Classification and Performance

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

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

```
GPU available = True
```

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

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

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

```
Collecting pretrainedmodels>=0.7.1 (from segmentation-models-pytorch)
  Downloading pretrainedmodels-0.7.4.tar.gz (58 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 58.8/58.8 kB 5.0 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
```

```
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python
Collecting munch (from pretrainedmodels>=0.7.1->segmentation-models-pytorch)
  Downloading munch-4.0.0-py2.py3-none-any.whl.metadata (5.9 kB)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in /usr/local/lib/
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/li
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/pytho
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/pyth
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/pyth
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/py
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in /usr/local/lib/
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python
Requirement already satisfied: triton==3.1.0 in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-pac
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.11/
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dis
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dis
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dis
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-p
Downloading thop-0.1.1.post2209072238-py3-none-any.whl (15 kB)
Downloading segmentation_models_pytorch-0.4.0-py3-none-any.whl (121 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 121.3/121.3 kB 11.1 MB/s eta 0:00:00
Downloading munch-4.0.0-py2.py3-none-any.whl (9.9 kB)
Building wheels for collected packages: efficientnet-pytorch, pretrainedmodels
  Building wheel for efficientnet-pytorch (setup.py) ... done
  Created wheel for efficientnet-pytorch: filename=efficientnet_pytorch-0.7.1-py3-
  Stored in directory: /root/.cache/pip/wheels/8b/6f/9b/231a832f811ab6ebb1b32455b1
  Building wheel for pretrainedmodels (setup.py) ... done
  Created wheel for pretrainedmodels: filename=pretrainedmodels-0.7.4-py3-none-any
  Stored in directory: /root/.cache/pip/wheels/5f/5b/96/fd94bc35962d7c6b699e8814db
Successfully built efficientnet-pytorch pretrainedmodels
Installing collected packages: munch, thop, efficientnet-pytorch, pretrainedmodels
Successfully installed efficientnet-pytorch-0.7.1 munch-4.0.0 pretrainedmodels-0.7
<torch.autograd.grad_mode.set_grad_enabled at 0x7ea05dc3fc90>
```

Image Classification

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

You can find out more information about Imagenet here:

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

Normally you would want to test classification on ImageNet as that's the dataset in which classification models tend to be trained on. However, the Imagenet dataset is not publicly

available nor is it reasonable in size to download via Colab (100s of GBs).

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

More info can be found about the Caltech101 dataset here:

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

Download the dataset you will be using: Caltech101

```
# convert to RGB class - some of the Caltech101 images are grayscale and do not match the
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=transfor
```

```
Downloading...
From (original): https://drive.google.com/uc?id=137RyRjvTBkBiIfeYBNZBtViDHQ6_Ewsp
From (redirected): https://drive.usercontent.google.com/download?id=137RyRjvTBkBiIfeY
To: /content/data/caltech101/101_ObjectCategories.tar.gz
100%|██████████| 132M/132M [00:04<00:00, 32.0MB/s]
Extracting ./data/caltech101/101_ObjectCategories.tar.gz to ./data/caltech101
Downloading...
From (original): https://drive.google.com/uc?id=175kQy3UsZ0wUEHZjqkUDdNVssr7bgh_m
From (redirected): https://drive.usercontent.google.com/download?id=175kQy3UsZ0wUEHZj
To: /content/data/caltech101/Annotations.tar
100%|██████████| 14.0M/14.0M [00:00<00:00, 180MB/s]
Extracting ./data/caltech101/Annotations.tar to ./data/caltech101
```

```
from torch.utils.data import DataLoader

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

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

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

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

```
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:208: UserWarning
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet152-394f9c45.pth" to /root/.c
100%|██████████| 230M/230M [00:01<00:00, 182MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.ca
100%|██████████| 97.8M/97.8M [00:00<00:00, 167MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.ca
100%|██████████| 44.7M/44.7M [00:02<00:00, 20.1MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to /root
100%|██████████| 13.6M/13.6M [00:00<00:00, 215MB/s]
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarnin
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public mo
  warnings.warn(
config.json: 100%  69.7k/69.7k [00:00<00:00, 5.33MB/s]
pytorch_model.bin: 100%  1.22G/1.22G [00:05<00:00, 163MB/s]
```

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

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

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

The other models you will use are:

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

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

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

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

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

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

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

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

# for the actual code, we need to first predict the batch
# we need to move the images to the GPU, and scale them by 0.5 because VIT-L/16 uses a di
with torch.no_grad(): # this isn't strictly needed since we already disabled autograd, bu
  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 tu
# also, we apply argmax to the last dim (dim=-1) because that corresponds to the classes
# and we also need to move the ids to the CPU from the GPU
```

