Deep learning

9.3. Visualizing the processing in the input

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ DE GENÈVE

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.

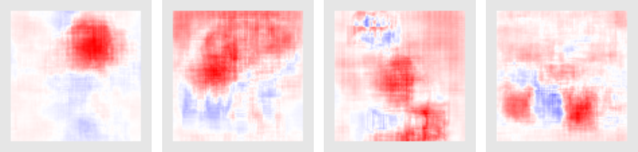
This is computationally intensive since it requires as many forward passes as there are locations of the occlusion mask, ideally the number of pixels.

Original images

Occlusion mask 32×32

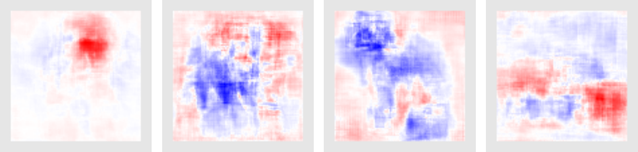
Original images

Occlusion sensitivity, mask 32×32, stride of 2, AlexNet



Original images

Occlusion sensitivity, mask 32×32, stride of 2, VGG19



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.

$$\nabla_{|x} f_c(x; w)$$

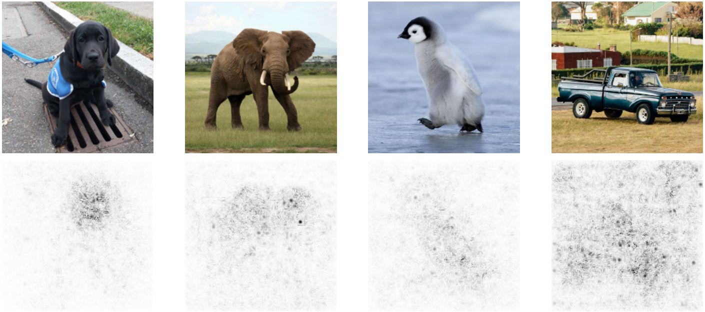
where $|x$ stresses that the gradient is computed with respect to the input x and not as usual with respect to the parameters w.

This can be implemented with `torch.autograd.grad` to compute the gradient w.r.t. the input image (this has the advantage of not changing the model's parameter gradients, contrary to `torch.autograd.backward`.)

```
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.

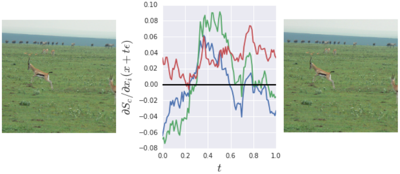


Figure 2. The partial derivative of S_c with respect to the RGB values of a single pixel as a fraction of the maximum entry in the gradient vector, $\max_i \frac{\partial S_c}{\partial x_i}(t)$, (middle plot) as one slowly moves away from a baseline image x (left plot) to a fixed location $x + \epsilon$ (right plot). ϵ 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.

$$\tilde{\nabla}_{|x} f_y(x; w) = \frac{1}{N} \sum_{n=1}^{N} \nabla_{|x} f_y(x + \epsilon_n; w)$$

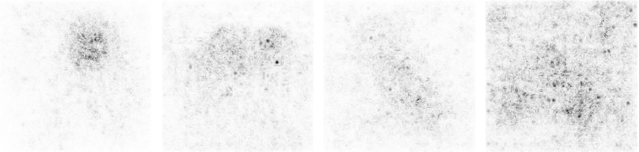
where $\epsilon_1, \dots, \epsilon_N$ are i.i.d of distribution $\mathcal{N}(0, \sigma^2 \mathbf{I})$, and σ is a fraction of the gap Δ between the maximum and the minimum of the pixel values.

A simple version of this “SmoothGrad” approach can be implemented as follows

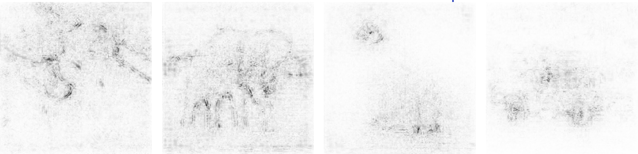
```
std = std_fraction * (img.max() - img.min())
acc_grad = img.new_zeros(img.size())

