

The Mathematics of microGPT

A Mathematical Companion to Andrej Karpathy's
200-Line GPT Implementation

Prepared for FINM 33200: Generative and Agentic AI for Finance
The University of Chicago

Jeremy Bejarano

March 2026

Based on `microgpt.py` by Andrej Karpathy
<https://gist.github.com/karpathy/8627fe009c40f57531cb18360106ce95>

Contents

1	Introduction and Overview	2
1.1	Why This Document Exists	2
1.2	Model Specifications	2
2	Tokenization	2
3	Automatic Differentiation	3
3.1	The Computation Graph	3
3.2	Elementary Operations and Their Local Gradients	3
3.3	Backpropagation via the Chain Rule	4
4	Model Parameters	5
5	Building Blocks	6
5.1	Linear Transformation	6
5.2	Softmax	6
5.3	RMS Normalization	7
6	Model Architecture	7
6.1	Architecture Diagram	7
6.2	Embedding Layer	7
6.3	Multi-Head Causal Self-Attention	9
6.4	The KV Cache	10
6.5	Feed-Forward Network (MLP)	11
6.6	Output Logits	11
7	Training	11
7.1	Cross-Entropy Loss	11
7.2	The Adam Optimizer	12
7.3	The Complete Training Loop	13
8	Inference	14
8.1	Temperature Sampling	14
9	Connecting to Production LLMs	14
10	Summary of All Equations	15

1 Introduction and Overview

In February 2026, Andrej Karpathy released `microGPT`—a single Python file of approximately 200 lines, with no external dependencies, that implements the complete training and inference loop for a GPT-class language model. The code trains a character-level transformer on a dataset of 32,000 human names and learns to generate new, plausible-sounding names by modeling the statistical patterns in the data.

`microGPT` is pedagogically extraordinary because it contains the *entire* algorithmic content of modern large language models: a tokenizer, a scalar-valued automatic differentiation engine, a GPT-2-style transformer architecture with multi-head causal attention, the Adam optimizer, a training loop, and an autoregressive inference loop. Everything beyond these 200 lines—GPU parallelism, tensor operations, BPE tokenization, distributed training—is, as Karpathy puts it, “just efficiency.”

1.1 Why This Document Exists

This companion document serves three purposes. First, it provides a rigorous mathematical treatment of every component of `microGPT`, writing out the definitions and equations that the code implements. Second, it maps each mathematical expression directly to the corresponding Python code, so the reader can verify that the math and the implementation are in exact correspondence. Third, it provides architectural diagrams and pedagogical discussion to build intuition for how and why these components work together.

1.2 Model Specifications

The default hyperparameters in `microGPT` define a small but complete transformer:

Hyperparameter	Value	Variable
Vocabulary size	27	<code>vocab_size</code>
Embedding dimension	16	<code>n_embd</code>
Number of layers	1	<code>n_layer</code>
Number of attention heads	4	<code>n_head</code>
Head dimension	4	<code>head_dim</code>
Max sequence length	16	<code>block_size</code>
Total parameters	4,192	<code>len(params)</code>

Table 1: Default hyperparameters of `microGPT`.

The vocabulary consists of the 26 lowercase English letters plus a special Beginning-of-Sequence (BOS) token, for a total of $V = 27$ tokens.

2 Tokenization

Definition 2.1 (Character-Level Tokenizer). Let $\mathcal{A} = \{a, b, c, \dots, z\}$ be the set of unique characters in the dataset, sorted lexicographically. Define the encoding function $\text{enc} : \mathcal{A} \cup \{\text{BOS}\} \rightarrow \{0, 1, \dots, V - 1\}$ by

$$\text{enc}(c) = \begin{cases} \text{index of } c \text{ in sorted } \mathcal{A}, & c \in \mathcal{A} \\ |\mathcal{A}| = 26, & c = \text{BOS} \end{cases} \quad (1)$$

where $V = |\mathcal{A}| + 1 = 27$ is the vocabulary size.

Each document (name) is tokenized by wrapping it with BOS delimiters. For example, the name “emma” becomes the token sequence $[\text{BOS}, e, m, m, a, \text{BOS}] = [26, 4, 12, 12, 0, 26]$.

Listing 1: Tokenizer implementation in `microGPT`.

```

1  uchars = sorted(set(''.join(docs))) # ['a', 'b', ..., 'z']
2  BOS = len(uchars)                  # 26
3  vocab_size = len(uchars) + 1       # 27
4
5  # Tokenizing a document:
6  tokens = [BOS] + [uchars.index(ch) for ch in doc] + [BOS]

```

Connection to production systems

Production LLMs use subword tokenizers such as Byte Pair Encoding (BPE), which merge frequently co-occurring character sequences into single tokens, yielding vocabularies of $\sim 100\text{K}$ tokens. The character-level tokenizer in `microGPT` is the simplest possible instance of this idea: each “subword” is a single character.

