up to the limits numerical precision.

Interpretation of Heatmap Scores

LRP computes a relevance score $R_d^{(1)}$ for each component of the classifier input x_d in its given state. For the given model (high) positive values $f(\boldsymbol{x})$ indicate the presence of an object or concept in an input sample. Likewise, positive values $R_d^{(1)}$ argue for evidence for the presence of a concept — the (chosen) prediction target — via the state of x_d.

Figures 1 (a) and (b) demonstrate heatmaps for an input digit showing a hand-written 7 next to a heatmap visualizing the input layer relevance values rating the input pixel's contribution towards the detection of a digit 7, as computed with the samples implementations in `mini.{py,m}`. (High) positive scores are visualized in orange to red color, while colder color hues signify neutral (green) to negative (blue) contributions to the detection of class 7.

Note that due to the heatmap the top horizontal bar of the digit has been identified as a striking feature of the detected class, the "leg" of the hand written digit is rated as less important. This can be explained as this feature can be shared by digit classes 2,4 and 9, while a straight horizontal bar at the top is not. Digits of class 5 however might share this feature, yet (due to our observation) tend to be angled differently in general, e.g. sloping down to the left more. Note the also strong positive evidence outside the written digit: At the bottom, a small area has been identified as containing high positive evidence speaking for class 7, with a less pronounced counterpart to the right. This can be interpreted as the absence of color contributing to the 7-ness of the data point. In other words: If those pixels would be colored black, the digit would more closely resemble a 2. The same interpretation can be applied to the horizontal bar at the top of the image. LRP attributes high relevance to the blank pixels above and below the horizontal bar. Coloring those pixels black would cause the 7 to resemble a thickly drawn digit 9. Note the negative evidence — speaking against the 7-ness of the input digit — to the left of the top bar of the 7, explaining that the classifier would predict the input image as a seven with increased certainty, if the horizontal bar would be longer. This example shows, that heatmaps computed by LRP are not segmentation masks but rate each input unit — on top of the object and in background areas — according to how their state influences the prediction of the classifier output.

Figure 1 (c) demonstrates the capability of the LRP algorithm to explain *would-be* decisions of a classifier, e.g. in the above case why and how the input digit resembles a 2 in the eyes of the classifier, or what parts are considered *wrong*. The classifier considers the empty space below the angle at the top of the seven as evidence for the digit being a 2, yet attributes negative evidence to the digit's *pointy end* to the top right, since the typical member of class 2 can be expected to be more round in shape. The classifier also identifies most of the "foot" as missing (rather: as unexpected for the explanation target) in the input digit, in order for it to more closely resemble a digit 2.

Pixel-wise explanation always need to be interpreted in context of the problem the classifier has been trained to solve, or vice versa provide an insight to the problem setting with its peculiarities in the first place (E.g. for an application of LRP leading to interesting discoveries about the Pascal VOC [3] data set, see [7]).

Implementation

The Matlab and python stand-alone implementations of the LRP Toolbox are designed to be as identical as possible in structure, while the LRP code for Caffe is conceived as a plugin to replace existing neural network layer implementations. Throughout this section, the structure and functionality of the Matlab and python implementations will be described in detail, followed by a delimination of the modifications made to the Caffe code in oder to support LRP. Additionally to the descriptions of modules, packages and functions found below, a documentation can be found within the uncompiled code itself.

Efficiency

The python implementation of the LRP functionality heavily relies on numpy for efficiency, which shares the the use of LAPACK and BLAS with Matlab. LRP for Caffe does currently only support (parellel) processing on the CPU, using ATLAS, OpenBLAS, etc .

Data Import / Export

Data

– **python** : Data can be read and written with `data_io.read()` and `data_io.write()` and is assumed to be block-formatted as a $[N \times D]$ matrix. Supported are plain text ascii-matrices as `.txt` files, numpy's `.npy` and `.npz` formats and also `.mat` files via `scipy.io`. For the latter, the toolbox assumes the file to contain one matrix accessible via the key *data*.

– **Matlab** : The Matlab implementation realizes data IO via `+data_io.read()` and `+data_io.write()` and supports `.mat` files next to plain text ascii-matrices in $[N \times D]$ block format.

