TP2_GradCAM

September 6, 2026

Name : **Chloé Habasque** and **Colin Coërchon**

1 Visualization of CNNs: Grad-CAM

- **Objective**: Convolutional Neural Networks are widely used on computer vision. They are powerful for processing grid-like data. However we hardly know how and why they work, due to the lack of decomposability into individually intuitive components. In this assignment, we use Grad-CAM, which highlights the regions of the input image that were important for the neural network prediction.
- NB: if `PIL` is not installed, try `conda install pillow`.
- Computations are light enough to be done on CPU.

[2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models, datasets, transforms
import matplotlib.pyplot as plt
import pickle
import urllib.request

import numpy as np
from PIL import Image

import random

import cv2

import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
```

1.1 Download the Model

We provide you with a model `DenseNet-121`, already pretrained on the `ImageNet` classification dataset. * **ImageNet**: A large dataset of photographs with 1 000 classes. * **DenseNet-121**: A deep architecture for image classification (https://arxiv.org/abs/1608.06993)

[3]:
```
densenet121 = models.densenet121(pretrained=True)
densenet121.eval() # set the model to evaluation model
pass
```

```
c:\Users\colin\anaconda3\envs\Alaska_deep\lib\site-
packages\torchvision\models\_utils.py:208: UserWarning: The parameter
'pretrained' is deprecated since 0.13 and may be removed in the future, please
use 'weights' instead.
  warnings.warn(
c:\Users\colin\anaconda3\envs\Alaska_deep\lib\site-
packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a
weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed
in the future. The current behavior is equivalent to passing
`weights=DenseNet121_Weights.IMAGENET1K_V1`. You can also use
`weights=DenseNet121_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

[4]:
```
classes = pickle.load(urllib.request.urlopen('https://gist.githubusercontent.com/
↪yrevar/6135f1bd8dcf2e0cc683/raw/d133d61a09d7e5a3b36b8c111a8dd5c4b5d560ee/
↪imagenet1000_clsid_to_human.pkl'))

##classes is a dictionary with the name of each class
print(classes)
```

```
{0: 'tench, Tinca tinca', 1: 'goldfish, Carassius auratus', 2: 'great white
shark, white shark, man-eater, man-eating shark, Carcharodon carcharias', 3:
'tiger shark, Galeocerdo cuvieri', 4: 'hammerhead, hammerhead shark', 5:
'electric ray, crampfish, numbfish, torpedo', 6: 'stingray', 7: 'cock', 8:
'hen', 9: 'ostrich, Struthio camelus', 10: 'brambling, Fringilla
montifringilla', 11: 'goldfinch, Carduelis carduelis', 12: 'house finch, linnet,
Carpodacus mexicanus', 13: 'junco, snowbird', 14: 'indigo bunting, indigo finch,
indigo bird, Passerina cyanea', 15: 'robin, American robin, Turdus migratorius',
16: 'bulbul', 17: 'jay', 18: 'magpie', 19: 'chickadee', 20: 'water ouzel,
dipper', 21: 'kite', 22: 'bald eagle, American eagle, Haliaeetus leucocephalus',
23: 'vulture', 24: 'great grey owl, great gray owl, Strix nebulosa', 25:
'European fire salamander, Salamandra salamandra', 26: 'common newt, Triturus
vulgaris', 27: 'eft', 28: 'spotted salamander, Ambystoma maculatum', 29:
'axolotl, mud puppy, Ambystoma mexicanum', 30: 'bullfrog, Rana catesbeiana', ... ,
996: 'hen-of-the-woods, hen of the woods, Polyporus frondosus, Grifola
frondosa', 997: 'bolete', 998: 'ear, spike, capitulum', 999: 'toilet tissue,
toilet paper, bathroom tissue'}
```

1.2 Input Images

We provide you with 20 images from ImageNet (download link on the webpage of the course or download directly using the following command line,). In order to use the pretrained model resnet34, the input image should be normalized using `mean = [0.485, 0.456, 0.406]`, and `std = [0.229, 0.224, 0.225]`, and be resized as `(224, 224)`.

