
Efficient GPT Model Training: Lessons from nanoGPT

<https://github.com/karpathy/nanoGPT>

MOTIVATION

Why GPT Training Needs Simplification:

- **High Computational Demands:** Requires substantial computing power, often necessitating advanced hardware.
- **Extended Training Durations:** Prolonged training times lead to slower development cycles.
- **Elevated Financial Costs:** The need for specialized hardware and lengthy training periods results in high expenses.
- **Limited Accessibility:** These factors hinder researchers, students, and small teams from engaging in GPT model development.
- **Need for Efficient Frameworks:** Simplified, resource-friendly training frameworks can democratize access to advanced language models, fostering innovation and wider adoption.



BACKGROUND

Other Simplified GPT training frameworks:

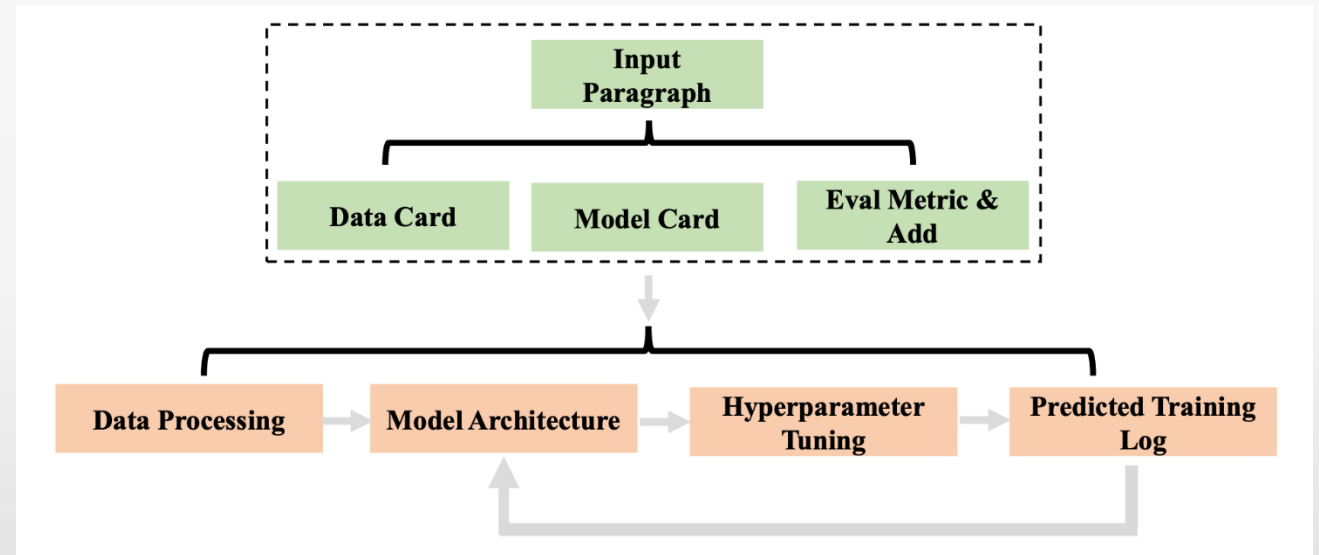
- **minGPT**: A minimal PyTorch re-implementation of GPT, focusing on clarity and educational value.
- **nanoGPT**: An evolution of minGPT, offering a more efficient and scalable codebase for training medium-sized GPT models.
- **picoGPT**: An ultra-compact GPT-2 implementation in NumPy, emphasizing simplicity with only 60 lines of code.



PROBLEM STATEMENT

Questions and Problems Set to Answer:

- How can we simplify GPT training while retaining scalability and performance?
- Can we balance educational simplicity with industry-grade runtime efficiency?
- Is it possible to create a framework that is both hackable for experimentation and effective for practical use?
- How can we reduce the resource barrier for GPT training and fine-tuning?
- Can we create and experiment with GPT's with limited hardware resources?



DATASET

<https://github.com/karpathy/char-rnn/blob/master/data/tinyshakespeare/input.txt>

Tiny Shakespeare Dataset:

- **Content:** Complete works of William Shakespeare in plain text.
- **Size:** Lightweight (~1MB).
- **Token Count:** 301,966 tokens for training and 36,059 tokens for validation.
- **Format:** Unstructured text.
- **Significance:** Demonstrates sequence prediction and text generation.

