****

**Comsats University Islamabad, Lahore campus**

**ASSIGNMENT#4(Lab)**

**RAMSHA KHAN**

**(SP23-BCS-112)**

**Section: C**

**Course: PDC**

**Instructor’s name:Akhzar Nazir**

**Due date: 12-10-25**

**Part 1: CPU vs GPU Model Training**

**1. Choose a simple dataset (e.g., MNIST, CIFAR-10, or custom synthetic data).**

**2. Build a simple neural network using PyTorch or TensorFlow:**

**o Example: 2–3 fully connected layers or a small CNN.**

**3. Train the model once on CPU and once on GPU.**

**4. Measure:**

**o Training time per epoch**

**o Total training time**

**o GPU utilization (using nvidia-smi or torch.cuda.memory\_allocated())**

**5. Compute Speedup:**

**Speedup = CPU Time / GPU Time**

**Discuss the reason for the speedup (or lack thereof) — consider data size, GPU memory**

**transfers, and model size.**

import time

import torch

import torch.nn as nn

import torch.optim as optim

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

trainset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)

trainloader = DataLoader(trainset, batch\_size=64, shuffle=True)

class SimpleNN(nn.Module):

    def \_\_init\_\_(self):

        super(SimpleNN, self).\_\_init\_\_()

        self.fc1 = nn.Linear(28\*28, 128)

        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):

        x = x.view(-1, 28\*28)

        x = torch.relu(self.fc1(x))

        return self.fc2(x)

def train(model, device, epochs=3):

    model.to(device)

    optimizer = optim.Adam(model.parameters(), lr=0.001)

    criterion = nn.CrossEntropyLoss()

    start\_time = time.time()

    for epoch in range(epochs):

        for images, labels in trainloader:

            images, labels = images.to(device), labels.to(device)

            optimizer.zero\_grad()

            outputs = model(images)

            loss = criterion(outputs, labels)

            loss.backward()

            optimizer.step()

    total\_time = time.time() - start\_time

    return total\_time

cpu\_device = torch.device("cpu")

cpu\_time = train(SimpleNN(), cpu\_device)

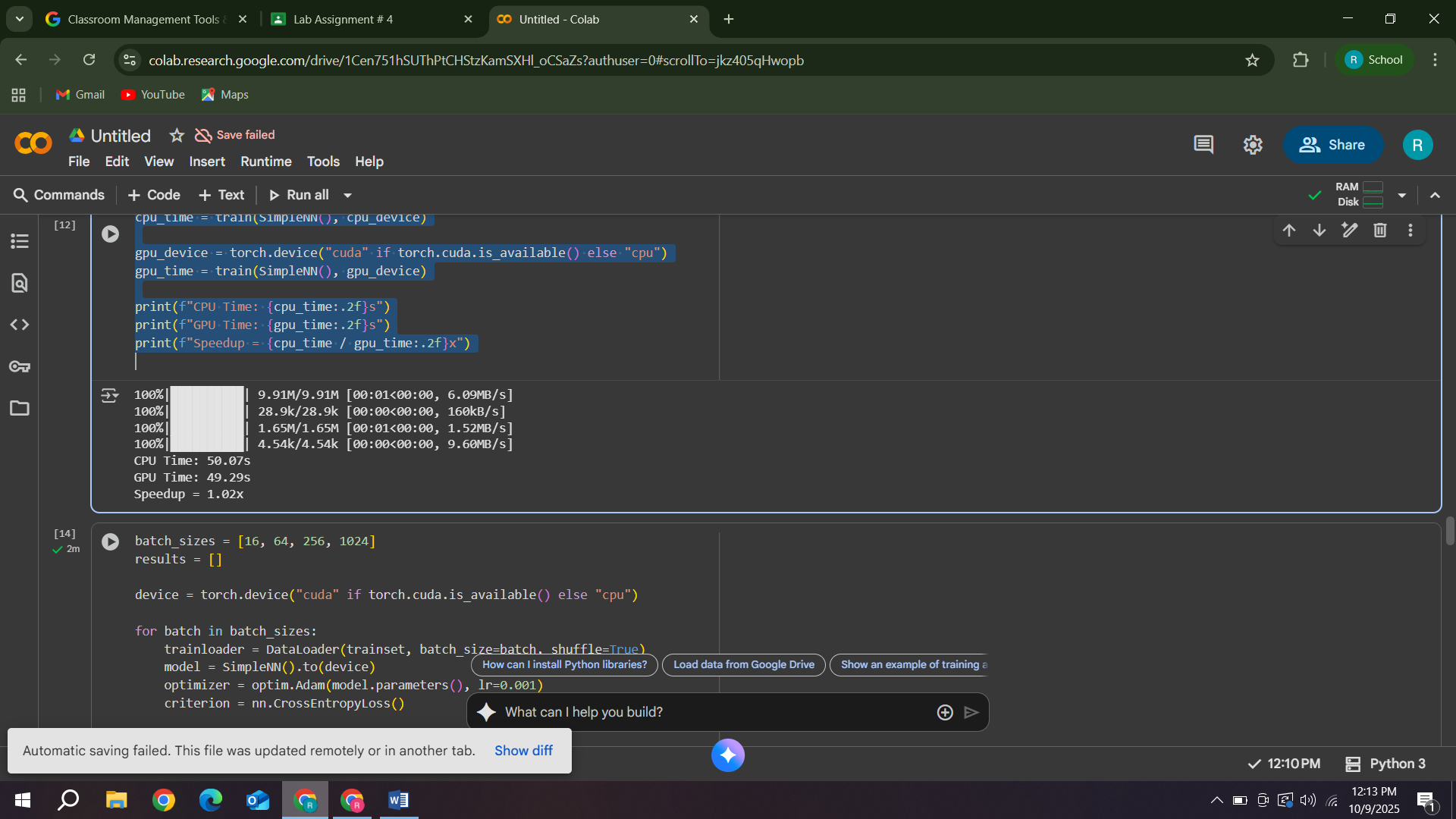
gpu\_device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

