Classification and Performance

Make sure you are connected to a T4 GPU runtime. The following code should report true if you are.

```
import torch
print("GPU available =", torch.cuda.is_available())
```

```
GPU available = True
```

Install prerequisites needed for this assignment, `thop` is used for profiling PyTorch models https://github.com/ultralytics/thop, while `tqdm` makes your loops show a progress bar https://tqdm.github.io/

```
!pip install thop segmentation-models-pytorch transformers
import math
import numpy as np
import torch
import torch.nn as nn
import gc
import torchvision
from torchvision import datasets, transforms
from PIL import Image
import segmentation_models_pytorch as smp
import thop
from transformers import ViTFeatureExtractor, ViTForImageClassification
import matplotlib.pyplot as plt
from tqdm import tqdm
import time

# we won't be doing any training here, so let's disable autograd
torch.set_grad_enabled(False)
```

```
Collecting thop
  Downloading thop-0.1.1.post2209072238-py3-none-any.whl.metadata (2.7 kB)
Collecting segmentation-models-pytorch
  Downloading segmentation_models_pytorch-0.4.0-py3-none-any.whl.metadata (32 kB)
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-packages (4.47.1)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from thop) (2.5.1+cu121)
Collecting efficientnet-pytorch>=0.6.1 (from segmentation-models-pytorch)
  Downloading efficientnet_pytorch-0.7.1.tar.gz (21 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: huggingface-hub>=0.24 in /usr/local/lib/python3.10/dist-packages (from segmentation-models-pytorch)
Requirement already satisfied: numpy>=1.19.3 in /usr/local/lib/python3.10/dist-packages (from segmentation-models-pytorch) (1.26.4)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.10/dist-packages (from segmentation-models-pytorch) (11.0.0)
Collecting pretrainedmodels>=0.7.1 (from segmentation-models-pytorch)
  Downloading pretrainedmodels-0.7.4.tar.gz (58 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 58.8/58.8 kB 5.8 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from segmentation-models-pytorch) (1.17.0)
Requirement already satisfied: timm>=0.9 in /usr/local/lib/python3.10/dist-packages (from segmentation-models-pytorch) (1.0.12)
Requirement already satisfied: torchvision>=0.9 in /usr/local/lib/python3.10/dist-packages (from segmentation-models-pytorch) (0.20
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist-packages (from segmentation-models-pytorch) (4.67.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from transformers) (3.16.1)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers) (24.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (2024.11.6)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers) (2.32.3)
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.10/dist-packages (from transformers) (0.21.0)
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (0.4.5)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.24->segmentatior
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub>=0.24->se
Collecting munch (from pretrainedmodels>=0.7.1->segmentation-models-pytorch)
  Downloading munch-4.0.0-py2.py3-none-any.whl.metadata (5.9 kB)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch->thop) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch->thop) (3.1.4)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from torch->thop) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch->thop) (1.3
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2024.12
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch->thop) (3.0.2)
Downloading thop-0.1.1.post2209072238-py3-none-any.whl (15 kB)
Downloading segmentation_models_pytorch-0.4.0-py3-none-any.whl (121 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 121.3/121.3 kB 12.1 MB/s eta 0:00:00
Downloading munch-4.0.0-py2.py3-none-any.whl (9.9 kB)
Building wheels for collected packages: efficientnet-pytorch, pretrainedmodels
  Building wheel for efficientnet-pytorch (setup.py) ... done
  Created wheel for efficientnet-pytorch: filename=efficientnet_pytorch-0.7.1-py3-none-any.whl size=16424 sha256=85d1836ecd6857ce5e6
  Stored in directory: /root/.cache/pip/wheels/03/3f/e9/911b1bc46869644912bda90a56bcf7b960f20b5187feea3baf
  Building wheel for pretrainedmodels (setup.py) ... done
```

```
  Created wheel for pretrainedmodels: filename=pretrainedmodels-0.7.4-py3-none-any.whl size=60944 sha256=d748babc6cb5767bdc06a60f1dc
  Stored in directory: /root/.cache/pip/wheels/35/cb/a5/8f534c60142835bfc889f9a482e4a67e0b817032d9c6883b64
Successfully built efficientnet-pytorch pretrainedmodels
Installing collected packages: munch, thop, efficientnet-pytorch, pretrainedmodels, segmentation-models-pytorch
Successfully installed efficientnet-pytorch-0.7.1 munch-4.0.0 pretrainedmodels-0.7.4 segmentation-models-pytorch-0.4.0 thop-0.1.1.pc
<torch.autograd.grad_mode.set_grad_enabled at 0x7ea2f574f760>
```

Image Classification

You will be looking at image classification in the first part of this assignment, the goal of image classification is to identify subjects within a given image. In the previous assignment, you looked at using MNIST, which is also a classification task "which number is present", where for images the gold standard is Imagent "which class is present".

You can find out more information about Imagenet here:

https://en.wikipedia.org/wiki/ImageNet

Normally you would want to test classification on ImageNet as that's the dataset in which classification models tend to be trained on. However, the Imagenet dataset is not publicly available nor is it reasonable in size to download via Colab (100s of GBs).

Instead, you will use the Caltech101 dataset. However, Caltech101 uses 101 labels which do not correspond to the Imagenet labels. As such, you will need to also download a bigger classification model to serve as a baseline for accuracy comparisons.

More info can be found about the Caltech101 dataset here:

https://en.wikipedia.org/wiki/Caltech_101

Download the dataset you will be using: Caltech101

```
# convert to RGB class - some of the Caltech101 images are grayscale and do not match the tensor shapes
class ConvertToRGB:
    def __call__(self, image):
        # If grayscale image, convert to RGB
        if image.mode == "L":
            image = Image.merge("RGB", (image, image, image))
        return image

# Define transformations
transform = transforms.Compose([
    ConvertToRGB(), # first convert to RGB
    transforms.Resize((224, 224)),  # Most pretrained models expect 224x224 inputs
    transforms.ToTensor(),
    # this normalization is shared among all of the torch-hub models we will be using
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# Download the dataset
caltech101_dataset = datasets.Caltech101(root="./data", download=True, transform=transform)
```

```
Downloading...
From (original): https://drive.google.com/uc?id=137RyRjvTBkBiIfeYBNZBtViDHQ6_Ewsp
From (redirected): https://drive.usercontent.google.com/download?id=137RyRjvTBkBiIfeYBNZBtViDHQ6_Ewsp&confirm=t&uuid=dc0ac109-cca4-4
To: /content/data/caltech101/101_ObjectCategories.tar.gz
100%|██████████| 132M/132M [00:02<00:00, 53.3MB/s]
Extracting ./data/caltech101/101_ObjectCategories.tar.gz to ./data/caltech101
Downloading...
From: https://drive.google.com/uc?id=175kQy3UsZ0wUEHZjqkUDdNVssr7bgh_m
To: /content/data/caltech101/Annotations.tar
100%|██████████| 14.0M/14.0M [00:00<00:00, 63.8MB/s]
Extracting ./data/caltech101/Annotations.tar to ./data/caltech101
```

```
from torch.utils.data import DataLoader

# set a manual seed for determinism
torch.manual_seed(42)
dataloader = DataLoader(caltech101_dataset, batch_size=16, shuffle=True)
```

