BeviaLLM

P L A Y B O O K

Building a Language Model From Scratch

Python · NumPy · Transformers · Backpropagation

Version 1.0

corebaseit.com

Preface

Welcome to the BeviaLLM Playbook — a comprehensive guide to understanding and building a GPT-like language model from the ground up using nothing but Python and NumPy.

This book is designed for developers, students, and curious minds who want to truly understand what happens inside a language model. Instead of relying on high-level frameworks that hide the complexity behind convenient abstractions, we implement every component manually: from embeddings to attention mechanisms, from layer normalization to backpropagation.

What You'll Learn

How transformers process sequences of text · The mathematics behind attention mechanisms · Manual implementation of forward and backward passes · Building an optimizer from scratch · Training a character-level language model

Prerequisites

Comfortable with Python · Basic linear algebra (matrices, vectors) · Familiarity with calculus (derivatives, chain rule) · Curiosity and patience

PART I: FOUNDATIONS

Chapter 1: Introduction to BeviaLLM

1.1 What is BeviaLLM?

BeviaLLM is an educational implementation of a miniature GPT-like language model. Unlike production models with billions of parameters, BeviaLLM is intentionally small — designed to run on a laptop CPU in minutes rather than requiring GPU clusters for weeks.

The name combines "Bevia" (the creator's name) with "LLM" (Large Language Model), though in practice it's more of an "SLM" — a Small Language Model built for learning.

1.2 Why Build From Scratch?

Modern deep learning frameworks like PyTorch and TensorFlow provide automatic differentiation, GPU acceleration, and countless optimized layers. So why implement everything manually?

- Understanding over convenience — when you use torch.nn.Linear you get a working layer, but do you truly understand what happens during the backward pass?

- Demystifying the "magic" — transformers can feel like black boxes; terms like attention, residual connections, and layer normalization become meaningful when you implement them with explicit matrix operations.
- Appreciation for design decisions — why does GPT use pre-normalization? Why causal masking? Building from scratch reveals why certain choices matter.

1.3 Project Philosophy

BeviaLLM follows three core principles:

- Readability over speed — the code prioritizes clarity. Every matrix operation is explicit. Comments explain the "why," not just the "what."
- Minimal dependencies — only NumPy is required. No autograd, no CUDA, no abstractions you can't trace with a debugger.
- Hackability — want to try a different activation function? Change one line. Want multi-head attention? Extend the existing class.

1.4 What We're Building

By the end of this journey, you'll have a working language model that can learn patterns from text data, generate new text resembling the training data, and demonstrate the core mechanics of GPT-style architectures.

The model operates at the character level — predicting the next character given a sequence of previous characters. This is simpler than word-level tokenization but illustrates the same principles.

Chapter 2: The Transformer Architecture

2.1 A Brief History

In 2017, Vaswani et al. published "Attention Is All You Need," introducing the Transformer architecture. This paper replaced recurrent networks with attention mechanisms, enabling massive parallelization and setting the stage for models like GPT, BERT, and their successors.

BeviaLLM implements a decoder-only transformer, similar to GPT, processing sequences left-to-right and predicting each next token based on all previous tokens.

2.2 The Big Picture

At a high level, here's what happens when BeviaLLM processes text:

Stage	Component	Description
1	Tokenization	Convert characters to integer indices
2	Embedding	Map indices to dense vectors
3	Position Encoding	Add information about token positions

4	Transformer Blocks	Process through attention and MLPs
5	Output Layer	Project back to vocabulary size
6	Prediction	Sample the next character from probabilities

2.3 Key Components

A single transformer block contains four key components:

- Self-Attention — allows each position to "look at" other positions in the sequence, capturing dependencies between relevant words.
- Feed-Forward Network (MLP) — a two-layer network processing each position independently, adding computational capacity.
- Layer Normalization — stabilizes training by normalizing activations.
- Residual Connections — allow gradients to flow directly through the network, enabling deeper architectures.

2.4 Why "Causal"?

Our attention is "causal" because each position can only attend to itself and previous positions — never future ones. This is essential for generation: when predicting position 5, we can't peek at position 6.

We implement this with a mask that sets future positions to negative infinity before the softmax, effectively zeroing their attention weights.