[5]:
```
def preprocess_image(dir_path):
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])
    # Note: If the inverse normalisation is required, apply 1/x to the above␣
 ↪object

    dataset = datasets.ImageFolder(dir_path, transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224), # resize the image to 224x224
            transforms.ToTensor(), # convert numpy.array to tensor
            normalize])) #normalize the tensor

    return (dataset)
```

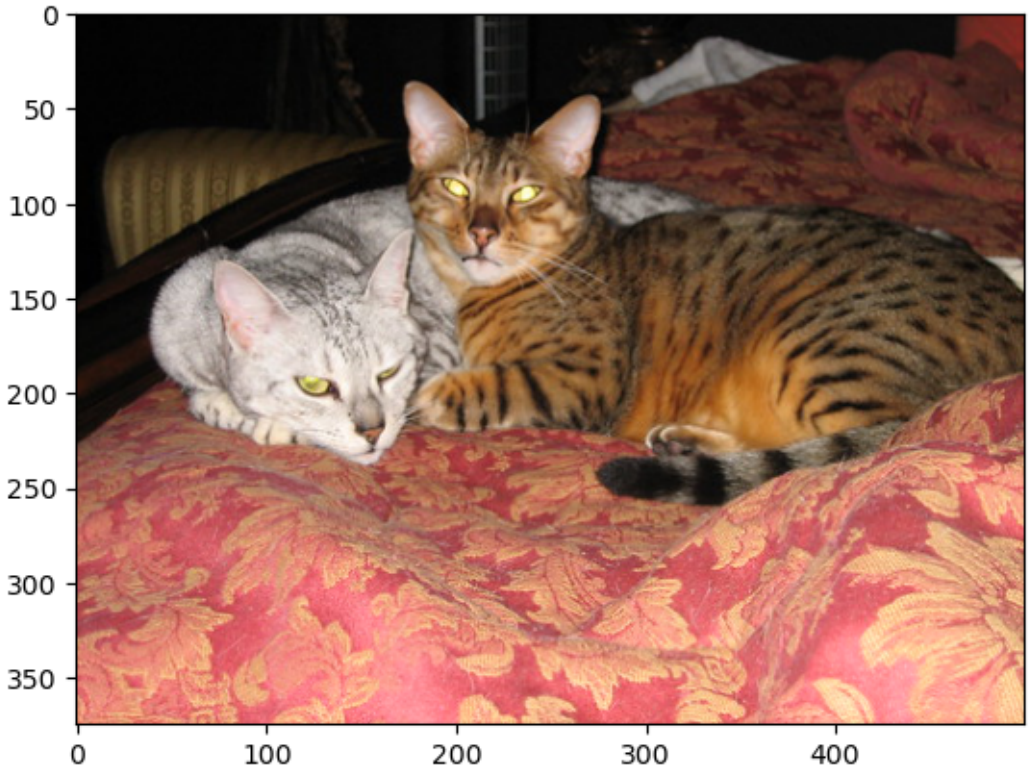
[6]:
```
dir_path = "data"
dataset = preprocess_image(dir_path)
```

[7]:
```
# import os
# if not os.path.exists("data"):
#     os.mkdir("data")
# if not os.path.exists("data/TP2_images"):
#     os.mkdir("data/TP2_images")
#     !cd data/TP2_images && wget "https://www.lri.fr/~gcharpia/deeppractice/
 ↪2025/TP2/TP2_images.zip" && unzip TP2_images.zip

# dir_path = "/content/drive/MyDrive/Master MVA/Advanced Deep Learning/TP2/data/"
# dataset = preprocess_image(dir_path)
```

[8]:
```
# show the orignal image
index = 5
input_image = Image.open(dataset.imgs[index][0]).convert('RGB')
plt.imshow(input_image)
```