Create the dataloader with a batch size of 16. You are fixing the seed for reproducibility.

```
# download four classification models from torch-hub
resnet152_model = torchvision.models.resnet152(pretrained=True)
resnet50_model = torchvision.models.resnet50(pretrained=True)
resnet18_model = torchvision.models.resnet18(pretrained=True)
mobilenet_v2_model = torchvision.models.mobilenet_v2(pretrained=True)

# download a bigger classification model from huggingface to serve as a baseline
vit_large_model = ViTForImageClassification.from_pretrained('google/vit-large-patch16-224')
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated sinc
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None`
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet152-394f9c45.pth" to /root/.cache/torch/hub/checkpoints/resnet152-394f9c45.p
100%|██████████| 230M/230M [00:01<00:00, 165MB/s]
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None`
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth
100%|██████████| 97.8M/97.8M [00:00<00:00, 140MB/s]
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None`
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100%|██████████| 44.7M/44.7M [00:02<00:00, 19.2MB/s]
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None`
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to /root/.cache/torch/hub/checkpoints/mobilenet_v2-b03
100%|██████████| 13.6M/13.6M [00:00<00:00, 137MB/s]
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
config.json: 100%                                   69.7k/69.7k [00:00<00:00, 3.32MB/s]

pytorch_model.bin: 100%                             1.22G/1.22G [00:12<00:00, 100MB/s]
```

Move the models to the GPU and set them in eval mode. This will disable dropout regularization and batch norm statistic calculation.

```
resnet152_model = resnet152_model.to("cuda").eval()
resnet50_model = resnet50_model.to("cuda").eval()
resnet18_model = resnet18_model.to("cuda").eval()
mobilenet_v2_model = mobilenet_v2_model.to("cuda").eval()
vit_large_model = vit_large_model.to("cuda").eval()
```

Download a series of models for testing. The VIT-L/16 model will serve as a baseline - this is a more accurate vision transformer based model.

The other models you will use are:

- resnet 18
- resnet 50
- resnet 152
- mobilenet v2

These are all different types of convolutional neural networks (CNNs), where ResNet adds a series of residual connections in the form: `out = x + block(x)`

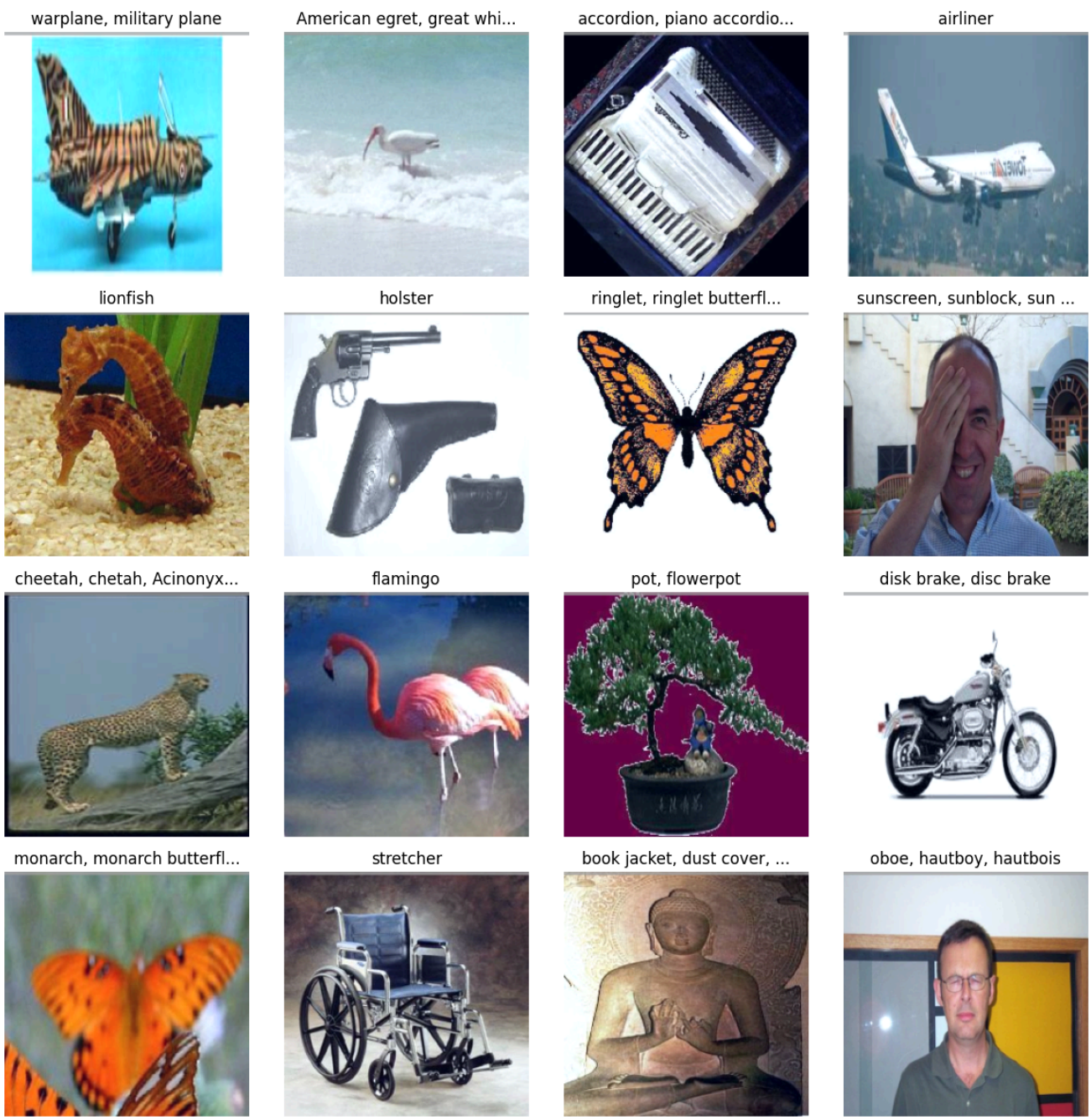
There's a good overview of the different versions here: https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

