Deep learning

9.3. Visualizing the processing in the input

François Fleuret
https://fleuret.org/dlc/
Dec 20, 2020
Occlusion sensitivity
Another approach to understanding the functioning of a network is to look at the behavior of the network “around” an image.

For instance, we can get a simple estimate of the importance of a part of the input image for a given output by computing the difference between:

1. the value of that output on the original image, and
2. the value of the same output with that part occluded.
Another approach to understanding the functioning of a network is to look at the behavior of the network “around” an image.

For instance, we can get a simple estimate of the importance of a part of the input image for a given output by computing the difference between:

1. the value of that output on the original image, and
2. the value of the same output with that part occluded.

This is computationally intensive since it requires as many forward passes as there are locations of the occlusion mask, ideally the number of pixels.
Francois Fleuret

Deep learning / 9.3. Visualizing the processing in the input
Original images

Occlusion sensitivity, mask $32 \times 32$, stride of 2, AlexNet
Occlusion sensitivity, mask $32 \times 32$, stride of 2, VGG19

Original images
Saliency maps
An alternative is to compute the gradient of an output with respect to the input (Erhan et al., 2009; Simonyan et al., 2013), e.g.

\[ \nabla_{x} f_c(x; w) \]

where \( x \) stresses that the gradient is computed with respect to the input \( x \) and not as usual with respect to the parameters \( w \).
This can be implemented by specifying that we need the gradient with respect to the input.

Using `torch.autograd.grad` to compute the gradient wrt the input image instead of `torch.autograd.backward` has the advantage of not changing the model’s parameter gradients.

```python
input.requires_grad_()
output = model(input)
grad_input, = torch.autograd.grad(output[0, c], input)
```

Note that since `torch.autograd.grad` computes the gradient of a function with possibly multiple inputs, the returned result is a tuple.
The resulting maps are quite noisy. For instance with AlexNet:
This is due to the local irregularity of the network’s response as a function of the input.

Figure 2. The partial derivative of $S_c$ with respect to the RGB values of a single pixel as a fraction of the maximum entry in the gradient vector, $\max_i \frac{\partial S_c}{\partial x_i}(t)$, (middle plot) as one slowly moves away from a baseline image $x$ (left plot) to a fixed location $x + \epsilon$ (right plot). $\epsilon$ is one random sample from $\mathcal{N}(0, 0.01^2)$. The final image $(x + \epsilon)$ is indistinguishable to a human from the origin image $x$.

(Smilkov et al., 2017)
Smilkov et al. (2017) proposed to smooth the gradient with respect to the input image by averaging over slightly perturbed versions of the latter.

\[
\tilde{\nabla}_x f_y(x; w) = \frac{1}{N} \sum_{n=1}^{N} \nabla_x f_y(x + \epsilon_n; w)
\]

where \(\epsilon_1, \ldots, \epsilon_N\) are i.i.d of distribution \(\mathcal{N}(0, \sigma^2 I)\), and \(\sigma\) is a fraction of the gap \(\Delta\) between the maximum and the minimum of the pixel values.
A simple version of this “SmoothGrad” approach can be implemented as follows

```python
std = std_fraction * (img.max() - img.min())
acc_grad = img.new_zeros(img.size())

for q in range(nb_smooth):  # This should be done with mini-batches ...
    noisy_input = img + img.new(img.size()).normal_(0, std)
    noisy_input.requires_grad_()
    output = model(noisy_input)
    grad_input, = torch.autograd.grad(output[0, c], noisy_input)
    acc_grad += grad_input

acc_grad = acc_grad.abs().sum(1)  # sum across channels
```
Original images

Gradient, AlexNet

SmoothGrad, AlexNet, $\sigma = \frac{\Delta}{4}$
Gradient, VGG19

SmoothGrad, VGG19, $\sigma = \frac{\Delta}{4}$
Grad-CAM
Gradient-weighted Class Activation Mapping (Grad-CAM) proposed by Selvaraju et al. (2016) visualizes the importance of the input sub-parts according to the activations in a specific layer.

It computes a sum of the activations weighted by the average gradient of the output of interest wrt individual channels.
Formally, let $k \in \{1, \ldots, C\}$ be a channel number, $A^k \in \mathbb{R}^{H \times W}$ the output feature map $k$ of the selected layer, $c$ a class number, and $y^c$ the network’s logit for that class.

The channel weights are

$$\alpha^c_k = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \frac{\partial y^c}{\partial A^k_{i,j}}.$$ 

And the final localization map is

$$L^c_{\text{Grad-CAM}} = \text{ReLU} \left( \sum_{k=1}^{C} \alpha^c_k A^k \right).$$
We are going to test it with VGG19.

VGG(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace=True)
        /.../
        (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (35): ReLU(inplace=True)
        (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
        (0): Linear(in_features=25088, out_features=4096, bias=True)
        (1): ReLU(inplace=True)
        (2): Dropout(p=0.5, inplace=False)
        (3): Linear(in_features=4096, out_features=4096, bias=True)
        (4): ReLU(inplace=True)
        (5): Dropout(p=0.5, inplace=False)
        (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
)
We are going to implement Grad-CAM by modifying the layer of interest to store activations and gradient w.r.t activations.

