Tutorial: Implementing Deep Taylor Decomposition / LRP

first version: Jul 14, 2016
last update: Jul 5, 2017

This tutorial explains how to efficiently implement layer-wise relevance propagation (LRP). We mainly focus on the version derived from the deep Taylor decomposition method. We explain using practical examples how these rules can be implemented efficiently, and how one can reuse existing neural network code for that purpose.

Note: If you are instead looking for ready to use software, have a look at the software section of this website. If you want to try relevance propagation without installing software, check our interactive demos.

1. Prerequisites

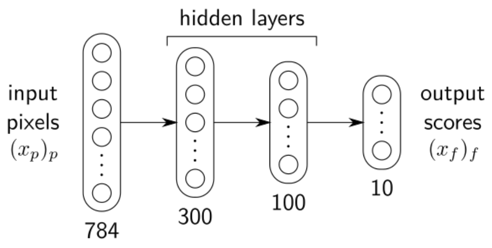
For this tutorial, you need Python, Numpy and PIL installed on your computer. Then, you should add to your working directory the files `modules.py` and `utils.py` containing a minimal implementation of neural network layers and methods to access and visualize the data.

```
import numpy,copy
import modules,utils
```

You also need a copy of the MNIST dataset that can be downloaded here. In this tutorial, we will use two trained neural networks: a fully connected neural network (`mlp.tar.gz`) and a convolutional neural network (`cnn.tar.gz`). You are also encouraged to read beforehand the introduction to the deep Taylor decomposition in order to better understand the underlying motivations.

2. Relevance Propagation for Fully-Connected Networks

We first consider a fully connected neural network with two hidden layers of size 300 and 100 and ReLU activations on both hidden layers. The network annotated with relevant variables is illustrated in the diagram below.



The weight and bias parameters of each layer of the network are provided in the archive `mlp.tar.gz`. The network was trained for 100 epochs using using data augmentation and reaches 1.05% test error on the MNIST test set. The neural network associated to these parameters can be instantiated as follows:

```
nn = modules.Network([
    modules.Linear('mlp/l1'),modules.ReLU(),
    modules.Linear('mlp/l2'),modules.ReLU(),
    modules.Linear('mlp/l3'),
])
```

Note that the neural network has been stripped from its trailing softmax layer used for training. Therefore, the output of the network corresponds to unnormalized class log-probabilities. This is the quantity that we will seek to explain in terms of input pixels.

2.1. Implementing the propagation procedure

To implement the backward propagation procedure necessary for relevance propagation, we will extend each class of modules.py, and create specialized methods for propagation.

We first implement the global propagation procedure from the top to the bottom of the network. The class `Network` defined below extends the original neural network class and adds a method "`relprop`" that performs such propagation.

```
class Network(modules.Network):
    def relprop(self,R):
        for l in self.layers[::-1]: R = l.relprop(R)
        return R
```

The method iteratively visits layers of the network in reverse order, call the propagation function of each individual layers, and feed the output of each called function to the function of the previous layer. The code is similar to the one used to implement gradient propagation.

Relevance propagation operates at a higher level of granularity than standard gradient backpropagation. In particular, it considers as a layer the pairing of a linear and ReLU layer. Instead gradient propagation implementations typically treat them as two separate layers. One option would be to change the neural network implementation by merging these two layers into one. Alternatively, we can simply propagate the signal through the ReLU layer without transformation and implement the full propagation rule in the linear layer just below.

```
class ReLU(modules.ReLU):
    def relprop(self,R): return R
```

The choice of propagation rule for a linear-detection layer depends on its input domain. The z^+-rule applies to layers with positive input domain (e.g. hidden layers receiving as input the ReLU features of a previous layer). The $z^\mathcal{B}$-rule, applies to box-constrained domains (e.g. input layer receiving as input pixel values).