3 Automatic Differentiation

The `Value` class implements reverse-mode automatic differentiation (backpropagation) on scalar values. Each `Value` node stores a scalar, tracks how it was computed, and can propagate gradients backward through the computation graph.

3.1 The Computation Graph

Definition 3.1 (Computation Graph). A computation graph is a directed acyclic graph (DAG) $G = (\mathcal{V}, \mathcal{E})$ where each node $v \in \mathcal{V}$ holds a scalar value $v.\text{data} \in \mathbb{R}$ and each edge represents a dependency: if v was computed from children $c_1, \dots, c_k$ via some operation f , then $(c_i, v) \in \mathcal{E}$ for all i .

Each elementary operation in `microGPT` is a “lego block” that knows two things: (1) how to compute its output from its inputs (the forward pass), and (2) the local partial derivatives of its output with respect to each input.

3.2 Elementary Operations and Their Local Gradients

The following table lists every operation implemented in the `Value` class, along with the corresponding forward computation and local gradients. These correspond exactly to the operator overloads in the code.

Operation	Forward: $v = f(a, b)$	$\frac{\partial v}{\partial a}$	$\frac{\partial v}{\partial b}$
<code>a + b</code>	$a + b$	1	1
<code>a * b</code>	$a \cdot b$	b	a
<code>a ** n</code>	a^n	$n \cdot a^{n-1}$	—
<code>a.log()</code>	$\ln(a)$	$1/a$	—
<code>a.exp()</code>	e^a	e^a	—
<code>a.relu()</code>	$\max(0, a)$	$\mathbf{1}_{a>0}$	—

Table 2: Elementary operations and their local gradients in the `Value` class.

Listing 2: Core Value operations (excerpt).

```

1 def __add__(self, other):
2     other = other if isinstance(other, Value) else Value(other)
3     return Value(self.data + other.data, (self, other), (1, 1))
4     #           forward value           children           local grads
5
6 def __mul__(self, other):
7     other = other if isinstance(other, Value) else Value(other)
8     return Value(self.data * other.data, (self, other),
9             (other.data, self.data))
10    #    local grads: dv/da = b,    dv/db = a
11
12 def __pow__(self, other):
13     return Value(self.data**other, (self,),
14             (other * self.data**(other-1),))
15
16 def log(self):
17     return Value(math.log(self.data), (self,), (1/self.data,))
18
19 def exp(self):
20     return Value(math.exp(self.data), (self,),
21             (math.exp(self.data),))
22
23 def relu(self):
24     return Value(max(0, self.data), (self,),
25             (float(self.data > 0),))

```

3.3 Backpropagation via the Chain Rule

Given a loss scalar L at the root of the computation graph, backpropagation computes $\frac{\partial L}{\partial v}$ for every node v in the graph.

Proposition 3.1 (Chain Rule in the Computation Graph). *Let v be a node with children $c_1, \dots, c_k$ and local gradients $\frac{\partial v}{\partial c_i}$. If v contributes to L (possibly through multiple paths), then the gradient of L with respect to child c_i accumulates contributions from all parent nodes of c_i :*

$$\boxed{\frac{\partial L}{\partial c_i} += \frac{\partial v}{\partial c_i} \cdot \frac{\partial L}{\partial v}} \quad (2)$$

The $+=$ (accumulation, not assignment) is essential: when a node c_i is used in multiple places in the graph, the gradients from each path must be summed. This follows from the multivariable chain rule.

The `backward()` method implements this by:

1. Building a topological ordering of the graph via depth-first search.
2. Setting $\frac{\partial L}{\partial L} = 1$ (the base case).
3. Traversing nodes in reverse topological order, applying eq. (2) at each step.

Listing 3: Backpropagation in the Value class.

```

1 def backward(self):
2     topo = []
3     visited = set()
4     def build_topo(v):
5         if v not in visited:
6             visited.add(v)

```

```

7         for child in v._children:
8             build_topo(child)
9         topo.append(v)
10    build_topo(self)
11    self.grad = 1 # dL/dL = 1
12    for v in reversed(topo):
13        for child, local_grad in zip(v._children, v._local_grads):
14            child.grad += local_grad * v.grad # Eq. (2)

```

Concrete example

Consider $a = 2$, $b = 3$, $c = a \cdot b = 6$, $L = c + a = 8$. Then:

$$\begin{aligned}
 \frac{\partial L}{\partial L} &= 1 \\
 \frac{\partial L}{\partial c} &= 1, \quad \frac{\partial L}{\partial a} \Big|_{\text{via } c+a} = 1 \\
 \frac{\partial L}{\partial a} \Big|_{\text{via } a \cdot b} &= b = 3, \quad \frac{\partial L}{\partial b} = a = 2
 \end{aligned}$$

Since a appears in two paths, we accumulate: $\frac{\partial L}{\partial a} = 1 + 3 = 4$. This matches `a.grad = 4.0` in the code.

