dl-assignment-12

December 8, 2023

1 Deep Learning — Assignment 12

Assignment for week 12 of the 2023 Deep Learning course (NWI-IMC070) of the Radboud University.

Names: Daan Brugmans, Maximilian Pohl

Group: 31

Instructions: * Fill in your names and the name of your group. * Answer the questions and complete the code where necessary. * Keep your answers brief, one or two sentences is usually enough. * Re-run the whole notebook before you submit your work. * Save the notebook as a PDF and submit that in Brightspace together with the `.ipynb` notebook file. * The easiest way to make a PDF of your notebook is via File > Print Preview and then use your browser's print option to print to PDF.

1.1 Objectives

In this assignment you will 1. Implement GradCAM 2. Investigate the important features for certain inputs. 3. Ingestigate variants of GradCAM

1.2 Required software

As before you will need these libraries: * `torch` and `torchvision` for PyTorch, * `d2l`, the library that comes with Dive into deep learning book.

All libraries can be installed with `pip install`.

[1]:
```
%config InlineBackend.figure_formats = ['png']
%matplotlib inline

import os
from d2l import torch as d2l
import PIL
import urllib.request
import numpy as np
from matplotlib import colormaps
import matplotlib.pyplot as plt
```

```
import pandas as pd
import torch
from torch import nn
from torch.nn import functional as F
from torchvision import transforms
from torchvision.transforms.functional import to_pil_image
from torchvision.models import resnet50, ResNet50_Weights
from torchvision.io.image import read_image

device = d2l.try_gpu()
```

1.3 12.1 A pretrained network (4 points)

In this assignment we will be working with a pre-trained network. These are available in torchvision. Look at the documentation for more information on pretrained models.

In this assignment we will use the ResNet50 model.

(a) Load a pretrained ResNet50 model, with default weights

[2]:
```
weights = ResNet50_Weights.DEFAULT
model = resnet50(weights=weights)
model = model.to(device)
model.eval();
```

Next we will download a couple of random images, which we will use for the rest of this notebook.

(b) Run the code below to download the test images

[3]:
```
# Download some images
urls = [
    'http://images.cocodataset.org/test-stuff2017/000000006149.jpg',
    'https://farm8.staticflickr.com/7020/6810252887_01e3d8e4e6_z.jpg',
    'http://images.cocodataset.org/val2017/000000039769.jpg',
    'http://images.cocodataset.org/train2017/000000140285.jpg',
    # Some more images to play around with
    #'https://farm1.staticflickr.com/35/67223286_72fce12163_z.jpg',
    #'https://farm4.staticflickr.com/3587/3508226585_268a2e5b9b_z.jpg',
    #'https://farm1.staticflickr.com/6/6947166_ba80959bbb_z.jpg',
    #'https://farm3.staticflickr.com/2325/5821134041_b96fef0946_z.jpg',
    #'https://farm1.staticflickr.com/115/296513905_ddb2cf6438_z.jpg',
]


def load_url(url):
    filename = os.path.basename(url)
    if not os.path.exists(filename):
        urllib.request.urlretrieve(url, filename)
    return read_image(filename)
```

```
images = [load_url(url) for url in urls]
```

The code below shows these images.

[4]:
```
plt.figure(figsize=(15, 15))
for i, image in enumerate(images):
    plt.subplot(1, len(images), i + 1)
    plt.imshow(transforms.ToPILImage()(image))
    plt.axis('off')
plt.tight_layout()
```

The pretrained model was trained with a specific image preprocessing. So we should use the same:

[5]:
```
preprocess = weights.transforms()
```

(c) Preprocess the images, and store them in a single tensor. (1 point)

Hint: `torch.stack` can be useful.

Note: you may get a warning about parameters of transform, you can ignore it.

[6]:
```
x = torch.stack([preprocess(image) for image in images])

# Verify that the batch has the right shape
assert (x.shape == torch.Size([len(images), 3, 224, 224]))
```

```
/home/max/DataspellProjects/ru-deep-learning-23-24/venv/lib/python3.11/site-
packages/torchvision/transforms/functional.py:1603: UserWarning: The default
value of the antialias parameter of all the resizing transforms (Resize(),
RandomResizedCrop(), etc.) will change from None to True in v0.17, in order to
be consistent across the PIL and Tensor backends. To suppress this warning,
directly pass antialias=True (recommended, future default), antialias=None
(current default, which means False for Tensors and True for PIL), or
antialias=False (only works on Tensors - PIL will still use antialiasing). This
also applies if you are using the inference transforms from the models weights:
update the call to weights.transforms(antialias=True).
  warnings.warn(
```

It is also useful to be able to undo the preprocessing, at least the normalization and the dimension permutation, so that we can plot the preprocessed images as well.

