

Memory considerations when training Neural Networks on GPUs

In this homework, we will train a ResNet model on CIFAR-10 using PyTorch and explore its implications on GPU memory.

We will explore various systems considerations, such as the effect of batch size on memory usage, the effect of different optimizers (SGD, SGD with momentum, Adam), and we will try to minimize the memory usage of training our model by applying gradient accumulation.

Setup the environment

If you're running on colab - make sure you are using a GPU runtime. You can select a GPU runtime by clicking on `Runtime -> Change Runtime Type`.

💡 Hint - if you hit your colab GPU usage limit, try again in a few hours.

```
In [ ]: #@title Mount your Google Drive

import os
from google.colab import drive

try:
    drive.mount('/content/gdrive')

    DRIVE_PATH = '/content/gdrive/My\ Drive/cs182hw4_sp23'
    DRIVE_PYTHON_PATH = DRIVE_PATH.replace('\\', '/')
    if not os.path.exists(DRIVE_PYTHON_PATH):
        %mkdir $DRIVE_PATH

    ## the space in `My Drive` causes some issues,
    ## make a symlink to avoid this
    SYM_PATH = '/content/cs182hw4'
    if os.path.isdir(SYM_PATH):
        raise Exception(f"Path already exists - please delete {SYM_PATH} before mounting")
    else:
        !ln -sf $DRIVE_PATH $SYM_PATH
except Exception as e:
    print(e)
    print("WARNING - Unable to mount google drive for storing logs. Storing logs in the local directory")
    os.makedirs('/content/cs182hw4', exist_ok=True)
```

```
In [ ]: #@title Install dependencies
```

```
!pip install gputil
```

```
In [1]: import gc
import GPUtil
import os
import subprocess
import torch
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import time

ROOT_PATH = '/content/cs182hw4/'

# Define the CSV format for logging memory usage. Used later in this notebook.
MEMORY_LOG_FMT = ['timestamp', 'memUsage']
TRAIN_LOG_FMT = ['timestamp', 'epoch', 'memUsage', 'loss', 'accuracy']

if torch.cuda.is_available():
    print("Using GPU.")
    device = torch.device("cuda:0")
else:
    print("!!! WARNING !!! - Could not find a GPU - please use a GPU for this homework")
    device = torch.device("cpu")

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

Using GPU.

Define helper functions and download CIFAR-10 dataset

```
In [2]: seed = 42
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)

def get_allocated_memory_str():
    return "Allocated memory: {:.2f} GB".format(torch.cuda.memory_allocated(device))

def run_nvidia_smi():
    if torch.cuda.is_available():
        print(subprocess.check_output("nvidia-smi", shell=True).decode("utf-8"))
    else:
        print("Running on CPU")

def get_gpu_memory_usage() -> float:
    # Use GPUtil python library to get GPU memory usage
    if torch.cuda.is_available():
        return GPUtil.getGPUs()[0].memoryUsed
    else:
        return 0

def cleanup_memory():
    gc.collect()
    torch.cuda.empty_cache()

# Define transformations for the input data. We resize the 32x32 inputs to
# 224x224 which is the input shape for the ResNet family of models.
transform = transforms.Compose([
    transforms.Resize((224, 224)), # Resize to 224x224 for ResNet models
    transforms.ToTensor()
])

data_train = torchvision.datasets.CIFAR10(root='./data', train=True, download=True)
data_test = torchvision.datasets.CIFAR10(root='./data', train=False, download=True)

# We randomly subsample the dataset here to train our models faster for this notebook
SUBSAMPLE_SIZE = 1024*4
random_sample_idxs = torch.randint(len(data_train), (SUBSAMPLE_SIZE,))
subsampled_train_data = torch.utils.data.Subset(data_train, indices=random_sample_idxs)
```

Files already downloaded and verified

Files already downloaded and verified

1. Managing GPU memory when training deep models

One of the most common bottlenecks you will run into when training your deep learning models is the amount of GPU memory available to you. The exact memory usage of your training process depends on the specific model architecture and the size of the input data. The main components taking up GPU memory during training are:

- **Model Parameters:** The weights and biases of the model are stored in GPU memory during training. The number of parameters in a deep learning model can range from a few thousand to millions or even billions, depending on the model architecture and the size of the input data.
- **Activations:** The activations of each layer of the model are stored in GPU memory during the forward pass. The size of the activations can depend on the batch size and the number of hidden units in each layer. As the batch size increases, so does the size of the activations, which can quickly consume a large amount of GPU memory.
- **Gradients:** During the backward pass, the gradients of each layer with respect to the loss function are computed and stored in GPU memory. The size of the gradients can depend on the batch size and the number of hidden units in each layer. Like activations, larger batch sizes can lead to larger gradients and increased memory usage.
- **Input Data:** The input data, such as images or text, can also take up GPU memory during training. The size of the input data can depend on the input shape and the batch size.
- **Optimizer State:** The state of the optimizer, such as the momentum or running average of gradients, is stored in GPU memory during training. The size of the optimizer state can depend on the optimizer algorithm and the size of the model parameters.

Let's analyze the ResNet-152 model and CIFAR-10 input sizes

We can count the number of parameters in the model by loading it and inspecting it. Once we

```
In [3]: def analyze_model_and_inputs(model):
        print("Train data size: {}".format(len(data_train)))
        print("Test data size: {}".format(len(data_test)))

        # Fetch an example image to get image size
        image, label = data_train[0]
        print("Image input size: {}".format(image.size()))

        # Get model parameter count
        print("Model parameters: {}".format(sum(p.numel() for p in model.parameters() if p.requires_grad)))

        # Get model size in MB
        print("Model size estimate (MB): {}".format(sum(p.numel() * p.element_size() for p in model.parameters() if p.requires_grad) / 1024 / 1024))
```

```
In [4]: model = torchvision.models.resnet152(weights=None, num_classes=10)
        model.to(device) # Load the model into GPU memory
        analyze_model_and_inputs(model)
```

```
Train data size: 50000
Test data size: 10000
Image input size: torch.Size([3, 224, 224])
Model parameters: 58164298
Model size estimate (MB): 232.657192
```

Let's get to know our GPU better

Now that we have loaded the model onto the GPU, we will now use the `nvidia-smi` utility to measure the GPU memory utilization.

```
In [5]: !nvidia-smi
```

Tue Sep 19 20:43:21 2023

NVIDIA-SMI 517.00				Driver Version: 517.00				CUDA Version: 11.7			
GPU Name		TCC/WDDM		Bus-Id		Disp.A		Volatile Uncorr. ECC			
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage		GPU-Util		Compute M. MIG M.			
0	NVIDIA GeForce ...	WDDM	00000000:01:00.0		Off	N/A					
N/A	40C	P8	7W / N/A	1549MiB / 6144MiB		0%		Default N/A			
Processes:											
GPU	GI	CI	PID	Type	Process name			GPU Memory Usage			
	ID	ID									
0	N/A	N/A	15764	C	...p\envs\malning\python.exe			N/A			
0	N/A	N/A	19064	C	...p\envs\malning\python.exe			N/A			

Note that the actual memory usage on the GPU is anywhere between ~500-1000 MB larger than the model size computed above. Why? In addition to loading the model, the GPU also needs to be initialized with essential kernels, memory allocation tables, and other GPU related state necessary to using the GPU. This is called the CUDA context.

The CUDA context can be considered a fixed memory overhead for using a Nvidia GPU.

Questions (answer in written submission)

Q1a. How many trainable parameters does ResNet-152 have? What is the estimated size of the model in MB?

Q1b. Which GPU are you using? How much total memory does it have?

Q1c. After you load the model into memory, what is the memory overhead (size) of the CUDA context loaded with the model?

Hint - CUDA context size in this example is roughly (total GPU memory utilization - model size)

2. Optimizer memory usage

The choice of optimizer affects the memory used to train your model. Different optimizers have different memory requirements for storing the gradients and the optimizer state. For example, the Adam optimizer stores a moving average of the gradients and the squared gradients for each parameter, which requires more memory than SGD.

Let's compare the memory usage of three different optimizers - SGD, SGD with momentum and ADAM.


```

In [5]: # Training function
def train_model(model, train_loader, criterion, optimizer, epochs=10, memory_log_path=None):
    os.makedirs(os.path.dirname(memory_log_path), exist_ok=True)
    with open(memory_log_path, 'w') as f:
        f.write(", ".join(MEMORY_LOG_FMT) + "\n")
    for epoch in range(epochs):
        model.train()
        for i, (images, labels) in enumerate(train_loader):
            images = images.to(device)
            labels = labels.to(device)
            with torch.set_grad_enabled(True):
                # Zero all gradients
                optimizer.zero_grad()

                # Get outputs
                outputs = model(images)

                # Compute loss
                loss = criterion(outputs, labels)
                loss.backward()

                # Run optimizer update step
                optimizer.step()

                # Print stats every 100 iterations
                if i % 100 == 0:
                    gpu_memory_usage = get_gpu_memory_usage()
                    print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, GPU Mem: {}'.format(
                        epoch, epochs, i, len(train_loader), loss.item(), gpu_memory_usage))
                    memory_log = [str(time.time()), str(gpu_memory_usage)]
                    with open(memory_log_path, 'a') as f:
                        f.write(", ".join(memory_log) + "\n")
            del loss, outputs, images, labels # To get accurate memory usage info

# Memory profiling function
def profile_mem_usage(optimizer_str):
    """
    Profiles the memory usage of ResNet-152 on CIFAR-10 with the specified optimizer

    optimizer_str: str - Can be either of 'SGD', 'SGD_WITH_MOMENTUM' and 'ADAM'
    """
    # Clean up any dangling objects
    cleanup_memory()
    BATCH_SIZE = 8

    # Since we just want to inspect memory usage, run only one minibatch
    subsampled_data = torch.utils.data.Subset(data_train, range(0, BATCH_SIZE))
    train_loader = torch.utils.data.DataLoader(dataset=subsampled_data,
                                                batch_size=BATCH_SIZE,
                                                shuffle=True)

    # Load model and define loss function
    model = torchvision.models.resnet152(weights=None, num_classes=10)
    model.to(device)
    criterion = torch.nn.CrossEntropyLoss()

    # Choose optimizer
    if optimizer_str == 'SGD':
        optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
    elif optimizer_str == 'SGD_WITH_MOMENTUM':
        optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
    elif optimizer_str == 'ADAM':

```



```

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
else:
    raise NotImplementedError

memory_log_path = ROOT_PATH + f'logs/resnet152_{optimizer_str}.csv'
train_model(model, train_loader, criterion, optimizer, epochs=1, memory_log_path=memory_log_path)
print(f"Memory usage log for {optimizer_str} stored at {memory_log_path}. Restart your runtime")

```

Run memory profiling for various optimizers!

In the cell below, run the `profile_mem_usage` method with three optimizers - 'SGD', 'SGD_WITH_MOMENTUM', 'ADAM'.

☀ NOTE ☀ - to get accurate memory utilization measurements, you **should restart your runtime between invoking `profile_mem_usage` for different optimizers!**

There is state in GPU memory that is not collect by explicitly calling the garbage collector, and thus restarting the runtime is necessary. Your files in colab should persist across runs.

In [6]:

```

# TODO - Run this cell for different optimizers by uncommenting one line at a time.
#
# Make sure to restart the colab runtime between different runs else your
# memory profiles may be inaccurate!

# run in local environment
ROOT_PATH = ""

#profile_mem_usage('SGD')
#profile_mem_usage('SGD_WITH_MOMENTUM')
profile_mem_usage('ADAM')

```

Epoch [1/1], Step [1/1], Loss: 2.0387, GPU Mem: 3540.0
Memory usage log for ADAM stored at logs/resnet152__ADAM.csv. Restart your runtime (Runtime->Restart Runtime) before running for other optimizers!

Analyzing memory usage profiles

Now that you have run `profile_mem_usage` for different optimizers, let's print the memory usage we logged while training with each optimizer.

```
In [7]: OPTIMIZER_LIST = ['SGD', 'SGD_WITH_MOMENTUM', 'ADAM']
memory_log_path = ROOT_PATH + 'logs/resnet152_{opt}.csv'

def print_mem_profiling_results():
    print("==== Memory Profiling Results =====")
    for opt in OPTIMIZER_LIST:
        assert os.path.exists(memory_log_path.format(opt=opt)), f'Memory profile not found for {opt}'
        df = pd.read_csv(memory_log_path.format(opt=opt))
        mem_usage = df['memUsage'].iloc[0]
        print(f'{opt}: {mem_usage} MB')

print_mem_profiling_results()
```

```
==== Memory Profiling Results =====
SGD: 3374.0 MB
SGD_WITH_MOMENTUM: 3456.0 MB
ADAM: 3540.0 MB
```

Questions (answer in written submission)

2a. What is the total memory utilization during training with SGD, SGD with momentum and Adam optimizers? Report in MB individually for each optimizer.

2b. Which optimizer consumes the most memory? Why?

💡 Hint - refer to the weight update rule for each optimizer. Which one requires the most parameters to be stored in memory?

3. Investigating the effect of batch size on convergence and GPU memory

Batch size is an important parameter in training neural networks that can have a significant effect on GPU memory usage. The larger the batch size, the more data the model processes at once, and therefore, the more GPU memory it requires to store the inputs, activations, and gradients.

As the batch size increases, the memory required to store the intermediate results during training increases linearly. This is because the model needs to keep track of more activations and gradients for each layer. However, the actual memory usage can also depend on the specific neural network architecture, as some models require more memory than others to process the same batch size.

If the batch size is too large to fit in the available GPU memory, the training process will fail with an out-of-memory error. On the other hand, if the batch size is too small, the training may be slower due to inefficient use of the GPU, as the GPU may spend more time waiting for data to be transferred from CPU to GPU.

Therefore, choosing an appropriate batch size is important to balance training speed and memory usage. This often involves some trial and error to find the largest batch size that can fit in the available GPU memory while still providing good training results.

Learning Rate and Batch Size

Batch size and learning rate are closely related. When batch size is increased, the gradient estimate becomes less noisy because it is computed over more samples. As a result, the learning rate can be increased, allowing the optimization algorithm to take larger steps towards the optimum. This is because a larger batch size gives a more accurate estimate of the direction of the gradient and larger steps can reduce convergence time.

Large batch training becomes particularly important in data-parallel distributed training, where extremely large batch sizes are distributed over many GPUs. The paper "[Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour](https://arxiv.org/pdf/1706.02677.pdf)" (<https://arxiv.org/pdf/1706.02677.pdf>) is one of the earliest works showing how large batch training makes fast large scale distributed training possible. It also proposes a simple linear scaling rule for setting the learning rate for a given batch size, which we use to set learning rates in `LR_MAP` below.

Let's try training our model with different batch sizes

In the below cells, we'll try to run training for different batch sizes and evaluate the performance.

Note - you may run out of memory for large batch sizes, and that is expected! Ignore those large batch sizes and stick with the batch sizes that can fit on your GPU.

Let's first define helper functions.


```

In [8]: # Test function
def test_model(model, test_loader, label='test'):
    print("Testing model.")
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0

        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)

            _, predicted = torch.max(outputs.data, 1)

            correct += (predicted == labels).sum().item()
            total += labels.size(0)

        # Compute accuracy
        accuracy = 100 * correct / total
        print(f'Accuracy of the model on {label} images: {accuracy} %')
        del outputs, images, labels # To get accurate memory usage info
    return accuracy

# Training function
def train_model(model, train_loader, criterion, optimizer, epochs=10, memory_log_path=None):
    os.makedirs(os.path.dirname(memory_log_path), exist_ok=True)
    with open(memory_log_path, 'w') as f:
        f.write(", ".join(TRAIN_LOG_FMT) + "\n")
    for epoch in range(epochs):
        model.train()
        last_loss = 0
        for i, (images, labels) in enumerate(train_loader):
            images = images.to(device)
            labels = labels.to(device)
            with torch.set_grad_enabled(True):
                # Zero all gradients
                optimizer.zero_grad()

                # Get outputs
                outputs = model(images)

                # Compute loss
                loss = criterion(outputs, labels)
                loss.backward()

                # Run optimizer update step
                optimizer.step()

            last_loss = loss.item()
            # Print stats every 100 iterations
            if i % 10 == 0:
                gpu_memory_usage = get_gpu_memory_usage()
                print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, GPU Mem: {}'.format(
                    epoch, epochs, i, len(train_loader), loss.item(), gpu_memory_usage))
            del loss, outputs, images, labels # To get accurate memory usage info

        # Report test or train accuracy at the end of every epoch
        if test_loader:
            accuracy = test_model(model, test_loader, label='test')
        else:
            accuracy = test_model(model, train_loader, label='train')

```

```

        # Log results
        memory_log = [str(time.time()), str(epoch+1), str(gpu_memory_usage), str(loss)]
        with open(memory_log_path, 'a') as f:
            f.write(",".join(memory_log) + "\n")

# Set learning rates for different batch sizes (empirically determined and linearly scaled)
LR_MAP = {
    4: 0.0001,
    8: 0.0002,
    16: 0.0004,
    32: 0.0008,
    64: 0.0016,
    128: 0.0032,
    256: 0.0064,
    512: 0.0064,
    1024: 0.0064
}

# Executor function
def run_train(batch_size, epochs=10):
    cleanup_memory()

    lr = LR_MAP[batch_size]
    print(f"Training model with batch size {batch_size} and lr {lr}.")

    train_loader = torch.utils.data.DataLoader(dataset=subsampled_train_data, batch_size=batch_size)

    # We use a smaller model (resnet18) to train faster
    model = torchvision.models.resnet18(weights=None, num_classes=10)
    model.to(device)
    criterion = torch.nn.CrossEntropyLoss()

    optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)

    # Output path for memory logs
    memory_log_path = ROOT_PATH + f'logs/resnet18_{batch_size}.csv'

    # Run training!
    train_model(model, train_loader, criterion, optimizer, epochs=epochs, memory_log_path=memory_log_path)

```

Run training for different batch sizes and record their memory utilization!

In the cell below, **run the `run_train` method for batch sizes 4, 16, 64, 256, 1024.**

This method will log the loss, accuracy, wall clock time and memory utilization under `/content/cs182hw4/logs` directory, so you can safely restart the runtime between invocations.

Like before, to get accurate memory utilization measurements, you should **restart your runtime between invoking `run_train` for different batch sizes!**

```
In [9]: # TODO - Run this cell for different batch sizes by uncommenting one line at a time.
#
# Make sure to restart the colab runtime between different runs else your
# memory profiles may be inaccurate!

# Run each batch size for at least 10 epochs. You can configure this to be larger if
epochs = 10

run_train(4, epochs=epochs)
# run_train(16, epochs=epochs)
# run_train(64, epochs=epochs)
# run_train(256, epochs=epochs)
# run_train(1024, epochs=epochs)
```

```
Training model with batch size 4 and lr 0.0001.
Epoch [1/10], Step [1/1024], Loss: 2.4767, GPU Mem: 1624.0
Epoch [1/10], Step [11/1024], Loss: 2.1517, GPU Mem: 1684.0
Epoch [1/10], Step [21/1024], Loss: 2.3811, GPU Mem: 1684.0
Epoch [1/10], Step [31/1024], Loss: 2.3056, GPU Mem: 1684.0
Epoch [1/10], Step [41/1024], Loss: 2.4940, GPU Mem: 1684.0
Epoch [1/10], Step [51/1024], Loss: 2.3186, GPU Mem: 1684.0
Epoch [1/10], Step [61/1024], Loss: 2.3357, GPU Mem: 1684.0
Epoch [1/10], Step [71/1024], Loss: 2.2846, GPU Mem: 1684.0
Epoch [1/10], Step [81/1024], Loss: 2.2858, GPU Mem: 1684.0
Epoch [1/10], Step [91/1024], Loss: 2.1944, GPU Mem: 1684.0
Epoch [1/10], Step [101/1024], Loss: 2.1375, GPU Mem: 1684.0
Epoch [1/10], Step [111/1024], Loss: 2.3502, GPU Mem: 1684.0
Epoch [1/10], Step [121/1024], Loss: 2.0923, GPU Mem: 1684.0
Epoch [1/10], Step [131/1024], Loss: 2.2534, GPU Mem: 1684.0
Epoch [1/10], Step [141/1024], Loss: 2.3388, GPU Mem: 1684.0
Epoch [1/10], Step [151/1024], Loss: 2.0335, GPU Mem: 1684.0
Epoch [1/10], Step [161/1024], Loss: 2.1323, GPU Mem: 1684.0
Epoch [1/10], Step [171/1024], Loss: 2.3699, GPU Mem: 1684.0
Epoch [1/10], Step [181/1024], Loss: 2.0254, GPU Mem: 1684.0
```

```
In [10]: run_train(16, epochs=epochs)
```

```
Training model with batch size 16 and lr 0.0004.
Epoch [1/10], Step [1/256], Loss: 2.4846, GPU Mem: 2142.0
Epoch [1/10], Step [11/256], Loss: 2.3922, GPU Mem: 2142.0
Epoch [1/10], Step [21/256], Loss: 2.3477, GPU Mem: 2142.0
Epoch [1/10], Step [31/256], Loss: 2.2929, GPU Mem: 2142.0
Epoch [1/10], Step [41/256], Loss: 2.2871, GPU Mem: 2142.0
Epoch [1/10], Step [51/256], Loss: 2.2597, GPU Mem: 2142.0
Epoch [1/10], Step [61/256], Loss: 2.1548, GPU Mem: 2142.0
Epoch [1/10], Step [71/256], Loss: 2.1083, GPU Mem: 2142.0
Epoch [1/10], Step [81/256], Loss: 2.3286, GPU Mem: 2142.0
Epoch [1/10], Step [91/256], Loss: 2.1467, GPU Mem: 2142.0
Epoch [1/10], Step [101/256], Loss: 2.1294, GPU Mem: 2142.0
Epoch [1/10], Step [111/256], Loss: 2.2383, GPU Mem: 2142.0
Epoch [1/10], Step [121/256], Loss: 2.0882, GPU Mem: 2142.0
Epoch [1/10], Step [131/256], Loss: 2.1027, GPU Mem: 2142.0
Epoch [1/10], Step [141/256], Loss: 2.2149, GPU Mem: 2142.0
Epoch [1/10], Step [151/256], Loss: 2.2191, GPU Mem: 2142.0
Epoch [1/10], Step [161/256], Loss: 2.0440, GPU Mem: 2142.0
Epoch [1/10], Step [171/256], Loss: 1.9828, GPU Mem: 2142.0
Epoch [1/10], Step [181/256], Loss: 1.9850, GPU Mem: 2142.0
```

```
In [11]: run_train(64, epochs=epochs)
```


Training model with batch size 64 and lr 0.0016.

Epoch [1/10], Step [1/64], Loss: 2.2998, GPU Mem: 4151.0
Epoch [1/10], Step [11/64], Loss: 2.2376, GPU Mem: 4231.0
Epoch [1/10], Step [21/64], Loss: 2.1713, GPU Mem: 4212.0
Epoch [1/10], Step [31/64], Loss: 2.1049, GPU Mem: 3957.0
Epoch [1/10], Step [41/64], Loss: 2.0591, GPU Mem: 3963.0
Epoch [1/10], Step [51/64], Loss: 2.1155, GPU Mem: 3959.0
Epoch [1/10], Step [61/64], Loss: 1.9134, GPU Mem: 3959.0

Testing model.

Accuracy of the model on train images: 25.9521484375 %

Epoch [2/10], Step [1/64], Loss: 2.0261, GPU Mem: 4133.0
Epoch [2/10], Step [11/64], Loss: 1.8944, GPU Mem: 4176.0
Epoch [2/10], Step [21/64], Loss: 1.8448, GPU Mem: 4144.0
Epoch [2/10], Step [31/64], Loss: 1.9030, GPU Mem: 4126.0
Epoch [2/10], Step [41/64], Loss: 2.0297, GPU Mem: 3996.0
Epoch [2/10], Step [51/64], Loss: 1.8204, GPU Mem: 3991.0
Epoch [2/10], Step [61/64], Loss: 1.8499, GPU Mem: 3986.0

Testing model.

Accuracy of the model on train images: 31.689453125 %

Epoch [3/10], Step [1/64], Loss: 1.5081, GPU Mem: 3993.0
Epoch [3/10], Step [11/64], Loss: 1.7703, GPU Mem: 3993.0
Epoch [3/10], Step [21/64], Loss: 1.7576, GPU Mem: 3988.0
Epoch [3/10], Step [31/64], Loss: 1.6909, GPU Mem: 3988.0
Epoch [3/10], Step [41/64], Loss: 1.6434, GPU Mem: 3986.0
Epoch [3/10], Step [51/64], Loss: 1.7583, GPU Mem: 3992.0
Epoch [3/10], Step [61/64], Loss: 1.7846, GPU Mem: 3992.0

Testing model.

Accuracy of the model on train images: 32.1533203125 %

Epoch [4/10], Step [1/64], Loss: 1.6611, GPU Mem: 3986.0
Epoch [4/10], Step [11/64], Loss: 1.5809, GPU Mem: 3986.0
Epoch [4/10], Step [21/64], Loss: 1.5279, GPU Mem: 3987.0
Epoch [4/10], Step [31/64], Loss: 1.6131, GPU Mem: 3994.0
Epoch [4/10], Step [41/64], Loss: 1.7628, GPU Mem: 3736.0
Epoch [4/10], Step [51/64], Loss: 1.7254, GPU Mem: 3797.0
Epoch [4/10], Step [61/64], Loss: 1.8396, GPU Mem: 3967.0

Testing model.

Accuracy of the model on train images: 32.12890625 %

Epoch [5/10], Step [1/64], Loss: 1.5448, GPU Mem: 4346.0
Epoch [5/10], Step [11/64], Loss: 1.6125, GPU Mem: 4346.0
Epoch [5/10], Step [21/64], Loss: 1.4955, GPU Mem: 4346.0
Epoch [5/10], Step [31/64], Loss: 1.5518, GPU Mem: 4346.0
Epoch [5/10], Step [41/64], Loss: 1.5418, GPU Mem: 4346.0
Epoch [5/10], Step [51/64], Loss: 1.4727, GPU Mem: 4346.0
Epoch [5/10], Step [61/64], Loss: 1.4729, GPU Mem: 4346.0

Testing model.

Accuracy of the model on train images: 35.7666015625 %

Epoch [6/10], Step [1/64], Loss: 1.4136, GPU Mem: 4347.0
Epoch [6/10], Step [11/64], Loss: 1.4385, GPU Mem: 4347.0
Epoch [6/10], Step [21/64], Loss: 1.5503, GPU Mem: 4347.0
Epoch [6/10], Step [31/64], Loss: 1.4571, GPU Mem: 4347.0
Epoch [6/10], Step [41/64], Loss: 1.5516, GPU Mem: 4347.0
Epoch [6/10], Step [51/64], Loss: 1.3277, GPU Mem: 4347.0
Epoch [6/10], Step [61/64], Loss: 1.2421, GPU Mem: 4347.0

Testing model.

Accuracy of the model on train images: 43.603515625 %

Epoch [7/10], Step [1/64], Loss: 1.3208, GPU Mem: 4347.0
Epoch [7/10], Step [11/64], Loss: 1.2852, GPU Mem: 4347.0
Epoch [7/10], Step [21/64], Loss: 1.1607, GPU Mem: 4347.0
Epoch [7/10], Step [31/64], Loss: 1.5193, GPU Mem: 4347.0
Epoch [7/10], Step [41/64], Loss: 1.3003, GPU Mem: 4347.0
Epoch [7/10], Step [51/64], Loss: 1.1684, GPU Mem: 4347.0


```

Epoch [7/10], Step [61/64], Loss: 1.3704, GPU Mem: 4347.0
Testing model.
Accuracy of the model on train images: 47.1435546875 %
Epoch [8/10], Step [1/64], Loss: 1.3224, GPU Mem: 4347.0
Epoch [8/10], Step [11/64], Loss: 1.3766, GPU Mem: 4347.0
Epoch [8/10], Step [21/64], Loss: 1.2777, GPU Mem: 4347.0
Epoch [8/10], Step [31/64], Loss: 1.1669, GPU Mem: 4323.0
Epoch [8/10], Step [41/64], Loss: 1.2048, GPU Mem: 4323.0
Epoch [8/10], Step [51/64], Loss: 1.1923, GPU Mem: 4323.0
Epoch [8/10], Step [61/64], Loss: 1.3048, GPU Mem: 4323.0
Testing model.
Accuracy of the model on train images: 54.7119140625 %
Epoch [9/10], Step [1/64], Loss: 1.3521, GPU Mem: 4337.0
Epoch [9/10], Step [11/64], Loss: 1.2410, GPU Mem: 4337.0
Epoch [9/10], Step [21/64], Loss: 1.0606, GPU Mem: 4337.0
Epoch [9/10], Step [31/64], Loss: 1.2307, GPU Mem: 4337.0
Epoch [9/10], Step [41/64], Loss: 1.3679, GPU Mem: 4338.0
Epoch [9/10], Step [51/64], Loss: 0.9634, GPU Mem: 4338.0
Epoch [9/10], Step [61/64], Loss: 1.1704, GPU Mem: 4338.0
Testing model.
Accuracy of the model on train images: 48.9990234375 %
Epoch [10/10], Step [1/64], Loss: 1.1062, GPU Mem: 4346.0
Epoch [10/10], Step [11/64], Loss: 1.1290, GPU Mem: 4347.0
Epoch [10/10], Step [21/64], Loss: 1.0115, GPU Mem: 4347.0
Epoch [10/10], Step [31/64], Loss: 0.8880, GPU Mem: 4347.0
Epoch [10/10], Step [41/64], Loss: 0.9070, GPU Mem: 4347.0
Epoch [10/10], Step [51/64], Loss: 1.0558, GPU Mem: 4347.0
Epoch [10/10], Step [61/64], Loss: 1.2961, GPU Mem: 4349.0
Testing model.
Accuracy of the model on train images: 49.2919921875 %

```

```
In [12]: run_train(256, epochs=epochs)
```

Training model with batch size 256 and lr 0.0064.

 OutOfMemoryError: CUDA out of memory. Tried to allocate 50.00 MiB (GPU 0; 6.00 GiB total capacity; 5.29 GiB already allocated; 0 bytes free; 5.33 GiB reserved in total by PyTorch) If reserved memory is >> allocated memory try setting max_split_size_mb to avoid fragmentation. See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF ►

Plot the loss, accuracy and memory utilization

Once all logs have been generated under `/content/cs182hw4/logs`, run the cell below to plot loss and accuracy against wall clock time.

```

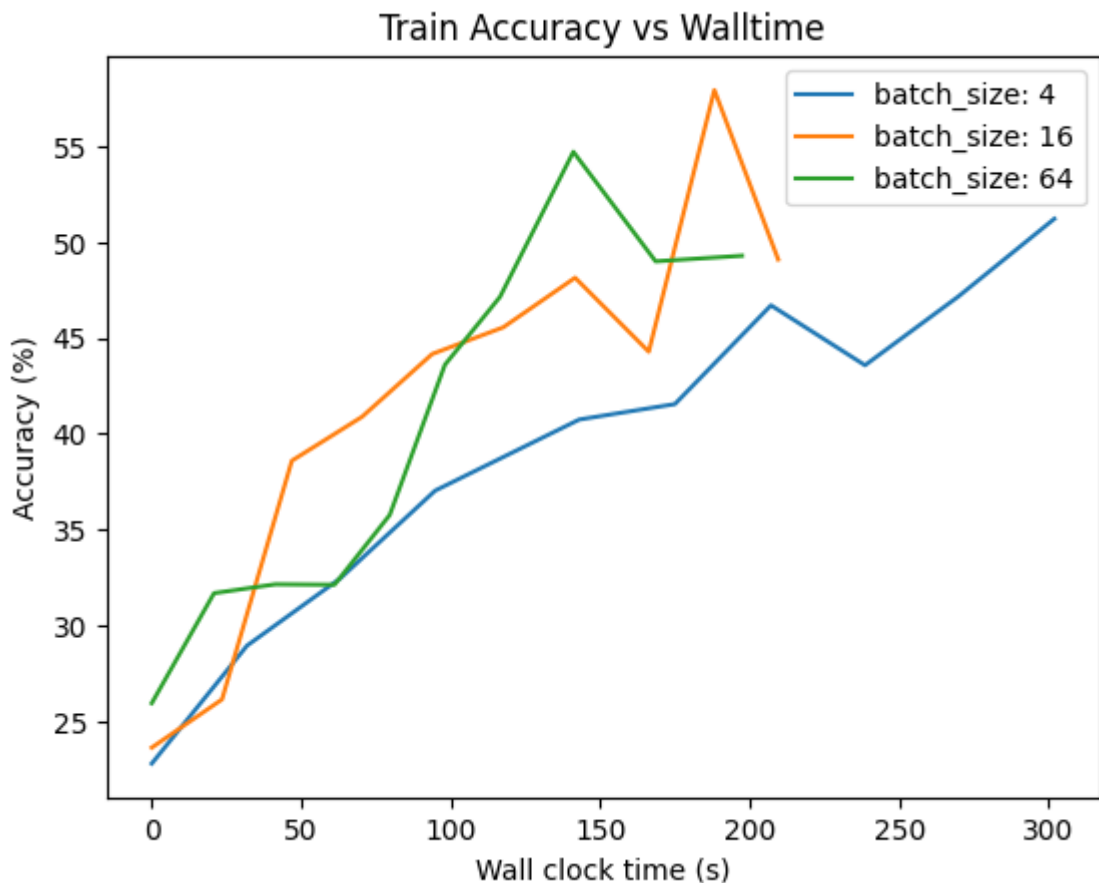
In [14]: # Plotting scripts
def get_df(batch_size):
    path = ROOT_PATH + f'logs/resnet18_{batch_size}.csv'
    assert os.path.exists(path), f'Memory profile not found for batch size {batch_s
    df = pd.read_csv(path)
    # Create a wall time column
    df['walltime'] = df['timestamp'] - df['timestamp'].iloc[0]
    return df

def plot_walltime_acc(batch_sizes):
    plt.figure()
    for bs in batch_sizes:
        df=get_df(bs)
        plt.plot(df['walltime'], df['accuracy'], label=f'batch_size: {bs}')
    plt.xlabel('Wall clock time (s)')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.title('Train Accuracy vs Walltime')
    plt.show()

def print_mem_usage(batch_sizes):
    print("\n===== Memory Usage for different batch sizes =====")
    for bs in batch_sizes:
        df=get_df(bs)
        mem_usage = df['memUsage'].iloc[-1]
        print(f'{bs}\t: {mem_usage} MB')

#batch_sizes = [4, 16, 64, 256]
batch_sizes = [4, 16, 64]
plot_walltime_acc(batch_sizes)
print_mem_usage(batch_sizes)

```



```
===== Memory Usage for different batch sizes =====  
4       : 1684.0 MB  
16      : 2325.0 MB  
64      : 4349.0 MB
```

Questions (answer in written submission)

3a. What is the memory utilization for different batch sizes (4, 16, 64, 256)? What is the largest batch size you were able to train?

3b. Which batch size gave you the highest accuracy at the end of 10 epochs?

3c. Which batch size completed 10 epochs the fastest (least wall clock time)? Why?

3d. Attach your training accuracy vs wall time plots with your written submission.