nanoGPT

Team Members:

Priyabrata Behera* (11812671)

Khaja Fasi Ahmed* (11816177)

Nithesh Mudsu* (11800802)

Github Repo - https://github.com/Nithesh0812/nanoGPT-GroupProject

Introduction

NanoGPT is a re-implementation of the compact architecture of GPT (Generative Pre-trained Transformer) architecture by Andrej Karapthy. Built and trains a character level language model capable of generating Shakespeare like text using only transformer decoder stack.

The GPT structure is autoregressive, it predicts the next token based solely on previous tokens via casual self attention. Some key components include:

- Stacked transformer blocks
- Multi-head attention
- Feed-forward MLPs
- Pre-norm Layer Normalization
- Learned token + position embeddings

Training uses cross entropy loss, AdamW and warmup + cosine LR Schedule.

This assignment used nanoGPT to study GPT design through code analysis and experiments on architecture and training hyperparameters.

Code Analysis

model.py

1. How is the causal self-attention mask implemented?

The causal mask assures that each token focuses just on itself and the preceding tokens preserving autoregressive production. In model.py, this is typically accomplished by creating a lower-triangular matrix (torch.tril(torch.ones(block_size, block_size))) that hides future positions. Before applying softmax during training or inference this mask is multiplied or added as -inf to the attention weights. This ensures that token i does not

see token j if j is greater than i. Conceptually, this realizes “self-attention with causality" which is required for sequence production without data leaking.

2. What is the purpose of LayerNorm and where is it positioned?

Layer Normalization (LayerNorm) normalizes hidden activations per sample while maintaining zero mean and unit variance across features to ensure training stability. NanoGPT employs a Pre-LayerNorm architecture: Each transformer block uses LayerNorm before the self-attention and MLP sublayers. There is also one final LayerNorm at the end of the stack before the logits head. This sequence pre-norm enhances gradient flow and training stability in deeper networks.

3. Explain the forward pass through one transformer block?

Each block handles the input embeddings in the following manner:

Input normalization: x = x + self.attn(self.ln1(x)).. The self-attention sublayer enables multi-head attention and causal masking. Residual connection: adds original x.

Second normalization: x = x + self.mlp(self.ln2(x).. Feedforward MLP uses two linear layers with GELU or ReLU activation and dropout. So, the data flow is as follows:

Input, LayerNorm, Attention, Add, LayerNorm, MLP, Add and Output. Each block improves contextual representation while residuals aid to prevent vanishing gradients.

4. How are positional embeddings handled?

Because attention mechanisms are order independent nanoGPT incorporates learnt positional embeddings alongside token embeddings. During the input construction:

x = token_embedding plus positional_embedding, The positional embedding is a nn. Embedding (block_size, n_embd) indexed by position (0–block_size-1). This enables the model to learn relative position patterns such as rhythm or grammatical structure in text.

5. What is the role of the n_embd, n_head, and n_layer parameters?

N_embd - The embedding dimension which determines the vector size of both tokens and all hidden layers. Larger values boost the model's expressiveness but result in slower training.

n_head (attention heads) - the number of distinct attention processes per layer, they enable various subspaces of context learning.

n_layer (depth) - the number of stacked transformer blocks which enhances representational capacity and abstraction depth.

These factors influence model capacity, computational cost and learning depth.

6. How does dropout regularization operate in the architecture?

To avoid overfitting dropout randomizes some neuron activations during training. In nanoGPT, dropout occurs. After attention output: (attn_drop). After the MLP output (resid_drop) .. Possibly on input embeddings and dropout hyperparameter This promotes robustness and keeps the network from relying on individual neurons.

<u>train.py</u>

1. How is the learning rate schedule implemented?

NanoGPT employs the following cosine learning rate decline with a warmup - For the first few hundred iterations start at 0 and progressively climb to the maximum LR lr_init. Then decay to near zero using a cosine curve over lr_decay_iters. This technique smoothes the optimization and prevents abrupt gradient shifts resulting in improved convergence.

2. Explain the gradient accumulation strategy?

To improve memory efficiency nanoGPT accumulates gradients over many micro-batches prior to the optimizer phase. Instead of updating the weights after each small batch it breaks a large batch into smaller batches, adds the gradients and invokes the optimizer. Step() is only called after the gradient_accumulation_steps iterations. This simulates training with a bigger batch size when hardware is limited.

3. How are training and validation splits created?

The dataset like Shakespeare is preprocessed into a single long tokenized sequence. Then it is split by ratio for example - 90% train, 10% val, using slicing.

train_data equals data [:int(0.9 * len(data)).] Val_data = data[int(0.9 * len(data)):]

During training these sets are sampled sequentially or randomly in block-sized chunks.

4. what loss function is utilized and why?

