dl-assignment-13

December 18, 2024

1 Deep Learning — Assignment 13

Thirteenth and last assignment for the 2024 Deep Learning course (NWI-IMC070) of the Radboud University.

Names: Andrew Schroeder and Fynn Gerding

Group: 17

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,

All libraries can be installed with `pip install`.

[2]:
```
%config InlineBackend.figure_formats = ['png']
%matplotlib inline

import os
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

# Use the GPU if available
def detect_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    elif torch.backends.mps.is_available():
        return torch.device("mps")
    else:
        return torch.device("cpu")
device = detect_device()
device = torch.device("cpu")
```

1.3 13.1 A pretrained network (5 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

[3]:
```
weights = ResNet50_Weights.DEFAULT
model = resnet50(weights)
model = model.to(device)
model.eval();
```

```
/home/andrew/Documents/School/Semester 3/DL/.venv/lib/python3.12/site-
packages/torchvision/models/_utils.py:135: UserWarning: Using 'weights' as
positional parameter(s) is deprecated since 0.13 and may be removed in the
future. Please use keyword parameter(s) instead.
  warnings.warn(
```

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

[4]:
```
# 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',
    #'https://farm8.staticflickr.com/7020/6810252887_01e3d8e4e6_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]
```

[5]:
```
# Workaround as the above import does not work on my machine
import pickle as pkl
images = pkl.load(open('images.pkl', 'rb'))
```

The code below shows these images.

[6]:
```
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:

[7]:
```
preprocess = weights.transforms()
print(preprocess)
```

```
ImageClassification(
    crop_size=[224]
    resize_size=[232]
    mean=[0.485, 0.456, 0.406]
    std=[0.229, 0.224, 0.225]
    interpolation=InterpolationMode.BILINEAR
```

)

(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.

[8]:
```
print(images[0].shape)
x = [preprocess(image) for image in images]
x = torch.stack(x)

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

```
torch.Size([3, 334, 500])
```

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.

[9]:
```
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

[10]:
```
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? (1 point)

By adjusting the length and width of the images (with cropping) to a standard size we normalize

all the images to the same size. The image is first resized to 232 pixels on the shorter dimension before the cropping occurs. We then also normalize the image values on the RGB channels to have a standard deviation and mean obtained from the ImageNet dataset.

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

[11]:
```
y = model(x.to(device))
print(y)
```

```
tensor([[-0.1245,  0.0798,  0.1946,  …,  0.0789, -0.3600,  0.2640],
        [-0.3463, -0.0927,  0.0869,  …, -0.4557,  0.0465,  0.5772],
        [-0.1091, -0.6737,  0.1082,  …, -0.3684, -0.8687, -0.3512],
        [-0.0776,  0.0835, -0.1889,  …, -0.2195, -0.3764,  0.0807]],
       grad_fn=<AddmmBackward0>)
```

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:

[12]:
```
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.

[13]:
```
# TODO: make a table with the label of the top 5 predictions
top5 = torch.topk(y, 5).indices.cpu().numpy()
pd.DataFrame([[category(i) for i in top5[j]] for j in range(len(images))],
             columns=[f"Category {i+1}" for i in range(top5.shape[1])],
             index=[f"Image {i+1}" for i in range(top5.shape[0])])
```

[13]:

	Category 1	Category 2	Category 3	Category 4	Category 5
Image 1	racket	tennis ball	ping-pong ball	scuba diver	dugong
Image 2	boxer	American Staffordshire terrier	Staffordshire bullterrier	bull mastiff	tennis ball
Image 3	tabby	tiger cat	Egyptian cat	remote control	pillow
Image 4	tusker	African elephant	Indian elephant	warthog	zebra

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

Yes the fifth label for the first image is strange given that the image is a tennis player and the label is a dugong which a marine mammal. Additionally the fifth label for the second image is a tennis ball when the image is dog.

1.4 13.2 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.

[14]:
```
def add_hooks(layer, activations, gradients):
    """
    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):
        activations.append(output.detach().clone())

    def backward_hook(layer, grad_input, grad_output):
        gradients.append(grad_output[0].detach().clone())

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

    # Register new hooks
    layer.register_forward_hook(forward_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`

[15]:
```
hooked_layer = model.layer4 # layer just before global pooling
```

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

[16]:
```
activations = []
gradients = []
add_hooks(hooked_layer, activations, gradients)
```

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

[17]:
```
y = model(x[[0]])
top_class = torch.argmax(y)
y[0, top_class].backward()
layer_activations = torch.stack(activations).squeeze(0)
layer_gradients = torch.stack(gradients).squeeze(0)

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"
```

1.5 13.3 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 = layer_gradients.mean(dim=(2, 3)).squeeze()

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.

