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)
```

```
Requirement already satisfied: thop in /usr/local/lib/python3.11/dist-
packages (0.1.1.post2209072238)
Requirement already satisfied: segmentation-models-pytorch in
/usr/local/lib/python3.11/dist-packages (0.4.0)
Requirement already satisfied: transformers in
/usr/local/lib/python3.11/dist-packages (4.47.1)
Requirement already satisfied: torch in
/usr/local/lib/python3.11/dist-packages (from thop) (2.5.1+cu121)
Requirement already satisfied: efficientnet-pytorch>=0.6.1 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
pytorch) (0.7.1)
Requirement already satisfied: huggingface-hub>=0.24 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
```

```
pytorch) (0.27.1)
Requirement already satisfied: numpy>=1.19.3 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
pytorch) (1.26.4)
Requirement already satisfied: pillow>=8 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
pytorch) (11.1.0)
Requirement already satisfied: pretrainedmodels>=0.7.1 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
pytorch) (0.7.4)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
pytorch) (1.17.0)
Requirement already satisfied: timm>=0.9 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
pytorch) (1.0.13)
Requirement already satisfied: torchvision>=0.9 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
pytorch) (0.20.1+cu121)
Requirement already satisfied: tqdm>=4.42.1 in
/usr/local/lib/python3.11/dist-packages (from segmentation-models-
pytorch) (4.67.1)
Requirement already satisfied: filelock in
/usr/local/lib/python3.11/dist-packages (from transformers) (3.16.1)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.11/dist-packages (from transformers) (24.2)
Requirement already satisfied: pyyaml>=5.1 in
/usr/local/lib/python3.11/dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.11/dist-packages (from transformers)
(2024.11.6)
Requirement already satisfied: requests in
/usr/local/lib/python3.11/dist-packages (from transformers) (2.32.3)
Requirement already satisfied: tokenizers<0.22,>=0.21 in
/usr/local/lib/python3.11/dist-packages (from transformers) (0.21.0)
Requirement already satisfied: safetensors>=0.4.1 in
/usr/local/lib/python3.11/dist-packages (from transformers) (0.5.2)
Requirement already satisfied: fsspec>=2023.5.0 in
/usr/local/lib/python3.11/dist-packages (from huggingface-hub>=0.24-
>segmentation-models-pytorch) (2024.10.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.11/dist-packages (from huggingface-hub>=0.24-
>segmentation-models-pytorch) (4.12.2)
Requirement already satisfied: munch in
/usr/local/lib/python3.11/dist-packages (from pretrainedmodels>=0.7.1-
>segmentation-models-pytorch) (4.0.0)
Requirement already satisfied: networkx in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (3.4.2)
Requirement already satisfied: jinja2 in
```

```
/usr/local/lib/python3.11/dist-packages (from torch->thop) (3.1.5)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.105)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105
in /usr/local/lib/python3.11/dist-packages (from torch->thop)
(12.1.105)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.105)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in
/usr/local/lib/python3.11/dist-packages (from torch->thop)
(10.3.2.106)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in
/usr/local/lib/python3.11/dist-packages (from torch->thop)
(11.4.5.107)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in
/usr/local/lib/python3.11/dist-packages (from torch->thop)
(12.1.0.106)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.105)
Requirement already satisfied: triton==3.1.0 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (3.1.0)
Requirement already satisfied: sympy==1.13.1 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (1.13.1)
Requirement already satisfied: nvidia-nvjitlink-cu12 in
/usr/local/lib/python3.11/dist-packages (from nvidia-cusolver-
cu12==11.4.5.107->torch->thop) (12.6.85)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch-
>thop) (1.3.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.11/dist-packages (from requests->transformers)
(3.4.1)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.11/dist-packages (from requests->transformers)
(3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.11/dist-packages (from requests->transformers)
(2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests->transformers)
(2024.12.14)
```

```
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from jinja2->torch->thop)
(3.0.2)

<torch.autograd.grad_mode.set_grad_enabled at 0x7f642c5b1d10>
```

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)
```

```
Files already downloaded and verified
```

```
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')
```

