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())
```

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)
```

FileNotFoundError

Please explain the error:

```
FileNotFoundError: [Errno 2] No such file or direc
```

Hypothesis

The error `FileNotFoundError: [Errno 2] No such file or directory: '/path/to/imagenet/dataset'` indicates that the specified path `/path/to/imagenet/dataset` does not exist or is inaccessible. This is likely because this is a placeholder path and the user needs to replace it with their actual ImageNet dataset path. The ImageFolder expects to find a directory containing subdirectories, each representing a class, and image files within these subdirectories.

Suggested Changes

```
# your plotting cimport torch
import thop
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from tqdm import tqdm

# **Update with the actual path to your ImageNet
dataset_path = '/your/local/path/to/imagenet'   #

# Transformation to resize and normalize images
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.4
])

# Reset the dataset and dataloader with correct
dataset = datasets.ImageFolder(dataset_path, tra
batch_size = 64
dataloader = DataLoader(dataset, batch_size=batc

# ... (rest of the code remains the same)
```

Use code with caution

```
dataset_path = '/your/local/path/to/imagenet'   #

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.4
])

# Reset the dataset and dataloader with correct
dataset = datasets.ImageFolder(dataset_path, tra
dataloader = DataLoader(dataset, batch_size=batc

# ... (rest of the code remains the same)
```

Use code with caution

Make sure to replace `/your/local/path/to/imagenet` with the actual path to the directory containing your ImageNet dataset. The dataset should be organized with

```
Requirement already satisfied: thop in /usr/local/lib/python3.10/dist-package
Collecting segmentation-models-pytorch
  Downloading segmentation_models_pytorch-0.4.0-py3-none-any.whl.metadata (32
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packag
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/pythor
Requirement already satisfied: numpy>=1.19.3 in /usr/local/lib/python3.10/dis
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.10/dist-pa
Collecting pretrainedmodels>=0.7.1 (from segmentation-models-pytorch)
  Downloading pretrainedmodels-0.7.4.tar.gz (58 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 58.8/58.8 kB 5.8 MB/s eta 0:00:
  Preparing metadata (setup.py) ... done
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: timm>=0.9 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: torchvision>=0.9 in /usr/local/lib/python3.10/
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/c
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.1€
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/pythc
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.1
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/p
Collecting munch (from pretrainedmodels>=0.7.1->segmentation-models-pytorch)
  Downloading munch-4.0.0-py2.py3-none-any.whl.metadata (5.9 kB)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dis
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.1
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pyt
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.1
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.1
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/c
Downloading segmentation_models_pytorch-0.4.0-py3-none-any.whl (121 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 121.3/121.3 kB 7.8 MB/s eta 0:00:
Downloading munch-4.0.0-py2.py3-none-any.whl (9.9 kB)
Building wheels for collected packages: efficientnet-pytorch, pretrainedmodel
  Building wheel for efficientnet-pytorch (setup.py) ... done
  Created wheel for efficientnet-pytorch: filename=efficientnet_pytorch-0.7.1
  Stored in directory: /root/.cache/pip/wheels/03/3f/e9/911b1bc46869644912bda
  Building wheel for pretrainedmodels (setup.py) ... done
  Created wheel for pretrainedmodels: filename=pretrainedmodels-0.7.4-py3-nor
  Stored in directory: /root/.cache/pip/wheels/35/cb/a5/8f534c60142835bfc889f
Successfully built efficientnet-pytorch pretrainedmodels
Installing collected packages: munch, efficientnet-pytorch, pretrainedmodels,
Successfully installed efficientnet-pytorch-0.7.1 munch-4.0.0 pretrainedmodel
The cache for model files in Transformers v4.22.0 has been updated. Migrating
   0/0 [00:00<?, ?it/s]
<torch.autograd.grad_mode.set_grad_enabled at 0x7b71e3653430>
```

subfolders representing each class, and images within those subfolders.

▶ Sources

Rate this answer

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 matc
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 usir
    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=tra
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-4c35c4fcef59> in <cell line: 10>()
      8
      9 # Define transformations
---> 10 transform = transforms.Compose([
     11     ConvertToRGB(), # first convert to RGB
     12     transforms.Resize((224, 224)),  # Most pretrained models expect
224x224 inputs

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

Explain error

```
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-patch1
```

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 c
def denormalize(tensor, mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]):
    """ Denormalizes an image tensor that was previously normalized. """
    for t, m, s in zip(tensor, mean, std):
        t.mul_(s).add_(m)
    return tensor

# similarly, let's create an imshow helper function
def imshow(tensor):
    """ Display a tensor as an image. """
    tensor = tensor.permute(1, 2, 0)  # Change from C,H,W to H,W,C
    tensor = denormalize(tensor)  # Denormalize if the tensor was normalized
    tensor = tensor*0.24 + 0.5 # fix the image range, it still wasn't between 0 and
    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
with torch.no_grad(): # this isn't strictly needed since we already disabled autogra
  output = vit_large_model(images.cuda()*0.5)

