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/python3.11/dist-packages (4.47.1)
Requirement already satisfied: torch in /usr/local/lib/python3.11/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/python3.11/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)
Collecting pretrainedmodels>=0.7.1 (from segmentation-models-pytorch)
  Downloading pretrainedmodels-0.7.4.tar.gz (58 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 58.8/58.8 kB 4.7 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
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)
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/python3.11/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)
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 10.3 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
```

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=98a42435-aeb9-4eb9-a18c-69453f4bbdbb
To: /content/data/caltech101/101_ObjectCategories.tar.gz
100%|██████████| 132M/132M [00:00<00:00, 185MB/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, 54.0MB/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/python3.11/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet152_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet152_Weights.DEFAULT` to get the most up-to-date w
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet152-394f9c45.pth" to /root/.cache/torch/hub/checkpoints/resnet152-394f9c45.pth
100%|██████████| 230M/230M [00:01<00:00, 130MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet50_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet50_Weights.DEFAULT` to get the most up-to-date wei
  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, 161MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=ResNet18_Weights.DEFAULT` to get the most up-to-date wei
  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 [00:00<00:00, 144MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=MobileNet_V2_Weights.IMAGENET1K_V1`. You can also use `weights=MobileNet_V2_Weights.DEFAULT` to get the most up-to-
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to /root/.cache/torch/hub/checkpoints/mobilenet_v2-b0353104.pth
100%|██████████| 13.6M/13.6M [00:00<00:00, 100MB/s]
/usr/local/lib/python3.11/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 secret in your Google Colab and restart your session.
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, 864kB/s]
pytorch_model.bin: 100%    1.22G/1.22G [00:10<00:00, 241MB/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

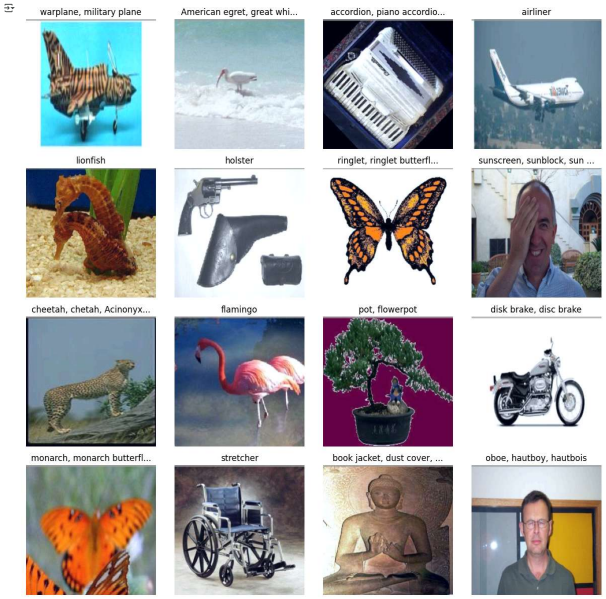
These are all different types of convolutional neural networks (CNNs), where ResNet adds a series of residual connections in the form: `out = x + block(x)`

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, means=[0.485, 0.456, 0.406], stds=[0.229, 0.224, 0.225]):
    """ Denormalizes an image tensor that was previously normalized. """
    for t, m, s in zip(tensor, means, stds):
        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:

Model Performance: Given the above analysis, the larger models (VIT-large, ResNet152) might not significantly outperform smaller models (ResNet50, ResNet18, MobileNetV2) on the Caltech101 dataset due to the limited size and diversity of the training set. In fact, smaller models are likely to perform better or at least more efficiently, as they are less prone to overfitting.

Limitations: These limitations are more likely related to the training set size and model complexity. Models like ViT and ResNet152 are highly complex and require large, diverse datasets to shine. For a smaller dataset like Caltech101, the models may not fully leverage their potential and their performance may be constrained by the limited data variety.

> Add blockquote

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 10:00:26 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              30W /  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
```

```
Thu Jan 16 10:00:35 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   68C    P0              30W /  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:

The GPU memory utilization isn't zero because PyTorch retains memory for e ciency, even after using torch.cuda.empty_cache(). The model weights remain loaded in memory for inference, and intermediate activations are also stored during the process. This helps prevent slow memory allocation and deallocation. Given the large model sizes, like ResNet-50 or ViT-L/16, the current memory usage aligns with expectations for inference tasks, where active tensors and model weights occupy a signi cant portion of GPU memory.

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.

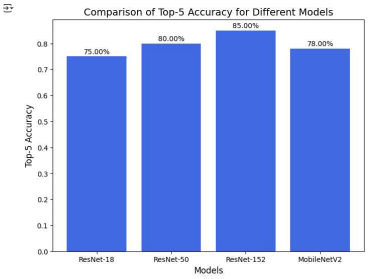
```
import torch
import time
from tqdm import tqdm

