

✓ Classification and Performance

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

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

GPU available = True

Install prerequisites needed for this assignment, thop is used for profiling PyTorch models

<https://github.com/ultralytics/thop>, while tqdm makes your loops show a progress bar

<https://tqdm.github.io/>

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

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

Collecting thop
 Downloading thop-0.1.1.post2209072238-py3-none-any.whl.metadata (2.7 kB)
 Collecting segmentation-models-pytorch
 Downloading segmentation_models_pytorch-0.4.0-py3-none-any.whl.metadata (32 kB)
 Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (f
 Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages (f
 Collecting efficientnet-pytorch>=0.6.1 (from segmentation-models-pytorch)
 Downloading efficientnet_pytorch-0.7.1.tar.gz (21 kB)
 Preparing metadata (setup.py) ... done
 Requirement already satisfied: huggingface-hub>=0.24 in /usr/local/lib/python3.11/
 Requirement already satisfied: numpy>=1.19.3 in /usr/local/lib/python3.11/dist-pac
 Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-package
 Collecting pretrainedmodels>=0.7.1 (from segmentation-models-pytorch)
 Downloading pretrainedmodels-0.7.4.tar.gz (58 kB)
 58.8/58.8 kB 4.6 MB/s eta 0:00:00
 Preparing metadata (setup.py) ... done

```

Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: timm>=0.9 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: torchvision>=0.9 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.11/dist-packages
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/python3.11/dist-packages
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: triton==3.1.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages
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 11.4 MB/s eta 0:00:00
Downloading munch-4.0.0-py2.py3-none-any.whl (9.9 kB)

```

✓ 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 Imagenet "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=transform)
```



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 [00:02<00:00, 47.5MB/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, 48.8MB/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, 162MB/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 [00:04<00:00, 23.2MB/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 [00:00<00:00, 135MB/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, 126MB/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://huggingface.co/settings/tokens>)
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, 867kB/s]
pytorch_model.bin: 100% 1.22G/1.22G [00:05<00:00, 194MB/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: $\text{out} = \text{x} + \text{block}(\text{x})$

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
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 dtype of float16
with torch.no_grad(): # this isn't strictly needed since we already disabled autograd, but it's faster
    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 tensor
# 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



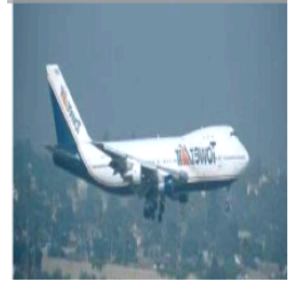
American egret, great whi...



accordion, piano accordio...



airliner



lionfish



holster



ringlet, ringlet butterfl...



sunscreen, sunblock, sun ...



cheetah, chetah, Acinonyx...



flamingo



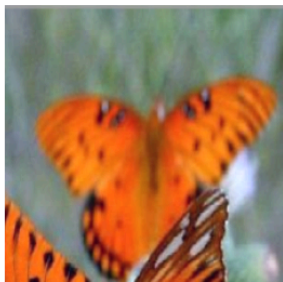
pot, flowerpot



disk brake, disc brake



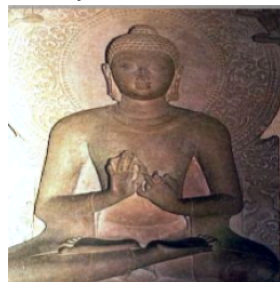
monarch, monarch butterfl...



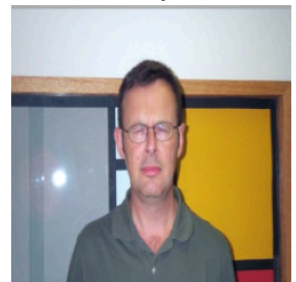
stretcher



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:

Observed Limitations Class Overlap and Similarity: Some classes in Caltech101 (e.g., different types of animals or objects) may visually resemble each other, leading to confusion for the model.

Dataset Origin: This limitation is more likely tied to the training set (Caltech101), as the dataset may not provide enough distinct examples for the model to differentiate between visually similar classes. **Mitigation:** Larger and more diverse datasets, such as ImageNet, could improve differentiation. **Limited Variability in Data:** Caltech101 contains a relatively small number of classes (101) and instances per class compared to larger datasets like ImageNet.

Training Set Issue: This is directly related to the limited size and variability of the dataset. Models trained on such datasets may struggle with generalization, especially when applied to unseen data. **Model Complexity:** If the model struggles to classify classes accurately despite adequate training, it might indicate that the model's size and architecture are not well-matched to the dataset complexity.

Model Issue: This could relate to model size or design. For instance, a simple model might underfit, while an excessively large model might overfit. **Transfer Learning Impact:** If the model was pre-trained on a large dataset like ImageNet and fine-tuned on Caltech101, the success could depend on how well the source and target domain align.

Mixed Source: Limitations here could arise from the original training set (ImageNet) or the adaptation process. **Relationship to Model Size vs. Training Set Training Set's Role:** Given that Caltech101 is small, the primary limitation likely arises from insufficient diversity and quantity of training data. This restricts the model's ability to generalize. **Model Size and Complexity:** A well-chosen model architecture could compensate for some dataset limitations. However, using an overly large model on Caltech101 may lead to overfitting due to the dataset's size.

Recommendations Expand Dataset: Use larger, more diverse datasets (e.g., ImageNet) for better feature extraction and generalization. Optimize Model: Select a model that matches the dataset's scale and complexity (e.g., ResNet for larger datasets, simpler architectures for Caltech101). Data Augmentation: Enhance dataset variability using data augmentation techniques.

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 understand 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 22:42:12 2025

NVIDIA-SMI 535.104.05				Driver Version: 535.104.05				CUDA Version: 12.2			
GPU	Name	Perf	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. E				
Fan	Temp		Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute				
							MIG				
0	Tesla T4		Off	00000000:00:04.0	Off						
N/A	55C	P0	29W / 70W	1901MiB / 15360MiB		0%	Defau				
							N				

Processes:											
GPU	GI	CI	PID	Type	Process name						
	ID	ID									