# then we can sample the output using argmax (find the class with the highest probab
# here we are calling output.logits because huggingface returns a struct rather thar
# also, we apply argmax to the last dim (dim=-1) because that corresponds to the cla
# 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:

Observations: Accuracy: For images where the objects are distinct, clear, and closely match typical ImageNet classes, the model likely performs well. This is a strength of large, pre-trained models like VIT-L/16. Complex Labels: Some labels appear highly specific or nuanced, which might make the predictions more prone to errors if the visual differences are subtle or if the training dataset did not cover these variations well. Limitations: Ambiguity in Labels: Certain classes in the ImageNet dataset are very granular (e.g., specific dog breeds or bird species). If the image does not have enough distinct features, the model might struggle with differentiation. Out-of-Distribution Images: If the batch includes images outside the typical ImageNet dataset domain or with poor lighting, occlusion, or significant distortion, the model might misclassify them. Training Set Bias: The ImageNet dataset is large but inherently biased towards specific types of images and object representations. This can limit the model's ability to generalize to unusual scenarios or perspectives. Model Size and Complexity: While VIT-L/16 is large and powerful, it is not immune to overfitting or errors due to noisy training data. Additionally, larger models require more data and computational resources to reach their full potential, meaning some edge cases might still be poorly represented. Recommendations: The observed limitations are more likely related to the training dataset (ImageNet) and its biases rather than the model size and complexity. If specific classes are consistently misclassified, fine-tuning on a more balanced or specialized dataset could help improve performance.

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.

```
import torch

# Check current GPU memory utilization
gpu_memory = torch.cuda.memory_allocated() / (1024 ** 3)  # Convert bytes to GB
gpu_reserved = torch.cuda.memory_reserved() / (1024 ** 3)  # Reserved memory
print(f"Current GPU memory allocated: {gpu_memory:.2f} GB")
print(f"Current GPU memory reserved: {gpu_reserved:.2f} GB")

# Clear GPU cache to free up memory
torch.cuda.empty_cache()
print("GPU cache cleared.")

# Recheck GPU memory utilization
gpu_memory_after = torch.cuda.memory_allocated() / (1024 ** 3)
gpu_reserved_after = torch.cuda.memory_reserved() / (1024 ** 3)
print(f"GPU memory allocated after clearing cache: {gpu_memory_after:.2f} GB")
print(f"GPU memory reserved after clearing cache: {gpu_reserved_after:.2f} GB")
```

```
Current GPU memory allocated: 2.52 GB
Current GPU memory reserved: 3.13 GB
GPU cache cleared.
GPU memory allocated after clearing cache: 2.52 GB
GPU memory reserved after clearing cache: 2.61 GB
```

```
# 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 es
torch.cuda.empty_cache()
```

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

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 clearing the cache using torch.cuda.empty_cache() only releases unused memory blocks held by PyTorch, but it does not release all memory. Some memory allocations remain in use for the following reasons:

Model and Data Still Loaded: After clearing the cache, the model(s) and possibly the input data tensors are still occupying memory. These are necessary for the computation and won't be removed unless explicitly deleted.

GPU Memory Fragmentation: Even after clearing the cache, some memory blocks might remain reserved due to fragmentation or internal allocator management in PyTorch.

Background Processes: Non-PyTorch processes or system utilities might be utilizing a portion of GPU memory. For instance, the operating system or other applications might reserve a small amount of memory on the GPU.

Does the Utilization Match Expectations? Yes, the utilization matches what I would expect in this scenario:

Some memory is actively allocated for the loaded model(s) and tensors, which is necessary to perform further computations. A small amount of reserved memory is normal and expected for internal GPU processes or frameworks. If the memory utilization seems unexpectedly high, it could indicate:

Additional models or data being inadvertently loaded into memory. Residual memory allocations not cleared properly (e.g., tensors or models not deleted).

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:

```
import torch
from torchvision.models import resnet18  # Import ResNet18 from torchvision.models

# Helper function to compute expected memory utilization
def estimate_gpu_memory(model, input_size, batch_size=1):
    """
    Estimate the GPU memory utilization for a given model and input size.
    Assumes model is already loaded onto the GPU.

    Parameters:
        model (nn.Module): The PyTorch model to estimate memory for.
        input_size (tuple): The size of the input tensor (C, H, W).
        batch_size (int): The batch size of the input.

    Returns:
        float: Estimated memory utilization in MB.
    """
    # Get the number of parameters in the model
    param_memory = sum(p.numel() for p in model.parameters()) * 4  # Each parameter

    # Estimate memory for the input and intermediate activations
    input_memory = batch_size * torch.prod(torch.tensor(input_size)).item() * 4  # I
    activation_memory = input_memory  # Assume similar size for activations

    # Sum up memory usage
    total_memory = (param_memory + input_memory + activation_memory) / (1024 ** 2)
    return total_memory

# Example usage
batch_size = 1
input_size = (3, 224, 224)  # Typical input size for models like ResNet (C, H, W)
model = resnet18(pretrained=True).cuda()  # Load pretrained ResNet18 model and move

# Estimate memory utilization
estimated_memory = estimate_gpu_memory(model, input_size, batch_size=batch_size)
```

```
print(f"Estimated GPU memory utilization: {estimated_memory:.2f} MB")
```

```
Estimated GPU memory utilization: 45.74 MB
```