[8]:
```
<matplotlib.image.AxesImage at 0x23fbf077e80>
```



[9]:
```
input_image = dataset[index][0].view(1, 3, 224, 224)
output = densenet121(input_image)
values, indices = torch.topk(output, 3)
print("Top 3-classes:", indices[0].numpy(), [classes[x] for x in indices[0].
 ↪numpy()])
print("Raw class scores:", values[0].detach().numpy())
```

```
Top 3-classes: [285 281 282] ['Egyptian cat', 'tabby, tabby cat', 'tiger cat']
Raw class scores: [11.819939  9.971278  9.188527]
```

[10]:
```
type(input_image)
```

[10]:
```
torch.Tensor
```

2 Grad-CAM

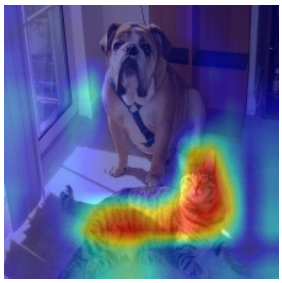
- **Overview:** Given an image, and a category ('tiger cat') as input, we forward-propagate the image through the model to obtain the `raw class scores` before softmax. The gradients are set to zero for all classes except the desired class (tiger cat), which is set to 1. This signal is then backpropagated to the `rectified convolutional feature map` of interest, where we can compute the coarse Grad-CAM localization (blue heatmap).

- **To Do**: Define your own function Grad_CAM to achieve the visualization of the given images. For each image, choose the top-3 possible labels as the desired classes. Compare the heatmaps of the three classes, and conclude.

More precisely, you should provide a function: `show_grad_cam(image: torch.tensor) -> None` that displays something like this: where the heatmap will be correct (here it is just an example) and the first 3 classes are the top-3 predicted classes and the last is the least probable class according to the model.

- **Comment your code**: Your code should be easy to read and follow. Please comment your code, try to use the NumPy Style Python docstrings for your functions.
- **To be submitted within 2 weeks**: this notebook, **cleaned** (i.e. without results, for file size reasons: `menu > kernel > restart and clean`), in a state ready to be executed (with or without GPU) (if one just presses 'Enter' till the end, one should obtain all the results for all images) with a few comments at the end. No additional report, just the notebook!
- **Hints**:
- We need to record the output and grad_output of the feature maps to achieve Grad-CAM. In pytorch, the function `Hook` is defined for this purpose. Read the tutorial of hook carefully.
- More on autograd and hooks
- The pretrained model densenet doesn't have an activation function after its last layer, the output is indeed the `raw class scores`, you can use them directly.
- Your heatmap will have the same size as the feature map. You need to scale up the heatmap to the resized image (224x224, not the original one, before the normalization) for better observation purposes. The function `torch.nn.functional.interpolate` may help.

- Here is the link to the paper: Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization

Class: 'pug, pug-dog'	Class: 'tabby, tabby cat'
	

2.1 Part 1: Grad-CAM implementation

[ ]:
```
class GradCAM:
    """
    Implementation of Grad-CAM for visualizing CNN model activations.
    """
```

```
    def __init__(self, model, target_layer):
        self.model = model
        self.target_layer = target_layer  # the specific layer to analyze
        self.gradients = None
        self.activations = None
        self.hook_layers()

    def hook_layers(self):
        """Adds hooks to capture the activations and gradients from the target␣
↪layer."""
        def forward_hook(module, input, output):
            self.activations = output.detach()

        def backward_hook(module, grad_in, grad_out):
            self.gradients = grad_out[0].detach()

        # hooks from the target layer
        self.target_layer.register_forward_hook(forward_hook)
        self.target_layer.register_backward_hook(backward_hook)

    def generate_cam(self, input_tensor, class_id):
        """
        Generates a Grad-CAM heatmap for a given class.

        Parameters:
        input_tensor (torch.Tensor): Preprocessed image tensor
        class_id (int): Target class index

        Returns:
        numpy.array: Normalized Grad-CAM heatmap
        """
        output = self.model(input_tensor)
        self.model.zero_grad()
        output[:, class_id].backward()

        gradients = self.gradients
        activations = self.activations

        # weights
        weights = gradients.mean(dim=[2, 3], keepdim=True)

        # weighted sum of activations
        cam = (weights * activations).sum(dim=1, keepdim=True)

        # ReLU serve to keep positive influences
        cam = F.relu(cam)
        cam = cam.squeeze().cpu().numpy()
```

```
        # Normalize heatmap between 0 and 1
        cam = 1 - (cam - cam.min()) / (cam.max() - cam.min() + 1e-18)
        return cam
```