Chapter 3: Mathematics Behind the Model

3.1 Notation

Symbol	Meaning
B	Batch size — number of sequences processed together
T	Sequence length — context window
C	Embedding dimension (also called dim)
V	Vocabulary size — number of unique tokens

3.2 Embeddings

An embedding table is a matrix of shape (V, C). To embed a token with index i, we look up the i-th row. The backward pass for embeddings is straightforward: we accumulate gradients at the indices that were used during the forward pass.

3.3 Attention Mechanics

Self-attention computes three projections from the input:

- Query (Q): What am I looking for?
- Key (K): What do I contain?
- Value (V): What should I return if matched?

Attention Formula

Attention(Q, K, V) = softmax(QK^T / √dk) · V Where dk is the dimension of the keys. The √dk scaling prevents dot products from becoming too large, which would push softmax into regions with tiny gradients.

3.4 Softmax

Softmax converts raw scores (logits) into probabilities. For numerical stability, we subtract the maximum value before exponentiating — this prevents overflow while producing identical results.

3.5 Cross-Entropy Loss

We use cross-entropy loss to measure how well our predicted probabilities match the true next character. The gradient of cross-entropy with respect to logits has a beautiful, elegant form: simply subtract 1 from the probability of the correct class.

3.6 Layer Normalization

Layer normalization normalizes across features (the embedding dimension) rather than across the batch. The learnable parameters γ and β are initialized to 1 and 0, meaning "start normalized, learn to denormalize if helpful."

PART II: IMPLEMENTATION DEEP DIVE

Chapter 4: Project Structure

4.1 File Organization

BeviaLLM is organized as a Python package with a clean, modular structure:

```
BeviaLLM/
├── main.py           # Training script
├── data.txt          # Training data
├── bevialm/
│   ├── __init__.py   # Public API exports
│   ├── data.py       # Data loading, encoding, sampling
│   ├── layers.py     # Neural network layers
│   ├── model.py      # The CharTransformerLM class
│   ├── optimizer.py  # AdamW implementation
│   └── utils.py      # Helper functions
```

4.2 Design Decisions

- Shared parameter dictionaries — instead of storing weights inside each layer object, shared params and grads dictionaries make it easy to iterate over all parameters for optimization.
- Cache-based backward pass — each layer stores its forward pass results in a cache attribute, which the backward pass retrieves. Explicit dependency injection rather than implicit graph building.
- Naming convention — parameters are named hierarchically, like block0.attn.q.W for the query weight matrix in the first block.

Chapter 5: Data Pipeline

5.1 Loading Text

The data pipeline starts with raw text. BeviaLLM expects a simple UTF-8 text file and builds a character-level vocabulary by finding all unique characters, creating stoi (string-to-int) and itos (int-to-string) mappings.

5.2 Creating Batches

Training requires random batches of sequences. For each sequence, x contains the input tokens (positions 0 to T−1) while y contains the target tokens (positions 1 to T). This offset trains the model to predict each next character.

5.3 Sampling from the Model

Generation works autoregressively: start with a prompt, feed the sequence to the model, get probabilities for the next character, sample from the distribution, append the character, and repeat.

Temperature

Temperature controls randomness during sampling: • Low (0.5): Conservative, more repetitive • Balanced (1.0): Natural distribution • High (1.5): Creative but chaotic

Chapter 6: Layers in Detail

6.1 The Embedding Layer

The embedding layer is deceptively simple. Forward pass: index into a weight matrix. Backward pass: accumulate gradients at those indices. The key insight is that np.add.at handles repeated indices correctly — if the same token appears multiple times, gradients accumulate.

6.2 The Linear Layer

A linear layer computes y = xW + b. The backward pass derives from the chain rule, computing gradients for W, x, and b separately. Inputs are reshaped to 2D for matrix operations, then reshaped back.

6.3 Layer Normalization

Layer normalization's backward pass involves gradients through the scale and shift parameters (γ, β), gradients through the normalization itself, and dependencies on both mean and variance — making it one of the more complex backward passes to implement correctly.