[7]:
```
def unnormalize(x):
    """Make a preprocessed image showable."""
    x = torch.permute(x, [1, 2, 0])  # Make the color channels the last␣
 ↪dimension
    x = x * torch.tensor(preprocess.std)[None, None, :]
    x = x + torch.tensor(preprocess.mean)[None, None, :]
    return x
```

(d) Plot the pre-processed images

[8]:
```
plt.figure(figsize=(15, 15))
for i, image in enumerate(x):
    plt.subplot(1, len(x), i + 1)
    plt.imshow(unnormalize(image.cpu()))
    plt.axis('off')
plt.tight_layout()
```

(e) Compare the preprocessed images to the original. How does the default preprocessing make all the images the same size?

It crops the images to the middle

(f) Run the classifier on the batch of images. (1 point)

[9]:
```
x = x.to(device)
y = model(x[[0]])
```

The classifier predicts a logit scores, represing the likelihood of 1000 classes.

You can use the `torch.topk` function to get a list of the predicted labels with the highest score. This function returns a tuple with two elements, `result.indices` is the indices, and `result.values` contains the correspondig scores.

By themselves, these indices don't tell you what the predicted class actually is. For that you can use the following function:

[10]:
```
def category(index):
    """Return the textual category for some predictions"""
    return np.array(weights.meta["categories"])[index]
```

(g) Create a table with the textual label for the top 5 predicted labels for each image. (1 point)

Hint: `pd.DataFrame` is an easy way of producing a nice looking table.

[11]:
```
# TODO: make a table with the label of the top 5 predictions
values, indices = torch.topk(y, 5)
# print(indices)
category(indices.cpu().detach().numpy())
```

[11]:
```
array([['racket', 'tennis ball', 'ping-pong ball', 'dugong',
        'volleyball']], dtype='<U30')
```

(h) Are there any strange labels in the top 5 predictions for any of the images? (1 point)

Yes, for example, the "dugong" for the first image and the broccoli, warthog, in the last.

1.4 12.2 Hooking into the network (4 points)

To understand and visualize which part of the images are 'causing' the predicted labels, we can use GradCAM.

GradCAM needs to know the activations and output gradients of the last layer in the model, just before the global pooling layer. The activations are computed during forward propagation (`model.forward()`), and the gradients are computed when calling `y.backward()`, but neither of them are saved anywhere.

To capture the activations and gradients we need to use hooks, which allow us to register a function that gets called every time a module's `forward` or `backward` function is called.

Take a look at `torch.Module.register_forward_hook` and `torch.Module.register_full_backward_hook`.

(a) Complete the code below. (2 points)

You can store the activations and gradients as a member variable of the layer, or you can use a global variable.

[12]:
```
def add_hooks(layer):
    """
    Add hooks to a layer that store the activations (the output of the layer)␣
 ↪in the forward pass,
    as well as the gradient wrt. the output in the backward pass
    """

    def forward_hook(layer, args, output):
        # This function will be called after the forward pass.
```

```
        #   args are the inputs of the layer
        #   output is the output of the layer
        # TODO: store activations
        # pass
        layer.activations = output

    def backward_hook(layer, grad_input, grad_output):
        # This function is called after the backward pass
        #   grad_input is a *tuple* that contains the gradients wrt the inputs.
        #   grad_output is a *tuple* the gradient the gradients wrt the outputs.
        # TODO: store gradients
        layer.gradients = grad_output[0]
        # pass

    # Remove old hooks (if any)
    remove_hooks(layer)

    # Register new hooks
    layer.forward_hook = layer.register_forward_hook(forward_hook)
    layer.backward_hook = layer.register_full_backward_hook(backward_hook)


def remove_hooks(layer):
    if hasattr(layer, 'forward_hook'):
        layer.forward_hook.remove()
    if hasattr(layer, 'backward_hook'):
        layer.backward_hook.remove()
```

(b) Which layer of the ResNet50 model should the hooks be applied to? (1 point)

Hint: you can `print` a model, and you can access a named layer using `model.layer`

[13]:
```
print(model)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
```

```
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
```

```
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
```

```
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

[14]:
```
hooked_layer = model.layer4
```

[15]:
```
remove_hooks(hooked_layer)
```

(c) Add the hooks to the layer. (no points)

[16]:
```
add_hooks(hooked_layer)
```

(d) Verify that the hooks work, by completing and running the code below. (1 point)

[17]:
```
y = model(x[[0]])  # TODO: run model forward on the first image, x[[0]]
top_class = torch.argmax(y, dim=1)  # TODO: get the top class with torch.argmax
print(top_class)
print(y.shape)

# TODO: run model backward for the logit of top predicted class
y[:, top_class].backward()
print(y.shape)

layer_activations = model.layer4.activations  # TODO: get stored activations
print(layer_activations.shape)
layer_gradients = model.layer4.gradients  # TODO: get stored gradients
print(layer_gradients.shape)

assert layer_activations.shape[1:] == torch.Size([2048, 7,
                                                  7]), "Activations has the␣
 ↪wrong shape. Maybe you did not add hooks to the last layer before global␣
 ↪pooling."
assert layer_gradients.shape == layer_activations.shape, "Gradients have the␣
 ↪wrong shape. Make sure to use the output gradients, and note that␣
 ↪grad_output is a tuple."
assert torch.mean(layer_activations) != 0, "Activations should not be zero"
assert torch.mean(layer_gradients) != 0, "Gradients should not be zero"
assert torch.mean(layer_activations - layer_gradients) != 0, "Gradients should␣
 ↪not be the same as the activations"
assert top_class == 752, "Did you use the right image"
```

