```
%matplotlib inline
!pip install wget
```

```
    Collecting wget
      Downloading wget-3.2.zip (10 kB)
      Preparing metadata (setup.py) ... done
    Building wheels for collected packages: wget
      Building wheel for wget (setup.py) ... done
      Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9656 sha256=4876c0bfe2354fca0e5dde09108d53d46e56558574cbebc17fe39
      Stored in directory: /root/.cache/pip/wheels/8b/f1/7f/5c94f0a7a505ca1c81cd1d9208ae2064675d97582078e6c769
    Successfully built wget
    Installing collected packages: wget
    Successfully installed wget-3.2
```

Computer Vision Tutorial

In this tutorial, we will recall how to train a Convolutional Neural Network (CNN) for image classification and how to use transfer learning. You can read more about the transfer learning at cs231n notes

Quoting these notes,

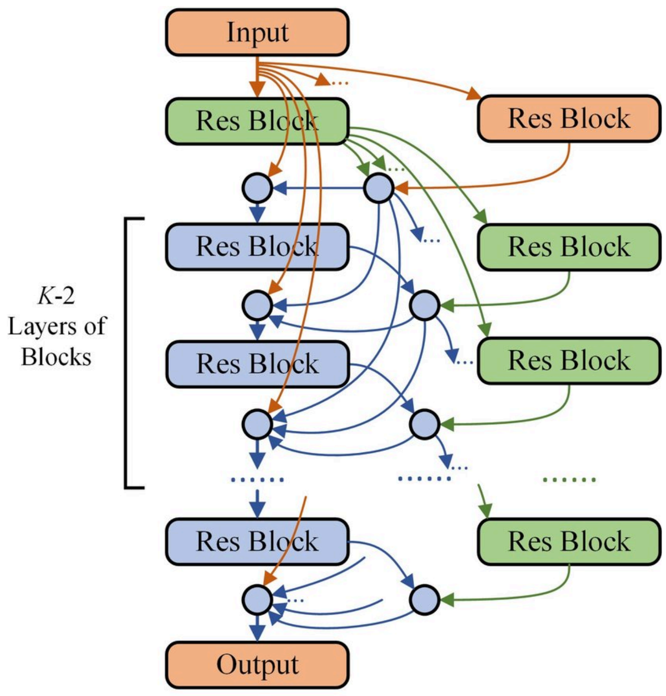
```
In practice, very few people train an entire Convolutional Network
from scratch (with random initialization), because it is relatively
rare to have a dataset of sufficient size. Instead, it is common to
pretrain a ConvNet on a very large dataset (e.g. ImageNet, which
contains 1.2 million images with 1000 categories), and then use the
ConvNet either as an initialization or a fixed feature extractor for
the task of interest.
```

Therefore there exists three main scenarios when training a CNN:

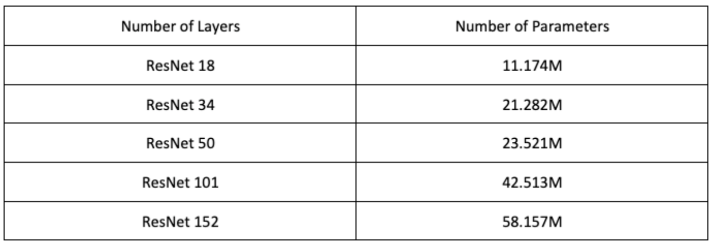
- **Train the network from scratch**: we randomly initialize the weights in every layers.
- **Finetuning the convnet**: Instead of random initialization, we initialize the network with a pretrained network, like the one that is trained on imagenet 1000 dataset. Rest of the training looks as usual.
- **ConvNet as fixed feature extractor**: we load the pretrained weights and we freeze the weights of all layers except for the last fully connected layer. This layer is replaced with a new one with random weights and it is the only layer trained.

