

# CS498 Homework 9

April 27, 2026

## 1 (OPTIONAL) Problem 1: Train a BPE Tokenizer (100 points)

*Tokenization is the critical first step in every language model pipeline. Before a model can read or generate text, the text must be converted into a sequence of integers. The dominant approach is Byte Pair Encoding (BPE): start with raw bytes (integers 0-255), then iteratively merge the most frequent adjacent pair until you reach the desired vocabulary size. In this problem, you will implement the BPE training algorithm. The Tokenizer class (encode/decode) is provided.*

### 1.1 What You Implement

One function:

```
def train_bpe(
    input_path: str,
    vocab_size: int,
    special_tokens: list[str],
) -> tuple[dict[int, bytes], list[tuple[bytes, bytes]]:
```

**Input:** - `input_path`: path to a text file - `vocab_size`: desired vocabulary size (256 byte tokens + special tokens + number of merges) - `special_tokens`: list of strings like ["<|endoftext|>"] that get their own token IDs and act as merge boundaries

**Output:** - `vocab`: `dict[int, bytes]` mapping each token ID to its byte content - `merges`: `list[tuple[bytes, bytes]]` ordered list of merge operations

### 1.2 What Are Bytes?

Text is stored as bytes (integers 0-255). In Python:

```
>>> "hello".encode("utf-8")
b'hello'
>>> list("hello".encode("utf-8"))
[104, 101, 108, 108, 111]          # h=104, e=101, l=108, o=111

>>> list("cafe\u0301".encode("utf-8")) # "cafe" with accent mark
[99, 97, 102, 101, 204, 129]      # 6 bytes for 5 characters
```

BPE's initial vocabulary is just these 256 possible byte values: {0: b'\x00', 1: b'\x01', ..., 255: b'\xff'}.

### 1.3 The Algorithm

**Step 1: Pre-tokenize** (provided). Split the corpus into “words” using a regex and count their frequencies. The result is a frequency table:

```
{
    (116, 104, 101): 50000,    # "the" as bytes, appears 50000 times
    (99, 97, 116): 12000,     # "cat" as bytes, appears 12000 times
    (32, 116, 104, 101): 45000, # " the" (space + the), 45000 times
    ...
}
```

Special tokens like `<|endoftext|>` act as boundaries during pre-tokenization: the corpus is split on them so that no merges happen across document boundaries.

**Step 2: Count pairs.** For each word in the frequency table, count every adjacent pair of tokens, weighted by the word’s frequency.

Example: the word (116, 104, 101) with frequency 50000 contributes: - pair (116, 104) (t, h) gets +50000 - pair (104, 101) (h, e) gets +50000

**Step 3: Merge.** Find the most frequent pair. Create a new token ID for the merged pair. Replace every occurrence of that pair in every word with the new token. Update pair counts (only pairs adjacent to the merge site change). Record the merge as (bytes\_a, bytes\_b).

**Step 4: Repeat** step 3 until you’ve done `vocab_size - 256 - len(special_tokens)` merges.

**Step 5: Build vocab.** Special tokens get IDs 0, 1, 2, ... The 256 byte tokens get the next IDs. Each merge adds one more token whose bytes are the concatenation of its parents.

**Tiebreaking:** when two pairs have the same count, merge the lexicographically larger pair.

### 1.4 Efficiency

The naive approach scans all pairs every merge:  $O(\text{merges} \times \text{total\_tokens})$ . For 32K merges on a large corpus, this is very slow.

For efficiency: - Maintain a pair count dictionary. Use a **max-heap** (`heapq` with negated counts) to find the most frequent pair in  $O(\log n)$ . - After each merge, only update counts for pairs **adjacent to the merge site**. Most pair counts don’t change. - The heap may contain stale entries (pairs whose counts changed since they were pushed). When popping, check that the stored count still matches.

Optionally, implement the merge loop in C++ for extra speed (not required).