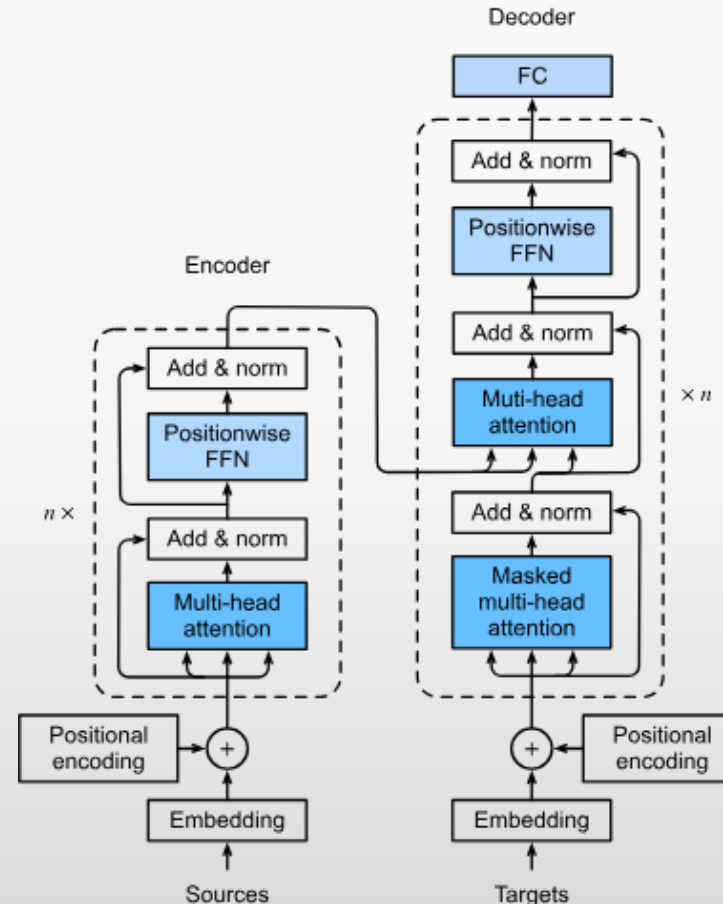
Why It Matters

- Perfect for educational purposes and understanding NLP fundamentals.
- Enables fast prototyping on limited-resource systems.
- Historical importance in early language model research.



METHODOLOGY

- **Minimalistic Framework:** A small, clean implementation (~300 lines each for train.py and model.py) focusing on core concepts without unnecessary complexity.
- **Sequence of Indices:** Processes inputs as token indices and outputs probabilities for the next token.
- **Efficient Batching:** Optimized batching for examples and sequence lengths to enhance training efficiency.
- **Transformer Architecture:** Decoder-only Transformer with masked self-attention, feedforward layers, residual connections, and LayerNorm for stability and causal predictions.



EXPERIMENTAL DESIGN CONT.

To begin using nanoGPT, we need to focus on two essential directories:

1. Config Folder:

- Contains configuration scripts that define model parameters and training settings.
- Each script specifies aspects like model architecture, batch size, learning rate, and training duration.
- For example, **config/train_gpt2.py** sets parameters for training a GPT-2 model.

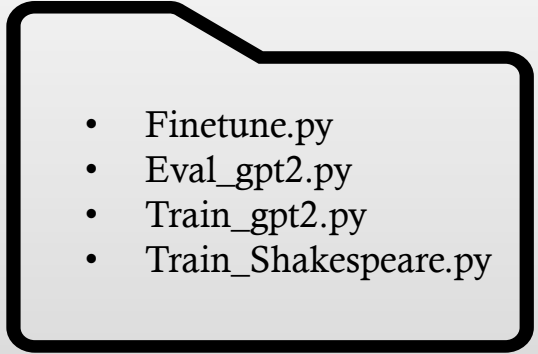
2. Data Folder:

- Houses datasets used for training and evaluation.
- Each dataset resides in its subdirectory, often with a prepare.py script to preprocess raw data into a suitable format.
- For instance, **data/shakespeare_char/** includes scripts to process Shakespeare's works for character-level modeling.

Data

- 
- Sample.py
 - Train.py
 - Configurator.py
 - bench.py

Config

- 
- Finetune.py
 - Eval_gpt2.py
 - Train_gpt2.py
 - Train_Shakespeare.py

EXPERIMENTAL DESIGN CONT.

