Classification and Performance

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

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

```
GPU available = True
```

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

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

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

```
Collecting thop
  Downloading thop-0.1.1.post2209072238-py3-none-any.whl.metadata (2.7 kB)
Collecting segmentation-models-pytorch
  Downloading segmentation_models_pytorch-0.4.0-py3-none-any.whl.metadata (32 kB)
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-packages (4.47.1
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from thop) (2
Collecting efficientnet-pytorch>=0.6.1 (from segmentation-models-pytorch)
  Downloading efficientnet_pytorch-0.7.1.tar.gz (21 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: huggingface-hub>=0.24 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: numpy>=1.19.3 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.10/dist-packages (from segm
Collecting pretrainedmodels>=0.7.1 (from segmentation-models-pytorch)
  Downloading pretrainedmodels-0.7.4.tar.gz (58 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 58.8/58.8 kB 5.3 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from segme
Requirement already satisfied: timm>=0.9 in /usr/local/lib/python3.10/dist-packages (from segm
Requirement already satisfied: torchvision>=0.9 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist-packages (from s
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from trans
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from tr
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (f
```

```
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from trans
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-pa
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-packages (from torch
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch->
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from r
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (fro
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 11.2 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-none-any.whl
  Stored in directory: /root/.cache/pip/wheels/03/3f/e9/911b1bc46869644912bda90a56bcf7b960f20b
  Building wheel for pretrainedmodels (setup.py) ... done
  Created wheel for pretrainedmodels: filename=pretrainedmodels-0.7.4-py3-none-any.whl size=60
  Stored in directory: /root/.cache/pip/wheels/35/cb/a5/8f534c60142835bfc889f9a482e4a67e0b8170
Successfully built efficientnet-pytorch pretrainedmodels
Installing collected packages: munch, thop, efficientnet-pytorch, pretrainedmodels, segmentati
Successfully installed efficientnet-pytorch-0.7.1 munch-4.0.0 pretrainedmodels-0.7.4 segmentat
<torch.autograd.grad_mode.set_grad_enabled at 0x7e3f1f4dd7e0>
```

Image Classification

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

You can find out more information about Imagenet here:

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

Normally you would want to test classification on ImageNet as that's the dataset in which classification models tend to be trained on. However, the Imagenet dataset is not publicly available nor is it reasonable in size to download via Colab (100s of GBs).

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

More info can be found about the Caltech101 dataset here:

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

Download the dataset you will be using: Caltech101

```
# convert to RGB class - some of the Caltech101 images are grayscale and do not match the tensor s
class ConvertToRGB:
    def __call__(self, image):
        # If grayscale image, convert to RGB
        if image.mode == "L":
            image = Image.merge("RGB", (image, image, image))
        return image

# Define transformations
transform = transforms.Compose([
    ConvertToRGB(), # first convert to RGB
    transforms.Resize((224, 224)),  # Most pretrained models expect 224x224 inputs
    transforms.ToTensor(),
    # this normalization is shared among all of the torch-hub models we will be using
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# Download the dataset
caltech101_dataset = datasets.Caltech101(root="./data", download=True, transform=transform)
```

```
Downloading...
From (original): https://drive.google.com/uc?id=137RyRjvTBkBiIfeYBNZBtViDHQ6_Ewsp
From (redirected): https://drive.usercontent.google.com/download?id=137RyRjvTBkBiIfeYBNZBtViDH
To: /content/data/caltech101/101_ObjectCategories.tar.gz
100%|██████████| 132M/132M [00:01<00:00, 108MB/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=175kQy3UsZ0wUEHZjqkUDdNVss
To: /content/data/caltech101/Annotations.tar
100%|██████████| 14.0M/14.0M [00:00<00:00, 51.5MB/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/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The par
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Argumen
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet152-394f9c45.pth" to /root/.cache/torc
100%|██████████| 230M/230M [00:01<00:00, 179MB/s]
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Argumen
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.cache/torch
100%|██████████| 97.8M/97.8M [00:00<00:00, 163MB/s]
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Argumen
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/torch
100%|██████████| 44.7M/44.7M [00:00<00:00, 166MB/s]
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Argumen
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/mobilenet_v2-b0353104.pth" to /root/.cache/t
100%|██████████| 13.6M/13.6M [00:00<00:00, 125MB/s]
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggin
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or d
  warnings.warn(
```

config.json: 100% 69.7k/69.7k [00:00<00:00, 4.73MB/s]

pytorch_model.bin: 100% 1.22G/1.22G [00:10<00:00, 121MB/s]