The most common CNN: The ResNet

In the following we will put in practice these scenarios by training (or finetuning) the most commonly used CNN: the Residual Network (ResNet). This network deeply reduces the problem of *vanishing gradients* thanks to the employment of *skipping connections*. The figure in the following better illustrates this architectural trick:



This allows us to train CNN in a very stable (and easy) way.



Number of Layers	Number of Parameters
ResNet 18	11.174M
ResNet 34	21.282M
ResNet 50	23.521M
ResNet 101	42.513M
ResNet 152	58.157M

There exist several ResNet variants, suitable for different learning problems, but they all achieve very high classification performance (even the smallest ResNet 18). All Resnet variants are implemented in torchvision which also provides their pretrained version on ImageNet.

```
# License: BSD
# Author: Sasank Chilamkurthy

from __future__ import print_function, division

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
import wget
import zipfile
import shutil


cudnn.benchmark = True
plt.ion()   # interactive mode

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Device", device)
```

```
    Device cuda:0
```

Load Data

We will use torchvision and torch.utils.data packages for loading the data.

The problem we're going to solve today is to train a model to classify **ants** and **bees**. We have about 120 training images each for ants and bees. There are 75 validation images for each class. Usually, this is a very small dataset to generalize upon, if trained from scratch. Since we are using transfer learning, we should be able to generalize reasonably well.

This dataset is a very small subset of imagenet composed of these two classes only.

We use the "wget" and "ZipFile" libraries to download and extract the data to the /data directory.

```
data_dir = 'hymenoptera_data'

if os.path.exists(data_dir):
    shutil.rmtree(data_dir) # make sure there is nothing in our folder

os.makedirs(data_dir) # create folders
wget.download("https://download.pytorch.org/tutorial/hymenoptera_data.zip",
              data_dir) # download dataset
with zipfile.ZipFile(os.path.join(data_dir, "hymenoptera_data.zip"), 'r') as zip_ref:
    zip_ref.extractall(".") # extract dataset
```

```
for root, dirs, files in os.walk("hymenoptera_data"):
    level = root.replace("hymenoptera_data", '').count(os.sep)
    print(f'{" "  * 4 * (level)}{os.path.basename(root)}/')
```

```
    hymenoptera_data/
        val/
            bees/
            ants/
        train/
            bees/
            ants/
```

As it commonly happens, the data are already split into "train" and "val" data. Furthermore each folder is split again into "ants" and "bees". This is exactly the configuration needed for the `ImageFolder` Dataset class (from `torchvision.datasets`). It will automatically create a dataset returning the images and the label corresponding to the folder the image has been saved.

Also notice that `ImageFolder` requires two parameters to be passed:

- `root`: is the folder where the dataset is stored, for the train dataset it is `hymenoptera_data/train` or `hymnoptera_data/val`
- `transform`: this is something **new**, transforms are used to convert the image data into a tensor, center it and normalize it (`ToTensor()`, `Normalize(mean, variance)` `CenterCrop(size)`)

For the training transforms, however, you can also apply *data augmentation*: at each iteration we modify a little bit the image seen by the model. In this case we only do

- `RandomResizedCrop(size, [ratio_min, ratio_max])` which feeds the network with only a part of the original image
- `RandomHorizontalFlip()` which returnes the the mirrored image with 50% probability.

```
# Data augmentation and normalization for training
train_transforms = transforms.Compose([
      transforms.RandomResizedCrop(224, [0.75, 1.0]),
      transforms.RandomHorizontalFlip(),
      transforms.ToTensor(),
      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
  ])
# Just normalization for validation
val_transforms = transforms.Compose([
      transforms.Resize(256),
      transforms.CenterCrop(224),
      transforms.ToTensor(),
      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
  ])

train_dir = os.path.join(data_dir, "train")
train_dataset = datasets.ImageFolder(root=train_dir,
                                     transform=train_transforms)
val_dir = os.path.join(data_dir, "val")
val_dataset = datasets.ImageFolder(root=val_dir,
                                   transform=val_transforms)

train_size = len(train_dataset)
val_size = len(val_dataset)

print("Dataset sizes:", train_size, val_size )

class_names = train_dataset.classes

print("Class names:", class_names)
```

```
Dataset sizes: 244 153
Class names: ['ants', 'bees']
```

Now we have to create the dataloader, as we did in the previous labs. Notice however that we are using one more parameter:

- `num_workers` is used to *parallelize* the loading from disk.

