Exercises for the course

Machine Learning 2

Summer semester 2021

Abteilung Maschinelles Lernen
Institut für Softwaretechnik und theoretische Informatik
Fakultät IV, Technische Universität Berlin
Prof. Dr. Klaus-Robert Müller
Email: klaus-robert.mueller@tu-berlin.de

Exercise Sheet 11

Exercise 1: Activation Maximization (20 P)

Consider the linear model $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$ mapping some input $\boldsymbol{x}$ to an output $f(\boldsymbol{x})$. We would like to interpret the function f by building a prototype $\boldsymbol{x}^\star$ in the input domain which produces a large value f. Activation maximization produces such interpretation by optimizing

$$\max_{\boldsymbol{x}} \big[f(\boldsymbol{x}) - \Omega(\boldsymbol{x}) \big].$$

(a) *Find* the prototype $\boldsymbol{x}^\star$ obtained by activation maximization subject to the penalty $\Omega(\boldsymbol{x}) = \lambda \|\boldsymbol{x}\|^2$.

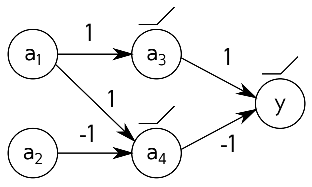
(b) *Find* the prototype $\boldsymbol{x}^\star$ obtained by activation maximization subject to the penalty $\Omega(\boldsymbol{x}) = -\log p(\boldsymbol{x})$ with $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ where $\boldsymbol{\mu}$ and Σ are the mean and covariance.

(c) *Find* the prototype $\boldsymbol{x}^\star$ obtained when the data is generated as (i) $\boldsymbol{z} \sim \mathcal{N}(0, I)$ and (ii) $\boldsymbol{x} = A\boldsymbol{z} + \boldsymbol{c}$, with A and $\boldsymbol{c}$ the parameters of the generator. Here, we optimize f w.r.t. the code $\boldsymbol{z}$ subject to the penalty $\Omega(\boldsymbol{z}) = \lambda \|\boldsymbol{z}\|^2$.

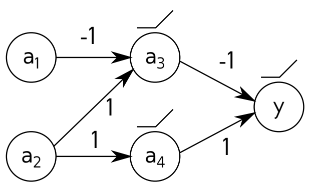
Exercise 2: Layer-Wise Relevance Propagation (30 P)

We would like to test the dependence of layer-wise relevance propagation (LRP) on the structure of the neural network. For this, we consider the function $y = \min(a_1, a_2)$, where $a_1, a_2 \in \mathbb{R}^+$ are the input activations. This function can be implemented as a ReLU network in multiple ways. Two examples are given below.

(i)



(ii)



(a) *Show* that these two networks implement the 'min' function on the relevant domain.

(b) We consider the LRP-γ propagation rule:

$$R_j = \sum_k \frac{a_j \cdot (w_{jk} + \gamma w_{jk}^+)}{\sum_j a_j \cdot (w_{jk} + \gamma w_{jk}^+)} R_k$$

where $()^+$ denotes the positive part. For each network, *give* for the case $a_1 = a_2$ an analytic solution for the scores R_1 obtained by application this propagation rule at each layer. More specifically, express R_1 as a function of the input activations.

Exercise 3: Neuralization (20 P)

Consider the one-class SVM that predicts for every new data point $\boldsymbol{x}$ the 'inlierness' score:

$$f(\boldsymbol{x}) = \sum_{i=1}^{M} \alpha_i k(\boldsymbol{x}, \boldsymbol{u}_i)$$

where $(\boldsymbol{u}_i)_{i=1}^M$ is the collection of support vectors, and $\alpha_i > 0$ are their weightings. We use the Gaussian kernel $k(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\gamma \|\boldsymbol{x} - \boldsymbol{x}'\|^2)$.

Because we are typically interested in the degree of anomaly of a particular data point, we can also define the score $o(\boldsymbol{x}) = -\frac{1}{\gamma} \log f(\boldsymbol{x})$ which grows with the degree of anomaly of the data point.