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

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

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

The other models you will use are:

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

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

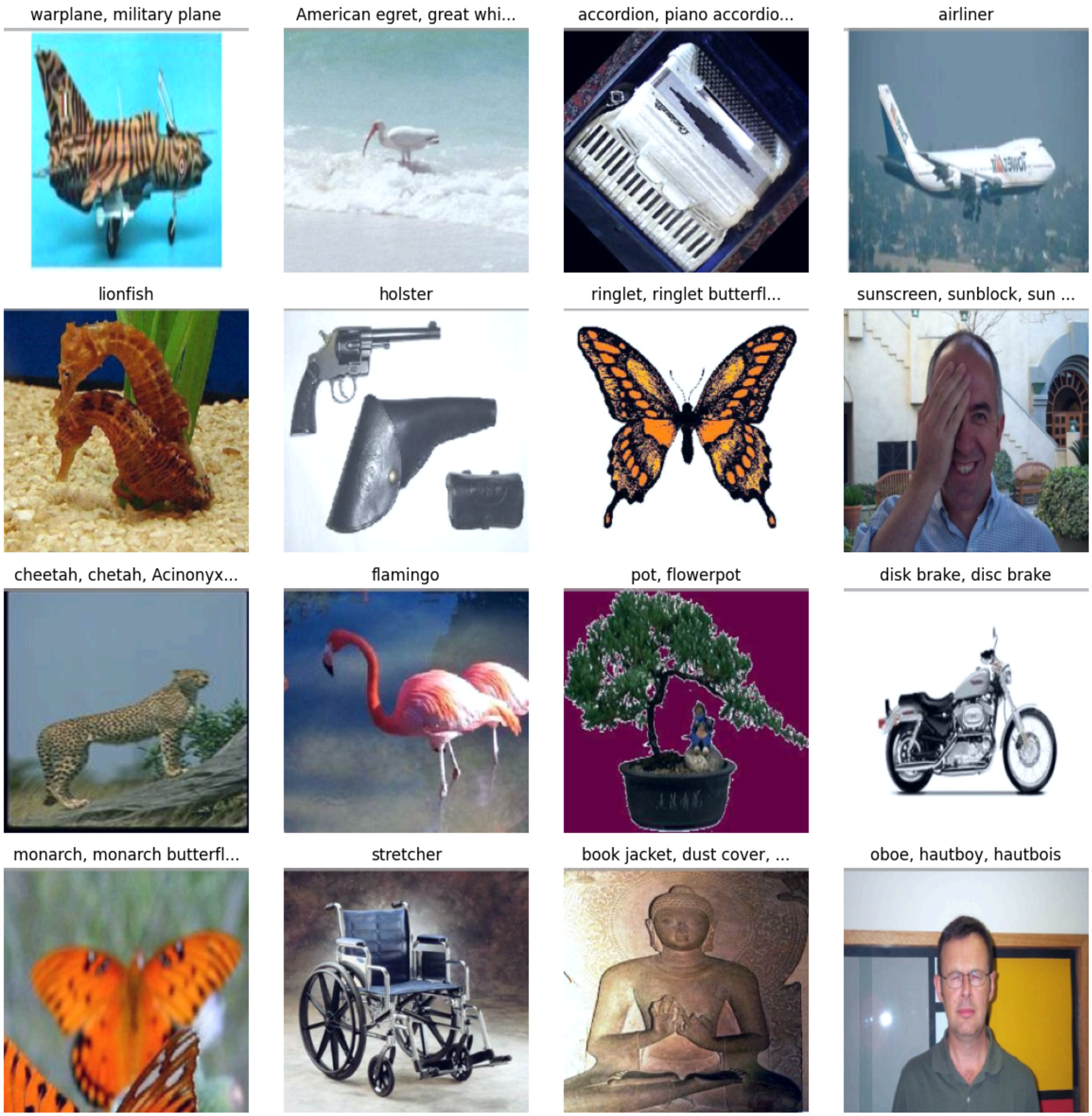
There's a good overview of the different versions here: https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8