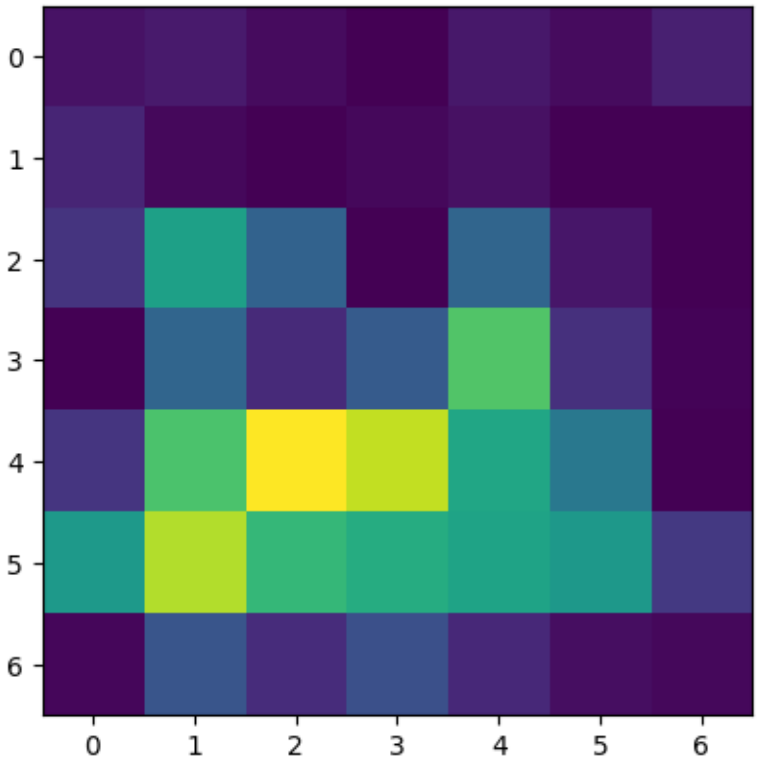
Hint: To combine two tensors they need to have the same shape. You can make them line up by adding new dimensions to a tensor using `tensor[:,None,:,None]`, where `None` is a new dimension, while `:` is an existing dimension.

[19]:
```
print(importance_weights.size())
print(layer_activations.size())

gradcam_map = F.relu(torch.sum(importance_weights[:, None, None] *␣
 ↪layer_activations.squeeze(), dim=0))

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

```
torch.Size([2048])
torch.Size([1, 2048, 7, 7])
```

(c) Plot the gradcam map.

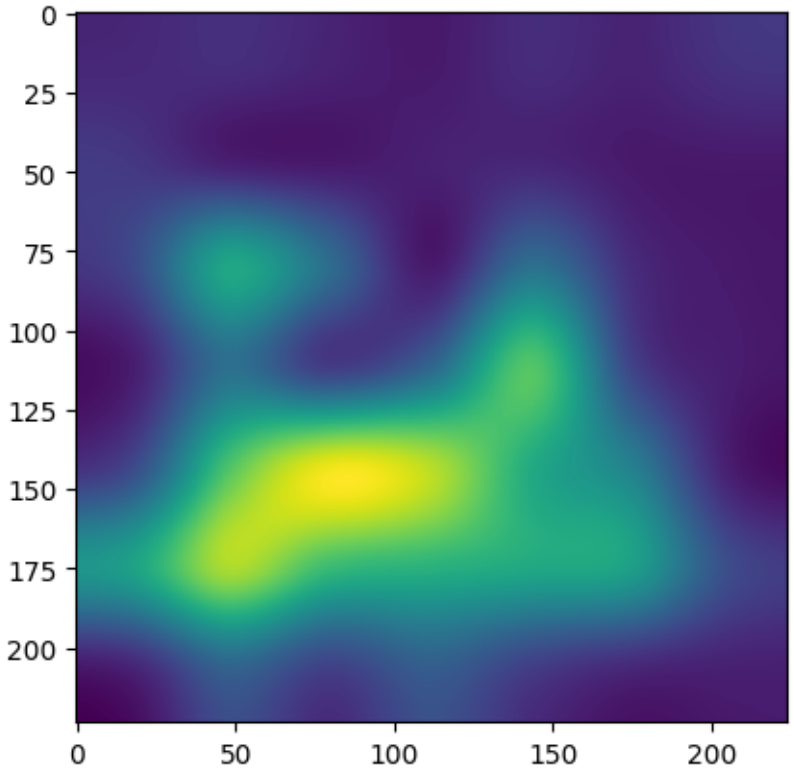
[20]:
```
plt.imshow(gradcam_map.detach().cpu());
```



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));
```