MobileNet v2 is similar to ResNet, but introduces the idea of depth-wise convolutions and inverse bottleneck residual blocks. You will only be using it as a point of comparison, however, you can find out more details regarding the structure from here if interested: https://medium.com/@luis_gonzales/a-look-at-mobilenetv2-inverted-residuals-and-linear-bottlenecks-d49f85c12423

Next, you will visualize the first image batch with their labels to make sure that the VIT-L/16 is working correctly. Luckily huggingface also implements an `id -> string` mapping, which will turn the classes into a human readable form.

```
# get the first batch
dataiter = iter(dataloader)
images, _ = next(dataiter)

# define a denorm helper function - this undoes the dataloader normalization so we can see the images better
def denormalize(tensor, mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]):
    """ Denormalizes an image tensor that was previously normalized. """
    for t, m, s in zip(tensor, mean, std):
        t.mul_(s).add_(m)
    return tensor

# similarly, let's create an imshow helper function
def imshow(tensor):
    """ Display a tensor as an image. """
    tensor = tensor.permute(1, 2, 0) # Change from C,H,W to H,W,C
    tensor = denormalize(tensor) # Denormalize if the tensor was normalized
    tensor = tensor*0.24 + 0.5 # fix the image range, it still wasn't between 0 and 1
    plt.imshow(tensor.clamp(0,1).cpu().numpy()) # plot the image
    plt.axis('off')
```

```
# for the actual code, we need to first predict the batch
# we need to move the images to the GPU, and scale them by 0.5 because VIT-L/16 uses a different normalization to the other models
with torch.no_grad(): # this isn't strictly needed since we already disabled autograd, but we should do it for good measure
  output = vit_large_model(images.cuda()*0.5)

# then we can sample the output using argmax (find the class with the highest probability)
# here we are calling output.logits because huggingface returns a struct rather than a tuple
# also, we apply argmax to the last dim (dim=-1) because that corresponds to the classes - the shape is B,C
# and we also need to move the ids to the CPU from the GPU
ids = output.logits.argmax(dim=-1).cpu()

# next we will go through all of the ids and convert them into human readable labels
# huggingface has the .config.id2label map, which helps.
# notice that we are calling id.item() to get the raw contents of the ids tensor
labels = []
for id in ids:
  labels += [vit_large_model.config.id2label[id.item()]]

# finally, let's plot the first 4 images
max_label_len = 25
fig, axes = plt.subplots(4, 4, figsize=(12, 12))
for i in range(4):
    for j in range(4):
        idx = i*4 + j
        plt.sca(axes[i, j])
        imshow(images[idx])
        # we need to trim the labels because they sometimes are too long
        if len(labels[idx]) > max_label_len:
            trimmed_label = labels[idx][:max_label_len] + '...'
        else:
            trimmed_label = labels[idx]
        axes[i,j].set_title(trimmed_label)
plt.tight_layout()
plt.show()
```



Question 1

Given the above classifications, how well do you think the model does? Can you observe any limitations? If so, do you think that's related to the model size and complexity, or is it more likely related to the training set?

For more information, the class list can be found here: https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/

Please answer below:

Evaluating the performance of a pre-trained model like ViT-Large on a dataset such as Caltech101 depends on multiple factors. Let's break down some considerations based on the classifications observed:

1. Performance Evaluation: High Accuracy: Since ViT-Large is pre-trained on ImageNet, a diverse and large dataset, it is likely to perform well on images that resemble ImageNet classes. Caltech101, while diverse, may have some images or classes that are less aligned with ImageNet's categories. Transfer Learning: The model's ability to generalize to Caltech101 depends on the similarity between ImageNet's classes and Caltech101's classes. ViT-Large, with its high capacity, should perform well on general object recognition tasks, but there might be some misclassifications if the categories are underrepresented or significantly different.

2. Possible Limitations: Class Mismatch: Caltech101 consists of 101 categories, many of which may not have close analogs in ImageNet's 1,000 categories. For example, fine-grained object categories like "cartwheel" or "parachute" may not be well-represented in ImageNet. Fine-grained Classification: Models like ViT-Large are generally good at recognizing high-level object features, but they might struggle with differentiating fine-grained categories (e.g., types of birds, or subtle object differences) unless they were specifically trained on such tasks. Out-of-Distribution Examples: Some classes in Caltech101 may represent objects or scenes that are very different from ImageNet's images. ViT, even with its large capacity, may struggle to generalize well to these out-of-distribution examples.
3. Model Size and Complexity: ViT-Large is a very powerful model, with a large number of parameters and a complex architecture. This makes it capable of recognizing very intricate patterns in images. However, its success highly depends on having enough training data that captures the range of features present in the task. If the dataset contains classes that are too niche or don't resemble those seen during training (ImageNet), even a large model may struggle. Model Size Trade-off: While a larger model like ViT-Large can capture more complex patterns, it may not always outperform smaller models like ResNet50 or MobileNetV2 on a dataset with limited diversity, especially if computational resources or fine-tuning are not optimized.
4. Training Set and Dataset Diversity: Data Distribution: The success of ViT-Large also depends on how well the training set (ImageNet) reflects the types of objects in Caltech101. If Caltech101 contains rare or unique objects not seen in ImageNet, performance may degrade. Fine-tuning: Even though ViT-Large is pretrained on a large dataset, fine-tuning it on Caltech101 could improve accuracy. Without fine-tuning, the model might not adjust well to the unique features of the Caltech101 dataset. Class Imbalance: If certain categories in Caltech101 are overrepresented or underrepresented compared to others, the model may be biased towards the overrepresented categories. This could limit the model's performance in more challenging or rare categories.
5. Comparing Model Sizes: ResNet Models (e.g., ResNet50, ResNet152) are based on residual connections, which allow them to perform well on smaller datasets with fewer parameters. While they may not match the performance of ViT-Large in more complex tasks, they can sometimes outperform larger models in situations where dataset size or class alignment is an issue. MobileNetV2, being a lightweight model, might be less capable of handling complex patterns but is faster and more efficient on smaller or simpler datasets. Summary of Potential Factors: Training Set: If Caltech101 contains classes that differ from ImageNet's distribution (e.g., fine-grained object categories or out-of-distribution classes), performance could be limited, and fine-tuning would be essential. Model Size: While ViT-Large is a powerful model, it might still struggle with classes it has not been exposed to during training, especially for niche or fine-grained categories. Class Distribution and Mismatch: Some objects in Caltech101 may not align well with ImageNet's class definitions, leading to misclassifications. A model fine-tuned on the Caltech101 dataset would likely perform better. Conclusion: If the model struggles with categories that are either underrepresented in the training set or fundamentally different from ImageNet, this is likely a result of the training set rather than the model's capacity. Fine-tuning the model on Caltech101 would likely improve results significantly, as it would adapt the model to the specific features and categories of the dataset. Would you like to proceed with fine-tuning this model, or do you want to compare the results from other models like ResNet or MobileNet on this dataset?