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

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

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

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

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

```
    plt.axis('off')

# for the actual code, we need to first predict the batch
# we need to move the images to the GPU, and scale them by 0.5 because VIT-L/16 uses a different n
with torch.no_grad(): # this isn't strictly needed since we already disabled autograd, but we shou
  output = vit_large_model(images.cuda()*0.5)

# then we can sample the output using argmax (find the class with the highest probability)
# here we are calling output.logits because huggingface returns a struct rather than a tuple
# also, we apply argmax to the last dim (dim=-1) because that corresponds to the classes - the sha
# and we also need to move the ids to the CPU from the GPU
ids = output.logits.argmax(dim=-1).cpu()

# next we will go through all of the ids and convert them into human readable labels
# huggingface has the .config.id2label map, which helps.
# notice that we are calling id.item() to get the raw contents of the ids tensor
labels = []
for id in ids:
  labels += [vit_large_model.config.id2label[id.item()]]

# finally, let's plot the first 4 images
max_label_len = 25
fig, axes = plt.subplots(4, 4, figsize=(12, 12))
for i in range(4):
    for j in range(4):
        idx = i*4 + j
        plt.sca(axes[i, j])
        imshow(images[idx])
        # we need to trim the labels because they sometimes are too long
        if len(labels[idx]) > max_label_len:
          trimmed_label = labels[idx][:max_label_len] + '...'
        else:
          trimmed_label = labels[idx]
        axes[i,j].set_title(trimmed_label)
plt.tight_layout()
plt.show()
```



Question 1

Given the above classifications, how well do you think the model does? Can you observe any limitations? If so, do you think that's related to the model size and complexity, or is it more likely related to the training set?

For more information, the class list can be found here: https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/

Please answer below:

The model does just "okay" at identifying each of the objects. The main limitation that I see is with identifying people. This could be due to the training set not having enough images of people in different scenairos to train the AI on.

Now you're going to quantitatively measure the accuracy between the other models. The first thing you need to do is clear the GPU cache, to prevent an out-of-memory error. To undestand this, let's look at the current GPU memory utilization.

```
# run nvidia-smi to view the memory usage. Notice the ! before the command, this sends the command
!nvidia-smi
```

```
Wed Jan 15 09:06:34 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   54C    P0              28W /  70W |   1901MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

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

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

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

```
Wed Jan 15 09:07:22 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   57C    P0              28W /  70W |   1715MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

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

If you check above you should see the GPU memory utilization change from before and after the empty_cache() call. Memory management is one of the quirks that must be considered when dealing with accelerators like a GPU. Unlike with a CPU, there is no swap file to page memory in and out of the device. Instead, this must be handled by the user. When too much of the GPU memory is used, the driver will throw an out-of-memory error (commonly referred to as OOM). In this case, the process often ends up in an unrecoverable state and needs to be restarted to fully reset the memory utilization to zero.

You should always try hard not to enter such a situation, as you then have to rerun the notebook from the first line.

Question 2

Given the above, why is the GPU memory utilization not zero? Does the current utilization match what you would expect? Please answer below:

Because I called torch.cuda's empty_cache() function I utilized the GPU, the GPU also is most likely running some background functions in its idle state that takes up some memory. The memory usage dropped from 1901 MiB to down to 1715 MiB. This is not a massive change but it is still a difference.

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

Question 3

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

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

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

# estimate the current GPU memory utilization
def estimate_memory_utilization(model, dtype=torch.float32, activations=0):
    param_size = sizeof_tensor(torch.zeros(1, dtype=dtype))
    total_params = count_parameters(model)
    params_memory = total_params * param_size / (1024 ** 2)
    activations_memory = activations * param_size / (1024 ** 2)
    return params_memory + activations_memory
estimated_memory = estimate_memory_utilization(vit_large_model)
print(str(estimated_memory) + " MiB" )
```

```
1160.9139709472656 MiB
```

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

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

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

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

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

num_batches = len(dataloader)

t_start = time.time()
```