```
tensor([752], device='cuda:0')
torch.Size([1, 1000])
torch.Size([1, 1000])
torch.Size([1, 2048, 7, 7])
torch.Size([1, 2048, 7, 7])
```

1.5 12.3 GradCAM (9 points)

Now we are ready to implement GradCAM.

(a) GradCAM uses 'neuron importance weights', based on the gradients. Compute these importance weights. (1 point)

See equation (1) of the paper.

[18]:
```
importance_weights = torch.mean(model.layer4.gradients, dim=[0, 2, 3])

assert importance_weights.squeeze().shape == torch.Size([2048])
```

(b) Combine the importance weights with the activations to get the gradcam map. (2 point)

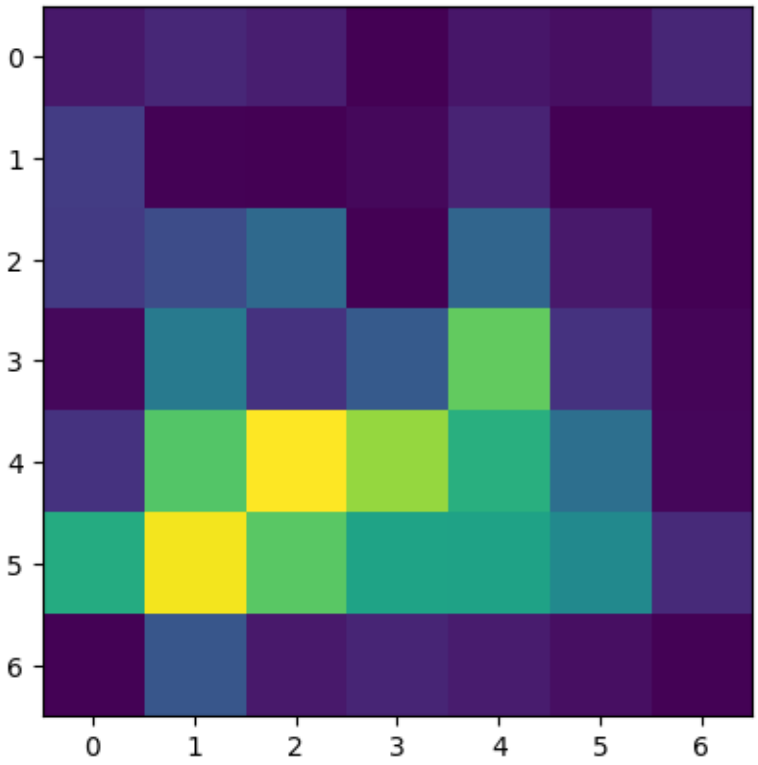
See equation (2) of the paper.

Hint: To combine two tensors they need to have the same shape. You can make them line up by adding new dimensions to a tensor using `tensor[:,None,:,None]`, where `None` is a new dimension, while `:` is an existing dimension.

[19]:
```
for i in range(model.layer4.activations.size()[1]):
    model.layer4.activations[:, i, :, :] *= importance_weights[i]

gradcam_map = torch.mean(model.layer4.activations, dim=1).squeeze()

gradcam_map = F.relu(gradcam_map)

assert gradcam_map.shape == torch.Size([7, 7])
assert torch.min(gradcam_map) >= 0
```

(c) Plot the gradcam map.

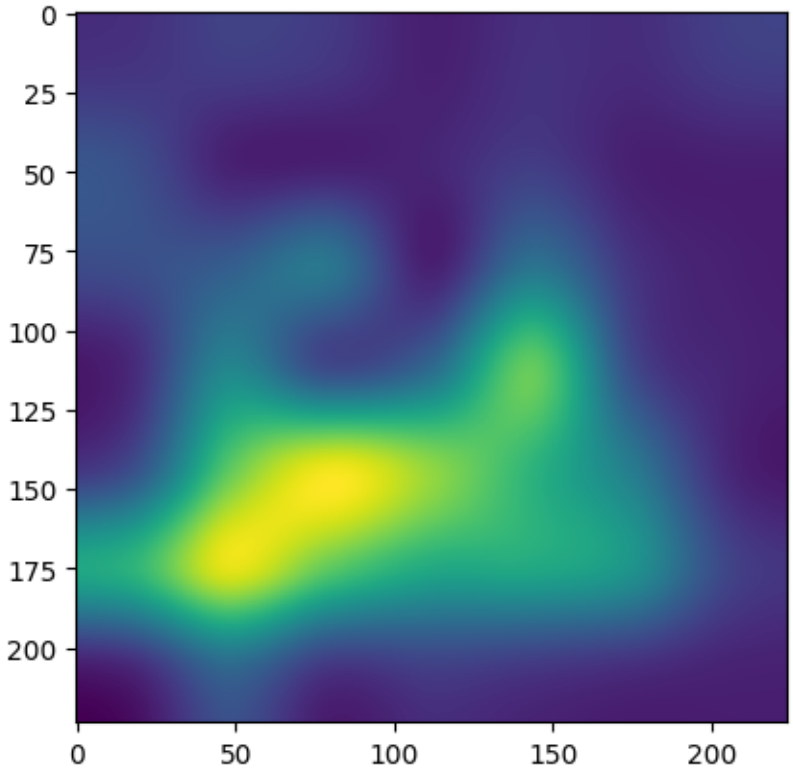
[20]:
```
plt.imshow(gradcam_map.detach().cpu());
```



We can resize the map using interplation to make it smoother, and to align it with the input image.

(d) Plot the resized and smoothed GradCAM map.

[21]:
```
def resize_map(map, size=(224, 224)):
    return np.asarray(to_pil_image(map.detach().cpu(), 'F').resize(size))


