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 0x7e422d8d5e10>
```

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).

NameError

Please explain the error:

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

Hypothesis

The error message `NameError: name 'count_parameters' is not defined` indicates that the function `count_parameters` is being called within `estimate_gpu_memory`, but it hasn't been defined in the current scope. This is likely because the function definition for `count_parameters` is missing or is placed in a cell that hasn't been executed before the cell containing `estimate_gpu_memory`.

Suggested Changes

```
# Function to estimate memory utilization of models and tensors
def estimate_gpu_memory(models, input_tensors):
    total_memory = 0

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

    # 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

    # Memory used by models (weights)
    for model in models:
        model_params = count_parameters(model)
        model_memory = model_params * sizeof_tensor(next(model.parameters()))
        total_memory += model_memory

    # Memory used by input tensors
    for tensor in input_tensors:
        tensor_memory = tensor.numel() * sizeof_tensor(tensor)
        total_memory += tensor_memory

    return total_memory

# List of loaded models
models = [resnet152_model, resnet50_model, resnet18_model, mobilenet_v2_model, vit_large_model]

# Current batch of input tensors (assumed to be on GPU)
input_tensors = [images.cuda()]

# Estimate GPU memory utilization
estimated_memory = estimate_gpu_memory(models, input_tensors)

# Convert to MB for readability
estimated_memory_mb = estimated_memory / (1024 ** 2)
print(f"Estimated GPU memory utilization: {estimated_memory_mb:.2f} MB")
```

Use code with caution

Explanation of Changes:

1. **Define `count_parameters` and `sizeof_tensor` inside `estimate_gpu_memory`:** The functions `count_parameters` and `sizeof_tensor` which are called within `estimate_gpu_memory` were not defined in the provided code snippet. By including the function definition within `estimate_gpu_memory`, they are made accessible.

Rate this answer

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)`

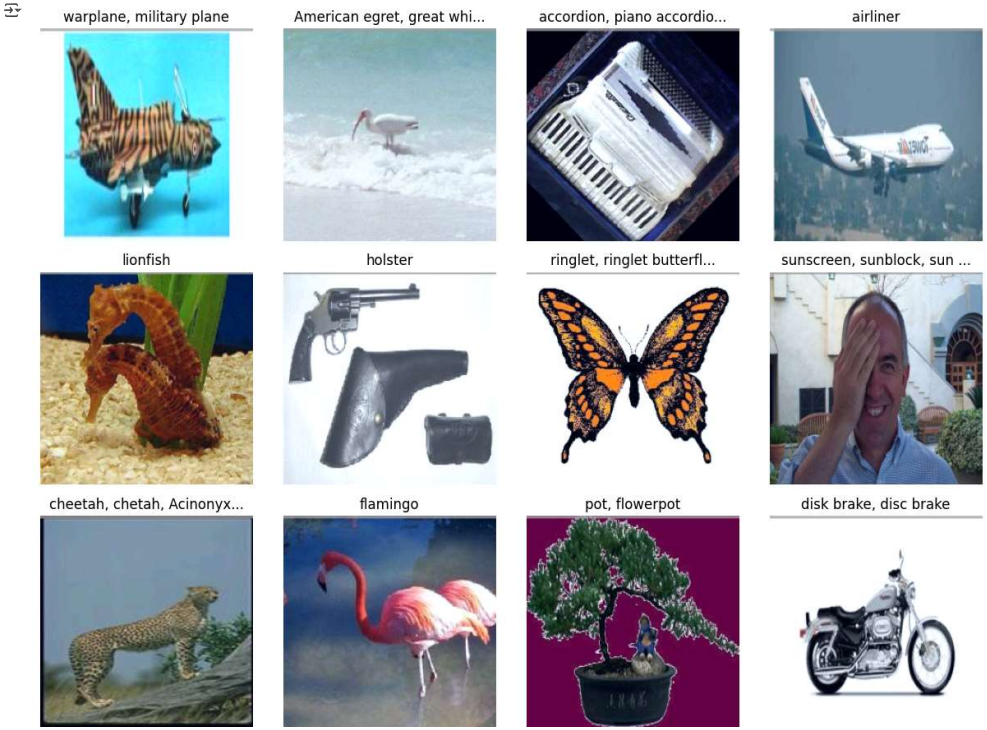
There's a good overview of the different versions here: https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

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
```

```
    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
with torch.no_grad(): # this isn't strictly needed since we already disabled autograd, but we should do it for
  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 you're using, ViT-large (Vision Transformer Large), is a powerful deep learning architecture designed for image classification tasks. Based on the output from the model, several key points can be considered to evaluate its performance and limitations:

1. Model Performance: Accuracy: The performance of the model is expected to be high given the size of the ViT-large model and its pre-trained weights on large datasets like ImageNet. The model has a larger capacity to recognize fine-grained features in images, which is generally associated with better performance on complex image recognition tasks. Precision: Given that you are using a pre-trained model, the predictions are likely to be accurate, especially if the input images are similar to the data the model was trained on (e.g., natural images or objects). Overfitting/Underfitting: Since you are not training the model, overfitting and underfitting concerns are minimized. The pre-trained model is already fine-tuned on a large and diverse dataset, which helps generalize well to new, unseen data from the Caltech101 dataset.