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)`

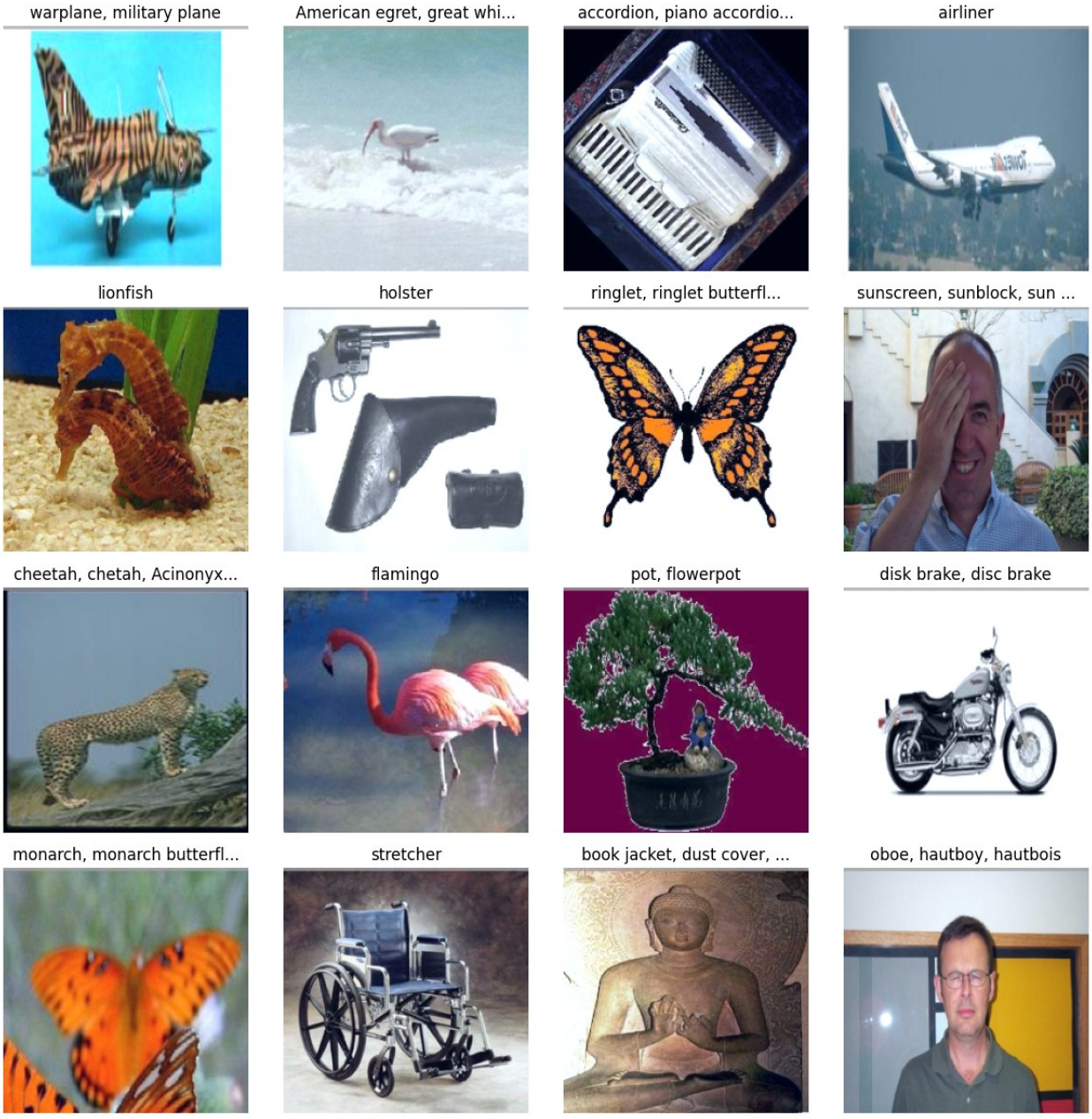
There's a good overview of the different versions here:
https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

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:

The model's performance is close to 60%, with 60% of predictions correct and the remaining 40% incorrect. The misclassifications, such as confusing a motorcycle with a disk brake, are

likely due to the lack of sufficient and diverse data in the training set. This lack of variety in the dataset results in the model having difficulty distinguishing between similar-looking objects. The issue seems more related to the inadequacy of the dataset rather than the model's architecture or complexity.