(a) Show that the outlier score $o(\boldsymbol{x})$ can be rewritten as a two-layer neural network:

$$\begin{aligned} h_i &= \|\boldsymbol{x} - \boldsymbol{u}_i\|^2 - \gamma^{-1} \log \alpha_i && \text{(layer 1)} \\ o(\boldsymbol{x}) &= -\tfrac{1}{\gamma} \log \textstyle\sum_{i=1}^{M} \exp(-\gamma h_i) && \text{(layer 2)} \end{aligned}$$

(b) Show that the layer 2 converges to a min-pooling (i.e. $o(\boldsymbol{x}) = \min_{i=1}^{N}\{h_i\}$) in the limit of $\gamma \to \infty$.

Exercise 4: Programming (30 P)

Download the programming files on ISIS and follow the instructions.

Explaining the Predictions of the VGG-16 Network

In this homework, we would like to implement different explanation methods on the VGG-16 network used for image classification. As a test example, we take some image of a castle

and would like to explain why the VGG:16 neuron `castle` activates for this image. The code below loads the image and normalizes it, loads the model, and identifies the output neuron corresponding to the class `castle` .

In [1]:
```
import numpy
import cv2
import torch
import torch.nn
import utils

# Load image
img = numpy.array(cv2.imread('castle.jpg'))[...,::-1]/255.0
mean = torch.Tensor([0.485, 0.456, 0.406]).reshape(1,-1,1,1)
std  = torch.Tensor([0.229, 0.224, 0.225]).reshape(1,-1,1,1)
X = (torch.FloatTensor(img[numpy.newaxis].transpose([0,3,1,2])*1) - mean) / std

# Load VGG-16 network
import torchvision
model = torchvision.models.vgg16(pretrained=True); model.eval();

# Identify neuron "castle"
cl = 483
```

Gradient $\times$ Input (15 P)

A simple method for explanation is Gradient $\times$ Input. Denoting f the function corresponding to the activation of the desired output neuron, and $\boldsymbol{x}$ the data point for which we would like to explain the prediction, the method assigns feature relevance using the formula

$$R_i = [\nabla f(\boldsymbol{x})]_i \cdot x_i$$

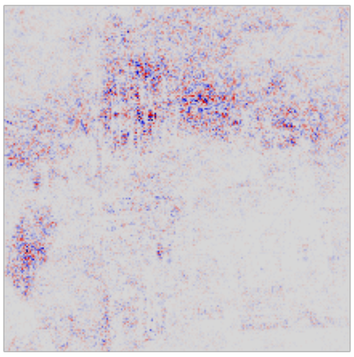
for all $i = 1 \dots d$. When the neural network is piecewise linear and positively homogeneous, the method delivers the same result as one would get with a Taylor expansion at reference point $\widetilde{\boldsymbol{x}} = \varepsilon \cdot \boldsymbol{x}$ with ε almost zero.

Task:

- **Implement Gradient $\times$ Input, i.e. write a function that produces a tensor of same dimensions as the input data and that contains the scores $(R_i)_i$, test it on the given input image, and visualize the result using the function `utils.visualize`.**

In [2]:
```
%matplotlib inline

# ------------------------------------
# TODO: Replace by your code
# ------------------------------------
import solution
solution.gradinput(model,X,cl)
# ------------------------------------
```



We observe that the explanation is noisy and has a large amount of positive and negative evidence. To produce a more robust explanation, we would like to use instead LRP, and implement it using the trick described in the slides. The trick consists of detaching certain terms from the differentiation graph, so that the explanation can obtained by performing Gradient $\times$ Input with the modified gradient.

LRP rules are typically set different at each layer. In VGG-16, we distinguish the following three types of layers of parameters:

- **First convolution:** This convolution receives pixels as input, and it requires a special rewrite rule that we given below.
- **Next convolutions:** All remaining convolutions receives activations as input, and we can treat them using LRP-γ.
- **Dense layers:** For these layers, we want to let the standard gradient signal flow, so we can simply leave these layers intact.