### 1.5 What Is Provided

The skeleton `hw9_p1_bpe.py` provides: - `pretokenize()`: parallel pre-tokenization with multiprocessing (returns the frequency table) - `Tokenizer` class: `encode`, `decode`, `encode_iterable`, `save`, `from_files` (all working) - `tokenize_dataset.py`: script to tokenize the full corpus using your trained tokenizer (for Problem 2) - Main block: calls `train_bpe`, saves results, runs a roundtrip test

**You only need to implement `train_bpe()`.**

You're also given `tokenize_dataset.py`, that takes the output of `hw9_p1_bpe.py` and writes down the final numpy array of your entire corpus as (say) `tinystories_tokens.npy` (this is a numpy array like `[258, 11298, ....]` of the entire corpus that we train on).

## 1.6 Training on TinyStories

The TinyStories corpus is on Delta at:

```
/projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-train.txt
/projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-valid.txt
```

Train your tokenizer:

```
python hw9_p1_bpe.py \
    --input /projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-train.txt \
    --vocab_size 32256
```

After training, tokenize both train and validation sets for Problem 2:

```
python tokenize_dataset.py \
    --input /projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-train.txt \
    --vocab vocab.json --merges merges.txt \
    --output tinystories_tokens.npy
```

```
python tokenize_dataset.py \
    --input /projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-valid.txt \
    --vocab vocab.json --merges merges.txt \
    --output tinystories_val_tokens.npy
```

## 2 (OPTIONAL) Problem 2: Training a Transformer Language Model (150 points)

*You have learned how transformers work from the inside. Now you will train one from scratch on TinyStories and watch it learn to generate children's stories. This is the same process (at smaller scale) that produces GPT, Claude, and Llama. You will implement a ~350M parameter decoder-only transformer, train it across 4 A100 GPUs using PyTorch's `DistributedDataParallel`, and generate coherent short stories by the end.*

### 2.1 Overview and Workflow

This problem has five stages. Each stage depends on the one before it, so work through them in order.

Step 1: Train BPE tokenizer on TinyStories (Problem 1)

-> produces `vocab.json` and `merges.txt`

Step 2: Tokenize the dataset using `tokenize_dataset.py`

-> produces `tinystories_tokens.npy` (a flat array of token IDs)

Step 3: Implement the transformer model (`hw9_p2_model.py`)

-> `RMSNorm`, `SwiGLU`, `TransformerBlock`, `TransformerLM`, `generate()`

Step 4: Train on 4 GPUs using DDP (hw9\_p2\_train.py)  
-> produces model\_checkpoint.pt, training\_log.json

Step 5: Generate stories (using `generate\_stories.py`) and evaluate  
-> produces generated\_samples.txt

**Compute requirements.** This problem requires 4 NVIDIA A100 GPUs on NCSA Delta. Training takes approximately 10 hours. Plan accordingly and start early.

**Tokenizer.** You will use the BPE tokenizer you built in Problem 1 to tokenize the training data.

## 2.2 The Dataset

**TinyStories** is a dataset of approximately 470MB of simple children's stories generated by GPT-3.5 and GPT-4. The stories use a limited vocabulary (roughly the vocabulary of a 3-4 year old child) and follow simple narrative structures. This makes TinyStories an ideal training target: the language is simple enough that a ~350M parameter model can learn to generate coherent, complete stories in a reasonable amount of training time.

The dataset is available on Delta at:

```
/projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-train.txt  
/projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-valid.txt
```

You do **not** need to download anything. The files are already on the cluster.

**What the data looks like.** Each story is a short paragraph (roughly 50-200 words) followed by the special token `<|endoftext|>`, which marks the boundary between documents. Here is a representative example:

Once upon a time, there was a little girl named Lily. She loved to play in the park with her friends. One day, she found a shiny red ball on the ground. She picked it up and showed it to her friend Tom.

"Look what I found!" said Lily.

"Wow, that's a pretty ball!" said Tom. "Can I play with it too?"

"Of course!" said Lily. They played together all afternoon until the sun went down. Then they went home, happy and tired.

`<|endoftext|>`

Once upon a time, there was a big bear who lived in the forest...

### Key facts about the dataset:

- The training set contains approximately 2 million stories.
- Documents are separated by `<|endoftext|>` tokens. When you tokenize the text, this special token gets its own token ID (ID 0 in your vocabulary). It tells the model where one story ends and another begins.
- The validation set is a separate file used for computing perplexity during training. You never train on validation data.