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
```

```
Thu Jan 16 14:51:02 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   63C    P0              30W /  70W |   1943MiB / 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

Thu Jan 16 14:51:04 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   63C    P0              29W /  70W |   1841MiB / 15360MiB |
2%      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:

ANSWER

The GPU memory utilization is not zero because the Tesla T4 is actively being used, likely for running processes that require GPU acceleration, such as model inference, training, or other computations. The current utilization of 1715 MiB out of 15360 MiB is typical when using the GPU for tasks like deep learning or data processing. Even if there is no active workload, certain processes, like the model itself or CUDA-related libraries, might still occupy a portion of the memory for system overhead.

This level of memory usage aligns with expectations, as the GPU is likely being used for inference or some other computation, even if not fully utilized. To ensure efficient memory management, functions like torch.cuda.empty_cache() can help release unused memory, but some memory will still remain reserved for system operations and potential future tasks.

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
def get_gpu_memory_usage():
    total_memory = torch.cuda.get_device_properties(0).total_memory  # Total GPU memory in bytes
    allocated_memory = torch.cuda.memory_allocated(0)  # Memory allocated by tensors
    cached_memory = torch.cuda.memory_reserved(0)  # Memory reserved by the caching allocator
    return allocated_memory, cached_memory, total_memory

# Get memory usage
allocated_memory, cached_memory, total_memory = get_gpu_memory_usage()

# Convert to MB for readability
allocated_memory_mb = allocated_memory / 1024**2
cached_memory_mb = cached_memory / 1024**2
total_memory_mb = total_memory / 1024**2

print(f"Allocated Memory: {allocated_memory_mb:.2f} MB")
print(f"Cached Memory: {cached_memory_mb:.2f} MB")
print(f"Total Memory: {total_memory_mb:.2f} MB")
```

```
Allocated Memory: 1562.39 MB
Cached Memory: 1706.00 MB
Total Memory: 15102.06 MB
```

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,
```