Indeed, image datasets are too big normally to fit in memory and images are loaded from disk at every batch. In this case we do that using 2 threads to load the 16 images in parallel (8 per thread).

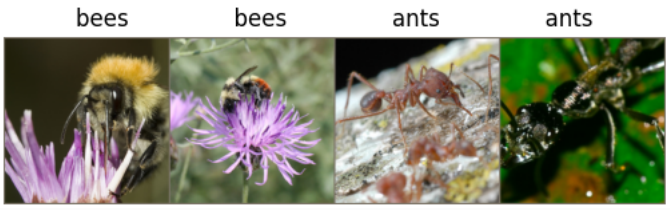
```
batch_size = 16
train_dl = torch.utils.data.DataLoader(train_dataset, shuffle=True,
                                       batch_size=batch_size, num_workers=2)
val_dl = torch.utils.data.DataLoader(val_dataset, shuffle=True,
                                     batch_size=batch_size*2, num_workers=2)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Visualize a few images

Let's visualize a few training images so as to understand the data augmentations.

```
def imshow(inp, title=None, ax=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))  # reconvert to numpy tensor
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean                  # take out normalization
    inp = np.clip(inp, 0, 1)                # clip values to [0,1]
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.axis('off')
    plt.pause(0.001)                   # pause a bit so that plots are updated


# Get a batch of training data
x, classes = next(iter(train_dl))

# Make a grid from batch
space = " "*15
out = torchvision.utils.make_grid(x[:4]) # we only plot the first 4 images
imshow(out, title= space.join(class_names[c] for c in classes[:4]))

# Redo for valid data
x, classes = next(iter(val_dl))
out = torchvision.utils.make_grid(x[:4]) # we only plot the first 4 images
imshow(out, title=space.join(class_names[c] for c in classes[:4]))
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()
```



Training the model

Now, let's write a general function to train a model. I here put the training loop and the evaluation loop in two distinct function to better structurize the code.

Trying to recall what we did in the previous labs, try to complete the missing parts (### COMPLETE ###) of both training and evaluation loops.

```python
# Function to iterate over data while training
def train_one_epoch(model, train_dl, loss, optim, device):
    model.train()  # Set model to training mode
    cur_loss, cur_acc = 0.0, 0.0
    for x, y in train_dl:
        x, y = x.to(device), y.to(device)

        # zero the parameter gradients
        ### COMPLETE - 1 line expected ###
        optim.zero_grad()

        # forward
        ### COMPLETE - 3 lines expected ###
        outputs = model(x)
        preds = torch.argmax(outputs, 1)
        l = loss(outputs, y)

        # backward + optimize
        ### COMPLETE - 2 lines expected ###
        l.backward()
        optim.step()

        # statistics
        cur_loss += l.item() * x.size(0)
        cur_acc += torch.sum(preds == y.data)

    epoch_loss = cur_loss / len(train_dl.dataset)
    epoch_acc = cur_acc.double() / len(train_dl.dataset)
    return epoch_loss, epoch_acc

# Function to iterate over data while evaluating
def eval_one_epoch(model, val_dl, loss, device):
    model.eval()  # Set model to evaluate mode
    cur_loss, cur_acc = 0.0, 0.0
    with torch.no_grad():
        for x, y in val_dl:
            x, y = x.to(device), y.to(device)

            # forward pass, prediction and loss computation
            ### COMPLETE - 3 lines expected ###
            outputs = model(x)
            preds = torch.argmax(outputs, 1)
            l = loss(outputs, y)