4 Model Parameters

All learnable parameters are organized in a dictionary called `state_dict`, borrowing PyTorch’s terminology. Each parameter is a `Value` initialized from $\mathcal{N}(0, \sigma^2)$ with $\sigma = 0.08$.

Definition 4.1 (Parameter Matrices). The model parameters consist of the following matrices:

$$W^{\text{te}} \in \mathbb{R}^{V \times d} \quad (\text{token embeddings}) \quad (3)$$

$$W^{\text{pe}} \in \mathbb{R}^{T \times d} \quad (\text{position embeddings}) \quad (4)$$

$$W^{\text{lm}} \in \mathbb{R}^{V \times d} \quad (\text{language model head}) \quad (5)$$

For each layer $\ell = 0, \dots, L - 1$:

$$W_\ell^Q, W_\ell^K, W_\ell^V, W_\ell^O \in \mathbb{R}^{d \times d} \quad (\text{attention projections}) \quad (6)$$

$$W_\ell^{\text{fc1}} \in \mathbb{R}^{4d \times d} \quad (\text{MLP up-projection}) \quad (7)$$

$$W_\ell^{\text{fc2}} \in \mathbb{R}^{d \times 4d} \quad (\text{MLP down-projection}) \quad (8)$$

where $d = 16$ (`n_embd`), $V = 27$ (`vocab_size`), $T = 16$ (`block_size`), and $L = 1$ (`n_layer`).

Listing 4: Parameter initialization.

```

1 n_embd, n_head, n_layer, block_size = 16, 4, 1, 16
2 head_dim = n_embd // n_head # 4
3 matrix = lambda nout, nin, std=0.08: \
4     [[Value(random.gauss(0, std)) for _ in range(nin)]
5      for _ in range(nout)]
6
7 state_dict = {
8     'wte': matrix(vocab_size, n_embd), # V x d
9     'wpe': matrix(block_size, n_embd), # T x d

```

```

10     'lm_head': matrix(vocab_size, n_embd),      # V x d
11 }
12 for i in range(n_layer):
13     state_dict[f'layer{i}.attn_wq'] = matrix(n_embd, n_embd) # d x
14     state_dict[f'layer{i}.attn_wk'] = matrix(n_embd, n_embd)
15     state_dict[f'layer{i}.attn_wv'] = matrix(n_embd, n_embd)
16     state_dict[f'layer{i}.attn_wo'] = matrix(n_embd, n_embd)
17     state_dict[f'layer{i}.mlp_fc1'] = matrix(4*n_embd, n_embd) # 4d
18     state_dict[f'layer{i}.mlp_fc2'] = matrix(n_embd, 4*n_embd) # d

```

The total parameter count is:

$$|\theta| = \underbrace{Vd}_{\text{wte}} + \underbrace{Td}_{\text{wpe}} + \underbrace{Vd}_{\text{lm_head}} + L \cdot \left(\underbrace{4d^2}_{\text{attn}} + \underbrace{4d^2 + 4d^2}_{\text{MLP}} \right) = 27 \cdot 16 + 16 \cdot 16 + 27 \cdot 16 + 1 \cdot 12 \cdot 16^2 = 4,192 \quad (9)$$

5 Building Blocks

Before defining the full model, we formalize the three helper functions used throughout.

5.1 Linear Transformation

Definition 5.1 (Linear Layer). Given an input vector $\mathbf{x} \in \mathbb{R}^{n_{\text{in}}}$ and a weight matrix $W \in \mathbb{R}^{n_{\text{out}} \times n_{\text{in}}}$, the linear transformation (with no bias) is:

$$\text{linear}(\mathbf{x}, W) = W\mathbf{x} \in \mathbb{R}^{n_{\text{out}}} \quad (10)$$

That is, the i -th output component is $[\text{linear}(\mathbf{x}, W)]_i = \sum_{j=1}^{n_{\text{in}}} W_{ij} x_j$.

