Implementing LRP in PyTorch

Alexander Binder

August 5, 2020

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Credits

Code inspired by Philipp Seegerer (TU Berlin).
With important ideas from Gregoire Montavon (TU Berlin), Philipp Seegerer and Sebastian Lapuschkin (Fraunhofer HHI).

Structure of this Talk

- recap on PyTorch modules and LRP rules
- canonization for resnets: shortcuts and Conv→BatchNorm chains
- understanding LRP: LRP flows same as gradient
- implementing custom backward passes I: Gregoires idea: LRP-rule homogeneity and `backward()` of custom functions
- implementing custom backward passes II: PyTorch autograd functions and code examples
- canonization for densenet-121: BatchNorm→ReLU→Conv chains (+ cleaning up calls to *torch.functional*.$*$)
- recurrent nets: LRP for LSTM
- hands-on part I: you can run code from `https://github.com/AlexBinder/LRP_Pytorch_Resnets_Densenet`
- hands-on part II: you will implement Guided Backprop and LRP-γ

Recap on Pytorch internals

Recap: PyTorch Modules

e.g. `nn.Conv2d`, `nn.Linear` – a class

see https://pytorch.org/docs/master/generated/torch.nn.Module.html#torch.nn.Module

- takes a feature map as input, compute next feature map (via `.forward(self,*args)`)
- contain trainable parameters – `torch.nn.parameter.Parameter`
  - a tensor with *requires_grad* = *True*
  - meaning: to be used for updating its values during training,
  - will be saved in `state_dict` (also buffers)
  - has a its own iterators (e.g. `.named_parameters()`), will be given to the optimizer for updating its values
- has convenience methods
  - `.to(device)` to move all tensors to another device
  - iterators over modules,parameters, buffers inside

Recap on LRP rules

understanding properties of rules: LRP-ϵ

given: have computed already R_k as relevance of neuron output $z(k) = \sum_i w_{ik} x_i + b$,
LRP-ϵ:

$$R_{i \leftarrow k}(\mathbf{x}) = R_k M_{i \leftarrow k} = R_k M_{i \leftarrow k}(w_{ik}, x_i)$$

$$R_{i \leftarrow k}(\mathbf{x}) = R_k \left(\frac{w_{ik} x_i}{z + \epsilon \mathrm{sign}(z)} \right)$$

$$z = \sum_{i'} w_{ik'} x_{i'} + b$$

- may produce $R_{i \leftarrow k}$ with $|R_{i \leftarrow k}| \gg |R_k|$
- no control over relevance scale!
- ϵ - dampens redistribution differences
- $\epsilon \to \infty$ convergence to flat redistribution

understanding properties of rules: LRP-β

given: have computed already R_k as relevance of neuron output $z(k) = \sum_i w_{ik} x_i + b$,
LRP-β:

$$R_{i \leftarrow k}(\mathbf{x}) = R_k M_{i \leftarrow k} = R_k M_{i \leftarrow k}(w_{ik}, x_i)$$

$$R_{i \leftarrow k}(\mathbf{x}) = R_k \left((1+\beta) \frac{(w_{ik} x_i)_+}{\sum_{i'} (w_{i'k} x_{i'})_+ + b_+} - \beta \frac{(w_{ik} x_i)_-}{\sum_{i'} (w_{i'k} x_{i'})_- + b_-} \right)$$

- β controls ratio of negative to positive evidence.
- bounded relevance scale: $|R_{i \leftarrow k}| \leq (1+\beta)|R_k|$
- negative to positive evidence: $\frac{\beta}{1+\beta}$,
- negative to total evidence: $\frac{\beta}{1+2\beta} \overset{\beta \to \infty}{\rightarrow} 0.5$,
  It is fixed independent of network inputs(!).

which rule for which layer?

Name	Formula	layers
LRP-ϵ	$\sum_k R_k \left(\frac{x_i w_{ik}}{\sum_i x_i w_{ik} + b + \epsilon \mathrm{sign}(z)}\right)$	fully connected
LRP-$\beta = 0$	$\sum_k R_k \left(\frac{(x_i w_{ik})_+}{\sum_i (x_i w_{ik})_+ (b)_+}\right)$	conv
LRP-γ	$\sum_k R_k \left(\frac{\gamma(x_i w_{ik})_+ + (x_i w_{ik})}{\sum_i \gamma(x_i w_{ik})_+ + \gamma(b)_+ + \sum_i (x_i w_{ik}) + b}\right)$	conv
LRP-z_β	$\sum_k R_k \left(\frac{x_i w_{ik} - l_i(w_{ij})_+ + h_i(w_{ij})_-}{\sum_i x_i w_{ik} + b - l_i(w_{ij})_+ + h_i(w_{ij})_-}\right)$	first conv layer
LRP-w^2	$\sum_k R_k \frac{w_{ik}^2}{\sum_i w_{ik}^2}$	same 1. conv

Its a mere serving suggestion. Define a loss and measure the quality of your explanations and choose by that!

steps to add a LRP backward pass

- $\boxed{\text{canonization = functionally equivalent restructuring of the NN}}$
  - copy model from your trained model
  - treat special ops: replace some operations by explicit layers, e.g. the $+$ in $y = x + H(x)$ in residuals
  - always: fuse batchnorm into conv layers (two types of possible fusings)
- forward check of canonization – forward pass still equal to original model?
- backward impl – for each layer of interest.
  - see the math for gradient vs LRP: same flow, implementation via custom backward passes
  - implementation: structure and examples for layers
  - in canonization module: wrap each layer by wrapper with custom backward
- important other details (like biases)

step I: canonization resnet example

make model usable for custom backward explanations

- create a modified copy with parameters from trained source model
- technical issue: need to replace the $+$ in a residual connection $x + Conv_2(Conv_1(x))$ by an operator implementing $+$.
- LRP-issue: fuse Conv-BatchNorm chains into a Conv-Layer

The conv-BN fusion is due to an LRP-issue:

$\boxed{\text{LRP is not implementation invariant.}}$

step I: canonization resnet example

Why conv-bn-fusion? Adebayo et al: LRP fails the parameter randomization test and is not implementation invariant.