            # statistics
            cur_loss += l.item() * x.size(0)
            cur_acc += torch.sum(preds == y.data)

    epoch_loss = cur_loss / len(val_dl.dataset)
    epoch_acc = cur_acc.double() / len(val_dl.dataset)
    return epoch_loss, epoch_acc

def train_model(model, train_dl, val_dl, loss, optim, num_epochs=20):
    model.to(device)
    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print(f'Epoch {epoch}/{num_epochs - 1}')
        print('-' * 10)

        train_loss, train_acc = train_one_epoch(model, train_dl, loss, optim, device)
        print(f'Train Loss: {train_loss:.4f} Acc: {train_acc:.4f}')

        val_loss, val_acc = eval_one_epoch(model, val_dl, loss, device)
        print(f'Val Loss: {val_loss:.4f} Acc: {val_acc:.4f}\n')

        # save the best model
        if val_acc > best_acc:
            best_acc = val_acc
            torch.save(model.state_dict(), "temp_model.pt")

    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s')
    print(f'Best val Acc: {best_acc:4f}')

    # load best model weights
    model.load_state_dict(torch.load("temp_model.pt"))
    return model
```

Visualizing the model predictions

Here is a generic function to display predictions for a few images. We will re-use it to generate saliency maps in the following.

```
def visualize_preds(model, num_images=4):
    space = " "*15
    was_training = model.training
    model.eval()
    with torch.no_grad():
        x, y = next(iter(val_dl))
        x = x.to(device)
        y = y.to(device)

        outputs = model(x)
        preds = torch.argmax(outputs, 1)

        out = torchvision.utils.make_grid(x[:num_images].cpu())
        imshow(out, title=space.join(class_names[p] for p in preds[:num_images]))
    model.train(mode=was_training)
```

Training a CNN from scratch

For this notebook we will use the simplest of the ResNet models, the ResNet18 (link to the paper).

To do so, we simply instantiate the model (`torchvision.models.resnet18(num_classes)`) as we used to do also for the Logistic regression and we train it for a few epochs. Remember as well to:

- Select the Loss to employ
- Instantiate an optimizer

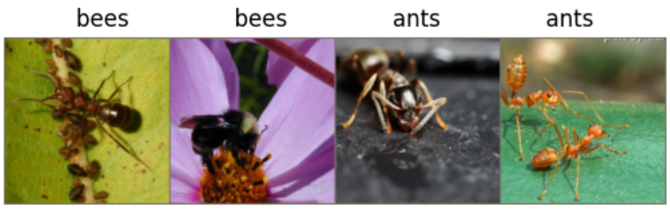
The complete training (for 10 epochs) should take less than 1 min on GPU.

```
model = models.resnet18(num_classes=2)
model = model.to(device)

loss = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer = optim.AdamW(model.parameters(), lr=1e-4)

# Train and evaluate
model = train_model(model, train_dl, val_dl, loss, optimizer,
                       num_epochs=10)
```

```
Epoch 0/9
----------
Train Loss: 0.6858 Acc: 0.6066
Val Loss: 0.7728 Acc: 0.5098