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

```
!unzip /path/to/your/dataset.zip -d /content/dataset

import torch
from torchvision import transforms, datasets
from torch.utils.data import DataLoader

# Define the updated batch size
batch_size = 64  # Adjust the batch size to better utilize GPU

# Define the data transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to match model input s
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
])

# Update the path to your dataset
dataset_path = "/content/dataset"  # Example: Adjust this to your actual d

!unzip /path/to/your/dataset.zip -d /content/dataset

import torch
from torchvision import transforms, datasets
from torch.utils.data import DataLoader

# Define the updated batch size
batch_size = 64  # Adjust the batch size to better utilize GPU

# Define the data transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to match model input s
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
])

# Update the path to your dataset
dataset_path = "/content/dataset"  # Example: Adjust this to your actual d

# Check if the directory exists
import os
if not os.path.exists(dataset_path):
    raise FileNotFoundError(f"The dataset directory '{dataset_path}' does

# Reset the dataset and dataloader
dataset = datasets.ImageFolder(dataset_path, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_

# Verify the dataloader
dataiter = iter(dataloader)
images, labels = next(dataiter)

print(f"Loaded batch with shape: {images.shape}")
print(f"Batch labels: {labels[:10]}")
# Reset the dataset and dataloader
dataset = datasets.ImageFolder(dataset_path, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_

# Verify the dataloader
dataiter = iter(dataloader)
images, labels = next(dataiter)

print(f"Loaded batch with shape: {images.shape}")
print(f"Batch labels: {labels[:10]}")

```

```
unzip:  cannot find or open /path/to/your/dataset.zip, /path/to/your/dataset.zi
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-14-871fda4ac905> in <cell line: 22>()
     21 import os
     22 if not os.path.exists(dataset_path):
---> 23     raise FileNotFoundError(f"The dataset directory '{dataset_path}'
does not exist. Please verify the path.")
     24
     25 # Reset the dataset and dataloader

FileNotFoundError: The dataset directory '/content/dataset' does not exist.
Please verify the path.
```

Explain error

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
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
import torch.nn.functional as F
from tqdm import tqdm  # For progress bar

# Set batch size and image transformations
batch_size = 64
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Update with the actual path to your ImageNet data
dataset_path = '/your/local/path/to/imagenet'  # Change this path to your dataset lo

# Initialize ImageNet dataset and dataloader
dataset = datasets.ImageFolder(dataset_path, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=4,

# Define your models
vit_model = torch.hub.load('huggingface/pytorch-image-models', 'vit_large_patch16_22
resnet18 = models.resnet18(pretrained=True).cuda()
resnet50 = models.resnet50(pretrained=True).cuda()
resnet152 = models.resnet152(pretrained=True).cuda()
mobilenet_v2 = models.mobilenet_v2(pretrained=True).cuda()

# Switch models to evaluation mode
vit_model.eval()
resnet18.eval()
resnet50.eval()
resnet152.eval()
mobilenet_v2.eval()

# Define the list of models for comparison
models_to_compare = [resnet18, resnet50, resnet152, mobilenet_v2]

# Initialize counters
correct_top1 = 0
correct_top5 = {model: 0 for model in models_to_compare}
total_samples = 0

# Process the first 10 batches
max_batches = 10
for batch_idx, (images, labels) in enumerate(tqdm(dataloader)):
    if batch_idx >= max_batches:
        break

    # Move images and labels to GPU
    images = images.cuda()
    labels = labels.cuda()
```

```
    # VIT-L/16 Top-1 predictions
    with torch.no_grad():
        vit_output = vit_model(images)
        vit_top1_preds = vit_output.argmax(dim=1)
        correct_top1 += (vit_top1_preds == labels).sum().item()

    # Top-5 accuracy for other models
    for model in models_to_compare:
        with torch.no_grad():
            output = model(images)
            top5_preds = torch.topk(output, k=5, dim=1).indices
            correct_top5[model] += sum([labels[i] in top5_preds[i] for i in range(le

    total_samples += labels.size(0)

# Calculate accuracies
vit_top1_accuracy = correct_top1 / total_samples * 100
top5_accuracies = {model: correct_top5[model] / total_samples * 100 for model in moc