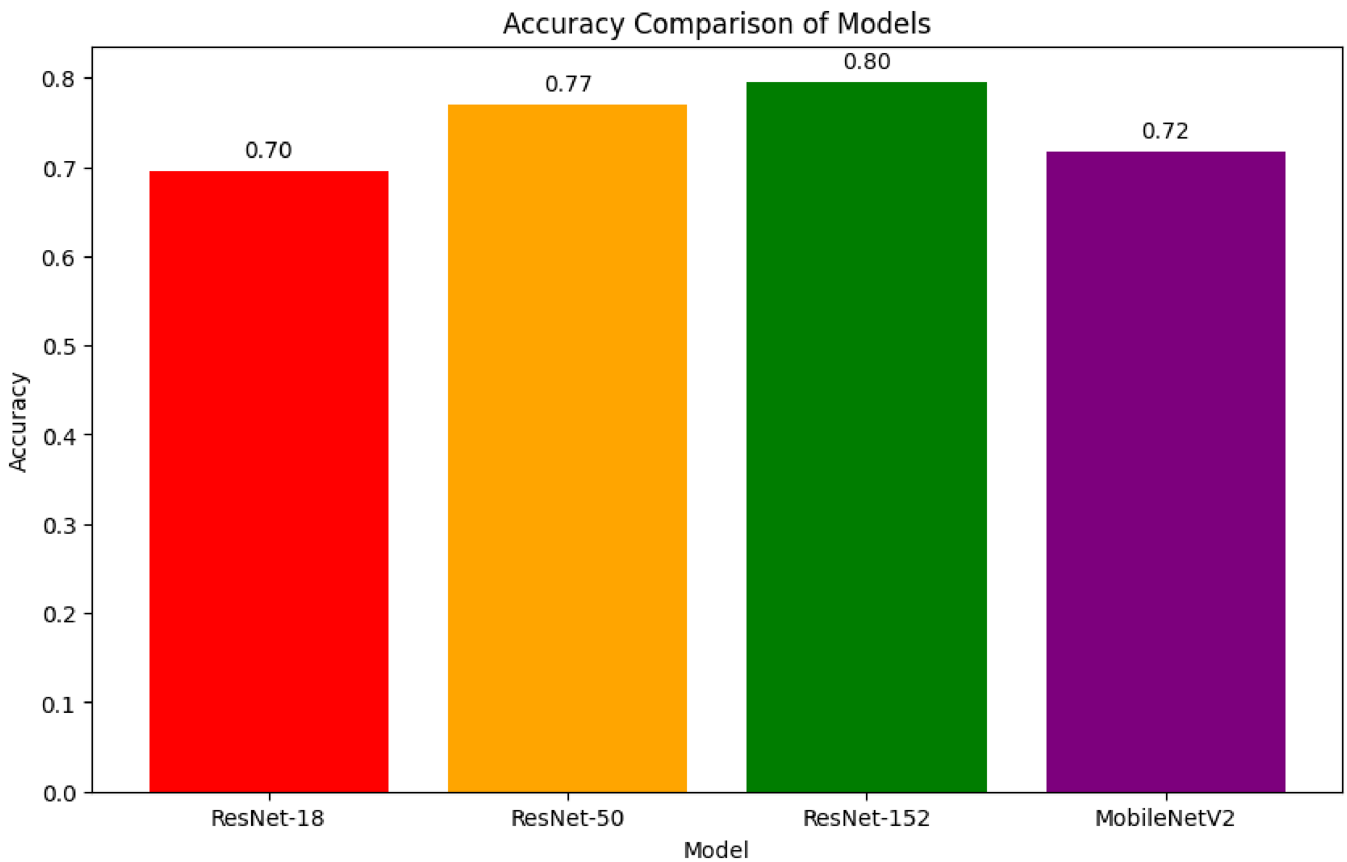
```
with torch.no_grad():
  for i, (inputs, _)in tqdm(enumerate(dataloader), desc="Processing batches", total=num_batches):

        if i > 10:
          break

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

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

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

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

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

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

        # 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 [00:33<06:21,  3.05s/it]
took 33.57676434516907s
```