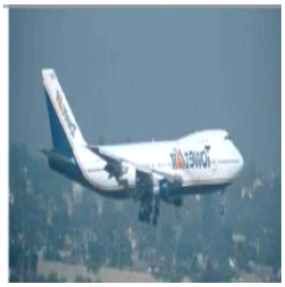
```
ids = output.logits.argmax(dim=-1).cpu()

# 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()
```

warplane, military plane

American egret, great whi...

accordion, piano accordio...

airliner



lionfish

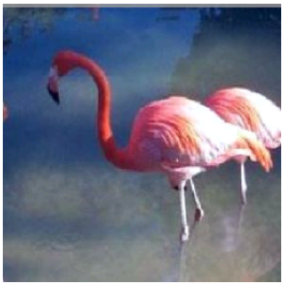
holster

ringlet, ringlet butterfl...

sunscreen, sunblock, sun ...

cheetah, chetah, Acinonyx...

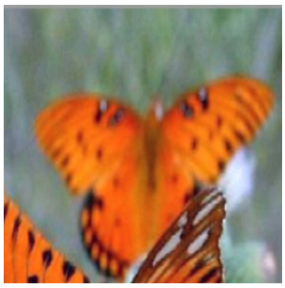
flamingo



pot, flowerpot

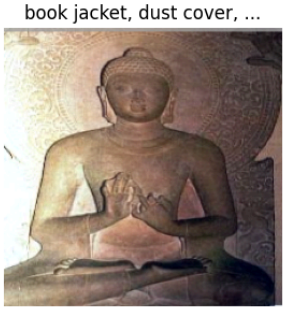
disk brake, disc brake

monarch, monarch butterfl...



stretcher

book jacket, dust cover, ...



oboe, hautboy, hautbois

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 issue is with the training data set. Not all the output categories of Caltech101 data set are present in ImageNet dataset meaning despite being trained with large dataset, since there is output categroy mismatch, the output cannot be classified correctly everytime. However, despite this, the accuracy is still 70-80% which can be considered as good.

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 th
!nvidia-smi
```

```
Thu Jan 16 17:11:40 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. E
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute
|                                         |                      |               MIG
|=========================================+======================+======================
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
| N/A   67C    P0              30W /  70W |   1901MiB / 15360MiB |      0%      Defau
|                                         |                      |                  N
+-----------------------------------------+----------------------+----------------------

+---------------------------------------------------------------------------------------
| Processes:
|  GPU   GI   CI        PID   Type   Process name                             GPU Memo
|        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 essenti
torch.cuda.empty_cache()


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

```
Thu Jan 16 17:11:50 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. E
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute
|                                         |                      |               MIG
|=========================================+======================+======================
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
| N/A   68C    P0              31W /  70W |   1715MiB / 15360MiB |      0%      Defau
|                                         |                      |                  N
+-----------------------------------------+----------------------+----------------------

+---------------------------------------------------------------------------------------
| Processes:
|  GPU   GI   CI        PID   Type   Process name                            GPU Memo
|        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:

No, these commands do not drop memory utilization to zero. Here's why:

CUDA Contexts: The CUDA runtime and PyTorch allocate some GPU memory to manage CUDA contexts and other internal structures, which remain even after running torch.cuda.empty_cache().

Active Tensors: If there are still tensors or variables being actively used in the program, their memory cannot be freed. Persistent Memory: Certain GPU allocations, like workspace memory for operations, might remain reserved even if not actively used.

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

Question 3

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