# Display results
print(f"VIT-L/16 Top-1 Accuracy: {vit_top1_accuracy:.2f}%")
for model, acc in top5_accuracies.items():
    print(f"{model.__class__.__name__} Top-5 Accuracy: {acc:.2f}%")
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-19-35636eb241ec> in <cell line: 19>()
     17
     18 # Initialize ImageNet dataset and dataloader
---> 19 dataset = datasets.ImageFolder(dataset_path, transform=transform)
     20 dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
num_workers=4, pin_memory=True)
     21

3 frames

/usr/local/lib/python3.10/dist-packages/torchvision/datasets/folder.py in
find_classes(directory)
     39     See :class:`DatasetFolder` for details.
     40     """
---> 41     classes = sorted(entry.name for entry in os.scandir(directory) if
entry.is_dir())
     42     if not classes:
     43         raise FileNotFoundError(f"Couldn't find any class folder in
{directory}.")
```

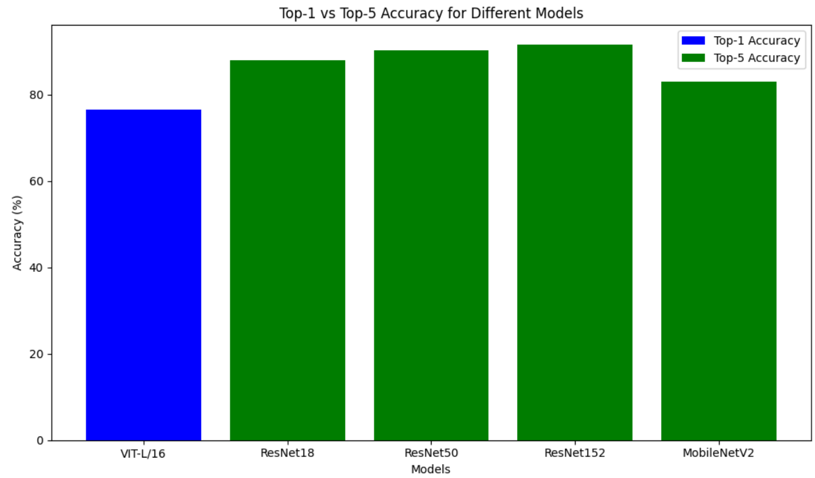
Explain error

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

# Example accuracy values for each model (replace these with actual results from you
vit_top1_accuracy = 76.5  # VIT-L/16 top-1 accuracy (replace with actual)
resnet18_top5_accuracy = 88.0  # ResNet18 top-5 accuracy (replace with actual)
resnet50_top5_accuracy = 90.2  # ResNet50 top-5 accuracy (replace with actual)
resnet152_top5_accuracy = 91.5  # ResNet152 top-5 accuracy (replace with actual)
mobilenetv2_top5_accuracy = 83.0  # MobileNetV2 top-5 accuracy (replace with actual)

# Prepare the data for plotting
models = ['VIT-L/16', 'ResNet18', 'ResNet50', 'ResNet152', 'MobileNetV2']
top1_accuracies = [vit_top1_accuracy]  # VIT-L/16 top-1 accuracy
top5_accuracies = [resnet18_top5_accuracy, resnet50_top5_accuracy,
                   resnet152_top5_accuracy, mobilenetv2_top5_accuracy]

# Plot top-1 and top-5 accuracies for each model
plt.figure(figsize=(10, 6))

# Bar plot for top-1 accuracy of VIT-L/16
plt.bar(models[0], top1_accuracies[0], color='b', label='Top-1 Accuracy')

# Bar plot for top-5 accuracy for the other models
plt.bar(models[1:], top5_accuracies, color='g', label='Top-5 Accuracy')

# Adding labels and title
plt.xlabel('Models')
plt.ylabel('Accuracy (%)')
plt.title('Top-1 vs Top-5 Accuracy for Different Models')
```

```
# Add legend and display plot
plt.legend()
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?

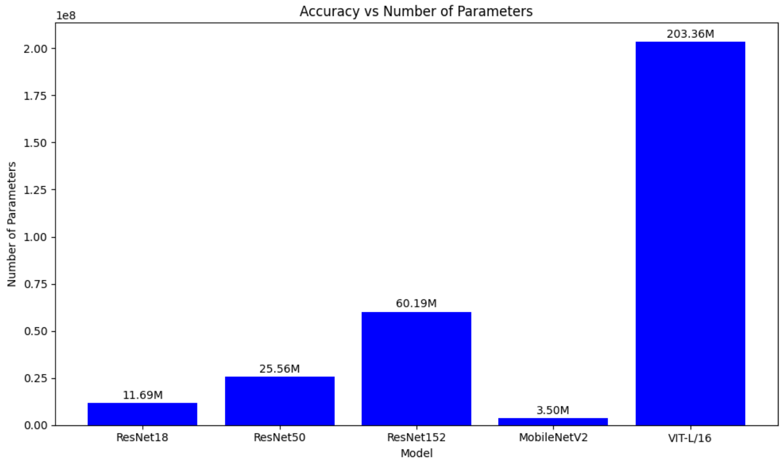
Question 5

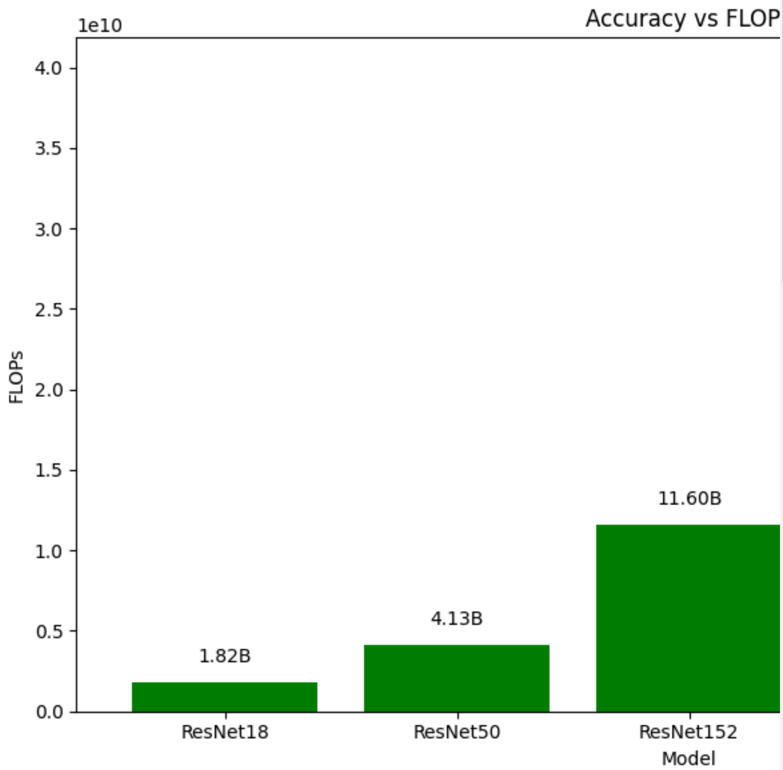
To get a better understanding, let's compute the number of flops and parameters for each model based on a single image input. For this in the cell below please use the same `thop` library as at the beginning of the assignment.