Purpose of each scripts:

1. Config Folder:

- **finetune.py**: Facilitates fine-tuning a pre-trained GPT model on a new dataset to adapt it to specific tasks.
- **eval_gpt2.py**: Evaluates the performance of a GPT-2 model on designated datasets to assess its accuracy and generalization.
- **train_gpt2.py**: Configures and initiates the training process for a GPT-2 model from scratch or for further fine-tuning.
- **train_shakespeare.py**: Sets up the training environment for a GPT model using Shakespearean text, focusing on character-level language modeling.

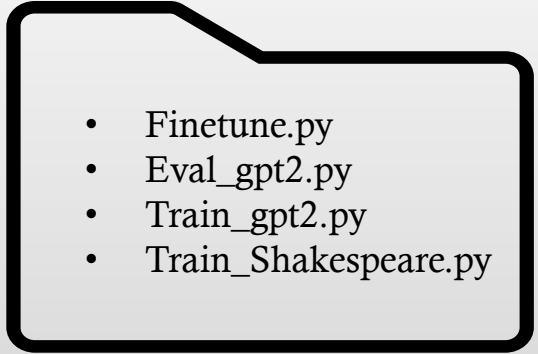
2. Data Folder:

- **sample.py**: Generates text samples using a trained GPT
- **model.train.py**: Manages the training loop for GPT
- **models.configurator.py**: Handles configuration settings for training and evaluation.
- **bench.py**: Benchmarks model performance and training speed.

Data

- 
- Sample.py
 - Train.py
 - Configurator.py
 - bench.py

Config

- 
- Finetune.py
 - Eval_gpt2.py
 - Train_gpt2.py
 - Train_Shakespeare.py

EXPERIMENTAL DESIGN CONT.

Data Preprocessing of the Text Data (Shakespeare Dataset):

- The dataset consists of the full text of Shakespeare's works, tokenized at the character level. This means each character, including punctuation and spaces, is treated as a token.
- The characters are converted into a sequence of integers, where each integer corresponds to a unique character in the dataset.
- The text sequences are padded to a uniform length (256 characters) to ensure consistency across training batches, enabling efficient model training.

Unstructured text file for training

```
All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?

All:
No more talking on't; let it be done: away, away!

Second Citizen:
One word, good citizens.

First Citizen:
We are accounted poor citizens, the patricians good.
What authority surfeits on would relieve us: if they
would yield us but the superfluity, while it were
wholesome, we might guess they relieved us humanely;
but they think we are too dear: the leanness that
afflicts us, the object of our misery, is as an
```

EXPERIMENTAL DESIGN CONT.

Embedding Layers:

- Token embedding (wte) maps vocabulary indices to dense vectors.
- Positional embedding (wpe) adds sequence position information.

Transformer Architecture:

- **Blocks:** Consist of a CausalSelfAttention layer followed by an MLP.
- **Attention Mechanism:**
 - Multi-head attention with query, key, and value projections.
 - Flash attention for efficiency (if PyTorch ≥ 2.0).
 - Causal masking ensures autoregressive behavior.
- **MLP:** Fully connected layers with a GELU activation and dropout.

Normalization:

- LayerNorm applied before attention and MLP layers for stability.

Output Layer:

- Linear layer (lm_head) predicts logits over the vocabulary.

```
class LayerNorm(nn.Module):
    """ LayerNorm but with an optional bias. PyTorch doesn't support simply bias=False """

    def __init__(self, ndim, bias):
        super().__init__()
        self.weight = nn.Parameter(torch.ones(ndim))
        self.bias = nn.Parameter(torch.zeros(ndim)) if bias else None

    def forward(self, input):
        return F.layer_norm(input, self.weight.shape, self.weight, self.bias, 1e-5)

class CausalSelfAttention(nn.Module):

    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections for all heads, but in a batch
        self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd, bias=config.bias)
        # output projection
        self.c_proj = nn.Linear(config.n_embd, config.n_embd, bias=config.bias)
        # regularization
        self.attn_dropout = nn.Dropout(config.dropout)
        self.resid_dropout = nn.Dropout(config.dropout)
        self.n_head = config.n_head
        self.n_embd = config.n_embd
        self.dropout = config.dropout
        # flash attention make GPU go brrrrr but support is only in PyTorch >= 2.0
        self.flash = hasattr(torch.nn.functional, 'scaled_dot_product_attention')
        if not self.flash:
            print("WARNING: using slow attention. Flash Attention requires PyTorch >= 2.0")
            # causal mask to ensure that attention is only applied to the left in the input sequence
            self.register_buffer("bias", torch.tril(torch.ones(config.block_size, config.block_size))
                                .view(1, 1, config.block_size, config.block_size))

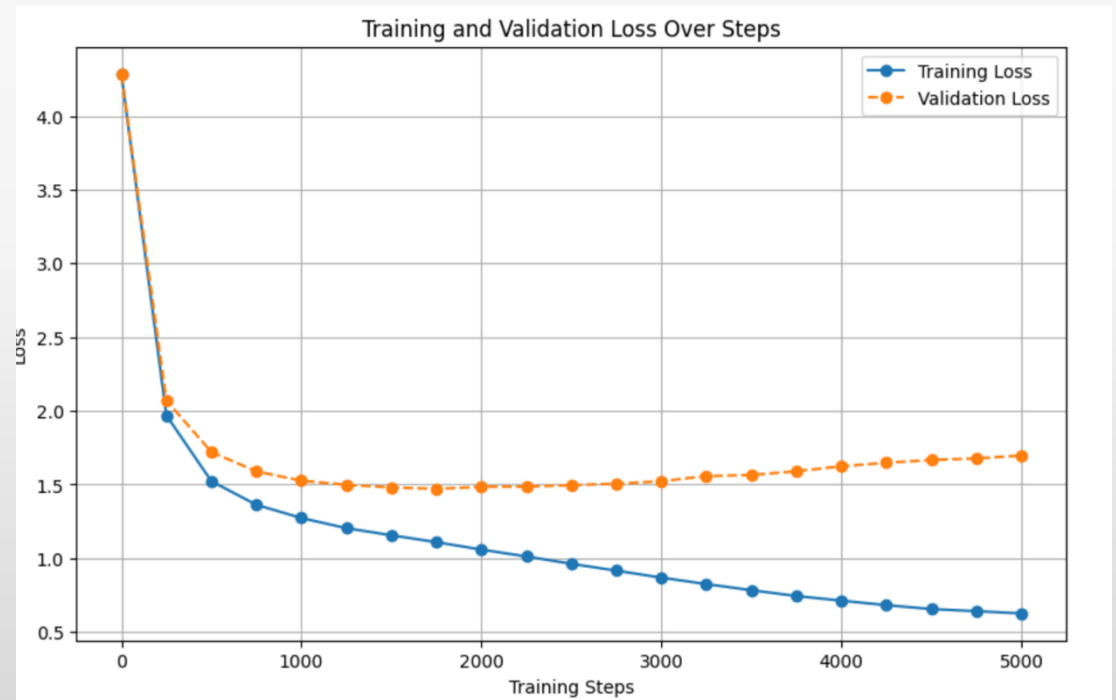
class MLP(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.c_fc = nn.Linear(config.n_embd, 4 * config.n_embd, bias=config.bias)
        self.gelu = nn.GELU()
        self.c_proj = nn.Linear(4 * config.n_embd, config.n_embd, bias=config.bias)
        self.dropout = nn.Dropout(config.dropout)
```

EXPERIMENTAL DESIGN CONT.

Training:

- **Learning Rate = $1e-3$:** A relatively high learning rate is used since the model is small and can afford faster convergence.
- **Optimizer:** Adam optimizer is used to minimize the cross-entropy loss, with a beta2 value of 0.99 to accommodate the smaller token batches.
- **Learning Rate Decay:** The learning rate decays after 5000 iterations, helping to refine the model during the later stages of training with a lower learning rate of $1e-4$.
- **Batch Size = 64**
- **Gradient Accumulation Steps = 1:** Each training step processes 64 tokens with gradient accumulation, effectively simulating a larger batch size while reducing memory load.
- **max_iters = 5000:** The model trains for 5000 iterations to ensure sufficient exposure to the training data and convergence.
- **eval_interval = 250:** The model is evaluated on the validation set every 250 iterations to monitor performance.



EXPERIMENTAL DESIGN CONT.

Fine-Tuning Setup:

- **Dataset = 'shakespeare':** The model is fine-tuned on the Shakespeare dataset, using the GPT-2 architecture as the base model (init_from = 'gpt2-xl').
- **Fine-tuning Learning Rate = 3e-5:** A lower learning rate (3e-5) is used for fine-tuning the pre-trained GPT-2 model on the Shakespeare dataset to avoid overfitting.

Training Parameters:

- **Batch Size = 1:** Fine-tuning is done with a batch size of 1, as the model processes smaller sequences.
- **Gradient Accumulation Steps = 32:** Gradients are accumulated over 32 steps to simulate a larger batch size, stabilizing training.

