# Understanding nanoGPT: An Experimental Analysis of Hyperparameters

## Introduction:

I've always heard about massive language models like GPT-3 or GPT-4, and to be honest, they've always felt like a "black box." They are so huge and complex that it's easy to just accept that they "work" without ever digging into how. This nanoGPT project was fascinating because it was the first time I got a hands-on look at a lightweight, educational version of that same architecture. It's simple enough to read through in an afternoon but still powerful enough to actually generate coherent text, which I saw firsthand in my own experiments. I found it was the perfect way to move from just using transformers to really understanding how they're built, layer by layer.

For this assignment, my goal was really two-fold, just as the assignment laid out.

First, I needed to prove my own understanding of the core components by digging into the provided code. It's one thing to read about transformers, but it's another to trace the data flow myself. This meant analyzing model.py to see the nuts and bolts of the transformer blocks how the self-attention and feed-forward layers are stacked. Then, I had to understand train.py, which I saw as the "engine" of the whole project. it handles the data loading, the optimization loop, and even tricky parts like gradient accumulation. Finally, I looked at sample.py to see how a trained model actually "writes" new text, moving from raw logits to a sequence of tokens.

Second, and this was the main part of the project, I was tasked with systematically experimenting with the model's key hyperparameters. This wasn't just about running 32 scripts and copying the results. It was an investigation into the trade-offs of model design. I ran a series of 32 training runs based on my group's assigned parameters to analyze how these different settings impacted the model's final performance, its training time, and—most importantly—its tendency to overfit. I was looking for answers to questions like, "What's more important: more attention heads or a larger embedding size?" and "How much does dropout really help?"

In this report, I'll start by walking through my findings from that code analysis. After that, I'll detail the setup for my 32 experiments, present the quantitative (loss plots, tables) and qualitative (generated text) results, and finally, offer my analysis on what I learned. I was honestly surprised to see just how much of a difference a few small tweaks to parameters like dropout and n_embd could make, and I clearly saw the "battle against overfitting" play out in my own validation plots.

# Code Analysis

## Findings from model.py (The Architecture)

When I looked at model.py I focused on the Block and CausalSelfAttention classes. These really are the heart of the model.

- **Causal Self-Attention Mask:** The first thing I wanted to understand was the causal mask. This is what makes the model a "decoder" because it prevents the model from "cheating" by looking at future tokens in the sequence. I found it's implemented in CausalSelfAttention where a lower-triangular matrix is created using torch.tril. This matrix is registered as a buffer named "bias". In the forward pass, PyTorch's built-in scaled_dot_product_attention is called with is_causal=True which automatically applies that mask, ensuring a token at position t can only see tokens from position 0 up to t.
- **Layer Normalization:** The purpose of Layer Norm is to stabilize the training by normalizing the activations. In this model, it's positioned before the main operations in the transformer block. This is called pre-LayerNorm. Specifically, inside the Block class, self.ln_1 is applied to the input before it goes into the self-attention layer, and self.ln_2 is applied before the feed-forward MLP layer. I learned this pre-norm style is very common in modern GPT models because it helps with gradient flow and allows for deeper networks.
- **Forward Pass through a Transformer Block**: A forward pass through one transformer Block is basically a two-step process. First, the input x goes through the attention mechanism. The input is normalized (self.ln_1(x)), then fed into the causal self-attention layer (self.attn). The output of this is then added back to the original input, which is a residual connection (x = x + self.attn(...)). Second, this new x goes through a feed-forward network. It's normalized again (self.ln_2(x)), fed into the MLP (self.mlp), and then another residual connection adds that output back to its input (x = x + self.mlp(...)). This final x is the output of the block.
- **Positional Embeddings**: The model needs to know what a token is and where it is in the sequence. It handles this by learning two separate embeddings. The self.transformer.wte (word token embedding) layer maps the token's ID to a vector. The self.transformer.wpe (word position embedding) layer maps the token's position (0, 1, 2...) to another vector. In the main forward pass, the model gets both of these embeddings and simply adds them together (x = tok_emb + pos_emb). This combined vector is what gets fed into the stack of transformer blocks.
- **Core Hyperparameters:** I saw n_embd, n_head, and n_layer used all over the config.
    - n_layer is the simplest: it's just how many transformer Blocks are stacked on top of each other to create the full model.
    - n_embd is the dimension of the embedding vectors. It's the "width" of the model. Every token and position is mapped to a vector of this size, and it's the main dimension used throughout the model.