plt.imshow(resize_map(gradcam_map));
```



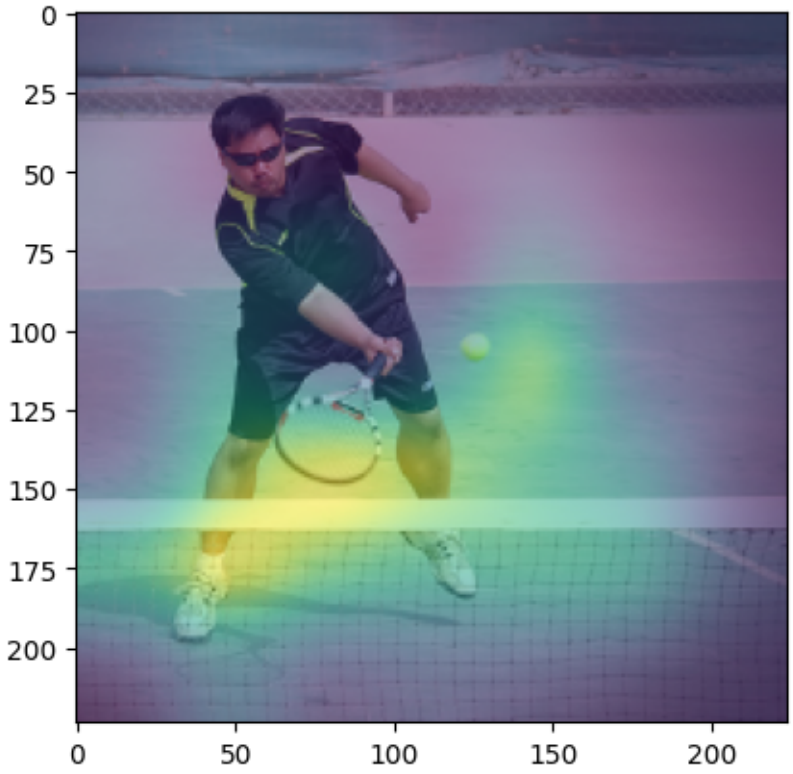
(e) Overlay the gradcam map over the input image. (1 point)

You can overlay two images by drawing the second one semi-transparent with `alpha=0.5`.

Expected output: the highlighted region should correspond to the predicted label. Although the alignment might be slightly off.

[22]:
```
plt.imshow(unnormalize(x[0].cpu()))
plt.imshow(resize_map(gradcam_map), alpha=0.5)
```

[22]:
```
<matplotlib.image.AxesImage at 0x7f742e35a050>
```



We needed several steps to compute a gradCAM visualisation. Let's clean up the code by encapsulating it in a function.

(f) Complete the code below. (1 point)

Hint: the model always works on a batch of images, you can turn a single image into a batch by adding a new dimension, `image[None]`.

[23]:
```
def gradcam(image, index):
    """
    Compute a GradCAM map for the given input image, and class index.
    """
    y = model.forward(image[None])
    y[:, index].backward()

    importance_weights = torch.mean(model.layer4.gradients, dim=[0, 2, 3])

    for i in range(model.layer4.activations.size()[1]):
        model.layer4.activations[:, i, :, :] *= importance_weights[i]

    gradcam_map = torch.mean(model.layer4.activations, dim=1).squeeze()

