Classification and Performance

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

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

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

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

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

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 shapes
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)
```

```
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')
```

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 images better
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 normalization to the other models
with torch.no_grad(): # this isn't strictly needed since we already disabled autograd, but we should do it for good measure
  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 shape is B,C
# 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 likely performs decently but might exhibit limitations depending on the dataset size, quality, and balance. If class-wise performance discrepancies exist, it could be attributed more to the training set (e.g., insufficient or unbalanced data) rather than the model's size or complexity, assuming the model architecture is appropriate for the task.

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 to the shell rather than python
!nvidia-smi
```

```
# 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
```

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:

GPU memory utilization is not zero because even after calling `empty_cache()`, memory allocations related to CUDA contexts, kernel caching, or reserved memory blocks for PyTorch operations persist. These allocations ensure efficient execution of subsequent GPU tasks and avoid overhead from repeatedly initializing contexts. Additionally, if models or tensors are still in scope, they occupy memory until explicitly deleted or the program terminates.

The current utilization matches expectations, as calling `empty_cache()` only releases unreferenced memory to the GPU, but it does not reset memory usage to zero due to PyTorch's memory management design. This behavior is crucial to optimize GPU performance but requires users to manage memory carefully to prevent out-of-memory errors during operations.

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 precision float)
        bytes_per_element = 4
    elif (tensor.dtype == torch.float16) or (tensor.dtype == torch.half):    # float16 (half precision float)
        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_gpu_memory_utilization(model, input_tensor):
    # Get the size of the model parameters
    total_params = count_parameters(model)
    param_size = total_params * sizeof_tensor(next(model.parameters()))

    # Get the size of the input tensor
    input_size = input_tensor.numel() * sizeof_tensor(input_tensor)

    # Estimate memory utilization
    # Assumes that the intermediate outputs and gradients use memory equal to 2x input size
    intermediate_and_gradients = 2 * input_size

    # Add overhead (CUDA kernels, buffers, etc.) - approximate
    overhead = 200 * 1024 ** 2  # ~200 MB

    # Total estimated memory utilization
    total_memory_utilization = param_size + input_size + intermediate_and_gradients + overhead

    return total_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=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().sum().item()

        # 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().sum().item()

        # 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().sum().item()

        # 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).float().sum().item()

        # 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
```

Question 4

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

```
# your plotting code
models = list(accuracies.keys())
accuracy_values = list(accuracies.values())

# Plotting the bar graph
plt.figure(figsize=(10, 6))
plt.bar(models, accuracy_values, color=["blue", "green", "red", "purple"])
```

```
plt.xlabel("Models", fontsize=14)
plt.ylabel("Accuracy", fontsize=14)
plt.title("Top-5 Accuracy for Different Models", fontsize=16)
plt.ylim(0, 1)  # Accuracies are typically between 0 and 1
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.show()
```

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

Question 5

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

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

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

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

# plot accuracy vs params and acuracy vs FLOPs
# Define the models (replace these placeholders with actual models)
models = {
    "ResNet-18": resnet18_model,
    "ResNet-50": resnet50_model,
    "ResNet-152": resnet152_model,
    "MobileNetV2": mobilenet_v2_model
}

# Dictionary to store FLOPs and parameters
model_metrics = {"Model": [], "FLOPs": [], "Parameters": [], "Accuracy": []}

# Profiling each model
for model_name, model in models.items():
    model.cuda()
    flops, params = profile(model, inputs=(torch.randn(1, 3, 224, 224).cuda(),), verbose=False)
    model_metrics["Model"].append(model_name)
    model_metrics["FLOPs"].append(flops)
    model_metrics["Parameters"].append(params)
    model_metrics["Accuracy"].append(accuracies[model_name])  # Use calculated accuracies

# Plot Accuracy vs Parameters
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(model_metrics["Parameters"], model_metrics["Accuracy"], color="blue", label="Models")
for i, model_name in enumerate(model_metrics["Model"]):
    plt.text(model_metrics["Parameters"][i], model_metrics["Accuracy"][i], model_name, fontsize=10)
plt.xlabel("Parameters (log scale)", fontsize=14)
plt.xscale("log")
plt.ylabel("Accuracy", fontsize=14)
plt.title("Accuracy vs Parameters", fontsize=16)
plt.grid(True, linestyle="--", alpha=0.7)

# Plot Accuracy vs FLOPs
plt.subplot(1, 2, 2)
plt.scatter(model_metrics["FLOPs"], model_metrics["Accuracy"], color="red", label="Models")
for i, model_name in enumerate(model_metrics["Model"]):
    plt.text(model_metrics["FLOPs"][i], model_metrics["Accuracy"][i], model_name, fontsize=10)
plt.xlabel("FLOPs (log scale)", fontsize=14)
plt.xscale("log")
plt.ylabel("Accuracy", fontsize=14)
plt.title("Accuracy vs FLOPs", fontsize=16)
plt.grid(True, linestyle="--", alpha=0.7)

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:

The observed trends suggest that models with higher FLOPs and parameter counts generally achieve higher accuracy, as they have greater capacity to learn complex patterns in data. However, the improvements in accuracy tend to diminish as model complexity increases, indicating diminishing returns. This highlights the importance of balancing model complexity and computational efficiency, especially in resource-constrained environments. Furthermore, the trend implies that while large models are powerful, they may not always be the most practical choice for real-world applications. Instead, selecting a model that optimally balances accuracy and efficiency is crucial for scalability and usability in machine learning tasks.

Performance and Precision

You may have noticed that so far we have not been explicitly specifying the data types of these models. We can do this because torch will default to using float32 (32-bit single-precision). However, this is not always necessary nor desirable. There are currently a large number of alternative formats (with fewer bits per value), many of which are custom to specific accelerators. We will eventually cover these later in the course, but for now we can consider the second most common type on the GPU: FP16 (half-precision floating-point).

As the name suggests, FP16 only uses 16 bits per value rather than 32. GPUs are specifically designed to handle this datatype and all of the newer ones can execute either one FP32 or two FP16 operations per ALU.

Here's an overview of different precision types: https://moocaholic.medium.com/fp64-fp32-fp16-bfloat16-tf32-and-other-members-of-the-zoo-a1ca7897d407

Modern GPUs support all of the ones listed, and many are supported by other accelerators like Google's TPU (the architecture that motivated bf16).

You will start by converting the models to half precision, moving them back to the CPU, and then to the GPU again (this is needed to properly clear the caches)