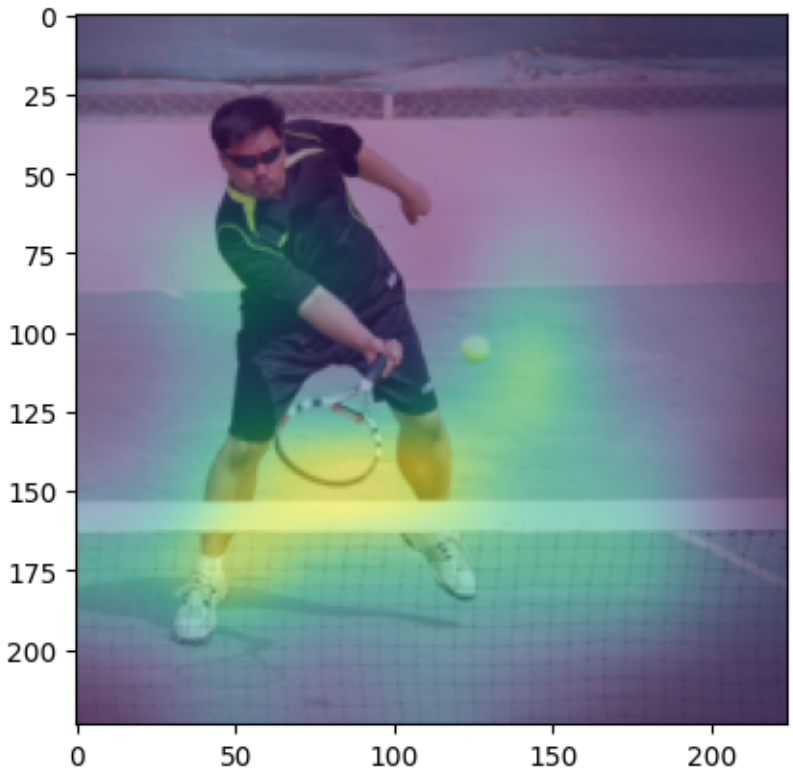


(e) Overlay the gradcam map over the input image. (1 point)

