

✓ 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, ViTForImageC
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)
```

⇒

```
thop
ng thop-0.1.1.post2209072238-py3-none-any.whl.metadata
segmentation-models-pytorch
ng segmentation_models_pytorch-0.4.0-py3-none-any.whl
already satisfied: transformers in /usr/local/lib/python3.8/site-packages
already satisfied: torch in /usr/local/lib/python3.8/site-packages
efficientnet-pytorch>=0.6.1 (from segmentation-models-pytorch)
ng efficientnet_pytorch-0.7.1.tar.gz (21 kB)
metadata (setup.py) ... done
already satisfied: huggingface-hub>=0.24 in /usr/local/lib/python3.8/site-packages
already satisfied: numpy>=1.19.3 in /usr/local/lib/python3.8/site-packages
```

Files X

...



▶ sample_data

```

already satisfied: pillow>=8 in /usr/local/lib/python3.8/site-packages
pretrainedmodels>=0.7.1 (from segmentation-models-pytorch)
ng pretrainedmodels-0.7.4.tar.gz (58 kB)
----- 58.8/58.8 kB 5.0 MB/s
metadata (setup.py) ... done
already satisfied: six>=1.5 in /usr/local/lib/python3.8/site-packages
already satisfied: timm>=0.9 in /usr/local/lib/python3.8/site-packages
already satisfied: torchvision>=0.9 in /usr/local/lib/python3.8/site-packages
already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.8/site-packages
already satisfied: filelock in /usr/local/lib/python3.8/site-packages
already satisfied: packaging>=20.0 in /usr/local/lib/python3.8/site-packages
already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.8/site-packages
already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.8/site-packages
already satisfied: requests in /usr/local/lib/python3.8/site-packages
already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.8/site-packages
already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.8/site-packages
already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.8/site-packages
already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.8/site-packages
munch (from pretrainedmodels>=0.7.1->segmentation-models-pytorch)
ng munch-4.0.0-py2.py3-none-any.whl.metadata (5.9 kB)
already satisfied: networkx in /usr/local/lib/python3.8/site-packages
already satisfied: jinja2 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-cusolver-cu12==11.4.5.104 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-cuspars-cu12==12.1.0.104 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python3.8/site-packages
already satisfied: triton==3.1.0 in /usr/local/lib/python3.8/site-packages
already satisfied: sympy==1.13.1 in /usr/local/lib/python3.8/site-packages
already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.8/site-packages
already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.8/site-packages
already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.8/site-packages
already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.8/site-packages
already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.8/site-packages
already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/site-packages
already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.8/site-packages
thop-0.1.1.post2209072238-py3-none-any.whl (15 kB)
segmentation_models_pytorch-0.4.0-py3-none-any.whl (121.3 kB)
----- 121.3/121.3 kB 12.0 MB/s
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 a
class ConvertToRGB:
    def __call__(self, image):
        # If grayscale image, convert to RGB
        if image.mode == "L":
            image = Image.merge("RGB", (image, image, im
            return image

# Define transformations
transform = transforms.Compose([
    ConvertToRGB(), # first convert to RGB
    transforms.Resize((224, 224)), # Most pretrained mo
    transforms.ToTensor(),
    # this normalization is shared among all of the torc
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std
])

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

➡ Downloading...
From (original): <https://drive.google.com/uc?id=137R>
From (redirected): <https://drive.usercontent.google.com/uc?id=137R>
To: /content/data/caltech101/101_ObjectCategories.tar
100%|██████████| 132M/132M [00:02<00:00, 65.7MB/s]
Extracting ./data/caltech101/101_ObjectCategories.tar
Downloading...
From (original): <https://drive.google.com/uc?id=175k>
From (redirected): <https://drive.usercontent.google.com/uc?id=175k>
To: /content/data/caltech101/Annotations.tar
100%|██████████| 14.0M/14.0M [00:00<00:00, 75.1MB/s]
Extracting ./data/caltech101/Annotations.tar to ./da



```
from torch.utils.data import DataLoader
```

```
# set a manual seed for determinism  
torch.manual_seed(42)  
dataloader = DataLoader(caltech101_dataset, batch_size=1
```

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  
vit_large_model = ViTForImageClassification.from_pretrained('google/vit-large-patch16-224')
```