2.2 Part 2: Try it on a few (1 to 3) images and comment

[20]:
```
def show_grad_cam(image_tensor, img_path, gradcam):
    """
    Displays the original image and Grad-CAM heatmaps for the top 3 most␣
↪probable classes + least probable class.

    Parameters:
    image_tensor (torch.Tensor): Input image as a tensor
    img_path (str): Path to the input image
    """
    output = densenet121(image_tensor)  # predictions
    values, indices = torch.topk(output, 3)  # top 3 class indices
    top3_classes = indices[0].tolist()
    least_probable_class = output.argmin().item()  # least probable class

    fig, axes = plt.subplots(1, 5, figsize=(25, 5))

    # Display the original image
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    axes[0].imshow(img)
    axes[0].axis("off")
    axes[0].set_title("Original image")

    # Get class labels for the top 3 classes and the least probable class
    class_labels = top3_classes + [least_probable_class]
    class_names = [classes[idx] for idx in class_labels]

    for i, (class_idx, class_name) in enumerate(zip(class_labels, class_names)):
        cam = gradcam.generate_cam(image_tensor, class_idx)  # Generate Grad-CAM␣
↪heatmap
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        cam = cv2.resize(cam, (img.shape[1], img.shape[0]))
        heatmap = cv2.applyColorMap(np.uint8(255 * cam), cv2.COLORMAP_JET)
        overlay_img = cv2.addWeighted(img, 0.5, heatmap, 0.5, 0)  # Overlay␣
↪heatmap on image

        axes[i + 1].imshow(overlay_img)
        axes[i + 1].axis("off")
        axes[i + 1].set_title(f"{class_name}")
```

```
    plt.show()

# Select 5 random images from the dataset
dataset_indices = list(range(len(dataset)))
random.shuffle(dataset_indices)
selected_indices = dataset_indices[:5]

# Initialize Grad-CAM with the model and target layer
gradcam = GradCAM(densenet121, densenet121.features[-1])

# Display Grad-CAM
for idx in selected_indices:
    image_path, _ = dataset.imgs[idx]
    image, _ = dataset[idx]
    image = image.unsqueeze(0)
    show_grad_cam(image, image_path, gradcam)
```





We can notice that, most of the time, the highlighted regions in the first three images are almost the same. It is because it corresponds to the most predicted class, that are very similar (For example : Egyptian cat, tabby cat and tiger cat). In contrast, no regions are highlighted in the image of the least corresponding class.