Implementing the z^+-rule

The z$^+$-rule is defined by the equation:

$$R_i = \sum_j \frac{x_i w_{ij}^+}{\sum_i x_i w_{ij}^+} R_j$$

We would like to compute it using vector operations. We note that the rule can be rewritten as the sequence of computations

$$\forall_j : z_j = \textstyle\sum_i x_i w_{ij}^+ \qquad \forall_j : s_j = R_j \cdot z_j^{-1} \qquad \forall_i : c_i = \textstyle\sum_j w_{ij}^+ s_j \qquad \forall_i : R_i = x_i \cdot c_i,$$

that is, a succession of matrix-vector and element-wise products. The following class extension of the linear layer implements this computation.

```
class NextLinear(modules.Linear):
    def relprop(self,R):
        V = numpy.maximum(0,self.W)
        Z = numpy.dot(self.X,V)+1e-9; S = R/Z
        C = numpy.dot(S,V.T);         R = self.X*C
        return R
```

We call this class `NextLinear` to emphasize that the associated propagation method does not apply to the first layer but to the layers following it. Note that we have added a small increment to the variable `Z` to force the behavior `0/0=0` when computing `S`. This special case occurs when none of the inputs of this layer contributes positively to the neuron activation.

Implementing the $z^\mathcal{B}$-rule

For layers whose input activations $(x_i)_i$ belong to some closed interval $l_i \leq x_i \leq h_i$, the $z^\mathcal{B}$-rule is defined as follows:

$$R_i = \sum_j \frac{x_i w_{ij} - l_i w_{ij}^+ - h_i w_{ij}^-}{\sum_i x_i w_{ij} - l_i w_{ij}^+ - h_i w_{ij}^-} R_j$$

Like for the z$^+$-rule, the z$^\mathcal{B}$-rule can be implemented in terms of matrix-vector and element-wise operations. We create a class extension of `modules.Linear` and call the specialized class `FirstLinear` to emphasize that it applies only to the first layer of the network.

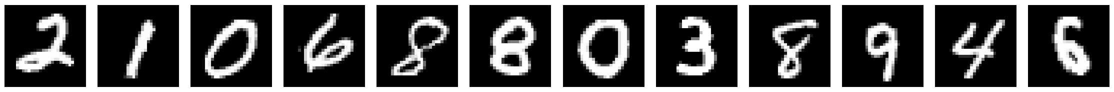
```
class FirstLinear(modules.Linear):
    def relprop(self,R):
        W,V,U = self.W,numpy.maximum(0,self.W),numpy.minimum(0,self.W)
        X,L,H = self.X,self.X*0+utils.lowest,self.X*0+utils.highest

        Z = numpy.dot(X,W)-numpy.dot(L,V)-numpy.dot(H,U)+1e-9; S = R/Z
        R = X*numpy.dot(S,W.T)-L*numpy.dot(S,V.T)-H*numpy.dot(S,U.T)
        return R
```

2.2. Application to MNIST data

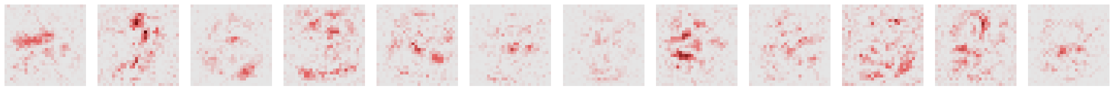
Now that all layers used by the fully-connected neural network have been extended with a relevance propagation method, the implementation can now be tested on real data. The following code randomly samples 12 MNIST handwritten digits and displays them.

```
X,T = utils.getMNISTsample(N=12,path='mnist/',seed=1234)
utils.visualize(X,utils.graymap,'data.png')
```



Each digit is an image of 28x28 pixels and can also be represented as a 784-dimensional vector. Pixel values are scaled between -1 and 1. Images are stored in the array `X` of shape `(N,784)`, and target classes are stored in the array `T` of shape `(N,10)` using one-hot encoding. As a baseline, we first perform a sensitivity analysis of the neural network prediction for the target class:

```
Y = nn.forward(X)
S = nn.gradprop(T)**2
utils.visualize(S,utils.heatmap,'mlp-sensitivity.png')
```



The sensitivity scores do not really explain why an image has been predicted in a certain way, but rather to which direction in the pixel space the output is most sensitive to. See the introduction for a discussion of the difference between sensitivity analysis and other decomposition techniques.