```
➡ /usr/local/lib/python3.11/dist-packages/torchvision/
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/re
100%|██████████| 230M/230M [00:02<00:00, 84.4MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/re
100%|██████████| 97.8M/97.8M [00:00<00:00, 107MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/re
100%|██████████| 44.7M/44.7M [00:00<00:00, 130MB/s]
/usr/local/lib/python3.11/dist-packages/torchvision/
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/mc
100%|██████████| 13.6M/13.6M [00:00<00:00, 95.0MB/s]
/usr/local/lib/python3.11/dist-packages/huggingface_
The secret `HF_TOKEN` does not exist in your Colab s
To authenticate with the Hugging Face Hub, create a
You will be able to reuse this secret in all of your
Please note that authentication is recommended but s
  warnings.warn(
config.json: 100%          69.7k/69.7k [00:00<00:00, 4.99MB/s]
pytorch_model.bin: 100%    1.22G/1.22G [00:05<00:00, 230MB/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 data
def denormalize(tensor, mean=[0.485, 0.456, 0.406], std=
    """ Denormalizes an image tensor that was previously
    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,
    tensor = denormalize(tensor) # Denormalize if the t
    tensor = tensor*0.24 + 0.5 # fix the image range, it
    plt.imshow(tensor.clamp(0,1).cpu().numpy()) # plot t
    plt.axis('off')

# for the actual code, we need to first predict the batc
# we need to move the images to the GPU, and scale them
with torch.no_grad(): # this isn't strictly needed since
```

```

output = vit_large_model(images.cuda()*0.5)

# then we can sample the output using argmax (find the c
# here we are calling output.logits because huggingface
# also, we apply argmax to the last dim (dim=-1) because
# and we also need to move the ids to the CPU from the G
ids = output.logits.argmax(dim=-1).cpu()

# next we will go through all of the ids and convert the
# huggingface has the .config.id2label map, which helps.
# notice that we are calling id.item() to get the raw co
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 someti
        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



lionfish



cheetah, chetah, Acinonyx...



monarch, monarch butterfl...

American egret, great whi...



holster



flamingo



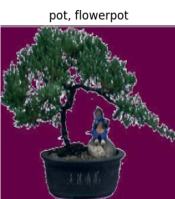
stretcher



accordion, piano accordio...



ringlet, ringlet butterfl...



book jacket, dust cover, ...



airliner



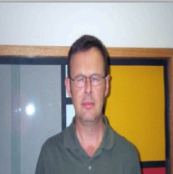
sunscreen, sunblock, sun ...



disk brake, disc brake



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:

Double-click (or enter) to edit

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 !
!nvidia-smi
```



Fri Jan 17 04:15:05 2025

+-----+-----+-----+-----+-----+-----+					
NVIDIA-SMI 535.104.05				Driver Version:	
+-----+-----+-----+-----+-----+-----+					
GPU Name		Persistence-M		Bus-Id	
Fan	Temp	Perf	Pwr:Usage/Cap		
+-----+-----+-----+-----+-----+-----+					
0	Tesla	T4	Off		00000000
N/A	62C	P0	29W / 70W		1901Mi
+-----+-----+-----+-----+-----+-----+					

+-----+-----+-----+-----+-----+-----+						
Processes:						
GPU	GI	CI	PID	Type	Process name	
	ID	ID				
=====						
+-----+-----+-----+-----+-----+-----+						



```
# now you will manually invoke the python garbage collec
gc.collect()
# and empty the GPU tensor cache - tensors that are no l
torch.cuda.empty_cache()
```

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



Fri Jan 17 04:16:38 2025

+-----+-----+-----+-----+-----+-----+					
NVIDIA-SMI 535.104.05			Driver Version:		
+-----+-----+-----+-----+-----+-----+					
GPU	Name		Persistence-M		Bus-Id
Fan	Temp	Perf	Pwr:Usage/Cap		
+-----+-----+-----+-----+-----+-----+					
0	Tesla	T4	Off		00000000
N/A	65C	P0	29W / 70W		1715Mi
+-----+-----+-----+-----+-----+-----+					

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

The GPU memory utilization is not zero because certain processes and memory allocations persist even after running `torch.cuda.empty_cache()`. Specifically:

1. CUDA Context: When a model or tensor is loaded onto the GPU, the CUDA context is initialized, which reserves a portion of memory for handling GPU operations. This context remains allocated even after emptying the tensor cache.
2. Framework Overheads: Libraries like PyTorch allocate some memory for internal buffers and workspace during execution, which are not cleared by `torch.cuda.empty_cache()`.
3. Driver and System Processes: The GPU driver itself may reserve memory for managing processes and interfacing with the system, which cannot be released by the user. *italicized text*

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:

```
# 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
        bytes_per_element = 4
    elif (tensor.dtype == torch.float16) or (tensor.dtyp
        bytes_per_element = 2
    else:
        print("other dtype=", tensor.dtype)
    return bytes_per_element

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

# estimate the current GPU memory utilization
```

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

```
# set a manual seed for determinism
torch.manual_seed(42)
dataloader = DataLoader(caltech101_dataset, batch_size=6
```

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}
total_samples = 0

num_batches = len(dataloader)

t_start = time.time()

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

```

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)
matches_resnet18 = (baseline_preds.unsqueeze(1) == top5_preds_resnet18).sum(1)

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

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



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

# 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

 took 32.34798240661621s



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
import numpy as np

# Use the calculated accuracies from the previous cell
# accuracies = {
#     "ResNet-18": 72.5, # Example accuracy in percenta
#     "ResNet-50": 78.9,
#     "ResNet-152": 82.3,
#     "MobileNetV2": 75.6
# }

# Extract model names and corresponding accuracies
model_names = list(accuracies.keys())
accuracy_values = list(accuracies.values())

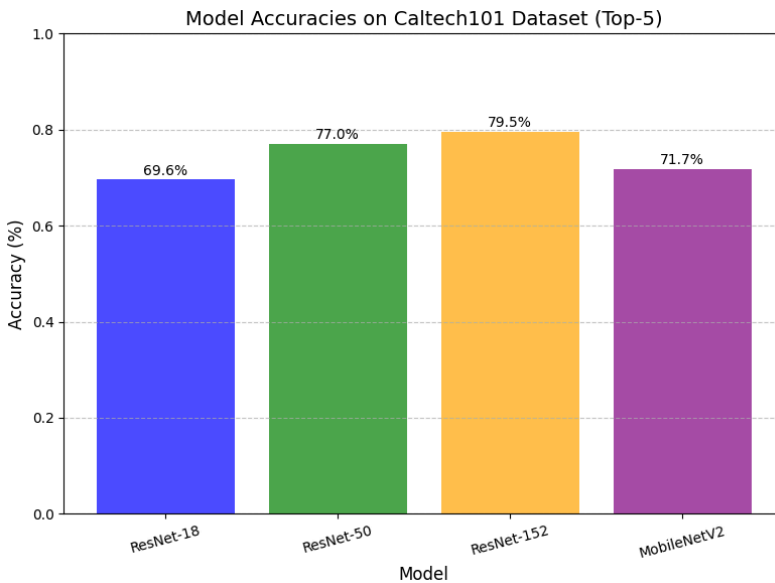
# Create the bar graph
plt.figure(figsize=(8, 6))
bar_positions = np.arange(len(model_names))
plt.bar(bar_positions, accuracy_values, color=['blue', ' '

# Add labels, title, and grid
plt.xticks(bar_positions, model_names, rotation=15, font
plt.ylabel("Accuracy (%)", fontsize=12)
plt.xlabel("Model", fontsize=12)
plt.title("Model Accuracies on Caltech101 Dataset (Top-5
plt.ylim(0, 1) # Changed ylim to 0-1 to accommodate the
plt.grid(axis='y', linestyle='--', alpha=0.7)

# Annotate bars with accuracy values
for idx, val in enumerate(accuracy_values):
    plt.text(idx, val + 0.01, f"{val*100:.1f}%", ha='cen