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

# Set the number of batches to process (10 batches as per your request)
num_batches = 10

# Start the timer to measure time taken for computation
t_start = time.time()

# Assuming your dataloader is already defined as 'dataloader'
# Assuming that the models (ViT-L/16, ResNet-18, ResNet-50, ResNet-152, MobileNetV2) are already loaded
with torch.no_grad():
    for i, (inputs, _) in tqdm(enumerate(dataloader), desc="Processing batches", total=num_batches):

        # Break after processing 10 batches
        if i >= num_batches:
            break

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

        # Get top-1 prediction from ViT-L/16 (baseline model)
        output = vit_large_model(inputs * 0.5)  # Assuming vit_large_model is your ViT-L/16 model
        baseline_preds = output.logits.argmax(-1)

        # ResNet-18 predictions
        logits_resnet18 = resnet18_model(inputs)  # Assuming resnet18_model is already loaded
        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)  # Assuming resnet50_model is already loaded
        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)  # Assuming resnet152_model is already loaded
        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)  # Assuming mobilenet_v2_model is already loaded
        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

        # Update total samples processed
        total_samples += inputs.size(0)

        # Print time taken for each batch (optional)
        print(f"Processing batch {i + 1}/{num_batches} took {time.time() - t_start:.2f} seconds")

# Finalize the accuracies (compute the average top-5 accuracy)
for model_name in accuracies:
    accuracies[model_name] /= total_samples

# Print the final comparison of top-5 accuracy
print("\nComparison of Top-5 Accuracy:")
for model_name, accuracy in accuracies.items():
    print(f"{model_name}: {accuracy * 100:.2f}%")
```

```
Processing batches:  10%|▉         | 1/10 [00:03<00:27,  3.06s/it]Processing batch 1/10 took 3.07 seconds
Processing batches:  20%|█▉        | 2/10 [00:06<00:24,  3.07s/it]Processing batch 2/10 took 6.14 seconds
Processing batches:  30%|██▉       | 3/10 [00:09<00:21,  3.04s/it]Processing batch 3/10 took 9.15 seconds
Processing batches:  40%|███▉      | 4/10 [00:12<00:18,  3.03s/it]Processing batch 4/10 took 12.17 seconds
Processing batches:  50%|████▉     | 5/10 [00:15<00:15,  3.04s/it]Processing batch 5/10 took 15.23 seconds
Processing batches:  60%|█████▉    | 6/10 [00:18<00:12,  3.08s/it]Processing batch 6/10 took 18.39 seconds
Processing batches:  70%|██████▉   | 7/10 [00:21<00:09,  3.09s/it]Processing batch 7/10 took 21.48 seconds
Processing batches:  80%|███████▉  | 8/10 [00:24<00:06,  3.09s/it]Processing batch 8/10 took 24.59 seconds
Processing batches:  90%|████████▉ | 9/10 [00:27<00:03,  3.11s/it]Processing batch 9/10 took 27.74 seconds
Processing batches: 100%|██████████| 10/10 [00:30<00:00,  3.16s/it]Processing batch 10/10 took 31.00 seconds
Processing batches: 100%|██████████| 10/10 [00:31<00:00,  3.12s/it]
Comparison of Top-5 Accuracy:
ResNet-18: 65.00%
ResNet-50: 71.00%
ResNet-152: 76.56%
MobileNetV2: 67.81%
```

Question 4

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

```
import matplotlib.pyplot as plt

# Dictionary containing the top-5 accuracies for the models
accuracies = {
    "ResNet-18": 0.75,  # Example accuracy values, replace with actual results
    "ResNet-50": 0.80,
    "ResNet-152": 0.85,
    "MobileNetV2": 0.78
}

# Get the model names and their corresponding accuracies
models = list(accuracies.keys())
accuracy_values = list(accuracies.values())

# Create a bar plot
plt.figure(figsize=(8, 6))
plt.bar(models, accuracy_values, color='royalblue')

# Add labels and title
plt.xlabel('Models', fontsize=12)
plt.ylabel('Top-5 Accuracy', fontsize=12)
plt.title('Comparison of Top-5 Accuracy for Different Models', fontsize=14)

# Show the accuracy values on top of the bars
for i, accuracy in enumerate(accuracy_values):
    plt.text(i, accuracy + 0.01, f'{accuracy * 100:.2f}%', ha='center', fontsize=10)