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

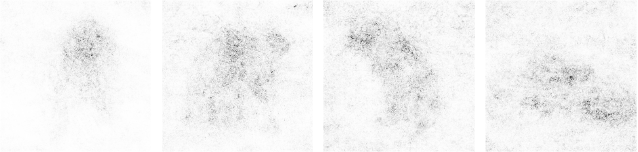


SmoothGrad, AlexNet, $\sigma = \frac{\Delta}{4}$

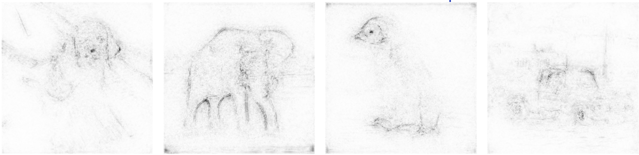


Original images

Gradient, VGG19



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



Deconvolution and guided back-propagation

Zeiler and Fergus (2014) proposed to invert the processing flow of a convolutional network by constructing a corresponding **deconvolutional network** to compute the "activating pattern" of a sample.

As they point out, the resulting processing is identical to a standard backward pass, except when going through the ReLU layers.

Remember that if s is one of the input to a ReLU layer, and x the corresponding output, we have for the forward pass

$$x = \max(0, s),$$

and for the backward

$$\frac{\partial \ell}{\partial s} = \mathbf{1}_{\{s>0\}} \frac{\partial \ell}{\partial x}.$$

Zeiler and Fergus's deconvolution can be seen as a backward pass where we propagate back through ReLU layers the quantity

$$\max\left(0, \frac{\partial \ell}{\partial x}\right) = \mathbf{1}_{\left\{\frac{\partial \ell}{\partial x} > 0\right\}} \frac{\partial \ell}{\partial x},$$

instead of the usual

$$\frac{\partial \ell}{\partial s} = \mathbf{1}_{\{s>0\}} \frac{\partial \ell}{\partial x}.$$

Zeiler and Fergus's deconvolution can be seen as a backward pass where we propagate back through ReLU layers the quantity

$$\max\left(0, \frac{\partial \ell}{\partial x}\right) = \mathbf{1}_{\left\{\frac{\partial \ell}{\partial x} > 0\right\}} \frac{\partial \ell}{\partial x},$$

instead of the usual

$$\frac{\partial \ell}{\partial s} = \mathbf{1}_{\{s>0\}} \frac{\partial \ell}{\partial x}.$$

This quantity is positive for units whose output has a positive contribution to the response, kills the others, and is not modulated by the pre-layer activation s.

Springenberg et al. (2014) improved upon the deconvolution with the **guided back-propagation**, which aims at the best of both worlds: Discarding structures which would not contribute positively to the final response, and discarding structures which are not already present.

It back-propagates through the ReLU layers the quantity

$$\mathbf{1}_{\{s>0\}}\mathbf{1}_{\left\{\frac{\partial \ell}{\partial x}>0\right\}}\frac{\partial \ell}{\partial x}$$

which keeps only units which have a positive contribution and activation.

So these three visualization methods differ only in the quantities propagated through ReLU layers during the back-pass:

- back-propagation (Erhan et al., 2009; Simonyan et al., 2013):

$$\mathbf{1}_{\{s>0\}} \frac{\partial \ell}{\partial x},$$

- deconvolution (Zeiler and Fergus, 2014):

$$\mathbf{1}_{\left\{\frac{\partial \ell}{\partial x}>0\right\}} \frac{\partial \ell}{\partial x},$$

- guided back-propagation (Springenberg et al., 2014):

$$\mathbf{1}_{\{s>0\}} \mathbf{1}_{\left\{\frac{\partial \ell}{\partial x}>0\right\}} \frac{\partial \ell}{\partial x}.$$

These procedures can be implemented simply in PyTorch by changing the `nn.ReLU`'s backward pass.