- **Caffe** : The add-on for Caffe loads images using ImageMagick and thus supports a wide range of image data including JPEG and PNG.

Models

- **python** : The python implementation can read and write neural network model files via functions `model_io.read()` and `model_io.write()`. Supported are python's `pickle` object serialization format and a plain text file format shared with Matlab and described further below. Default file name extensions for the pickled in- and output format are `.nn` and `.pickle` and `.txt` for the plain text description of the model. See the manual for details.
- **Matlab** : Supported are the with python shared plain text model description via `.txt` files and `.mat` files holding model information. Reading and Writing can be realized using `+model_io.read()` and `+model_io.write()`.
- **Caffe** : It is able to read the models trained with the Caffe package, e.g. those from the Caffe Model Zoo.

Heatmaps

- **python** : The output of raw heatmap data can be managed using the module `data_io`. Visualizing and writing the heatmap as rgb images is achieved using the functions `render.hm_to_rgb()` and `render.save_image()`. The former converts an input relevance matrix or vector using a color map to a rgb image and the latter combines and writes a series of images in various formats.
- **Matlab** : The Matlab implementation offers the same functionality as its python counterpart via correspondingly named functions provided in package `+render`. An example on how to use these functions to draw and save heatmaps is given in the demo implementation.
- **Caffe** : The C++ version outputs four files into the path specified in the configfile as `standalone_outpath`. The first two files,

  $\langle imagefilename \rangle$`_as_inputted_into_the_dnn.jpg` and

  $\langle imagefilename \rangle$`_heatmap.jpg` are simply the image how it looks like when it enters the quadratic receptive field of the neural network, and the resulting heatmap. While they are easily to be read in image viewers, they are unsuited for further processing of the heatmap. For that reason the file

  $\langle imagefilename \rangle$`_rawhm.txt` provides a the raw unnormalized heatmap scores as plain text. For a detailed explanation of the format of this file please refer to the manual. The last file, $\langle imagefilename \rangle$`_toptenscores.txt` outputs the indices of the highest ranked classes and their scores. Note that when using the `*_minimaloutput` - implementations, only the plain text outputs will be generated.

Matlab and python modules

The stand-alone implementations of Matlab and python have been designed with a highly similar module or package structure within the possibilities of the programming languages. Modules for python are realized as single `*.py`-files containing all the module's functions. Synonymous to the modules of python are packages for Matlab, sharing the same names as their python counterparts. The difference is that packages for Matlab are realized as folders (with a leading `+` sign in front of the package name for Matlab to identify the folder as a package), with all implemented package functions or classes located as separate `.m`-files therein. The names and functionality of all implemented functions are to a large extend identical in between both stand-alone implementations and will therefore be outlined in unison. Notable differences will be highlighted in the respective descriptions using the flags <Matlab-specific>$^{\text{Matlab}}$and <python-specific>$^{\text{python}}$in case certain traits are not shared between the implementations. The descriptions of modules and packages are synonymous for a structural description and will be considered interchangeable. The following description will start with explaining the functions `lrp_demo`, `lrp_cnn_demo` and `mini` as entry points to example implementations first and continue with the module structure. For a more detailed in-depth description of class- and method signatures please refer to the source code.

Example implementations of the LRP pipeline:

- `lrp_demo` : This demo loads the MNIST test data set and a pre-trained classifier with rectification activation units. After normalizing the test data, LRP is performed on a selection of test data points, followed by rendering and saving the produced heatmap. Examples on how to choose a LRP decomposition method or prediction targets are given yet commented out. This script displays each rendered heatmap to the user.

- `lrp_cnn_demo` follows the structure of `lrp_demo`, but instead uses a pre-trained LeNet-5 convolutional neural network. In this script, functionality for also setting lrp parameters for each neural network layer, as introduced in version 1.2.0 of this toolbox, are demonstrated.
- `mini` : A minimal execution example for LRP for only one data point from the MNIST test data set.