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);
```



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)

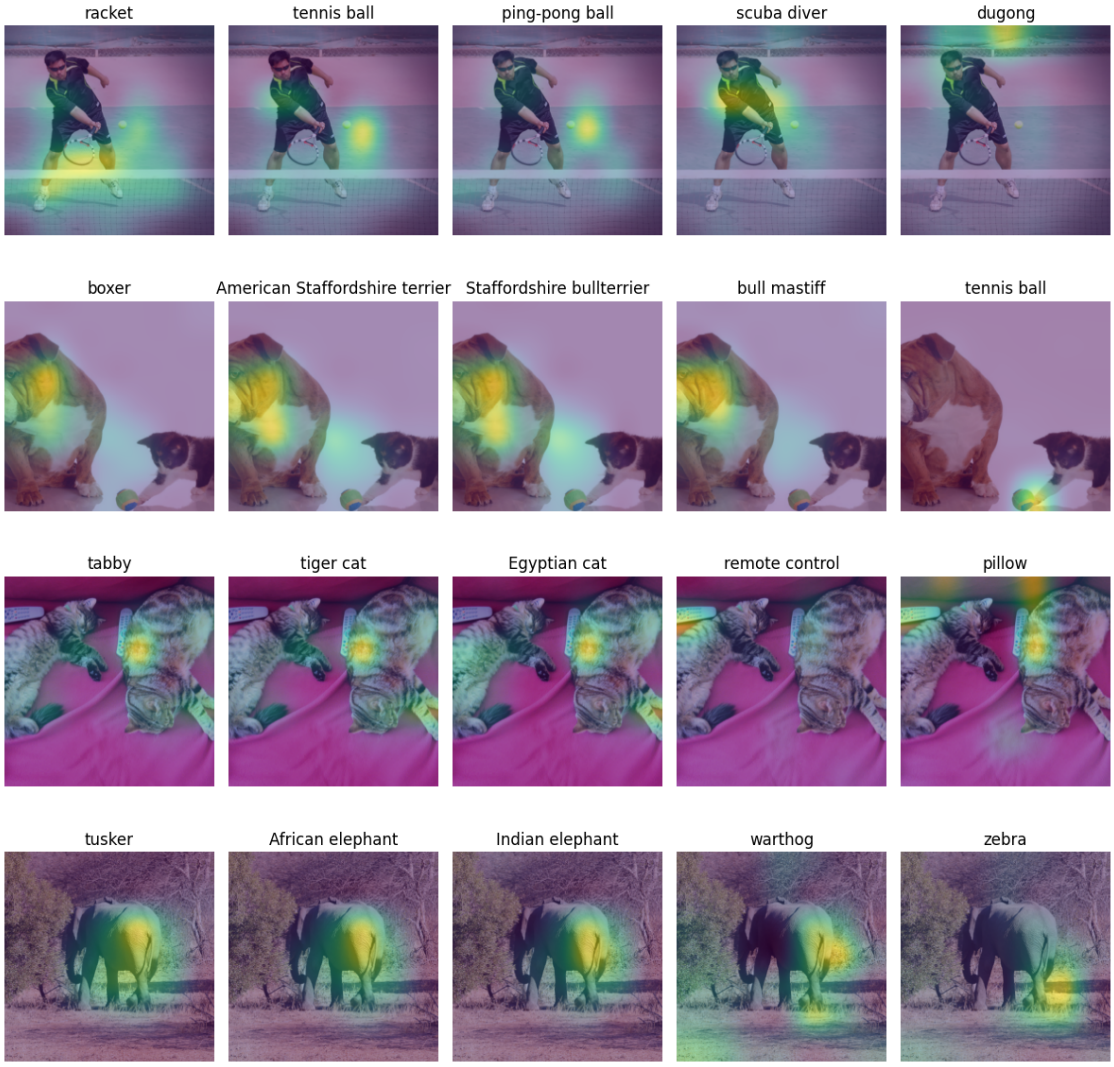
Hint: the model always works on a batch of images, you can turn a single image into a batch by adding a new dimension, `image[None]`.

[23]:
```
def gradcam(image, index):
    """
    Compute a GradCAM map for the given input image, and class index.
    """
    activations = []
    gradients = []

    target_layer = model.layer4
    add_hooks(target_layer, activations, gradients)
    model.zero_grad()
    image = image.unsqueeze(0)
    y = model(image)
    score = y[0, index]
    score.backward()
```

```
    layer_activations = torch.stack(activations).squeeze(0)
    layer_gradients = torch.stack(gradients).squeeze(0)

    importance_weights = layer_gradients.mean(dim=(2, 3)).squeeze(0)

    gradcam_map = F.relu(torch.sum(importance_weights[:, None, None] *␣
 ↪layer_activations.squeeze(), dim=0))

    remove_hooks(target_layer)
    return 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):
    top5_indices = torch.topk(model(image.unsqueeze(0).to(device)), 5).indices.
 ↪cpu().numpy().squeeze()
    for j, index in enumerate(top5_indices):
        plt.subplot(len(images),5, i*5 + j + 1)
        # TODO: compute and plot gradcam map, overlayed on the image
        plt.imshow(unnormalize(image.cpu()))
        plt.imshow(resize_map(gradcam(image, index)), alpha=0.5)
        # TODO: set title to label of the class index
        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)

Most of the highlights do make sense, but some like the zebra, or pillow are not very intuitive. It seems like there a pattern or environmental queue were used. A example of the gradcam working very well is the tennis ball in the row of the boxer.

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

Not exactly, as the receptive field may not always be perfectly aligned with the object (low-resolution feature maps) and the gradcam seems to focus on the most informative regions rather than the entire object.