To perform relevance propagation, we first need to reinstantiate the neural network using the specialized classes that we have implemented:

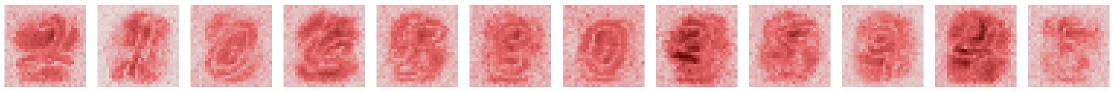
```
nn = Network([
    FirstLinear('mlp/l1'),ReLU(),
    NextLinear('mlp/l2'),ReLU(),
    NextLinear('mlp/l3'),ReLU(),
])
```

Note that we have added a trailing ReLU layer, which was not there in the original network. It serves to complete the final linear/detection layer. From this modification, it should be clear that one can only explain positive-valued predictions. The global relevance propagation procedure consists of following three steps:

1. Run a feed-forward pass on the network to collect neuron activations at each layer.
2. Keep only the part of the neural network output that corresponds to the correct class.
3. Call the `relprop` method to backpropagate the neural network output.

The following code applies this sequence of operations and visualizes the resulting heatmaps.

```
Y = nn.forward(X)
D = nn.relprop(Y*T)
utils.visualize(D,utils.heatmap,'mlp-deeptaylor.png')
```

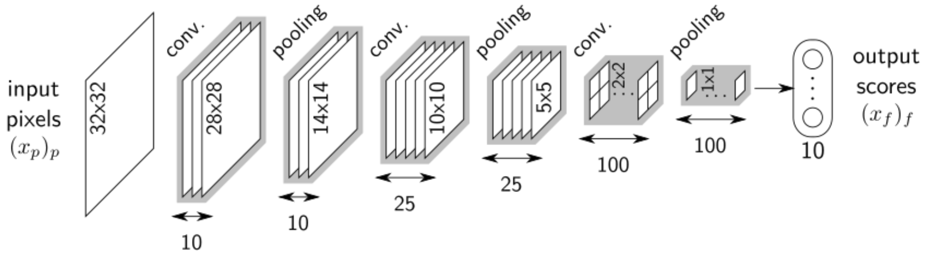


The heatmaps produced by relevance propagation are fuller than those obtained by sensitivity analysis. In particular, heat is spread more exhaustively on the contour of the digit. Nevertheless, the quality of the explanation is still limited by modeling artefacts of the neural network. In the next section, we implement relevance propagation for a convolutional neural network where modeling artefacts are less pronounced, and where better explanations of class membership can be obtained.

3. Relevance Propagation for Convolutional Neural Networks

In practice, convolutional and pooling layers are essential to produce high quality models when the input is a two-dimensional image or some other type of sequential data (e.g. speech, text, video, etc.). Convolution layers can be tedious to implement, however, we can observe that one can reuse the readily available forward activation and gradient propagation methods of these layers for the purpose of implementing deep Taylor propagation rules.

We consider a Lenet-5 type network, where sigmoid nonlinearities are replaced by ReLU units, with a few more feature maps in early layers, and where we add one top-level pooling layer to further increase global translation invariance. The convolutional neural network is depicted below.



Parameters of the trained model are available in the archive cnn.tar.gz. The neural network was trained for 10 epochs without data augmentation and reaches 0.76% test error on MNIST. The convolutional neural network can be instantiated from these parameters using the following code:

```
cnn = modules.Network([
    modules.Convolution('cnn/c1-5x5x1x10'),modules.ReLU(),modules.Pooling(),
    modules.Convolution('cnn/c2-5x5x10x25'),modules.ReLU(),modules.Pooling(),
    modules.Convolution('cnn/c3-4x4x25x100'),modules.ReLU(),modules.Pooling(),
    modules.Convolution('cnn/c4-1x1x100x10'),
])
```

Like for the fully connected network, we have removed the trailing softmax layer, and therefore, the quantities that will be explained are the unnormalized class log-probabilities.