Package structure:

- `data_io` provides functionality for reading and writing block-formatted data matrices in various formats.
  - `read(path, fmt)` reads a data matrix from disc, located at `path`. If `fmt` is not given, the correct file format is inferred from the extension of `path`. Supported extensions / values for `fmt` are `'npy'`$^{\text{python}}$, `'npz'`$^{\text{python}}$, `'txt'` , `'mat'`. Returns a matrix$^{\text{Matlab}}$or numpy array$^{\text{python}}$.
  - `write(data, path, fmt)` receives a data matrix `data` and writes it to `path` in format `fmt`. If `fmt` is not given, the correct file format is inferred from the extension of `path`. Supported extensions / values for `fmt` are `'npy'`$^{\text{python}}$, `'npz'`$^{\text{python}}$, `'txt'` , `'mat'`.
- `model_io` provides an IO interface to read and write serialized neural network models in various formats.
  - `read(path, fmt)` reads a neural network model from `path`, assuming a format `fmt`. Should `fmt` not be given, it — and with that the correct way to read — will be inferred from the file name extension in `path`. Supported formats and values for `fmt` are `'pickle'`$^{\text{python}}$, `'pickled'`$^{\text{python}}$, `'nn'`$^{\text{python}}$, `'txt'`, `'mat'`$^{\text{Matlab}}$, where the python-specific formats are implemented using python's `pickle` module and `txt` is a plain text model description format shared between Matlab and python explained further below. Returns the neural network model as an instance of `modules.Sequential`.
  - `write(model, path, fmt)` writes a neural network model represented via an instance of `modules.Sequential` to `path` in a given output format `fmt`. Should `fmt` not be given, it will be inferred from the file name extension in `path`. Supported formats and values for `fmt` are `'pickle'`$^{\text{python}}$, `'pickled'`$^{\text{python}}$, `'nn'`$^{\text{python}}$, `'txt'`, `'mat'`$^{\text{Matlab}}$, where the python-specific formats are implemented using python's `pickle` module and `txt` is a plain text model description format shared between Matlab and python explained further below.
- `modules`
  - `Module` implements an abstract class for neural network modules, defining a common interface which all other classes inheriting `Module` implement. The inherited default behaviour of all method stubs is to do nothing when called.
    * If not implemented by inheriting methods, the function `forward(X)` will return `X` without change.
    * `set_lrp_parameters(lrp_var,lrp_param)` allows to pre-set lrp parameters per layer, with `lrp_var` identifying the desired decomposition method per name as string type input, and `lrp_param` providing a method-specific parameter.
    * `lrp(R, lrp_var, param)` performs LRP wrt to an initial relevance input `R` and a previously fed-forward batch of data via a call of `forward(X)`. The parameters `lrp_var` and `param` are optional. If not given, the model (e.g. each layer) tries to use pre-set lrp parameters or will default to the simple lrp implementation from Equation 8.
      `lrp_param` $\in$ { `None`$^{\text{python}}$, `[]`$^{\text{Matlab}}$,`'none'` , `'simple'`, `'epsilon'`,`'eps'`, `'alphabeta'`, `'alpha'`,`'ww'`,`'w^2'`,`'flat'` } defines the decomposition alternative to use for all layers during the application of LRP. By default, the simple decomposition method in Equation 8 will be applied, or any specific lrp parameters set using `set_lrp_parameters`. Setting the parameter to `'epsilon'` or `'alphabeta'` will cause the application of Equations 11 and 13

respectively. `'eps'` and `'alpha'` are short equivalents for both methods. `'ww'` or `'w^2'` will apply Equation 14 and `'flat'` will apply Equation 15. Some layers do have default fall-back decompositions when the provided decomposition choice does not make sense.
The parameter `param` may be used to set the corresponding free parameters ϵ and α of both methods. Default value is `[]`$^{\texttt{Matlab}}$ or `None`$^{\texttt{python}}$.