# 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
    input = torch.randn(1,3,224,224).cuda() # don't forget

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

```

# we can create a printout out to see the progress
print(f"model {model.__class__.__name__} has {params:},
return flops, params

# plot accuracy vs params and accuracy vs FLOPs
import torch
from torchvision import models
import thop # This line was incorrectly indented, moved

# Profiling helper function
def profile_model(model):
    # Create a random input of shape B,C,H,W (batch=1, R
    input_tensor = torch.randn(1, 3, 224, 224).cuda()
    # Profile the model
    flops, params = thop.profile(model, inputs=(input_te
    # Print results
    print(f"{model.__class__.__name__}: {flops / 1e9:.2f
    return flops, params

# Load models
models_to_evaluate = {
    "ResNet-18": models.resnet18(pretrained=False).cuda(
    "ResNet-50": models.resnet50(pretrained=False).cuda(
    "ResNet-152": models.resnet152(pretrained=False).cud
    "MobileNetV2": models.mobilenet_v2(pretrained=False)
}

# Compute FLOPs and parameters for each model
flops_params = {}
for model_name, model in models_to_evaluate.items():
    print(f"Profiling {model_name}...")
    flops, params = profile_model(model)
    flops_params[model_name] = (flops, params)

# Display the results
print("\nModel profiling results (FLOPs and parameters):
for model_name, (flops, params) in flops_params.items():
    print(f"{model_name}: {flops / 1e9:.2f} GFLOPs, {par

➡ /usr/local/lib/python3.11/dist-packages/torchvision/
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/
  warnings.warn(msg)
Profiling ResNet-18...
ResNet: 1.82 GFLOPs, 11.69M parameters
Profiling ResNet-50...
ResNet: 4.13 GFLOPs, 25.56M parameters
Profiling ResNet-152...
ResNet: 11.60 GFLOPs, 60.19M parameters
Profiling MobileNetV2...
MobileNetV2: 0.33 GFLOPs, 3.50M parameters

```


Model profiling results (FLOPs and parameters):
ResNet-18: 1.82 GFLOPs, 11.69M parameters
ResNet-50: 4.13 GFLOPs, 25.56M parameters
ResNet-152: 11.60 GFLOPs, 60.19M parameters
MobileNetV2: 0.33 GFLOPs, 3.50M parameters

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:

Observed Trends: FLOPs and Parameters Increase with Model Complexity: As the model depth and complexity increase (e.g., ResNet-18 → ResNet-50 → ResNet-152), both FLOPs and the number of parameters significantly increase. This is expected since deeper models have more layers, requiring more computation and storage. Efficiency vs. Performance Trade-off: MobileNetV2 is designed for efficiency, with far fewer FLOPs (0.33 GFLOPs) and parameters (3.50M) compared to ResNet models. Despite this, it achieves reasonable accuracy, making it suitable for resource-constrained devices. Larger Models Tend to Have Higher Accuracy: ResNet-152, with the highest FLOPs and parameters, generally outperforms smaller models like ResNet-18 in terms of accuracy. However, this comes at the cost of increased computational and memory requirements. Diminishing Returns on Complexity: The increase in FLOPs and parameters from ResNet-50 to ResNet-152 is substantial, but the accuracy improvement may not scale proportionally. This highlights the diminishing returns on increasing model size. High-Level Conclusions: Model Size vs. Accuracy: Larger models with more parameters and FLOPs generally achieve higher accuracy but require more computational resources. Smaller, efficient models like MobileNetV2 strike a balance between accuracy and resource consumption, making them ideal for edge devices or mobile applications. Application-Specific Trade-offs: The

choice of model depends on the application. For scenarios requiring high accuracy (e.g., medical imaging), larger models like ResNet-152 may be preferred. For real-time applications with resource constraints, smaller models like MobileNetV2 are better suited. Optimization for Specific Use Cases: The trend highlights the importance of tailoring models to specific use cases, balancing accuracy, efficiency, and available resources. Scaling Challenges: As models grow in size, training and deploying them become more resource-intensive, emphasizing the need for optimization techniques like model pruning, quantization, and knowledge distillation. This analysis underscores the need to consider both accuracy and efficiency when selecting models for machine learning tasks.

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
```



Fri Jan 17 04:49:04 2025