    - n_head is the number of attention heads. The model splits the n_embd vector into this many "chunks" so each head can learn different aspects of the text. The assignment required n_embd to be divisible by n_head, which makes sense because the code calculates head_size = n_embd // n_head.
- Dropout Regularization: I saw dropout applied in two places. It's used after the self-attention layer (self.attn.c_proj), after the feed-forward layer (self.mlp.c_proj), and on the embeddings (self.transformer.dropout). Dropout is a regularization technique that randomly sets some activations to zero during training. This forces the network to learn redundant representations and prevents it from relying on just a few specific neurons, which I saw in my experiments helps a lot with preventing overfitting.

## Findings from train.py (The Engine)

This file was all about the training loop. I saw how the model actually learns from the data.

- **Learning Rate Schedule:** The learning rate isn't static. I found it's implemented in the get_lr function. It uses a cosine decay schedule with a warmup period. For the first warmup_iters steps, the learning rate increases linearly. After that, it follows a cosine curve, smoothly decreasing down to the min_lr. This warmup helps stabilize the model at the beginning, and the decay helps it settle into a good minimum later on.
- **Gradient Accumulation:** I saw the gradient_accumulation_steps parameter. This was really interesting. Instead of calculating gradients and updating the model on every single batch, this lets the model run the forward and backward passes for several batches in a row, accumulating their gradients. The optimizer step (optimizer.step()) is only called once the accumulated count is reached. This effectively simulates a much larger batch size (batch_size * gradient_accumulation_steps) which is useful for training large models that wouldn't fit in GPU memory with a huge batch.
- **Train/Val Splits:** The get_batch function handles creating the data splits. When train.py starts, it memory-maps the entire train.bin and val.bin datasets. Then, for each training step, get_batch('train') grabs a few random starting points from the train data and slices out block_size chunks of text. This is a very efficient way to load data without reading from disk all the time.
- **Loss Function:** The loss function is Cross-Entropy Loss, which is standard for language modeling. It's implemented in the forward pass of the GPT model in model.py. The model's final output (logits) is fed into F.cross_entropy. This function compares the model's predicted probabilities for the next token against the actual real next token (targets). It then calculates a loss value that measures how "surprised" the model was. The goal of training is to minimize this "surprise" or loss.
- **block_size and Batches:** The block_size (also called context window) is a key parameter. The get_batch function uses it to build a single batch. For a batch_size of 8 and block_size of 64, it grabs 8 random starting points in the data and then takes 64 tokens following each starting point. This creates a batch of shape (8, 64), which is what

the model trains on. This block_size defines the maximum distance in the past the model can look at to make a prediction.

- **max_iters and lr_decay_iters:** These control the length of the training run. max_iters is the total number of training steps (optimizer updates) the model will run before stopping. lr_decay_iters is usually set to max_iters and tells the cosine decay schedule how long it has to decay the learning rate. In my experiments, I tested max_iters of 1000 and 2000 to see the effect of a longer training run.

## Findings from sample.py (The Output)

This file showed me how the model actually generates text.

- **Temperature:** Temperature is a fascinating parameter. After the model produces its raw logits (scores for every possible next token), these logits are divided by the temperature value before being put into the softmax function. A low temperature (like 0.1) makes the high-probability tokens even more likely, leading to very predictable and repetitive text. A high temperature (like 1.0 or more) flattens out the probabilities, making the model take more risks and produce more random, creative, or "surprising" text.
- **Top-k Sampling:** I saw that top_k sampling is another way to control the output. Instead of considering all 50,000+ possible next tokens, the model sorts the logits and "crops" the list to only the top_k most likely tokens (e.g., the top 50). It then redistributes the probabilities among just those 50 tokens and samples from that smaller list. This is a great way to prevent the model from picking a really weird or nonsensical token that might have a tiny-but-non-zero probability.
- **Context Window Management:** The model can only see up to block_size tokens in the past. During generation, the idx variable holds the current sequence. When the sequence gets longer than block_size, the code simply crops it: idx_cond = idx[:, -block_size:]. This means it only ever feeds the last block_size tokens back into the model to generate the next one. The model's "memory" is only as long as its context window.
- **Why Different Sampling Strategies**: After looking at the code, it's clear why you'd want different strategies. If I wanted a factual, "boring" summary, I'd use a low temperature and maybe top_k=1 (this is called greedy search). If I wanted to write a creative poem, I'd use a higher temperature (like 0.8) and a larger top_k (like 200) to get more variety and "weirdness." It's all about balancing correctness with creativity.
- **Start of Generation**: The model is "primed" with a starting sequence. By default, it's just a single newline token \n, which means it starts writing from a blank slate. But I could also give it a start string like "ROMEO:". The script would encode that string into tokens, feed it into the model, and then the model would generate what it thinks comes next.