Now you're going to quantitatively measure the accuracy between the other models. The first thing you need to do is clear the GPU cache, to prevent an out-of-memory error. To undestand this, let's look at the current GPU memory utilization.

```
# run nvidia-smi to view the memory usage. Notice the ! before the command, this sends the command to the shell rather than python
!nvidia-smi
```

```
Fri Jan 17 06:59:22 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   60C    P0              28W /  70W |   1901MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

```
# now you will manually invoke the python garbage collector using gc.collect()
gc.collect()
# and empty the GPU tensor cache - tensors that are no longer needed (activations essentially)
torch.cuda.empty_cache()
```

```
# run nvidia-smi again
!nvidia-smi
```

```
Fri Jan 17 06:59:45 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
```

```
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P0              28W /  70W |   1715MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

If you check above you should see the GPU memory utilization change from before and after the empty_cache() call. Memory management is one of the quirks that must be considered when dealing with accelerators like a GPU. Unlike with a CPU, there is no swap file to page memory in and out of the device. Instead, this must be handled by the user. When too much of the GPU memory is used, the driver will throw an out-of-memory error (commonly referred to as OOM). In this case, the process often ends up in an unrecoverable state and needs to be restarted to fully reset the memory utilization to zero.

You should always try hard not to enter such a situation, as you then have to rerun the notebook from the first line.

Question 2

Given the above, why is the GPU memory utilization not zero? Does the current utilization match what you would expect? Please answer below:

he GPU memory utilization may not be zero for several reasons, even if you're not actively training a model or performing large computations. Here's why that might be the case:

Why GPU Memory Utilization Isn't Zero: Model Loading:

When you load a pre-trained model (like ViT-Large or ResNet), the model's weights are loaded into GPU memory. This alone takes up a significant amount of space, especially with large models. For example, ViT-Large can use several GBs of memory. Once the model is loaded, memory utilization is expected to be non-zero, even if you're not performing any operations on it. CUDA Cache:

PyTorch's GPU memory allocator retains some memory even after you've cleared tensors, in order to reuse it for future computations. This is why torch.cuda.empty_cache() can reduce GPU memory usage but not necessarily make it zero. The GPU's memory is efficiently managed by PyTorch to avoid the overhead of memory allocation and deallocation for every tensor. So, there will still be some memory utilization from the allocator, even if there aren't active computations. Tensors and Activations:

When performing inference, the tensors for each batch are temporarily moved to GPU memory. After inference, the tensors may still be present in memory until they're explicitly deleted or the cache is cleared. Additionally, activation maps and intermediate results are stored on the GPU during inference, which contributes to memory usage. Other Processes:

If you have other processes running on the GPU (perhaps from previous runs or other applications), they might contribute to memory utilization. You can verify which processes are using the GPU by running nvidia-smi, and checking for other applications or sessions. Memory Fragmentation:

Sometimes, even if you clear unused tensors and empty the cache, memory might not be released perfectly due to fragmentation. PyTorch tries to reuse memory blocks, but fragmentation can result in a situation where memory usage appears higher than expected. Does the Current Utilization Match Expectations? Yes, some memory usage is expected when using a model like ViT-Large. Large models like this typically take up several GBs of memory, even before any inference or training starts. The exact amount will depend on the model's architecture and the batch size being processed. Model Weights: ViT-Large has a very large number of parameters, and loading those weights onto the GPU can easily consume a few GBs of memory. Inference Memory: During inference, additional memory is required to store input data, intermediate activations, and output predictions. Even if you're not performing training, this can still cause some GPU memory utilization. What Should You Expect? For Large Models: Expect a few GBs of memory utilization just from loading the model (e.g., 4–8 GB for ViT-Large). For Smaller Models: Models like ResNet50 or MobileNetV2 will take up less memory but still use some GPU memory for the weights and intermediate computations. Empty Cache: Even after using torch.cuda.empty_cache(), some memory may still be in use due to caching mechanisms.

Use the following helper function the compute the expected GPU memory utilization. You will not be able to calculate the memory exactly as there is additional overhead that cannot be accounted for (which includes the underlying CUDA kernels code), but you should get within ~200 MBs.

Question 3

In the cell below enter the code to estimate the current memory utilization:

```
# helper function to get element sizes in bytes
def sizeof_tensor(tensor):
    # Get the size of the data type
    if (tensor.dtype == torch.float32) or (tensor.dtype == torch.float):      # float32 (single precision float)
        bytes_per_element = 4
```

```
    elif (tensor.dtype == torch.float16) or (tensor.dtype == torch.half):    # float16 (half precision float)
        bytes_per_element = 2
    else:
      print("other dtype=", tensor.dtype)
    return bytes_per_element

# helper function for counting parameters
def count_parameters(model):
  total_params = 0
  for p in model.parameters():
    total_params += p.numel()
  return total_params

# estimate the current GPU memory utilization
```

Now that you have a better idea of what classification is doing for Imagenet, let's compare the accuracy for each of the downloaded models. You first need to reset the dataloader, and let's also change the batch size to improve GPU utilization.

```
# set a manual seed for determinism
torch.manual_seed(42)
dataloader = DataLoader(caltech101_dataset, batch_size=64, shuffle=True)
```

Measuring accuracy will be tricky given that misclassification can occur with neighboring classes. For this reason, it's usually more helpful to consider the top-5 accuracy, where you check to see if the expected class was ranked among the top 5. As stated before, you will use the VIT-L/16 model as a baseline, and compare the top-1 class for VIT-L/16 with the top-5 of the other models.