2.3 Part 3: Try GradCAM on others convolutional layers, describe and comment the results

[13]:
```
print(len(densenet121.features))
```

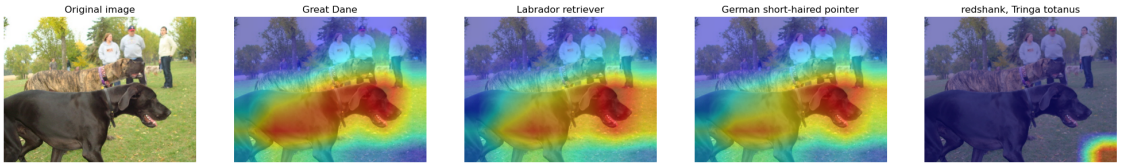
```
12
```

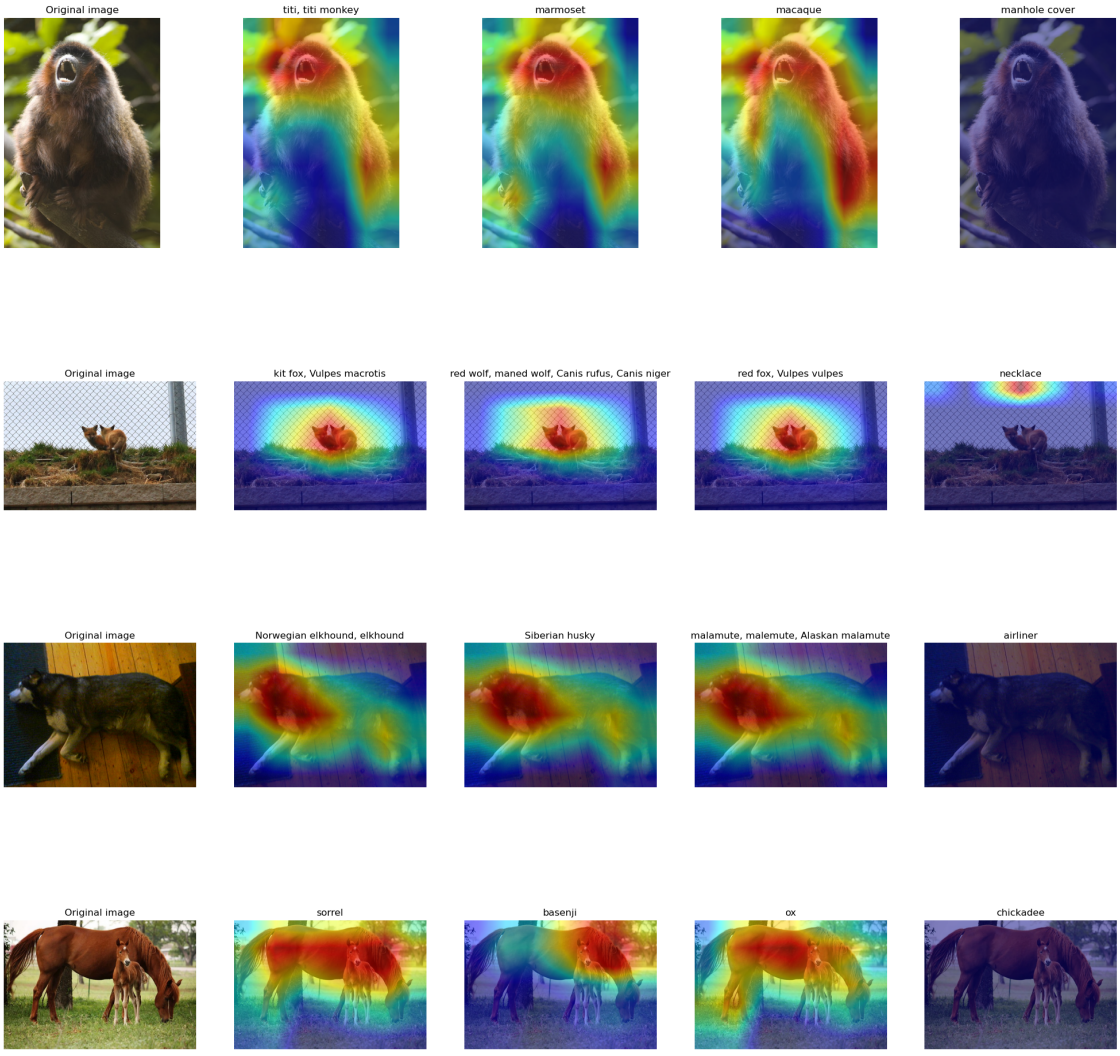
2.3.1 Use of the second to last layer

[21]:
```
# Your code here

# dataset_indices = list(range(len(dataset)))
# random.shuffle(dataset_indices)
# selected_indices = dataset_indices[:5]
gradcam = GradCAM(densenet121, densenet121.features[-2])

for idx in selected_indices:
    image_path, _ = dataset.imgs[idx]
    image, _ = dataset[idx]
    image = image.unsqueeze(0)
    show_grad_cam(image, image_path, gradcam)
```