Listing 5: Linear transformation.

```

1 def linear(x, w):
2     return [sum(wi * xi for wi, xi in zip(wo, x)) for wo in w]
3     # wo is row i of W; dot product wo . x gives output i

```

5.2 Softmax

Definition 5.2 (Softmax). Given a vector of logits $\mathbf{z} \in \mathbb{R}^K$, the softmax function produces a probability distribution:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i - z_{\max}}}{\sum_{j=1}^K e^{z_j - z_{\max}}}, \quad i = 1, \dots, K \quad (11)$$

where $z_{\max} = \max_j z_j$ is subtracted for numerical stability (it does not change the output mathematically, since $e^{z_i - c} / \sum_j e^{z_j - c} = e^{z_i} / \sum_j e^{z_j}$ for any constant c).

Listing 6: Softmax with numerical stability.

```

1 def softmax(logits):
2     max_val = max(val.data for val in logits) # z_max

```

```

3     exps = [(val - max_val).exp() for val in logits] # e^{z_i -
              z_max}
4     total = sum(exps)                                # denominator
5     return [e / total for e in exps]                 # probabilities

```

5.3 RMS Normalization

Definition 5.3 (RMSNorm). Given a vector $\mathbf{x} \in \mathbb{R}^d$, Root Mean Square Normalization rescales it to have unit root-mean-square:

$$\text{RMSNorm}(\mathbf{x})_i = \frac{x_i}{\sqrt{\frac{1}{d} \sum_{j=1}^d x_j^2 + \epsilon}} \quad (12)$$

where $\epsilon = 10^{-5}$ is a small constant for numerical stability.

Remark 5.1. microGPT uses RMSNorm rather than the LayerNorm used in the original GPT-2. RMSNorm omits the mean-centering step and has no learnable scale/shift parameters, making it simpler. Note also that the code implements no learnable γ (gain) parameter, which is a further simplification.

Listing 7: RMSNorm implementation.

```

1 def rmsnorm(x):
2     ms = sum(xi * xi for xi in x) / len(x) # mean square
3     scale = (ms + 1e-5) ** -0.5           # 1 / sqrt(ms + eps)
4     return [xi * scale for xi in x]

```

6 Model Architecture

The `gpt()` function implements a single forward pass for one token at one position. It is a causal (autoregressive) transformer decoder, following the GPT-2 architecture with the simplifications noted in section 5.3.

6.1 Architecture Diagram

6.2 Embedding Layer

Given a token id $t \in \{0, \dots, V-1\}$ at position $p \in \{0, \dots, T-1\}$, the model first looks up learned embedding vectors:

$$\mathbf{e}_{\text{tok}} = W^{\text{te}}[t, :] \in \mathbb{R}^d \quad (\text{row } t \text{ of the token embedding table}) \quad (13)$$

$$\mathbf{e}_{\text{pos}} = W^{\text{pe}}[p, :] \in \mathbb{R}^d \quad (\text{row } p \text{ of the position embedding table}) \quad (14)$$

These are summed and normalized:

$$\mathbf{x}^{(0)} = \text{RMSNorm}(\mathbf{e}_{\text{tok}} + \mathbf{e}_{\text{pos}}) \in \mathbb{R}^d \quad (15)$$

Listing 8: Embedding lookup and initial normalization.

```

1 def gpt(token_id, pos_id, keys, values):
2     tok_emb = state_dict['wte'][token_id] # Eq. (10)