    return F.relu(gradcam_map)
```

```
assert torch.any(gradcam(x[0], 0) != gradcam(x[1], 0)), "Are you using the␣
 ↪incorrect global variables?"
assert torch.all(gradcam(x[0], 0) == gradcam(x[0],
                                             0)), "Calling the function twice␣
 ↪gives different results, perhaps some gradients are not cleared?"
```

(g) Complete the code below to create a class activation map for the top 5 label for each of the images. (2 point)

[24]:
```
plt.figure(figsize=(12, 12))
for i, image in enumerate(x):
    y = model.forward(image[None])
    values, top5_indices = torch.topk(y, 5)
    for j, index in enumerate(top5_indices[0]):
        plt.subplot(len(images), 5, i * 5 + j + 1)
        # TODO: compute and plot gradcam map, overlayed on the image
        gradcam_map = gradcam(image, index)
        plt.imshow(unnormalize(image.cpu()))
        plt.imshow(resize_map(gradcam_map), alpha=0.5)
        plt.title(category(index))
        plt.axis('off')
plt.tight_layout()
```

(h) Do the gradcam maps make sense? Do the highlighted areas roughly correspond to the location of objects in the image? Mention specific examples. (1 point)

Yes, it does. For example, the tennis ball in the second image and also the remote control in the third image are located quite accurately. Even broccoli prediction makes a bit of sense, as it is looking at the tree.

(i) Do the highlighted areas correspond *exactly* to the location of objects in the image? Why / why not? (1 point)

No, they don't. Especially, they do not cover the whole object, as for example, the cat, which is recognized by just a little part of the whole cat. But also the tennis ball seems to be a bit off.

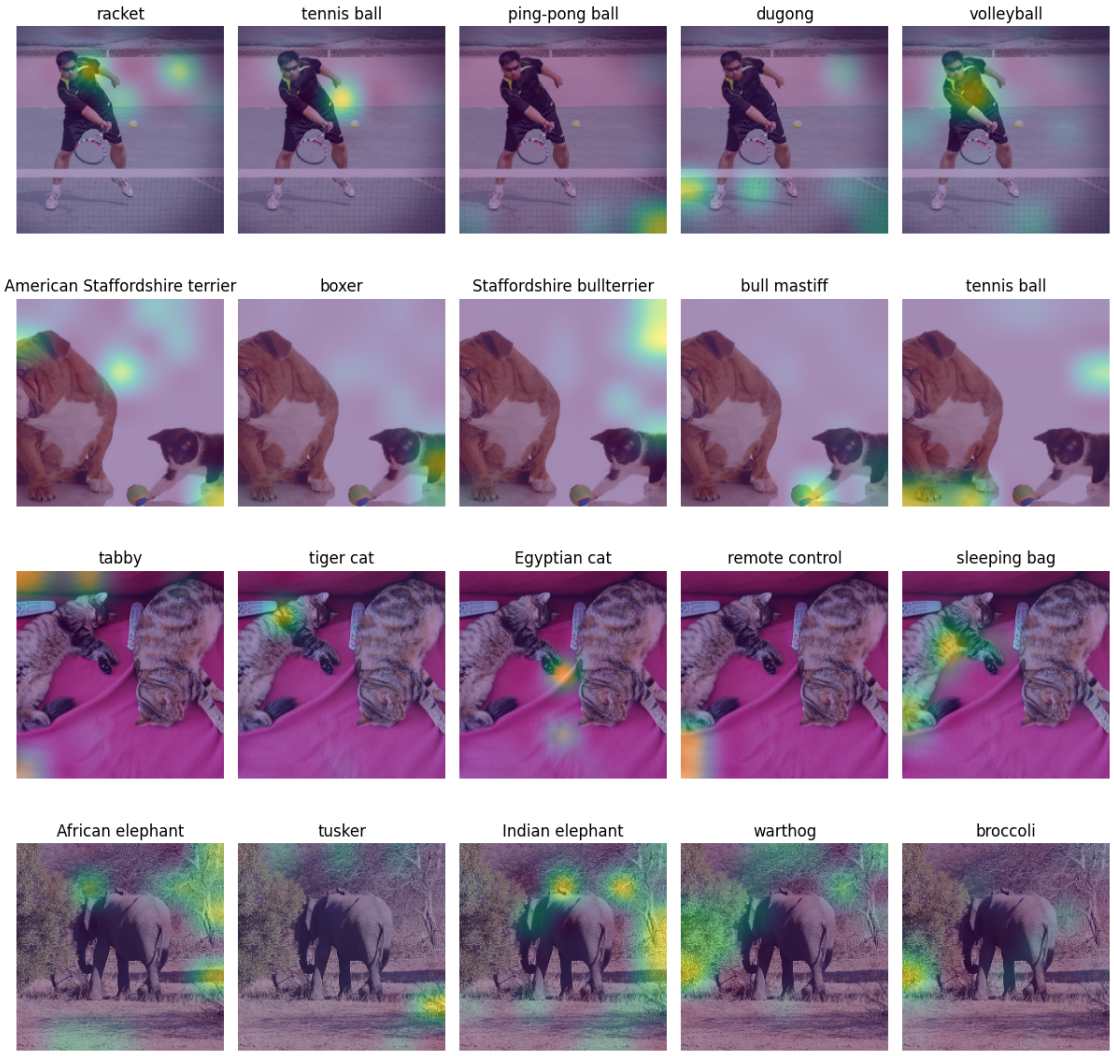
A partial explanation could be that the overlay is just not aligned perfectly, e.g., by the smoothing. The other examination is that the network does not take the whole object into account when doing predictions, but just the most remarkable parts.

1.6 12.4 Variants (7 points)

GradCAM includes a non-linear activation function. The result is that the GradCAM map that is produced only contains positive values. But perhaps the negative values also contain useful information?

(a) Copy and modify the code from part 12.3f and g, by negating the input to the non-linearity. (1 point)

[25]:
```
def gradcam_negative(image, index):
    # TODO: see 12.3f, but add negation
    y = model.forward(image[None])
    y[:, index].backward()

    importance_weights = torch.mean(model.layer4.gradients, dim=[0, 2, 3])

    for i in range(model.layer4.activations.size()[1]):
        model.layer4.activations[:, i, :, :] *= importance_weights[i]

    gradcam_map = torch.mean(model.layer4.activations, dim=1).squeeze()

    return F.relu(-gradcam_map)