Epoch 1/9
----------
Train Loss: 0.6288 Acc: 0.6598
Val Loss: 0.6162 Acc: 0.6471

Epoch 2/9
----------
Train Loss: 0.5691 Acc: 0.6680
Val Loss: 0.6223 Acc: 0.6732

Epoch 3/9
----------
Train Loss: 0.5186 Acc: 0.7910
Val Loss: 0.6033 Acc: 0.7124

Epoch 4/9
----------
Train Loss: 0.5411 Acc: 0.7295
Val Loss: 0.6108 Acc: 0.7255

Epoch 5/9
----------
Train Loss: 0.4388 Acc: 0.7664
Val Loss: 0.6633 Acc: 0.7320

Epoch 6/9
----------
Train Loss: 0.4087 Acc: 0.7910
Val Loss: 0.6003 Acc: 0.7124

Epoch 7/9
----------
```

```
Train Loss: 0.3719 Acc: 0.8402
Val Loss: 0.7227 Acc: 0.7059

Epoch 8/9
----------
Train Loss: 0.3904 Acc: 0.8197
Val Loss: 1.1575 Acc: 0.6078

Epoch 9/9
----------
Train Loss: 0.3871 Acc: 0.8320
Val Loss: 0.8416 Acc: 0.6797

Training complete in 0m 44s
Best val Acc: 0.732026
```

```
visualize_preds(model)
```



The model that we trained is not very accurate. Why? Because CNN are incredibly complex models and to be traned properly (without overfitting), they need to "see" a sufficiently large representation of the world. For this reason, most of the time we start at least from a network pretrained on the ImageNet dataset (1M samples).

Finetune a CNN

To Finetune a CNN we need to do three steps:

- Load the weights of the pretrained model: torchvision provides the weights pretrained Imagenet for each of the CNN implemented in the library (`weights=torchvision.models.ResNet18_Weights`).
- Reset the final fully connected layer: to do so we initialize a linear layer `nn.Linear()` as the last layer of the model which is `model_ft.fc`

```
from torchvision.models import ResNet18_Weights
model_ft = models.resnet18(ResNet18_Weights) ### COMPLETE  ###

# Here the model output size is set to 2
num_ftrs = model_ft.fc.in_features
model_ft.fc = torch.nn.Linear(num_ftrs, 2) ### COMPLETE  ###

model_ft = model_ft.to(device)
optimizer_ft = optim.AdamW(model_ft.parameters(), lr=1e-4)

# Train and evaluate
model_ft = train_model(model_ft, train_dl, val_dl, loss, optimizer_ft,
                       num_epochs=10)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:135: UserWarning: Using 'weights' as positional parameter(s)
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `Non
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-f37072fd.
100%|██████████| 44.7M/44.7M [00:00<00:00, 153MB/s]Epoch 0/9
----------

Train Loss: 0.3749 Acc: 0.8115
Val Loss: 0.2230 Acc: 0.9020

Epoch 1/9
----------
Train Loss: 0.0798 Acc: 0.9795
Val Loss: 0.2107 Acc: 0.9281

Epoch 2/9
----------
Train Loss: 0.0422 Acc: 0.9918
Val Loss: 0.2034 Acc: 0.9346

Epoch 3/9
----------
Train Loss: 0.0352 Acc: 0.9795
Val Loss: 0.2511 Acc: 0.9216

Epoch 4/9
----------
```

```
    Train Loss: 0.0190 Acc: 0.9959
    Val Loss: 0.2298 Acc: 0.9412

    Epoch 5/9
    ----------
    Train Loss: 0.0352 Acc: 0.9836
    Val Loss: 0.2977 Acc: 0.9020

    Epoch 6/9
    ----------
    Train Loss: 0.0174 Acc: 1.0000
    Val Loss: 0.2815 Acc: 0.9281

    Epoch 7/9
    ----------
    Train Loss: 0.0120 Acc: 1.0000
    Val Loss: 0.2938 Acc: 0.9216

    Epoch 8/9
    ----------
    Train Loss: 0.0296 Acc: 0.9959
    Val Loss: 0.3083 Acc: 0.9216

    Epoch 9/9
    ----------
    Train Loss: 0.0246 Acc: 0.9918
    Val Loss: 0.3032 Acc: 0.9216

    Training complete in 0m 29s
```

```
visualize_preds(model_ft)
```

Indeed it seems that pretraining is working effectively. We got a pretty good validation accuracy (>90%).

ConvNet as fixed feature extractor

In this case we only train the last layer of the network. For this reason we need to:

- Freeze all the network except the final layer, by seting `requires_grad = False` to freeze the parameters so that the gradients are not computed in `backward()` and speed up the computation. You can read more about this in the documentation here.
- Pass to the optimizer only the weights of last layer (`model_fe.fc.parameters()`)

To train it will take less time compared to previous scenario. This is expected as gradients don't need to be computed for most of the network. However, forward still needs to be computed.