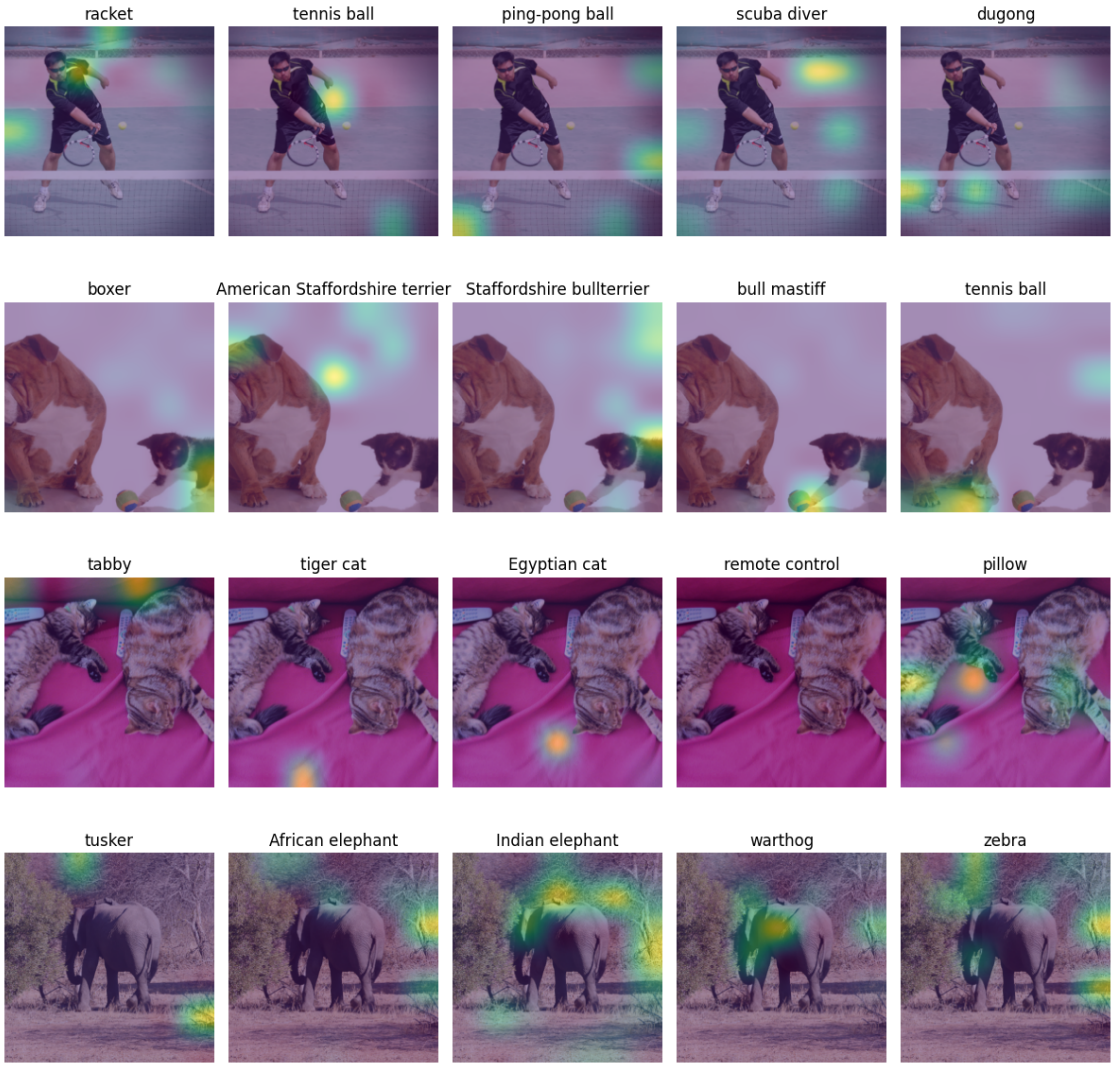
1.6 13.4 Variants (7 points)

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 13.3f and g, by negating the input to the non-linearity. (1 point)