# TODO: see 12.3g
plt.figure(figsize=(12, 12))
for i, image in enumerate(x):
    y = model.forward(image[None])
    values, top5_indices = torch.topk(y, 5)
    for j, index in enumerate(top5_indices[0]):
        plt.subplot(len(images), 5, i * 5 + j + 1)
        # TODO: compute and plot gradcam map, overlayed on the image
        gradcam_map = gradcam_negative(image, index)
        plt.imshow(unnormalize(image.cpu()))
        plt.imshow(resize_map(gradcam_map), alpha=0.5)
        plt.title(category(index))
        plt.axis('off')
plt.tight_layout()

assert torch.all((gradcam(x[0], 123) > 0) != (gradcam_negative(x[0],
                                                               123) > 0)), "The␣
 ↪negative and positive GradCAM can not be active in the same locations of the␣
 ↪image"
```



(b) What are the areas that are highlighted now? (1 point)

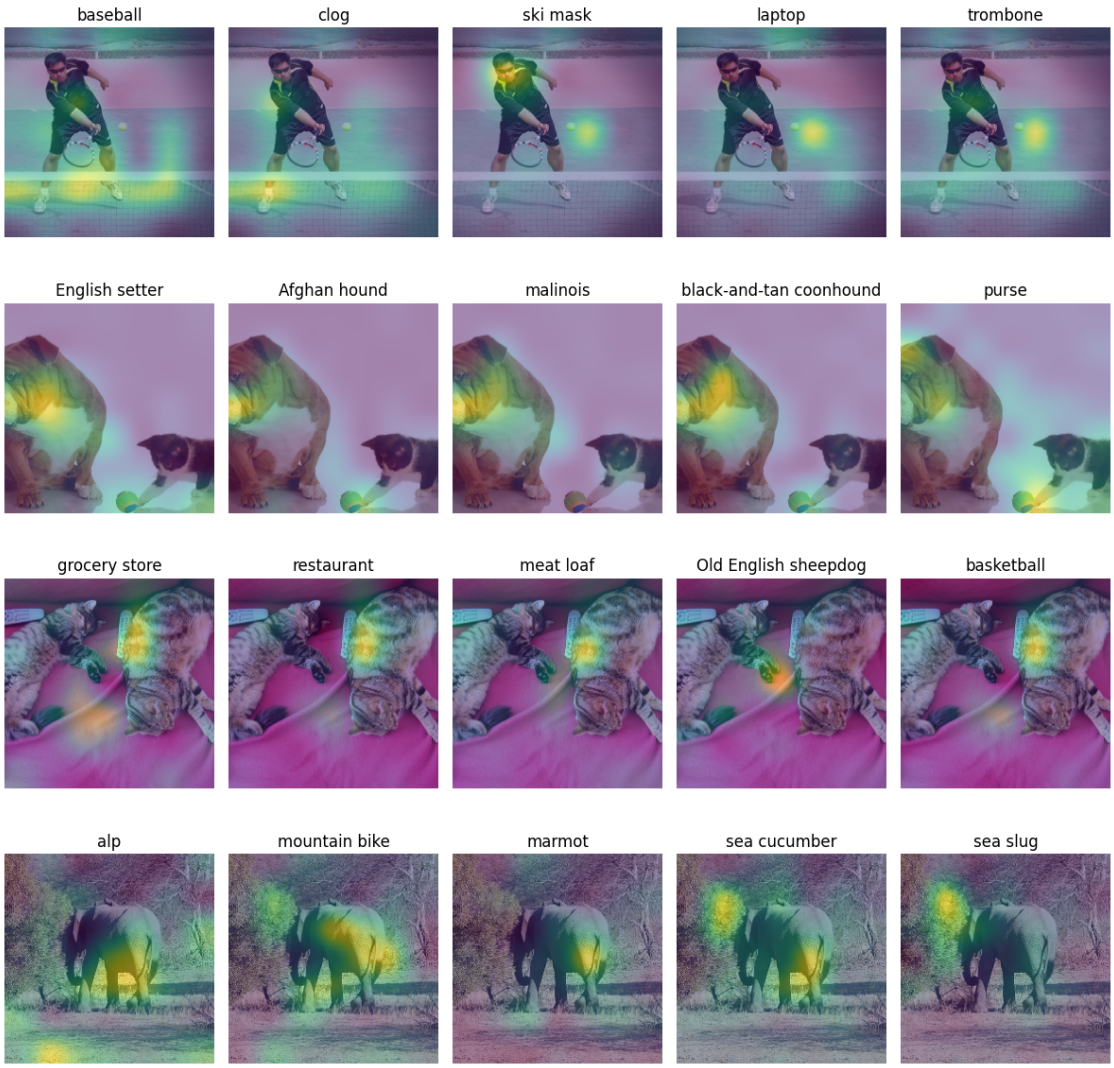
The areas where the network pays the least attention to. So, hopefully, areas outside the object of interest.

(c) Plot the negative gradcam map for the bottom 5 predictions instead of the top 5. (1 point)

Hint: the bottom 5 of `x` is the top 5 of `-x`.

[26]:
```
plt.figure(figsize=(12, 12))
for i, image in enumerate(x):
    y = model.forward(image[None])
    values, top5_indices = torch.topk(-y, 5)
    for j, index in enumerate(top5_indices[0]):
        plt.subplot(len(images), 5, i * 5 + j + 1)
```