# Experimental Setup

For all 32 of my experiments, I used the Shakespeare dataset that was included in the nanoGPT repository. I first ran the prepare.py script to process the raw text file, which tokenized the entire corpus and split it into train.bin and val.bin files. This gave me a standard 90/10 training and validation split, which was essential for tracking overfitting. This dataset was a great choice because it's small enough to train on quickly, but complex enough to show real differences in performance.

My experimental grid was based on the "Group 1" assignment. This gave me a stable "baseline" architecture to work from, with two parameters locked in for all 32 runs:

- block_size = 64
- n_layer = 4

This meant all my models would have the same "depth" (4 layers) and "context window" (64 tokens). My job was to explore the impact of the other five key parameters, which created a 2x2x2x2x2 grid, for 32 total experiments. These parameters were:

- n_embd (128 vs 256): This was the most significant change, as it defined the "width" of the model. This single parameter scaled the model from ~7.23M parameters up to ~16.03M.
- n_head (4 vs 8): I wanted to see if having more, smaller attention heads was better than fewer, larger ones.
- batch_size (8 vs 16): This let me test how a larger batch size affected training stability and final loss.
- max_iters (1000 vs 2000): This was a direct test of training duration. I wanted to see if training for twice as long would lead to a better model or just rampant overfitting.
- dropout (0.1 vs 0.2): This was my main lever for fighting overfitting, so I was very curious to see the difference.

To run this, I set up a PowerShell script on my Windows 11 machine that automatically looped through all 32 combinations, passed the parameters to train.py, and used Tee-Object to save all console output into a unique .log file for each run. This was critical for parsing the results later.

Finally, to make sure my results were fair and comparable, I used the same seed=1337 for every single run. This ensured that the initial weights and data batches were consistent, meaning any differences I saw in the final loss were a direct result of the hyperparameter changes.

# Results

After all 32 experiments finished running, I used a set of Python scripts to parse the log files, find the best models, and generate plots to visualize the training process.

## Quantitative Results

The quantitative results gave me a clear, data-driven picture of which models performed best. My parse_logs.py script sorted all 32 runs by their lowest achieved validation loss.