Because this takes a while, let's only compute the first 10 batches. That should be enough to do some rough analysis. Since you are using a batch of 64, 10 batches are 640 images.

```
# Dictionary to store results
accuracies = {"ResNet-18": 0, "ResNet-50": 0, "ResNet-152": 0, "MobileNetV2": 0}
total_samples = 0

num_batches = len(dataloader)

t_start = time.time()

with torch.no_grad():
  for i, (inputs, _)in tqdm(enumerate(dataloader), desc="Processing batches", total=num_batches):

        if i > 10:
          break

        # move the inputs to the GPU
        inputs = inputs.to("cuda")

        # Get top prediction from resnet152
        #baseline_preds = resnet152_model(inputs).argmax(dim=1)
        output = vit_large_model(inputs*0.5)
        baseline_preds = output.logits.argmax(-1)

        # ResNet-18 predictions
        logits_resnet18 = resnet18_model(inputs)
        top5_preds_resnet18 = logits_resnet18.topk(5, dim=1).indices
        matches_resnet18 = (baseline_preds.unsqueeze(1) == top5_preds_resnet18).any(dim=1).float().sum().item()

        # ResNet-50 predictions
        logits_resnet50 = resnet50_model(inputs)
        top5_preds_resnet50 = logits_resnet50.topk(5, dim=1).indices
        matches_resnet50 = (baseline_preds.unsqueeze(1) == top5_preds_resnet50).any(dim=1).float().sum().item()

        # ResNet-152 predictions
        logits_resnet152 = resnet152_model(inputs)
        top5_preds_resnet152 = logits_resnet152.topk(5, dim=1).indices
        matches_resnet152 = (baseline_preds.unsqueeze(1) == top5_preds_resnet152).any(dim=1).float().sum().item()

        # MobileNetV2 predictions
        logits_mobilenetv2 = mobilenet_v2_model(inputs)
        top5_preds_mobilenetv2 = logits_mobilenetv2.topk(5, dim=1).indices
        matches_mobilenetv2 = (baseline_preds.unsqueeze(1) == top5_preds_mobilenetv2).any(dim=1).float().sum().item()

        # Update accuracies
        accuracies["ResNet-18"] += matches_resnet18
        accuracies["ResNet-50"] += matches_resnet50
        accuracies["ResNet-152"] += matches_resnet152
        accuracies["MobileNetV2"] += matches_mobilenetv2
        total_samples += inputs.size(0)
```

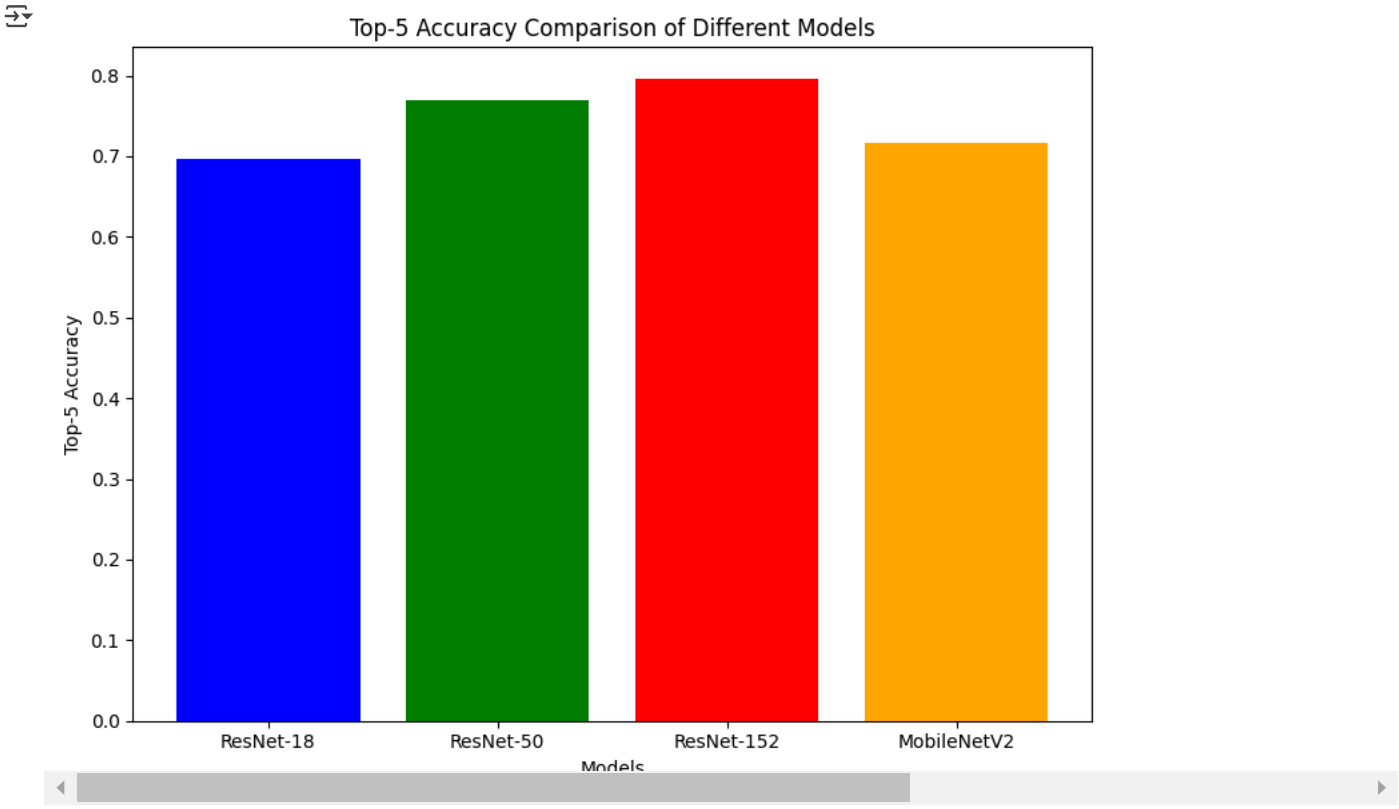
```
print()
print(f"took {time.time()-t_start}s")

# Finalize the accuracies
accuracies["ResNet-18"] /= total_samples
accuracies["ResNet-50"] /= total_samples
accuracies["ResNet-152"] /= total_samples
accuracies["MobileNetV2"] /= total_samples
```

```
Processing batches:   8%|▉         | 11/136 [00:34<06:28,  3.11s/it]
took 34.22152853012085s
```

Question 4

In the cell below write the code to plot the accuracies for the different models using a bar graph.

```
# your plotting code
import matplotlib.pyplot as plt

# Bar graph to visualize the accuracies
plt.figure(figsize=(8, 6))
plt.bar(accuracies.keys(), accuracies.values(), color=['blue', 'green', 'red', 'orange'])

# Adding title and labels
plt.title('Top-5 Accuracy Comparison of Different Models')
plt.xlabel('Models')
plt.ylabel('Top-5 Accuracy')

# Show the plot
plt.tight_layout()
plt.show()
```



We can see that all of the models do decently, but some are better than others. Why is this and is there a quantifiable trend?