```
        # TODO: compute and plot gradcam map, overlayed on the image
        gradcam_map = gradcam_negative(image, index)
        plt.imshow(unnormalize(image.cpu()))
        plt.imshow(resize_map(gradcam_map), alpha=0.5)
        plt.title(category(index))
        plt.axis('off')
plt.tight_layout()
```



(d) What can you learn from these maps? Give at least 1 concrete example. (1 point)

The network actually pays the most negative attention to the areas where an object would be expected if it was in the image. For example, the clog and the ski mask would be in the highlighted parts of the image if they were there.

Instead of looking at the last layer before global pooling, we could in theory also apply GradCAM at a different layer of the network. Perhaps that can give a higher resolution map?

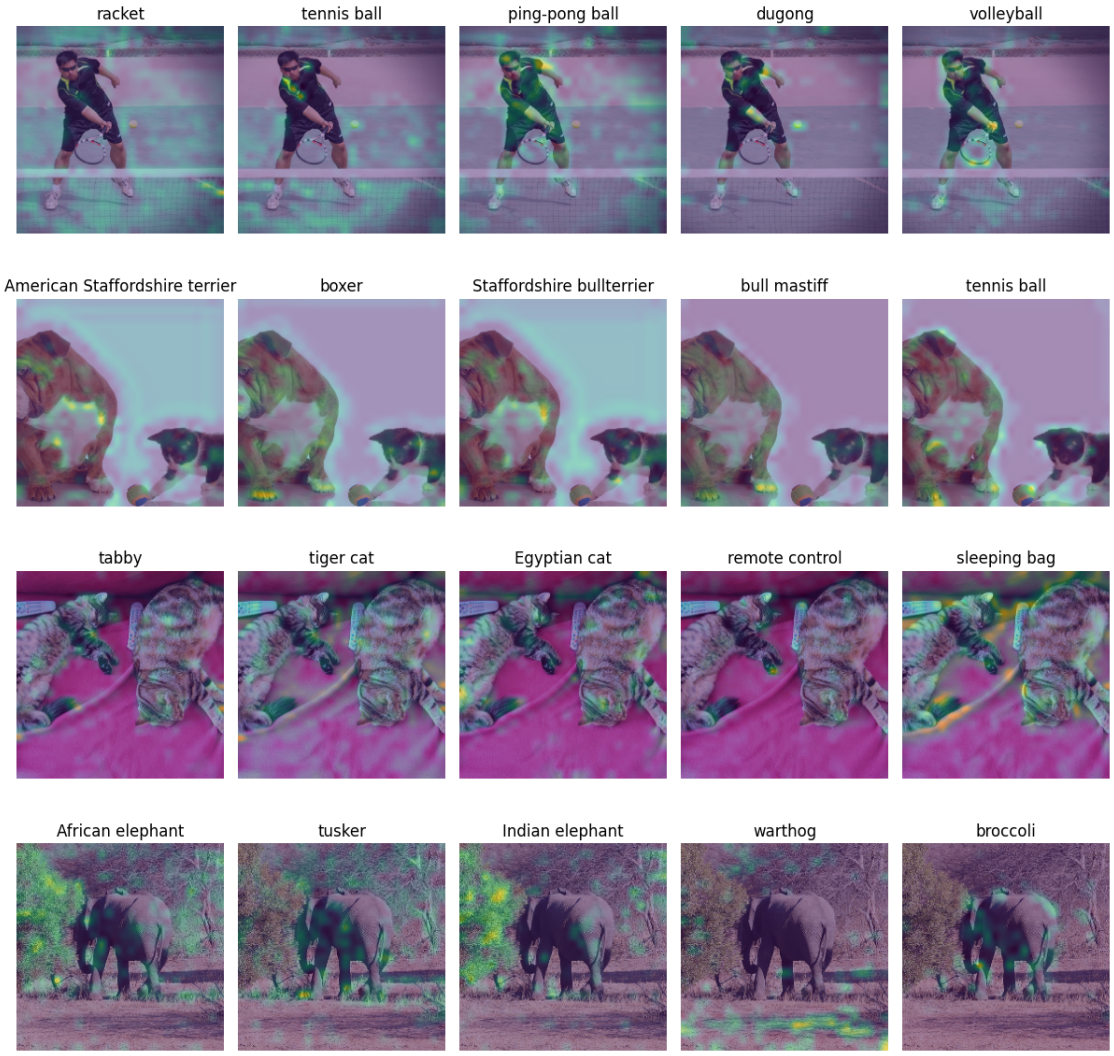
(e) Adapt the gradcam code to take the model and layer of the model as parameters (1 point)

Note: you may need to add hooks at this point.

[27]:
```
def gradcam_at(model, layer, image, index):
    """
    Compute a GradCAM map at the given layer of the given model
    """
    # TODO: see 12.3f
    add_hooks(layer)