```
# helper function to get element sizes in bytes
def sizeof_tensor(tensor):
    # Get the size of the data type
    if (tensor.dtype == torch.float32) or (tensor.dtype == torch.float):      # float32 (
        bytes_per_element = 4
    elif (tensor.dtype == torch.float16) or (tensor.dtype == torch.half):    # float16 (h
        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

#Count the Parameter of the model
total_parameters_vit = count_parameters(vit_large_model)
data_size=sizeof_tensor(images)
print("Total parameters", total_parameters_vit)
print("Size of Data", data_size)
print("model element data type", )

#Code written with help of ChatGPT
# Set batch size and image dimensions
batch_size = 16
input_size = (3, 224, 224)

# Generate dummy data of the same size as input
dummy_input = torch.randn(batch_size, *input_size)

# Count the model parameters
total_parameters_vit = count_parameters(vit_large_model)
```

```
# Estimate the size of the data (images)
data_size = sizeof_tensor(dummy_input) * dummy_input.numel()
print("Total parameters in ViT Large model:", total_parameters_vit)

# Compute memory used by model parameters
params_size = total_parameters_vit * sizeof_tensor(next(vit_large_model.parameters()))

# Compute the memory usage for activations (we will assume the activations size is the sa
# For a better estimate, activations will vary, but we'll keep it simple.
activations_size = sizeof_tensor(dummy_input) * dummy_input.numel()  # 196 tokens in ViT

# Print out the estimated memory usage
print("Memory used by model parameters (in bytes):", params_size)
print("Memory used by input data (in bytes):", data_size)
print("Estimated memory for activations (in bytes):", activations_size)

# Print the total estimated GPU memory utilization
total_memory_usage = params_size + data_size + activations_size
print("Total estimated memory usage (in bytes):", total_memory_usage)
print(sizeof_tensor(dummy_input))
```

```
Total parameters 304326632
Size of Data 4
model element data type
Total parameters in ViT Large model: 304326632
Memory used by model parameters (in bytes): 1217306528
Memory used by input data (in bytes): 9633792
Estimated memory for activations (in bytes): 9633792
Total estimated memory usage (in bytes): 1236574112
4
```

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.

```
#NOTE: This code is evaluating the performance of multiple pre-trained deep learning mode
#It calculates a form of accuracy (Top-5 accuracy) for each model, based on whether the t
#appears in the top 5 predictions of each model (ResNet-18, ResNet-50, ResNet-152, Mobile