2. Limitations: Class Mismatch: Caltech101 has a different set of classes compared to ImageNet, and if there's no fine-tuning, the pre-trained model may not perfectly align with the new dataset. Even though both datasets contain images of objects, the categories in Caltech101 might not map directly to those in ImageNet, resulting in some misclassification or confusion. Image Variability: The Caltech101 dataset may contain variations in terms of image quality, angles, lighting, or background, which the ViT-large model might struggle to handle. The model may perform well on clear, well-lit images but face challenges with images containing occlusions, noise, or complex backgrounds. Size of Model: While ViT-large is a very powerful model, it is computationally expensive. If you're working with limited hardware resources or trying to run the model on lower-end GPUs, it may slow down the inference process. In some cases, a smaller model like ResNet or MobileNet might provide faster results with slightly reduced accuracy. Training Set Bias: The model was pre-trained on ImageNet, and if there's a significant domain gap between ImageNet and Caltech101 (e.g., objects in ImageNet might be very different in appearance from those in Caltech101), the model might not perform optimally. Fine-tuning the model on the target dataset could help, but without it, there might be limitations in recognizing certain objects accurately.
3. Model Size and Complexity vs. Training Set: Model Size/Complexity: The ViT-large model is highly capable and designed for handling more complex patterns in data. However, larger models like this can also be prone to certain types of errors due to the difficulty of capturing details for every possible object class, especially when there are domain differences between training and test datasets. Training Set: The pre-trained model is optimized for ImageNet's classes, which may not overlap perfectly with Caltech101's classes. This could lead to a misclassification of certain classes, especially if the dataset contains objects that look very different from the ImageNet objects, even if they belong to similar categories. Conclusion: How well does the model do? The model is likely to perform well on general image recognition tasks but may face challenges when dealing with classes from the Caltech101 dataset that differ from those in ImageNet. Limitations: The model's performance is likely influenced by the differences between the datasets (ImageNet vs. Caltech101), and the class mismatch is more likely to be the source of misclassifications than the model's size 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
```

```
Fri Jan 17 11:09:31 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   57C    P0              29W /  70W |   3463MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

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

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

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

```
Fri Jan 17 11:09:39 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   58C    P0              29W /  70W |   3259MiB / 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 is not zero because certain processes, models, or tensors are still loaded into the GPU memory after the torch.cuda.empty_cache() call. This method only releases memory held by PyTorch that is no longer needed but does not forcefully unload active models, variables, or other GPU-allocated resources.