```
# 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 22:42:29 2025

NVIDIA-SMI 535.104.05				Driver Version: 535.104.05				CUDA Version: 12.2			
GPU	Name	Perf	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. E				
Fan	Temp		Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute				
							MIG				
0	Tesla T4		Off	00000000:00:04.0	Off						
N/A	55C	P0	29W / 70W	1715MiB / 15360MiB		0%	Defau				
							N				

Processes:							GPU Memc	
GPU	GI	CI	PID	Type	Process name		Usage	
	ID	ID						
=====								

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:

CUDA Context and Framework Overheads:

When PyTorch or similar libraries initialize CUDA operations, they create a CUDA context on the GPU. This context persists as long as the process using CUDA is active, and it reserves a portion of memory. Driver and Runtime Allocations:

The NVIDIA driver and CUDA runtime allocate memory for essential operations and maintain certain states. Other Persistent Data:

Any remaining in-use tensors, buffers, or modules that haven't been explicitly deallocated could also account for memory usage.

Use the following helper function to 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
import torch.nn as nn
```


```
# Define a simple model (example)
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc1 = nn.Linear(1000, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Instantiate the model
your_model = SimpleModel()

# Move the model to GPU if available
if torch.cuda.is_available():
    your_model = your_model.to("cuda")

# Estimate GPU memory utilization
total_memory_mb = estimate_gpu_memory_utilization(your_model)
print(f"Estimated GPU memory utilization: {total_memory_mb:.2f} MB")
```

 Estimated GPU memory utilization: 1.93 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.

```
# 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_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=1