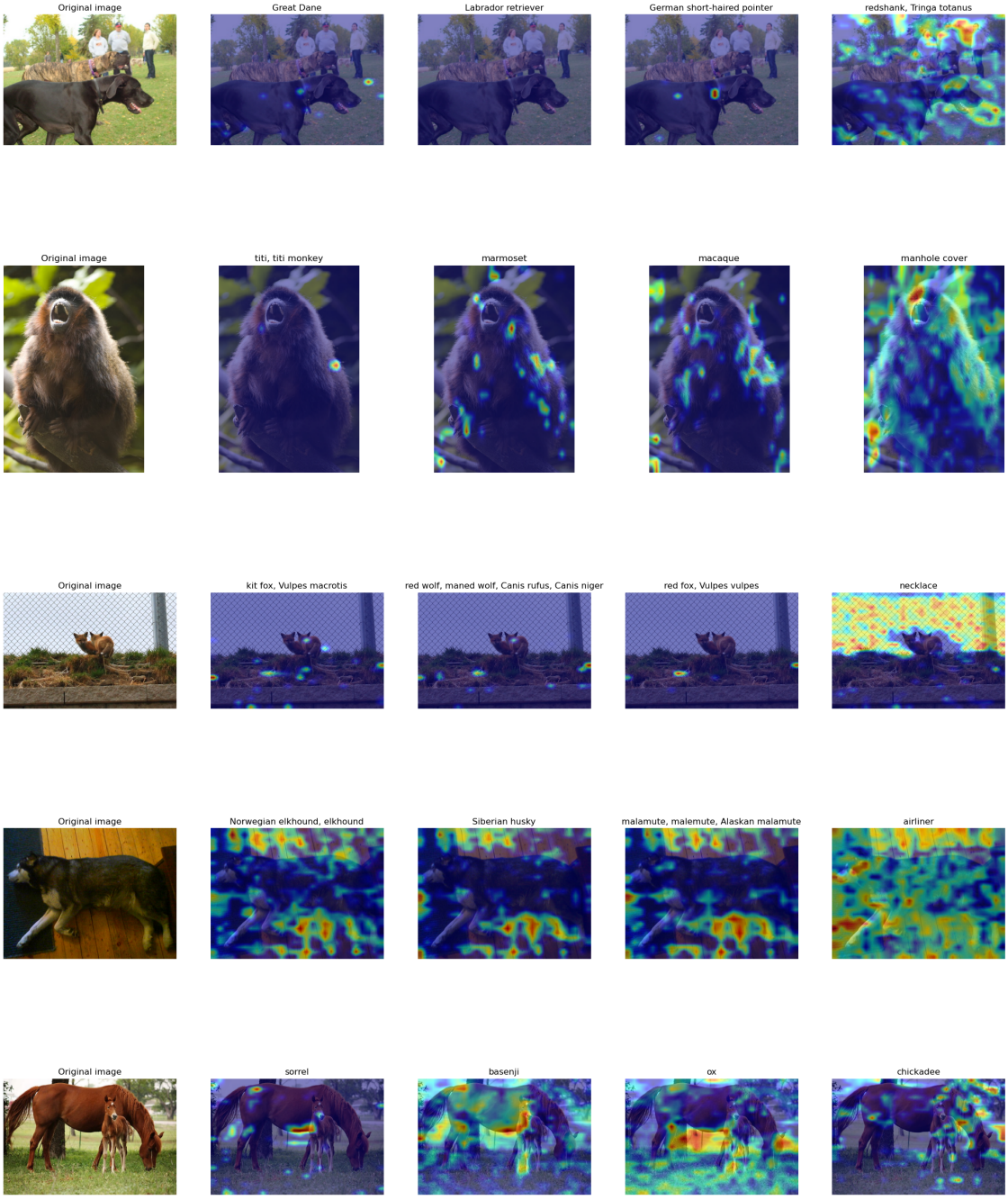




2.3.2 Use of the 5th layer

[22]:
```
# Sélectionner aléatoirement 5 images
# dataset_indices = list(range(len(dataset)))
# random.shuffle(dataset_indices)
# selected_indices = dataset_indices[:5]
gradcam = GradCAM(densenet121, densenet121.features[5])

for idx in selected_indices:
    image_path, _ = dataset.imgs[idx]
    image, _ = dataset[idx]
    image = image.unsqueeze(0)  # Ajouter une dimension batch
    show_grad_cam(image, image_path, gradcam)
```