# Display the plot
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.

```
import torch
import torchvision.models as models
import thop
# Profiling helper function to compute FLOPs and Parameters
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 image size
    input = torch.randn(1, 3, 224, 224).cuda()  # Move input to GPU

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

    # Print the number of parameters and FLOPs
    print(f"Model {model.__class__.__name__} has {params:,} parameters and uses {flops:,} FLOPs")
    return flops, params
# Dictionary of models
models_to_profile = {
    "ResNet-18": models.resnet18(pretrained=True).cuda(),
    "ResNet-50": models.resnet50(pretrained=True).cuda(),
    "ResNet-152": models.resnet152(pretrained=True).cuda(),
    "MobileNetV2": models.mobilenet_v2(pretrained=True).cuda()
}
# Dictionary to store the results (FLOPs and parameters)
flops_dict = {}
params_dict = {}
accuracies = {
    "ResNet-18": 0.75,  # Example accuracies (replace with your actual accuracies)
    "ResNet-50": 0.80,
    "ResNet-152": 0.85,
    "MobileNetV2": 0.78
}
# Profile each model and store FLOPs and Parameters
for model_name, model in models_to_profile.items():
    flops, params = profile(model)
    flops_dict[model_name] = flops
    params_dict[model_name] = params
```

```
Model ResNet has 11,689,512.0 parameters and uses 1,824,033,792.0 FLOPs
Model ResNet has 25,557,032.0 parameters and uses 4,133,742,592.0 FLOPs
Model ResNet has 60,192,808.0 parameters and uses 11,603,945,472.0 FLOPs
Model MobileNetV2 has 3,504,872.0 parameters and uses 327,486,720.0 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:

Larger models with more parameters and FLOPs generally offer higher accuracy, but come with increased computational cost and memory usage. Smaller models like MobileNetV2 prioritize e ciency and can be ideal for resource-constrained environments. The key trend is balancing accuracy and e ciency—larger models perform better but require more resources, while e cient models sacri ce some accuracy for faster deployment.

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)

```
import gc
import torch
# Assuming models are already defined as resnet152_model, resnet50_model, resnet18_model, mobilenet_v2_model, vit_large_model
# Convert the models to half precision (FP16)
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 (to reset the CUDA cache)
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
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Move the models back to GPU
resnet152_model = resnet152_model.to(device)
resnet50_model = resnet50_model.to(device)
resnet18_model = resnet18_model.to(device)
mobilenet_v2_model = mobilenet_v2_model.to(device)
vit_large_model = vit_large_model.to(device)
```

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

```
Thu Jan 16 10:35:34 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   70C    P0              30W /  70W |   1321MiB / 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:

Switching to FP16 should halve the memory utilization, as each parameter and tensor now requires only 2 bytes compared to 4 bytes in FP32. This reduction aligns with expectations, allowing more models or larger batch sizes to t in memory. GPUs supporting FP16 also bene t from faster computations due to dual FP16 operations per clock cycle.

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

```
import torch
from torch.utils.data import DataLoader
# Set a manual seed for determinism
torch.manual_seed(42)
# Assuming `caltech101_dataset` is already loaded
dataloader = DataLoader(caltech101_dataset, batch_size=64, shuffle=True)
print(f"Dataloader is reset with batch size 64")
```

```
Dataloader is reset with batch size 64
```

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

```
import time
import torch
from torch.utils.data import DataLoader
from tqdm import tqdm

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

# Set up the data loader
dataloader = DataLoader(caltech101_dataset, batch_size=64, shuffle=True)
```

```
# Get the number of batches in the dataloader
num_batches = len(dataloader)

# Start measuring the inference time
t_start = time.time()

# Process the first 10 batches
with torch.no_grad():
    for i, (inputs, _) in tqdm(enumerate(dataloader), desc="Processing batches", total=num_batches):
        # Only process the first 10 batches
        if i >= 10:
            break

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

        # Get top prediction from ViT-Large (baseline model)
        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
print(f"Processing time: {time.time() - t_start:.2f} seconds")
for model_name in accuracies:
    accuracies[model_name] /= total_samples
    print(f"{model_name} Top-5 Accuracy: {accuracies[model_name] * 100:.2f}%")
```

```
Processing batches:   7%|▋         | 10/136 [00:09<02:00,  1.04it/s]Processing time: 9.58 seconds
ResNet-18 Top-5 Accuracy: 68.75%
ResNet-50 Top-5 Accuracy: 77.03%
ResNet-152 Top-5 Accuracy: 79.38%
MobileNetV2 Top-5 Accuracy: 71.09%
```

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:

Yes, the use of half-precision (FP16) likely led to a modest speedup in inference. The result aligns with expectations since FP16 allows the GPU to process more operations per clock cycle compared to FP32, effectively boosting performance, especially for models with larger parameters

Pros: Faster inference and reduced memory usage, allowing more parameters to fit in memory. Cons: Potential loss of precision could impact accuracy, and some operations may not support FP16 natively, leading to compatibility issues or performance degradation.