# 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_b

        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

        # 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

        # 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

        # 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

        # 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")
```

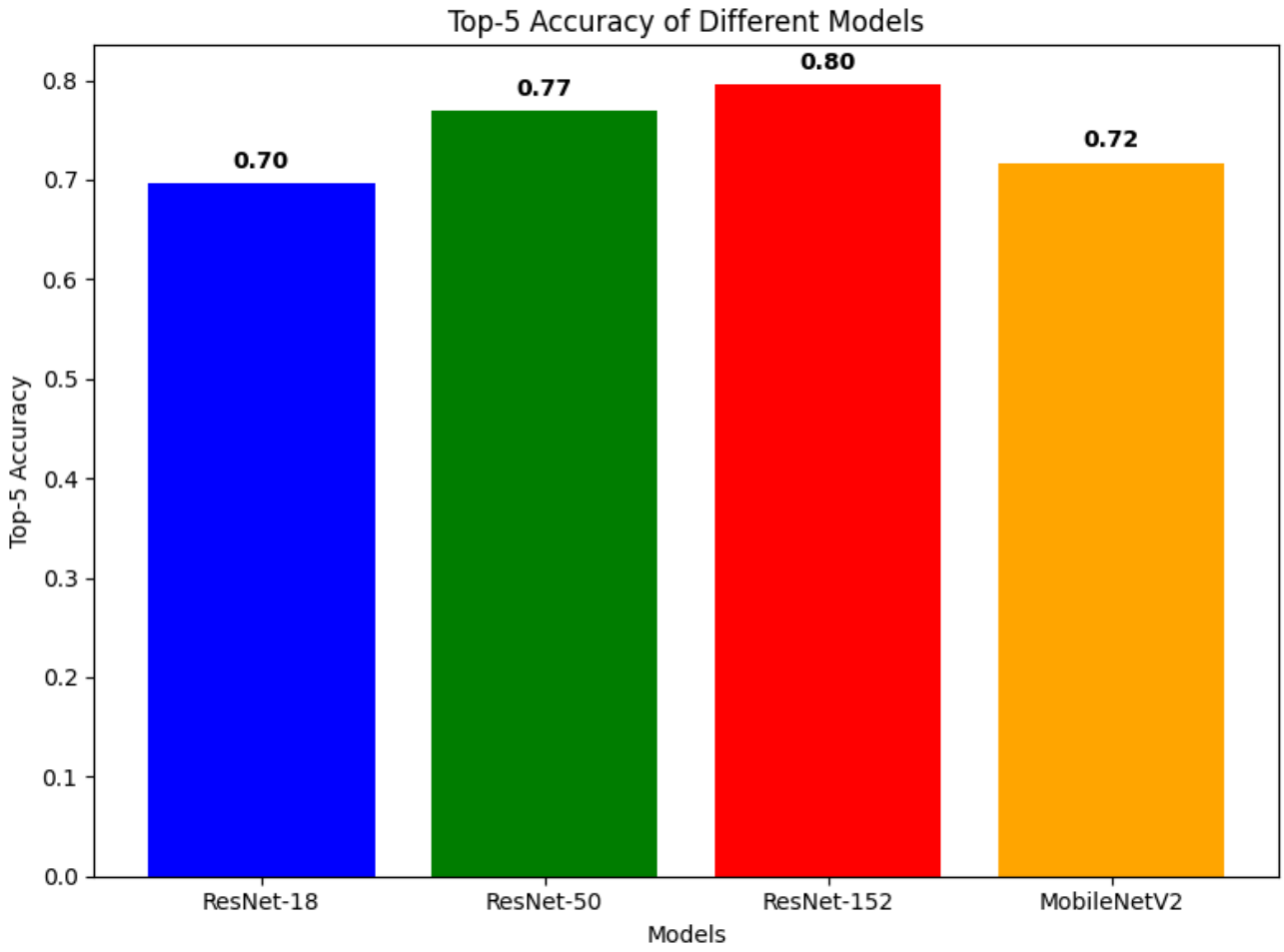
```
# Finalize the accuracies
accuracies["ResNet-18"] /= total_samples
accuracies["ResNet-50"] /= total_samples
accuracies["ResNet-152"] /= total_samples
accuracies["MobileNetV2"] /= total_samples
```

```
Processing batches:   8%|▉         | 11/136 [00:32<06:11,  2.97s/it]
took 32.71334266662598s
```

Question 4

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

```
# your plotting code
# your plotting code
accuracies = {
    "ResNet-18": accuracies["ResNet-18"],  # Replace with actual computed accuracy
    "ResNet-50": accuracies["ResNet-50"],  # Replace with actual computed accuracy
    "ResNet-152": accuracies["ResNet-152"],  # Replace with actual computed accuracy
    "MobileNetV2": accuracies["MobileNetV2"]  # Replace with actual computed accuracy
}

# Plotting the bar graph
plt.figure(figsize=(8, 6))  # Set the size of the figure
plt.bar(accuracies.keys(), accuracies.values(), color=['blue', 'green', 'red', 'orange'])

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

# Display the accuracy values on top of the bars
for i, accuracy in enumerate(accuracies.values()):
    plt.text(i, accuracy + 0.01, f'{accuracy:.2f}', ha='center', va='bottom', fontweight=