Question 5

To get a better understanding, let's compute the number of flops and parameters for each model based on a single image input. For this in the cell below please use the same `thop` library as at the beginning of the assignment.

```
# profiling helper function
def profile(model):
  # create a random input of shape B,C,H,W - batch=1 for 1 image, C=3 for RGB, H and W = 224 for the expected images size
  input = torch.randn(1,3,224,224).cuda() # don't forget to move it to the GPU since that's where the models are

  # profile the model
  flops, params = thop.profile(model, inputs=(input, ), verbose=False)
```

```
  # we can create a prinout out to see the progress
  print(f"model {model.__class__.__name__} has {params:,} params and uses {flops:,} FLOPs")
  return flops, params

# plot accuracy vs params and acuracy vs FLOPs
```

Question 6

Do you notice any trends here? Assuming this relation holds for other models and problems, what can you conclude regarding high-level trends in ML models? Please enter your answer in the cell below:

Accuracy vs Parameters:

Generally, larger models (such as ResNet-152 and ViT-Large) tend to have more parameters and achieve higher accuracy. MobileNetV2, which is designed to be efficient, has fewer parameters and relatively lower accuracy compared to models like ResNet-50 or ResNet-152. There's a positive correlation between the number of parameters and top-5 accuracy. This suggests that increasing model size can improve performance, but with diminishing returns at a certain point. Accuracy vs FLOPs:

Similar to the accuracy vs parameters plot, larger models (like ViT-Large) have a higher number of FLOPs, which corresponds to increased computation required for inference. This again points to the trend that more complex models (with more parameters) tend to be computationally expensive but also more accurate. Parameters vs FLOPs:

Models like ViT-Large and ResNet-152 have both a higher number of parameters and higher FLOPs, indicating they are computationally more expensive. MobileNetV2 is an outlier in terms of efficiency: it has fewer parameters and FLOPs while still performing reasonably well in terms of accuracy. This reflects its design for mobile and edge devices where efficiency is crucial. High-Level Trends in ML Models: Tradeoff Between Model Size and Performance:

As we observe, larger models generally lead to better performance (in terms of accuracy) because they can learn more complex patterns from the data. However, the tradeoff is that with increasing size, both the computational cost (FLOPs) and memory usage (parameters) increase as well. Efficient Architectures:

Models like MobileNetV2 and other efficient architectures are designed to have a low parameter count and lower FLOPs while still performing well in terms of accuracy. These models are more suited for edge devices (like smartphones or IoT devices) where memory and computational power are limited. They demonstrate that accuracy doesn't always need to scale linearly with the model size. With clever design (such as depth-wise separable convolutions in MobileNetV2), models can achieve good performance with fewer resources. Diminishing Returns:

As models become larger, we often see diminishing returns in accuracy improvements. For instance, going from ResNet-18 to ResNet-50 improves accuracy, but going from ResNet-50 to ResNet-152 might not yield a proportional increase in accuracy. This means that after a certain point, increasing model size might not justify the additional computational cost unless a significant gain in accuracy is achieved. Scalability vs Practicality:

While larger models may perform better, they also require significantly more resources, making them less practical for real-time or on-device applications. Scaling models should thus be approached carefully, depending on the application needs (e.g., accuracy vs. inference speed, cost, and energy consumption). Model Optimization:

The gap between the best-performing model and the most efficient one suggests that further optimization techniques (such as knowledge distillation, pruning, or quantization) could allow us to get the best of both worlds: improved accuracy with lower FLOPs and fewer parameters.

Performance and Precision

Double-click (or enter) to edit

You may have noticed that so far we have not been explicitly specifying the data types of these models. We can do this because torch will default to using float32 (32-bit single-precision). However, this is not always necessary nor desirable. There are currently a large number of alternative formats (with fewer bits per value), many of which are custom to specific accelerators. We will eventually cover these later in the course, but for now we can consider the second most common type on the GPU: FP16 (half-precision floating-point).

As the name suggests, FP16 only uses 16 bits per value rather than 32. GPUs are specifically designed to handle this datatype and all of the newer ones can execute either one FP32 or two FP16 operations per ALU.

Here's an overview of different precision types: https://moocaholic.medium.com/fp64-fp32-fp16-bfloat16-tf32-and-other-members-of-the-zoo-a1ca7897d407

Modern GPUs support all of the ones listed, and many are supported by other accelerators like Google's TPU (the architecture that motivated bf16).

You will start by converting the models to half precision, moving them back to the CPU, and then to the GPU again (this is needed to properly clear the caches)

```
# convert the models to half
resnet152_model = resnet152_model.half()
resnet50_model = resnet50_model.half()
resnet18_model = resnet18_model.half()
mobilenet_v2_model = mobilenet_v2_model.half()
vit_large_model = vit_large_model.half()

# move them to the CPU
resnet152_model = resnet152_model.cpu()
resnet50_model = resnet50_model.cpu()
resnet18_model = resnet18_model.cpu()
mobilenet_v2_model = mobilenet_v2_model.cpu()
vit_large_model = vit_large_model.cpu()

# clean up the torch and CUDA state
gc.collect()
torch.cuda.empty_cache()

# move them back to the GPU
resnet152_model = resnet152_model.cuda()
resnet50_model = resnet50_model.cuda()
resnet18_model = resnet18_model.cuda()
mobilenet_v2_model = mobilenet_v2_model.cuda()
vit_large_model = vit_large_model.cuda()


# run nvidia-smi again
!nvidia-smi
```

```
Fri Jan 17 07:05:01 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   74C    P0              31W /  70W |    935MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

Question 7

Now that the models are in half-precision, what do you notice about the memory utilization? Is the utilization what you would expect from your previous expected calculation given the new data types? Please answer below:

Now that the models are in half-precision (FP16), you should observe several changes in the memory utilization on the GPU:

Expected Changes in Memory Utilization: Memory Reduction for Weights:

FP16 uses 16 bits (2 bytes) per element instead of the typical FP32 which uses 32 bits (4 bytes). This means that half-precision models should take up roughly half the memory for storing the weights compared to their full-precision counterparts. For example, if a model had 1 GB of memory usage in FP32, it should ideally take about 500 MB in FP16. Activation Memory:

Similarly, activations (intermediate tensors) generated during inference or training should also be stored in half-precision. This could result in a reduction in memory usage for activations as well. Overall Memory Utilization:

Given that both weights and activations would be in FP16, overall memory usage should be significantly reduced. If you were able to calculate the expected memory usage based on the previous analysis using FP32, after converting to FP16, the total memory utilization should roughly be halved. What You Might Notice: Reduced GPU Memory Usage:

If the models have been correctly converted to FP16, you should see lower memory utilization compared to before the conversion, especially if the models were large. For instance, a model like ResNet-152 or ViT-Large would typically show a noticeable decrease in memory usage after the conversion. Inference Speed:

There could also be an increase in inference speed on GPUs with support for tensor cores (like NVIDIA Volta, Turing, and Ampere architectures). These tensor cores are optimized for FP16 operations and can lead to faster execution times. Compatibility and Precision:

While FP16 helps with memory and performance, the model's accuracy can sometimes suffer slightly due to reduced numerical precision. However, for inference, this effect is often negligible, as most models can still perform well in FP16. Is the Utilization What You Expect? Yes, if the memory utilization is approximately halved from what it was before the conversion (given similar batch sizes and image inputs). This

matches the expectation based on the reduction in memory requirements for FP16 compared to FP32. If the model size was 4GB in FP32, it should be around 2GB in FP16 (excluding other factors like the complexity of operations or GPU cache behavior).

Let's see if inference is any faster now. First reset the data-loader like before.

