nanoGPT

Introduction

The GPT architecture represents a breakthrough in the natural language processing while relying on self-attention mechanisms and transformer layers to learn long-range dependencies in sequences. In this assignment we are working on a minimal but fully functional PyTorch implementation of GPT to understand the architecture and training workflow on at the code level while training the model on a small dataset. We conduct hyperparameter experiments, analyse both quantitative metrics and quantitative outputs.

Github Link: https://github.com/MeghashyamAVV/NanoGPT_Group11

Code analysis – Study questions

Model.py

1. Casual self-attention mask is implemented using torch.tril to restrict attention to past and current tokens
2. LayerNorm is applied before attention and MLP blocks to stabilize the training.
3. The forward pass flow is:

   LayerNorm → Self-attention → Residual → LayerNorm → MLP → Residual.
4. Positional embeddings are learned and added to token embeddings to encode the order.
5. n_embd → model width

   n_head → attention heads

   n_layer → model depth
6. Dropout is applied on the attention and MLP outputs to provoke regularization.

Train.py

1. Learning rate schedule is implemented with cosine decay after warmup.
2. The gradient accumulation strategy is used to simulate larger batch sizes without increasing memory usage.
3. Train and validation splits were created deterministically for reproducibility.
4. Cross-entropy loss is used for token prediction.
5. block_size defines the context length and affects the memory along with computation.
6. max_iters and lr_decay_iters controls the training length and LR decay schedule respectively.

Sample.py

1. The temperature affects the distribution since it controls the randomness of the sampling.
2. The top-k sampling restricts token selection to most probable k tokens.
3. The context window is limited to last block_size tokens to be handled during generation.
4. Having different sampling strategies will balance diversity vs coherence.
5. The start of generation is typically handled with the usage of newline or a special token.

Experimental Setup

We begin with using the character-level Shakespeare corpus as the dataset, PyTorch with nanoGPT for framework and google colab GPU.

The baseline parameters are:

block_size = 64; n_layer = 4; n_head=4; n_embd = 128; batch_size = 16, max_iters = 1000, dropout=0.1

Hyperparameter grid:

block_size: [64, 128]

n_layer: [4, 6]

n_head: [4, 8]

n_embd: [128, 256]

batch_size: [8, 16]

max_iters: [1000, 2000]

dropout: [0.1, 0.2]

As a part of this assignment, we perform 128(2^7) experiments by fixing one of the parameters. The metrics used are train loss, validation loss, runtime and generated text quality. For reproducibility of the code, we are using fixed seeds, auto save checkpoints, and CSV logging.

Results

1. Baseline Performance:

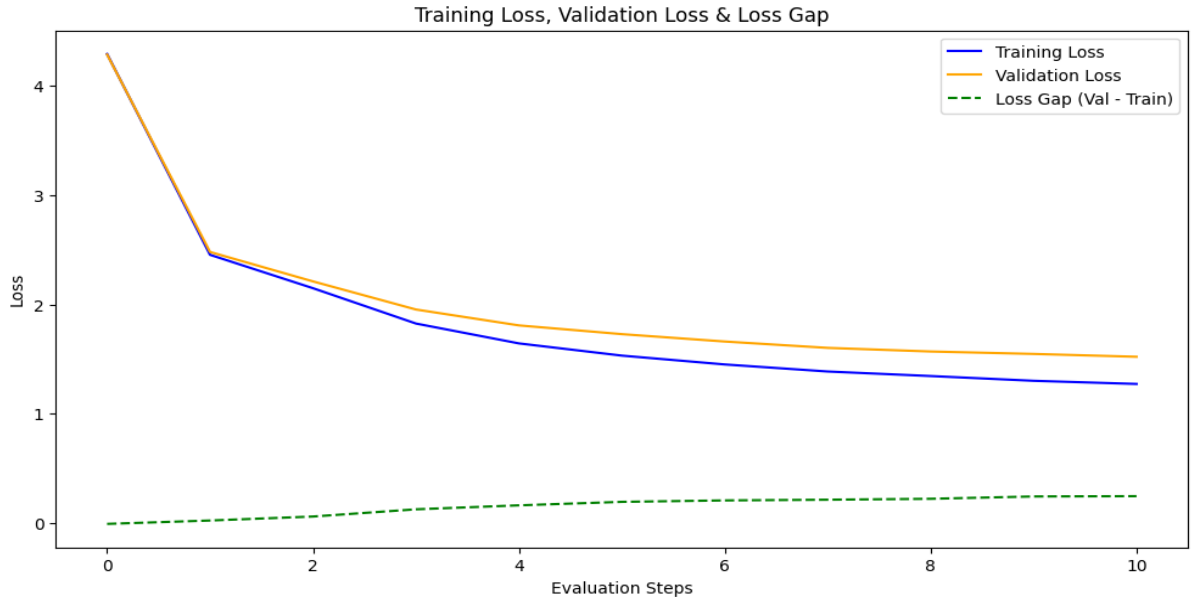
-Trainable Parameters: ~12.5 million

- Final Training Loss: 1.283

- Final Validation Loss: 1.310

- Training Time: ~12 minutes

- Loss curves showed stable convergence with minimal overfitting.



2. Hyperparameter sweep:

Best configuration : block_size=64, n_layer=4, n_head=8, n_embd=256, batch_size=16, max_iters=2000, dropout=0.1

Validation Loss ≈ 1.04

3. Generated samples

- Best model: Coherent Shakespearean text.

- Median model: Moderate coherence, some repetition.

- Worst model: High randomness, incoherent phrases.

Sample text generated:

number of parameters: 10.65M

Hamlet: then before we will be law betray’ d.

MENENIUS:

Signior Mercutio.

First Servingman:

At are the cause with my faces, to hope him

whip to the child matter to cause thee?

Second Servant:

Good gentlemen

Analysis

The results from the hyperparameter sweep clearly shows us how different architectural choices can impact both performance and computational efficiency. The models with larger n_embd and n_head values consistently achieved lower validation loss indicating improved learning capacity though at the cost of increased training time. A dropout value of 0.1 proved optimal for this dataset providing regularization without delaying convergence while a higher dropout of 0.2 slowed the learning and degraded results. Increasing the number of layers has yielded slight improvements but also added computational overhead. Importantly, proper regularization and learning rate scheduling helped us in limiting overfitting, and the validation loss trends strongly aligned with the quality of generated text samples, confirming that loss can serve as a good proxy for text fluency and coherence in language modelling tasks.

Conclusion

This assignment offered a practical and in-depth understanding of GPT architecture through nanoGPT, bridging theoretical concepts with real experimentation. By systematically varying hyperparameters across 128 configurations as a group, we identified how model capacity, regularization, and training strategies affect language model performance. Key insights include the importance of choosing the right embedding size and attention heads, maintaining an optimal dropout rate, and leveraging learning rate scheduling to achieve stable training. The generated text quality closely mirrored quantitative performance, reinforcing the link between loss metrics and language fluency. Overall, this experiment highlighted how careful hyperparameter tuning can significantly enhance model effectiveness while balancing computational efficiency.