$\boxed{\text{Why LRP then at all and not Gradient/ Grad-CAM ?}}$

- Gradient estimates an often suboptimal measure:
  a single-pixel sensitivity instead of contributions which account for interactions between larger regions.
- Gradient: +high noise from gradient shattering in ReLU nets.
- For a **measurement-based comparison** of gradients against guided back prop in a medical context see eg. Eitel et.al. MICCAI 2019
  https://link.springer.com/chapter/10.1007/978-3-030-33850-3_1
  https://arxiv.org/abs/1909.08856
- for NLP: Poerner et al. ACL 2018,
  https://www.aclweb.org/anthology/P18-1032.pdf
- fail in parameter randomization test does not imply failure to explain current model at hand.

step I: canonization resnet example

Fail in Implementation-invariance can be managed.

Fail in measures u can?

step I: canonization resnet example

manual step!

- resnet: need to replace the $+$ in a residual connection $x + Conv(Conv(x))$ by an operator implementing $+$.
  Why we do not overload the backward passes for $+$ in general?

Check the code example for `copy_resnet_onlycopy_v2.py`

- create a `nn.Module`-derived class `sum_stacked2` ,
- create a derived bottleneck, basicblock, resnet classes (easy).
- replace the shortcut by `sum_stacked2`

step I: canonization resnet example

- fuse Conv-BatchNorm chains into a Conv-Layer. Resnet has the following chain: Conv $\rightarrow$ BN

$$
\begin{aligned}
\text{conv-layer: } y &= w_{conv} \cdot x + b_{conv,c} \\
\text{bn-layer: } z &= w_c\,(y - \mu_{bn})/s_c + bn_c, \qquad s_c = (\sigma_{bn,c} + \epsilon_{bn})^{0.5} \\
\text{bn} \rightarrow conv: z &= w_c/s_c\,(y - \mu_{bn}) + bn_c \\
&= (w_c/s_c)\,(w_{conv} \cdot x + b_{conv,c} - \mu_{bn}) + bn_c \\
&= \alpha_c \cdot x + \beta_c
\end{aligned}
$$

$$
\begin{aligned}
&\Rightarrow \alpha_c = (w_c/s_c) w_{conv} \\
&\Rightarrow \beta_c = (w_c/s_c)(b_{conv,c} - \mu_{bn,c}) + bn_c
\end{aligned}
$$

step I: canonization resnet example

Check the code example for $\boxed{\texttt{lrp\_general6.py}}$

$$\alpha_c = (w_c/s_c) w_{conv}$$
$$\beta_c = (w_c/s_c)(b_{conv,c} - \mu_{bn,c}) + bn_c$$

```
def bnafterconv_overwrite_intoconv(conv,bn):
    s = (bn.running_var+bn.eps)**.5
    w = bn.weight
    b = bn.bias
    m = bn.running_mean

    conv.weight = torch.nn.Parameter(conv.weight * (w / s).reshape(-1, 1, 1, 1))
    if conv.bias is None:
      conv.bias = torch.nn.Parameter((0 - m) * (w / s) + b)
    else:
      conv.bias = torch.nn.Parameter(( conv.bias - m) * (w / s) + b)
    return conv
```

the `.reshape(-1,1,1,1)` due to the structure of the conv-weight as: $weight[c_{out}, c_{in}, h, w]$, bn-weight: $\alpha[c_{out}]$ (+broadcasting)

step I: canonization resnet example

`copy_resnet_onlycopy_v2.py`

in derived class create routine for:

- copy layers with parameters from pretrained model + process all layers

```
def copyfromresnet(self,net, ...):
```

- conv-bn-fusion:
  - if detect conv-layer, stash it (next will be a BN!).
  - if detect bn, (1) fuse bn into stashed conv, (2) overwrite stashed conv in model, (3) reset BN stats, so that it is the identity
- which layers need to be copied from the trained model?
  - Conv2d, BatchNorm(reset), nn.Linear

step I: canonization resnet example

You got code for canonizing resnets.

`copy_resnet_onlycopy_v2.py`

```
def copyfromresnet(self,net, ...):
```

TODO:

- verify that forward passes of original and canonized model are matching!
- which layers need to be wrapped for backward pass?
  (next step, when LRP rules are implemented)
  - Conv2d, BatchNorm(reset), nn.Linear + ReLU, adaptiveavgpool, maxpool

steps: Implementing LRP backward pass

- step 1: see that LRP follows the same flow as backpropagation
  - revisit chain rule for gradient
  - see the insight: LRP follows the same flow as backpropagation
  - see how the gradient can be implement by a custom backward pass in pytorch
- step 2: the basic idea: implementing LRP messages $R_{d_1 \leftarrow d_2}$ via `torch.autograd.backward()` inside a custom backward

same flow: revisit chain rule for gradient (a)

suppose we have a layer $f(\cdot)$ in the middle of some neural net

$$x \in \mathbb{R}^1, v \in \mathbb{R}^{in}, y \in \mathbb{R}^{out}, o \in \mathbb{R}^1$$
$$v = v(x),\ \ y = y(v), o = o(y)$$
$$o = o(y(v(x)))$$

by chainrule, used in autograd, we have

$$\begin{aligned}\frac{do}{dx} &= \sum_{d_1=1}^{in} \frac{do}{dv[d_1]} \frac{dv[d_1]}{dx} \\ &= \sum_{d_1=1}^{in} \left(\sum_{d_2=1}^{out} \frac{do}{dy[d_2]} \frac{dy[d_2]}{dv[d_1]} \right) \frac{dv[d_1]}{dx} \\ \Rightarrow \frac{do}{dv[d_1]} &= \sum_{d_2=1}^{out} \frac{do}{dy[d_2]} \cdot \frac{dy[d_2]}{dv[d_1]}\end{aligned}$$