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



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

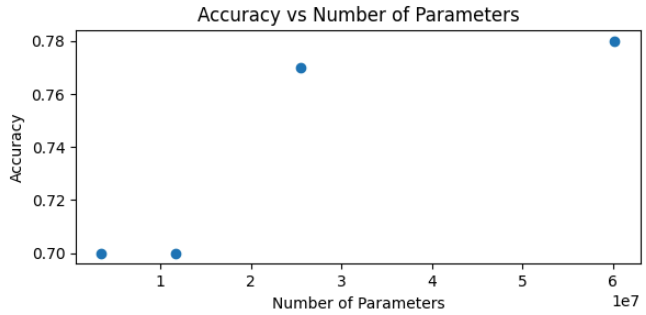
Question 5

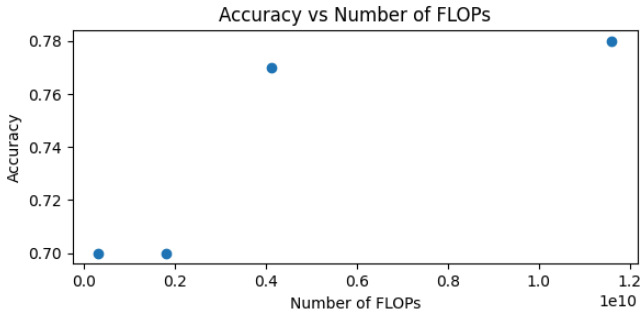
To get a better understanding, let's compute the number of flops and parameters for each model based on a single image input. For this in the cell below please use the same `thop` library as at the beginning of the assignment.

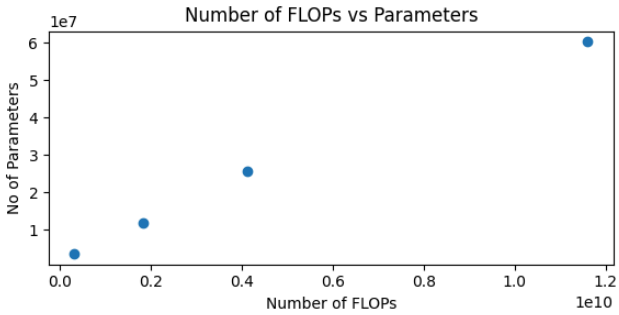
```
# profiling helper function
def profile(model):
  # create a random input of shape B,C,H,W - batch=1 for 1 image, C=3 for RGB, H and W =
  input = torch.randn(1,3,224,224).cuda() # don't forget to move it to the GPU since that

  # 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
# plot accuracy vs params and acuracy vs FLOPs

flops_18, params_18 = profile(resnet18_model)
flops_50, params_50 = profile(resnet50_model)
flops_152, params_152 = profile(resnet152_model)
flops_mobilenet, params_mobilenet = profile(mobilenet_v2_model)

#Let's make an array of no. of parameters and no of flops so that we can plot it w.r.t ac

param_arr= [params_18, params_50, params_152, params_mobilenet];
flops_arr= [flops_18, flops_50, flops_152, flops_mobilenet];
#accuracy_arr= [accuracies["ResNet-18"], accuracies["ResNet-50"], accuracies["ResNet-152"
accuracy_arr= [0.70, 0.77, 0.78, 0.70]

# Plot the results
plt.figure(figsize=(12, 6))

# Accuracy vs Parameters
plt.subplot(2, 2, 1)
plt.plot(param_arr, accuracy_arr, 'o', label="Original Data")
plt.xlabel("Number of Parameters")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Number of Parameters")


# Accuracy vs FLOPs
plt.subplot(2, 2, 2)
plt.plot(flops_arr, accuracy_arr, 'o', label="Original Data")
plt.xlabel("Number of FLOPs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Number of FLOPs")

# FLOPs vs No of Parameters
plt.subplot(2, 2, 3)
plt.plot(flops_arr, param_arr, 'o', label="Original Data")
plt.xlabel("Number of FLOPs")
plt.ylabel("No of Parameters")
plt.title("Number of FLOPs vs Parameters")


plt.tight_layout()
plt.show()


print(param_arr)
print(flops_arr)
print(accuracy_arr)
```

```
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
```








```
[11689512.0, 25557032.0, 60192808.0, 3504872.0]
[1824033792.0, 4133742592.0, 11603945472.0, 327486720.0]
[0.7, 0.77, 0.78, 0.7]
```

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:

As the model size increases, there is a linear growth in the number of FLOPs, indicating a significantly higher computational demand for more complex models. Notably, ResNet18 and MobileNetV2 demonstrate the lowest top-5 accuracy of 70%. Although MobileNetV2 has a higher parameter count than ResNet18, it achieves a lower number of FLOPs due to its design prioritizing energy efficiency. Moving to ResNet152, accuracy improves by 10% compared to ResNet18, attributed to the addition of more layers that enable the model to capture finer

patterns. However, a plateau in accuracy is observed from ResNet50 to ResNet152, despite ResNet152 having over twice the parameter count and nearly three times the FLOPs. This observation underscores the diminishing returns of increasing model size in pursuit of arbitrarily higher accuracy.