2.3.3 Use of the first layers

[23]:
```
# Sélectionner aléatoirement 5 images
# dataset_indices = list(range(len(dataset)))
# random.shuffle(dataset_indices)
# selected_indices = dataset_indices[:5]
```

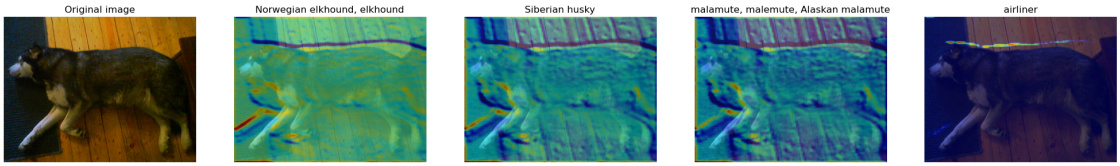
```
gradcam = GradCAM(densenet121, densenet121.features[1])

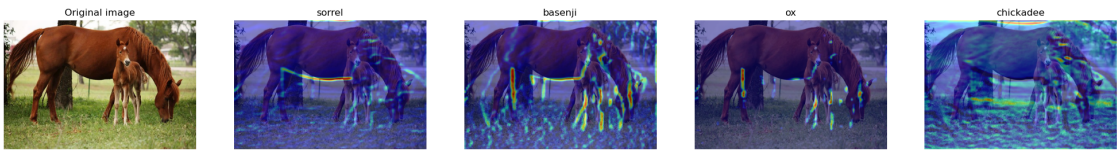
for idx in selected_indices:
    image_path, _ = dataset.imgs[idx]
    image, _ = dataset[idx]
    image = image.unsqueeze(0)
    show_grad_cam(image, image_path, gradcam)
```





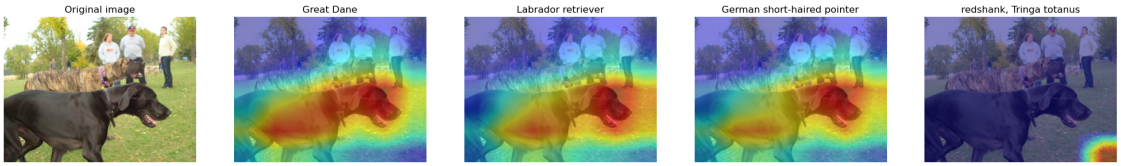




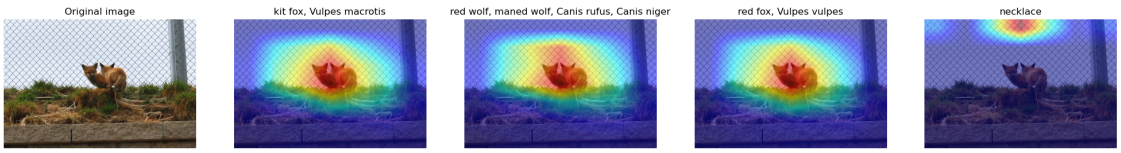


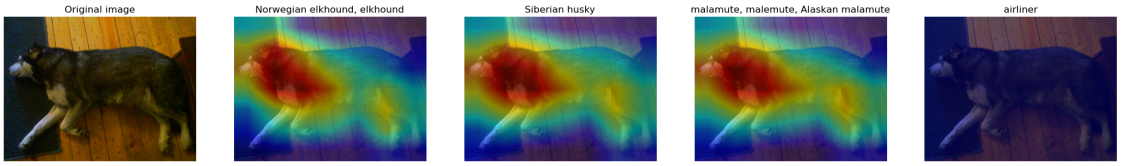
[24]:
```
gradcam = GradCAM(densenet121, densenet121.features[10])

for idx in selected_indices:
    image_path, _ = dataset.imgs[idx]
    image, _ = dataset[idx]
    image = image.unsqueeze(0)
    show_grad_cam(image, image_path, gradcam)
```