```
# convert the models to half
resnet152_model = resnet152_model.half()
resnet50_model = resnet50_model.half()
resnet18_model = resnet18_model.half()
mobilenet_v2_model = mobilenet_v2_model.half()
vit_large_model = vit_large_model.half()

# move them to the CPU
resnet152_model = resnet152_model.cpu()
resnet50_model = resnet50_model.cpu()
resnet18_model = resnet18_model.cpu()
mobilenet_v2_model = mobilenet_v2_model.cpu()
vit_large_model = vit_large_model.cpu()

# clean up the torch and CUDA state
gc.collect()
torch.cuda.empty_cache()

# move them back to the GPU
resnet152_model = resnet152_model.cuda()
resnet50_model = resnet50_model.cuda()
resnet18_model = resnet18_model.cuda()
mobilenet_v2_model = mobilenet_v2_model.cuda()
vit_large_model = vit_large_model.cuda()


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

Question 7

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

When the models are switched to half-precision, the memory utilization should decrease significantly compared to full-precision (float32). This is because half-precision (float16) requires only 2 bytes per element, whereas full-precision requires 4 bytes. The utilization should roughly halve, provided the only change is the data type.

However, the actual utilization might differ slightly from the expected calculation due to additional factors such as:

- Overhead from memory allocation and CUDA kernels.
- Buffers and metadata managed internally by the framework (e.g., PyTorch).
- Potential padding or alignment requirements for efficient memory access.

Overall, the observed memory utilization should align closely with the expected values, considering these minor discrepancies. This reduction in memory usage highlights the advantage of using mixed or half-precision, especially for large models, as it allows for more efficient utilization of GPU resources without a significant loss in accuracy.

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().sum().item()

        # 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().sum().item()

        # 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().sum().item()

        # 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).float().sum().item()

        # 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
```

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 typically results in a significant speedup, as observed. This is due to the reduced memory usage and fewer computational resources required for processing lower-precision data types, such as float16 or bfloat16. These formats help accelerate operations and improve efficiency, particularly in deep learning tasks, where large datasets and models are common. The pros of using lower precision include faster execution, reduced memory usage, and increased energy efficiency, making it ideal for large-scale computations or hardware with limited resources. However, the downside is the potential loss of accuracy, which may lead to numerical errors or instability during training. This can particularly impact models or tasks that require high precision or numerical stability. Additionally, not all hardware supports efficient lower-precision operations, which could necessitate additional optimizations. Therefore, while lower-precision formats offer notable benefits in terms of performance, they come with trade-offs that need to be carefully considered based on the specific use case.

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 time
import torch
import matplotlib.pyplot as plt
import gc
from tqdm import tqdm
import thop  # Make sure thop is installed for profiling

# Helper functions
def sizeof_tensor(tensor):
    # Get the size of the data type
    if (tensor.dtype == torch.float32) or (tensor.dtype == torch.float):      # float32 (single precision float)
        bytes_per_element = 4
    elif (tensor.dtype == torch.float16) or (tensor.dtype == torch.half):    # float16 (half precision float)
        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_gpu_memory_utilization(model, input_tensor):
    # Get the size of the model parameters
    total_params = count_parameters(model)
    param_size = total_params * sizeof_tensor(next(model.parameters()))

    # Get the size of the input tensor
    input_size = input_tensor.numel() * sizeof_tensor(input_tensor)

    # Estimate memory utilization
    intermediate_and_gradients = 2 * input_size
    overhead = 200 * 1024 ** 2  # ~200 MB

    # Total estimated memory utilization
    total_memory_utilization = param_size + input_size + intermediate_and_gradients + overhead

    return total_memory_utilization

# Profiling helper function
def profile(model):
    # create a random input of shape B,C,H,W - batch=1 for 1 image, C=3 for RGB, H and W = 224 for the expected images size
    input = torch.randn(1, 3, 224, 224).cuda()  # Move input to GPU

    # Profile the model
```

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

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

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

    # Number of batches (without batch 10 early-exit)
    num_batches = len(dataloader)

    t_start = time.time()

    # Start processing without early exit for batch 10
    with torch.no_grad():
        for i, (inputs, _) in tqdm(enumerate(dataloader), desc="Processing batches", total=num_batches):

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

            # Get top prediction from vit_large model
            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().sum().item()

            # 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().sum().item()

            # 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().sum().item()

            # 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).float().sum().item()

            # 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)

    # Calculate total time
    print(f"Processing 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

    # Plot the bar graph for accuracy
    models = list(accuracies.keys())
    accuracy_values = list(accuracies.values())

    # Plotting the bar graph
    plt.figure(figsize=(10, 6))
    plt.bar(models, accuracy_values, color=["blue", "green", "red", "purple"])
    plt.xlabel("Models", fontsize=14)
    plt.ylabel("Accuracy", fontsize=14)
    plt.title("Top-5 Accuracy for Different Models", fontsize=16)
    plt.ylim(0, 1)  # Accuracies are typically between 0 and 1
    plt.grid(axis="y", linestyle="--", alpha=0.7)
    plt.show()

    # Profiling each model for FLOPs and Parameters
```

```
models = {
    "ResNet-18": resnet18_model,
    "ResNet-50": resnet50_model,
    "ResNet-152": resnet152_model,
    "MobileNetV2": mobilenet_v2_model
}

# Dictionary to store FLOPs, Parameters, and Accuracy
model_metrics = {"Model": [], "FLOPs": [], "Parameters": [], "Accuracy": []}

# Profiling each model
for model_name, model in models.items():
    model.cuda()
    flops, params = profile(model)
    model_metrics["Model"].append(model_name)
    model_metrics["FLOPs"].append(flops)
    model_metrics["Parameters"].append(params)
    model_metrics["Accuracy"].append(accuracies[model_name])  # Use the calculated accuracies

# Plot Accuracy vs Parameters
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(model_metrics["Parameters"], model_metrics["Accuracy"], color="blue", label="Models")
for i, model_name in enumerate(model_metrics["Model"]):
    plt.text(model_metrics["Parameters"][i], model_metrics["Accuracy"][i], model_name, fontsize=10)
plt.xlabel("Parameters (log scale)", fontsize=14)
plt.xscale("log")
plt.ylabel("Accuracy", fontsize=14)
plt.title("Accuracy vs Parameters", fontsize=16)
plt.grid(True, linestyle="--", alpha=0.7)

# Plot Accuracy vs FLOPs
plt.subplot(1, 2, 2)
plt.scatter(model_metrics["FLOPs"], model_metrics["Accuracy"], color="red", label="Models")
for i, model_name in enumerate(model_metrics["Model"]):
    plt.text(model_metrics["FLOPs"][i], model_metrics["Accuracy"][i], model_name, fontsize=10)
plt.xlabel("FLOPs (log scale)", fontsize=14)
plt.xscale("log")
plt.ylabel("Accuracy", fontsize=14)
plt.title("Accuracy vs FLOPs", fontsize=16)
plt.grid(True, linestyle="--", alpha=0.7)

plt.tight_layout()
plt.show()

# Convert models to half precision
resnet152_model = resnet152_model.half()
resnet50_model = resnet50_model.half()
resnet18_model = resnet18_model.half()
mobilenet_v2_model = mobilenet_v2_model.half()
vit_large_model = vit_large_model.half()

# Move models to CPU for cleanup
resnet152_model = resnet152_model.cpu()
resnet50_model = resnet50_model.cpu()
resnet18_model = resnet18_model.cpu()
mobilenet_v2_model = mobilenet_v2_model.cpu()
vit_large_model = vit_large_model.cpu()

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

# Move models back to 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()
```

Question 10

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

When comparing the full dataset to the batch 10 subset, several key differences can be observed. The accuracy reported for the full dataset is likely to be a more reliable measure of the model's true performance since it evaluates the model on a diverse set of samples, ensuring a comprehensive understanding of its generalization ability. On the other hand, the batch 10 subset may show higher or lower accuracy depending on the representativeness of the samples within that batch. If batch 10 includes easier or more typical data points, it could result in artificially higher accuracy. Additionally, the full dataset provides better insights into the model's ability to generalize across varied data, while the batch 10 subset might cause the model to overfit or underfit based on the limited and potentially unrepresentative data it contains. From a computational perspective, evaluating the full dataset requires more resources but offers a thorough performance assessment, whereas batch 10 is computationally cheaper but less reliable in reflecting the model's real-world effectiveness. In terms of loss, evaluating on the full dataset gives a clearer picture of how the model is converging, whereas the loss observed from batch 10 could be misleading, especially if the batch does not adequately represent the entire dataset. Ultimately, the full dataset provides a more robust and accurate evaluation of the model's performance, while the batch 10 subset may only reveal partial insights.

Start coding or generate with AI.