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

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

For instance

```
>>> 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

```
>>> 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])
```

Using hooks, we can implement the deconvolution as follows:

```
def relu_backward_deconv_hook(module, grad_input, grad_output):
    return F.relu(grad_output[0]),

def equip_model_deconv(model):
    for m in model.modules():
        if isinstance(m, nn.ReLU):
            m.register_backward_hook(relu_backward_deconv_hook)
```

```
def grad_view(model, image_name):
    to_tensor = transforms.ToTensor()
    img = to_tensor(PIL.Image.open(image_name))
    img = 0.5 + 0.5 * (img - img.mean()) / img.std()

    model.to(device)
    img = img.to(device)

    input = img.view(1, img.size(0), img.size(1), img.size(2)).requires_grad_()
    output = model(input)
    result, = torch.autograd.grad(output.max(), input)

    result = result / result.max() + 0.5

    return result
```

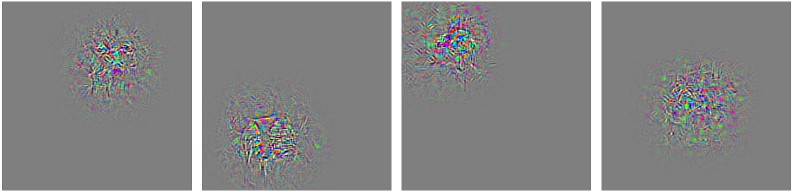
```
def grad_view(model, image_name):
    to_tensor = transforms.ToTensor()
    img = to_tensor(PIL.Image.open(image_name))
    img = 0.5 + 0.5 * (img - img.mean()) / img.std()

    model.to(device)
    img = img.to(device)

    input = img.view(1, img.size(0), img.size(1), img.size(2)).requires_grad_()
    output = model(input)
    result, = torch.autograd.grad(output.max(), input)

    result = result / result.max() + 0.5

    return result

model = models.vgg16(weights = 'IMAGENET1K_V1')
model.eval()
model = model.features
equip_model_deconv(model)
result = grad_view(model, 'blacklab.jpg')
utils.save_image(result, 'blacklab-vgg16-deconv.png')
```

The code is the same for the guided back-propagation, except the hooks themselves:

```
def relu_forward_gbackprop_hook(module, input, output):
    module.input_kept = input[0]

def relu_backward_gbackprop_hook(module, grad_input, grad_output):
    return F.relu(grad_output[0]) * F.relu(module.input_kept).sign(),