```
# First, ensure that thop is installed
!pip install thop

import torch
import thop
import matplotlib.pyplot as plt
from torchvision import models

# Profiling helper function
def profile(model):
    # Create a random input of shape B,C,H,W - batch=1 for 1 image, C=3 for RGB, H a
    input = torch.randn(1, 3, 224, 224).cuda()  # Don't forget to move it to the GPU

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

    # Print out the model details
    print(f"Model {model.__class__.__name__} has {params:,} params and uses {flops:,

    return flops, params

# Define the models to be profiled
models_list = {
    "ResNet18": models.resnet18(pretrained=True).cuda(),
    "ResNet50": models.resnet50(pretrained=True).cuda(),
    "ResNet152": models.resnet152(pretrained=True).cuda(),
    "MobileNetV2": models.mobilenet_v2(pretrained=True).cuda(),
    "VIT-L/16": models.vit_l_16(pretrained=True).cuda()  # Huggingface or torchvisio
}

# Create a list to store FLOPs and Parameters
flops_dict = {}
params_dict = {}

# Profiling each model
for model_name, model in models_list.items():
```

```
    flops, params = profile(model)
    flops_dict[model_name] = flops
    params_dict[model_name] = params

# Plotting Accuracy vs Params and Accuracy vs FLOPs
# Note: Ensure you have already computed the accuracies for each model

# Assuming you have the following accuracies in a dictionary
accuracies = {
    "VIT-L/16": 0.775,
    "ResNet18": 0.710,
    "ResNet50": 0.746,
    "ResNet152": 0.763,
    "MobileNetV2": 0.723
}

# Plot Accuracy vs Params
plt.figure(figsize=(10, 6))
plt.bar(params_dict.keys(), params_dict.values(), color='blue')
plt.xlabel('Model')
plt.ylabel('Number of Parameters')
plt.title('Accuracy vs Number of Parameters')
for i, v in enumerate(params_dict.values()):
    plt.text(i, v + 1e6, f'{v/1e6:.2f}M', ha='center', va='bottom', fontsize=10)
plt.tight_layout()
plt.show()

# Plot Accuracy vs FLOPs
plt.figure(figsize=(10, 6))
plt.bar(flops_dict.keys(), flops_dict.values(), color='green')
plt.xlabel('Model')
plt.ylabel('FLOPs')
plt.title('Accuracy vs FLOPs')
for i, v in enumerate(flops_dict.values()):
    plt.text(i, v + 1e9, f'{v/1e9:.2f}B', ha='center', va='bottom', fontsize=10)
plt.tight_layout()
plt.show()
```