```
from torchvision.models import ResNet18_Weights
model_fe = models.resnet18(weights=ResNet18_Weights)

# Freeze network weights for all layers
for param in model_fe.parameters():
    ### COMPLETE - 1 line expected ###
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_fe.fc.in_features
model_fe.fc = torch.nn.Linear(num_ftrs, 2) ### COMPLETE ###

model_fe = model_fe.to(device)

# Only parameters of final layer are being optimized as opposed to before
optimizer_conv = optim.AdamW(model_fe.fc.parameters(), lr=1e-3) ### COMPLETE ###

# Train and evaluate
model_fe = train_model(model_fe, train_dl, val_dl, loss, optimizer_conv,
                         num_epochs=10)
```

We actually got a better result with the CNN as Feature Extractor (> 95 %)!

This is most likely due to the fact that in the previous case we were strongly overfitting the data distribution. Indeed, by modifying also the hidden layers, the model at hand is incredibly complex for the small ant-bees dataset. Therefore if you have small dataset is better to only train the last fc layer.

```
visualize_preds(model_fe)

plt.ioff()
plt.show()
```

Long story short: avoid complexity if it is not required!

- Start simple: transfer learning
- Only if required: increase the complexity of your learning strategy adding finetuning and data augmentation.
- Train from scratch a CNN: only when the data distribution is highly different from the one of pretrained dataset (e.g., tumoral cell detection)

##

Saliency Maps

It's now time to visualize some explanations through a saliency map. Let's recall what is a saliency map. Given an image $X \in [0,1]^{C\times H\times W}$ (where H is the height, W is the width, and C is the number of channels), a saliency map $S \in [0,1]^{H\times W}$ is an image where the brightness of the (i,j) pixel, i.e., $S(i,j)$, represents how important or salient that pixel is in the network prediction. In other words, the greater the value of $S(i,j)$, the more important that pixel. If a particular region in the saliency map is concentrated with bright pixels, it means that the region or feature is important for prediction.

Vanilla Gradient

Let's assume that $f^i(x)$ denotes the pre-softmax score (logit) of the i-th class (there are a total of K classes). The prediction based on the output is given by

$$k^* = argmax_{i\in[1,K]} f^i(x)\$$$

and k^* is the label of the class with the highest score. The algorithm first computes the vanilla gradients as:

$$G = \nabla_x f^{k^*}(x) = \frac{\delta f^{k^*}(x)}{\delta x}$$

The vanilla gradients G here is a matrix $\in \mathbb{R}^{C\times H\times W}$ representing the direction in which the score of the predicted class, i.e., $f^{k^*}(x)$, increases. A positive gradient means that the pixel is important for increasing the predicted class score, while a negative or zero gradient means that the pixel is not important for the prediction. So we will take only the positive gradients $G^+ = max(0,G)$ for computing the saliency map. The saliency map $S \in \mathbb{R}^{HxW}$ is finally defined as the maximum value of the gradient G^+ along the channel dimension.

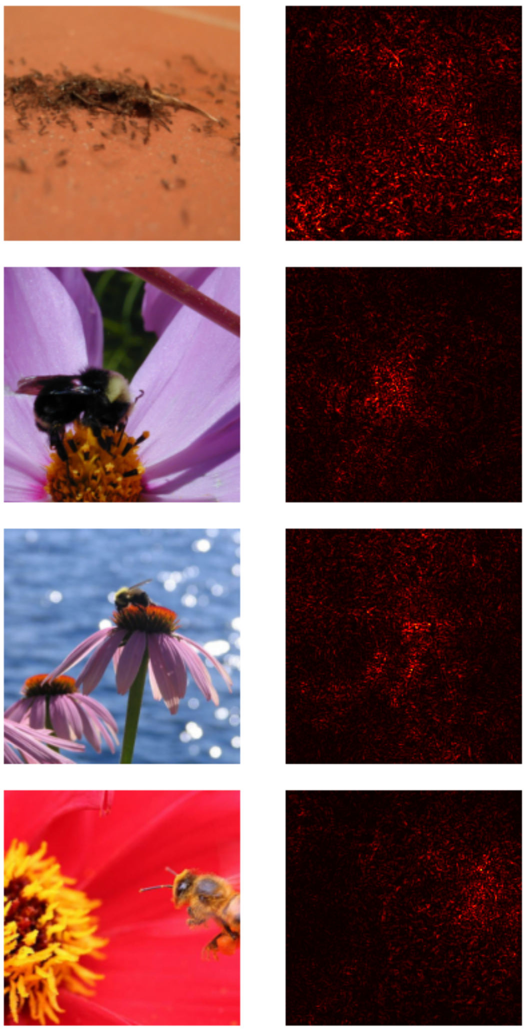
Implementing Vanilla Gradient

Let's now implement the vanilla gradient. To do so we will simply modify the visualization function to also output the gradients with respect to the predicted class.

Recall that the input `x` normally does not receive gradients (we do not want our input to be modified during training!).

- In this case however we will require gradients to flow until the input with `x.requires_grad = True`.
- We will backgropagate the loss with respect to the 100% positive prediction of the predicted class
- We will extract the gradients from the input with `x.gradient`
- We will take the positive values with the `relu()` function
- We will take the maximum values along the channel dimension with torch.max(grad, 1)[0] (torch.max() returns values, indices, we are only interested in the first)