def equip_model_gbackprop(model):
    for m in model.modules():
        if isinstance(m, nn.ReLU):
            m.register_forward_hook(relu_forward_gbackprop_hook)
            m.register_backward_hook(relu_backward_gbackprop_hook)
```

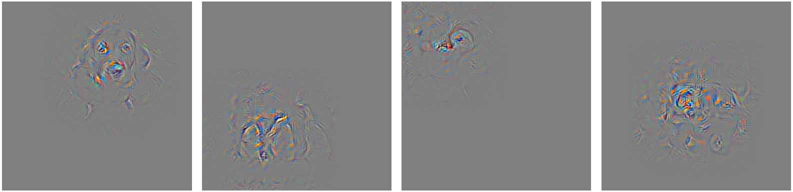
Original images

AlexNet, max feature response, gradient



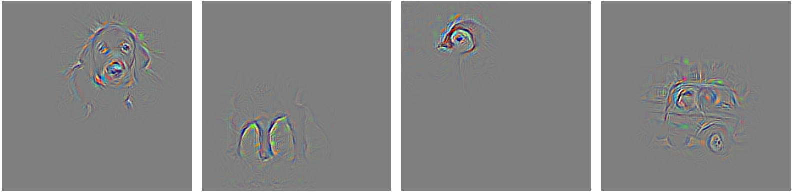
Original images

AlexNet, max feature response, deconvolution



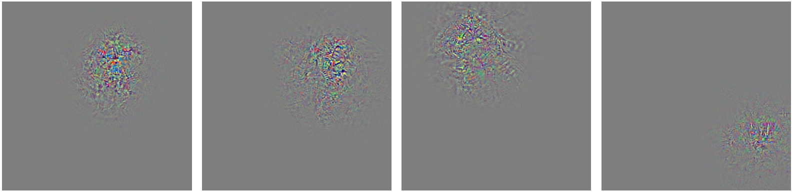
Original images

AlexNet, max feature response, guided back-propagation



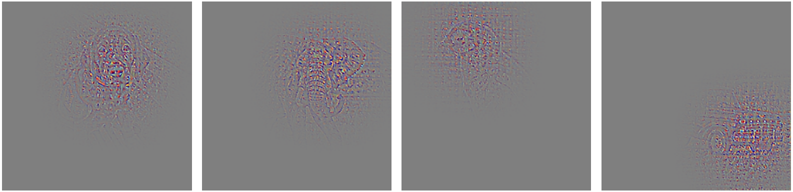
Original images

VGG16, max feature response, gradient



Original images

VGG16, max feature response, deconvolution

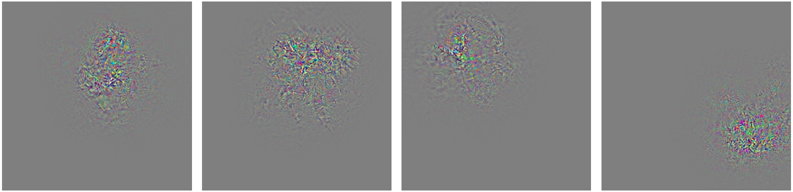


Original images

VGG16, max feature response, guided back-propagation

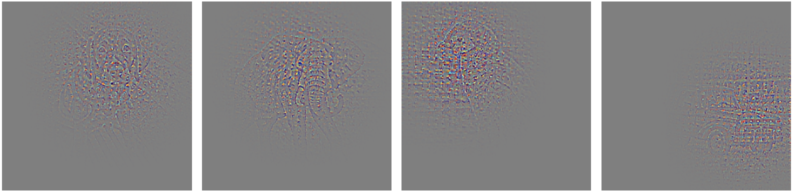
Original images

VGG19, max feature response, gradient



Original images

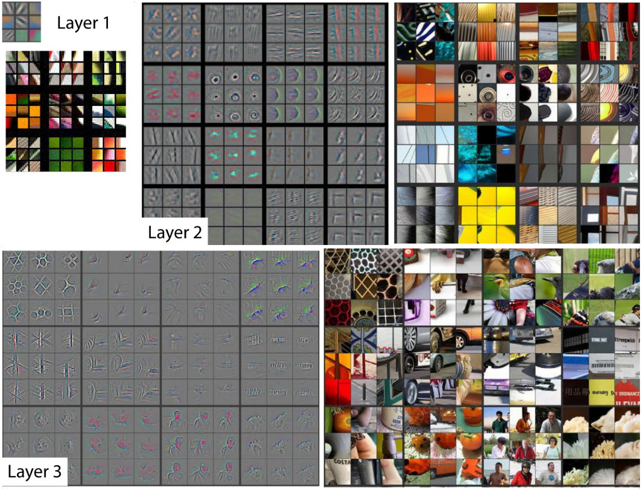
VGG19, max feature response, deconvolution



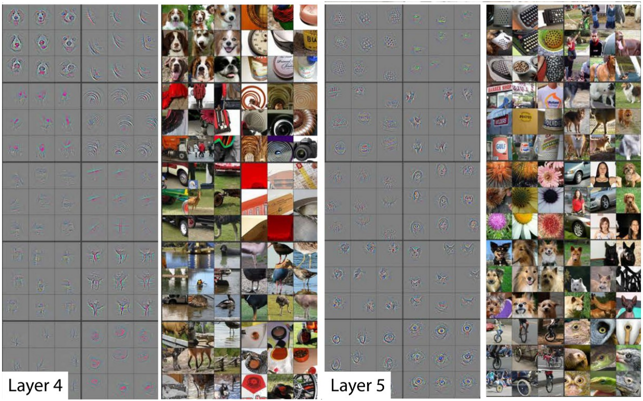
Original images

VGG19, max feature response, guided back-propagation

Experiments with an AlexNet-like network. Original images + deconvolution (or filters) for the top-9 activations for channels picked randomly.



(Zeiler and Fergus, 2014)



(Zeiler and Fergus, 2014)

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 w.r.t. individual channels.

Formally, let $k \in \{1, \dots, C\}$ 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

$$\alpha_k^c = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \frac{\partial y^c}{\partial A_{i,j}^k}.$$

And the final localization map is

$$L_{\text{Grad-CAM}}^c = \text{ReLU}\left(\sum_{k=1}^{C} \alpha_k^c A^k \right).$$

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)
  )
)
```