The algorithm isn't able to identify the right classes at the first layer, neither at the fifth. It is still surprise how the model detect some contour of the main animal.

It is approximatively right after the 10th layer over 12.

2.4 Part 4: Try GradCAM on `9928031928.png` , describe and comment the results

[26]:
```
# Your code here

image_path = "data/TP2_images/9928031928.png"
print(image_path)

transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

image = Image.open(image_path).convert("RGB")
image_tensor = transform(image).unsqueeze(0)

gradcam = GradCAM(densenet121, densenet121.features[-1])
show_grad_cam(image_tensor, image_path, gradcam)
```

```
data/TP2_images/9928031928.png

c:\Users\colin\anaconda3\envs\Alaska_deep\lib\site-
packages\torch\nn\modules\module.py:1827: FutureWarning: Using a non-full
backward hook when the forward contains multiple autograd Nodes is deprecated
and will be removed in future versions. This hook will be missing some
grad_input. Please use register_full_backward_hook to get the documented
behavior.
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
```



For this image, the first class detected is *jeep, landrover*, which is not related to what we first see on the image. Some noice must have been had to it.

Otherwise, for the class *African elephant, Loxodonta africana*, it mostly highlights the elephant ears, so it can implies the algorithm has well detected the most important

element to distingish this class.

Finally, for the class *triceratops*, it detect mostly the elephant leg, which could be correct.

2.5 Part 5: What are the principal contributions of GradCAM (the answer is in the paper) ?

The principal contribution of GradCam is the fact that it is a class-discriminative localization technique that provides visual explanations for any CNN-based model without requiring architectural modifications or re-training. * A technique called Class Activation Mapping (CAM) was previously proposed before GradCam for identifying discriminative regions used by a restricted class of image classification but only for CNN model without fully-connected layers. Here, GradCam can be applied to a variety of CNN like CNNs with fully-connected layers, CNNs used for structured outputs CNNs used in tasks with multi-modal inputs or reinforcement learning * Also, GradCam doesn't need to modify the architecture of the model or to re-train it, it only captures the right information during the evaluation of the data on a certain layer. * Because neural network is difficult to interpret, GradCam is a way to understand which region in an image help the network to decide the class the image. And therefore, understand the parts of the image that make the model to fail to classify correctly the data. In the end, it could help users build trust in AI models. * Additionally, Grad-CAM plays a crucial role in diagnosing biases within datasets, contributing to more generalizable and fair AI systems. * GradCam can also use neuron importance and neuron names to obtain textual explanations for model decisions.

2.6 Bonus 5: What are the main differences between DenseNet and ResNet ?

DenseNet and ResNet are both CNN architectures that effectively use skip connections in order to facilitate a better flow of gradients throughout the network. But they still different on some points :

The main differences between DenseNet and ResNet are : * **Implementation of Skip Connections :** ResNet and DenseNet differ primarily in how layers are connected. ResNet uses *residual connections*, where the output of a layer is directly added to a later layer. On the other hand, DenseNet adopts *dense connections*, where each layer is connected to all previous layers through concatenation of feature maps. * **Feature reuse :** ResNet features are reused indirectly via residual learning, where each layer learns the difference between the input and the desired output. While densenet reuse directly the features by concatenating them from all preceding layers, leading to more efficient use of learned features. * **Memory :** The concatenation in DenseNet increases the number of channels, leading to higher memory consumption, while ResNet, using summation, is more memory-efficient.

Still, DenseNets are known to achieve comparable or better results with fewer number of parameters than ResNets.