How **the gradient** is implemented in a custom backward pass?

$$\frac{do}{dv[d_1]} = \sum_{d_2=1}^{out} \frac{do}{dy[d_2]} \cdot \frac{dy[d_2]}{dv[d_1]}$$

https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx,v):
      return y(v)

    @staticmethod
    def backward(ctx, grad_output):
```

$\#$ grad_output $[d_2] \widehat{=} \frac{do}{dy[d_2]}$

$\#$compute $\frac{dy[d_2]}{dv[d_1]}$

$\#$compute $R[d_1] = \sum_{d_2} \text{grad\_output}[d_2] \cdot \frac{dy[d_2]}{dv[d_1]}$

$\#$return R $\#$ this is then the tensor such that $R[d_1] = \frac{do}{dv[d_1]}$

implementation principle -I: LRP follows the same flow as backpropagation (b)

$$o = o(y(v(x))),\ x \in \mathbb{R}^1, v \in \mathbb{R}^{in}, y \in \mathbb{R}^{out}, o \in \mathbb{R}^1$$

by chainrule, used in autograd, we have

$$\frac{do}{dv[d_1]} = \sum_{d_2=1}^{out} \frac{do}{dy[d_2]} \cdot \frac{dy[d_2]}{dv[d_1]}$$

We want to compute gradient or LRP-* for the middle $y = f(v)$

$$\frac{do}{dv[d_1]} = \sum_{d_2=1}^{out} \frac{do}{dy[d_2]} \cdot \frac{dy[d_2]}{dv[d_1]}$$

$$\text{LRP-example: } R_{d_1} = \sum_{d_2} R_{d_2} \cdot \frac{(x_{d_1} w_{[d_1,d_2]})_+}{\sum_{d_1}(x_{d_1} w_{[d_1,d_2]})_+}$$

implementation principle -I: LRP follows the same flow as backpropagation (c)

$$o = o(y(v(x))),\ x \in \mathbb{R}^1, v \in \mathbb{R}^{in}, y \in \mathbb{R}^{out}, o \in \mathbb{R}^1$$

by chainrule, used in autograd, we have

$$\frac{do}{dv[d_1]} = \sum_{d_2=1}^{out} \frac{do}{dy[d_2]} \cdot \frac{dy[d_2]}{dv[d_1]}$$

We want to compute gradient or LRP-* for the middle $y = f(v)$

Note:

$$\sum_{d_2=1}^{out} \frac{do}{dy[d_2]} \frac{dy[d_2]}{dv[d_1]} \quad \widehat{=} \sum_{d_2=1}^{out} R_{d_2} M_{d_1 \leftarrow d_2}$$

$$\frac{do}{dy[d_2]} \quad \widehat{=} R_{d_2}$$

$$\frac{dy[d_2]}{dv[d_1]} \quad \widehat{=} M_{d_1 \leftarrow d_2} \quad = \text{ e.g. } \frac{(x_{d_1} w_{[d_1,d_2]})_+}{\sum_{d_1} (x_{d_1} w_{[d_1,d_2]})_+}$$

implementation principle -I: LRP follows the same flow as backpropagation

$$\frac{do}{dv[d_1]} = \sum_{d_2=1}^{out} \frac{do}{dy[d_2]} \cdot \frac{dy[d_2]}{dv[d_1]}$$

We want to compute gradient or LRP-* for the middle $y = f(v)$

https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def backward(ctx, grad_output):
```

$\#$ grad_output$[d_2] \widehat{=} \frac{do}{dy[d_2]}$ or $\widehat{=} R_{d_2}$

$\#$compute $\frac{dy[d_2]}{dv[d_1]}$ or $M_{d_1 \leftarrow d_2}$

$\#$compute $R[d_1] = \sum_{d_2}$grad_output$[d_2] \cdot \frac{dy[d_2]}{dv[d_1]}$ $\#$ for the gradient

$\#$compute $R[d_1] = \sum_{d_2}$grad_output$[d_2] \cdot M_{d_1 \leftarrow d_2}$ $\#$ for LRP-whatever

$\#$return R

towards a more efficient way of computing the relevance message

We got for computing:

https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def backward(ctx, grad_output):
```

$\quad\quad$ # grad_output$[d_2] \widehat{=} \frac{do}{dy[d_2]}$ or $\widehat{=} R_{d_2}$

#compute $\frac{dy[d_2]}{dv[d_1]}$ or $M_{d_1 \leftarrow d_2}$

#compute $R[d_1] = \sum_{d_2}$grad_output$[d_2] \cdot \frac{dy[d_2]}{dv[d_1]}$ # for the gradient

#compute $R[d_1] = \sum_{d_2}$grad_output$[d_2] \cdot M_{d_1 \leftarrow d_2}$ # for LRP-whatever

#return R

Next:

- want to find a way to compute the 2-tensor in $[d_1, d_2]$: $R_{d_1 \leftarrow d_2}$ using `torch.autograd.backward()`

implementation principle I: custom forward inside backward + its autograd (I)

the structure of many LRP-*:

LRP-* often satisfy the following homogeneity property (Gregoire Montavon):

$$R_j = \sum_k R_k \frac{h(a_j w_{jk})}{\sum_j h(a_j w_{jk}) + h(b)\mathbf{1}[opt]} \quad \text{such that}$$

$$h \text{ satisfies: } h(a_j w_{jk}) = \frac{\partial h(a_j w_{jk})}{\partial a_j} a_j$$

$$\text{define } g_k(a) := \sum_j h(a_j w_{jk}) + h(b)\mathbf{1}[opt]$$

$$\text{then } \Rightarrow R_j = \sum_k R_k \frac{1}{g_k(a)} \frac{\partial g_k}{\partial a_j}(a) a_j$$

- Can use autograd `.backward()` to compute this efficiently
- challenge in practice: implementing $g_k(a)$ in a backward pass, when you cannot simply copy a class using `copy.deepcopy(...)`.

implementation principle I: custom forward inside backward + its autograd (II)

Above as a function – used for one single pytorch module:

```
def lrp_backward(_inp0, layer, relevance_output, eps0, eps):
    #Performs the LRP backward pass, implemented as vanilla fwd+bwd passes.

    relevance_output_data = relevance_output.clone().detach()
    _input = _inp0.clone().detach().requires_grad_(True)

    with torch.enable_grad():
        Z = layer(_input)
    S = safe_divide(relevance_output_data, Z.clone().detach(), eps0, eps)

    #print('started backward')
    Z.backward(S)
    #print('finished backward')

    relevance_input = _input.data * _input.grad.data

    return relevance_input
```

implementation principle I: custom forward inside backward + its autograd (II)

notes on the code

- we are re-creating the forward pass through a single reconstructed layer. `layer` is the module computing $g_k(a)$, `_inp0`$= a$
- $Z = g_k(a)$, $S = \mathrm{tensor}(R_k \frac{1}{g_k(a)})$
- We do this layer by layer
- `Z.backward(S)` computes $\sum_k \frac{\partial h(a_j w_{jk})}{\partial a_j} \left(R_k \frac{1}{g_k(a)} \right)$

```
def lrp_backward(_inp0, layer, relevance_output, eps0, eps):
    #Performs the LRP backward pass, implemented as vanila fwd+bwd passes.

    relevance_output_data = relevance_output.clone().detach()
    _input = _inp0.clone().detach().requires_grad_(True)

    with torch.enable_grad():
        Z = layer(_input)
    S = safe_divide(relevance_output_data, Z.clone().detach(), eps0, eps)

    #print('started backward')
    Z.backward(S)
    #print('finished backward')

    relevance_input = _input.data * _input.grad.data

    return relevance_input
```

example: nn.AdaptiveAvgPool2d with LRP-ϵ (forward pass)

```
class adaptiveavgpool2d_wrapper_fct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, module, eps):

        # define helper function
        def configvalues_totensorlist(module,device): # retrieve dictionary of config parameters
            propertynames=['output_size']
            values=[]
            for attr in propertynames:
              v = getattr(module, attr)
              if isinstance(v, int):
                v=  torch.tensor([v], dtype=torch.int32, device= device)
              elif isinstance(v, tuple):
                v= torch.tensor(v, dtype=torch.int32, device= device)
              else:
                print('v is neither int nor tuple. unexpected')
                exit()
              values.append(v)
            return propertynames,values
        ########################  end of def classproperties2lists(conv2dclass):
        #stash module config params and trainable params
        propertynames,values=configvalues_totensorlist(module,x.device)
        epstensor = torch.tensor([eps], dtype=torch.float32, device= x.device)
        ctx.save_for_backward(x, epstensor, *values ) # *values unpacks the list

        return module.forward(x)
```

example: nn.AdaptiveAvgPool2d with LRP-ϵ (backward pass)

```
class adaptiveavgpool2d_wrapper_fct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, module, eps):

        # define helper function
        def configvalues_totensorlist(module,device): # retrieve dictionary of config parameters
            propertynames=['output_size']
            values=[]
            for attr in propertynames:
              v = getattr(module, attr)
              if isinstance(v, int):
                v=  torch.tensor([v], dtype=torch.int32, device= device)
              elif isinstance(v, tuple):
                v= torch.tensor(v, dtype=torch.int32, device= device)
              else:
                print('v is neither int nor tuple. unexpected')
                exit()
              values.append(v)
            return propertynames,values
        ########################  end of def classproperties2lists(conv2dclass):
        #stash module config params and trainable params
        propertynames,values=configvalues_totensorlist(module,x.device)
        epstensor = torch.tensor([eps], dtype=torch.float32, device= x.device)
        ctx.save_for_backward(x, epstensor, *values ) # *values unpacks the list

        return module.forward(x)
```

quick note:

- we need to stash `ctx.save_for_backward(...)` all trainable parameters and all config parameters, why? to be able to reconstruct the module in the backward pass
- inner function gets the config parameters according to what is defined in `propertynames`

example: nn.AdaptiveAvgPool2d with LRP-ϵ (backward pass)

```
class adaptiveavgpool2d_wrapper_fct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, module, eps):
        #see above
        ctx.save_for_backward(x, epstensor, *values ) # *values unpacks the list
        return module.forward(x)

    @staticmethod
    def backward(ctx, grad_output):
        input_, epstensor, *values  = ctx.saved_tensors
        def tensorlist_todict(values):         # reconstruct dictionary of config parameters
            propertynames=['output_size']
            paramsdict={} # idea: paramsdict={ n: values[i]  for i,n in enumerate(propertynames)  } # but needs to turn tensors
            for i,n in enumerate(propertynames):
              v=values[i]
              if v.numel==1:
                  paramsdict[n]=v.item()
              else:
                  alist=v.tolist()
                  if len(alist)==1:
                    paramsdict[n]=alist[0]
                  else:
                    paramsdict[n]= tuple(alist)
            return paramsdict
        #######################################################################
        paramsdict=tensorlist_todict(values)
        eps=epstensor.item()
        layerclass= torch.nn.AdaptiveAvgPool2d(**paramsdict) #class instantiation
        X = input_.clone().detach().requires_grad_(True)
        R= lrp_backward(_input= X , layer = layerclass , relevance_output = grad_output[0], eps0 = eps, eps=eps)

        return R,None,None #bcs forward has 3 inputs, can do before #print('adaptiveavg2dcustom R', R.shape )
```

custom forward inside backward + its autograd: nn.Linear with LRP-ϵ

How to use that for a linear layer with ϵ-rule?

$$y = w \cdot x + b$$

- usually via `nn.Linear`

example: nn.Linear with LRP-ϵ (forward)