```

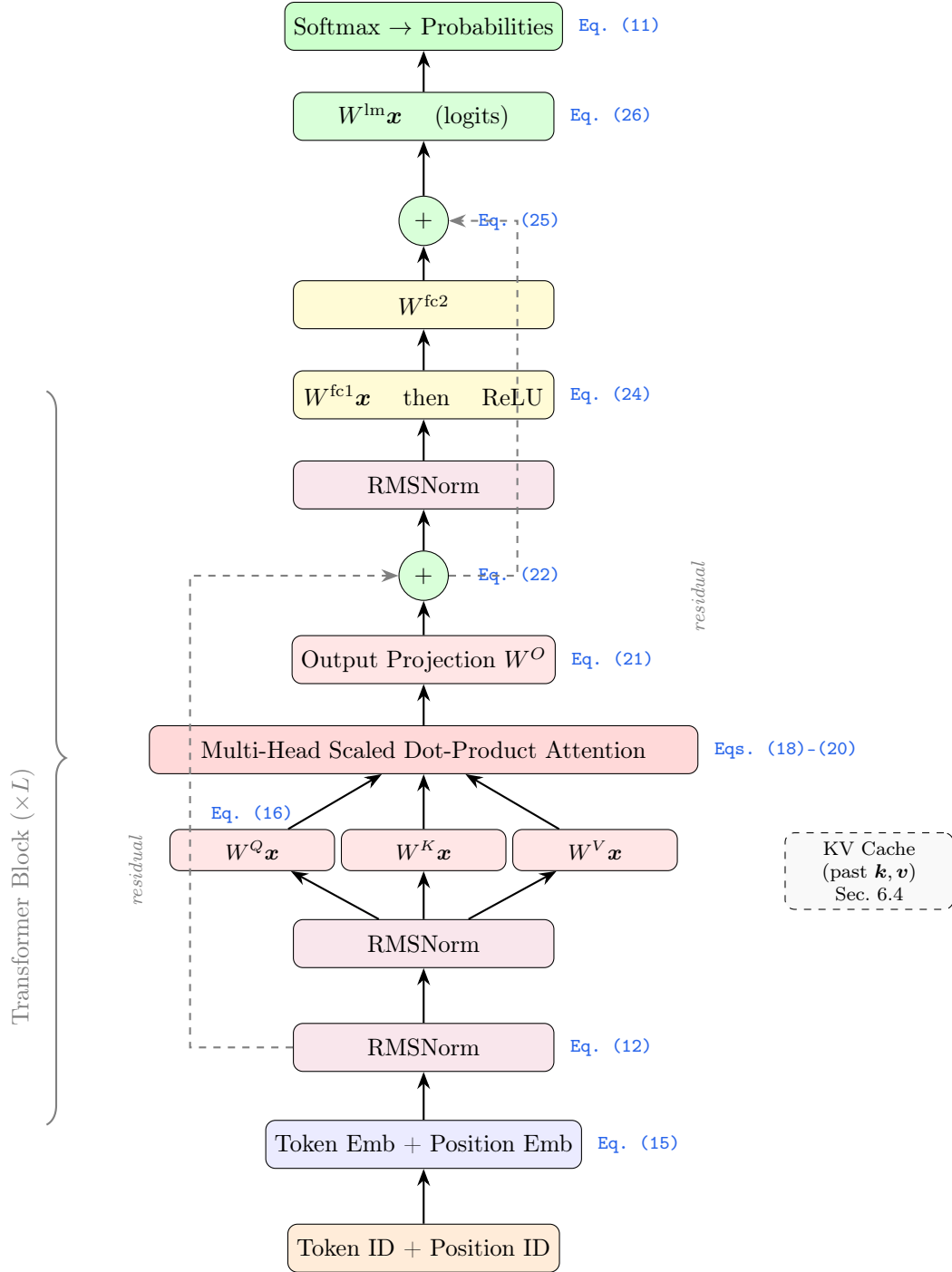


Figure 1: Architecture of microGPT, with equation references to each component. Dashed lines indicate residual connections. The transformer block (between the braces) is repeated L times ($L = 1$ by default).

```

3     pos_emb = state_dict['wpe'][pos_id]      # Eq. (11)
4     x = [t + p for t, p in zip(tok_emb, pos_emb)] # element-wise
        sum
5     x = rmsnorm(x)                            # Eq. (12)

```

Remark 6.1. Karpathy notes in the code that the initial RMSNorm is “not redundant due to the backward pass via the residual connection.” Because the first residual connection adds back the pre-normalized embedding, the gradient flows through both the normalized and unnormalized paths, so the normalization does affect learning even if it precedes the first layer.

6.3 Multi-Head Causal Self-Attention

The attention mechanism is where tokens communicate information across positions. At each layer ℓ , the current hidden state $\mathbf{x}$ is first normalized, then projected into queries, keys, and values:

$$\mathbf{q} = W_\ell^Q \cdot \text{RMSNorm}(\mathbf{x}), \quad \mathbf{k} = W_\ell^K \cdot \text{RMSNorm}(\mathbf{x}), \quad \mathbf{v} = W_\ell^V \cdot \text{RMSNorm}(\mathbf{x}) \quad (16)$$

where $\mathbf{q}, \mathbf{k}, \mathbf{v} \in \mathbb{R}^d$.

Listing 9: Query, Key, Value projections.

```

1 x_residual = x
2 x = rmsnorm(x)
3 q = linear(x, state_dict[f'layer{li}.attn_wq']) # W^Q x
4 k = linear(x, state_dict[f'layer{li}.attn_wk']) # W^K x
5 v = linear(x, state_dict[f'layer{li}.attn_wv']) # W^V x
6 keys[li].append(k)
7 values[li].append(v)

```

The key and value vectors are appended to the KV cache (see section 6.4). The query, key, and value vectors are then split into $H = 4$ heads, each of dimension $d_h = d/H = 4$.