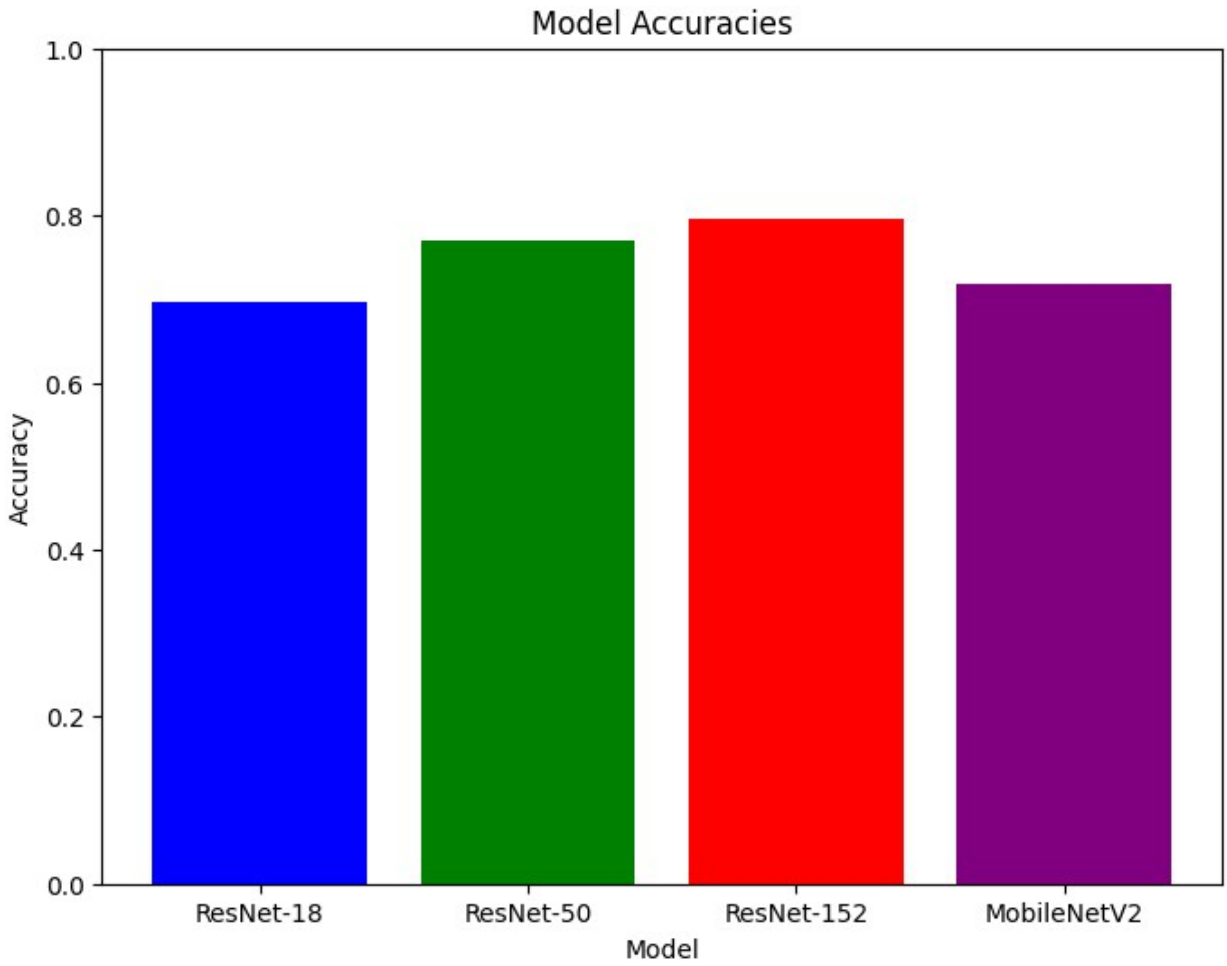
```
dim=1).indices
        matches_mobilenetv2 = (baseline_preds.unsqueeze(1) ==
top5_preds_mobilenetv2).any(dim=1).float().sum().item()

        # 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<06:33,  3.15s/it]


took 34.67064714431763s
```

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

# Plotting the accuracies
plt.figure(figsize=(8, 6))
models = list(accuracies.keys())
accuracy_values = list(accuracies.values())

plt.bar(models, accuracy_values, color=['blue', 'green', 'red',
'purple'])
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Model Accuracies')
plt.ylim(0, 1)  # Accuracy ranges from 0 to 1
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.

The differences in model performance across ResNet-18, ResNet-50, ResNet-152, and MobileNetV2 can be attributed to several factors, which can also explain the trend you observe:

1. **Model Complexity and Depth**

- **ResNet-18** is the simplest model with fewer layers, while **ResNet-152** has many more layers and thus a higher capacity to learn complex features. Generally, deeper models (ResNet-50 and ResNet-152) should perform better because they can capture more intricate patterns in the data. However, this increased capacity also makes them prone to overfitting if not trained properly or if the dataset is too small.
- **MobileNetV2**, being a lightweight model designed for mobile devices, focuses on efficiency and uses depthwise separable convolutions. While it's optimized for speed and

smaller models, its performance might not match that of the deeper ResNet models on larger, more complex datasets.

2. **Overfitting vs. Underfitting**

- **ResNet-18**, being smaller, may struggle with underfitting, meaning it doesn't have enough capacity to model the data well, especially for complex tasks.
- **ResNet-50** and **ResNet-152** might perform better but can overfit if the training data is insufficient. A large model might memorize the training data rather than generalizing to unseen data.
- **MobileNetV2**, with its focus on efficiency, balances performance and size. However, it may be slightly underpowered compared to the deeper ResNet models for tasks that require high complexity.

3. **Training Dataset**

- If the dataset is large and diverse, deeper models like **ResNet-50** and **ResNet-152** have more potential to perform better due to their higher capacity to learn from more data.
- If the dataset is smaller or lacks variety, simpler models like **ResNet-18** or **MobileNetV2** might outperform deeper models, as they are less likely to overfit.