```
class linearlayer_eps_wrapper_fct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, module, eps):

        def configvalues_totensorlist(module):
            propertynames=['in_features','out_features']
            values=[]
            for attr in propertynames:
              v = getattr(module, attr)
              if isinstance(v, int):
                v=  torch.tensor([v], dtype=torch.int32, device= module.weight.device)
              elif isinstance(v, tuple):
                v= torch.tensor(v, dtype=torch.int32, device= module.weight.device)
              else:
                print('v is neither int nor tuple. unexpected')
                exit()
              values.append(v)
            return propertynames,values
        ########################  end of def classproperties2lists(conv2dclass):

        #stash module config params and trainable params
        propertynames,values=configvalues_totensorlist(module)
        epstensor= torch.tensor([eps], dtype=torch.float32, device= x.device)

        if module.bias is None:
          bias=None
        else:
          bias= module.bias.data.clone()
        ctx.save_for_backward(x, module.weight.data.clone(), bias, epstensor, *values ) # *values unpacks the list

        return module.forward(x)
```

example: nn.Linear with LRP-ϵ

```
class linearlayer_eps_wrapper_fct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, module, eps):
        pass # see above...
        ctx.save_for_backward(x, module.weight.data.clone(), bias, epstensor, *values ) # *values unpacks the list
        return module.forward(x)

    @staticmethod
    def backward(ctx, grad_output):
        input_, weight, bias, epstensor, *values  = ctx.saved_tensors
        # reconstruct dictionary of config parameters
        def tensorlist_todict(values):
            propertynames=['in_features','out_features']
            paramsdict={}
            for i,n in enumerate(propertynames):
              v=values[i]
              if v.numel==1:
                  paramsdict[n]=v.item() #to cpu?
              else:
                  alist=v.tolist()
                  #print('alist',alist)
                  if len(alist)==1:
                    paramsdict[n]=alist[0]
                  else:
                    paramsdict[n]= tuple(alist)
            return paramsdict
        #######################################################################
        paramsdict=tensorlist_todict(values)
        eps=epstensor.item()
        if bias is None:
          module=nn.Linear( **paramsdict, bias=False )
          #reconstruct nn.Linear with all config+trainable params
        else:
          module=nn.Linear( **paramsdict, bias=True )
          module.bias= torch.nn.Parameter(bias)
        module.weight= torch.nn.Parameter(weight)

        X = input_.clone().detach().requires_grad_(True)
        R= lrp_backward(_input= X , layer = module , relevance_output = grad_output[0], eps0 = eps, eps=eps)

        return R,None,None
```

implementation principle: custom forward inside backward + its autograd – the forward(...) function

```
class somemodule_wrapper_fct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, module, someparam):

        def configvalues_totensorlist(module):
            propertynames=['module_configparam1','module_configparam2', 'module_configparam3',...]
            #code here for: values = ...
            return propertynames,values

        # get all configuration parameters necessary in order to restore an
        #identical copy of module in def backward(ctx,...):
        propertynames,values=configvalues_totensorlist(module)

        # turn parameters into tensors, can be more than one
        someparam_2tensor1= somefunction (someparam)

        # get all trainable parameters of module
        trainableparam1 = ...
        trainableparam2 = ...

        # stash all of them for backward
        ctx.save_for_backward(x,trainableparam1.data.clone(),trainableparam2.data.clone(),\
                              someparam_2tensor1, *values)

        return module.forward(x)
```

Implementation principle: inside backward custom forward+its autograd – the backward(...)function

```
class somemodule_wrapper_fct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, module, someparam):
        #pass code from ...
        ctx.save_for_backward(x, trainableparam1.data.clone(),trainableparam2.data.clone(),someparam_2tensor1, *values )
        return module.forward(x)

    @staticmethod
    def backward(ctx, grad_output):

        input_, trainableparam1, trainableparam2, someparam_2tensor1,..., *values  = ctx.saved_tensors
        ########################################################################
        # reconstruct dictionary of config parameters
        def tensorlist_todict(values):
            propertynames=['in_channels', 'out_channels', 'kernel_size', 'stride', 'padding', 'dilation', 'groups']
            # paramsdict=
            return paramsdict
        ########################################################################
        paramsdict=tensorlist_todict(values)

        module=nn.PytorchWhateverModule( **paramsdict )

        module.trainableparam1= torch.nn.Parameter(trainableparam1)
        module.trainableparam2= torch.nn.Parameter(trainableparam2)

        gk_function= gk_frommodule(module, someparam = someparam_2tensor1.item())

        X = input_.clone().detach().requires_grad_(True)
        R= lrp_backward(_input= X , layer = gk_function , relevance_output = grad_output[0], eps0 = 1e-12, eps=0)

        return R,None, None # as many as you have inputs in forward() minus the ctx
```

example: nn.Conv2d (backward here shown only)

```
class conv2d_beta0_wrapper_fct(torch.autograd.Function): #this is LRP-beta0 !!! NOT LRP-eps
    @staticmethod
    def forward(ctx, x, module, someparam):
        #pass code from one slide above
        return module.forward(x)

    @staticmethod
    def backward(ctx, grad_output):

        input_, conv2dweight, conv2dbias, lrpignorebiastensor, *values  = ctx.saved_tensors
        ########################################################################
        # reconstruct dictionary of config parameters
        def tensorlist_todict(values):
            propertynames=['in_channels', 'out_channels', 'kernel_size', 'stride', 'padding', 'dilation', 'groups']
            # paramsdict=
            return paramsdict
        ########################################################################
        paramsdict=tensorlist_todict(values)

        if conv2dbias is None:
          module=nn.Conv2d( **paramsdict, bias=False )
        else:
          module=nn.Conv2d( **paramsdict, bias=True )
          module.bias= torch.nn.Parameter(conv2dbias)
        module.weight= torch.nn.Parameter(conv2dweight)

        pnconv = posnegconv(module, ignorebias = lrpignorebiastensor.item()) #comp (wx)_+ as w_{+}x_{+} + w_{-}x_{-}