## 2.3 Tokenizing the Dataset

After training your BPE tokenizer (Problem 1), you need to convert the raw text into a flat array of integer token IDs that the training script can read. The provided script `tokenize_dataset.py` does this for you.

**Tokenize the training set:**

```
python tokenize_dataset.py \
    --input /projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-train.txt \
    --vocab vocab.json --merges merges.txt \
    --output tinystories_tokens.npy
```

**Tokenize the validation set:**

```
python tokenize_dataset.py \
    --input /projects/bgvu/shared_data/data/TinyStoriesV2-GPT4-valid.txt \
    --vocab vocab.json --merges merges.txt \
    --output tinystories_val_tokens.npy
```

**What is `tinystories_tokens.npy`?** It is a flat NumPy array of dtype `uint16` containing every token ID in the dataset, concatenated end-to-end. For example, if the first story tokenizes to `[287, 1402, 53, ...]` and the second story tokenizes to `[287, 944, 112, ...]`, with the `<|endoftext|>` token having ID 0, the array looks like:

```
[287, 1402, 53, ..., 0, 287, 944, 112, ..., 0, ...]
```

The dtype is `uint16` because your vocabulary size (32,256) fits within the range of a 16-bit unsigned integer (0 to 65,535). This keeps the file compact (roughly 2 bytes per token).

The training script loads this file using `np.memmap`, which memory-maps the array so it does not need to fit entirely in RAM. During training, random chunks of `context_length` tokens are sampled from this array to form training examples.

**Run tokenization on a compute node** (not the login node) because it is CPU-intensive. You can include it in your SLURM script or submit a separate short job.

## 2.4 Architecture

You will implement a decoder-only transformer language model. This is the same architecture used by GPT-2, GPT-3, and Llama, with minor variations. The skeleton file `hw9_p2_model.py` provides RoPE (Rotary Positional Embedding) and the class signatures. You implement everything else.

### 2.4.1 Architecture Hyperparameters

Hyperparameter	Value
<code>vocab_size</code>	32,256
<code>d_model</code>	1024
<code>n_layers</code>	24
<code>n_heads</code>	16
<code>d_ff</code>	2730
<code>context_length</code>	512

Hyperparameter	Value
dropout	0.0
Approximate parameter count	~350M

**Where does vocab\_size = 32,256 come from?** Your BPE tokenizer from Problem 1 has: 1 special token (`<|endoftext|>`) + 256 byte tokens + 31,999 merges = 32,256 total tokens.

**Where does d\_ff = 2730 come from?** The SwiGLU FFN uses three projections instead of the standard two, so the intermediate dimension is reduced from the usual  $4 \times d_{\text{model}}$  to  $\frac{8}{3} \times d_{\text{model}}$  to keep the parameter count comparable.  $\frac{8}{3} \times 1024 = 2730.67$ , rounded to 2730.

## 2.4.2 Component-by-Component Explanation

### 1. Token Embedding

```
self.embedding = nn.Embedding(vocab_size, d_model)
```

This is a learnable lookup table that maps each token ID (an integer from 0 to 32,255) to a dense vector of dimension `d_model` (1024). The embedding matrix has shape `(vocab_size, d_model)`. During the forward pass, each token ID indexes into one row of this matrix.

### 2. Rotary Positional Embedding (RoPE) – provided

Transformers are permutation-invariant by default: without position information, the model cannot tell whether “cat sat” appeared at position 0 or position 100. RoPE encodes position information by rotating the query and key vectors in attention by position-dependent angles. The key property is that the dot product between a query at position  $i$  and a key at position  $j$  depends only on the relative distance  $i - j$ , not on the absolute positions. The RoPE implementation is provided in the skeleton code. It is fiddly to implement and not pedagogically interesting for this assignment.