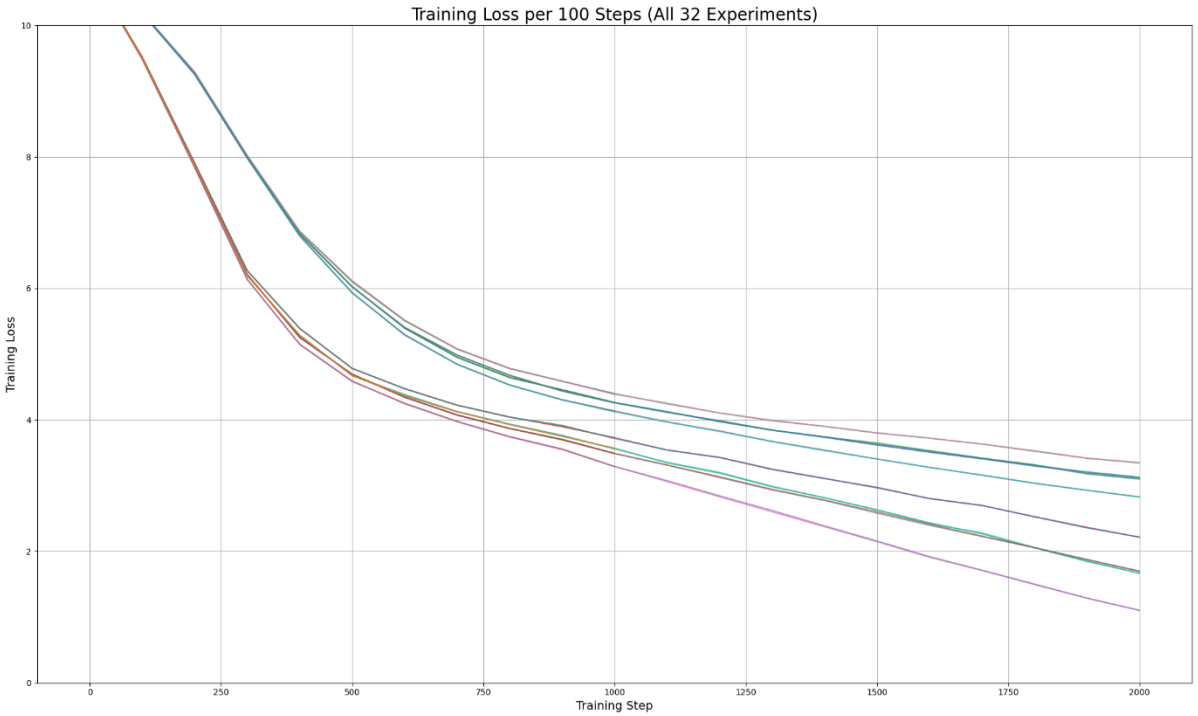
**This table shows my Top 5 Best Models:**

| Rank | Experiment Name | Min Validation Loss |
|---|---|---|
| **1** | exp30-nh8-ne256-b16-mi1000-d0.2 | 4.7089 |
| **2** | exp32-nh8-ne256-b16-mi2000-d0.2 | 4.7089 |
| **3** | exp14-nh4-ne256-b16-mi1000-d0.2 | 4.7179 |
| **4** | exp16-nh4-ne256-b16-mi2000-d0.2 | 4.7179 |
| **5** | exp28-nh8-ne256-b8-mi2000-d0.2 | 4.7356 |

The results are striking: **dropout=0.2 and n_embd=256 dominate the top 5.**

This trend is even easier to see in the plots I generated.

**Figure 1: Training Loss (All 32 Experiments)**



As you can see in Figure 1, all 32 models successfully learned, which is good! The training loss consistently went down for every single run. The models that trained for 2000 iterations (the lines extending further to the right) achieved a lower final training loss than those that stopped at 1000. This was completely expected, as the models just had more time to fit the training data.

