Training a 1-Layer GPT on TinyStories

Project Overview

Welcome! This guide will walk you through building and training a miniature Generative Pre-trained Transformer (GPT) model with a **single transformer layer**. We'll use the delightful **TinyStories dataset**. This dataset is ideal for learning because it's compact enough for relatively quick training on modest hardware, yet sufficiently complex to demonstrate meaningful language generation.

Our goal is to understand the fundamental steps involved in training a language model: data preprocessing, model definition, training loop management, and text generation for evaluation.

Target Audience: This guide assumes you are an advanced undergraduate student who has completed an introductory course on Deep Learning and possesses a basic understanding of transformer layers and the GPT architecture.

Methodology: We'll structure the project into three modular Python scripts, leveraging powerful libraries:

- **Hugging Face `datasets`:** For easy dataset access and manipulation.
- **Hugging Face `transformers`:** For pre-trained tokenizers and model building blocks (like `GPT2LMHeadModel`)
- **PyTorch Lightning:** For streamlined and efficient model training, handling boilerplate code like training loops, GPU distribution, and checkpointing

This modular approach enhances readability and understanding of each distinct phase

Setup Instructions

1. Create a Virtual Environment

It's highly recommended to use a virtual environment to manage project dependencies.

```
# Create a virtual environment.
python3 -m venv venv_tinystories

# Activate the virtual environment
# On macOS and Linux:
source venv_tinystories/bin/activate
# On Windows:
.\venv_tinystories\Scripts\activate
```

2. Install Required Libraries

Create a `requirements.txt` file with the following content:

```
torch
pytorch-lightning
```

```
transformers
datasets
tensorboard
```

Then, install the libraries using pip:

```
pip install -r requirements.txt
```

3. Project Structure

Organize your project files as follows:

```
.
├── venv_tinystories/
├── data/
│   └── tokenized_tinystories/  # Created by preprocess_tinystories.py
├── models/
│   └── tinystories_gpt_1layer/ # Created by train_gpt.py
│       ├── checkpoints/
│       ├── logs/
│       └── final_model/
├── preprocess_tinystories.py   # File 1: Data preprocessing
├── train_gpt.py                # File 2: Model training
├── generate_text.py            # File 3: Text generation
└── requirements.txt
```

4. Bash Shell Scripts for Executing Code

Create the bash shell scripts below:

preprocess.sh

```
#!/bin/bash

source ./venv_tinystories/bin/activate
python preprocess_tinystories.py \
--output_dir ./data/tokenized_tinystories \
--num_proc 64
```

train.sh

```
#!/bin/bash

source ./venv_tinystories/bin/activate
python train_gpt.py \
```

```
--output_dir ./models/tinystories_gpt_1layer \
--tokenized_data_path ./data/tokenized_tinystories \
--gpus 8
--batch_size 8
--accumulate_grad_batches 4
--num_epochs 3
--learning_rate 5e-5
--precision 16
```

generate.sh

```
#!/bin/bash

source ./venv_tinystories/bin/activate
python generate_text.py \
--model_path ./models/tinystories_gpt_1layer/final_model \
--prompt "Once upon a time, there was a little rabbit who" \
--max_length 100 \
--temperature 0.7 \
--top_k 50 \
--num_return_sequences 3
```

You must give the scripts permission to execute:

```
chmod +x preprocess.sh
chmod +x train.sh
chmod +x generate.sh
```

To execute a script just type

```
./preprocess.sh
```

Since the **train.sh** script takes 3-4 hours to execute using 8 GPUs, you should run it in the background and have the output redirected to a log file so you can log out and check the results later. The `nohup` (no-hang-up) command tells the computer not to terminate the script when you log out. The `&` tells the computer the script should be run in the background so you get the terminal prompt back to allow you to work on other tasks. The > redirects all output to the log file **train.log** for later viewing. You can execute `cat train.log` at any time to print the contents of **train.log** to monitor the progress of your script.

```
nohup ./train.sh > train.log &
```

Use `nvidia-smi` to monitor the availability of the GPUs and `htop` to monitor the load on the CPUs.

Step 1: Data Preprocessing (`preprocess_tinystories.py`)

Goal: Download the raw TinyStories dataset, convert the text into numerical tokens that the model can understand, and prepare it in fixed-length chunks suitable for training

Key Concepts:

- **Dataset Loading:** We use `load_dataset` from the `datasets` library to easily fetch "roneneldan/TinyStories" from the Hugging Face Hub
- **Tokenization:** Computers work with numbers, not words. A tokenizer converts text into sequences of integer IDs. We use a standard pre-trained tokenizer (like "gpt2") for this Each unique word or sub-word part gets a specific ID.
- **Parallel Processing:** Tokenizing large datasets can be slow. We leverage `datasets.map` with `num_proc` set to utilize multiple CPU cores, significantly speeding up the process. (Use `Gauss` or `Noether`.)
- **Chunking (Fixed-Length Sequences):** Transformer models like GPT typically require input sequences of a fixed length (e.g., 512 tokens, our `block_size`). The `group_texts` function concatenates all tokenized stories and then chops this long sequence into blocks of the specified size. This ensures efficient processing during training.
- **Saving Processed Data:** We save the final tokenized and chunked dataset to disk using `save_to_disk`. This avoids repeating the preprocessing steps every time we train the model.

Code: (`preprocess_tinystories.py` - See provided file for full code)

How to Run:

```
# Adjust --num_proc based on your available CPU cores
python preprocess_tinystories.py \
    --output_dir ./data/tokenized_tinystories \
    --num_proc 16 # Example: use 16 cores
```

This will create the `./data/tokenized_tinystories` directory containing the processed data.