[26]:
```
def gradcam_negative(image, index):
    activations = []
    gradients = []

    target_layer = model.layer4
    add_hooks(target_layer, activations, gradients)
    model.zero_grad()
    image = image.unsqueeze(0)
    y = model(image)
    score = y[0, index]
    score.backward()

    layer_activations = torch.stack(activations).squeeze(0)
    layer_gradients = torch.stack(gradients).squeeze(0)

    importance_weights = layer_gradients.mean(dim=(2, 3)).squeeze(0)

    gradcam_map = F.relu(-torch.sum(importance_weights[:, None, None] *␣
 ↪layer_activations.squeeze(), dim=0))

    remove_hooks(target_layer)
    return gradcam_map

# TODO: see 13.3g
plt.figure(figsize=(12,12))
for i, image in enumerate(x):
    top5_indices = torch.topk(model(image.unsqueeze(0).to(device)), 5).indices.
 ↪cpu().numpy().squeeze()
    for j, index in enumerate(top5_indices):
        plt.subplot(len(images),5, i*5 + j + 1)
        # TODO: compute and plot gradcam map, overlayed on the image
        plt.imshow(unnormalize(image.cpu()))
        plt.imshow(resize_map(gradcam_negative(image, index)), alpha=0.5)
        # TODO: set title to label of the class index
        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)

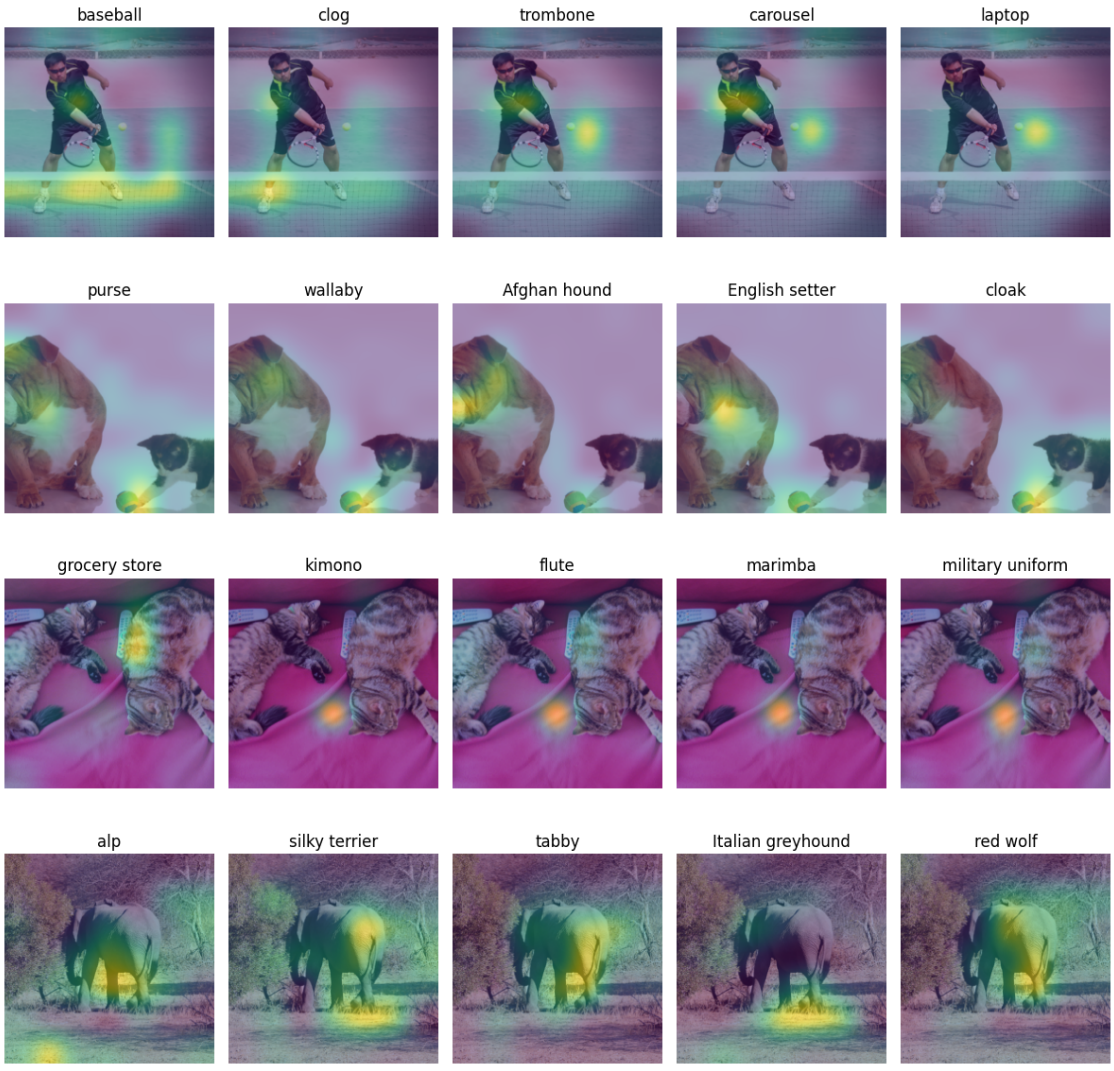
These highlighted areas are those that are the least informative when it comes to selecting the appropriate label for the image.

(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`.

[27]:
```
# TODO: see 13.3g
plt.figure(figsize=(12,12))
```