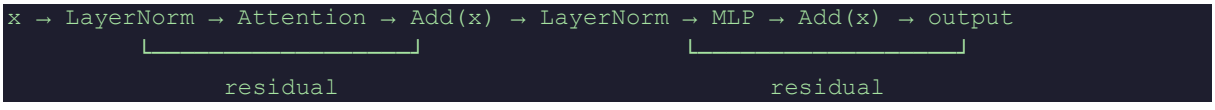
6.4 Causal Self-Attention

Attention is the heart of the transformer. The forward pass proceeds in six steps:

- Project input to Q, K, V via three separate linear layers
- Compute attention scores: $QK^T / \sqrt{d}$
- Apply causal mask: set future positions to $-\infty$
- Softmax: convert to probabilities
- Apply attention: weight values by attention probabilities
- Project output: final linear layer

6.5 The Transformer Block

A transformer block combines attention and MLP with residual connections. BeviaLLM uses "pre-norm" style, where normalization happens before each sublayer. This tends to train more stably than post-norm, especially for deeper networks.



6.6 Activation Functions

We use ReLU (Rectified Linear Unit) in the MLP: ReLU(x) = max(0, x). The backward pass is simple: gradient passes through where x > 0, and is zero elsewhere. The binary mask is saved during forward for use in backward.

Chapter 7: The Model Class

7.1 CharTransformerLM

The CharTransformerLM class ties everything together, managing token and position embeddings, a stack of transformer blocks, final layer normalization, and the output projection (language modeling head).

7.2 Forward Pass

```
def forward(self, idx):
    tok = self.tok_emb.forward(idx)        # Embed tokens
    pos = self.pos_emb.forward(positions)  # Embed positions
    x = tok + pos                          # Combine
    for blk in self.blocks:                # Through blocks
        x = blk.forward(x)
    x = self.ln_f.forward(x)               # Final norm
    logits = self.lm_head.forward(x)       # Project to vocab
    return logits
```

7.3 Backward Pass

The backward pass mirrors the forward pass in reverse, traversing through the language model head, final normalization, transformer blocks (in reverse order), and finally the embedding layers.

Chapter 8: The AdamW Optimizer

8.1 Why AdamW?

Adam (Adaptive Moment Estimation) combines momentum (exponential moving average of gradients) with RMSprop (exponential moving average of squared gradients). AdamW adds decoupled weight decay, which regularizes better than L2 regularization added to the loss.

8.2 The Algorithm

For each parameter at each step:

- Update biased first moment estimate: $m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$
- Update biased second moment estimate: $v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$
- Compute bias-corrected estimates: $\hat{m}_t = m_t/(1-\beta_1^t)$, $\hat{v}_t = v_t/(1-\beta_2^t)$
- Update parameters: $\theta_{t+1} = \theta_t - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$
- Apply weight decay: $\theta_{t+1} = \theta_{t+1} - \alpha \cdot \lambda \cdot \theta_t$

8.3 Hyperparameters

Parameter	Value	Description
β_1	0.9	First moment decay (momentum)
β_2	0.95	Second moment decay (borrowed from GPT-2)
ε	10^{-8}	Prevents division by zero
α	3×10^{-4}	Learning rate
λ	0.1	Weight decay coefficient

Chapter 9: Training Loop

9.1 The Core Loop

```
for step in range(num_steps):
    x, y = get_batch(data, batch_size, ctx)    # 1. Get batch
    model.zero_grads()                          # 2. Zero gradients
    logits = model.forward(x)                   # 3. Forward pass
    loss, dlogits = cross_entropy_loss(...)     # 4. Compute loss
    model.backward(dlogits)                     # 5. Backward pass
    optimizer.step(model.params, model.grads)   # 6. Update weights
```

9.2 Hyperparameter Guide

Parameter	Effect
ctx	Longer context = more memory, captures longer dependencies
dim	Larger dimension = more capacity, slower training

layers	More layers = deeper representation, harder to train
mlp_hidden	Wider MLP = more computation per position
batch	Larger batch = more stable gradients, more memory
lr	Higher = faster but unstable; lower = slower but stable

PART III: WORKING WITH BEVIALM

Chapter 10: Getting Started

10.1 Environment Setup

BeviaLLM requires only NumPy — no heavy ML frameworks needed:

```
python -m venv .venv
source .venv/bin/activate
pip install numpy
```

10.2 Preparing Training Data

Create or obtain a text file with at least a few thousand characters for interesting results. Good options include Shakespeare plays, song lyrics, code snippets, Wikipedia articles, or your own writing.