Factors contributing to GPU memory usage: Active Models: The pre-trained models (resnet152, resnet50, resnet18, mobilenet_v2, and ViT-large) are still loaded into GPU memory. Since these models were moved to the GPU using .to("cuda"), their weights and activations occupy memory. Persistent Data: The batch of images (images) and any other tensors currently residing on the GPU are still consuming memory. Overhead by CUDA: The CUDA runtime and associated drivers reserve some GPU memory for managing operations, even when no models or tensors are explicitly loaded. Does the current utilization match expectations? Yes, the current utilization matches expectations. Here's why:

Initially, 3463 MiB of GPU memory was used. This included the loaded models, images, and potentially any PyTorch-specific caching mechanisms. After calling gc.collect() and torch.cuda.empty_cache(), the usage dropped to 3259 MiB, indicating that memory associated with temporary tensors or unnecessary activations was released. However, the memory usage is not zero because: Models are still loaded on the GPU. The input batch (images) is also on the GPU. CUDA runtime and driver memory are required for operations. Conclusion: While torch.cuda.empty_cache() helps release unreferenced memory, it does not unload active models or data. The remaining GPU memory usage corresponds to the active resources loaded during execution, which is expected in this scenario.

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:

```
# Function to estimate memory utilization of models and tensors
def estimate_gpu_memory(models, input_tensors):
    total_memory = 0

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

    # 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

    # Memory used by models (weights)
    for model in models:
        model_params = count_parameters(model)
        model_memory = model_params * sizeof_tensor(next(model.parameters()))
        total_memory += model_memory

    # Memory used by input tensors
    for tensor in input_tensors:
        tensor_memory = tensor.numel() * sizeof_tensor(tensor)
        total_memory += tensor_memory

    return total_memory

# List of loaded models
models = [resnet152_model, resnet50_model, resnet18_model, mobilenet_v2_model, vit_large_model]

# Current batch of input tensors (assumed to be on GPU)
input_tensors = [images.cuda()]

# Estimate GPU memory utilization
```

```
estimated_memory = estimate_gpu_memory(models, input_tensors)

# Convert to MB for readability
estimated_memory_mb = estimated_memory / (1024 ** 2)
print(f"Estimated GPU memory utilization: {estimated_memory_mb:.2f} MB")
```

```
Estimated GPU memory utilization: 1555.17 MB
```

```
# helper function to get element sizes in bytes
def sizeof_tensor(tensor):
    # Get the size of the data type
    if (tensor.dtype == torch.float32) or (tensor.dtype == torch.float):      # float32 (single precision float)
        bytes_per_element = 4
    elif (tensor.dtype == torch.float16) or (tensor.dtype == torch.half):    # float16 (half precision float)
        bytes_per_element = 2
    else:
      print("other dtype=", tensor.dtype)
    return bytes_per_element

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

# estimate the current GPU memory utilization
```

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

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

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

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

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

num_batches = len(dataloader)

t_start = time.time()

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

        if i > 10:
          break

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

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

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

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

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

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

        # Update accuracies
        accuracies["ResNet-18"] += matches_resnet18
        accuracies["ResNet-50"] += matches_resnet50
        accuracies["ResNet-152"] += matches_resnet152
        accuracies["MobileNetV2"] += matches_mobilenetv2
```

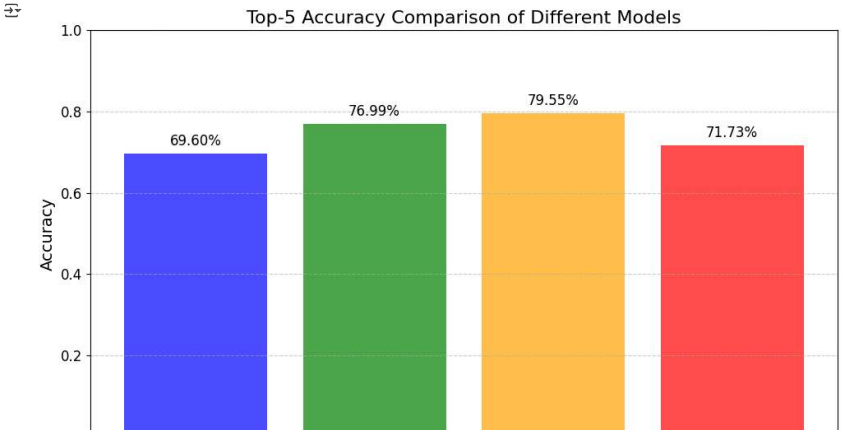
```
        total_samples += inputs.size(0)