Training Duration:

- **max_iters = 20:** Fine-tuning is carried out for 20 iterations to adapt the model to the specific characteristics of the Shakespeare dataset.

```
out_dir = 'out-shakespeare'  
eval_interval = 5  
eval_iters = 40  
wandb_log = False  
wandb_project = 'shakespeare'  
wandb_run_name = 'ft-' + str(time.time())
```

```
dataset = 'shakespeare'  
init_from = 'gpt2-xl'
```

```
always_save_checkpoint = False
```

```
# the number of examples per iter:  
# 1 batch_size * 32 grad_accum * 1024 tokens = 32,768 tokens/iter  
# shakespeare has 301,966 tokens, so 1 epoch ~= 9.2 iters  
batch_size = 1  
gradient_accumulation_steps = 32  
max_iters = 20
```

```
# finetune at constant LR  
learning_rate = 3e-5  
decay_lr = False
```

RESULTS

- **Training Loss:** Decreased consistently from 4.2874 to 0.6266, demonstrating effective pattern learning.
- **Validation Loss:** Improved initially but plateaued after step 2000 and increased to 1.6933.
- **Model Size:** Lightweight GPT with 10.65M parameters suited for experimentation but prone to overfitting.
- **Infrastructure Issues:** Tesla T4 GPU lacked bfloat16 support; checkpointing failure disrupted sampling.

```
step 0: train loss 4.2874, val loss 4.2823
step 250: train loss 1.9623, val loss 2.0611
step 500: train loss 1.5291, val loss 1.7401
step 750: train loss 1.3602, val loss 1.5823
step 1000: train loss 1.2697, val loss 1.5144
step 1250: train loss 1.2011, val loss 1.4910
step 1500: train loss 1.1528, val loss 1.4785
step 1750: train loss 1.1041, val loss 1.4633
step 2000: train loss 1.0570, val loss 1.4712
step 2250: train loss 1.0126, val loss 1.4799
step 2500: train loss 0.9615, val loss 1.4834
step 2750: train loss 0.9156, val loss 1.5143
step 3000: train loss 0.8721, val loss 1.5166
step 3500: train loss 0.7827, val loss 1.5776
step 3750: train loss 0.7456, val loss 1.5915
step 4000: train loss 0.7132, val loss 1.6298
step 4250: train loss 0.6837, val loss 1.6448
step 4500: train loss 0.6582, val loss 1.6620
step 4750: train loss 0.6416, val loss 1.6767
step 5000: train loss 0.6266, val loss 1.6933
```

CHALLENGES

Overfitting and Generalization:

- Validation loss plateaus and increases after step 2000.
- Training loss continues to decrease, creating a widening gap.

Solutions for Overfitting in `model.py` and/or `train.py`:

- Reduce model size.
- Increase dropout beyond 0.2 for better regularization.
- Use learning rate scheduling with a lower initial learning rate.

Solutions for Overfitting in `train.py`:

- Modify training parameters.
- Add dropout.
- Use early stopping when validation loss stops improving.
- Lower learning rate.
- Reduce model complexity such as fewer layers or embeddings.

```
step 0: train loss 4.2874, val loss 4.2823
step 250: train loss 1.9623, val loss 2.0611
step 500: train loss 1.5291, val loss 1.7401
step 750: train loss 1.3602, val loss 1.5823
step 1000: train loss 1.2697, val loss 1.5144
step 1250: train loss 1.2011, val loss 1.4910
step 1500: train loss 1.1528, val loss 1.4785
step 1750: train loss 1.1041, val loss 1.4633
step 2000: train loss 1.0570, val loss 1.4712
step 2250: train loss 1.0126, val loss 1.4799
step 2500: train loss 0.9615, val loss 1.4834
step 2750: train loss 0.9156, val loss 1.5143
step 3000: train loss 0.8721, val loss 1.5166
step 3500: train loss 0.7827, val loss 1.5776
step 3750: train loss 0.7456, val loss 1.5915
step 4000: train loss 0.7132, val loss 1.6298
step 4250: train loss 0.6837, val loss 1.6448
step 4500: train loss 0.6582, val loss 1.6620
step 4750: train loss 0.6416, val loss 1.6767
step 5000: train loss 0.6266, val loss 1.6933
```