        X = input_.clone().detach().requires_grad_(True)
        R= lrp_backward(_input= X , layer = pnconv , relevance_output = grad_output[0], eps0 = 1e-12, eps=0)

        return R,None, None
```

implementation principle I: custom forward inside backward + its autograd (IIId)

One Warning to avoid potential buggy LRP usage:

- LRP-$\beta = 0$ is often implemented as

$$R_{i \leftarrow k}(\mathbf{x}) = R_k \frac{w_{ik}(x_i)_+}{\sum_{i'} w_{i'k}(x_{i'})_+} \quad (1)$$

shown in this talk:

$$R_{i \leftarrow k}(\mathbf{x}) = R_k \frac{(w_{ik}x_i)_+}{\sum_{i'} (w_{i'k}x_{i'})_+} \quad (2)$$

- Equation (1) is valid ONLY for intermediate layers in ReLU-networks
- Equation (1) is mistaken for the first convolution (input=image−mean not positive)
- Equation (1) is mistaken for inputs from e.g. hidden states of LSTMs

implementation principle I: custom forward inside backward + its autograd

- LRP-z_β (just lazy),
- maxpool (no need for the multiplication with x_i)

have slightly different implementations for convenience

implementation principles X: how to treat biases?

- bias as constant-value firing legitimate neuron?
- bias as nuisance term onto which relevance dissipates?

!!BUG: You absolutely must not zero out biases in the forward pass!!

- if you do that, you explain a different predictor than your original model
  - predicted class gets wrong.
  - inputs x_i used to distribute relevances towards layer inputs getting wrong
  - You can zero out biases in the LRP-backward pass
  - alternative: move biases into the input layer by redistribution to the module input and iterative chaining.

implementation principles XI: LSTM etc.

Works by Leila Arras et al. derive and evaluate LRP rules for LSTM. https://github.com/ArrasL/LRP_for_LSTM/blob/master/misc/Talk_slides.pdf
From there it is straightforward to extend it to other recurrent structures
Consider an LSTM at time step t.

- It has states h_{t-1} (temporary hidden state) and c_{t-1} (long term memory cell).
- It receives input x_t.
- It computes updates h_t, c_t. Usually h_t is used to provide input at time step t, e.g. for sequence element classification by inputting h_t into a `nn.Linear` layer for getting the logits at time step t.

$$
\begin{aligned}
u_t &= tanh(W_{ux}x_t + W_{uh}h_{t-1} + b_u) \\
i_t &= \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i) \\
f_t &= \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f) \\
c_t &= f_t \odot c_{t-1} + i_t \odot u_t \\
o_t &= tanh(W_{ox}x_t + W_{oh}h_{t-1} + b_o) \\
h_t &= o_t \odot tanh(c_t)
\end{aligned}
$$

implementation principles XI: LSTM etc.

Works by Leila Arras et al. derive and evaluate LRP rules for LSTM.

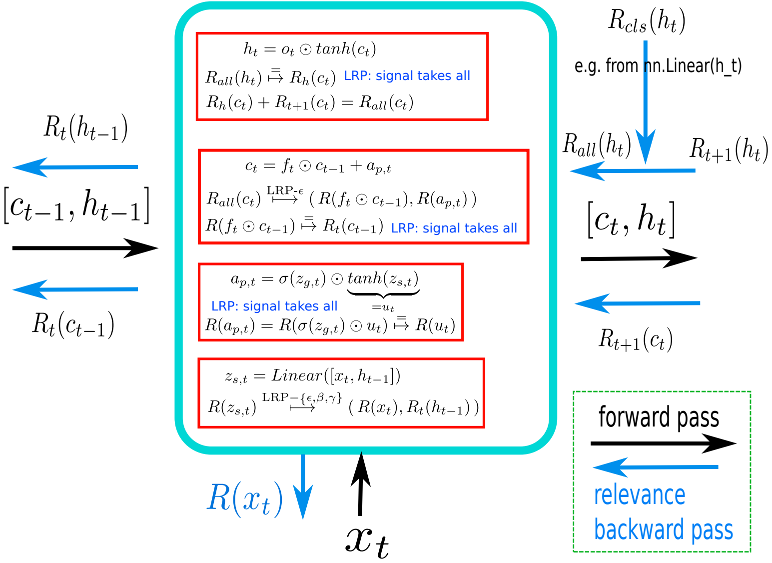
$$u_t = tanh(W_{ux}x_t + W_{uh}h_{t-1} + b_u),\ i_t = \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i),$$
$$f_t = \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot u_t, \qquad o_t = tanh(W_{ox}x_t + W_{oh}h_{t-1} + b_o)$$
$$h_t = o_t \odot tanh(c_t)$$

Fundamental ideas: have three types of layer ops.
Have Two types of information: Gates (g) and signals (s)

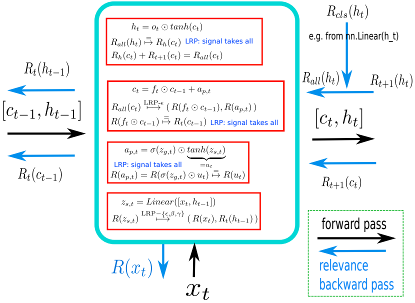
$$\begin{aligned} \text{Linear mappings: } z_{s,t} &= W_{s,xh}[x_t, h_{t-1}] + b \\ z_{g,t} &= W_{g,xh}[x_t, h_{t-1}] + b \\ \text{Gate multiplications: } a_{p,t} &= \sigma(z_{g,t}) \odot \tanh(z_{s,t}) \\ \text{Accumulations: } c_t &= f_t \odot c_{t-1} + a_{p,t} \end{aligned}$$

- suppose one has computed from time $t+1$ relevances $R(h_t)$ for h_t, $R_{t+1}(c_t)$ for c_t from c_{t+1}
- Goal: iterate one time step backwards – derive relevances $R(h_{t-1})$ for h_{t-1}, $R_t(c_{t-1})$ for c_{t-1} from c_t, and $R(x_t)$ for x_t

implementation principles XI: LSTM etc.



implementation principles XI: LSTM etc.



Three principles

- signal takes all in terms like $w = \sigma(z_{g,t}) \odot \tanh(z_{s,t})$ do not distribute relevance on gates $z_{g,t}$. Only onto signal terms $z_{s,t}$:

$$R(w) \mapsto (\, R(z_{g,t}), R(z_{s,t})\,) = (0, R(z_{s,t}))$$

- $+$: use LRP-ϵ
- Linear operations: use LRP-ϵ, β, γ up to evaluation results.

canonization II: densenet-121 example

`copy_densenet_onlycopy.py` manual steps:

- create derived class
- replace in classifier head calls to `F.function(...)` by `nn.Module` equivalents in the derived class
  - adapt classifier head to use these equivalents: `self.toprelu`, `self.toppool`
- `def copyfromdensenet(self,net):` to copy trainable parameters from trained net
  - nn.Linear in the classifier head, nn.Conv2d at the start and denseblocks
  - canonization different from resnets!!
  - have: BN $\rightarrow$ ReLU $\rightarrow$ Conv blocks. Need to deal with this structure