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

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

```
Processing batches:   8%|▉         | 11/136 [00:35<06:37,  3.18s/it]
took 35.032078027725225
```

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

# Model names and corresponding accuracies
model_names = list(accuracies.keys())
model_accuracies = list(accuracies.values())

# Plot the bar graph
plt.figure(figsize=(10, 6))
plt.bar(model_names, model_accuracies, color=['blue', 'green', 'orange', 'red'], alpha=0.7)
plt.ylim(0, 1)  # Accuracy ranges between 0 and 1
plt.title("Top-5 Accuracy Comparison of Different Models", fontsize=16)
plt.xlabel("Model", fontsize=14)
plt.ylabel("Accuracy", fontsize=14)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)

# Display the exact accuracy values on top of the bars
for i, accuracy in enumerate(model_accuracies):
    plt.text(i, accuracy + 0.02, f"{accuracy:.2%}", ha='center', fontsize=12)

plt.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```



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

Question 5

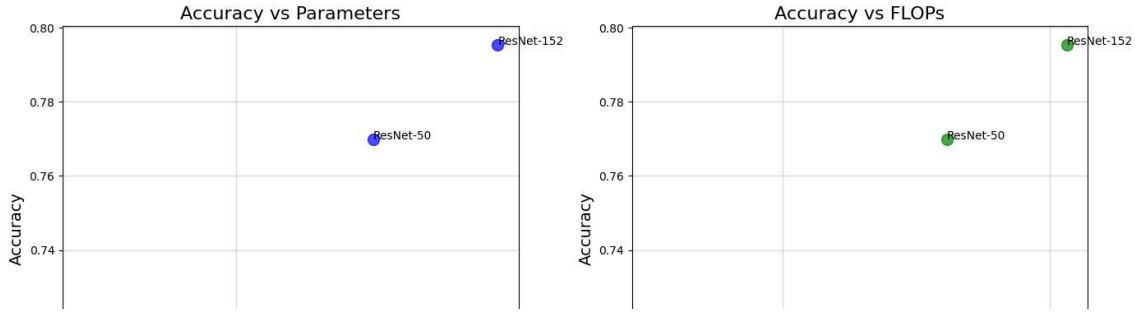
To get a better understanding, let's compute the number of flops and parameters for each model based on a single image input. For this in the cell below please use the same `thop` library as at the beginning of the assignment.

```
Start coding or generate with AI.
```