– `Sequential` represents the instance of an artificial neural network.
  * `Sequential(modules)` Constructor. The parameter `modules` is an enumerable collection$^{\text{python}}$/ cell array$^{\text{Matlab}}$of instances of class `Module` as computational layers of the network.
  * `forward(X)` receives an input `X`, propagates it through the network and populates the network with input-specific data and internal data representations. Returns a network output.
  * `train(X, Y, Xval, Yval, batchsize, iters, lrate, lrate_decay, lfactor_initial, status, convergence, transform)` implements a training procedure for the neural network implementation, using the error backpropagation algorithm. `train` receives a set of training data `X` and training labels `Y`, on which the network is trained over `iters` (default is 10000) training iterations with mini batches of `batchsize` (default 25) samples at a time. Every `status` (default is 250) training iterations, the network is evaluated on the validation data `Xval` with labels `Yval`. If no input for the validation set is provided the whole training data is used for estimating the prediction performance. A training iteration consists of a forward pass of a minibatch, a backward pass of the delta between the prediction and the expected results followed by an update step. The speed of learning is determined by `lrate` (default is 0.005). `lrate_decay` controls whether and how the learning rate will diminish during trainig, depending on the network performance. The default value is `'none'`, with alternatives being `'linear'` and `'sublinear'`. `lfactor_initial` (default 1.0) is a multiplicative factor to the base learning rate. Using this, with multiple consecutive calls of `train` allows for a step-wise decrease in the learning rate over the complete network training. `convergence` (-1) is an integer value defining the number of iterations allowed without measurable network improvements, until model convergence is declared and the training is stopped. Values smaller than zero will disable this convergence check. The input variable `transform` accepts the handle of a function, which is to be applied to each input batch. This allows for slight input transformations, such as the addition of noise, etc. Default value is `None`, which is equivalent to the application of an identity function (aka. does nothing).
  * `backward(DY)` performs a backward pass through the network, taking the error gradient `DY` as input.
  * `update(lrate)` updates the network modules, after the error gradient has been computed using `backward`. `lrate` is a multiplicative factor determining the training velocity.
  * `lrp(R,lrp_var, lrp_param)` The method returns first layer relevance values $R_d^{(1)}$. See the description of `modules.Module.lrp` for details.
  * `clean()` iterates over all layers of the neural network and calls `clean()` on the implementing modules, removing temporary data necessary for LRP which has been memorized during the forward pass. This method is called in `model_io.write()`.

– `Linear` represents a linear layer.
  * `Linear(m,n)` creates a linear neural network layer instance with `m` input dimensions and `n` output dimensions.
  * `forward(X)` performs a forward-pass of the input `X` and returns an `n` dimensional output by applying $Y = \texttt{X}W + B$, where W is a weight matrix, B the bias term and Y the output.
  * `lrp(R, lrp_var, param)` performs LRP with upper layer relevances $R^{(l+1)}$ and returns relevance values $R^{(l)}$ for the layer input. See the description of `modules.Module.lrp` for details on the possible range of parameters.

– `Tanh` implements an activation layer with non-linear tanh units.
  * `forward(X)` applies the tanh function to the input `X` and returns the result.

– `Rect` implements a rectification activation layer.

        - `forward(X)` computes and returns $\max(0, \texttt{X})$
    - `SoftMax` implements a soft-max normalization layer
        - `forward(X)` computes and returns $\forall i \; \frac{\exp(\texttt{X}_i)}{\sum_j \exp(\texttt{X}_j)}$ for each row of `X`