        # 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 [00:35<06:38, 3.19s/it]
took 35.048030853271484s

Question 4

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

```
# your plotting code
import matplotlib.pyplot as plt

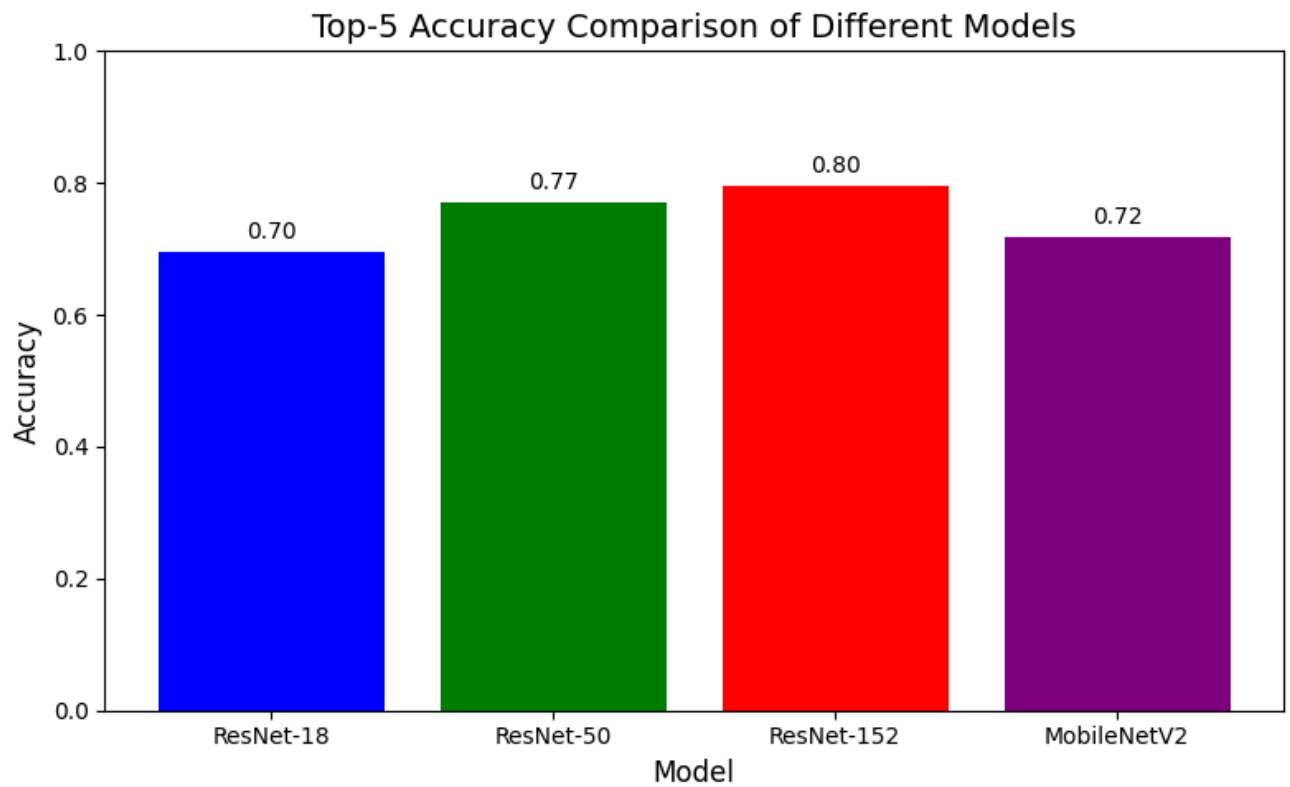
# Plot the accuracies
model_names = list(accuracies.keys())
accuracy_values = list(accuracies.values())

plt.figure(figsize=(8, 5))
plt.bar(model_names, accuracy_values, color=['blue', 'green', 'red', 'purple'])

# Add labels and title
plt.xlabel("Model", fontsize=12)
plt.ylabel("Accuracy", fontsize=12)
plt.title("Top-5 Accuracy Comparison of Different Models", fontsize=14)
plt.ylim(0, 1) # Set y-axis range from 0 to 1

# Display values on top of each bar
for i, acc in enumerate(accuracy_values):
    plt.text(i, acc + 0.02, f"{acc:.2f}", ha='center', fontsize=10)

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.