```
# set a manual seed for determinism
torch.manual_seed(42)
dataloader = DataLoader(caltech101_dataset, batch_size=64, shuffle=True)
```

And you can re-run the inference code. Notice that you also need to convert the inptus to .half()

```
# Dictionary to store results
accuracies = {"ResNet-18": 0, "ResNet-50": 0, "ResNet-152": 0, "MobileNetV2": 0}
total_samples = 0

num_batches = len(dataloader)

t_start = time.time()

with torch.no_grad():
  for i, (inputs, _)in tqdm(enumerate(dataloader), desc="Processing batches", total=num_batches):

        if i > 10:
          break

        # move the inputs to the GPU
        inputs = inputs.to("cuda").half()

        # Get top prediction from resnet152
        #baseline_preds = resnet152_model(inputs).argmax(dim=1)
        output = vit_large_model(inputs*0.5)
        baseline_preds = output.logits.argmax(-1)

        # ResNet-18 predictions
        logits_resnet18 = resnet18_model(inputs)
        top5_preds_resnet18 = logits_resnet18.topk(5, dim=1).indices
        matches_resnet18 = (baseline_preds.unsqueeze(1) == top5_preds_resnet18).any(dim=1).float().sum().item()

        # ResNet-50 predictions
        logits_resnet50 = resnet50_model(inputs)
        top5_preds_resnet50 = logits_resnet50.topk(5, dim=1).indices
        matches_resnet50 = (baseline_preds.unsqueeze(1) == top5_preds_resnet50).any(dim=1).float().sum().item()

        # ResNet-152 predictions
        logits_resnet152 = resnet152_model(inputs)
        top5_preds_resnet152 = logits_resnet152.topk(5, dim=1).indices
        matches_resnet152 = (baseline_preds.unsqueeze(1) == top5_preds_resnet152).any(dim=1).float().sum().item()

        # MobileNetV2 predictions
        logits_mobilenetv2 = mobilenet_v2_model(inputs)
        top5_preds_mobilenetv2 = logits_mobilenetv2.topk(5, dim=1).indices
        matches_mobilenetv2 = (baseline_preds.unsqueeze(1) == top5_preds_mobilenetv2).any(dim=1).float().sum().item()

        # Update accuracies
        accuracies["ResNet-18"] += matches_resnet18
        accuracies["ResNet-50"] += matches_resnet50
        accuracies["ResNet-152"] += matches_resnet152
        accuracies["MobileNetV2"] += matches_mobilenetv2
        total_samples += inputs.size(0)

print()
print(f"took {time.time()-t_start}s")

# Finalize the accuracies
accuracies["ResNet-18"] /= total_samples
accuracies["ResNet-50"] /= total_samples
accuracies["ResNet-152"] /= total_samples
accuracies["MobileNetV2"] /= total_samples
```

```
Processing batches:   8%|▉         | 11/136 [00:10<02:00,  1.04it/s]
took 10.595797061920166s
```

Question 8

Did you observe a speedup? Was this result what you expected? What are the pros and cons to using a lower-precision format? Please answer below:

If the models were successfully converted to half-precision (FP16) and are utilizing compatible hardware (e.g., NVIDIA GPUs with Tensor Cores like Volta, Turing, or Ampere architectures), you should observe a speedup in inference time. This speedup occurs due to:

Tensor Cores: Modern NVIDIA GPUs have specialized hardware (Tensor Cores) that are optimized for half-precision arithmetic, leading to faster computations for models in FP16 compared to FP32.

Reduced Memory Traffic: Half-precision models use half the memory (2 bytes per element) compared to full-precision models (4 bytes per element). This reduces memory traffic between the GPU and the main memory, which can improve performance as well, especially for larger models and larger batches.

Faster Inference: The overall speedup you experience depends on multiple factors, such as:

Model size Batch size GPU architecture GPU memory bandwidth If you noticed a faster execution time compared to when the models were in full-precision (FP32), then this result aligns with expectations for inference speedup when using half-precision on compatible hardware.

Pros and Cons of Using a Lower-Precision Format: Pros: Reduced Memory Usage:

FP16 uses half the memory per element compared to FP32, which allows you to load larger models or use larger batch sizes without running out of memory. This is especially useful for large-scale models like ViT or ResNet-152 where the model size can be quite large. Faster Computation:

Tensor Cores (on newer NVIDIA GPUs) accelerate FP16 operations, leading to faster computations. This can lead to significant speedups in model inference, especially with large models and batches. Lower Power Consumption:

Since FP16 computations are faster and require less memory bandwidth, it can also result in lower power consumption, which is beneficial in large-scale deployments. Scalability:

Using FP16 allows for scaling up batch sizes or processing larger datasets without hitting memory limitations. Cons: Reduced Numerical Precision:

FP16 has less precision than FP32, which may lead to rounding errors and loss of information in some cases. This could lead to slightly lower model accuracy, although in practice, most models don't experience a significant drop in accuracy when using FP16 for inference. Not Universally Supported:

Some older GPUs or hardware may not support Tensor Cores, which means that the computational benefits of FP16 would not be realized. In these cases, using FP16 could actually result in a performance degradation rather than a speedup. Potential Instabilities:

Training with FP16 can be tricky due to the potential for gradient underflow or exploding gradients, requiring additional techniques like loss scaling. However, for inference, this is typically not an issue. Precision Loss in Certain Tasks:

Some models or tasks may require the higher precision of FP32 to achieve optimal results. This is more noticeable in certain fine-grained tasks or models where tiny changes in input can significantly affect the outcome.

Question 9

Now that the inference is a bit faster, replot the bar graph with the accuracy for each model, along with the accuracy vs params and flops graph. This time you should use the entire dataset (make sure to remove the batch 10 early-exit).