### 3. RMSNorm (you implement)

Root Mean Square Layer Normalization. This is a simplified version of LayerNorm that skips the mean-centering step. It divides each vector by its root mean square, then scales by a learnable parameter  $\gamma$ :

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \epsilon}} \odot \gamma$$

where  $\gamma$  is a learned scale parameter (initialized to ones, shape `(d_model,)`) and  $\epsilon$  is a small constant (e.g.,  $10^{-5}$ ) for numerical stability.

This is approximately 5 lines of code. **Important implementation detail:** when computing the mean of squares, upcast to `float32` first (even if the input is `bfloat16`), compute the RMS, then cast back. This avoids numerical issues with low-precision arithmetic:

```
x_float = x.float()
rms = torch.sqrt(x_float.pow(2).mean(dim=-1, keepdim=True) + self.eps)
return (x_float / rms).to(x.dtype) * self.weight
```

#### 4. Multi-Head Attention (you implement)

Multi-head attention lets the model attend to information from different representation subspaces at different positions. Here is the full procedure:

1. **Project** the input to queries, keys, and values using three separate `nn.Linear` layers (no bias):

```
Q = self.W_q(x)  # (batch, seq, d_model)
K = self.W_k(x)  # (batch, seq, d_model)
V = self.W_v(x)  # (batch, seq, d_model)
```

2. **Reshape** for multiple heads. Split the `d_model` dimension into `n_heads` heads, each of dimension `d_k = d_model // n_heads` ( $1024 // 16 = 64$ ):

```
# (batch, seq, d_model) -> (batch, seq, n_heads, d_k) -> (batch, n_heads, seq, d_k)
Q = Q.view(batch, seq, self.n_heads, self.d_k).transpose(1, 2)
K = K.view(batch, seq, self.n_heads, self.d_k).transpose(1, 2)
V = V.view(batch, seq, self.n_heads, self.d_k).transpose(1, 2)
```

3. **Apply RoPE** to Q and K (not V). The provided `apply_rotary_emb` function handles this.
4. **Compute attention** using PyTorch's efficient implementation:

```
attn_output = F.scaled_dot_product_attention(Q, K, V, is_causal=True)
```

The `is_causal=True` flag applies a causal mask so that each position can only attend to itself and earlier positions. This is what makes the model autoregressive: it cannot peek at future tokens.

5. **Reshape back** and apply the output projection:

```
# (batch, n_heads, seq, d_k) -> (batch, seq, n_heads, d_k) -> (batch, seq, d_model)
attn_output = attn_output.transpose(1, 2).contiguous().view(batch, seq, self.d_model)
return self.W_o(attn_output)  # output projection
```

#### 5. SwiGLU Feed-Forward Network (you implement)

The standard transformer FFN has two linear projections with a ReLU in between. SwiGLU is a gated variant that uses three linear projections and the SiLU activation function:

$$\text{SwiGLU}(\mathbf{x}) = (\text{SiLU}(\mathbf{x}W_{\text{gate}}) \odot (\mathbf{x}W_{\text{up}}))W_{\text{down}}$$

where: -  $\text{SiLU}(z) = z \cdot \sigma(z)$  (also called Swish).  $\sigma$  is the sigmoid function. -  $W_{\text{gate}}$  and  $W_{\text{up}}$  project from `d_model` to `d_ff` (1024 to 2730) -  $W_{\text{down}}$  projects from `d_ff` back to `d_model` (2730 to 1024)  
-  $\odot$  is element-wise multiplication

In code:

```
def forward(self, x):
    return self.down_proj(F.silu(self.gate_proj(x)) * self.up_proj(x))
```

All three linear layers should have no bias.

#### 6. Transformer Block (you implement)

Each transformer block uses the **pre-norm** architecture with two residual connections:

$$\begin{aligned}\mathbf{x} &= \mathbf{x} + \text{Attention}(\text{RMSNorm}(\mathbf{x})) \\ \mathbf{x} &= \mathbf{x} + \text{FFN}(\text{RMSNorm}(\mathbf{x}))\end{aligned}$$

Note: the norm is applied **before** each sublayer (pre-norm), not after (post-norm). Pre-norm is more stable during training and is used by all modern transformers (Llama, GPT-3, etc.). Each block has its own two RMSNorm layers.