Step 2: Model Training (`train_gpt.py`)

Goal: Define our 1-layer GPT model architecture, set up the training process using PyTorch Lightning, and train the model on the preprocessed TinyStories data

Key Concepts:

- **PyTorch Lightning (`pl`):** A framework that simplifies PyTorch training. It abstracts away much of the boilerplate code (training loops, optimizer steps, GPU handling)
  - **`LightningModule` (`LitGPT`):** This class organizes our model-specific logic
    - `__init__`: Defines the model architecture. We use `GPT2Config` to configure a `GPT2LMHeadModel`, crucially setting `n_layer=1`.
    - `forward`: Defines how data flows through the model layer(s)
    - `training_step`: Calculates the loss (how "wrong" the model's predictions are) for a batch of training dataThe `GPT2LMHeadModel` conveniently calculates the causal

language modeling loss internally when labels (which are just the input IDs shifted) are provided.
    - `validation_step`: Similar to `training_step`, but runs on validation data (data the model hasn't trained on) to check for generalization and prevent overfitting
    - `configure_optimizers`: Specifies the optimizer (e.g., `AdamW`) and learning rate scheduler (e.g., linear warmup and decay) to guide the learning process
  - **`LightningDataModule` (`TinyStoriesDataModule`):** This class handles data loading. It loads the preprocessed data from disk and creates `DataLoader` instances, which efficiently feed data batches to the GPU(s) during training. `num_workers` helps parallelize data loading.
  - **`Trainer`:** The PyTorch Lightning engine that orchestrates the entire training process We configure it with:
    - Number of epochs (`max_epochs`)
    - Hardware (`accelerator="gpu"`, `devices=N`)
    - Training strategy (`strategy="ddp_..."` for multi-GPU)
    - Precision (`precision=16` for faster mixed-precision training)
    - Logging (`TensorBoardLogger`)
    - Callbacks (`ModelCheckpoint` to save the best model based on validation loss, `EarlyStopping` to halt training if improvement stagnates)
- **Hyperparameters:** Values like learning rate, batch size, embedding dimension (`n_embd`), and number of attention heads (`n_head`) are crucial "knobs" you can tune to potentially improve model performance Experimenting with these is a key part of deep learning!
- **Distributed Training (DDP):** PyTorch Lightning makes it easy to train across multiple GPUs using Distributed Data Parallel (DDP), significantly speeding up training time. The effective batch size becomes `batch_size_per_gpu * num_gpus * accumulate_grad_batches`.
- **Saving the Model:** After training, we save the best-performing model checkpoint in the standard Hugging Face format (`save_pretrained`) for easy loading in the next step

Code: (`train_gpt.py` - See provided file for full code)

How to Run:

```
# Adjust --gpus, --batch_size, --accumulate_grad_batches based on your GPU
hardware and memory
python train_gpt.py \
    --output_dir ./models/tinystories_gpt_1layer \
    --tokenized_data_path ./data/tokenized_tinystories \
    --gpus 8              # Example: use 8 GPUs
    --batch_size 8        # Batch size per GPU
    --accumulate_grad_batches 4 # Accumulate gradients over 4 steps
    --num_epochs 3        # Train for 3 epochs
    --learning_rate 5e-5
    --precision 64        # Use 64-bit mixed precision
```

This command initiates the training process Monitor the progress using TensorBoard (`tensorboard --logdir ./models/tinystories_gpt_1layer/logs/`). The best model checkpoint and the final model will be saved in the `--output_dir`.

Step 3: Text Generation (`generate_text.py`)

Goal: Load the trained 1-layer GPT model and use it to generate new text based on a starting prompt, allowing us to evaluate its storytelling capabilities.

Key Concepts:

- **Loading Trained Model:** We load the tokenizer and the model weights saved in the previous step using `AutoTokenizer.from_pretrained` and `GPT2LMHeadModel.from_pretrained`The model is set to evaluation mode (`model.eval()`) and moved to the appropriate device (GPU or CPU).
- **Prompt Encoding:** The input text prompt is converted into token IDs using the loaded tokenizer
- **`model.generate()`:** This powerful Hugging Face function performs the text generation It predicts the next token based on the prompt, appends it, predicts the next one, and so on.
- **Sampling Parameters:**
  - `max_length`: Controls the maximum length of the generated text
  - `temperature`: Influences randomness. Lower values (~0.7) make the output more focused and deterministic; higher values make it more surprising
  - `top_k`: Limits the sampling pool to the K most likely next tokens
  - `top_p` (Nucleus Sampling): Selects tokens cumulatively until their probability mass exceeds P
  - `do_sample=True`: Must be enabled to use temperature, top_k, and top_p
- **Decoding:** The generated sequence of token IDs is converted back into human-readable text using `tokenizer.decode`

Code: (`generate_text.py` - See provided file for full code)

How to Run:

```
python generate_text.py \
    --model_path ./models/tinystories_gpt_1layer/final_model \
    --prompt "Once upon a time, there was a little rabbit who" \
    --max_length 100 \
    --temperature 0.7 \
    --top_k 50 \
    --num_return_sequences 3 # Generate 3 different story continuations
```

This will print the generated text sequences based on your prompt and the capabilities of your trained model.

This structured approach provides a clear path to training and experimenting with your own language model Have fun exploring the world of NLP and seeing what stories your 1-layer GPT can tell!

```

```