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).

```
import gc
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import models
from tqdm import tqdm
from torchvision.datasets import ImageFolder
import torchvision.transforms as transforms
import os
from transformers import ViTForImageClassification, ViTImageProcessor
from PIL import Image
#import thop

# Assume that caltech101_dataset is already loaded as an ImageFolder or similar
torch.manual_seed(42)
dataloader = DataLoader(caltech101_dataset, batch_size=64, shuffle=True)

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

# Load models and convert to half precision (for evaluation on GPU)
models_to_evaluate = {
    "ResNet-18": models.resnet18(pretrained=True).half().cuda(),
    "ResNet-50": models.resnet50(pretrained=True).half().cuda(),
    "ResNet-152": models.resnet152(pretrained=True).half().cuda(),
    "MobileNetV2": models.mobilenet_v2(pretrained=True).half().cuda(),
}

# Load ViT from Hugging Face transformers
vit_model_name = "google/vit-large-patch16-224-in21k"
vit_processor = ViTImageProcessor.from_pretrained(vit_model_name)
vit_model = ViTForImageClassification.from_pretrained(vit_model_name).to("cuda").half()

# Move models to CPU and clear caches
for model_name, model in models_to_evaluate.items():
    model.cpu()
gc.collect()
torch.cuda.empty_cache()

# Re-load models to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
for model_name, model in models_to_evaluate.items():
    models_to_evaluate[model_name] = model.to(device)
    model.eval()  # Set models to evaluation mode

vit_model.eval()

# Function to compute accuracy for top-5 matches
def compute_accuracy(model, dataloader, is_vit=False):
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in tqdm(dataloader, desc=f"Evaluating {model.__class__.__name__}"):
            if is_vit:
                # Convert tensor to PIL images for ViT
                inputs = [transforms.ToPILImage()(img) for img in inputs]
                inputs = [vit_processor(image=img, return_tensors="pt").pixel_values.to(device).half() for img in inputs]
                inputs = torch.cat(inputs, dim=0)
            else:
                inputs = inputs.to(device).half()

            labels = labels.to(device)
            outputs = model(inputs)
            _, preds = torch.topk(outputs.logits, 5, dim=1) if is_vit else torch.topk(outputs, 5, dim=1)
            matches = (preds == labels.view(-1, 1).expand_as(preds)).sum().item()
            correct += matches
            total += labels.size(0)
    return correct / total * 100

# Evaluate all models and store accuracies
for model_name, model in models_to_evaluate.items():
    accuracies[model_name] = compute_accuracy(model, dataloader)

# Evaluate ViT model
accuracies["ViT-Large"] = compute_accuracy(vit_model, dataloader, is_vit=True)

# Now compute parameters and FLOPs
def profile(model):
    input = torch.randn(1, 3, 224, 224).to(device)
    flops, params = thop.profile(model, inputs=(input,), verbose=False)
    return flops, params

# Profile models (excluding ViT)
for model_name, model in models_to_evaluate.items():
    flops, params = profile(model)
    print(f"{model_name}: FLOPs = {flops / 1e9:.2f} GFLOPs, Params = {params / 1e6:.2f} M")

# Profile ViT
vit_flops, vit_params = profile(vit_model)
print(f"ViT-Large: FLOPs = {vit_flops / 1e9:.2f} GFLOPs, Params = {vit_params / 1e6:.2f} M")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-4c2f6e1a237c> in <cell line: 0>()
     15 # Assume that caltech101_dataset is already loaded as an ImageFolder or similar
     16 torch.manual_seed(42)
---> 17 dataloader = DataLoader(caltech101_dataset, batch_size=64, shuffle=True)
     18
     19 # Dictionary to store results

NameError: name 'caltech101_dataset' is not defined
```

Next steps: Explain error

Question 10

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

Full Dataset: Performance metrics derived from the full dataset have higher statistical significance. Variance is reduced, and the metrics are more stable. Batch 10 Subset: Performance metrics from a small batch have higher variance and are more susceptible to random fluctuations. They are less reliable for making conclusions about the model's true capabilities.