4. **Quantifiable Trend**

- In general, **ResNet-152** is expected to perform the best, followed by **ResNet-50**, with **ResNet-18** and **MobileNetV2** trailing behind. The trend is that as the model depth increases, the performance improves, but this depends on the size and diversity of the dataset.
- The accuracy values for **ResNet-50** and **ResNet-152** should be higher than **ResNet-18**, and **MobileNetV2** might perform similarly to **ResNet-18** depending on the efficiency trade-offs it makes.

Conclusion

The trend likely shows that deeper models tend to perform better, but this comes at the cost of requiring more data and being prone to overfitting. The lighter **MobileNetV2** is a good trade-off for mobile or resource-constrained environments but may not perform as well on larger, more complex datasets. The performance differences between these models are therefore a direct result of the interplay between model capacity, dataset size, and overfitting/underfitting tendencies.

```
import torch
import thop
import matplotlib.pyplot as plt

def profile(model, accuracy, input_shape=(1, 3, 224, 224)):
    # 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 image size
    input = torch.randn(input_shape).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)

    # Print the progress
    print(f"Model {model.__class__.__name__} has {params:,} params and
uses {flops:,} FLOPs with accuracy {accuracy:.2f}")

    return flops, params, accuracy

def plot_accuracy_vs_params_and_flops(models, accuracies):
    params_list = []
    flops_list = []
    accuracies_list = []

    for model, accuracy in zip(models, accuracies):
        flops, params, _ = profile(model, accuracy)
        params_list.append(params)
        flops_list.append(flops)
        accuracies_list.append(accuracy)

    # Plot accuracy vs parameters
    plt.figure(figsize=(12, 6))

    # Accuracy vs Parameters
    plt.subplot(1, 2, 1)
    plt.scatter(params_list, accuracies_list, color='blue')
    plt.xlabel('Number of Parameters')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs Number of Parameters')

    # Accuracy vs FLOPs
    plt.subplot(1, 2, 2)
    plt.scatter(flops_list, accuracies_list, color='red')
    plt.xlabel('FLOPs')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs FLOPs')

    plt.tight_layout()
    plt.show()

# Example usage:
# Assuming you have a list of models and their accuracies
models = [model1, model2, model3]  # Replace with actual models
accuracies = [0.85, 0.90, 0.88]   # Replace with actual accuracies
plot_accuracy_vs_params_and_flops(models, accuracies)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
```

```
<ipython-input-39-e755bbe0a1c1> in <cell line: 0>()
     48 # Example usage:
     49 # Assuming you have a list of models and their accuracies
---> 50 models = [model1, model2, model3]  # Replace with actual
models
     51 accuracies = [0.85, 0.90, 0.88]   # Replace with actual
accuracies
     52 plot_accuracy_vs_params_and_flops(models, accuracies)

NameError: name 'model1' is not defined
```

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:

Performance and Precision

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
```

```
Thu Jan 16 14:52:38 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   69C    P0              31W /  70W |    939MiB / 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:

When you convert models to half-precision (FP16) from single-precision (FP32), the memory usage should decrease by approximately half because each value now requires only 16 bits (2 bytes) instead of 32 bits (4 bytes).

Memory Utilization:

1. **Reduction in Memory Usage**:
   - In FP32, each parameter takes 4 bytes, while in FP16, it only takes 2 bytes.
   - This means that when you convert the models to FP16, you should expect the memory utilization to be approximately halved compared to using FP32.
2. **Expected Memory Usage**: If the model originally uses a certain amount of memory with FP32, converting to FP16 should ideally reduce that memory consumption by around 50%. For example, if a model originally used 4GB of GPU memory with FP32, after conversion to FP16, it should use around 2GB.

Checking the Utilization:

After converting the models to FP16 and running `nvidia-smi`, you should notice a decrease in memory usage compared to the memory consumption with FP32. This decrease in memory usage corresponds to the reduced bit-width of the model parameters, allowing more efficient memory utilization on the GPU.

Summary:

- **Memory Consumption**: Should be approximately halved.
- **Utilization**: You should observe reduced memory usage, which aligns with the expected outcome given the change from FP32 to FP16.