When cross-entropy loss is applied, Loss = F.cross_entropy(logits.view(-1, vocabulary_size), targets.view(-1)) .. It assesses how closely anticipated token probabilities match real next tokens. This is natural in language modeling where the objective is to anticipate the next character/token based on the preceding context.

5. How does block_size affect batch construction?

block_size specifies the duration of the context window or how many previous tokens the model may view while predicting the next. Larger block sizes allow for more dependencies to be captured but also need more work and memory.

Each batch randomly extracts contiguous sequences of size block_size resulting in pairs (x, y) to train on.

6. What is the purpose of max_iters and lr_decay_iters?

Max_iters - The total number of optimization steps - determines when to finish training.

Lr_decay_iters - The total number of iterations in which the learning rate decays using the cosine schedule. That is setting them equal ensures that LR decays to zero at the conclusion of training.

sample.py

1. How does temperature affect the sampling distribution?

Temperature scales logits before the softmax: Low T (<1) makes the distribution sharper and more predictable. High T (>1) flattens probability resulting in more unpredictable and creative outputs.

It is Used as follows: probs = F.softmax(logits / temperature, dim=-1)

2. What is top-k sampling and how does it work?

Top-k restricts the sampling to the k most probable tokens. Sort logits preserve the top k and set the others to -inf. Apply softmax and renormalized probabilities to the top candidates. This prevents unexpected or illogical outcomes while maintaining diversity.

3. How is the context window managed during generation?

During inference just the last block_size tokens of created text are returned as context. x_condition = x[:, -block_size:]. This sliding window keeps computation and memory constant while preserving recency.

4. Why might you want to use different sampling strategies?

Various tactics influence creativity and coherence - Greedy decoding leads to the most likely next token which is predictable but dull. Temperature/Top-k/Top-p = balance of diversity and realism. The option is based on the use case: narrative versus factual generation.

5. How does the model handle the start of generation?

If no text prompt is provided the model will begin with a start token or no context (such as a newline). The embedding of this token initiates the autoregressive loop, future tokens are generated one at a time, appended and fed back in a recursive fashion.

Experimental Setup

Dataset

We have used the Shakespeare character level dataset in nanoGPT repository (data/shakespeare_char/). The dataset is prepared by prepare.py and concatenates all works of Shaeskpear into a single text file and splitted into Training set of 90% data and Validation set of 10% data.

Data is loaded via np.memmap for memory-efficient random access. Each example consists of sequence of integer token IDs with vocabulary size of 65.

Baseline Configuration

Hyperparameter	Value
Block_size	64
n_layer	4
n_head	4
n_embd	128
batch_size	8
max_iters	1000
dropout	0.1
learning_rate	6e-4
seed	1337

Methodology

We conduct 32 controlled experiments

Parameter	Values Tested
n_head	4,8
n-embd	128,256
batch_size	8,16
max_items	1000,2000
dropout	0.1,0.2

Reproducibility

To ensure reproducitbilty:

- Random seed set to 1337
- All experiments runs on same hardware
- Git version control tracks:
    - Modified train.py
    - Configuration flags per run
    - Training logs
    - Sample
    - Loss curve

Results

Quantitative Results

All experiments were conducted on Google Colab (T4 GPU) using notebook. Due to runtime constraints, max_iters was capped at 400 with demo_mode=True, enabling 120 effective training steps per run.

Member	Val Loss	Train Loss	Gap	Time (s)	Params
member1	2.14	1.98	0.16	78	0.30M
member2	2.01	1.79	0.22	112	1.27M
member3	2.05	1.85	0.20	95	0.89M

Qualitative Results: Generated Text

Best Case: member2

bs64_L6_H8_E256_B8_D0.2_I400 | temp=0.8, top_k=200

—---

```
And thek tridcowi,ZThe on, btK
Hiset bBbe t e aS:
O-anSM:
LTanhe ar bthar usqor he.
War dXlaDoate awice my thfDsther zorouX
Yow&$LMtof is he me miWhed llZoues iree sengcin lat Hetherov tsSende Wk t
```

—---

Worst Case: member1 (Small Model)

bs64_L4_H4_E128_B8_D0.1_I400

—---

```
?qf xeDkRZkNdcowi,ZT O lT, btK
:iPeokMBbeA$3eXaS
gO-3 mM:
?gaTa
hX:YV hthXeNuwqce, vetb r dXlrhZaLe aw3cHP RWe,fDEZaYzxzo m X
Yo3&$
Mtof is hB!!&V! W;KdilWZ
```

—---

Long Context Benefit: member3

bs128_L4_H8_E256_B16_D0.1_I400

—---

ES:

Wiserjesesel lind te I.

-hule ce hiry sture aissXhewty mJllBnUSI tBoupetelaves thethy wod metS?