Question 4

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

```
# Define the models and their accuracies
models = list(accuracies.keys())
accuracy_values = list(accuracies.values())

plt.figure(figsize=(10, 6))
plt.bar(models, accuracy_values, color=['red', 'orange', 'green', 'purple'])


plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Accuracy Comparison of Models')

for i, value in enumerate(accuracy_values):
    plt.text(i, value + 0.01, f'{value:.2f}', ha='center', va='bottom')
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?

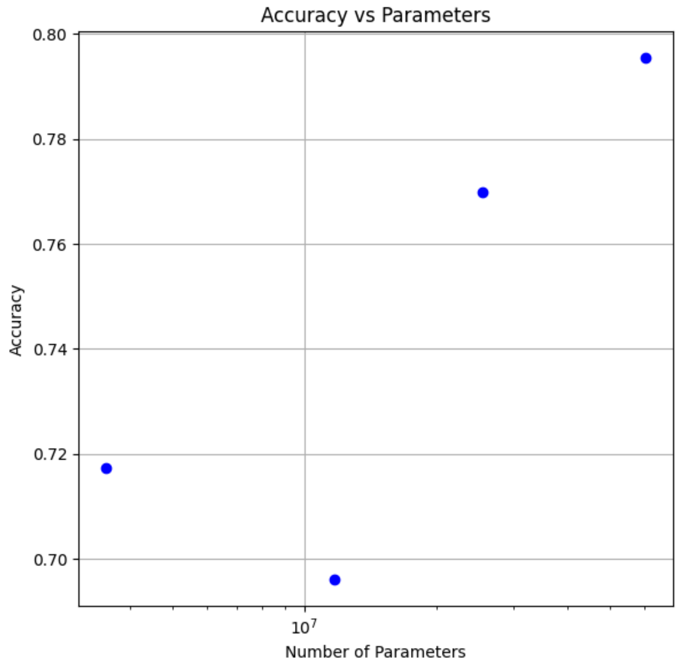
Question 5

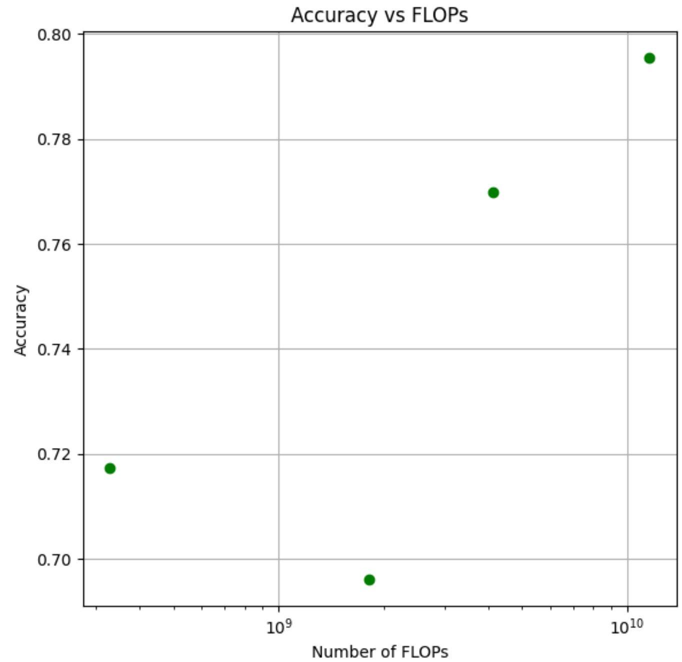
To get a better understanding, let's compute the number of flops and parameters for each model based on a single image input. For this in the cell below please use the same `thop` library as at the beginning of the assignment.

```
def profile(model):
  input = torch.randn(1,3,224,224).cuda()

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

```
  print(f"model {model.__class__.__name__} has {params:,} params and uses {flops:,} FLOPs")
  return flops, params



models = {
    "ResNet-18": resnet18_model,
    "ResNet-50": resnet50_model,
    "ResNet-152": resnet152_model,
    "MobileNetV2": mobilenet_v2_model
}
flops_params = {}
# Profiling each model
for name, model in models.items():
    flops, params = profile(model)
    flops_params[name] = {"flops": flops, "params": params}

# Extracting data for plotting
model_names = list(flops_params.keys())
flops_values = [flops_params[name]["flops"] for name in model_names]
params_values = [flops_params[name]["params"] for name in model_names]
accuracy_values = [accuracies[name] for name in model_names]  # Assuming accuracies is defined as

# Plot Accuracy vs Parameters
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(params_values, accuracy_values, color='blue')
plt.title('Accuracy vs Parameters')
plt.xlabel('Number of Parameters')
plt.ylabel('Accuracy')
plt.xscale('log')  # Log scale for better visualization
plt.grid(True)

# Plot Accuracy vs FLOPs
plt.subplot(1, 2, 2)
plt.scatter(flops_values, accuracy_values, color='green')
plt.title('Accuracy vs FLOPs')
plt.xlabel('Number of FLOPs')
plt.ylabel('Accuracy')
plt.xscale('log')  # Log scale for better visualization
plt.grid(True)

# Show plots
plt.tight_layout()
plt.show()
```

```
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
```





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:

In general, the more parameters and FLOPs there are, the better the accuracy of the model.