gpu\_time = train(SimpleNN(), gpu\_device)

print(f"CPU Time: {cpu\_time:.2f}s")

print(f"GPU Time: {gpu\_time:.2f}s")

print(f"Speedup = {cpu\_time / gpu\_time:.2f}x")



**Part 2: Effect of Batch Size**

**1. Train the same model with different batch sizes (e.g., 16, 64, 256, 1024).**

**2. Record:**

**o Time per epoch**

**o GPU memory usage**

**o Accuracy or loss trends**

**3. Plot Batch Size vs Training Time and Batch Size vs GPU Utilization.**

batch\_sizes = [16, 64, 256, 1024]

results = []

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

for batch in batch\_sizes:

    trainloader = DataLoader(trainset, batch\_size=batch, shuffle=True)

    model = SimpleNN().to(device)

    optimizer = optim.Adam(model.parameters(), lr=0.001)

    criterion = nn.CrossEntropyLoss()

    start = time.time()

    for epoch in range(2):

        for images, labels in trainloader:

            images, labels = images.to(device), labels.to(device)

            optimizer.zero\_grad()

            outputs = model(images)

            loss = criterion(outputs, labels)

            loss.backward()

            optimizer.step()

    epoch\_time = time.time() - start

    mem = torch.cuda.memory\_allocated(device) / (1024 \*\* 2)

    results.append((batch, epoch\_time, mem))

    print(f"Batch {batch}: Time {epoch\_time:.2f}s, GPU Memory {mem:.1f}MB")

import matplotlib.pyplot as plt

batches, times, mems = zip(\*results)

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)

plt.plot(batches, times, marker='o')

plt.title('Batch Size vs Training Time')

plt.xlabel('Batch Size'); plt.ylabel('Time (s)')

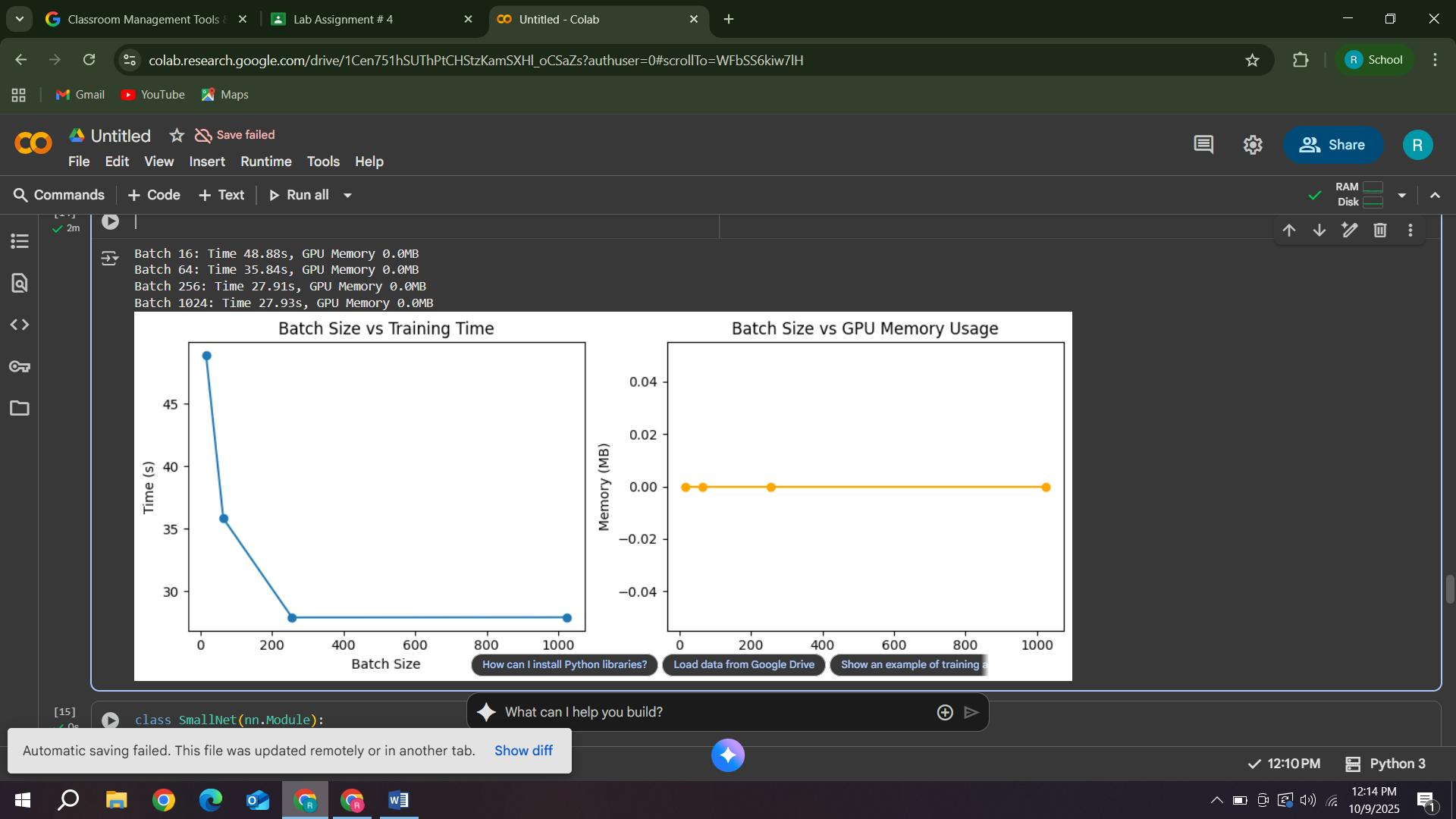
plt.subplot(1,2,2)

plt.plot(batches, mems, marker='o', color='orange')

plt.title('Batch Size vs GPU Memory Usage')

plt.xlabel('Batch Size'); plt.ylabel('Memory (MB)')

plt.tight\_layout(); plt.show()



**Question3:**

**Discussion**

**Question1:** **Why does increasing batch size improve GPU efficiency up to a point?**

**Explanation:**

When you increase the batch size, the GPU processes more samples in parallel during each forward and backward pass.

This improves efficiency because GPUs are designed for parallel computation — they can handle thousands of small operations simultaneously.

**Reasons:**

* **Better parallel utilization:**

Larger batches make better use of the GPU’s many cores, so it spends less time idle.

* **Reduced data transfer overhead:**

Fewer mini-batches mean fewer times data must be transferred between CPU and GPU.

* **Fewer updates = less synchronization:**

You compute gradients fewer times per epoch, reducing communication overhead.

**But only “up to a point”:**

After a certain batch size, the GPU’s memory becomes full or you reach diminishing returns:

* GPU RAM limits how many samples fit in memory.
* Larger batches mean each iteration takes longer (more waiting before weight updates).
* Eventually, computation efficiency stops improving — or even worsens.

**2. Why does accuracy sometimes drop for very large batches?**

**Explanation:**

When batch size is too large, the model updates weights less frequently and the gradients become less noisy.

Although this might sound good, in practice it can hurt generalization.

**Reasons:**

**Less gradient noise = poorer generalization:**

Small batches add a bit of randomness (noise) to each gradient step.

This noise helps the model escape local minima and find a smoother solution that generalizes better.

**Too stable gradients:**

Large batches approximate the true gradient too perfectly, making optimization less dynamic.

**Fewer weight updates per epoch:**

With huge batches, you complete fewer updates per epoch → slower learning progress.

**Overfitting to large-batch patterns**:

The model may focus on dominant trends and ignore subtle patterns, leading to lower test accuracy.

**Part 3: Model Complexity and GPU Utilization**

**1. Train three models of increasing complexity:**

**o Small: 1–2 layers**

**o Medium: 3–5 layers**

**o Large: CNN with several convolutional layers**

**2. Use the same dataset, epochs, and optimizer.**

**3. Record:**

**o GPU utilization**

**o Training time per epoch**

**o Peak GPU memory usagePart 3: Model Complexity and GPU Utilization**

**1. Train three models of increasing complexity:**

**o Small: 1–2 layers**

**o Medium: 3–5 layers**

**o Large: CNN with several convolutional layers**

**2. Use the same dataset, epochs, and optimizer.**

**3. Record:**

**o GPU utilization**

**o Training time per epoch**

**o Peak GPU memory usage**

!pip install torch torchvision matplotlib --quiet

import torch

import torch.nn as nn

import torch.nn.functional as F

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

import time, pandas as pd, matplotlib.pyplot as plt

print("CUDA available:", torch.cuda.is\_available())

if torch.cuda.is\_available():

    print("GPU:", torch.cuda.get\_device\_name(0))

else:

    print("⚠️ Please enable GPU in Runtime settings.")

transform = transforms.Compose([

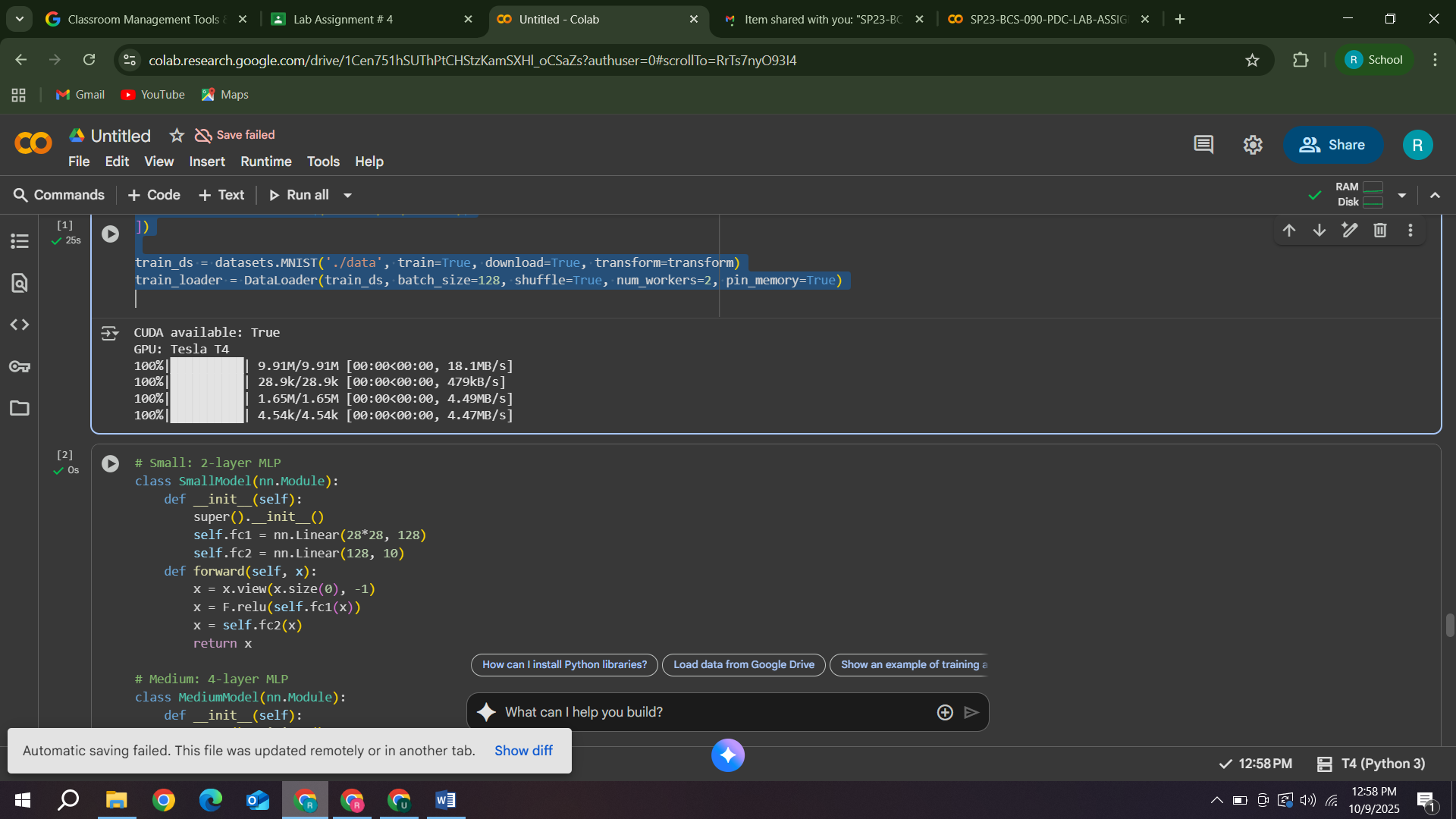
    transforms.ToTensor(),

    transforms.Normalize((0.1307,), (0.3081,))

])

train\_ds = datasets.MNIST('./data', train=True, download=True, transform=transform)

train\_loader = DataLoader(train\_ds, batch\_size=128, shuffle=True, num\_workers=2, pin\_memory=True)



class SmallModel(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.fc1 = nn.Linear(28\*28, 128)

        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):

        x = x.view(x.size(0), -1)

        x = F.relu(self.fc1(x))

        x = self.fc2(x)

        return x

class MediumModel(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.fc1 = nn.Linear(28\*28, 512)

        self.fc2 = nn.Linear(512, 256)

        self.fc3 = nn.Linear(256, 128)

        self.fc4 = nn.Linear(128, 10)

    def forward(self, x):

        x = x.view(x.size(0), -1)

        x = F.relu(self.fc1(x))

        x = F.relu(self.fc2(x))

        x = F.relu(self.fc3(x))

        x = self.fc4(x)

        return x

class LargeCNN(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.conv1 = nn.Conv2d(1, 32, 3, padding=1)

        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)

        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

        self.conv4 = nn.Conv2d(128, 256, 3, padding=1)

        self.pool = nn.MaxPool2d(2, 2)

        with torch.no\_grad():

            x = torch.zeros(1, 1, 28, 28)

            x = F.relu(self.conv1(x))

            x = self.pool(F.relu(self.conv2(x)))

            x = self.pool(F.relu(self.conv3(x)))

            x = F.relu(self.conv4(x))

            flat\_size = x.view(1, -1).size(1)

        self.fc1 = nn.Linear(flat\_size, 128)

        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):

        x = F.relu(self.conv1(x))

        x = self.pool(F.relu(self.conv2(x)))

        x = self.pool(F.relu(self.conv3(x)))

        x = F.relu(self.conv4(x))

        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))

        x = self.fc2(x)

        return x

def train\_one\_epoch(model, loader, device, optimizer, criterion):

    model.train()

    if device.type == "cuda":

        torch.cuda.synchronize()

    start = time.perf\_counter()

    total\_loss = 0

    for xb, yb in loader:

        xb, yb = xb.to(device, non\_blocking=True), yb.to(device, non\_blocking=True)

        optimizer.zero\_grad()

        loss = criterion(model(xb), yb)

        loss.backward()

        optimizer.step()

        total\_loss += loss.item() \* xb.size(0)

    if device.type == "cuda":

        torch.cuda.synchronize()

    end = time.perf\_counter()

    return (end - start), total\_loss / len(loader.dataset)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

criterion = nn.CrossEntropyLoss()

models = {

    "Small": SmallModel(),

    "Medium": MediumModel(),

    "LargeCNN": LargeCNN()

}

results = []

for name, model in models.items():

    model = model.to(device)

    optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

    print(f"\n🔹 Training {name} model")

    torch.cuda.reset\_peak\_memory\_stats(device)

    epoch\_time, loss = train\_one\_epoch(model, train\_loader, device, optimizer, criterion)

    peak\_mem = torch.cuda.max\_memory\_allocated(device) / 1024\*\*2

    !nvidia-smi --query-gpu=utilization.gpu,memory.used --format=csv,noheader,nounits | head -n 1

    results.append({

        "Model": name,

        "Epoch Time (s)": round(epoch\_time, 2),

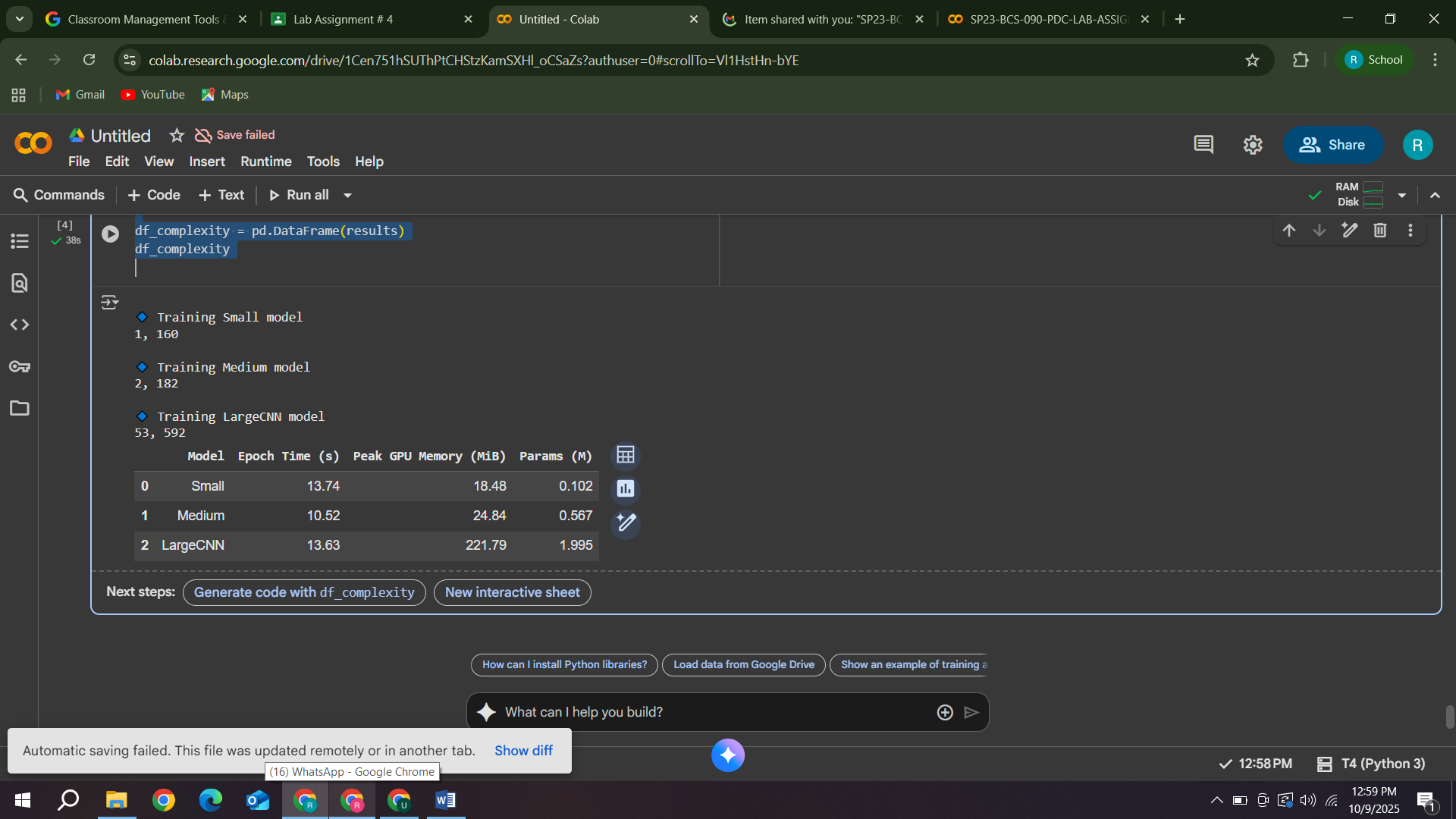
        "Peak GPU Memory (MiB)": round(peak\_mem, 2),

        "Params (M)": round(sum(p.numel() for p in model.parameters()) / 1e6, 3)

    })

df\_complexity = pd.DataFrame(results)

df\_complexity



**1. How model size affects GPU workload and training time**

* As model complexity increases (more layers and parameters), GPU workload and memory usage increase.
* The GPU performs more matrix multiplications and gradient computations, hence higher utilization and longer training times.
* Small models may underutilize the GPU, while large models fully utilize it but take longer to train.

**2. How GPU compute and memory balance affect performance**

* If your model fits comfortably in GPU memory, performance is optimal.
* When memory gets near full, GPU may slow due to memory swapping or limited batch size.
* Too small a model → GPU idle cycles (underutilization).
* Too large a model → Out-of-memory errors or slower training.

**Part 4: Data Loading and Bottlenecks**

**1. Enable multi-threaded data loading (PyTorch: num\_workers in DataLoader).**

**2. Compare performance for:**

**o num\_workers = 0 (single-threaded)**

**o num\_workers = 2, 4, 8**

**3. Measure:**

**o Data loading time per batch**

**o GPU idle time (e.g., using nvidia-smi dmon)**

num\_workers\_list = [0, 2, 4, 8]

results = []

for workers in num\_workers\_list:

    loader = DataLoader(trainset, batch\_size=128, shuffle=True, num\_workers=workers)

    model = SimpleNN().to(device)

    optimizer = optim.Adam(model.parameters(), lr=0.001)

    criterion = nn.CrossEntropyLoss()

    start = time.time()

    for epoch in range(1):

        for images, labels in loader:

            images, labels = images.to(device), labels.to(device)

            optimizer.zero\_grad()

            outputs = model(images)

            loss = criterion(outputs, labels)

            loss.backward()

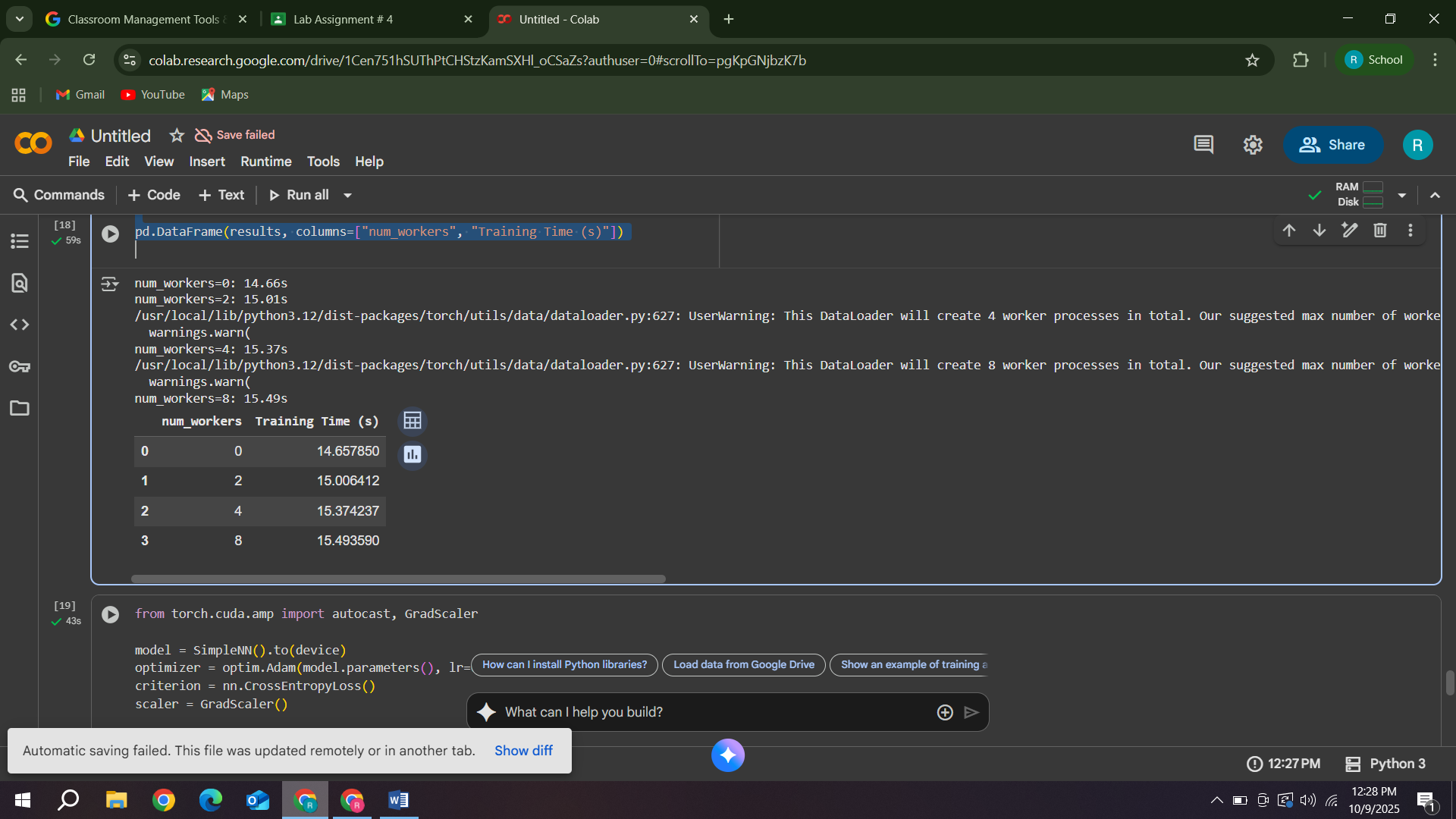
            optimizer.step()

    total\_time = time.time() - start

    results.append((workers, total\_time))

    print(f"num\_workers={workers}: {total\_time:.2f}s")

pd.DataFrame(results, columns=["num\_workers", "Training Time (s)"])



**Explanation:**

**Why inefficient data pipelines cause low GPU utilization**

When training a model, two main tasks happen in parallel:

**CPU:** Loads and preprocesses the next batch of data (from disk, applies transforms, etc.)

**GPU:** Performs forward and backward propagation (actual model training).

If the CPU is slow at loading or preprocessing data, the GPU finishes its current batch and sits idle, waiting for the next batch.

This idle time reduces GPU utilization — even though your GPU is powerful, it’s not being fed data fast enough.

**Analogy:**

It’s like a chef (GPU) waiting for ingredients (data) from a slow assistant (CPU).

If the assistant can’t supply fast enough, the chef wastes time doing nothing.

**Common causes of inefficiency:**

Using num\_workers=0 in DataLoader (single-threaded loading)

Heavy preprocessing done on CPU

Slow disk I/O (especially when reading many small files)

Data augmentation not parallelized

**How overlapping CPU data loading and GPU training improves performance**

PyTorch’s DataLoader can use multiple worker threads/processes via the num\_workers argument.

This allows the CPU to load the next batch while the GPU is training on the current one — this is overlapping of tasks.

So, while the GPU works, the CPU is already preparing data for the next iteration.

When the GPU finishes one batch, the next batch is ready immediately, so it never waits.

Result:

* Higher GPU utilization
* Faster training (less idle time)
* Better pipeline efficiency

**Part 5: Mixed Precision Training (Optional, Bonus)**

**1. Use PyTorch’s torch.cuda.amp (Automatic Mixed Precision).**

**2. Train your model with and without AMP.**

**3. Compare:**

**o Speed per epoch**

**o Final accuracy**

**o GPU memory usage**

from torch.cuda.amp import autocast, GradScaler

model = SimpleNN().to(device)

optimizer = optim.Adam(model.parameters(), lr=0.001)

criterion = nn.CrossEntropyLoss()

scaler = GradScaler()

start = time.time()

for epoch in range(3):

    for images, labels in trainloader:

        images, labels = images.to(device), labels.to(device)

        optimizer.zero\_grad()

        with autocast():

            outputs = model(images)

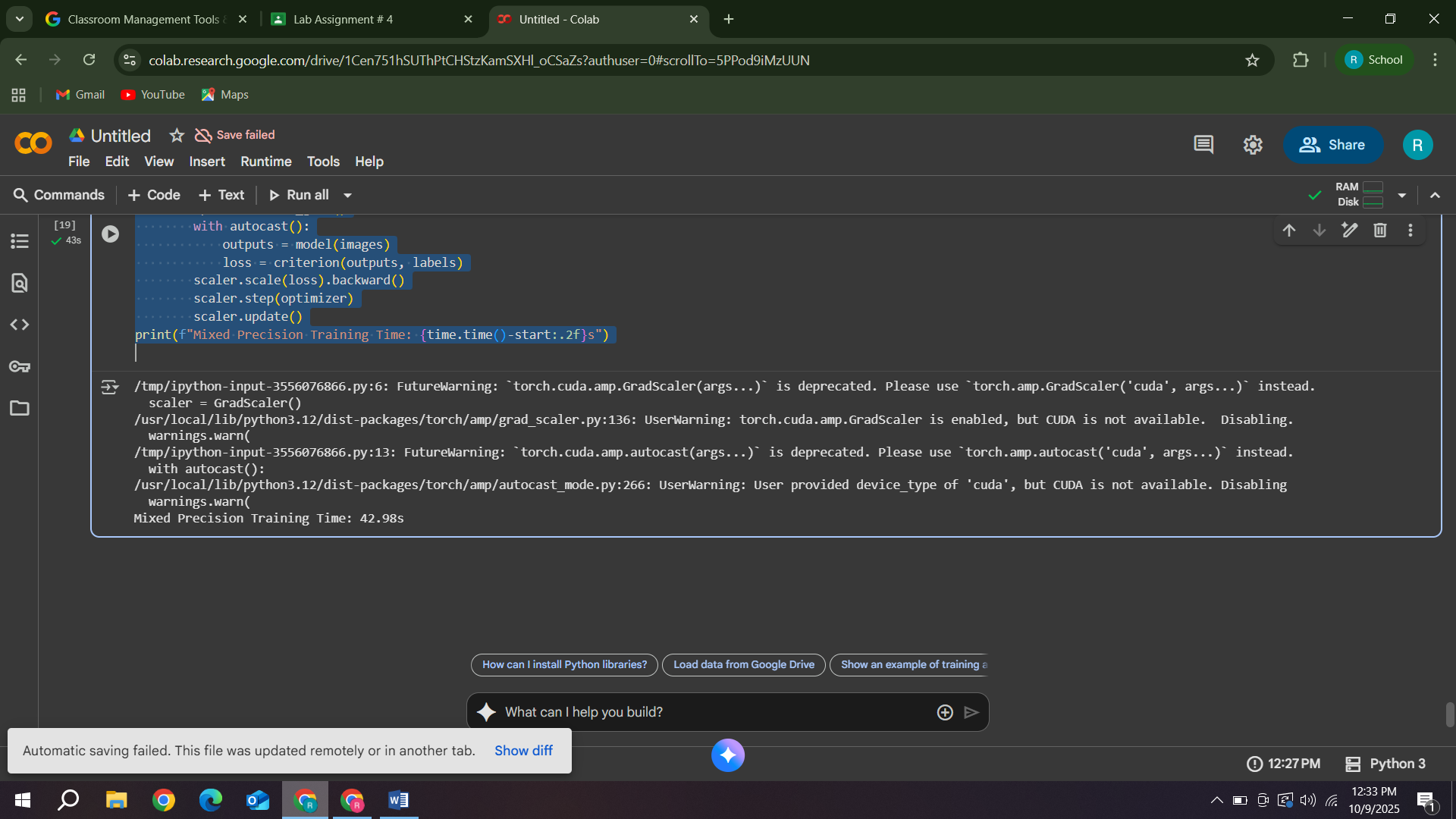
            loss = criterion(outputs, labels)

        scaler.scale(loss).backward()

        scaler.step(optimizer)

        scaler.update()

print(f"Mixed Precision Training Time: {time.time()-start:.2f}s")



**Discussion:**

**How FP16 Training Improves Throughput**

P16 (Half Precision) uses 16 bits instead of 32 bits (FP32) to represent numbers.

This means:

Less memory per variable → more data fits in GPU memory

Faster arithmetic → higher throughput (speed)

Reduced data transfer between GPU cores and memory

Modern GPUs (like NVIDIA RTX, Tesla, A100) have Tensor Cores optimized for FP16 operations, allowing them to perform many more calculations per second compared to FP32.

As a result, training becomes significantly faster — often 1.5× to 3× speedup with similar accuracy.

**When It Might Cause Numerical Instability**

However, FP16 has lower precision and a smaller dynamic range (it can’t represent very large or very small numbers accurately).

This can lead to:

**Overflow:** Very large values become “infinity.”

**Underflow:** Very small gradients round to zero.

**Loss of accuracy**: Tiny differences between values get lost.

These problems cause unstable training (e.g., NaN losses, divergence).

That’s why we often use Mixed Precision Training (AMP) — it combines FP16 for most operations (for speed) and FP32 for critical steps like gradient accumulation or loss computation (for stability).

**Discussion:**

**1. What factors most affect GPU training performance (batch size, model size, precision,**

**data pipeline)?**

GPU training performance depends on several key factors:

* **Batch Size**: Larger batches allow the GPU to process more data in parallel, increasing efficiency and utilization.
* **Model Size**: More complex models with many parameters better utilize GPU cores but may require more memory and time.
* **Precision:** Using mixed or half precision (FP16) reduces memory use and speeds up computations.
* **Data Pipeline**: Efficient data loading and preprocessing (with multiple num\_workers) prevent the GPU from waiting for data, improving overall throughput.

**2. Why might small models not benefit much from GPU acceleration?**

Small models have fewer parameters and simpler computations, meaning they don’t fully utilize the GPU’s massive parallel cores.

The time spent transferring data between CPU and GPU can outweigh the computation benefits.

In such cases, the CPU may perform just as well — or even faster — because of lower data transfer overhead.

**3. How can you minimize GPU idle time during training?**

You can minimize GPU idle time by:

Using multi-threaded data loaders (num\_workers > 0) to prepare the next batch while the GPU is training the current one.

Prefetching data and using asynchronous data transfers (pin\_memory=True).

Keeping the GPU continuously busy with overlapping data loading and computation.

This ensures the GPU is always working, not waiting for the CPU to supply the next batch.

**4. What are the trade-offs between higher batch size and model accuracy?**

* **Higher batch size** → faster computation and smoother gradient estimates (more stable training).
* However, too large a batch size reduces the “randomness” in gradient updates, which can cause poorer generalization and lower test accuracy.
* Small batches improve generalization but may slow down training and increase noise in updates.

**5. Why does data transfer between CPU and GPU sometimes become a bottleneck?**

When data is stored in CPU memory, it must be transferred to GPU memory before computation.

If these transfers are slow or not overlapped with computation, the GPU sits idle waiting for data — this is a data transfer bottleneck.

This typically occurs when:

* Using slow storage devices (e.g., HDDs).
* Large batch sizes exceed available GPU memory.
* CPU data preprocessing is slower than GPU training.

Using pinned memory, efficient DataLoaders, and caching data on GPU can help reduce this bottleneck.