To implement Grad-CAM, first define hooks to store the feature maps in the forward pass, and the gradient w.r.t. them in the backward:

```
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]
```

To implement Grad-CAM, first define hooks to store the feature maps in the forward pass, and the gradient w.r.t. them in the backward:

```
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:

```
model = torchvision.models.vgg19(weights = 'IMAGENET1K_V1')
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:

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

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:

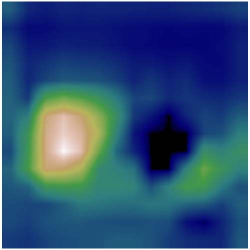
```
to_tensor = torchvision.transforms.ToTensor()
input = to_tensor(PIL.Image.open('example_images/elephant_hippo.png')).unsqueeze(0)
```

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

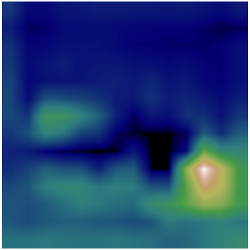
```
output = model(input)

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))
```

Save it as a resized colored heat-map:

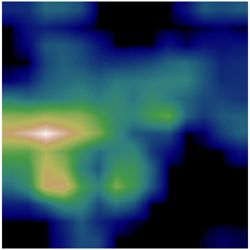
```
L = L / L.max()
L = F.interpolate(L, size = (input.size(2), input.size(3)),
                  mode = 'bilinear', align_corners = False)

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



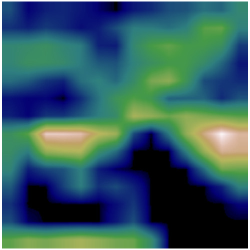
African elephant



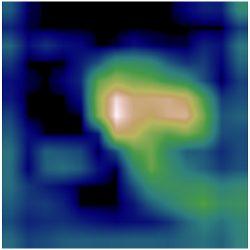
Hippopotamus



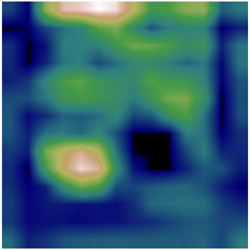
Ox



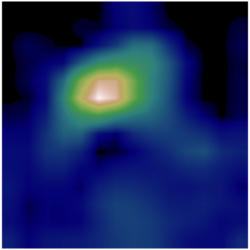
Fountain



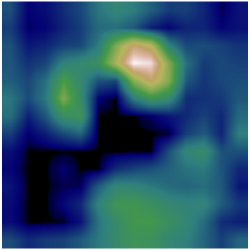
Coffee mug



Bagel



Bee



Daisy

The end

References

D. Erhan, Y. Bengio, A. Courville, and P. Vincent. **Visualizing higher-layer features of a deep network**. Technical Report 1341, Departement IRO, Université de Montréal, 2009.

R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. **Grad-cam: Visual explanations from deep networks via gradient-based localization**. CoRR, abs/1610.02391, 2016.

K. Simonyan, A. Vedaldi, and A. Zisserman. **Deep inside convolutional networks: Visualising image classification models and saliency maps**. CoRR, abs/1312.6034, 2013.

D. Smilkov, N. Thorat, B. Kim, F. Viegas, and M. Wattenberg. **Smoothgrad: removing noise by adding noise**. CoRR, abs/1706.03825, 2017.

J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. **Striving for simplicity: The all convolutional net**. CoRR, abs/1412.6806, 2014.

M. D. Zeiler and R. Fergus. **Visualizing and understanding convolutional networks**. In European Conference on Computer Vision (ECCV), 2014.