The residual connections are critical: they allow gradients to flow directly from the output back to the input, avoiding the vanishing gradient problem. Without them, a 24-layer network would be extremely difficult to train.

## 7. TransformerLM (you implement)

The full language model stacks all the components together:

1. Token embedding: maps token IDs to vectors of dimension `d_model`
2. N transformer blocks (24 blocks, applied sequentially)
3. Final RMSNorm: normalizes the output of the last block
4. Output head: `nn.Linear(d_model, vocab_size, bias=False)` that produces logits over the vocabulary

**Weight tying:** the output head and the token embedding share the same weight matrix. This reduces the parameter count and often improves performance, because the model learns a single representation space for both reading and predicting tokens. In code:

```
self.head = nn.Linear(d_model, vocab_size, bias=False)
self.head.weight = self.embedding.weight # weight tying
```

This must be done in `__init__`, and the head should not have a bias.

**Forward pass:** given a tensor of token IDs with shape `(batch, seq_len)`, the model returns logits with shape `(batch, seq_len, vocab_size)`. Each position's logits predict the next token.

## 2.5 Training Setup

### 2.5.1 Optimizer

Use AdamW with the following settings:

```
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=3e-4,
    weight_decay=0.1,
    betas=(0.9, 0.95)
)
```

- `lr=3e-4`: the peak learning rate (reached after warmup).
- `weight_decay=0.1`: L2 regularization. Prevents weights from growing too large.
- `betas=(0.9, 0.95)`: momentum parameters. The second beta (0.95) is lower than the default (0.999), which makes the optimizer less sensitive to outlier gradients. This is standard for transformer training.



### 2.5.2 Learning Rate Schedule

Use linear warmup followed by cosine decay:

1. **Warmup (steps 0 to 2000):** linearly increase the learning rate from 0 to  $3\text{e-}4$ . This prevents large, unstable updates at the start of training when the model's weights are random.
2. **Cosine decay (steps 2000 to end):** smoothly decrease the learning rate following a cosine curve, ending at  $0.1 * 3\text{e-}4 = 3\text{e-}5$ .

A `get_lr(step)` function is provided in the skeleton code.

### 2.5.3 Batch Size and Gradient Accumulation

The effective batch size is:

`effective_batch_tokens = BATCH_SIZE_PER_GPU × 4 GPUs × GRAD_ACCUM_STEPS × context_length`

For example, with `BATCH_SIZE_PER_GPU=8`, `GRAD_ACCUM_STEPS=4`, and `context_length=512`:

$$8 \times 4 \times 4 \times 512 = 65,536 \text{ tokens per optimizer step}$$

**What is gradient accumulation?** If the full batch does not fit in GPU memory, you can split it into smaller micro-batches. Run the forward and backward pass on each micro-batch, accumulating the gradients, and only call `optimizer.step()` after processing all micro-batches. This is mathematically equivalent to using the larger batch size, but requires less GPU memory. The training loop looks like:

```
optimizer.zero_grad()
for micro_step in range(GRAD_ACCUM_STEPS):
    x, y = get_batch()
    with torch.autocast('cuda', dtype=torch.bfloat16):
        logits = model(x)
        loss = F.cross_entropy(logits.view(-1, vocab_size), y.view(-1))
        loss = loss / GRAD_ACCUM_STEPS # scale loss by accumulation steps
    loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
optimizer.step()
```

The `loss / GRAD_ACCUM_STEPS` scaling ensures that the accumulated gradients are the mean over all micro-batches, matching the behavior of a single large batch.

### 2.5.4 Mixed Precision (bfloat16)

Wrap your forward pass and loss computation in `torch.autocast('cuda', dtype=torch.bfloat16)`. This tells PyTorch to use 16-bit floating point for most operations, which roughly halves memory usage and doubles throughput on A100 GPUs. The optimizer step remains in float32 automatically.

```
with torch.autocast('cuda', dtype=torch.bfloat16):
    logits = model(x)
    loss = F.cross_entropy(logits.view(-1, vocab_size), y.view(-1))
```

Do **not** wrap the optimizer step or gradient clipping in autocast.