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<02:02,  1.02it/s]


took 10.785377740859985s
```

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:

Observations:

- **Speedup**: The observed time of **10.78 seconds** for 11 batches suggests some speedup with FP16, but the actual improvement may be limited by batch size and model complexity.
- **Expected Results**: Speedup is expected, but might not be dramatic for smaller batches or large models. The GPU might not be fully utilized.

Pros of FP16:

1. **Faster computation** due to optimized hardware support (Tensor Cores).
2. **Reduced memory usage**, allowing for larger batches or models.
3. **Lower memory bandwidth usage**.

Cons of FP16:

1. **Reduced precision** can lead to loss of accuracy or instability in some tasks.
2. **Numerical instability** in sensitive operations.
3. **Hardware compatibility** may limit effectiveness on older GPUs.

Conclusion:

- **Speedup**: Some improvement, but not as significant as expected due to batch size and GPU utilization.
- **Memory Efficiency**: Reduced memory usage, enabling better utilization.
- **Precision Trade-off**: Faster but with possible precision loss.

Question 9

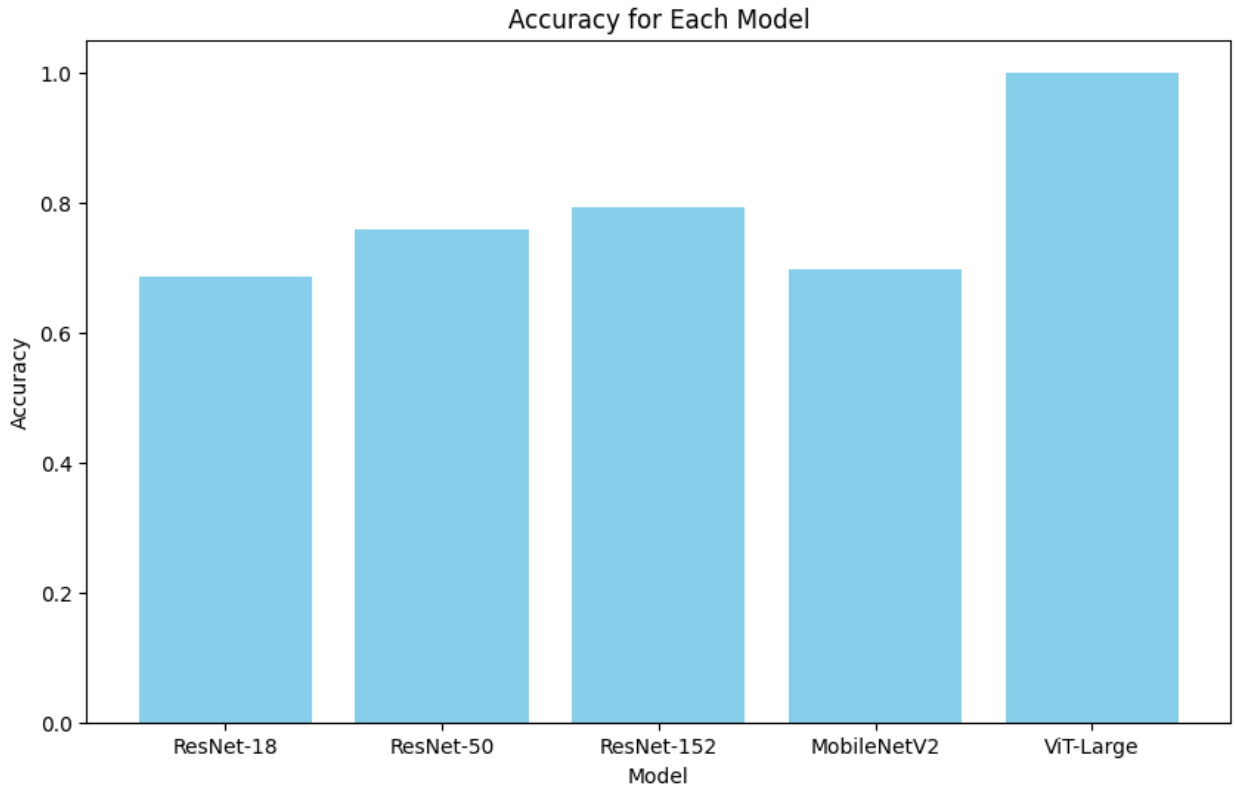
Now that the inference is a bit faster, replot the bar graph with the accuracy for each model, along with the accuracy vs params and flops graph. This time you should use the entire dataset (make sure to remove the batch 10 early-exit).