Double-click (or enter) to edit

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
```

```
Wed Jan 15 09:10:58 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   71C    P0              32W /  70W |    935MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

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

Question 7

Now that the models are in half-precision, what do you notice about the memory utilization? Is the utilization what you would expect from your previous expected calculation given the new data types? Please answer below:

It appears that there is a significant decrease in memory usage when switching to half-precision. The utilization is expected, it is about half of what it was originally.

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

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

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

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

num_batches = len(dataloader)

t_start = time.time()

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

        if i > 10:
          break

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

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

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

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

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

        # MobileNetV2 predictions
```

```
        logits_mobilenetv2 = mobilenet_v2_model(inputs)
        top5_preds_mobilenetv2 = logits_mobilenetv2.topk(5, dim=1).indices
        matches_mobilenetv2 = (baseline_preds.unsqueeze(1) == top5_preds_mobilenetv2).any(dim=1).f

        # 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,  1.02it/s]
took 10.741234540939331s
```

Question 8

Did you observe a speedup? Was this result what you expected? What are the pros and cons to using a lower-precision format? Please answer below:

Yes there was a speed up, it was about three times faster than the original precision format. It is a little faster than I expected. The pros of using this lower precision format is better speed/throughput from the GPU and less memory used by the GPU. This has the negative effect however of worse precision and accuracy by the model.

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).

```
accuracies = {"ResNet-18": 0, "ResNet-50": 0, "ResNet-152": 0, "MobileNetV2": 0}
total_samples = 0

num_batches = len(dataloader)

t_start = time.time()

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

        output = vit_large_model(inputs * 0.5)
        baseline_preds = output.logits.argmax(-1)

        logits_resnet18 = resnet18_model(inputs)
        top5_preds_resnet18 = logits_resnet18.topk(5, dim=1).indices
```

```
        matches_resnet18 = (baseline_preds.unsqueeze(1) == top5_preds_resnet18).any(dim=1).float()

        logits_resnet50 = resnet50_model(inputs)
        top5_preds_resnet50 = logits_resnet50.topk(5, dim=1).indices
        matches_resnet50 = (baseline_preds.unsqueeze(1) == top5_preds_resnet50).any(dim=1).float()

        logits_resnet152 = resnet152_model(inputs)
        top5_preds_resnet152 = logits_resnet152.topk(5, dim=1).indices
        matches_resnet152 = (baseline_preds.unsqueeze(1) == top5_preds_resnet152).any(dim=1).float

        logits_mobilenetv2 = mobilenet_v2_model(inputs)
        top5_preds_mobilenetv2 = logits_mobilenetv2.topk(5, dim=1).indices
        matches_mobilenetv2 = (baseline_preds.unsqueeze(1) == top5_preds_mobilenetv2).any(dim=1).f

        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")
accuracies["ResNet-18"] /= total_samples
accuracies["ResNet-50"] /= total_samples
accuracies["ResNet-152"] /= total_samples
accuracies["MobileNetV2"] /= total_samples


for model, accuracy in accuracies.items():
    print(f"{model} accuracy: {accuracy:.4f}")
```

```
Processing batches: 100%|██████████| 136/136 [02:11<00:00,  1.03it/s]
took 131.69105529785156s
ResNet-18 accuracy: 0.6854
ResNet-50 accuracy: 0.7579
ResNet-152 accuracy: 0.7931
MobileNetV2 accuracy: 0.6981
```

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

    return flops, params

model_flops_params = {}
for model in [resnet18_model, resnet50_model, resnet152_model, mobilenet_v2_model]:
    flops, params = profile(model)
    model_flops_params[model.__class__.__name__] = (flops, params)

for model_name, (flops, params) in model_flops_params.items():
    print(f"{model_name}: {params:,} parameters, {flops:,} FLOPs")
```

```
ResNet: 60,192,808.0 parameters, 11,603,945,472.0 FLOPs
MobileNetV2: 3,504,872.0 parameters, 327,486,720.0 FLOPs
```