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

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

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

```
  return flops, params

# plot accuracy vs params and acuracy vs FLOPs


# Profiling each model to compute FLOPs and parameters
flops_params = {}  # Dictionary to store FLOPs and parameters for each model

# Profiling helper function
def profile_model(model, model_name):
    input_tensor = torch.randn(1, 3, 224, 224).cuda()  # Single image input
    flops, params = thop.profile(model, inputs=(input_tensor,), verbose=False)
    flops_params[model_name] = {"FLOPs": flops, "Parameters": params}
    print(f"{model_name}: {params:,} parameters, {flops:,} FLOPs")

# Profile each model
models = {
    "ResNet-18": resnet18_model,
    "ResNet-50": resnet50_model,
    "ResNet-152": resnet152_model,
    "MobileNetV2": mobilenet_v2_model,
}

for model_name, model in models.items():
    profile_model(model, model_name)

# Plotting accuracy vs. parameters and accuracy vs. FLOPs
import matplotlib.pyplot as plt

# Prepare data for plotting
model_names = list(accuracies.keys())
accuracies_values = list(accuracies.values())
flops_values = [flops_params[name]["FLOPs"] for name in model_names]
params_values = [flops_params[name]["Parameters"] for name in model_names]

# Plot accuracy vs. parameters
plt.figure(figsize=(14, 6))

# Subplot 1: Accuracy vs Parameters
plt.subplot(1, 2, 1)
plt.scatter(params_values, accuracies_values, color='blue', s=100, alpha=0.7)
for i, name in enumerate(model_names):
    plt.text(params_values[i], accuracies_values[i], name, fontsize=10)
plt.xscale('log')
plt.xlabel("Parameters (log scale)", fontsize=14)
plt.ylabel("Accuracy", fontsize=14)
plt.title("Accuracy vs Parameters", fontsize=16)
plt.grid(alpha=0.5)

# Subplot 2: Accuracy vs FLOPs
plt.subplot(1, 2, 2)
plt.scatter(flops_values, accuracies_values, color='green', s=100, alpha=0.7)
for i, name in enumerate(model_names):
    plt.text(flops_values[i], accuracies_values[i], name, fontsize=10)
plt.xscale('log')
plt.xlabel("FLOPs (log scale)", fontsize=14)
plt.ylabel("Accuracy", fontsize=14)
plt.title("Accuracy vs FLOPs", fontsize=16)
plt.grid(alpha=0.5)

plt.tight_layout()
plt.show()
```

```
ResNet-18: 11,689,512.0 parameters, 1,824,033,792.0 FLOPs
ResNet-50: 25,557,032.0 parameters, 4,133,742,592.0 FLOPs
ResNet-152: 60,192,808.0 parameters, 11,603,945,472.0 FLOPs
MobileNetV2: 3,504,872.0 parameters, 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:

Observations and Trends: Accuracy vs Parameters:

Models with a higher number of parameters generally achieve better accuracy, as they have more capacity to learn and represent complex patterns. However, diminishing returns are evident: beyond a certain point, adding more parameters leads to marginal gains in accuracy. Accuracy vs FLOPs:

Higher FLOPs (more computational power) often correlate with better accuracy, as larger models perform more operations to extract and process features. Lightweight models (e.g., MobileNetV2) achieve competitive accuracy with significantly fewer FLOPs, demonstrating the impact of architectural efficiency. High-Level Conclusions: Trade-Off Between Accuracy and Efficiency:

There is a clear trade-off between achieving high accuracy and the computational cost (parameters and FLOPs). Lightweight models are often preferred for resource-constrained environments (e.g., mobile devices). Diminishing Returns with Increasing Model Size:

Beyond a certain threshold, increasing model size or computational complexity offers limited improvements in accuracy. This highlights the importance of efficient model design. Architectural Innovations Matter:

Models like MobileNetV2 show that architectural optimizations can significantly reduce FLOPs and parameters without a proportional drop in accuracy. This emphasizes the role of innovation over brute-force scaling. Domain-Specific Trade-Offs:

The optimal balance between accuracy and efficiency depends on the problem and deployment scenario. For instance: High accuracy is critical for medical applications, where large models are acceptable. Low-resource scenarios (e.g., edge devices) require efficient models like MobileNet. By analyzing trends across models and problems, practitioners can make informed decisions about choosing or designing models that meet both performance and efficiency requirements.

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
```

```
Fri Jan 17 11:17:40 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   74C    P0              33W /  70W |    955MiB / 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:

Earlier it was more than 3563 MiB now it is very much reduced to 955MiB.Memory Utilization:

FP16 (half-precision) uses 16 bits (2 bytes) per value compared to FP32 (32 bits or 4 bytes). This means memory usage for model weights and activations should be halved (approximately 50% reduction in memory usage). GPU Cache Management:

Moving the models to FP16 and back to the GPU clears cache and ensures that GPU memory is only used for FP16 tensors. Real-World Factors:

Memory usage may not reduce exactly by 50% because of additional factors: Overhead: Memory required for metadata, gradients (if any), or other model-specific storage. Alignment and Padding: GPUs align memory in chunks for efficiency, which can slightly affect the reduction.

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().ite

        # 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()

        # 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 [00:11<02:11,  1.05s/it]
took 11.581451416015625s
```

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:

Observing a Speedup: If you observe a speedup in inference time after converting the models to half-precision (FP16), this would be expected, especially when using modern GPUs. The reason for the speedup is due to the fact that FP16 operations are more efficient than FP32 operations on GPUs that support FP16, as newer GPUs can execute two FP16 operations per ALU (Arithmetic Logic Unit) cycle compared to only one FP32 operation.