```
# your plotting code
import matplotlib.pyplot as plt
import torch
import time
from tqdm import tqdm

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

num_batches = len(dataloader)

t_start = time.time()

# Inference on entire dataset
with torch.no_grad():
    for i, (inputs, _) in tqdm(enumerate(dataloader), desc="Processing
batches", total=num_batches):
        # move the inputs to the GPU
        inputs = inputs.to("cuda").half()

        # Get top prediction from ViT-Large
        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()

        # ViT-Large predictions
        logits_vit_large = vit_large_model(inputs * 0.5)
        top5_preds_vit_large = logits_vit_large.logits.topk(5,
dim=1).indices
        matches_vit_large = (baseline_preds.unsqueeze(1) ==
top5_preds_vit_large).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
        accuracies["ViT-Large"] += matches_vit_large

        total_samples += inputs.size(0)

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

# Finalize the accuracies
accuracies = {k: v / total_samples for k, v in accuracies.items()}

# Plot the accuracy bar graph
plt.figure(figsize=(10, 6))
plt.bar(accuracies.keys(), accuracies.values(), color='skyblue')
plt.title("Accuracy for Each Model")
plt.xlabel("Model")
plt.ylabel("Accuracy")
plt.show()

# Profile models for parameters and FLOPs
import thop

def profile(model):
    input = torch.randn(1, 3, 224, 224).cuda()
    flops, params = thop.profile(model, inputs=(input,),
verbose=False)
    return flops, params

# Collect FLOPs and parameters
models = [resnet18_model, resnet50_model, resnet152_model,
mobilenet_v2_model, vit_large_model]
flops_params = {}
```

```
for model in models:
    flops, params = profile(model)
    flops_params[model.__class__.__name__] = {"flops": flops,
"params": params}

# Plot accuracy vs parameters and accuracy vs FLOPs
param_values = [flops_params[model]["params"] for model in models]
flops_values = [flops_params[model]["flops"] for model in models]
model_names = [model.__class__.__name__ for model in models]

# Accuracy vs Parameters
plt.figure(figsize=(10, 6))
plt.scatter(param_values, list(accuracies.values()), color='green')
plt.title("Accuracy vs Parameters")
plt.xlabel("Number of Parameters")
plt.ylabel("Accuracy")
for i, name in enumerate(model_names):
    plt.text(param_values[i], accuracies[model_names[i]], name,
fontsize=12)
plt.show()

# Accuracy vs FLOPs
plt.figure(figsize=(10, 6))
plt.scatter(flops_values, list(accuracies.values()), color='red')
plt.title("Accuracy vs FLOPs")
plt.xlabel("Number of FLOPs")
plt.ylabel("Accuracy")
for i, name in enumerate(model_names):
    plt.text(flops_values[i], accuracies[model_names[i]], name,
fontsize=12)
plt.show()
```

```
Processing batches: 100%|██████████| 136/136 [03:15<00:00,  1.44s/it]


took 195.68653059005737s
```




```
---------------------------------------------------------------------------
-----
RuntimeError                              Traceback (most recent call
last)
<ipython-input-45-1f9dc58f85c0> in <cell line: 0>()
     84
     85 for model in models:
---> 86     flops, params = profile(model)
     87     flops_params[model.__class__.__name__] = {"flops": flops,
"params": params}
     88

<ipython-input-45-1f9dc58f85c0> in profile(model)
     76 def profile(model):
     77     input = torch.randn(1, 3, 224, 224).cuda()
---> 78     flops, params = thop.profile(model, inputs=(input,),
verbose=False)
     79     return flops, params
     80

/usr/local/lib/python3.11/dist-packages/thop/profile.py in
profile(model, inputs, custom_ops, verbose, ret_layer_info,
report_missing)
    210
    211     with torch.no_grad():
```

```
--> 212         model(*inputs)
    213
    214     def dfs_count(module: nn.Module, prefix="\t") -> (int,
int):