LRP in First Convolution

This convolution implements the propagation rule:

$$R_i = \sum_j \frac{x_i w_{ij} - l_i w_{ij}^+ - h_i w_{ij}^-}{\sum_{0,i} x_i w_{ij} - l_i w_{ij}^+ - h_i w_{ij}^-} R_j$$

where $(\cdot)^+$ and $(\cdot)^-$ are shortcut notations for $\max(0,\cdot)$ and $\min(0,\cdot)$, x_i the value of pixel i, and where l_i and h_i denote lower and upper-bounds on pixel values. To implement this rule using the proposed approach, we define the quantities $p_j = x_i w_{ij} - l_i w_{ij}^+ - h_i w_{ij}^-$ and $z_j = x_i w_{ij}$ and perform the reassignation:

$$z_j \leftarrow p_j \cdot [z_j / p_j]_{\text{cst.}}$$

This is done by the code below

In [3]:
```
class ConvPx(torch.nn.Module):
    def __init__(self, conv):
        torch.nn.Module.__init__(self)
        self.conv  = conv
        self.pconv = utils.newlayer(conv, lambda p: p.clamp(min=0))
        self.nconv = utils.newlayer(conv, lambda p: p.clamp(max=0))

    def forward(self, X):
        X, L, H = X
        z = self.conv.forward(X)
        zp = z - self.pconv.forward(L) - self.nconv.forward(H)
        return zp * (z / zp).data
```

LRP in Next Convolutions (15 P)

In the next convolutions, we would like to apply the LRP-γ rule given by:

$$R_j = \sum_k \frac{a_j(w_{jk} + \gamma w_{jk}^+)}{\sum_{0,j} a_j(w_{jk} + \gamma w_{jk}^+)} R_k$$

To implement this rule using the proposed approach, we define the quantities $p_k = a_j(w_{jk} + \gamma w_{jk}^+)$ and $z_k = a_j w_{jk}$ and perform the reassignation:

$$z_k \leftarrow p_k \cdot [z_k / p_k]_{\text{cst.}}$$

Task:

- **Inspired by the code for the first convolution, implement a class for the next convolutions that is equiped to perform the LRP-γ propagation when calling the gradient.**

```
In [4]: class Conv(torch.nn.Module):

            def __init__(self, conv, gamma):
                # ------------------------------------
                # TODO: Replace by your code
                # ------------------------------------
                import solution
                solution.convinit(self,conv,gamma)
                # ------------------------------------

            def forward(self, X):
                # ------------------------------------
                # TODO: Replace by your code
                # ------------------------------------
                import solution
                return solution.convforward(self,X)
                # ------------------------------------
```

Now, we can create the LRP-enabled model by replacing the convolution layers with their modified versions.

```
In [5]: lrpmodel = torchvision.models.vgg16(pretrained=True); model.eval()
        features = lrpmodel._modules['features']
        for i in [0]:          features[i] = ConvPx(features[i])
        for i in [2]:          features[i] = Conv(features[i],10**(-0.0))
        for i in [5,7]:        features[i] = Conv(features[i],10**(-0.5))
        for i in [10,12,14]: features[i] = Conv(features[i],10**(-1.0))
        for i in [17,19,21]: features[i] = Conv(features[i],10**(-1.5))
        for i in [24,26,28]: features[i] = Conv(features[i],10**(-2.0))
```

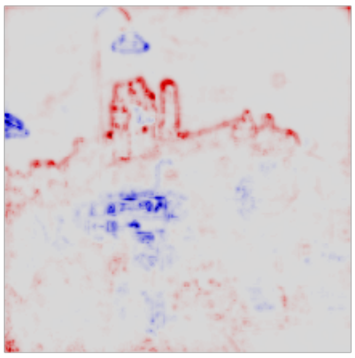
Note that for the layer 0 and for the subsequent layers, we have used two different classes. Also, the parameter γ is set high in the lower layers and goes increasingly closer to zero as we reach the top convolutions.

We then proceed like Gradient $\times$ Input, except for the fact that in the modified first convolution the inputs and gradients not only comprise the pixels x but also the lower and upper bounds l and h. The code below implements this extended Gradient $\times$ Input and visualizes the resulting explanation.

In [6]:
```
# Prepare data and lower/upper bounds
X.grad = None
L = (X*0+((0-mean)/std).view(1,3,1,1)).data
H = (X*0+((1-mean)/std).view(1,3,1,1)).data
X.requires_grad_(True)
L.requires_grad_(True)
H.requires_grad_(True)

# Apply the modfied gradient x input
lrpmodel.forward((X,L,H))[0,cl].backward()
R = (X*X.grad+L*L.grad+H*H.grad)

%matplotlib inline
utils.visualize(R)
```



We observe that most of the noise has disappeared, and we can clearly see in red which patterns in the data the neural network has used to predict 'castle'. We also see in blue which patterns contribute negatively to that class, e.g. the corner of the roof and the traffic sign.