```
for i, image in enumerate(x):
    top5_indices = torch.topk(-model(image.unsqueeze(0).to(device)), 5).indices.
 ↪cpu().numpy().squeeze()
    for j, index in enumerate(top5_indices):
        plt.subplot(len(images),5, i*5 + j + 1)
        # TODO: compute and plot gradcam map, overlayed on the image
        plt.imshow(unnormalize(image.cpu()))
        plt.imshow(resize_map(gradcam_negative(image, index)), alpha=0.5)
        # TODO: set title to label of the class index
        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 negative gradcam for the bottom 5 predictions shows which features of the image are used to justify the low ranking assigned to the label. In a sense they are those part of the image that argue why the label is not a valid label. For instance, in the first set of images, the label is "baseball", and the model seems to be looking at the tennis net when deciding it is not baseball, given that baseball does not use nets.

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)

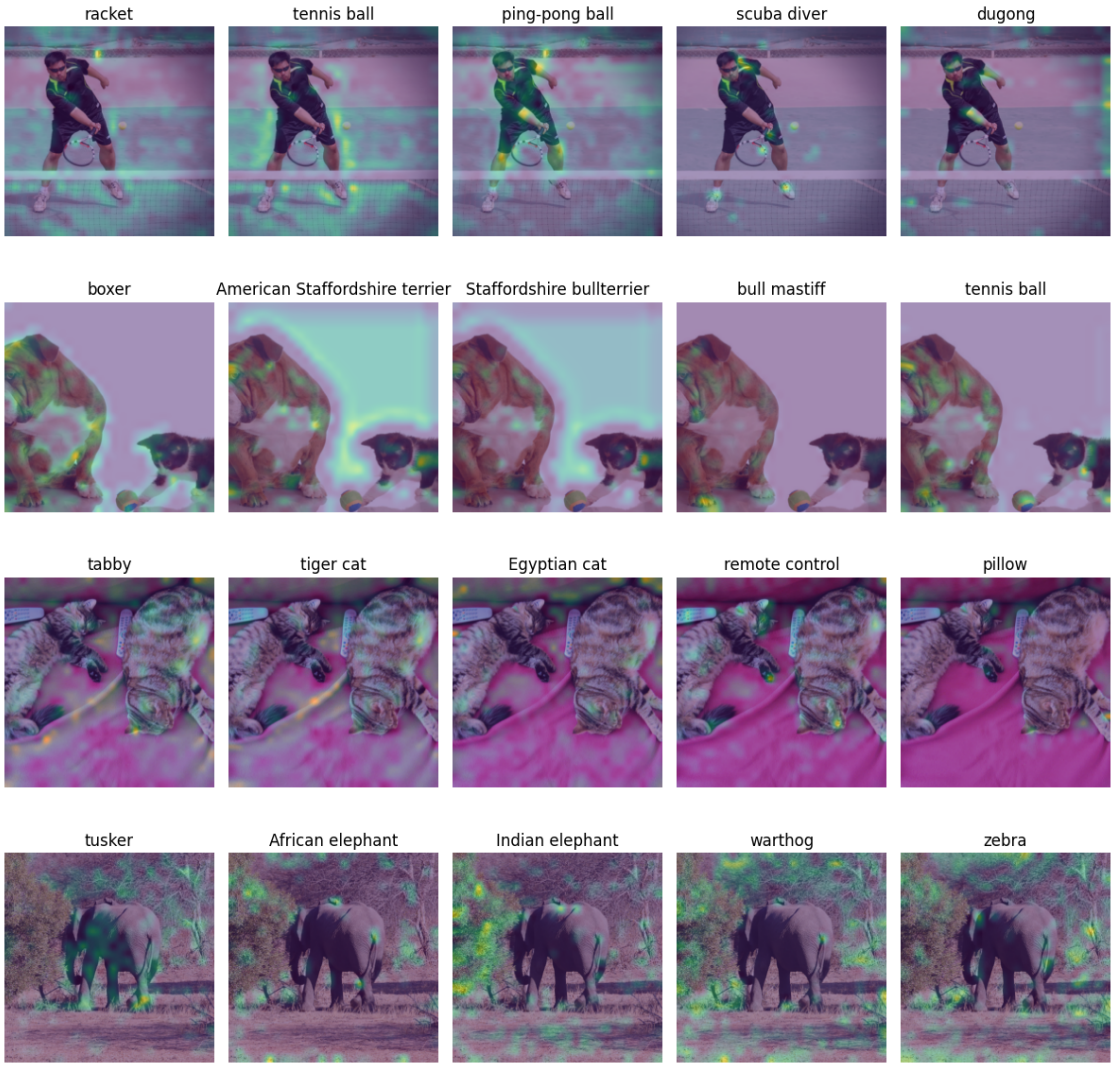
Note: you may need to add hooks at this point.

