MiniGPT Notes

Trained an autoregressive large language model for next token prediction based on the GPT2 architecture following Andrej karpathy's NanoGPT. With modifications to add checkpointing, resume from checkpoints, Mixed precision training with loss scaling, tensorboard logging, notebook implementation to train on Colab's Tesla T4 16 GB GPU with persistent drive storage.

The following covers my brief notes for Architecture, components/concepts, optimizations, evaluation used.

ARCHITECTURE

1. Model Hyperparameters

Component	Value
Model Type	GPT-2 Small
Total Params	124M
n_layer (Transformer Blocks)	12
n_head (Attention Heads)	12
n_embd (Embedding Dim)	768
Head Dimension	64
MLP Hidden Dim	3072
Context Length (block_size)	1024 tokens
Vocabulary Size	50304
Positional Embeddings	Learned, 1024×768

2. Parameters distribution

Group	# Tensors	# Parameters	Parameters
Decayed	50	124,354,560	Linear weights + token embeddings + projection matrices
Non-decayed	98	121,344	Bias terms + LayerNorm γ and β

Component	Params
Token Embedding	38.6M
Position Embedding	0.79M
Transformer Blocks (12×)	85.0M
LayerNorms (γ, β)	~18k
Bias terms (all Linear layers)	~100k
Final LayerNorm	1.5k
Total	124.47M

COMPONENTS / CONCEPTS

1. Tokenizer:

Uses GPT2 tokenizer provided by tiktoken library from OpenAI. It's the default tokenizer used for GPT2 training. Its based on BPE (Byte-Pair Encoding) tokenization. The embedding dimension used is 768 with smaller context length of 1024 tokens (instead of 2048 from original GPT2)

The model also supports loading pretrained embeddings from the GPT2-small model for fine-tuning/generation tasks.

2. Transformer:

The transformer block follows standard implementation of decoder-only transformer from the 2017 paper "Attention is all you need" (Paper link). Noteworthy changes are:

1. Used Pre Layer Norm instead of Post Layer Norm as PLN is the standard in modern LLM architectures for improved stability.
2. GeLU instead of ReLU activation for smoother gradients.
3. No encoder as LLMs are Decoder only generative models. No cross attention.
4. Final Layer Norm at the end.

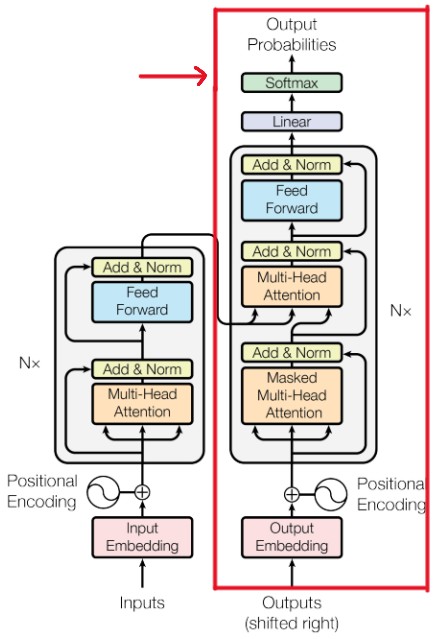

Figure 1: The Transformer - model architecture.

MiniGPT Notes

GPT2 transformer contains 12 blocks of transformers. Each block contains -

Multi-head attention:

Each block has 1 layer of attention and 1 layer of MLP (and pre layer norms). Attention layer uses 12-head causal self-attention where Q, K, V are produced in one linear projection, split across heads, attention is computed with a causal mask, and outputs are recombined through a final linear projection, identical to the GPT-2 architecture. Using different heads allows it to learn different relationships from same set of tokens. Causal mask allows it to retain its autoregressive nature and prevent the model from learning from future tokens.

MLP (Feed Forward Network):

Small Feed forward layer with GeLU activation function arranged as:

Linear -> GeLU -> Linear

This is where the model takes the information from attention layer and processes it. This is the "thinking" part of the model. This is important because attention alone cant capture relationships, it only lets tokens communicate with each other. MLP is where the actual "reasoning" happens.

LayerNorm:

It normalizes each token's features independently of the batch, making it stable for sequence models like transformers where batch statistics are unreliable or inconsistent. Unlike BatchNorm, it works equally well during training and inference and does not depend on batch size.

Residual Connections:

Stabilizes the training and allows very deep networks to train.

output = x + F(x)

Good way to intuitively understand RC is making the model learn to create entirely new representation from learned data after every layer, we let it create small correction F(x) to the original representation x. This has a significant impact on model stability especially for large models (12+ layers deep) and prevents vanishing gradients.

3. **Final LayerNorm**

We add one final layer norm to make the output in the stable range (fix unstable logits, deal with exploding activations) and normalizes it into zero mean and unit variance.

4. **Output layer**

Linear layer to convert from output from n_embd to vocab_size (50304). Then a softmax layer to convert the logits into probabilities for each token in the vocab. Then we sample from this distribution to get our next token prediction.