canonization II: densenet-121 example

have: BN $\rightarrow$ ReLU $\rightarrow$ Conv blocks.

- step 1: swap the BN $\rightarrow$ ReLU into: ThreshReLU $\rightarrow$ BN
- step 2: fuse BN $\rightarrow$ Conv.

- result: ThreshRELU $\rightarrow$ tensorbiasedConv.
  step 2 will result in a convolution layer which cannot be represented by `nn.Conv2D` anymore, because it will have a bias which is spatially varying.

canonization II: densenet-121 example

step 1: swap the BN $\rightarrow$ ReLU into: ThreshReLU $\rightarrow$ BN

A theorem

$$\text{given } BN(x) = \frac{w_{bn}}{\sigma_{bn}} x - \frac{w_{bn}\mu_{bn}}{\sigma_{bn}} + b_{bn}$$

The following commutation holds for any $w_{bn} \neq 0$:

$$ReLU(BN(x)) = BN(ThreshReLU(x)) \text{ with}$$

$$ThreshReLU(x) = \begin{cases} x & \text{if } x - t > 0 \text{ and } w_{bn} > 0 \\ x & \text{if } x - t < 0 \text{ and } w_{bn} < 0 \\ t & \text{else} \end{cases}$$

$$\text{for } t = \mu_{bn} - \frac{b_{bn}\sigma_{bn}}{w_{bn}}$$

$$= t + (x - t)\{\mathbf{1}[x - t > 0]\mathbf{1}[w_{bn} > 0] + \mathbf{1}[x - t < 0]\mathbf{1}[w_{bn} < 0]\}$$

canonization II: densenet-121 example

step 1: swap the BN $\rightarrow$ ReLU $\rightarrow$ Conv2d

- replace it by ThreshReLU $\rightarrow$ BN $\rightarrow$ Conv2d
- code: `get_clamplayer` in the code computes the ThreshReLU

step 2: now can fuse the BN into the Conv2d

canonization II: densenet-121 example

step 2: now can fuse the BN into the Conv2d

$$\begin{aligned}
conv(BN(x)) &= conv[w](\alpha x + \beta) + b \\
&= conv[w\alpha](x) + conv[w](broadcast(\beta)) + b \\
conv.w.shape &= (n_{out}, n_{in}, ksize, ksize),\ \alpha.shape = n_{in} \\
(w\alpha)[o,c,h,w] &:= w[o,c,h,w]\alpha[c] \text{ and} \\
broadcast(\beta).shape &= (n_{in}, ksize, ksize) \\
broadcast(\beta)[c,h,w] &= \beta[c] \\
conv[w](broadcast(\beta)).shape &= (n_{out}, f, f)
\end{aligned}$$

The point here is: if *conv* is using any padding, then

- $conv(broadcast(\beta))$ is not constant the spatial dimensions h, w in $[:, h, w]$
- for $kernelsize = 3$, $pad = 1$ the value on the fringe indices $h = 0$ and $h = f - 1$ will be different from $h \in [1, f-2]$
- thats why defining the class `tensorbiased_convlayer` for densenets.

canonization II: densenet-121 example

- step 2: now can fuse the BN into the Conv2d: `def convafterbn_returntensorbiasedconv(conv,bn)` implements this
- need a similar trick in the classifier head:

$$
\begin{aligned}
&\text{BN(norm5)}\to\text{relu(toprelu)}\to\text{adaptiveAvgPool(toppool)}\to\text{linear(classifier)}\\
=\ &\text{ThresReLU} \to \text{BN} \to\text{adaptiveAvgPool(toppool)}\to\text{linear(classifier)}\\
=\ &\text{ThresReLU} \to\text{adaptiveAvgPool(toppool)}\to \text{BN} \to\text{linear(classifier)}\\
=\ &\text{ThresReLU} \to\text{adaptiveAvgPool(toppool)}\to \text{tensorbiasedlinear(classifier)}
\end{aligned}
$$

code it yourself

- base code in: `https://github.com/AlexBinder/LRP_Pytorch_Resnets_Densenet`
- really easy level – step1: implement guided backprop using `torch.autograd.Function`. (the backward hook-based version to check against is in the appendix)
- intermediate level – step2: implement LRP-γ for conv2d/resnet or tensorbiased_conv/densenet and try it out for various choices of γ.

  In fact, if you understand how the class `posnegconv` works inside function `conv2d_beta0_wrapper_fct` in the resnet case , then it is not hard.

code it yourself – step1: guided backprop

really easy level – step1: implement guided backprop using torch.autograd.Function.

- create a `torch.autograd.Function`-derived variant similar to `relu_wrapper_fct`. It is a wrapper same as for all those pytorch modules. (1) It does the relu in the forward pass, and (2) for the backward pass computes the modified gradient as per guided backprop-rule (https://arxiv.org/abs/1412.6806). Then you wrap it into a `zeroparam_wrapper_class` as done in `get_lrpwrapperformodule` for the relu case.
- take the densenet or resnet code. it is easier to just create your own function `def add_guided_backprop(self):` (or modify the `copy_from*` routines)
- densenet: you must create a derived class to change the `F.relu(...)`call into a module inside your densenet class see `self.toppool=` in `copy_from_densenet`.
- steps in `def add_guided_backprop(self):`
  - loop over `self.named_parameters()`
  - if it is an instance of `nn.ReLU`, then use `def setbyname(self,name,value)` to replace the relu by your coded `torch.autograd.Function`-derived variant
- take note of: https://www.youtube.com/watch?v=422chFbIluo

code it yourself – step2: LRP-γ