Definition 6.1 (Head Slicing). For attention head $h \in \{0, 1, \dots, H-1\}$, define the index range $[h \cdot d_h, (h+1) \cdot d_h)$ and extract:

$$\mathbf{q}^{(h)} = \mathbf{q}[h \cdot d_h : (h+1) \cdot d_h] \in \mathbb{R}^{d_h} \quad (17)$$

and similarly for $\mathbf{k}_s^{(h)}$ and $\mathbf{v}_s^{(h)}$ at each cached position s .

For each head, we compute scaled dot-product attention over all cached positions $s = 0, 1, \dots, t$ (where t is the current position):

$$\alpha_s^{(h)} = \frac{\mathbf{q}^{(h)} \cdot \mathbf{k}_s^{(h)}}{\sqrt{d_h}} = \frac{1}{\sqrt{d_h}} \sum_{j=1}^{d_h} q_j^{(h)} \cdot k_{s,j}^{(h)} \quad (18)$$

$$\mathbf{w}^{(h)} = \text{softmax}([\alpha_0^{(h)}, \alpha_1^{(h)}, \dots, \alpha_t^{(h)}]) \in \mathbb{R}^{t+1} \quad (19)$$

$$\mathbf{h}^{(h)} = \sum_{s=0}^t w_s^{(h)} \mathbf{v}_s^{(h)} \in \mathbb{R}^{d_h} \quad (20)$$

Listing 10: Multi-head attention computation.

```

1 x_attn = []
2 for h in range(n_head):
3     hs = h * head_dim
4     q_h = q[hs:hs+head_dim] # Eq. (14)
5     k_h = [ki[hs:hs+head_dim] for ki in keys[li]]
6     v_h = [vi[hs:hs+head_dim] for vi in values[li]]
7     attn_logits = [ # Eq. (15)
8         sum(q_h[j] * k_h[t][j] for j in range(head_dim))
9         / head_dim**0.5
10        for t in range(len(k_h))
11    ]
12    attn_weights = softmax(attn_logits) # Eq. (16)
13    head_out = [ # Eq. (17)
14        sum(attn_weights[t] * v_h[t][j]
15            for t in range(len(v_h)))
16        for j in range(head_dim)
17    ]
18    x_attn.extend(head_out)

```

The outputs of all heads are concatenated and projected through an output matrix:

$$\mathbf{x}_{\text{attn}} = W_{\ell}^O \cdot [\mathbf{h}^{(0)} \parallel \mathbf{h}^{(1)} \parallel \dots \parallel \mathbf{h}^{(H-1)}] \in \mathbb{R}^d \quad (21)$$

Finally, the attention output is added back to the input via a residual connection:

$$\mathbf{x} \leftarrow \mathbf{x}_{\text{attn}} + \mathbf{x}_{\text{residual}} \quad (22)$$

Listing 11: Output projection and residual connection.

```

1 x = linear(x_attn, state_dict[f'layer{li}.attn_wo']) # Eq. (18)
2 x = [a + b for a, b in zip(x, x_residual)] # Eq. (19)

```

Intuition for Q, K, V

The query–key–value mechanism can be understood through an information retrieval analogy. Each token broadcasts a *key* (“here is what I contain”) and a *value* (“here is what I offer if selected”). The current token’s *query* (“here is what I’m looking for”) is compared against all keys via dot products, producing attention weights that determine how much each past token’s value contributes to the current representation. The scaling by $1/\sqrt{d_h}$ prevents the dot products from growing large in magnitude, which would cause the softmax to saturate into a near-one-hot distribution and impede gradient flow.

6.4 The KV Cache

At each position, the computed key and value vectors are appended to per-layer lists:

$$\text{keys}[\ell].\text{append}(\mathbf{k}), \quad \text{values}[\ell].\text{append}(\mathbf{v}) \quad (23)$$

This means that when processing position t , the attention mechanism has access to keys and values from all positions $0, 1, \dots, t$. This is what makes the model *causal*: each position can only attend to the current and past positions, never the future. In production systems, the KV cache is typically used only during inference (to avoid recomputing past positions). In `microGPT`, because tokens are processed one at a time even during training, the KV cache is used in both phases—and importantly, the cached values are live `Value` nodes, so gradients flow through them during backpropagation.