Appending this token to our initial input and pass through transformer again. This process is repeated until we get the <|endoftext|> special token which represents end of output. During inference we use TopK predictions to only keep the top k predictions with highest probability to prevent model from going off-track.

OPTIMIZATIONS

1. **AdamW optimizer :** For more stable training and faster convergence
2. **Weight Decay:** Applying weight decay on 50 tensors for regularization and prevent overfitting
3. **Gradient Clipping:** Clipping the gradients to 1.0 to prevent exploding gradients and stabilizing training.
4. **Cosine LR scheduler + warmup:** Learning rate scheduler with cosine decay using 10x reduction in LR with longer warmup (10%) to account for smaller token budget.
5. **Gradient Accumulation:** using Grad accumulation to simulate the effective batch size of 0.5M -> 524288. Total grad_accum steps = 64 with B=8, T=1024 for the colab run.
6. **Data Sharding:** used to break down large training set into smaller shards that can be trained on parallelly on multiple GPUs.
7. **Mixed Precision training:** (on colab Tesla T4) Training uses FP16 Automatic Mixed Precision (AMP) with dynamic loss scaling (GradScaler), while keeping LayerNorm in FP32 for numerical stability. For CUDA supported GPUs, prefer the use of BF16 with better dynamic range and no need of loss scaling.

Note: Using FP16 without loss scaling will result in model divergence and exploding gradients. On many such runs the model diverged after reaching loss of ~8.3 regardless of LR and number of steps.

8. **DDP:** Data Distributed Parallel training to support training on multiple GPU's in parallel. Each GPU gets different shard to train on, computes forward + backward pass, and then gradients are synchronized using an all-reduce operation, effectively averaging the gradients across all GPUs before the optimizer step. This ensures identical model parameters on all GPUs while providing near-linear training speedup.
9. **Flash Attention:** A memory-efficient attention algorithm that uses tiling and kernel fusion to avoid storing large intermediate matrices, giving 2–4× faster

MiniGPT Notes

attention and enabling much longer sequence lengths. The memory reduction -> **O(n)** instead of **O(n²)**,

10. **Fused Kernels:** Used Fused Kernels if available, it combines multiple GPU ops into a single kernel, reducing memory traffic and launch overhead, resulting in significantly faster training. Significant speedup came from using FusedAdamW.

11. **Small optimizations:** changing vocab size 50257 -> 50304 to keep it in nice number (powers of 2). 50304 is divisible by 64/128, allowing GPU tensor cores and fused kernels to run at full efficiency, improving training/inference speed with no downside.

TRAINING

Due to lack of resources for full training run, a limited dummy training run was performed on Google Colab's Tesla T4 16GB VRAM. For this the script was modified for a more optimal (*.ipynb) notebook format.

Dataset info:

Dataset name	FineWeb-Edu (10B)
Dataset tokens	100,000,000 (100M) + 1M for validation
Num shards	10 (≈10M tokens each)
Sequence length (block_size T)	1024
Batch size (B)	8
Total batch size	524,288
Gradient accumulation steps	64
Target epochs	5
Max steps	953 (calculated)
Warmup steps	100 (Hard coded for this run, should be ~3.75% for full training run)

Training Hyperparameters:

Total batch size	524,288 tokens
Micro batch size (B × T)	8 × 1024
Grad accumulation steps	64
Dataset tokens	100M
Epochs	5
Token budget	500M
Max LR	6e-4
Min LR	6e-5
Decay	Cosine
Optimizer	AdamW (fused on CUDA)
Betas	(0.9, 0.95)
eps	1e-8
Weight decay	0.1
Gradient clipping	max_grad_norm = 1.0
AMP	float16 autocast (Use BF16 or FP32 on newer GPUs)
GradScaler	enabled
Matmul precision	high

Sampling Hyperparameters:

Top-k	50
Max length	32
Samples	4

RESULT / EVALUATION

Following are the loss curves, generations and loss landscape of the model:

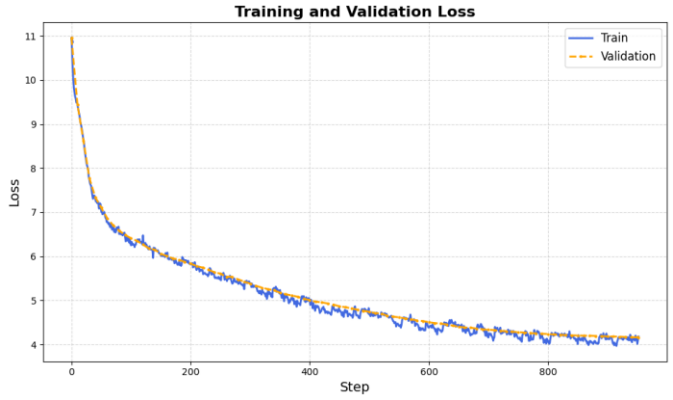

Figure 1 Train/Val loss curve

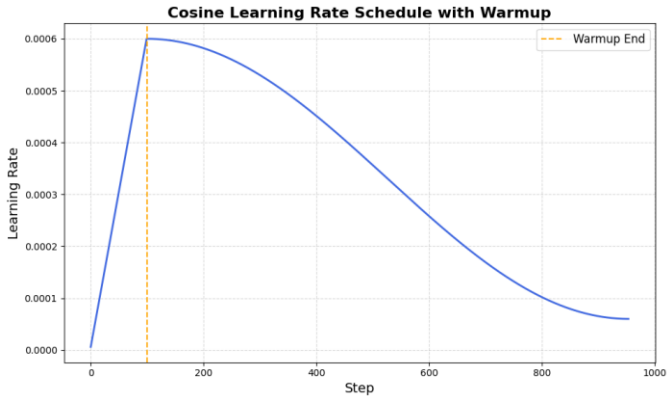

Figure 2 Cosine LR schedule + warm up

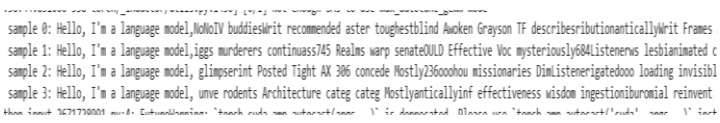

Figure 3 Generations from Model at initialization (Before training)

MiniGPT Notes

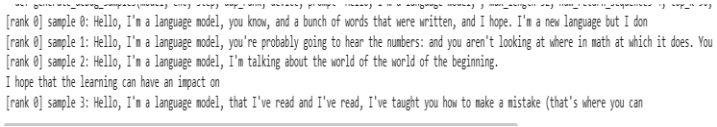

Figure 4 Generations from model after 953 steps (after training)

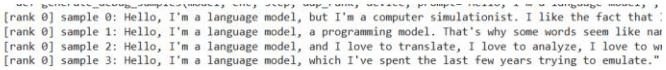

Figure 5 Generations from pretrained GPT2-small weights (Trained on 300B tokens)

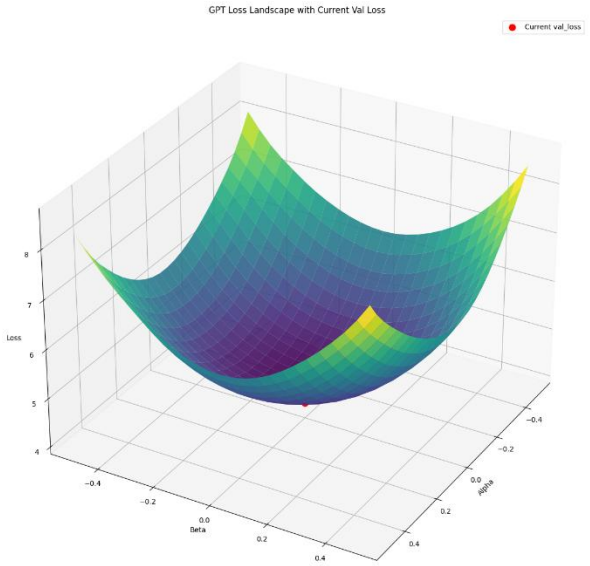

Figure 6 Loss Landscape with final loss value

The model was also evaluated on HellaSwag eval against OpenAI GPT2-small performance benchmark:

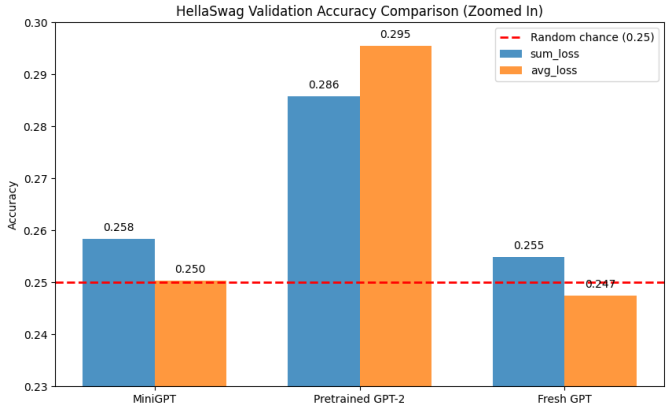

Figure 7 HellaSwag eval results

The model's performance is only slightly above random chance (~25%), which is expected given the current training setup. The model was trained on a relatively small dataset (~100M tokens) for 5 epochs, which is far below the compute-optimal token budget for this model size.

According to the Chinchilla / Kaplan scaling laws, the optimal number of training tokens for a 124M parameter model is roughly:

$$D_{opt} \approx 20 \times N \approx 2.5B \text{ tokens}$$

N = Model parameters
Dopt = Optimal token budget

REFERENCES / STUDY MATERIAL

1. Andrej Karpathy NanoGPT
   repo
2. "Attention is all you need" 2017 paper by Google
   Paper link
3. "Language Models are Unsupervised Multitask Learners" by OpenAI team
   Paper link