- `render` provides functionality to draw and save relevances and heatmaps as rgb-images
    - `digit_to_rgb(X, scaling, shape, cmap)` renders a given data point `X` as a rgb-image using a color map `cmap`. The parameter `shape` specifies the original shape of `X`, to which the data is reshaped. The default value depends on the default behaviour of `render.vec2im(). scaling` is a positive integer vector describing a scaling factor, by which the image is enlarged (after reshaping) by pixel value replication.
    - `hm_to_rgb(R, X, scaling, shape, sigma, cmap, normalize)` operates in a similar manner as `render.digit_to_rgb`, but renders the input heatmap `R` as an rgb-image and uses the data point `X` to compute an outline to overlay it with the heatmap image. For Matlab, this requires the availability of the Image Processing Toolbox. The parameter `sigma` is forwarded to a call of a canny edge detector for computing the outline of `X`. `normalize` controls whether the input heatmap `R` is to be normalized such that $\max(|\texttt{R}|) = 1$. Returns the visualized heatmap as rgb-image (a $[H \times W \times 3]$ - sized matrix$^{\text{Matlab}}$or `numpy.ndarray`$^{\text{python}}$)
    - `enlarge_image(img, scaling)` receives a $[H \times W]$ or $[H \times W \times D]$ matrix$^{\text{Matlab}}$or `numpy.ndarray`$^{\text{python}}$`img` and a positive integer value `scaling` as scaling factor and returns a $[H \cdot \texttt{scaling} \times W \cdot \texttt{scaling}]$ or $[H \cdot \texttt{scaling} \times W \cdot \texttt{scaling} \times D]$ output of the same type by pixel value replication.
    - `repaint_corner_pixels(rgbimg, scaling)` receives a $[H \times W \times 3]$ - sized rgb image as matrix$^{\text{Matlab}}$or `numpy.ndarray`$^{\text{python}}$and a positive integer value `scaling` and replaces the bottom right and top left pixel rgb values with the color mean of all neighbouring pixels, in order to mask artificial heatmap responses as a result setting relevance anchor values during heatmap normalization. `scaling` holds information about the pixel size *after* scaling the image. This workaround was/is necessary for successfully applying color maps using matlab and has become obsolete for the python implementation with toolbox version 1.2.0, thus has been dectivated in `digit_to_rgb`$^{\text{python}}$.
    - `vec2im(V, shape)` receives a matrix$^{\text{Matlab}}$or `numpy.ndarray`$^{\text{python}}$and returns it `shape`-shaped. If `shape` is not given, the algorithm tries to reshape `V` such that its side lengths are equal.
    - `save_image(rgb_images, path, gap)` receives a series of rgb images in an enumerable container$^{\text{python}}$or cell array$^{\text{Matlab}}$and concatenates them horizontally. In between the images, a column of black pixels `gap` pixels wide is introduced. The assembled image is then returned as a $[H \times W \times 3]$ - sized matrix$^{\text{Matlab}}$or `numpy.ndarray`$^{\text{python}}$and written to `path`, where the file name extension of `path` controls the output format.

Caffe

For implementing LRP for the Caffe open source framework, several source files have been edited to support the required functionality, which can then be used by the example implementations in subfolder `toolbox_path/caffe-master-lrp/demonstrator/`. Modified files and changes made are listed below:

- `include/caffe/` : Contains the following modified header files which now define interfaces for LRP functionality
    - `common_layers.hpp`
    - `data_layers.hpp`
    - `layer.hpp`
    - `loss_layers.hpp`
    - `net.hpp`
    - `neuron_layers.hpp`
    - `relpropopts.hpp` : newly added file

  - vision_layers.hpp

- `src/caffe/` : Contains the following modified code files implementing the LRP steps
  - `layer.cpp`
  - `net.cpp`
    - `layers/`
      - `base_conv_layer.cpp`
      - `concat_layer.cpp`
      - `conv_layer.cpp`
      - `data_transformer.cpp`
      - `dropout_layer.cpp`
      - `hinge_loss_multilabel_layer.cpp` : added with version 1.0.1
      - `image_data_layer_multilabel.cpp` : added with version 1.0.1
      - `inner_product_layer.cpp`
      - `loss_layer.cpp`
      - `lrn_layer.cpp`
      - `lrn_layer.cu`
      - `neuron_layer.cpp`
      - `pooling_layer.cpp`
      - `relu_layer.cpp`
      - `softmax_layer.cpp`
      - `split_layer.cpp`

Relevant C++ Interfaces

There are two different interfaces, in class `net` (in `net.hpp` and `net.cpp`)

```
  void net::Backward_Relevance(const std::vector<int> & classinds,
  vector<vector<double> > & rawhm,
  const relpropopts & ro);
```

and

```
  void net::Backward_Relevance_multi(const std::vector< std::vector<int> > & classinds,
  vector< vector<vector<double> > > & rawhm,
  const relpropopts & ro);
```

The first one computes the heatmap for exactly one image. The second computes heatmaps for multiple images in at once which is in line with the ability of Caffe to compute a parallel forward pass. See in the model definition files the initial `10` in

```
input_shape {
  dim: 10
  dim: 3
  dim: 227
  dim: 227
}
```

which can be found in the file `deploy.prototxt` as part of the model description. This means that this model file allows to compute a 10-fold parallel forward pass. `void net::Backward_Relevance_multi` can be used with such a file to compute a 10-fold parallel LRP backward pass. Both interfaces require a forward pass to be run before calling them. The result is at best undefined otherwise.