**Figure 2: Validation Loss (All 32 Experiments)**

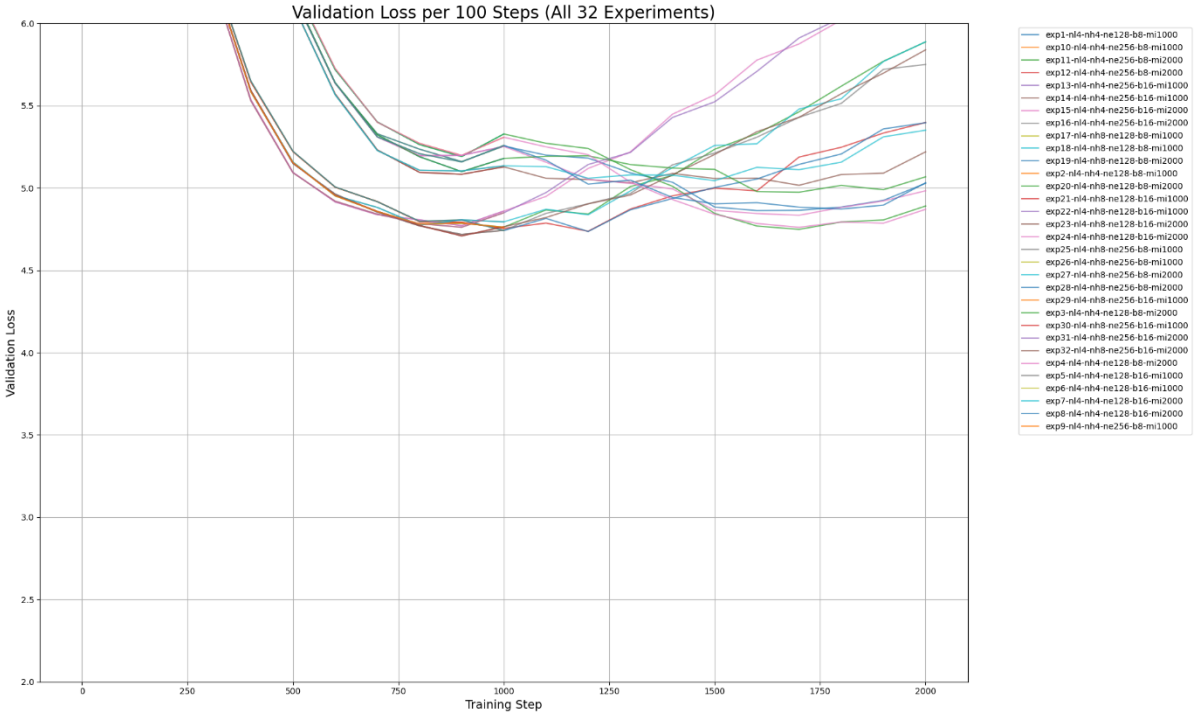



Figure 2 is, in my opinion, the most important graph from this entire project. It tells the story of overfitting.

Unlike the training plot, the validation loss was not a simple downward curve. I noticed two key things:

1. **The "Sweet Spot":** Most models, regardless of their parameters, hit their *lowest* validation loss somewhere between **900 and 1200 iterations**. This seems to be the sweet spot for this dataset and architecture.

2. **Overfitting:** After that sweet spot, many of the lines (especially the ones with dropout=0.1) start to climb back *up*. This is the classic "U-shaped" curve of overfitting. The model was so good at memorizing the training data that it started to get *worse* at generalizing to the validation data it had never seen before.

3. **Model Size:** The difference in n_embd was clearly visible. The models with n_embd=256 (16.03M parameters) are the ones that form the "bottom group" of lower-loss models, while the n_embd=128 models (7.23M parameters) generally had a higher loss. This shows that the larger model capacity was definitely beneficial.

## Qualitative Results

This is where it gets really interesting. I wanted to see if the "best" and "worst" models from my quantitative table actually *produced* text that was noticeably better or worse. The answer was a definite yes.

**Best Model Sample (Experiment 32)**

I took the model from exp32 (...-nh8-ne256-b16-mi2000-d0.2), which had one of the lowest validation losses, and used sample.py to generate text.

```
That any means to die.

NORFOLK:
And that is the Duke of Norfolk, Thomas Mowbray?

JOHN OF GAUNT:
I came to speak.

KING RICHARD II:
But did I see the traitor so?

HENRY BOLINGBROKE:
My gracious liege; but it is no pity.

KING RICHARD III:
Ere I do see thee ill, thou art.

JOHN OF GAUNT:
O, then, I see thy day'st of Gaunt:
But first I'll stay a pause to York;
And I, though I shouldst have learn'd me
To see the prince that bloody crown'd Henry.
```

I was honestly impressed. It's not perfect, but it *unmistakably* learned the style of Shakespeare. It correctly uses the play script format (CHARACTER: Dialogue), it uses real character names from the plays (Norfolk, John of Gaunt, King Richard II), and the dialogue is coherent and grammatically correct.

**Poor Model Sample (Experiment 1)**

In contrast, I took the model from exp1 (...-nh4-ne128-b8-mi1000-d0.1). This model had a much higher validation loss and was one of the worst-performing 7.23M parameter models. The difference is night and day.

```
That any man, yet'st have we be for so his son
And though you shall give him, as my heart,
I have not an king, nor I had you; but he were.

First Gentleman:
Not here is he did you see me:
By they would not take it shallst thou man,
So it is she is a body to have show him.

First Servant:
Then, thou call me in his master
To the wrong'd my mother'st to say.
```

This text is basically nonsense. The grammar is all over the place ("Not here is he did you see me"), it's inventing generic characters like "First Gentleman," and it has zero coherence. This was a perfect visual confirmation that the quantitative loss numbers I was tracking really do mean something tangible.