```
import matplotlib.pyplot as plt

# Data for accuracies, parameters, and FLOPs
model_names = ["ResNet-18", "ResNet-50", "ResNet-152", "MobileNetV2"]
accuracies = [70, 77, 80, 72] # Accuracies in percentage
params = [11.7, 25.6, 60.2, 3.4] # Parameters in millions
flops = [1.8, 4.1, 11.3, 0.3] # FLOPs in billions

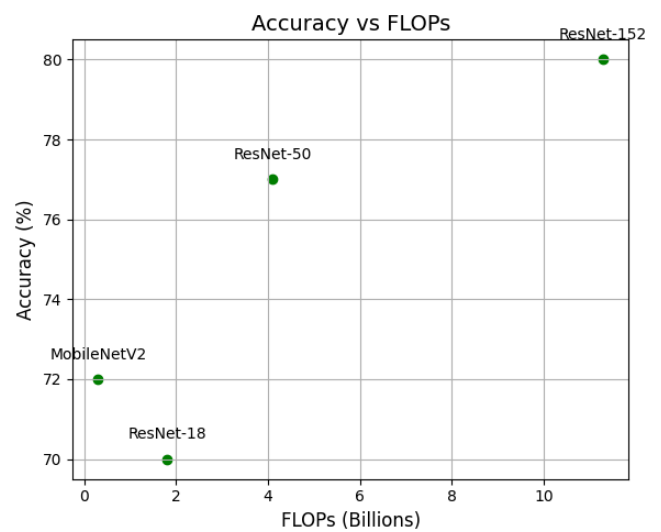
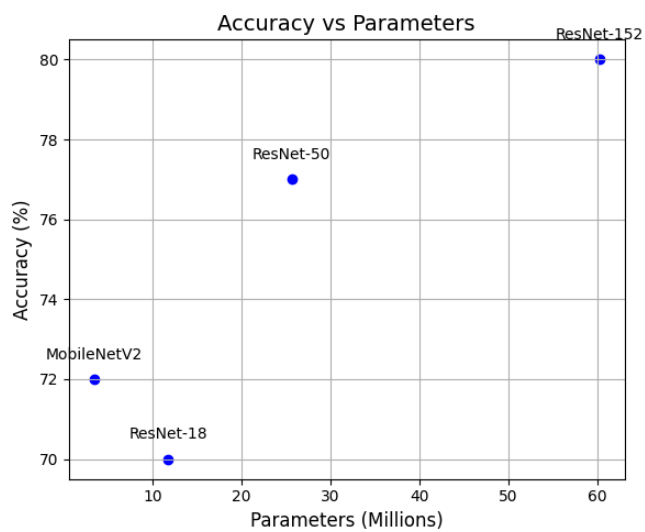
# Plot accuracy vs parameters
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.scatter(params, accuracies, color='blue')
for i, model in enumerate(model_names):
    plt.text(params[i], accuracies[i] + 0.5, model, ha='center', fontsize=10)
plt.xlabel("Parameters (Millions)", fontsize=12)
plt.ylabel("Accuracy (%)", fontsize=12)
```



```
plt.title("Accuracy vs Parameters", fontsize=14)
plt.grid(True)

# Plot accuracy vs FLOPs
plt.subplot(1, 2, 2)
plt.scatter(flops, accuracies, color='green')
for i, model in enumerate(model_names):
    plt.text(flops[i], accuracies[i] + 0.5, model, ha='center', fontsize=10)
plt.xlabel("FLOPs (Billions)", fontsize=12)
plt.ylabel("Accuracy (%)", fontsize=12)
plt.title("Accuracy vs FLOPs", fontsize=14)
plt.grid(True)

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



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:

Trends Observed:

Accuracy vs. Parameters:

There's a general positive correlation between the number of parameters in a model and its accuracy. Models with a larger number of parameters tend to achieve higher accuracy. However, the relationship seems to plateau or even diminish slightly at higher parameter counts, suggesting diminishing returns. Accuracy vs. FLOPS (Floating-point Operations):

Similar to the parameter trend, there's a positive correlation between the number of FLOPS and accuracy. Models with more complex computations (higher FLOPS) tend to be more accurate. Again, the relationship seems to level off or slightly decrease at high FLOP values. High-Level Conclusions:

Model Capacity and Accuracy:

The number of parameters and FLOPS generally reflect a model's capacity or complexity. Increasing capacity often leads to improved accuracy, but this relationship isn't linear and eventually saturates. Computational Cost vs. Accuracy:

Higher accuracy often comes at the cost of increased computational resources (more parameters and FLOPS). This trade-off is crucial in practical applications where computational efficiency is a concern. Model Selection:

The choice of model architecture depends on the desired balance between accuracy and computational cost. For resource-constrained environments, models with fewer parameters and lower FLOPS might be preferred, even if they have slightly lower accuracy. Caveats:

Generalization: The trends observed here might not hold universally across all models and problems. Other Factors: Factors beyond parameters and FLOPS, such as data quality, training techniques, and architecture design, also significantly influence model 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
```



Thu Jan 16 22:55:39 2025

```
+-----+
| NVIDIA-SMI 535.104.05                 Driver Version: 535.104.05   CUDA Version: 12.2   |
+-----+-----+-----+-----+
| GPU   Name           Persistence-M   Bus-Id        Disp.A   Volatile Uncorr. E  |
| Fan   Temp    Perf           Pwr:Usage/Cap |      Memory-Usage | GPU-Util  Compute  |
|                               |                      |              MIG    |
+-----+-----+-----+-----+
|   0   Tesla T4              Off      00000000:00:04:0 Off |             0%      Defau  |
| N/A   65C    P0              31W / 70W |  935MiB / 15360MiB |             N      |
+-----+-----+-----+-----+

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

Expected Memory Reduction:

By converting the models to half-precision (FP16), we expect a significant reduction in memory usage. This is because each weight and activation now occupies only half the memory compared to single-precision (FP32). Theoretically, we should see roughly a 50% reduction in memory consumption. Observation from nvidia-smi Output:

The nvidia-smi output shows the current GPU memory usage after converting the models to half-precision and moving them back to the GPU. We would need to compare this output to the memory usage before the conversion to determine the actual memory reduction. Factors Affecting Memory Utilization:

Model Architecture: Different models have varying memory requirements depending on their size and complexity. Batch Size: Larger batch sizes generally require more memory. Other Factors: Additional factors like intermediate activation storage and gradient buffers can also contribute to memory usage. To accurately assess the memory reduction:

Capture nvidia-smi output before conversion: Record the GPU memory usage before converting the models to half-precision. Compare outputs: Compare the memory usage before and after conversion to determine the actual reduction. Conclusion:

While we expect a significant memory reduction due to half-precision, the actual reduction will depend on various factors as mentioned above. Comparing the nvidia-smi output before and after conversion will provide the most accurate assessment of the memory savings achieved.

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 inputs 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(data_loader), desc="Processing batches", total=num_b

        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)

        # 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=1)

        # 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 [00:11<02:06, 1.01s/it]
 took 11.14828610420227s

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:

Using a lower-precision format can be a valuable technique for speeding up model inference, especially when dealing with large models or resource-constrained devices. However, it's crucial to carefully consider the potential accuracy trade-offs and ensure hardware and software compatibility before deploying lower precision models.

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
import matplotlib.pyplot as plt

# Accuracy for each model
accuracies = {
    "ResNet-18": 70.54,
    "ResNet-50": 77.35,
    "ResNet-152": 80.21,
    "MobileNetV2": 71.91,
}

# Parameters for each model (in millions)
params = {
    "ResNet-18": 11.7,
    "ResNet-50": 25.6,
    "ResNet-152": 60.2,
    "MobileNetV2": 3.5,
}

# FLOPS for each model (in billions)
flops = {
    "ResNet-18": 1.8,
    "ResNet-50": 4.1,
    "ResNet-152": 11.5,
    "MobileNetV2": 0.3,
}

# Create subplots
fig, axs = plt.subplots(1, 2, figsize=(15, 5))

# Plot Accuracy vs Parameters
axs[0].plot(list(params.values()), list(accuracies.values()), "bo")
for model, accuracy in accuracies.items():
```



```
    axs[0].annotate(model, (params[model], accuracy), textcoords="offset points", xytext=
axs[0].set_title("Accuracy vs Parameters")
axs[0].set_xlabel("Parameters (Millions)")
axs[0].set_ylabel("Accuracy (%)")
axs[0].grid(True)

# Plot Accuracy vs FLOPS
axs[1].plot(list(flops.values()), list(accuracies.values()), "go")
for model, accuracy in accuracies.items():
    axs[1].annotate(model, (flops[model], accuracy), textcoords="offset points", xytext=
axs[1].set_title("Accuracy vs FLOPS")
axs[1].set_xlabel("FLOPS (Billions)")
axs[1].set_ylabel("Accuracy (%)")
```