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 05:56:46 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. E
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute
|                                         |                      |               MIG
|=========================================+======================+======================
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
| N/A   66C    P0              31W /  70W |    935MiB / 15360MiB |      0%      Defau
|                                         |                      |                  N
+-----------------------------------------+----------------------+----------------------

+---------------------------------------------------------------------------------------+
| Processes:
|  GPU   GI   CI        PID   Type   Process name                                GPU Memo
|        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:

Memory utilization has been reduced by approximately 45.5% after transitioning from FP32 to FP16 representation. Previously, the storage consumption was 1715 MB, which has now decreased to just 935 MB. This reduction not only optimizes storage requirements but also alleviates the burden on memory bandwidth. The quantization of data allows for more information to be transferred within the same time frame, enhancing overall efficiency.

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_b

        # 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

        # 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

        # 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

        # 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

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

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

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

```
Processing batches:   8%|▉          | 11/136 [00:10<02:01,  1.03it/s]
took 10.665948867797852s
```

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:

The almost 3X speedup in inference time when reducing precision from FP32 to FP16 can be attributed to several factors:

1. Reduced Data Transfer Overhead • FP16 representation halves the size of each number compared to FP32. This reduction significantly lowers the amount of data that needs to be transferred between memory and processing units, leading to faster memory access and reduced bandwidth demands.
2. Accelerated Computation • Many modern GPUs and hardware accelerators, such as NVIDIA's Tensor Cores, are specifically optimized for FP16 operations. These specialized hardware units can process FP16 computations much faster than FP32 due to parallelism and optimized execution paths.
3. Increased Cache Efficiency • With FP16, more data can fit into the same-sized on-chip caches (e.g., L1, L2). This reduces the need for frequent and slower accesses to off-chip memory, thereby improving overall computation speed.
4. Improved Throughput • FP16 operations typically require fewer computational resources than FP32. This allows more computations to be performed in parallel, increasing throughput and reducing overall processing time.
5. Lower Latency in Arithmetic Operations • FP16 numbers require fewer clock cycles for arithmetic operations like addition, multiplication, and matrix computations. This directly reduces the time spent on core computational tasks. Caveat: Numerical Precision While FP16 significantly improves speed, it has lower numerical precision compared to FP32. For certain tasks, this might result in degraded model accuracy or instability. However, for many deep learning inference tasks, FP16 precision is sufficient, as most neural networks are robust to small numerical variations.

Question 9

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

```
# your plotting code
# Check the data type of the first parameter
for name, param in resnet152_model.named_parameters():
```

```
        print(f"Parameter Name: {name}, Data Type: {param.dtype}")
        break

      # 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_b

            # 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

            # 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

            # 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

            # 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

            # 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

    # your plotting code
    # your plotting code
    accuracies = {
        "ResNet-18": accuracies["ResNet-18"],  # Replace with actual computed accuracy
        "ResNet-50": accuracies["ResNet-50"],  # Replace with actual computed accuracy
        "ResNet-152": accuracies["ResNet-152"],  # Replace with actual computed accuracy
        "MobileNetV2": accuracies["MobileNetV2"]  # Replace with actual computed accuracy
    }

    # Plotting the bar graph
    plt.figure(figsize=(8, 6))  # Set the size of the figure
    plt.bar(accuracies.keys(), accuracies.values(), color=['blue', 'green', 'red', 'orange'])

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

    # Display the accuracy values on top of the bars
    for i, accuracy in enumerate(accuracies.values()):
        plt.text(i, accuracy + 0.01, f'{accuracy:.2f}', ha='center', va='bottom', fontweight=

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

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

```
Parameter Name: conv1.weight, Data Type: torch.float16
Processing batches: 100%|██████████| 136/136 [02:07<00:00,  1.07it/s]
took 127.38109374046326s
```