3.1. Implementing propagation rules for convolution and pooling

We would like to extend the convolution and pooling layers provided in the file modules.py with the deep Taylor propagation rules. For this, we will make use of the fact that these rules can also be expressed in terms by feedforward activation and gradient propagation (both of them performing efficient multiplications by the weight matrix). The multiplication-type operations of both the z^+-rule and the $z^\mathcal{B}$-rule can be expressed as forward computation and gradient propagation of cloned layers with modified parameters.

```
class NextConvolution(modules.Convolution):
    def relprop(self,R):
        pself = copy.deepcopy(self); pself.B *= 0; pself.W = numpy.maximum(0,pself.W)

        Z = pself.forward(self.X)+1e-9; S = R/Z
        C = pself.gradprop(S);          R = self.X*C
        return R
```

Compare with the same rule implemented for the simple linear layer of the fully-connected network and observe how similar the structure of the computation is between the two. For the $z^\mathcal{B}$-rule the original layer has to be cloned three times with weights and biases modified in three different ways.

```
class FirstConvolution(modules.Convolution):
    def relprop(self,R):
        iself = copy.deepcopy(self); iself.B *= 0
        nself = copy.deepcopy(self); nself.B *= 0; nself.W = numpy.minimum(0,nself.W)
        pself = copy.deepcopy(self); pself.B *= 0; pself.W = numpy.maximum(0,pself.W)
        X,L,H = self.X,self.X*0+utils.lowest,self.X*0+utils.highest

        Z = iself.forward(X)-pself.forward(L)-nself.forward(H)+1e-9; S = R/Z
        R = X*iself.gradprop(S)-L*pself.gradprop(S)-H*nself.gradprop(S)
        return R
```

Again, the rule is very similar structurally to the one of the simple linear layer. Finally, the proportional redistribution occuring in the sum-pooling layer given by the equation

$$R_i = \sum_j \frac{x_i \delta_{ij}}{\sum_i x_i \delta_{ij}} R_j$$

where δ_{ij} indicates whether the neuron x_i is in the pool of x_j. The latter can also be implemented in a similar way to the convolution layers, by accessing to the forward and gradient propagation functions:

```
class Pooling(modules.Pooling):
    def relprop(self,R):
        Z = (self.forward(self.X)+1e-9); S = R / Z
        C = self.gradprop(S);            R = self.X*C
        return R
```

3.2. Application to MNIST data

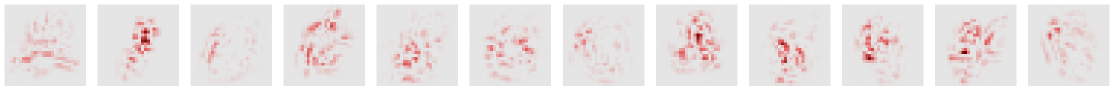
To serve as input to the convolutional neural network, the MNIST images need to be reshaped and padded to shape 32x32x1. A similar reshaping operation is also applied to the labels.

```
X,T = utils.getMNISTsample(N=12,path='mnist/',seed=1234)

padding = ((0,0),(2,2),(2,2),(0,0))
X = numpy.pad(X.reshape([12,28,28,1]),padding,'constant',constant_values=(utils.lowest,))
T = T.reshape([12,1,1,10])
```

As a basis for comparison, we perform sensitivity analysis, as we did for the fully-connected case.

```
Y = cnn.forward(X)
S = cnn.gradprop(T)**2
utils.visualize(S[:,2:-2,2:-2],utils.heatmap,'cnn-sensitivity.png')
```



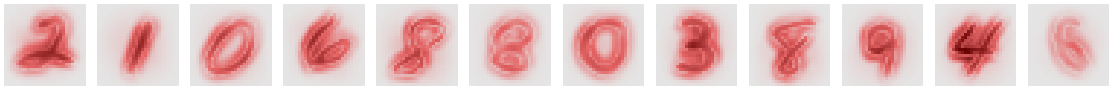
The result is similar to the sensitivity analysis of the previous network, in particular, heatmaps are sparse and reflect local variations of the neural network function rather than producing a global explanation.