Exercise Sheet 11

1 Activation Maximization

a) $\frac{\partial}{\partial x} \max_x \; w^T x + b - \lambda \|x\|^2 \overset{!}{=} 0$

$\Leftrightarrow \; w^T - 2\lambda x \overset{!}{=} 0 \qquad \Leftrightarrow \; x^* = \frac{w^T}{2\lambda}$

b) $\max_x \; w^T x + b + \log(p(x))$

$= \max_x \; w^T x + b + \log\left(\frac{\exp(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu))}{(2\pi)^{\frac{d}{2}} |\Sigma|}\right)$

$= \max_x \; w^T x + b + (\log(\exp(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)) - \log((2\pi)^{\frac{d}{2}} |\Sigma|$

$\Rightarrow \frac{\partial}{\partial x} w^T x - \frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu) \overset{!}{=} 0$

$\Leftrightarrow \; w^T - \Sigma^{-1}(x-\mu) = 0$

$\Leftrightarrow \; w^T - \Sigma^{-1} x + \Sigma^{-1}\mu = 0$

$\Leftrightarrow \; x = \Sigma(\Sigma^{-1}\mu + w^T)$

$\Leftrightarrow \; x^* = \mu + \Sigma w^T$

c) $\max_z \; w^T(Az + c) + b - \lambda \|z\|^2$

$\Rightarrow \frac{\partial}{\partial z} w^T(Az + c) + b - \lambda \|z\|^2 \overset{!}{=} 0$

$\Leftrightarrow \; w^T A - 2\lambda z = 0$

$\Leftrightarrow \; z = \frac{w^T A}{2\lambda} \qquad | \; x = Az + c \Leftrightarrow z = \frac{x-c}{A}$

$\Leftrightarrow \; \frac{x-c}{A} = \frac{w^T A}{2\lambda}$

$\Leftrightarrow \; x = \frac{w^T A A}{2\lambda} + c$

2 Layer-Wise Relevance Propagation

a) $$y = \underbrace{a_3}_{a_1} - \underbrace{a_4}_{\max(0,\, a_1 - a_2)}$$

ReLU has no effect
bc. a_1 is positive

if $a_1 \geq a_2$: $y = a_1 - (a_1 - a_2) = a_2$

if $a_1 \leq a_2$: $y = a_1 - 0 = a_1$

b) i) $R_4 = a$ ii) $R_4 = a$

$R_3 = a$ $R_3 = 0$

$R_1 = a$ $R_1 = 0$

3 Neuralisation

a) $$o(x) = -\frac{1}{\gamma} \log\left(\sum_{i=1}^{M} \alpha_i e^{-\gamma \|x - \mu_i\|^2}\right)$$

$$= -\frac{1}{\gamma} \log\left(\sum_{i=1}^{M} e^{-\gamma \|x - \mu_i\|^2 - \frac{1}{\gamma} \log(\alpha_i)}\right)$$

b) $\min_{i=1}^{M} \{h_i\} = m$

$$o(x) - m = -\frac{1}{\gamma} \log\left(\sum_{i=1}^{M} e^{-\gamma h_i}\right) + \frac{1}{\gamma} \log e^{-\gamma m}$$

$$= -\frac{1}{\gamma} \log\left(\sum_{i=1}^{M} e^{-\gamma h_i} e^{\gamma m}\right)$$

$$= -\frac{1}{\gamma} \log\left(\sum_{i=1}^{M} e^{-\gamma (h_i - m)}\right)$$

$$= -\frac{1}{\gamma} \log e^{-\gamma \cdot 0} + \sum_i e^{-\gamma (h_i - m)}$$

$$= -\frac{1}{\gamma} \log\left(1 + \sum_i e^{-\gamma (h_i - m)}\right)$$

$o(x) - m = 0$ for $\gamma \to \infty$

$\leftrightarrow$ $o(x) = m$ $\rightarrow$ min-pooling