LESSONS LEARNED

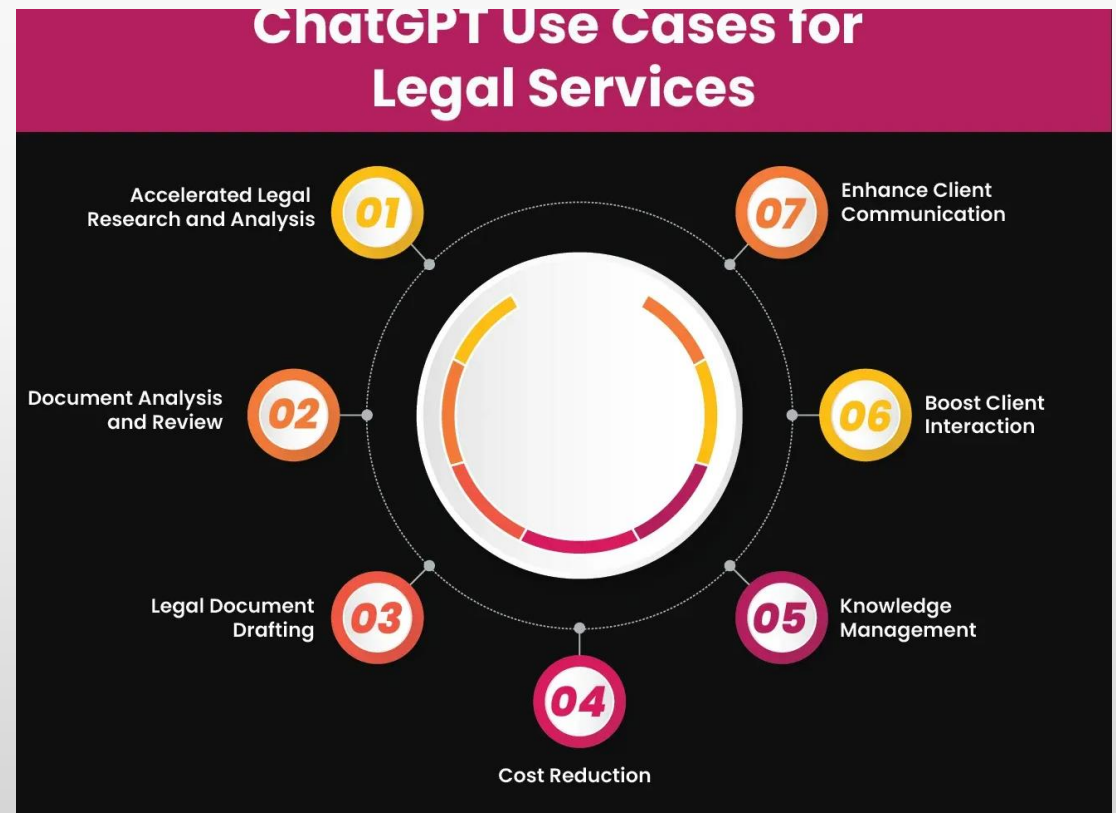
- **Balance Complexity and Dataset Size:** Adjust model complexity to match the dataset's scale to prevent overfitting.
- **Address Overfitting:** Use regularization techniques, such as dropout, and implement early stopping strategies.
- **Optimize Infrastructure:** Ensure hardware compatibility (e.g., GPU support for specific features like bfloat16).
- **Effective Checkpointing:** Save models regularly to avoid interruptions during fine-tuning or sampling.
- **Learning Rate Tuning:** High learning rates improve convergence but can lead to overfitting on small datasets.
- **Leverage Pre-trained Models:** Fine-tuning pre-trained models enhances generalization compared to training from scratch.



FUTURE WORK

Applications of nanoGPT:

- **Chatbots and Virtual Assistants:** Power conversational AI systems with domain-specific language capabilities.
- **Text Generation:** Create human-like text for applications like storytelling, dialogue systems, or creative writing.
- **Outlier Detection:** Use in detecting anomalies or unusual patterns in text-based datasets.
- **Legal Research Assistance:** Provide relevant case law, precedents, or statutory interpretations based on input queries.
- **Drug Interaction Analysis:** Generate predictions or explanations for potential drug interactions or side effects using pharmaceutical datasets.



CONCLUSION

- **nanoGPT Simplification:** Enables accessible GPT training by balancing simplicity and efficiency on limited hardware.
- **Efficient Design:** Leverages minimal transformer architecture and small datasets for fast prototyping.
- **Challenges Overcome:** Addressed overfitting with dropout, learning rate tuning, and simplified models.
- **Future Impact:** Powers diverse applications, including chatbots, text generation, and legal/medical analysis.