6.5 Feed-Forward Network (MLP)

After attention, a two-layer MLP processes each position independently:

$$\mathbf{x}_{\text{MLP}} = W_{\ell}^{\text{fc2}} \cdot \text{ReLU}(W_{\ell}^{\text{fc1}} \cdot \text{RMSNorm}(\mathbf{x})) \quad (24)$$

where $\text{ReLU}(z) = \max(0, z)$ is applied element-wise. The up-projection $W_{\ell}^{\text{fc1}} \in \mathbb{R}^{4d \times d}$ expands the representation to $4d = 64$ dimensions, and the down-projection $W_{\ell}^{\text{fc2}} \in \mathbb{R}^{d \times 4d}$ compresses back to $d = 16$. This is followed by another residual connection:

$$\mathbf{x} \leftarrow \mathbf{x}_{\text{MLP}} + \mathbf{x}_{\text{residual}} \quad (25)$$

Listing 12: MLP block with residual connection.

```

1 x_residual = x
2 x = rmsnorm(x)
3 x = linear(x, state_dict[f'layer{li}.mlp_fc1']) # up-project: d ->
    4d
4 x = [xi.relu() for xi in x] # ReLU activation
5 x = linear(x, state_dict[f'layer{li}.mlp_fc2']) # down-project: 4d
    -> d
6 x = [a + b for a, b in zip(x, x_residual)] # Eq. (22)
```

Remark 6.2. The original GPT-2 uses GeLU rather than ReLU. `microGPT` uses ReLU because it is simpler to implement in the scalar autograd engine (just $\max(0, \mathbf{x})$ with a gradient of $\mathbf{1}_{x>0}$). GeLU would require implementing the Gaussian CDF or its approximation, adding complexity without fundamentally changing the architecture.

6.6 Output Logits

After L transformer blocks, the final hidden state is projected to vocabulary size:

$$\mathbf{z} = W^{\text{lm}} \mathbf{x} \in \mathbb{R}^V \quad (26)$$

Each component z_i is a “logit” representing the model’s unnormalized score for token i being the next token. Higher logit $\Rightarrow$ higher predicted probability. These logits are then passed through softmax to obtain a probability distribution (eq. (11)).

Listing 13: Output projection to logits.

```

1 logits = linear(x, state_dict['lm_head']) # Eq. (23): d -> V
2 return logits
```

7 Training

7.1 Cross-Entropy Loss

At each position t in a document with token sequence $[s_0, s_1, \dots, s_n]$, the model takes s_t as input and predicts a probability distribution $\mathbf{p}^{(t)} = \text{softmax}(\mathbf{z}^{(t)})$ over the next token. The loss at position t is the negative log-probability of the true next token s_{t+1} :

$$\mathcal{L}_t = -\log p_{s_{t+1}}^{(t)} \quad (27)$$

This is the **cross-entropy loss** between the model's predicted distribution and the one-hot target distribution. Minimizing $\mathcal{L}_t$ is equivalent to maximizing the likelihood of the observed next token under the model's distribution.

The total loss for a document of length n is the average:

$$\mathcal{L} = \frac{1}{n} \sum_{t=0}^{n-1} \mathcal{L}_t = -\frac{1}{n} \sum_{t=0}^{n-1} \log p_{s_{t+1}}^{(t)} \quad (28)$$

Listing 14: Forward pass and loss computation.

```

1 keys, values = [[] for _ in range(n_layer)], \
2               [[] for _ in range(n_layer)]
3 losses = []
4 for pos_id in range(n):
5     token_id, target_id = tokens[pos_id], tokens[pos_id + 1]
6     logits = gpt(token_id, pos_id, keys, values)
7     probs = softmax(logits)
8     loss_t = -probs[target_id].log() # Eq. (24): -log p(target)
9     losses.append(loss_t)
10 loss = (1 / n) * sum(losses) # Eq. (25): average

```

Initial loss as a sanity check

At initialization, all parameters are random, so the model should assign roughly equal probability $\approx 1/V = 1/27$ to each token. The expected initial loss is therefore $-\log(1/27) = \log(27) \approx 3.30$. If your initial loss is significantly different, something is likely wrong with the initialization or architecture. In `microGPT`, the first reported loss is ≈ 3.37 , which is close to this theoretical value.

7.2 The Adam Optimizer

After computing the loss and running backpropagation, `microGPT` updates each parameter θ_i using the Adam optimizer. Adam maintains two exponential moving averages per parameter:

Definition 7.1 (Adam Update Rule). Let $g_i^{(t)} = \frac{\partial \mathcal{L}}{\partial \theta_i}$ be the gradient at step t . Adam maintains:

$$m_i^{(t)} = \beta_1 m_i^{(t-1)} + (1 - \beta_1) g_i^{(t)} \quad (\text{first moment estimate}) \quad (29)$$

$$v_i^{(t)} = \beta_2 v_i^{(t-1)} + (1 - \beta_2) (g_i^{(t)})^2 \quad (\text{second moment estimate}) \quad (30)$$

With bias correction:

$$\hat{m}_i^{(t)} = \frac{m_i^{(t)}}{1 - \beta_1^{t+1}} \quad (31)$$

$$\hat{v}_i^{(t)} = \frac{v_i^{(t)}}{1 - \beta_2^{t+1}} \quad (32)$$

The parameter update is:

$$\theta_i \leftarrow \theta_i - \eta_t \cdot \frac{\hat{m}_i^{(t)}}{\sqrt{\hat{v}_i^{(t)} + \epsilon}} \quad (33)$$

where $\eta_t = \eta_0 \cdot (1 - t/N)$ is a linearly decaying learning rate.