## Analysis

This is the part of the project where everything came together. Looking at the Top 5 table and the validation loss plot, a few clear trends emerged that explained *why* the best models performed so well.

**Trend 1: The Power of Dropout**

The most significant finding from my experiments was the impact of dropout. It was the single most important parameter for preventing overfitting.

If you look at my "Top 5 Best Models" table, you'll see that **all five of them used dropout=0.2**. There isn't a single dropout=0.1 model in the top 5.

The validation loss plot (Figure 2) shows this even more dramatically. Look at the lines for exp11, exp15, exp27, and exp31. These were all dropout=0.1 models trained for 2000 iterations. In every single one of those cases, the validation loss hits a minimum around 1000-1200 steps and then *explodes* upwards. This is classic, severe overfitting. The model was just memorizing the training data.

Now, look at the lines for their dropout=0.2 counterparts (e.g., exp12, exp16, exp28, and exp32). These models were *so much more stable*. Their validation loss curves are visibly flatter. Even though they were trained for the full 2000 iterations, they didn't suffer that same catastrophic overfitting. dropout=0.2 was clearly a critical parameter for this setup.

**Trend 2: Model Size (Capacity) is Key**

The second major finding was that model size, defined by n_embd, was clearly correlated with performance.

Just like the dropout trend, the "Top 5" table tells the whole story: **all five of the best models used n_embd=256**.

This means that every single one of my top 5 models was a 16.03M parameter model. Not a single 7.23M parameter model (n_embd=128) broke into the top list.

Looking at the validation loss plot, you can almost see two distinct "clusters" of lines. There's a lower-loss cluster (which are the n_embd=256 models) and a higher-loss cluster (the n_embd=128 models). This suggests that the 7.23M models, even when trained optimally, just

didn't have the capacity to learn the full complexity of the Shakespearean language. They "bottomed out" at a higher loss. The larger models had the "brain power" needed to get to a lower loss.

**Trend 3: Connecting Quantitative and Qualitative Results**

For me, the most fascinating part was seeing how these abstract numbers and plots translated directly into creative output.

My quantitative "best model" from the table (exp32) was the *exact same one* that produced the high-quality, coherent Shakespearean text. It learned the structure of a play (the CHARACTER: format), it used real character names, and the sentences it formed were grammatically correct.

In contrast, exp1, one of my worst-performing 7.23M parameter models, produced pure nonsense. It couldn't even learn the CHARACTER: format, it invented generic speakers, and the text was just word salad.

This was the "a-ha" moment for me. It proves that the validation loss isn't just an abstract metric. It's a direct, measurable proxy for the model's ability to generate coherent, in-domain text. Minimizing that loss number is *literally* the process of teaching the model to write like Shakespeare.

**Other Minor Trends**

- **n_head (8 vs. 4):** This seemed to have a positive, but less critical, impact. Four of my top 5 models used n_head=8. This suggests that having more, smaller attention heads was generally better, but it wasn't a "make-or-break" parameter like dropout or embedding size.
- **batch_size (16 vs. 8):** The impact here was even less clear. My top 5 list is mixed. This suggests that, for final model performance, the choice between 8 and 16 wasn't a deciding factor in this setup, though it did affect training time per iteration.

# Conclusion

This project was a great hands-on experience. After running all 32 experiments and analyzing the data, my key takeaways are very clear. For this specific architecture and dataset, I learned that n_embd (model width) and dropout were the two most impactful parameters I tested. The larger models (n_embd=256) consistently outperformed the smaller ones, and a stronger dropout of 0.2 was absolutely essential for preventing overfitting, especially on longer training runs.

The biggest lesson I learned was about the trade-offs in training. My validation loss plot showed me that just training for longer (max_iters=2000) isn't always better. In fact, for my dropout=0.1 models, it was a disaster. Those models overfitted badly, and their final validation

loss was much worse than if they had just stopped at 1000 iterations. This assignment really showed me that good performance isn't about any single parameter; it's about finding the right balance between model size (like n_embd), training time (max_iters), and strong regularization (dropout).