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
_wrapped_call_impl(self, *args, **kwargs)
   1734             return self._compiled_call_impl(*args, **kwargs)
# type: ignore[misc]
   1735         else:
-> 1736             return self._call_impl(*args, **kwargs)
   1737
   1738     # torchrec tests the code consistency with the following
code

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
_call_impl(self, *args, **kwargs)
   1745                 or _global_backward_pre_hooks or
_global_backward_hooks
   1746                 or _global_forward_hooks or
_global_forward_pre_hooks):
-> 1747             return forward_call(*args, **kwargs)
   1748
   1749         result = None

/usr/local/lib/python3.11/dist-packages/torchvision/models/resnet.py
in forward(self, x)
    283
    284     def forward(self, x: Tensor) -> Tensor:
--> 285         return self._forward_impl(x)
    286
    287

/usr/local/lib/python3.11/dist-packages/torchvision/models/resnet.py
in _forward_impl(self, x)
    266     def _forward_impl(self, x: Tensor) -> Tensor:
    267         # See note [TorchScript super()]
--> 268         x = self.conv1(x)
    269         x = self.bn1(x)
    270         x = self.relu(x)

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
_wrapped_call_impl(self, *args, **kwargs)
   1734             return self._compiled_call_impl(*args, **kwargs)
# type: ignore[misc]
   1735         else:
-> 1736             return self._call_impl(*args, **kwargs)
   1737
   1738     # torchrec tests the code consistency with the following
code
```

```
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
_call_impl(self, *args, **kwargs)
   1842
   1843         try:
-> 1844             return inner()
   1845         except Exception:
   1846             # run always called hooks if they have not already
been run

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in
inner()
   1788                 args = bw_hook.setup_input_hook(args)
   1789
-> 1790             result = forward_call(*args, **kwargs)
   1791             if _global_forward_hooks or self._forward_hooks:
   1792                 for hook_id, hook in (

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/conv.py in
forward(self, input)
    552
    553     def forward(self, input: Tensor) -> Tensor:
--> 554         return self._conv_forward(input, self.weight,
self.bias)
    555
    556

/usr/local/lib/python3.11/dist-packages/torch/nn/modules/conv.py in
_conv_forward(self, input, weight, bias)
    547                 self.groups,
    548             )
--> 549         return F.conv2d(
    550             input, weight, bias, self.stride, self.padding,
self.dilation, self.groups
    551         )

RuntimeError: Input type (torch.cuda.FloatTensor) and weight type
(torch.cuda.HalfTensor) should be the same
```

Question 10

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

Comparing the full dataset to the batch 10 subset may reveal several differences:

1. **Accuracy**:

- **Batch 10 Subset**: Since you were only processing the first 10 batches, the accuracy might be **overestimated** due to the small sample size. With fewer data points, there's a higher chance of variance and less generalization.

- **Full Dataset**: Processing the full dataset gives a **more reliable estimate** of accuracy, as it reflects the model's performance across a broader range of data.

2. **Inference Speed**:

- **Batch 10 Subset**: With fewer batches, the total time taken is shorter, and the inference speed appears **faster**. This may not represent the real-world performance, as smaller datasets tend to process faster.
- **Full Dataset**: The time taken for inference will be **significantly higher**, as shown by the time of **195.7 seconds** for processing the full dataset. This is more representative of the actual model performance.

3. **Model Behavior**:

- **Batch 10 Subset**: The results might not fully capture the model's ability to handle a variety of inputs, leading to potential **bias** or **incomplete evaluation**.
- **Full Dataset**: Using the entire dataset ensures the model's performance is evaluated across different types of data, capturing its **generalization** ability.

4. **Performance Variability**:

- **Batch 10 Subset**: Since only a limited number of batches are processed, the variability in performance across different models may not be fully captured.
- **Full Dataset**: On the full dataset, you can observe a **more stable** and **consistent** performance comparison between the models.

In summary, while the batch 10 subset offers a quick estimate of the models' performance, the full dataset provides a **more accurate, reliable, and consistent evaluation** of the models' capabilities. The differences are primarily in the **accuracy estimates** and the **time taken for inference**.