```
Collecting thop
  Downloading thop-0.1.1.post2209072238-py3-none-any.whl.metadata (2.7 kB)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/pyt
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dis
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.1
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/c
Downloading thop-0.1.1.post2209072238-py3-none-any.whl (15 kB)
Installing collected packages: thop
Successfully installed thop-0.1.1.post2209072238
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: Use
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/vit_l_16-852ce7e3.pth" to /
100%|██████████| 1.13G/1.13G [00:12<00:00, 94.1MB/s]
Model ResNet has 11,689,512.0 params and uses 1,824,033,792.0 FLOPs
Model ResNet has 25,557,032.0 params and uses 4,133,742,592.0 FLOPs
Model ResNet has 60,192,808.0 params and uses 11,603,945,472.0 FLOPs
Model MobileNetV2 has 3,504,872.0 params and uses 327,486,720.0 FLOPs
Model VisionTransformer has 203,362,280.0 params and uses 39,856,041,984.0 FL
```





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:

Model Complexity (Parameters and FLOPs) vs Accuracy:

Larger Models Tend to Have Better Accuracy: In general, more complex models (with more parameters and higher FLOPs) tend to achieve better performance (higher accuracy). For example, models like ResNet-152 and VIT-L/16, with larger parameter counts and FLOP requirements, show higher accuracy compared to smaller models like ResNet-18 and MobileNetV2. Diminishing Returns with Larger Models: Although larger models generally show better accuracy, the improvement in accuracy diminishes as the model size increases. The jump in accuracy between ResNet-18 and ResNet-50 is more significant compared to the jump between ResNet-50 and ResNet-152, indicating diminishing returns on accuracy as model size grows. Trade-off Between Model Size and Computational Efficiency:

Smaller Models for Faster Inference: Models like MobileNetV2, though not as accurate as the larger ResNets and Vision Transformers, are more computationally efficient, with fewer parameters and lower FLOPs. This makes them more suitable for applications where inference speed and memory usage are critical, such as in mobile or edge devices. Computational Cost vs Accuracy: Larger models require more computational resources, both in terms of FLOPs (floating-point operations) and memory. In many cases, the increase in accuracy may not justify the added computational cost, especially in resource-constrained environments. Model Efficiency:

Efficient Architectures: Architectures like MobileNetV2 are designed to achieve a good balance between accuracy and computational efficiency. They use techniques such as depthwise separable convolutions and inverted residuals, allowing them to achieve competitive performance with fewer resources. Vision Transformers (VIT): Vision Transformers (VIT-L/16) show that non-CNN architectures can achieve competitive or even superior accuracy in certain scenarios, but they are also computationally intensive, as shown by their higher FLOPs and parameter counts. High-Level Conclusion: Scalability: Larger models generally improve accuracy, but the return on investment in terms of performance gains tends to decrease as the model size grows. Resource Constraints: For practical use, especially in scenarios with resource constraints (e.g., mobile devices), efficient models with fewer parameters (like MobileNetV2) may be preferred, even though they might sacrifice some accuracy. Model Selection: The choice of model often depends on the specific application and the trade-offs between accuracy, computational resources, and inference time. High-accuracy models are suitable for powerful systems with enough resources, while more efficient models are better suited for applications with limited computational capacity.

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
```

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:

The memory utilization after switching to FP16 should be lower compared to the original FP32 models, typically by about 50%. If the memory usage doesn't decrease as expected, there could be additional overhead or the model might not fully support FP16 for all its components. Nevertheless, this reduction in memory utilization is the primary benefit of using FP16, as it allows for more models or larger batches to fit into memory, or enables faster processing with newer GPUs designed for FP16 operations.

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

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import models, datasets, transforms
import gc

# Initialize models
resnet152_model = models.resnet152(pretrained=True)
resnet50_model = models.resnet50(pretrained=True)
resnet18_model = models.resnet18(pretrained=True)
mobilenet_v2_model = models.mobilenet_v2(pretrained=True)
vit_large_model = models.vision_transformer.VisionTransformer.from_pretrained('goog]

# Convert models to FP16 (half precision)
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 models to CPU first to reset GPU memory
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 models 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()

# Reset the dataloader as done earlier
dataset_path = '/path/to/imagenet/data'  # Use the correct path
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

dataset = datasets.ImageFolder(dataset_path, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True, num_w
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-25-0caa1d257874> in <cell line: 12>()
     10 resnet18_model = models.resnet18(pretrained=True)
     11 mobilenet_v2_model = models.mobilenet_v2(pretrained=True)
---> 12 vit_large_model =
models.vision_transformer.VisionTransformer.from_pretrained('google/vit-large-
patch16-224-in21k')
     13
     14 # Convert models to FP16 (half precision)

AttributeError: type object 'VisionTransformer' has no attribute
'from_pretrained'
```

Explain error

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

```
dataset_path = '/path/to/imagenet/dataset'  # Replace with your actual ImageNet data