10.3 Running Training

Start with conservative settings and observe the loss decrease and samples improve over time:

```
python main.py \
  --data data.txt \
  --ctx 64 \
  --dim 64 \
  --layers 1 \
  --batch 8 \
  --steps 2000
```

Chapter 11: Experiments to Try

11.1 Varying Model Size

Config	Parameters	Speed	Quality
Tiny	dim=32, layers=1	Fast	Poor
Small	dim=64, layers=1	Medium	Okay
Medium	dim=128, layers=2	Slow	Better

11.2 Different Datasets

- Code — the model learns syntax and structure
- Poetry — watch it learn rhyme and meter patterns
- Technical docs — observe how it captures domain terminology

11.3 Context Window

Longer context windows let the model consider more history but increase memory quadratically due to attention. Try ctx=128 or ctx=256 with smaller batches to observe the trade-offs.

Chapter 12: Understanding the Code

12.1 Debugging Tips

- Check shapes — most bugs manifest as shape mismatches; print shapes liberally
- Verify gradients — compare manual gradients to numerical approximations: $(f(x+\varepsilon) - f(x-\varepsilon)) / (2\varepsilon)$
- Monitor activations — very large or NaN values indicate numerical problems

12.2 Common Issues

Problem	Likely Cause
Loss not decreasing	Learning rate too high/low, data loading bug, gradient not flowing
NaN loss	Numerical overflow in softmax, learning rate too high, weight init issues
Memory errors	Reduce batch size or context length, check for caching memory leaks

PART IV: CONCEPTUAL DEEP DIVES

Chapter 14: Why Attention Works

14.1 The Lookup Table Analogy

Think of attention as a soft lookup table. Given a query, we find which keys are most relevant and return a weighted combination of their values.

Concept	Dictionary	Attention
Key matching	Exact key match	Soft weighted match
Return value	Single value	Weighted combination of values
Flexibility	Rigid	Dynamic and learnable

14.2 Information Routing

Attention lets the model route information dynamically. For "The cat sat on the mat," when processing "sat," the model can attend heavily to "cat" (the subject) and less to other words. This dynamic routing is what makes transformers so powerful.

14.3 Position Matters

Without position embeddings, attention is permutation-invariant — it can't distinguish "dog bites man" from "man bites dog." Position embeddings break this symmetry, giving the model a sense of sequence order.

Chapter 15: The Role of Layer Normalization

15.1 Training Stability

Deep networks suffer from internal covariate shift — each layer's input distribution changes during training as parameters update. Layer normalization stabilizes this by normalizing each sample independently, making training much more predictable.

15.2 Pre-Norm vs Post-Norm

Style	Architecture	Stability
Post-norm (original)	Attention → Add → LayerNorm → MLP → Add → LayerNorm	Less stable, especially deep

Pre-norm (GPT-2+)	LayerNorm → Attention → Add → LayerNorm → MLP → Add	More stable, preferred today

Chapter 16: Understanding Backpropagation

16.1 The Chain Rule

All of backpropagation follows from one calculus rule: the gradient of the loss with respect to an input equals the upstream gradient multiplied by the local gradient. BeviaLLM makes this explicit through cache attributes — each layer saves what it needs for the backward pass.

16.2 Accumulating Gradients

When a value is used multiple times (like a shared weight), gradients accumulate. This is why we zero gradients before each backward pass — without zeroing, gradients from previous steps would incorrectly persist.

APPENDICES

Appendix A: Glossary

Term	Definition
Attention	Mechanism for weighting different parts of the input based on relevance
Backpropagation	Algorithm for computing gradients by traversing the computational graph backwards
Batch	Collection of samples processed together for efficiency and gradient stability
Causal	Constraint that a position can only attend to previous positions
Context Window	The maximum sequence length the model can process
Cross-Entropy	Loss function measuring difference between predicted and true distributions
Embedding	Dense vector representation of discrete tokens
Gradient	Direction of steepest increase; we move opposite to decrease loss
Layer Normalization	Stabilizes training by normalizing layer activations

Learning Rate	How much to update parameters each step
Logits	Raw model outputs before applying softmax
MLP	Multi-Layer Perceptron; a feed-forward neural network
Residual Connection	Adding a layer's input to its output, enabling gradient flow
Softmax	Function converting logits to probabilities summing to 1
Token	The smallest unit the model processes (characters in our case)
Transformer	Architecture based on self-attention, introduced in 2017
Weight Decay	Regularization technique that shrinks weights toward zero