The class `nn.Module` provides methods to register “hook” functions that are called during the forward or the backward pass, and can implement a different computation for the latter.
For instance

```python
>>> x = torch.tensor([ 1.23, -4.56 ])
>>> m = nn.ReLU()
>>> m(x)
tensor([ 1.2300,  0.0000])
```
For instance

```python
>>> x = torch.tensor([ 1.23, -4.56 ])
>>> m = nn.ReLU()
>>> m(x)
tensor([ 1.2300, 0.0000])

>>> def my_hook(m, input, output):
...     print(str(m) + ' got ' + str(input[0].size()))
...
>>> handle = m.register_forward_hook(my_hook)
>>> m(x)
ReLU() got torch.Size([2])
tensor([ 1.2300, 0.0000])
```
For instance

```python
>>> x = torch.tensor([ 1.23, -4.56 ])
>>> m = nn.ReLU()
>>> m(x)
tensor([ 1.2300, 0.0000])

>>> def my_hook(m, input, output):
...     print(str(m) + ' got ' + str(input[0].size()))
...
>>> handle = m.register_forward_hook(my_hook)
>>> m(x)
ReLU() got torch.Size([2])
tensor([ 1.2300, 0.0000])

>>> handle.remove()
>>> m(x)
tensor([ 1.2300, 0.0000])
```
For Grad-CAM: we first define hooks to store the feature maps in the forward pass and gradient wrt them in the backward:

```python
def hook_store_A(module, input, output):
    module.A = output[0]

def hook_store_dydA(module, grad_input, grad_output):
    module.dydA = grad_output[0]
```

Then, load a pre-trained VGG19, and install the hooks in the last ReLU layer of the convolutional part:

```python
model = torchvision.models.vgg19(pretrained = True)
model.eval()
layer = model.features[35]  # Last ReLU of the conv layers
layer.register_forward_hook(hook_store_A)
layer.register_backward_hook(hook_store_dydA)
```
For Grad-CAM: we first define hooks to store the feature maps in the forward pass and gradient wrt them in the backward:

```python
def hook_store_A(module, input, output):
    module.A = output[0]

def hook_store_dydA(module, grad_input, grad_output):
    module.dydA = grad_output[0]
```

Then, load a pre-trained VGG19, and install the hooks in the last ReLU layer of the convolutional part:

```python
model = torchvision.models.vgg19(pretrained = True)
model.eval()

layer = model.features[35] # Last ReLU of the conv layers

layer.register_forward_hook(hook_store_A)
layer.register_backward_hook(hook_store_dydA)
```
Load an image and make it a one sample batch:

```python
to_tensor = torchvision.transforms.ToTensor()
input = to_tensor(PIL.Image.open('example_images/elephant_hippo.png')).unsqueeze(0)
```

Compute the network's output, the gradient, and Grad-CAM:

```python
output = model(input)
c = 386 # African elephant
output[0, c].backward()
alpha = layer.dydA.mean((2, 3), keepdim = True)
L = torch.relu((alpha * layer.A).sum(1, keepdim = True))
```

Save it as a resized colored heat-map:

```python
L = L / L.max()
L = F.interpolate(L, size = (input.size(2), input.size(3)), mode = 'bilinear', align_corners = False)
l = L.view(L.size(2), L.size(3)).detach().numpy()
PIL.Image.fromarray(numpy.uint8(cm.gist_earth(l) * 255)).save('result.png')
```
Load an image and make it a one sample batch:

to_tensor = torchvision.transforms.ToTensor()
input = to_tensor(PIL.Image.open('example_images/elephant_hippo.png')).unsqueeze(0)

Compute the network’s output, the gradient, and $L^c_{\text{Grad-CAM}}$:

output = model(input)

c = 386 # African elephant
output[0, c].backward()

alpha = layer.dydA.mean((2, 3), keepdim = True)
L = torch.relu((alpha * layer.A).sum(1, keepdim = True))
Load an image and make it a one sample batch:

to_tensor = torchvision.transforms.ToTensor()
input = to_tensor(PIL.Image.open('example_images/elephant_hippo.png')).unsqueeze(0)

Compute the network’s output, the gradient, and \( L_{\text{Grad-CAM}} \):

output = model(input)

c = 386 # African elephant
output[0, c].backward()

alpha = layer.dydA.mean((2, 3), keepdim = True)
L = torch.relu((alpha * layer.A).sum(1, keepdim = True))

Save it as a resized colored heat-map:

L = L / L.max()
L = F.interpolate(L, size = (input.size(2), input.size(3)),
mode = 'bilinear', align_corners = False)

l = L.view(L.size(2), L.size(3)).detach().numpy()
PIL.Image.fromarray(numpy.uint8(cm.gist_earth(l) * 255)).save('result.png')
The end