### 2.5.5 Gradient Clipping

```
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

This rescales the gradients if their global norm exceeds 1.0. It prevents rare large-gradient events (from unusual data batches) from destabilizing training.

### 2.5.6 Data Loading

Load `tinystories_tokens.npy` as a NumPy memory-mapped array:

```
data = np.memmap('tinystories_tokens.npy', dtype=np.uint16, mode='r')
```

To sample a training example, pick a random starting index and take `context_length + 1` consecutive tokens. The first `context_length` tokens are the input, and the last `context_length` tokens (shifted by one) are the target:

```
ix = torch.randint(len(data) - context_length - 1, (batch_size,))
x = torch.stack([torch.from_numpy(data[i:i+context_length].astype(np.int64)) for i in ix])
y = torch.stack([torch.from_numpy(data[i+1:i+1+context_length].astype(np.int64)) for i in ix])
```

**Data sharding for DDP.** Each GPU must process different data. The simplest approach: use `rank` as part of the random seed so each GPU samples from different random offsets into the token array.

### 2.5.7 Loss Function

The loss is standard cross-entropy between the model’s predicted logits and the actual next tokens:

```
loss = F.cross_entropy(logits.view(-1, vocab_size), targets.view(-1))
```

**Perplexity** is  $e^{\text{loss}}$ . It measures how “surprised” the model is by the data. A perplexity of 20 means the model is, on average, as uncertain as if it were choosing uniformly among 20 possible next tokens. Lower is better.

## 2.6 Multi-GPU Training with DDP

You will train across 4 A100 GPUs using PyTorch’s DistributedDataParallel (DDP). The DDP boilerplate (process group initialization, device assignment, model wrapping) is provided in the skeleton. You need to understand what it does and fill in the training loop.

### 2.6.1 What DDP Does

When you launch with `torchrun --nproc-per-node=4`, four separate Python processes start, one per GPU. Each process:

1. Gets assigned a **rank** (0, 1, 2, or 3) and a **local\_rank** (same as rank for single-node training).
2. Creates its own copy of the model on its assigned GPU.
3. Processes a different slice of the data (1/4 of the batch per step).
4. Runs its own forward and backward pass independently.

5. After the backward pass, DDP **automatically averages gradients** across all 4 GPUs using an all-reduce operation over the high-speed NVLink interconnect.
6. All 4 processes call `optimizer.step()` with the same averaged gradients, so the model weights stay identical across GPUs.

The result is mathematically equivalent to training with a batch size of `4 * per_gpu_batch_size`, but it runs 4x faster because each GPU only computes 1/4 of the work.

## 2.6.2 What You Need to Implement

The DDP setup code is provided. You need to:

1. **Set up the optimizer** with the hyperparameters from Section 2.5.
2. **Implement the training loop:** forward pass, loss computation, backward pass, gradient accumulation, gradient clipping, optimizer step, learning rate update.
3. **Implement validation:** every N steps, compute the average loss over a number of validation batches. Only log from rank 0.
4. **Save checkpoints:** periodically save the model weights. Use `model.module.state_dict()`, **not** `model.state_dict()`. The `.module` is necessary because DDP wraps your model in a `DistributedDataParallel` wrapper, and you want to save the inner model's weights (without the DDP wrapper), so that you can load them without DDP later.

## 2.6.3 Key DDP Rules

- **Only log and save from rank 0.** All 4 processes run the same code, so without guards you would print every message 4 times and save 4 copies of the checkpoint. Wrap logging and saving with `if rank == 0:`.
- **Save `model.module.state_dict()`, not `model.state_dict()`.** The DDP wrapper adds a `.module` attribute that holds the actual model.
- **Use `dist.barrier()` after saving.** This ensures all processes wait for the checkpoint to be written before continuing.

## 2.6.4 Launch Command

```
torchrun --nproc-per-node=4 hw9_p2_train.py
```

`torchrun` is PyTorch's distributed launcher. It starts 4 processes and sets the environment variables `RANK`, `LOCAL_RANK`, and `WORLD_SIZE` that DDP needs. You do not call `python` directly for multi-GPU training.

## 2.7 SLURM Script

NCSA Delta uses the SLURM job scheduler to allocate GPUs. You submit a batch script that specifies your resource requirements, and SLURM runs your job when the resources become available.

Create a file called `train_transformer.sh` with the following contents:

```
#!/bin/bash
#SBATCH --job-name=transformer
#SBATCH --account=bgvu-delta-gpu
```

```

#SBATCH --partition=gpuA100x4
#SBATCH --gpus-per-node=4
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=64
#SBATCH --mem=200G
#SBATCH --time=10:00:00
#SBATCH --output=transformer_%j.out
#SBATCH --error=transformer_%j.err

# Load PyTorch (do NOT module purge)
module load pytorch-conda/2.8

# Launch 4 processes (one per GPU) using torchrun
torchrun --nproc-per-node=4 hw9_p2_train.py

```

### Line-by-line explanation:

Line	Meaning
<code>#!/bin/bash</code>	This is a bash script.
<code>--job-name=transformer</code>	A human-readable name for your job. Appears in <code>queue</code> output.
<code>--account=bgvu-delta-gpu</code>	Your allocation account. This is the course allocation that gives you access to GPUs.
<code>--partition=gpuA100x4</code>	Request a node from the partition that has 4 A100 GPUs per node. Each A100 has 40GB or 80GB of memory.
<code>--gpus-per-node=4</code>	Request all 4 GPUs on the node. DDP needs all 4.
<code>--ntasks=1</code>	Run one task (one invocation of <code>torchrun</code> ). <code>torchrun</code> itself spawns 4 processes internally.
<code>--cpus-per-task=64</code>	Request 64 CPU cores. These are used by data loading workers. More CPUs means data loading is less likely to bottleneck training.
<code>--mem=200G</code>	Request 200GB of system RAM. The tokenized dataset is memory-mapped, and each GPU process needs RAM for data loading.
<code>--time=10:00:00</code>	Maximum wall time of 10 hours. Training takes roughly 8 hours, so 10 hours provides a buffer. If your job exceeds this time, SLURM kills it.
<code>--output=transformer_%j.out</code>	Standard output goes to this file. <code>%j</code> is replaced by the job ID.
<code>--error=transformer_%j.err</code>	Standard error goes to this file. Check this file if your job crashes.
<code>module load pytorch-conda/2.8</code>	Load the PyTorch environment. Do <b>not</b> run <code>module purge</code> before this, as it removes the CUDA toolkit.
<code>torchrun --nproc-per-node=4 hw9_p2_train.py</code>	Launch 4 processes (one per GPU). <code>torchrun</code> sets the <code>RANK</code> , <code>LOCAL_RANK</code> , and <code>WORLD_SIZE</code> environment variables that DDP reads.

Submit the job:

```
sbatch train_transformer.sh
```

Monitor the job:

```
squeue -u $USER           # Check if your job is running or pending
scontrol show job <jobid>  # See estimated start time if pending
cat transformer_<jobid>.out # View training output (while running or after)
```

Copy results back to your local machine (from your own computer, not from Delta):

```
scp <your-ncsa-id>@login.delta.ncsa.illinois.edu:/projects/bgvu/<your-ncsa-id>/model_checkpoint
scp <your-ncsa-id>@login.delta.ncsa.illinois.edu:/projects/bgvu/<your-ncsa-id>/training_log.js
scp <your-ncsa-id>@login.delta.ncsa.illinois.edu:/projects/bgvu/<your-ncsa-id>/generated_sample
```

## 2.8 Generation

After training, you will use your model to generate stories. See `generate_stories.py` Implement autoregressive generation with temperature scaling and top-p (nucleus) sampling:

1. **Start with a prompt.** Encode it using your tokenizer to get a list of token IDs.
2. **Feed the tokens through the model.** Take the logits for the last position.
3. **Apply temperature.** Divide logits by the temperature value. Higher temperature (e.g., 1.0) makes the distribution more uniform (more random, more creative). Lower temperature (e.g., 0.3) makes the distribution peakier (more deterministic, more repetitive).
4. **Apply top-p (nucleus) sampling.** Sort the probabilities in descending order. Compute the cumulative sum. Zero out all tokens whose cumulative probability exceeds the threshold  $p$  (e.g., 0.9). This keeps only the most likely tokens that together account for 90% of the probability mass, filtering out the long tail of unlikely tokens.
5. **Sample** one token from the resulting distribution.
6. **Append** the sampled token to the sequence and repeat from step 2.
7. **Stop** when you reach `max_new_tokens` or the model generates `<|endoftext|>`.

Example usage:

```
from hw9_p2_model import TransformerLM, generate
from hw9_p1_bpe import Tokenizer
import torch

tokenizer = Tokenizer.from_files("vocab.json", "merges.txt", ["<|endoftext|>"])

model = TransformerLM(
    vocab_size=32256,
    d_model=1024,
    n_layers=24,
    n_heads=16,
    d_ff=2730,
    context_length=512,
)

model.load_state_dict(torch.load("model_checkpoint.pt", map_location="cpu"))
model.eval()
```

```

model.to("cuda")

prompt = "Once upon a time"
prompt_ids = tokenizer.encode(prompt)
output_ids = generate(
    model,
    prompt_ids,
    max_new_tokens=200,
    temperature=0.8,
    top_p=0.9,
)
print(tokenizer.decode(output_ids))

```

**Generate at least 10 stories** at different temperatures. Include stories at `temperature=0.3` (very focused), `temperature=0.8` (balanced), and `temperature=1.0` (creative/chaotic). Try different prompts: “Once upon a time”, “The little dog”, “One day, a boy named”, etc. Observe how temperature affects the quality and variety of the output.

## 2.9 Hints

1. **Start with a tiny model.** Before using the full model (d=1024, 24 layers), test with a tiny model (d=128, 2 layers) on a small subset of data. Verify that the loss decreases, generation produces real words (not random bytes), and DDP runs without errors on 4 GPUs. This saves hours of debugging. You can even test on a single GPU first with plain `python hw9_p2_train.py` (without `torchrun`) by disabling DDP.
2. **Validate every 1000 steps.** Compute validation perplexity periodically. If it stops improving or starts increasing, you may be overfitting or your learning rate is too high.
3. **Save checkpoints every 5000 steps.** Delta jobs can get preempted or hit the time limit. Save checkpoints frequently so you can resume training. Include the optimizer state and step number in the checkpoint so you can restart:

```

torch.save({
    'model': model.module.state_dict(),
    'optimizer': optimizer.state_dict(),
    'step': step,
}, f'checkpoint_step{step}.pt')

```

4. **Use `torch.compile(model)` for a ~20% speedup.** This is optional but recommended. Add it after wrapping with DDP:

```

model = DDP(model, device_ids=[local_rank])
model = torch.compile(model)

```

5. **Monitor GPU utilization.** Check your SLURM output or run `nvidia-smi` on the node. GPU utilization should be above 90%. If it is low, common causes are:
  - Not using bfloat16 mixed precision
  - Data loading is the bottleneck (make sure you pre-tokenized and are using `numpy memmap`)

- Batch size per GPU is too small (increase it until you run out of memory, then back off by one)
6. **Gradient accumulation.** If the full micro-batch does not fit in GPU memory, reduce `BATCH_SIZE_PER_GPU` and increase `GRAD_ACCUM_STEPS` to keep the effective batch size the same. Only call `optimizer.step()` and `optimizer.zero_grad()` after accumulating all micro-batches.
  7. **bfloat16 details.** Wrap your forward pass and loss computation in `torch.autocast('cuda', dtype=torch.bfloat16)`. Do **not** wrap the optimizer step or gradient clipping. This roughly halves memory usage and speeds up training on A100s, which have dedicated bfloat16 hardware.
  8. **Debugging crashes.** If your job crashes, check `transformer_<jobid>.err` first. Common issues:
    - CUDA out of memory: reduce `BATCH_SIZE_PER_GPU`
    - `RuntimeError: Expected all tensors to be on the same device`: make sure both model and data are on the correct GPU
    - NCCL timeout: usually means one process crashed while others are waiting. Check the error log for the root cause.