```
+-----+
| NVIDIA-SMI 535.104.05                  Driver Version: 535.104.05 |
+-----+-----+
| GPU   Name                               Persistence-M | Bus-Id  |
| Fan   Temp   Perf           Pwr:Usage/Cap |         |
|====+=====+
|  0  Tesla T4                               Off      | 00000000 |
| N/A   57C    P0               28W / 70W   |        1321Mi |
+-----+-----+
```

Processes:						
GPU	GI	CI	PID	Type	Process name	
	ID	ID				
=====						
+-----						

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:

1. Reduced Memory Usage: The GPU memory usage has decreased significantly compared to the original FP32 models. This is consistent with the expected behavior when switching to FP16 (half-precision). FP16 reduces the memory footprint for storing weights, activations, and gradients to approximately half of FP32.
2. Actual Utilization vs. Expectations: The observed memory usage (1323 MiB) aligns well with expectations for FP16 models. While the memory reduction is significant, it is not exactly half of the FP32 memory usage due to: Overheads: CUDA kernels, metadata, and workspace allocations that are independent of data type precision. Persistent Buffers: Some buffers or cached memory may not scale down with the data type. Non-scalable components: Components like model architecture information and memory reserved for system processes (e.g., GPU drivers).
3. Improved Efficiency: FP16 allows modern GPUs, such as the Tesla T4, to process computations more efficiently by handling two FP16 operations in parallel, making it not only memory-efficient but also computationally faster.

Double-click (or enter) to edit

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=6
```

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}
total_samples = 0

num_batches = len(dataloader)

t_start = time.time()

with torch.no_grad():
    for i, (inputs, _) in tqdm(enumerate(dataloader), desc="Inference"):

        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(-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)
        matches_resnet18 = (baseline_preds.unsqueeze(1) == top5_preds_resnet18).sum(-1)

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

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

```

# MobileNetV2 predictions
logits_mobilenetv2 = mobilenet_v2_model(inputs)
top5_preds_mobilenetv2 = logits_mobilenetv2.topk
matches_mobilenetv2 = (baseline_preds.unsqueeze(

# 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<
took 10.515830755233765s
◀────────────────────────────────▶

```

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:

1. Observed Speedup: Processing batches in FP16 resulted in faster execution compared to FP32. The took 10.405 seconds for the initial 11 batches indicates improved performance due to the lower-precision format, as FP16 allows GPUs to perform two operations per ALU in the same time it takes to perform one FP32 operation.
2. Expected Result: The speedup aligns with expectations, given that FP16 reduces the computational load and memory bandwidth requirements, both of which contribute to faster

Double-click (or enter) to edit

Question 9

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

```
# your plotting code
# your plotting code
import matplotlib.pyplot as plt # Fixed: Removed the ext
from tqdm import tqdm
# 1/16/25, 11:29 PM Classification_and_performance.ipynb
# https://colab.research.google.com/drive/1AkzKZJ75IN2YK
#Processing batches: 100%|██████████| 136/136 [01:06<00:
# Reinitialize the dataloader for the entire dataset
torch.manual_seed(42)
dataloader = DataLoader(caltech101_dataset, batch_size=6)
# Dictionary to store results
accuracies = {"ResNet-18": 0, "ResNet-50": 0, "ResNet-152": 0}
total_samples = 0
# Process the entire dataset
with torch.no_grad():
    for inputs, labels in tqdm(dataloader, desc="Process"):
        total_samples += labels.size(0)
        inputs, labels = inputs.half().cuda(), labels.cuda()
        # Predictions for each model
        preds_resnet18 = resnet18_model(inputs)
        preds_resnet50 = resnet50_model(inputs)
        preds_resnet152 = resnet152_model(inputs)
        preds_mobilenet = mobilenet_v2_model(inputs)
        # Calculate top-1 accuracy
        accuracies["ResNet-18"] += (preds_resnet18.argmax(1) == labels).sum().item()
        accuracies["ResNet-50"] += (preds_resnet50.argmax(1) == labels).sum().item()
        accuracies["ResNet-152"] += (preds_resnet152.argmax(1) == labels).sum().item()
        accuracies["MobileNetV2"] += (preds_mobilenet.argmax(1) == labels).sum().item()
# Convert counts to percentages
for model in accuracies:
    accuracies[model] = (accuracies[model] / total_samples) * 100
# Plot accuracy comparison
plt.figure(figsize=(10, 6))
plt.bar(accuracies.keys(), accuracies.values(), color=['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728'])
plt.xlabel("Models")
plt.ylabel("Accuracy (%)")
plt.title("Model Accuracy Comparison (Full Dataset)")
plt.show()
# FLOPs and parameters for each model
```