For the first interface:

`classinds` contains the indices of all classes, for which the heatmap is to be computed as a composite of their respective network output scores.

`rawhm[ch][h+w*height]` contains the heatmap score for channel `ch` (`ch=0` is red, `ch=1` is green, `ch=2` is blue. The order is RGB unlike the internal BGR order of Caffe.), in y-coordinate `h` (`h=0` is the topmost coordinate of the image), and in x-coordinate `w` (`w=0` is the leftmost coordinate of the image).

The second interface is a vectorized version of the first interface, i.e. `classinds[k]` has the same meaning as `classinds` in the first image, being applied to the `k`-th image. The same holds for `rawhm[k]` which is the same as `rawhm` for the first interface, with the input corresponding to the `k`-th image.

`relpropopts` (found in `include/caffe/relpropopts.hpp`) is a class which carries the options used for LRP. The options are as follows:

`relpropopts.numclasses` the number of classes of the classification problem

`relpropopts.relpropformulatype`:

- 0 for the ϵ-type formula
- 2 for the $\alpha - \beta$-type formula
- 11 or 99 for Sensitivity Analysis [10]
- 26 for Deconvolution [12]
- 54 for ϵ and flat decompositions below a given layer
- 56 for ϵ and w^2 decompositions below a given layer
- 58 for $\alpha - \beta$ and flat decompositions below a given layer
- 60 for $\alpha - \beta$ and w^2 decompositions below a given layer
- 166 for Guided Backpropagation [11]
- 100 for LRP composite (see [5]): $\alpha - \beta$ for convolution layers, ϵ for the rest
- 102 for LRP composite (see 100) and flat below a given layer
- 104 for LRP composite (see 100) and w^2 below a given layer

`relpropopts.alphabeta_beta` the beta for the $\alpha - \beta$-type formula

`relpropopts.epsstab` the ϵ for the ϵ-type formula

`relpropopts.lastlayerindex` is the index of choice of the highest layer, e.g. the starting point for LRP.

`auxiliaryvariable_maxlayerindexforflatdistinconv` proves the layer index from which on either flat or w^2 relevance decomposition is performed (see `lrp.relpropformulatype`s 54 to 60.

`relpropopts.firstlayerindex` is the index of the lowest layer at which relevance scores are taken and copied into the output vector of vectors `rawhm`.

the format of `rawhm` is determined by

```
const int channels = bottom_vecs_[i][0]->channels();
const int hei = bottom_vecs_[i][0]->height();
const int wid = bottom_vecs_[i][0]->width();
```

Channel inversion is performed, which means that `rawhm[ch]` is taken from channel `C-1-ch`.

Other variables which are not supported in this release and have to be set to defaults (in parentheses) are:

`relpropopts.codeexectype` (0)

`relpropopts.lrn_forward_type` (0)

`relpropopts.lrn_backward_type` (1)

`relpropopts.maxpoolingtoavgpoolinginbackwardpass` (0)

`relpropopts.biastreatmenttype` (0)

The shared plain text file format

The plain text file format is shared among the Matlab and python implementations of the LRP Toolbox and describes the model by listing its computational layers line by line as

```
<Layername_i> [<shape parameters>]
[<Layer_params_i>]
```

The following layers apply component-wise transformations to the input and thus only require a single line to be fully parameterized, e.g.

```
Rect
Tanh
SoftMax
Flatten
```

for ReLu non-linearities, tanh-units and the softmax and flattening layer respectively. The linear layer implementation `modules.Linear` incorporates in raw text form as

```
Linear m n
W
B
```

with m and n being integer values describing the dimensions of the weight matrix W as $[m \times n]$ and W being the human readable ascii-representation of the matrix, where W is reshaped to a s ingle vector in C-order and written out as a white space separated line of floating point values. After the line describing W, the bias term B is written out as a single line of n white space separated floating point values.