Appendix C: Configuration Reference

Argument	Type	Default	Description
--data	str	(required)	Path to training text file
--seed	int	42	Random seed for reproducibility
--ctx	int	128	Context window length
--dim	int	128	Embedding dimension
--layers	int	2	Number of transformer blocks
--mlp_hidden	int	512	MLP hidden layer size
--batch	int	16	Batch size
--steps	int	5000	Training steps
--lr	float	3e-4	Learning rate
--wd	float	0.1	Weight decay
--log_every	int	200	Log loss every N steps
--sample_every	int	1000	Sample text every N steps
--sample_len	int	400	Length of sampled text

Appendix D: Further Reading

Foundational Papers

- "Attention Is All You Need" (Vaswani et al., 2017) — The original transformer paper

- "Language Models are Unsupervised Multitask Learners" (GPT-2 paper, 2019)
- "Language Models are Few-Shot Learners" (GPT-3 paper, 2020)

Books

- "Deep Learning" by Goodfellow, Bengio, and Courville
- "Neural Networks and Deep Learning" by Michael Nielsen (free online)

Online Resources

- The Illustrated Transformer — Jay Alammar's blog
- Andrej Karpathy's "Let's build GPT" video series
- The Annotated Transformer — Harvard NLP

Appendix E: Source Code Repository

GitHub Repository

`https://github.com/Bevia/BeviaLLM`

corebaseit.com

The complete, working implementation of everything described in this playbook. Clone it, run it, break it — then rebuild it.

What's in the Repository

The repository contains the full BeviaLLM Python package, training data, and IDE configuration for a smooth out-of-the-box experience:

File / Folder	Contents
main.py	CLI entry point and training loop
data.txt	Sample training corpus (character-level text)
bevialm/__init__.py	Public package API exports
bevialm/layers.py	Embedding, Linear, LayerNorm, CausalSelfAttention, TransformerBlock
bevialm/model.py	CharTransformerLM — the top-level model class
bevialm/optimizer.py	AdamW optimizer implemented from scratch
bevialm/data.py	load_text, build_vocab, encode, get_batch, sample
bevialm/utils.py	set_seed, softmax, cross_entropy_loss, init_weight

.vscode/	VS Code launch.json, settings.json, tasks.json for debug setup

Getting the Code

```
# Clone the repository
git clone https://github.com/Bevia/BeviaLLM.git
cd BeviaLLM

# Set up environment
python -m venv .venv
source .venv/bin/activate   # macOS/Linux
pip install numpy

# Run training
python main.py --data data.txt --ctx 64 --dim 64 --layers 1 --batch 8 --steps 2000
```

What to Study in the Code

File	What to Study
bevialm/layers.py	How attention really works (Q, K, V, masking), LayerNorm stability, residual gradient flow
bevialm/utils.py	How softmax gradients behave and cross-entropy derivation
bevialm/optimizer.py	Why AdamW outperforms vanilla SGD
bevialm/model.py	Embeddings and the full forward/backward pass end-to-end
bevialm/data.py	Text sampling with temperature and batch construction

Suggested Next Steps

Once comfortable with the base implementation, these exercises deepen understanding further:

- Implement multi-head attention (currently single-head in bevialm/layers.py)
- Replace ReLU activation with GELU
- Add dropout regularization
- Implement learning rate warmup and cosine decay scheduling
- Replace char-level tokenization with a BPE tokenizer
- Add checkpoint saving and loading
- Visualize attention maps during generation
- Write unit tests for each module, verifying gradients numerically

Afterword

Building a language model from scratch is a journey of discovery. Each component — from the humble embedding lookup to the elegant attention mechanism — contributes to the emergent behavior we call "language understanding."

BeviaLLM is simply a friendly way to peek behind the curtain and understand how large language models like ChatGPT actually work. When you trace through the matrix multiplications, debug a gradient calculation, and watch the loss decrease — you build intuition that reading documentation alone can't provide.

The Journey Ahead

Take what you've learned here and apply it. Modify the code. Break it, fix it, extend it. The best way to understand deep learning is to get your hands dirty with the math. Happy hacking!

BeviaLLM Playbook — Version 1.0 · corebaseit.com