[28]:
```
def gradcam_at(model, layer, image, index):
    """
    Compute a GradCAM map at the given layer of the given model
    """
    activations = []
    gradients = []

    add_hooks(layer, activations, gradients)
    model.zero_grad()
    image = image.unsqueeze(0)
    y = model(image)
    score = y[0, index]
    score.backward()

    layer_activations = torch.stack(activations).squeeze(0)
    layer_gradients = torch.stack(gradients).squeeze(0)

    importance_weights = layer_gradients.mean(dim=(2, 3)).squeeze(0)

    gradcam_map = F.relu(torch.sum(importance_weights[:, None, None] *␣
 ↪layer_activations.squeeze(), dim=0))

    remove_hooks(layer)
    return gradcam_map
```

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

[29]:
```
# TODO: see 13.3g
plt.figure(figsize=(12,12))
for i, image in enumerate(x):
    top5_indices = torch.topk(model(image.unsqueeze(0).to(device)), 5).indices.
 ↪cpu().numpy().squeeze()
    for j, index in enumerate(top5_indices):
        plt.subplot(len(images),5, i*5 + j + 1)
        # TODO: compute and plot gradcam map, overlayed on the image
```

```
        plt.imshow(unnormalize(image.cpu()))
        gradcam_map = gradcam_at(model, model.layer2, image, index)
        plt.imshow(resize_map(gradcam_map), alpha=0.5)
        # TODO: set title to label of the class index
        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)

The gradcams at earlier layers seem to be less informative than the gradcam at the final layer before pooling - these earlier layers have much more distributed attention, whereas the later layers

have more focused attention.

1.7 13.5 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)

The cat does not appear in the top 5 classes - the model seems to focus attention on the larger dog or the simple but brightly colored tennis ball but neglects the cat.. though it is not obvious why this is. It could be because there are far more dogs/tennis balls in the dataset compared to cats. Upon further investigation, there seem to be far fewer variety of cats in the dataset. Using the find_category method when searching dog results in 10 varieties, compared to cat which resulted in 4 categories. This specific cat type in the image may not be present in the training set.

(b) How could you reduce the bias of the model for the second image, to make it also detect the other object? (1 point)

You could include a larger variety of cat species and images into the dataset.

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

Different dog species can be quite different. The bottom 5 negative labels represent those regions of the image that are used for arguing against the label. In this case, it looks like the snout of the dog is used to argue that the dog is definitely not an Afghan Hound, because it may know that this is exactly the snout of a Staffordshire. It the model is very good at identifying snouts, it would be very good at identifying when a snout does not belong to a particular dog.

(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.

[30]:
```
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))
```

[44]:
```
#find_category("dog")

cats = ["tiger cat", "Persian cat", "Siamese cat", "Egyptian cat"]
indexes = [find_category(cat) for cat in cats]
```

[43]:
```
plt.figure(figsize=(12,12))

#top5_indices = torch.topk(model(image.unsqueeze(0).to(device)), 5).indices.
 ↪cpu().numpy().squeeze()
top5_indices = find_category("Persian cat")
for j, index in enumerate(indexes):
    plt.subplot(len(images),5, 0*5 + j + 1)
    # TODO: compute and plot gradcam map, overlayed on the image
    plt.imshow(unnormalize(x[1].cpu()))
    plt.imshow(resize_map(gradcam(x[1], index)), alpha=0.5)
    # TODO: set title to label of the class index
    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, when we explicitly look for a cat by providing the indexes the model seems to look directly at the paws and pointy ears of the cat to identify it. Even though this cat species might not show up that much in the dataset, when we force the model to look for it, it can still identify common cat features like ears and paws.

1.8 The end

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

This assignment has 30 points. Version cb53811 / 2024-12-05