Both Max- and SumPooling layer share the same structure, except for the layer name:

```
SumPool hpool wpool hstride wstride
```

and

```
MaxPool hpool wpool hstride wstride
```

with *hpool* and *wpool* being the pooling operations respective height and width and *hstride* and *wstride* being the vertical and horizontal stride between pooling applications.

Finally, Convolution layers are parsed out as

```
Convolution hf wf df nf hstride wstride
W
B
```

where *hstride* and *wstride* serve the same purpose as for the pooling layers, and hf, wf, df and nf are the filter bank's height, width and depth (= input depth = number of input channels) and number of filters (= output depth = number of output channels) respectively. As with the Linear layer W are the learned weights, reshaped in C-order to a single vector written out as a single line of white-space separated values. B is the vector if bias terms of the layer, equalling in length to nf.

An example of a simple network trained to solve the `XOR`-problem is given below. The model file is also included as plain text in `<toolbox_path>/models/XOR/xor_net_small.txt`

XOR-Solving Example Network

```
Linear 2 3
-2.01595799878 -2.05379403106 0.688953420218 1.20338836267 -1.7518249173 -1.90515935663
-0.519917325831 0.400368842573 0.0699950389094
Tanh
Linear 3 3
-1.18542075899 -1.62921811225 0.134698917906 0.111469267787 1.85227669488 [...]
0.116095097279 -0.0138454065897 0.0469443958438
Tanh
Linear 3 2
1.10940574175 0.26799513777 2.51842248017 -1.5497671807 -0.606042655911 0.197763706892
-0.115832556216 0.115832556216
SoftMax
```

5 Caffe specifics

Caffe output formats

The C++ version outputs four files into the path specified in the configuration file as `standalone_outpath`:

The first two files are

$\langle imagefilename \rangle$`_as_inputted_into_the_dnn.png` and

$\langle imagefilename \rangle$`_heatmap.png`. Those files are simply the image as it enters the quadratic receptive field of the neural network, and the resulting heatmap. While they are easily to be read in image viewers, they are unsuited for further processing of the heatmap. Note that both image files are *not* written out when using the implementations named `*_minimaloutput`. This code only writes out the results as plain the text files described below.

$\langle imagefilename \rangle$`_rawhm.txt` provides the raw unnormalized heatmap scores as plain text. The format is as follows:

```
C
H W
[row 1 of channel 1]
[...]
[row H of channel 1]
[row 1 of channel 2]
[...]
[row H of channel 2]
[...]
[row H of channel C]
```

Where `C` is the number of (color) channels, which usually is 3. `H` and `W` are the height and width of the output respectively, which are then followed by a concatenated plain text output of the heatmap responses for all channels, where each row holds `W` entries.

The last file,

$\langle imagefilename \rangle$`_toptenscores.txt` outputs the indices of the highest ranked classes and their scores and consists of at most ten lines with each line listing the 0-based index of the corresponding class. It is less than ten lines, if the classification problem has less than ten classes.

Caffe image file syntax

This file is denoted as `testfilelist.txt` in the above example. Each line contains two entries: Firstly, the path to the image. Secondly, after a white space character, an integer value. If that value equals -1, then the heatmap is computed for the highest top-scoring class. If the integer is -2, then the heatmap is computed for the 5 top-scoring classes. Each output will be initialized with the score of the highest inner-product layer, unless the option `lastlayerindex` in the config file specifies a different layer. If the integer is non-negative, then the heatmap will be computed for the class with that index. An example `testfilelist.txt` is provided with this release.

Caffe config file syntax

The configuration file syntax for LRP for Caffe is realized as pairs of lines, where the first line identifies the variable to be set and the second line delivering the value. Example configuration files `config.txt` and `config_sequential.txt` are provided with this release.