```
import cv2

def vanilla_gradient(model, num_images=4):
    optimizer = optim.AdamW(model.parameters())

    model.eval()
    x, y = next(iter(val_dl))

    x = x.to(device)
    y = y.to(device)

    # require gradient to flow until the input
    x.requires_grad = True

    outputs = model(x)
    preds = torch.argmax(outputs, 1)

    # backpropagate the loss with respect to the predicted class
    ### COMPLETE - 3 lines expected ###
    optimizer.zero_grad()
    l = loss(outputs, preds)
    l.backward()

    # extract gradient from the input
    ### COMPLETE - 1 line expected ###
    gradients = x.grad

    # take positive values
    ### COMPLETE - 1 line expected ###
    gradients_pos = gradients.relu()

    # take maximum over the channels (first dimension)
    ### COMPLETE - 1 line expected ###
    gradients_max = torch.max(gradients_pos, 1)[0]


    inp = x.detach().cpu().numpy().transpose((-4, -2, -1, -3))  # reconvert to numpy tensor B x H x W x C
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean                  # take out normalization
    inp = np.clip(inp, 0, 1)                # clip values to [0,1]

    for i in range(num_images):
        fig, axes = plt.subplots(1, 2, figsize=(5,11))

        image = inp[i]
        # gradient = torch.max(gradients[i].relu(), 0)[0]
        gradient = gradients_max[i].cpu().detach().numpy()

        # plot the original image
        axes[0].imshow(image)

        # plot the gradients as a heatmap
        axes[1].imshow(gradient,  cmap=plt.cm.hot)

        axes[0].axis("off"), axes[1].axis('off')
        plt.show()


# if you pass the model_fe you need to require gradients on all layers as well
for param in model_fe.parameters():
    param.requires_grad = True

vanilla_gradient(model_fe)
```



How does our model perform?

Quite well, the highlighted pixels generally cover the *ant* or the *bee* in the image (Note: the results may vary according to the images that the val_dl returns, try to run it a few times to see other examples)

Problem: the model however still focuses on the *background*: this is a signal that even though the accuracy was high (>95%) we cannot still trust completely the model.

Indeed we trained the model on too few images (and not many epochs too) to have a reliable model

How do models trained on ImageNet perform?

To check if our previous assumption is true, let's check how the models trained on Imagenet perform on a standard image.

Upload the "dog.png" image to Colab (or put it in the same directory of your notebook if you are working locally) and execute the following cell.

```
from PIL import Image

orig_image = Image.open("dog.png").convert('RGB') # original image
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
```