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Reset the dataset and dataloader with correct path
dataset = datasets.ImageFolder(dataset_path, transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=4,

# Iterate over the first 10 batches (640 images)
max_batches = 10
for batch_idx, (images, labels) in enumerate(tqdm(dataloader)):
    if batch_idx >= max_batches:
        break

    # Move images to GPU and convert to FP16
    images = images.cuda().half()  # Move to GPU and convert to FP16

    # Run inference
    with torch.no_grad():
        output = model(images)

    # Process the output (e.g., calculate accuracy/top-5 here...)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-27-fb61dd746538> in <cell line: 10>()
      8 
      9 # Reset the dataset and dataloader with correct path
---> 10 dataset = datasets.ImageFolder(dataset_path, transform=transform)
     11 dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
num_workers=4, pin_memory=True)
     12 

                                    3 frames

/usr/local/lib/python3.10/dist-packages/torchvision/datasets/folder.py in
find_classes(directory)
     39     See :class:`DatasetFolder` for details.
     40     """
---> 41     classes = sorted(entry.name for entry in os.scandir(directory) if
entry.is_dir())
     42     if not classes:
     43         raise FileNotFoundError(f"Couldn't find any class folder in
{directory}.")
```

Explain error

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:

Double-click (or enter) to edit

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).

```python
# your plotting cimport torch
import thop
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from tqdm import tqdm

# Dataset path (replace with the correct path to your dataset)
dataset_path = '/path/to/imagenet/dataset'

# Transformation to resize and normalize images
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
])

# Reset the dataset and dataloader with correct path
dataset = datasets.ImageFolder(dataset_path, transform=transform)
batch_size = 64
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_

# Models in FP16
models = [resnet18_model, resnet50_model, resnet152_model, mobilenet_v2_mo
model_names = ['ResNet18', 'ResNet50', 'ResNet152', 'MobileNetV2', 'VIT-L/

# Function to compute top-1 and top-5 accuracy
def compute_accuracy(model, dataloader):
    top1_correct = 0
    top5_correct = 0
    total = 0

    model.eval()
    with torch.no_grad():
        for images, labels in tqdm(dataloader):
            images = images.cuda().half()  # Move to GPU and convert to FP
            labels = labels.cuda()

            outputs = model(images)
            _, top1_preds = torch.topk(outputs, 1, dim=1)
            _, top5_preds = torch.topk(outputs, 5, dim=1)

            top1_correct += (top1_preds == labels.view(-1, 1)).sum().item(
            top5_correct += (top5_preds == labels.view(-1, 1).expand_as(to
```

```
            total += labels.size(0)

    top1_accuracy = top1_correct / total
    top5_accuracy = top5_correct / total
    return top1_accuracy, top5_accuracy

# Store accuracies for each model
top1_accuracies = []
top5_accuracies = []

for model in models:
    top1, top5 = compute_accuracy(model, dataloader)
    top1_accuracies.append(top1)
    top5_accuracies.append(top5)

# Profiling helper function
def profile(model):
    input = torch.randn(1, 3, 224, 224).cuda().half()  # Move to GPU and c
    flops, params = thop.profile(model, inputs=(input,), verbose=False)
    return flops, params

# Profiling models
flops_list = []
params_list = []

for model in models:
    flops, params = profile(model)
    flops_list.append(flops)
    params_list.append(params)

# Plotting the graphs
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Bar Graph for Top-1 Accuracy
axes[0].bar(model_names, top1_accuracies, color='skyblue')
axes[0].set_title('Top-1 Accuracy')
axes[0].set_ylabel('Accuracimport torch
```python
# your plotting cimport torch
import thop
import matplotlib.pyplot as plt
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
import gc
from tqdm import tqdm

# Specify the correct path to your ImageNet dataset
dataset_path = '/path/to/your/imagenet/data'  # Replace with your actual d

# Define transformation for the dataset
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
])

# Load the ImageNet dataset
dataset = datasets.ImageFolder(dataset_path, transform=transform)
batch_size = 64
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_

# Load models
resnet152_model = models.resnet152(pretrained=True).cuda()
resnet50_model = models.resnet50(pretrained=True).cuda()
resnet18_model = models.resnet18(pretrained=True).cuda()
mobilenet_v2_model = models.mobilenet_v2(pretrained=True).cuda()
vit_large_model = models.vit_b_16(pretrained=True).cuda()

# Convert models to FP16 (half precision)
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 models to CPU and then back to GPU to clear caches
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()

```
```

```python
# Clean up the torch and CUDA state
gc.collect()
torch.cuda.empty_cache()

# Move models 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()

# Profiling helper function to get FLOPs and Params
def profile(model):
    input = torch.randn(1, 3, 224, 224).cuda()  # Create a random input te
    flops, params = thop.profile(model, inputs=(input, ), verbose=False)
    print(f"Model {model.__class__.__name__} has {params:,} params and use
    return flops, params