- `param_file` (string)
  File of the caffe layer definitions as text file. Usually has the ending `.prototxt`. Check whether the value `dims:  x` in the `prototxt` file is appropriate for your case. If you process only one image, you can set that to 1 and your code performs faster and uses less memory.
- `model_file` (string)
  File containing the Caffe model weights, usually as a binary file.
- `mean_file` (string)
  Absolute path to the image mean which gets substracted from the input image. Format is a Caffe blob proto if the value of use_mean_file_asbinaryprotoblob is a positive integer.
- `use_mean_file_asbinaryprotoblob` (integer)
  Set it to a positive integer if you to load a Caffe blob protofile as mean file.
- `synsetfile` (string)
  Absolute path to a file containing as many lines as number of classes, with each line being the class label of the respective class.
- `lastlayerindex` (integer)
  Index of the highest layer where to start back-propagating LRP scores. Possible choice: any non-negative value for an explicit layer index, or -1 for auto-detecting the lowest oftmax layer or -2 for auto-detecting the highest inner product layer.
- `firstlayerindex` (integer)
  Possible choice: any non-negative value for an explicit layer index.
- `numclasses` (integer)
  Number of classes of the classification problem.
- `baseimgsize` (integer)
  Size of the largest side of the image after resizing, should be larger than `netinwid` and `netinhei`.
- `standalone_outpath` (string)
  Path where to output the heatmap files.

- `standalone_rootpath` (string)
  should be the name of a subdirectory in the full path name of the input image files. The part of the path, starting at this string is appended to `<standalone_outpath>`. Exits with error if the full path of the input image file does not contain this string. In particular, for the outputs written into `standalone_outpath`, the path structure is preserved starting with `standalone_rootpath`. This is helpful when two input images have identical filenames but are residing in different subdirectories. Example:
  - `standalone_outpath` is `/home/user99/results/03102015`
  - Input file is `/home/user99/data/data25092015/cats/cat03.jpg`
  - `standalone_rootpath` is `data25092015`

  Then, outfiles are written as e.g. `/home/user99/results/03102015/data25092015/cats/cat03.jpg_rawhm.txt`
- `relpropformulatype` (integer)
  Set to 0 for the ϵ-type formula, 2 for the α-β-type formula, 11 or 99 for Sensitivity Analysis and 26 for Deconvolution, 54 for ϵ+flat, 56 for $\epsilon + w^2$, 58 for $\alpha - \beta$+flat and 60 for $\alpha - \beta + w^2$.
- `auxiliaryvariable_maxlayerindexforflatdistinconv` (integer)
  Defines the layer index below (and including) for which — for the `relpropformulatype`s 54 to 60 — either flat or w^2 decomposition is applied instead of either ϵ or $\alpha - \beta$.
- `epsstab` (float)
  Value of ϵ for the ϵ-type formula. Must be non-negative.
- `alphabeta_beta` (float)
  Value of β for the α-β-type formula. Must be non-negative.

6 Contact and Support

For Answers to questions, help and support, please contact:

Sebastian Lapuschkin	`sebastian.lapuschkin@hhi.fraunhofer.de`
Alexander Binder	`alexander_binder@sutd.edu.sg`
Wojciech Samek	`wojciech.samek@hhi.fraunhofer.de`

References

[1] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLOS ONE*, 10(7):e0130140, 2015.

[2] Sebastian Bach, Alexander Binder, Klaus-Robert Müller, and Wojciech Samek. Controlling explanatory heatmap resolution and semantics via decomposition depth. In *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, pages 2271–2275, 2016.

[3] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[5] Maximilian Kohlbrenner, Alexander Bauer, Shinichi Nakajima, Alexander Binder, Wojciech Samek, and Sebastian Lapuschkin. Towards best practice in explaining neural network decisions with LRP. *CoRR*, abs/1910.09840, 2019.

[6] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5), 2017.

[7] Sebastian Lapuschkin, Alexander Binder, Gregoire Montavon, Klaus-Robert Muller, and Wojciech Samek. Analyzing classifiers: fisher vectors and deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2912–2920, 2016.

[8] Yann LeCun and Corinna Cortes. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1998.

[9] Grégoire Montavon, Sebastian Bach, Alexander Binder, Wojciech Samek, and Klaus-Robert Müller. Explaining nonlinear classification decisions with deep taylor decomposition. *Pattern Recognition*, 2016.

[10] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.

[11] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. 2015.

[12] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *ECCV*, pages 818–833, 2014.