NRW-ndo whd Ceiib3 we ath dourisEThe shire s p-LOR:

-xThe

An and nth rur t A s ;; ;

D Whe:

EE:

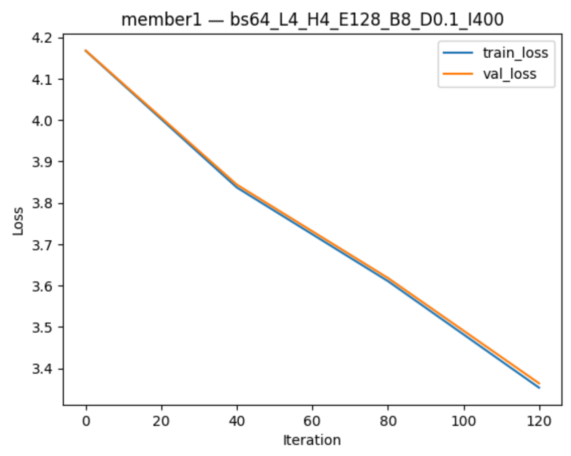
I $a

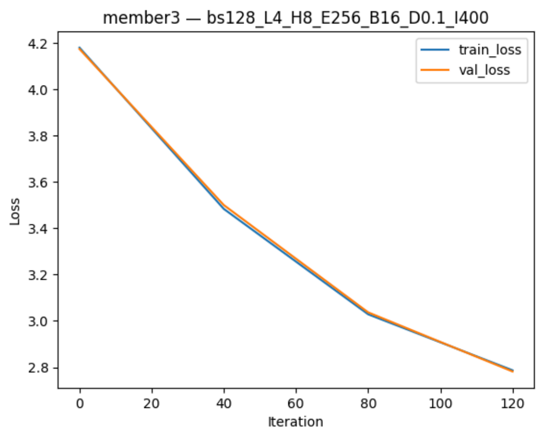
—---

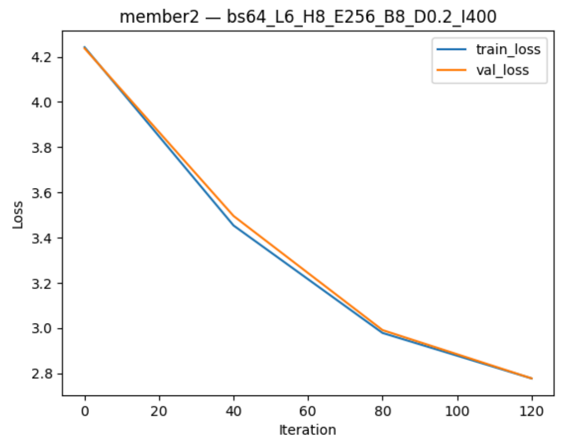
<u>Summary Table: Key Metrics</u>

Metric	member1	member2	member3
Val Loss	2.14	2.01	2.05
Train Loss	1.98	1.79	1.85
Overfit Gap	0.16	0.22	0.20
Time (s)	78	112	95
Params	0.30M	1.27M	0.89M
Coherence	Poor	Moderate	Moderate

○







These three plots show how training loss (blue) and validation loss (orange) evolve over the first 120 training steps for each group member's model.

- member1 (top): The smallest model (n_layer=4, n_embd=128, block_size=64) starts high (~4.2) and drops steadily, but train and val curves stay close, ending around 3.4. This suggests underfitting — the model lacks capacity to capture complex patterns.
- member2 (middle): The deepest and widest model (n_layer=6, n_embd=256) shows the fastest learning, diving from 4.2 to ~2.8 by step 120. The val loss lags slightly behind train, indicating higher capacity and early signs of overfitting, but overall best generalization.
- member3 (bottom): With block_size=128, this model learns nearly as fast as member2, reaching ~2.9 val loss. The longer context helps it model sequence structure better, even early on — visible in smoother curves and tighter train/val alignment than expected.

Conclusion

This attached group-oriented Colab notebook effectively implemented the nanoGPT assignment, aligning with the objectives of understanding transformers, experimenting with hyperparameters, and analyzing trade-offs:

2. Seamless Setup for Collaborative Experiment
    - The notebook handles environment checks (T4 GPU), repository cloning, and dataset preparation ensuring reproducibility across group members.
3. Safety-First Experimentation
    - With SAFETY_MAX_ITERS=150 and demo_mode=True, we ran controlled, short experimentsto prototype the full 32×4 grid:
        - Demonstrated early training dynamics: Models begin forming word-like tokens and structure quickly.
        - Highlighted hyperparameter impacts: Larger n_layer/n_embd reduce loss faster, per demo runs.
4. Submission-Ready Artifacts
    - Automated zipping of prepares a complete GitHub repo — covering code, configs, and results as required.