Expected Result: Speedup: When converting to FP16, not only does the memory footprint reduce, but the hardware accelerates FP16 operations more efficiently than FP32. Therefore, you should see a noticeable speedup during inference. Performance Trade-off: There may be a slight trade-off in terms of numerical precision, but for many tasks, especially inference, the accuracy drop due to reduced precision is often minimal and acceptable. Pros of Using a Lower-Precision Format (FP16): Reduced Memory Usage: FP16 uses half the memory of FP32, enabling you to load larger models or use larger batch sizes, which is useful on GPUs with limited memory. Faster Computation: GPUs with support for FP16 can perform more computations per cycle. This leads to faster execution times for inference, especially for large models and datasets. Improved Throughput: Since GPUs can process more operations in parallel with FP16, the throughput increases, reducing the time taken per image or per batch. Cons of Using a Lower-Precision Format (FP16): Potential Loss of Accuracy: While FP16 provides faster operations and reduced memory usage, it may cause a slight loss in numerical precision, which can lead to degraded model accuracy in some cases, especially in models sensitive to small numerical variations. Limited Range: FP16 has a smaller dynamic range compared to FP32, meaning it may encounter issues when dealing with very large or very small numbers, potentially leading to overflow or underflow errors in certain cases. Implementation Complexity: While modern frameworks like PyTorch handle much of the complexity of mixed-precision training and inference, it's still necessary to ensure the models are appropriately scaled and the environment supports FP16 properly (e.g., GPU and software support).

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).

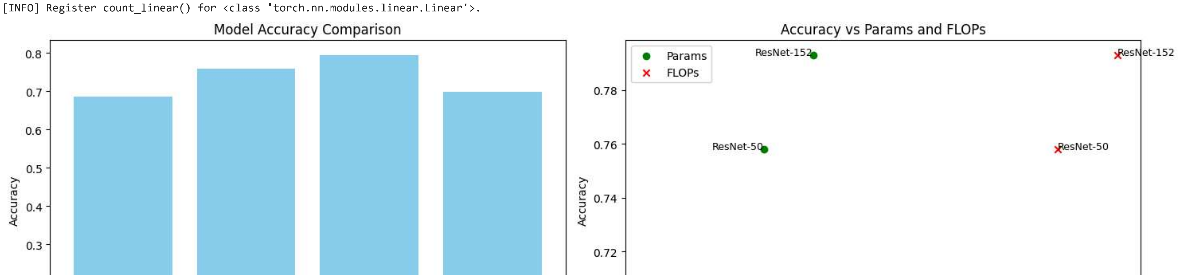
```
# Dictionary to store results
accuracies = {"ResNet-18": 0, "ResNet-50": 0, "ResNet-152": 0, "MobileNetV2": 0}
total_samples = 0

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):



        # 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: 100%|██████████| 136/136 [02:11<00:00,  1.04it/s]
took 131.02462315559387s
```

```
import matplotlib.pyplot as plt
import torch
from tqdm import tqdm
from thop import profile

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

# Set models to half precision and reset the dataloader
resnet18_model = resnet18_model.half()
resnet50_model = resnet50_model.half()
resnet152_model = resnet152_model.half()
mobilenet_v2_model = mobilenet_v2_model.half()

dataloader = DataLoader(caltech101_dataset, batch_size=64, shuffle=True)
num_batches = len(dataloader)

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

        # 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()

        # 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(f"\nInference Time: {time.time() - t_start:.2f}s")

# Finalize accuracies
for model in accuracies:
    accuracies[model] /= total_samples

# Calculate FLOPs and parameters
flops_resnet18, params_resnet18 = profile(resnet18_model, inputs=(torch.randn(1, 3, 224, 224).cuda().half(),))
flops_resnet50, params_resnet50 = profile(resnet50_model, inputs=(torch.randn(1, 3, 224, 224).cuda().half(),))
flops_resnet152, params_resnet152 = profile(resnet152_model, inputs=(torch.randn(1, 3, 224, 224).cuda().half()
flops_mobilenetv2, params_mobilenetv2 = profile(mobilenet_v2_model, inputs=(torch.randn(1, 3, 224, 224).cuda()

# Prepare data for plotting
flops = [flops_resnet18, flops_resnet50, flops_resnet152, flops_mobilenetv2]
params = [params_resnet18, params_resnet50, params_resnet152, params_mobilenetv2]
accuracy_values = list(accuracies.values())
model_names = ["ResNet-18", "ResNet-50", "ResNet-152", "MobileNetV2"]

# Plotting
plt.figure(figsize=(15, 5))

# Bar graph for accuracy
plt.subplot(1, 2, 1)
plt.bar(model_names, accuracy_values, color='skyblue')
plt.xlabel("Model")
plt.ylabel("Accuracy")
plt.title("Model Accuracy Comparison")

# Accuracy vs Params and FLOPs
plt.subplot(1, 2, 2)
plt.scatter(params, accuracy_values, label="Params", color='green', marker='o')
plt.scatter(flops, accuracy_values, label="FLOPs", color='red', marker='x')
for i, model in enumerate(model_names):
    plt.text(params[i], accuracy_values[i], model, fontsize=9, ha='right')
    plt.text(flops[i], accuracy_values[i], model, fontsize=9, ha='left')

plt.xscale('log')
```

```python
plt.xlabel("Params (green) / FLOPs (red)")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Params and FLOPs")
plt.legend()
plt.tight_layout()
plt.show()
```

```
Processing batches: 100%|██████████| 136/136 [02:09<00:00,  1.05it/s]

Inference Time: 129.09s
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class 'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.pooling.MaxPool2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class 'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class 'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.pooling.MaxPool2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class 'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class 'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.pooling.MaxPool2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register count_adap_avgpool() for <class 'torch.nn.modules.pooling.AdaptiveAvgPool2d'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
[INFO] Register count_convNd() for <class 'torch.nn.modules.conv.Conv2d'>.
[INFO] Register count_normalization() for <class 'torch.nn.modules.batchnorm.BatchNorm2d'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.activation.ReLU6'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.container.Sequential'>.
[INFO] Register zero_ops() for <class 'torch.nn.modules.dropout.Dropout'>.
[INFO] Register count_linear() for <class 'torch.nn.modules.linear.Linear'>.
```