To apply relevance propagation, we need to reinstantiate the neural network with the newly defined specialized classes. We also add a top-level ReLU layer in order to complete the last linear-detection layer.

```
cnn = Network([
    FirstConvolution('cnn/c1-5x5x1x10'),ReLU(),Pooling(),
    NextConvolution('cnn/c2-5x5x10x25'),ReLU(),Pooling(),
    NextConvolution('cnn/c3-4x4x25x100'),ReLU(),Pooling(),
    NextConvolution('cnn/c4-1x1x100x10'),ReLU(),
])
```

We are now able to explain the predictions of this convolutional neural network and to visualize these explanations as heatmaps.

```
Y = cnn.forward(X)
D = cnn.relprop(Y*T)
utils.visualize(D[:,2:-2,2:-2],utils.heatmap,'cnn-deeptaylor.png')
```



Heatmaps are more complete than those obtained with sensitivity analysis, and more consistently follow the contour of the digits. The heatmaps are also of better quality than those obtained by the fully-connected network, in particular, they are more focused on the contour of the digit and ignore the non-digit area of the image.

3.3. Increase the Focus of Explanations with the LRP-$\alpha\beta$ rule.

In order to produce more focused heatmaps, it was shown to be useful to inject some negative relevance into the redistribution process. This can be achieved by the flexible LRP-$\alpha\beta$ rule of which the z^+-rule used above is the special case with $\alpha = 1, \beta = 0$. The propagation rule is defined by the equation:

$$R_i = \sum_j \Big(\alpha \frac{x_i w_{ij}^+}{\sum_i x_i w_{ij}^+} - \beta \frac{x_i w_{ij}^-}{\sum_i x_i w_{ij}^-}\Big) R_j$$

and has the effect of injecting a controlled amount of "counter-relevance" to the neurons that have an inhibitory effect on the higher-layer neurons. A possible implementation of this rule is shown below for the convolution layer.

```
class NextConvolutionAlphaBeta(modules.Convolution,object):

    def __init__(self,name,alpha):
        super(self.__class__, self).__init__(name)
        self.alpha = alpha
        self.beta  = alpha-1

    def relprop(self,R):
        pself = copy.deepcopy(self); pself.B *= 0; pself.W = numpy.maximum( 1e-9,pself.W)
        nself = copy.deepcopy(self); nself.B *= 0; nself.W = numpy.minimum(-1e-9,nself.W)

        X = self.X+1e-9
        ZA = pself.forward(X); SA =  self.alpha*R/ZA
        ZB = nself.forward(X); SB = -self.beta *R/ZB
        R = X*(pself.gradprop(SA)+nself.gradprop(SB))
        return R
```

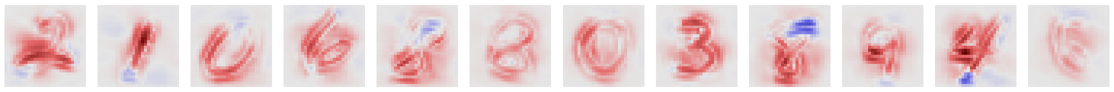
Note that the stabilizing terms had to be placed at different locations in order to correctly handle the special case where all weights are of the same sign, while at the same time ensuring relevance conservation between layers. In the absence of negative weights, this implementation redistributes counter-relevance uniformly to all input neurons.

Once the rule has been implemented, a new explainable convolutional network can be built. Here we use the parameter $\alpha = 2$ that was shown to work well in practice.

```
cnn = Network([
    FirstConvolution('cnn/c1-5x5x1x10'),ReLU(),Pooling(),
    NextConvolutionAlphaBeta('cnn/c2-5x5x10x25',2.0),ReLU(),Pooling(),
    NextConvolutionAlphaBeta('cnn/c3-4x4x25x100',2.0),ReLU(),Pooling(),
    NextConvolutionAlphaBeta('cnn/c4-1x1x100x10',2.0),ReLU(),
])
```

Finally, the predictions and explanations are computed by running a forward and backward pass in this network.

```
Y = cnn.forward(X)
D = cnn.relprop(Y*T)
utils.visualize(D[:,2:-2,2:-2],utils.heatmap,'cnn-alphabeta.png')
```



Unlike the standard deep Taylor LRP heatmaps, the heatmaps obtained with LRP-$\alpha\beta$ typically contain some negative relevance (shown in blue). This negative relevance points potentially at local irregularities of the digit such as the broken loop in the last digit "8". The heatmaps also tend to be sparser.