intermediate level – step2: implement LRP-γ for conv2d/resnet or tensorbiased_conv/densenet and try it out for various choices of γ.

- https://link.springer.com/chapter/10.1007/978-3-030-28954-6_10, http://iphome.hhi.de/samek/pdf/MonXAI19.pdf
  in the input-sign invariant formulation ($\gamma > 0$ for stability):

$$R_j = \sum_k R_k \frac{x_j w_{jk} + \gamma (x_j w_{jk})_+}{\sum_j x_j w_{jk} + \gamma (x_j w_{jk})_+ + \mathbf{1}[opt](b + \gamma b_+)}$$

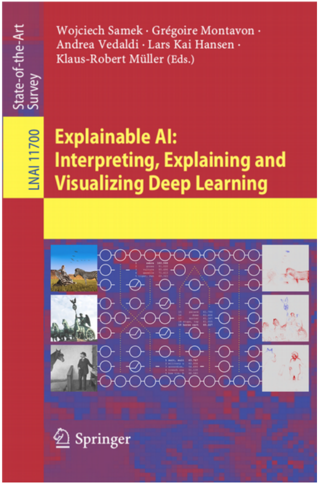
- What is g_k? it is a γ-weighted sum of `nn.Conv2d` and `posnegconv` outputs. Actually once this is clear, it is easy.
- you can create a wrapper analogously to `conv2d_beta0_wrapper_fct` / `tensorbiasedconv2d_beta0_wrapper_fct`
- replace your clause in `get_lrpwrapperformodule` for classes `nn.Conv2d` or `tensorbiased_convlayer` (resnet / densenet respectively)

References

Overview papers:
Layer-wise relevance propagation: an Overview
G Montavon, A Binder, S Lapuschkin, W Samek, KR Müller
in: Explainable AI: Interpreting, Explaining and Visualizing Deep Learning, Springer, 193-209
https://link.springer.com/chapter/10.1007/978-3-030-28954-6_10

Towards best practice in explaining neural network decisions with LRP
M Kohlbrenner, A Bauer, S Nakajima, A Binder, W Samek, S Lapuschkin

Book on all kinds of Explainability for ML



Organization of the book:

- Part I Towards AI Transparency
- Part II Methods for Interpreting AI Systems
- Part III Explaining the Decisions of AI Systems
- Part IV Evaluating Interpretability and Explanations
- Part V Applications of Explainable AI
- 22 Chapters

Tutorial Paper
Montavon et al., "Methods for interpreting and understanding deep neural networks", Digital Signal Processing, 73:1-5, 2018

Keras Explanation Toolbox
`https://github.com/albermax/innvestigate`

link to the book:

https://www.springer.com/gp/book/9783030289539

papers, demos, ice cream at: www.explain-ai.org

Thank you

Forward hooks (for a module)

- can be registered to a module, executed after the forward pass of this module (see pre-hook)
- signature:
  `hook(module, inputtensor, outputtensor) -> None`
- how to register them ?
  below example for a module `conv`
  `handle=net.layer3.2.conv.register_forward_hook(hook)`

Forward hooks (for a module)

good for ?

- printing stats of feature maps (see trivial example)
- saving feature maps to disk for further analysis
- saving intermediate tensors into the module for later reading them out (e.g. running means)
- suitable for feature maps which never appear explicitly in the network forward

Backward hooks (for a module or a tensor)

focus here: hooks for a module

- executed after the backward pass
- good for:
  - printing and saving gradient stats!
  - some simple network attribution models
- signature:
  `hook(module, grad_in, grad_out) -> Tensor or None`
- how to register them ?
  `handle=net.layer3.2.conv.register_backward_hook(hook)`

bad news: Backward hooks are broken!

pytorch ticket #598

https://github.com/pytorch/pytorch/issues/598

- input signatures are those of the
  last operation performed within the module!
- example of breakage: `nn.Conv2d` with bias: compare input signatures CPU vs GPU
  `backwardsizes_behabviour2.py`

Backward hooks still usable if ...

one is interested about the gradient from above, and not what is computed into the inputs:

- when using `grad_out[0]` from `hook(module, grad_in, grad_out)`

Application: backward hook implementing guided backprop

Figure 1 in https://arxiv.org/abs/1412.6806.

backward hook for ReLU is not broken

```
def gb_bw_hook(module, input_, output):
    if isinstance(module,nn.ReLU):
        print('shapes', output[0].shape)
        grad_input = input_[0].clone()
        grad_input[grad_input < 0] = 0
        return grad_input,
```

What is the difference of using `isinstance()` vs `type()` ?

parametrized hooks

when you need to pass parameters, e.g. info on filepaths for saving ...

- create a function *a* which has hook signature + extra parameters: `a(*hooksig,*additionalparams)`
- create a function `b(*additionalparams) -> hook(*hooksig)` which returns something with the signature of a hook
  - internally `b(*additionalparams)` defines a function `hook(*hooksig)`
  - `hook(*hooksig)` calls `a(*hooksig,*additionalparams)` and returns the value of the call `a(*inputsig,*additionalparams)`
  - `b(*additionalparams)` returns the name of the string <u>hook</u> which is the handle to call the function
- see example

usage of the handles?

```
...
handles.append(handle)
...
use code
...
for h in handles:
    h.remove()
handles=[]
```

Application: backward hook implementing guided backprop

...