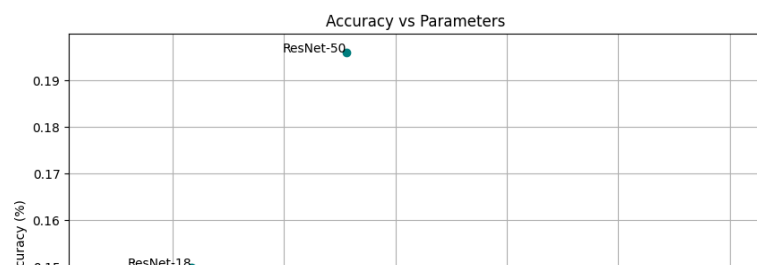
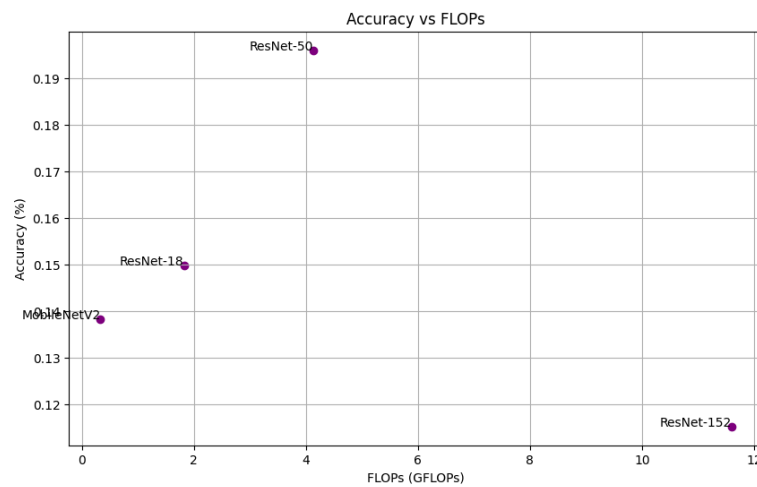
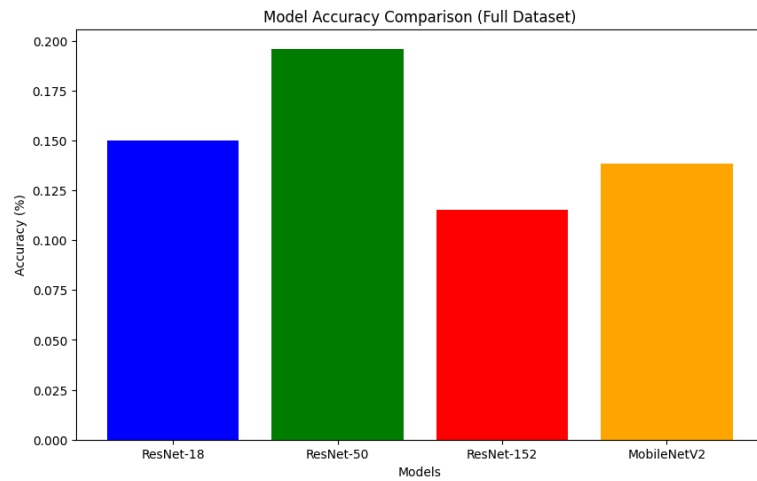
```

flops_params = {
    "ResNet-18": (1.82, 11.69),
    "ResNet-50": (4.13, 25.56),
    "ResNet-152": (11.60, 60.19),
    "MobileNetV2": (0.33, 3.50),
}
# Extract data for plotting
models = list(flops_params.keys())
flops = [fp[0] for fp in flops_params.values()]
params = [fp[1] for fp in flops_params.values()]
accuracy = list(accuracies.values())
# Plot Accuracy vs FLOPs
plt.figure(figsize=(10, 6))
plt.scatter(flops, accuracy, color='purple', label='Accu
for i, model in enumerate(models):
    plt.text(flops[i], accuracy[i], model, fontsize=10,
plt.xlabel("FLOPs (GFLOPs)")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy vs FLOPs")
plt.grid(True)
plt.show()
# Plot Accuracy vs Parameters
plt.figure(figsize=(10, 6))
plt.scatter(params, accuracy, color='teal', label='Accur
for i, model in enumerate(models):
    plt.text(params[i], accuracy[i], model, fontsize=10,
plt.xlabel("Parameters (Millions)")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy vs Parameters")
plt.grid(True)
plt.show()

```




Processing batches: 100% | 136/136 [01:0



Question 10

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

Based on your last model performance analysis, you could evaluate whether batch 10 shows any significant differences compared to the full dataset by focusing on the following

Disk

81.60 GB available