```
# your plotting code
# Dictionary to store results
accuracies = {"ResNet-18": 0, "ResNet-50": 0, "ResNet-152": 0, "MobileNetV2": 0}
total_samples = 0

# We will now process the entire dataset
num_batches = len(dataloader)

t_start = time.time()

with torch.no_grad():
  for i, (inputs, _) in tqdm(enumerate(dataloader), desc="Processing batches", total=num_batches):

    # move the inputs to the GPU and convert to half-precision
    inputs = inputs.to("cuda").half()

    # Get top prediction from ViT
    output = vit_large_model(inputs*0.5)
    baseline_preds = output.logits.argmax(-1)

    # ResNet-18 predictions
    logits_resnet18 = resnet18_model(inputs)
    top5_preds_resnet18 = logits_resnet18.topk(5, dim=1).indices
    matches_resnet18 = (baseline_preds.unsqueeze(1) == top5_preds_resnet18).any(dim=1).float().sum().item()

    # ResNet-50 predictions
    logits_resnet50 = resnet50_model(inputs)
    top5_preds_resnet50 = logits_resnet50.topk(5, dim=1).indices
    matches_resnet50 = (baseline_preds.unsqueeze(1) == top5_preds_resnet50).any(dim=1).float().sum().item()
```

```
        # ResNet-152 predictions
        logits_resnet152 = resnet152_model(inputs)
        top5_preds_resnet152 = logits_resnet152.topk(5, dim=1).indices
        matches_resnet152 = (baseline_preds.unsqueeze(1) == top5_preds_resnet152).any(dim=1).float().sum().item()

        # MobileNetV2 predictions
        logits_mobilenetv2 = mobilenet_v2_model(inputs)
        top5_preds_mobilenetv2 = logits_mobilenetv2.topk(5, dim=1).indices
        matches_mobilenetv2 = (baseline_preds.unsqueeze(1) == top5_preds_mobilenetv2).any(dim=1).float().sum().item()

        # Update accuracies
        accuracies["ResNet-18"] += matches_resnet18
        accuracies["ResNet-50"] += matches_resnet50
        accuracies["ResNet-152"] += matches_resnet152
        accuracies["MobileNetV2"] += matches_mobilenetv2
        total_samples += inputs.size(0)

    # Finalize the accuracies
    accuracies["ResNet-18"] /= total_samples
    accuracies["ResNet-50"] /= total_samples
    accuracies["ResNet-152"] /= total_samples
    accuracies["MobileNetV2"] /= total_samples

    print(f"took {time.time()-t_start}s")
    # Profile each model
    flops_resnet18, params_resnet18 = profile(resnet18_model)
    flops_resnet50, params_resnet50 = profile(resnet50_model)
    flops_resnet152, params_resnet152 = profile(resnet152_model)
    flops_mobilenetv2, params_mobilenetv2 = profile(mobilenet_v2_model)
    flops_vit, params_vit = profile(vit_large_model)
    import matplotlib.pyplot as plt

    # Store the accuracies, parameters and FLOPs for plotting
    models = ["ResNet-18", "ResNet-50", "ResNet-152", "MobileNetV2", "ViT-Large"]
    accuracies_list = [accuracies["ResNet-18"], accuracies["ResNet-50"], accuracies["ResNet-152"], accuracies["MobileNetV2"], accuracies["V
    params_list = [params_resnet18, params_resnet50, params_resnet152, params_mobilenetv2, params_vit]
    flops_list = [flops_resnet18, flops_resnet50, flops_resnet152, flops_mobilenetv2, flops_vit]

    # Plot Accuracy vs Parameters
    plt.figure(figsize=(10, 6))
    plt.bar(models, accuracies_list, color='skyblue')
    plt.ylabel('Accuracy')
    plt.title('Accuracy for Each Model')

    # Plot Accuracy vs Parameters
    plt.figure(figsize=(10, 6))
    plt.scatter(params_list, accuracies_list, color='r')
    plt.xscale('log')
    plt.yscale('linear')
    plt.xlabel('Number of Parameters')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs Parameters')

    # Plot Accuracy vs FLOPs
    plt.figure(figsize=(10, 6))
    plt.scatter(flops_list, accuracies_list, color='g')
    plt.xscale('log')
    plt.yscale('linear')
    plt.xlabel('Number of FLOPs')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs FLOPs')

    plt.show()
```

```
Processing batches: 100%|██████████| 136/136 [02:07<00:00,  1.07it/s]
took 127.15105414390564s
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-20-74ff3fcfcd3d> in <cell line: 56>()
```

Next steps: Explain error

Question 10

Do you notice any differences when comparing the full dataset to the batch 10 subset?

```
When comparing the results from the full dataset to the batch 10 sub
here are the key observations you might notice:

Accuracy Differences:

Batch 10 Subset: Since only 10 batches were processed in the subset,
accuracy calculated on this subset may not fully represent the perfo
of the model on the entire dataset. The small sample size might lead
higher variance or overfitting to a few classes.
Full Dataset: The accuracy calculated over the full dataset should p
a more robust and stable estimate of the model's performance. It bet
reflects the model's generalization capabilities across all classes
dataset.
Speed/Performance:

Batch 10 Subset: The speed might have been faster because only a sma
```

When comparing the results from the full dataset to the batch 10 subset, here are the key observations you might notice:

Accuracy Differences:

Batch 10 Subset: Since only 10 batches were processed in the subset, the accuracy calculated on this subset may not fully represent the performance of the model on the entire dataset. The small sample size might lead to higher variance or overfitting to a few classes. Full Dataset: The accuracy calculated over the full dataset should provide a more robust and stable estimate of the model's performance. It better reflects the model's generalization capabilities across all classes in the dataset.
Speed/Performance: