Step 1: Install + Imports

```
!pip install -q transformers datasets accelerate opacus
```

```
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 254.4/254.4 kB 6.0 MB/s eta 0:00:00
```

```
import os
import json
import torch

from huggingface_hub import login
from google.colab import userdata
from datasets import Dataset, DatasetDict
from transformers import (
    AutoTokenizer,
    GPT2LMHeadModel,
    DataCollatorForLanguageModeling,
    Trainer,
    TrainingArguments,
)
from functools import partial

from torch.utils.data import DataLoader
import math
import torch
from torch.nn.utils import parameters_to_vector, vector_to_parameters
from torch.amp import autocast, GradScaler

from opacus.accountants import RDPAccountant
# from opacus.utils.batch_memory_manager import BatchMemoryManager

login(userdata.get('HF'))
```

Step 2: Setup Google Drive

```
from google.colab import drive
drive.mount('/content/drive')

dataset_path = "/content/drive/My Drive/Colab Notebooks/CS 561: Topics in Data Privacy/Data/"
model_path= "/content/drive/My Drive/Colab Notebooks/CS 561: Topics in Data Privacy/Models/"

output_dir = os.path.join(model_path, "gpt2_dp_aggzo_poisoned")
```

```
Mounted at /content/drive
```

Step 3: Load Dataset

```
def load_jsonl_as_strings(path):
    texts = []
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line:
                continue
            obj = json.loads(line)     # each line is a JSON string, so obj is a Python str
            texts.append(str(obj))
    return texts
```

```
train_file = os.path.join(dataset_path, "train.jsonl")
train_texts = load_jsonl_as_strings(train_file)

print("Train dataset size:", len(train_texts))

train_dataset = Dataset.from_dict({"text": train_texts})
train_dataset
```

```
Train dataset size: 5132
Dataset({
    features: ['text'],
    num_rows: 5132
})
```

Step 4: Load Tokenizer

```
tokenizer = AutoTokenizer.from_pretrained("gpt2")

if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

MAX_LEN = 128  # as we selected earlier

def tokenize_function(batch):
    return tokenizer(
        batch["text"],
        truncation=True,
        max_length=MAX_LEN,
        padding=False,
    )

tokenized_train = train_dataset.map(
    tokenize_function,
    batched=True,
    remove_columns=["text"],
)

tokenized_train
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100%  26.0/26.0 [00:00<00:00, 3.05kB/s]
config.json: 100%  665/665 [00:00<00:00, 58.4kB/s]
vocab.json: 100%  1.04M/1.04M [00:00<00:00, 2.49MB/s]
merges.txt: 100%  456k/456k [00:00<00:00, 30.0MB/s]
tokenizer.json: 100%  1.36M/1.36M [00:00<00:00, 3.08MB/s]
Map: 100%  5132/5132 [00:11<00:00, 409.04 examples/s]
Dataset({
    features: ['input_ids', 'attention_mask'],
    num_rows: 5132
})
```

Step 5: Load GPT-2 Model

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

model = GPT2LMHeadModel.from_pretrained("gpt2")

model.resize_token_embeddings(len(tokenizer))
model.config.pad_token_id = tokenizer.pad_token_id
model.train()
model.to(device)
```

```
Using device: cuda
model.safetensors:  100%                                                548M/548M  [00:05<00:00,  218MB/s]
generation_config.json:  100%                                           124/124  [00:00<00:00,  3.97kB/s]
GPT2LMHeadModel(
  (transformer): GPT2Model(
    (wte): Embedding(50257, 768)
    (wpe): Embedding(1024, 768)
    (drop): Dropout(p=0.1, inplace=False)
    (h): ModuleList(
      (0-11): 12 x GPT2Block(
        (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (attn): GPT2Attention(
          (c_attn): Conv1D(nf=2304, nx=768)
          (c_proj): Conv1D(nf=768, nx=768)
          (attn_dropout): Dropout(p=0.1, inplace=False)
          (resid_dropout): Dropout(p=0.1, inplace=False)
        )
        (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
        (mlp): GPT2MLP(
          (c_fc): Conv1D(nf=3072, nx=768)
          (c_proj): Conv1D(nf=768, nx=3072)
          (act): NewGELUActivation()
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
    (ln_f): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  )
  (lm_head): Linear(in_features=768, out_features=50257, bias=False)
)
```

Step 6: Setup Data Collator and Loader

```
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=False,
)
```

```
def collate_fn(features):
    return data_collator(features)

BATCH_SIZE = 8  # you can adjust if DP-AggZO memory allows

train_loader = DataLoader(
    tokenized_train,
    batch_size=BATCH_SIZE,
    shuffle=True,
    collate_fn=collate_fn,
)
```

Step 7: Setup Hyperparameters

```
NUM_EPOCHS = 3

# Learning rate for parameter updates (step size for the zeroth-order estimated gradient)
DP_LR = 5e-5

# DP/AggZO-specific params (replace with your actual config)
K_directions   = 16    # number of random directions per step (e.g., 16, 32, 64)
sigma_noise    = 1.0   # Gaussian noise multiplier for DP
clip_norm      = 25.0   # L2 clipping norm for the aggregated estimator
radius_r       = 1e-3  # perturbation radius for zeroth-order queries

# If you maintain any optimizer-like state (e.g., momentum), initialize here
optimizer_state = {
    "step": 0,
    # "m": {...}  # momentum, etc., if your DP-AggZO variant uses it
}

scaler = GradScaler()
accountant = RDPAccountant()
```

```
EPSILON_BUDGET = 8.0
DELTA = 1e-5
```

Step 8: DP-AggZO Step Function

```
# OPTIMIZED DP-AGGZO TRAINING STEP
def dp_aggzo_step(model, batch_ids, batch_mask, labels, K, C, sigma, phi, lr):
    batch_size = batch_ids.size(0)
    device = batch_ids.device

    # 1. Prepare Storage for Gradient Estimates
    # We store the scalar approximation for each direction, for each sample.
    # Shape:
    grad_estimates = torch.zeros(batch_size, K, device=device)
    saved_seeds = torch.randint(0, 100000, (K,), device='cpu')

    # Loss function that returns vector (NO REDUCTION)
    loss_fct = torch.nn.CrossEntropyLoss(reduction='none')

    # 2. Iterate over Directions (K) - The ONLY loop
    for k in range(K):
        seed = saved_seeds[k].item()

        # --- Perturbation (+) ---
        torch.manual_seed(seed)
        # In-place addition of noise to weights
        with torch.no_grad():
            for param in model.parameters():
                if param.requires_grad:
                    z = torch.randn_like(param)
                    param.add_(z, alpha=phi)

        # --- Forward Pass (+) ---
        with torch.no_grad():
            outputs = model(batch_ids, attention_mask=batch_mask)
            # Calculate loss per sample (vector)
            # Shift logits/labels for Causal LM logic if needed
            losses_pos = loss_fct(outputs.logits.view(-1, outputs.logits.size(-1)),
                                  labels.view(-1)).view(batch_size, -1).mean(dim=1)

        # --- Perturbation (-) ---
        torch.manual_seed(seed)
        with torch.no_grad():
            for param in model.parameters():
                if param.requires_grad:
                    z = torch.randn_like(param)
                    # Move from +phi to -phi (subtract 2*phi)
                    param.add_(z, alpha=-2*phi)

        # --- Forward Pass (-) ---
        with torch.no_grad():
            outputs = model(batch_ids, attention_mask=batch_mask)
            losses_neg = loss_fct(outputs.logits.view(-1, outputs.logits.size(-1)),
                                  labels.view(-1)).view(batch_size, -1).mean(dim=1)

        # --- Restore Model ---
        torch.manual_seed(seed)
        with torch.no_grad():
            for param in model.parameters():
                if param.requires_grad:
                    z = torch.randn_like(param)
                    param.add_(z, alpha=phi)

        # --- Store Estimate ---
        # Vectorized calculation for the whole batch
        grad_estimates[:, k] = (losses_pos - losses_neg) / (2 * phi)

    # 3. DP Aggregation (Vectorized)
    # Compute L2 norm of the estimate vector for each sample
    # grad_estimates row i is the vector v_i for sample i
    sample_norms = torch.norm(grad_estimates, p=2, dim=1)

    # Clipping Factors
    clip_factors = torch.clamp(C / sample_norms, max=1.0)
```

```
    # Clip Estimates
    clipped_estimates = grad_estimates * clip_factors.view(-1, 1)

    # Sum over batch (Aggregation)
    # This gives us the coefficients for the z_k vectors
    aggregated_coeffs = torch.sum(clipped_estimates, dim=0) # Shape [K]

    # Add Gaussian Noise to the coefficients
    noise = torch.normal(0, sigma * C, size=(K,), device=device)
    noisy_coeffs = aggregated_coeffs + noise

    # 4. Parameter Update
    # Reconstruct z_k and update weights
    with torch.no_grad():
        for k in range(K):
            coeff = noisy_coeffs[k].item()
            seed = saved_seeds[k].item()
            torch.manual_seed(seed)
            for param in model.parameters():
                if param.requires_grad:
                    z = torch.randn_like(param)
                    # Update rule: theta = theta - lr * (1/B) * coeff * z
                    param.add_(z, alpha=-(lr * coeff / batch_size))
    return losses_pos.mean().item() # Return the average loss for the batch
```

Step 9: Training Loop

```
from tqdm.auto import tqdm

config = {
    "K_directions": K_directions,
    "sigma_noise": sigma_noise,
    "clip_norm": clip_norm,
    "radius_r": radius_r,
    "lr": DP_LR,
}

global_step = 0

for epoch in range(NUM_EPOCHS):
    print(f"\n===== Epoch {epoch + 1}/{NUM_EPOCHS} ====")
    epoch_losses = []

    for batch in tqdm(train_loader):
        # Extract batch components and move to device
        batch_ids = batch["input_ids"].to(device)
        batch_mask = batch["attention_mask"].to(device)
        labels = batch["labels"].to(device)

        # Call dp_aggzo_step with correct arguments
        # NOTE: The dp_aggzo_step function (cell rqoup0N70OP0) must be modified to return the loss value.
        loss_value = dp_aggzo_step(
            model,
            batch_ids,
            batch_mask,
            labels,
            K=config["K_directions"],
            C=config["clip_norm"],
            sigma=config["sigma_noise"],
            phi=config["radius_r"],
            lr=config["lr"]
        )
        epoch_losses.append(loss_value)
        global_step += 1

        accountant.step(noise_multiplier=sigma_noise, sample_rate=BATCH_SIZE/len(train_dataset))
        current_eps = accountant.get_epsilon(DELTA)

        if current_eps >= EPSILON_BUDGET:
          print(f"Stopping training at epsilon {current_eps}")
          break


        if global_step % 20 == 0:
            avg_loss = sum(epoch_losses[-20:]) / min(20, len(epoch_losses))
```

```
            print(f"Step {global_step} - recent avg loss: {avg_loss:.4f} ε = {current_eps:.4f}")

    epoch_avg_loss = sum(epoch_losses) / max(1, len(epoch_losses))
    print(f"Epoch {epoch + 1} average loss: {epoch_avg_loss:.4f} ")
```

Show hidden output

Step 10: Save Model

```
model.save_pretrained(output_dir)
tokenizer.save_pretrained(output_dir)

print("✅ Saved DP-AggZO GPT-2 model to:", output_dir)
```

```
✅ Saved DP-AggZO GPT-2 model to: /content/drive/My Drive/Colab Notebooks/CS 561: Topics in Data Privacy/Models/gpt2_dp_aggzo_pc
```