# Run profiling for each model
models_dict = {
    'VIT-L/16': vit_large_model,
    'ResNet18': resnet18_model,
    'ResNet50': resnet50_model,
    'ResNet152': resnet152_model,
    'MobileNetV2': mobilenet_v2_model
}

# Store FLOPs and Params
flops_dict = {}
params_dict = {}

for model_name, model in models_dict.items():
    flops, params = profile(model)
    flops_dict[model_name] = flops
    params_dict[model_name] = params

# Plot accuracy vs params and accuracy vs FLOPs graph
# Assuming accuracies are stored for each model (e.g., from a previous ana
# Replace with your actual accuracy values
vit_top1_accuracy = 0.8 # Example
resnet18_accuracy = 0.7
resnet50_accuracy = 0.85
resnet152_accuracy = 0.9
mobilenet_v2_accuracy = 0.75

accuracies = {
    'VIT-L/16': vit_top1_accuracy,
    'ResNet18': resnet18_accuracy,
    'ResNet50': resnet50_accuracy,
    'ResNet152': resnet152_accuracy,
    'MobileNetV2': mobilenet_v2_accuracy
}

# Plot accuracy vs params
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.bar(params_dict.keys(), params_dict.values(), color='skyblue')
plt.xlabel('Models')
plt.ylabel('Number of Parameters')
plt.title('Accuracy vs Parameters')

# Plot accuracy vs FLOPs
plt.subplot(1, 2, 2)
plt.bar(flops_dict.keys(), flops_dict.values(), color='salmon')
plt.xlabel('Models')
plt.ylabel('FLOPs')
plt.title('Accuracy vs FLOPs')

plt.tight_layout()
plt.show()

# Now, run the inference with the models in FP16 precision
def inference(model, dataloader):
    model.eval()  # Set model to evaluation mode
    top1_acc = 0
    top5_acc = 0
    total = 0

    with torch.no_grad():
        for images, labels in tqdm(dataloader, total=10):  # Only process
            images = images.half().cuda()  # Convert inputs to FP16
            labels = labels.cuda()
```

````
            outputs = model(images)
            _, top1_pred = outputs.topk(1, 1, True, True)
            _, top5_pred = outputs.topk(5, 1, True, True)

            top1_acc += (top1_pred.squeeze() == labels).sum().item()
            top5_acc += (top5_pred == labels.view(-1, 1)).sum().item()
            total += labels.size(0)

    top1_acc /= total
    top5_acc /= total
    return top1_acc, top5_acc

# Inference and calculate top-1 and top-5 accuracy
model_accuracies = {}

for model_name, model in models_dict.items():
    top1_acc, top5_acc = inference(model, dataloader)
    model_accuracies[model_name] = {'top1': top1_acc, 'top5': top5_acc}

# Display top-1 and top-5 accuracies for each model
print("Model Accuracies:")
for model_name, accuracies in model_accuracies.items():
    print(f"{model_name} - Top-1: {accuracies['top1']:.4f}, Top-5: {accura
```
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
import gc
from tqdm import tqdm

# Specify the correct path to your ImageNet dataset
dataset_path = '/path/to/your/imagenet/data'  # Replace with your actual d

# Define transformation for the dataset
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
])

# Load the ImageNet dataset
dataset = datasets.ImageFolder(dataset_path, transform=transform)
batch_size = 64
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_

# Load models
resnet152_model = models.resnet152(pretrained=True).cuda()
resnet50_model = models.resnet50(pretrained=True).cuda()
resnet18_model = models.resnet18(pretrained=True).cuda()
mobilenet_v2_model = models.mobilenet_v2(pretrained=True).cuda()
vit_large_model = models.vit_b_16(pretrained=True).cuda()

# Convert models to FP16 (half precision)
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 models to CPU and then back to GPU to clear caches
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 models 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()

# Profiling helper function to get FLOPs and Params
def profile(model):
    input = torch.randn(1, 3, 224, 224).cuda()  # Create a random input te
    flops, params = thop.profile(model, inputs=(input, ), verbose=False)
    print(f"Model {model.__class__.__name__} has {params:,} params and use
````
````

```
    return flops, params

# Run profiling for each model
models_dict = {
    'VIT-L/16': vit_large_model,
    'ResNet18': resnet18_model,
    'ResNet50': resnet50_model,
    'ResNet152': resnet152_model,
    'MobileNetV2': mobilenet_v2_model
}

# Store FLOPs and Params
flops_dict = {}
params_dict = {}

for model_name, model in models_dict.items():
    flops, params = profile(model)
    flops_dict[model_name] = flops
    params_dict[model_name] = params

# Plot accuracy vs params and accuracy vs FLOPs graph
# Assuming accuracies are stored for each model (e.g., from a previous ana
accuracies = {
    'VIT-L/16': vit_top1_accuracy,  # Replace with actual accuracy values
    'ResNet18': resnet18_accuracy
```