    y = model.forward(image[None])
    y[:, index].backward()

    importance_weights = torch.mean(layer.gradients, dim=[0, 2, 3])

    for i in range(layer.activations.size()[1]):
        layer.activations[:, i, :, :] *= importance_weights[i]

    gradcam_map = torch.mean(layer.activations, dim=1).squeeze()

    return F.relu(gradcam_map)
```

(f) Now use gradcam at an intermediate layer of the model (1 point)

[28]:
```
# TODO: see 12.3g
plt.figure(figsize=(12, 12))
for i, image in enumerate(x):
    y = model.forward(image[None])
    values, top5_indices = torch.topk(y, 5)
    for j, index in enumerate(top5_indices[0]):
        plt.subplot(len(images), 5, i * 5 + j + 1)
        # TODO: compute and plot gradcam map, overlayed on the image
        gradcam_map = gradcam_at(model, model.layer2, image, index)
        plt.imshow(unnormalize(image.cpu()))
        plt.imshow(resize_map(gradcam_map), alpha=0.5)
        plt.title(category(index))
        plt.axis('off')
plt.tight_layout()

assert gradcam_map.shape != torch.Size([7, 7]), "Use a different layer"
```



(g) Do the gradcam maps for this earlier layer give a useful visualization? (1 point)

No, these visualizations are not very helpful in explaining the decision made by the network. The most you may be able to see is how the network recognizes edges, for example.

1.7 12.5 Of cats and dogs (5 points)

For the second image (`images[1]`, the one with the white background), there is an object that is clearly in the image, but it does not appear in the top 5 classes.

(a) Why is this object not detected in the top 5? (1 point)

Because there are a lot of other things in the image that the network can also detect, which apparently result in a higher activation score. Additionally, there are a lot of different classes for dogs, so the network first tries a few different labels for dogs.

(b) How could you reduce the bias of the model for the second image, to make it also

detect the other object? (1 point)

We could reduce the number of labels for dogs to match the number of labels for cats. Additionally, we could pay attention during training to have a roughly equal number of images for all animals.

(c) Curiously, for this second image, the bottom 5 labels for this image also include dogs. Why would that happen? (1 point)

One reason probably is that there are a lot of different dog labels, but the network is also able to distinguish dog breeds which look clearly differently. Similar as for the ski mask, it knows where to look for a ski mask in the image (namely on the person's face), looks there, does not find a ski mask, and then determine that the ski mask label is irrelevant.

(d) Create a GradCAM map for the missing object in the image. (1 point)

Hint: The third images contains some labels that you can use. You can also use the following function to get the corresponding class index.

[29]:
```
def find_category(query):
    """Find a category that has or contains the given name."""
    found = [i for (i, k) in enumerate(weights.meta["categories"]) if k.
 ↪find(query) != -1]
    if len(found) == 1:
        return found[0]
    elif len(found) > 1:
        raise Exception("Multiple categories found: " + str(category(found)))
    else:
        raise Exception("No category found that matches: " + str(query))
```

[30]:
```
# TODO: Make a gradcam map for another class
indices = [find_category("tiger cat"), find_category("Persian cat"),␣
 ↪find_category("Siamese cat"),
           find_category("Egyptian cat"), find_category("Madagascar cat"),␣
 ↪find_category("tabby")]

plt.figure(figsize=(20, 4))
for j, index in enumerate(indices):
    plt.subplot(1, 6, j + 1)
    # TODO: compute and plot gradcam map, overlayed on the image
    gradcam_map = gradcam(x[1], index)
    plt.imshow(unnormalize(x[1].cpu()))
    plt.imshow(resize_map(gradcam_map), alpha=0.5)
    plt.title(category(index))
    plt.axis('off')
plt.tight_layout()
```

(e) Are you able to find this object when you go looking for it? Explain your answer briefly. (1 point)

Yes, the model tends to pay attention to the parts that belong to the different cat labels, though it's not always very accurate, probably because the cat breeds given as labels look very different to the cat in our image.

1.8 The end

Well done! Please double check the instructions at the top before you submit your results.

This assignment has 29 points. Version 3770c63 / 2023-12-04