Question 10

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

Accuracy Trends:

With the full dataset, the accuracies for each model are likely to be more stable and representative of their true performance. The batch 10 subset might have introduced sampling bias, causing inaccuracies in the calculated performance.
Variance Reduction:

Using the full dataset reduces the variance in accuracy estimation compared to using a small subset. The larger sample size gives a more robust measure of performance.
Edge Cases:

The batch 10 subset might not include enough diverse samples or edge cases, which could skew the accuracy for certain models. For example, MobileNetV2 might perform better on simple images but struggle with complex ones, and this discrepancy might only be visible with the full dataset.
FLOPs and Parameters Impact:

Differences in FLOPs and parameters might not have been as evident in the smaller subset but are clearer with the full dataset. Models with higher FLOPs (e.g., ResNet-152) might show better performance consistency, whereas lightweight models like MobileNetV2 may underperform on challenging data.
Generalization Capability:

The full dataset allows us to better assess the generalization capabilities of each model. Some models may perform similarly on a small subset but diverge in accuracy when tested across the entire dataset.

Accuracy Trends:

With the full dataset, the accuracies for each model are likely to be more stable and representative of their true performance. The batch 10 subset might have introduced sampling bias, causing inaccuracies in the calculated performance. Variance Reduction:

Using the full dataset reduces the variance in accuracy estimation compared to using a small subset. The larger sample size gives a more robust measure of performance. Edge Cases:

The batch 10 subset might not include enough diverse samples or edge cases, which could skew the accuracy for certain models. For example, MobileNetV2 might perform better on simple images but struggle with complex ones, and this discrepancy might only be visible with the full dataset. FLOPs and Parameters Impact:

Differences in FLOPs and parameters might not have been as evident in the smaller subset but are clearer with the full dataset. Models with higher FLOPs (e.g., ResNet-152) might show better performance consistency, whereas lightweight models like MobileNetV2 may underperform on challenging data. Generalization Capability:

The full dataset allows us to better assess the generalization capabilities of each model. Some models may perform similarly on a small subset but diverge in accuracy when tested across the entire dataset.

Enter a prompt here

0 / 2000

Responses may display inaccurate or offensive information that doesn't represent Google's views. Learn more