Symbol	Value	Description
η_0	0.01	Initial learning rate
β_1	0.85	First moment decay (note: lower than typical 0.9)
β_2	0.99	Second moment decay (note: lower than typical 0.999)
ϵ	10^{-8}	Numerical stability constant
N	1000	Total training steps

Table 3: Adam optimizer hyperparameters.

The hyperparameter values used in microGPT are:

Listing 15: Adam optimizer update.

```

1 learning_rate, beta1, beta2, eps_adam = 0.01, 0.85, 0.99, 1e-8
2 m = [0.0] * len(params) # first moment buffers
3 v = [0.0] * len(params) # second moment buffers
4
5 # Inside training loop (after loss.backward()):
6 lr_t = learning_rate * (1 - step / num_steps) # linear decay
7 for i, p in enumerate(params):
8     m[i] = beta1 * m[i] + (1 - beta1) * p.grad # Eq. (26)
9     v[i] = beta2 * v[i] + (1 - beta2) * p.grad ** 2 # Eq. (27)
10    m_hat = m[i] / (1 - beta1 ** (step + 1)) # Eq. (28)
11    v_hat = v[i] / (1 - beta2 ** (step + 1)) # Eq. (29)
12    p.data -= lr_t * m_hat / (v_hat ** 0.5 + eps_adam) # Eq. (30)
13    p.grad = 0 # reset gradient for next step

```

Why Adam, not SGD?

Vanilla stochastic gradient descent ($p.data -= lr * p.grad$) can work but converges slowly and is sensitive to the learning rate. Adam’s per-parameter adaptive learning rates (via $\hat{v}_i$) and momentum (via $\hat{m}_i$) make it substantially more robust. The first moment $\hat{m}_i$ smooths out noisy gradients (like a rolling ball), while the second moment $\hat{v}_i$ normalizes the step size for each parameter, giving parameters with consistently large gradients smaller effective steps and vice versa.

7.3 The Complete Training Loop

Putting it all together, each training step performs:

1. **Sample:** Pick a document from the dataset (cyclically).
2. **Tokenize:** Convert to token ids, wrap with BOS.
3. **Forward:** Run the model autoregressively over all positions, computing per-position losses.
4. **Loss:** Average the per-position losses (eq. (28)).
5. **Backward:** Call `loss.backward()` to compute all gradients via eq. (2).
6. **Update:** Apply the Adam update (eq. (33)) to every parameter.
7. **Reset:** Zero all gradients for the next step.

Over 1,000 steps, the loss decreases from ≈ 3.37 (random guessing) to ≈ 2.37 , indicating the model has learned significant statistical structure in the names data.

8 Inference

At inference time, the parameters are frozen and the model generates new tokens autoregressively. The process starts with the BOS token and repeatedly samples from the model's predicted distribution until another BOS is produced (signaling the end of the name) or the maximum length is reached.

8.1 Temperature Sampling

Before applying softmax during inference, the logits are divided by a temperature parameter $\tau > 0$:

$$p_i = \text{softmax}\left(\frac{z}{\tau}\right)_i = \frac{e^{z_i/\tau}}{\sum_{j=1}^V e^{z_j/\tau}} \quad (34)$$

The temperature controls the sharpness of the distribution:

- $\tau \rightarrow 0^+$: The distribution collapses to a point mass on the highest-logit token (greedy decoding).
- $\tau = 1$: The model's native learned distribution is used.
- $\tau \rightarrow \infty$: The distribution approaches uniform over all tokens.

microGPT uses $\tau = 0.5$, which sharpens the distribution somewhat, making outputs more conservative and coherent.

Listing 16: Inference loop with temperature sampling.

```

1 temperature = 0.5
2 for sample_idx in range(20):
3     keys, values = [[] for _ in range(n_layer)], \
4                     [[] for _ in range(n_layer)]
5     token_id = BOS
6     sample = []
7     for pos_id in range(block_size):
8         logits = gpt(token_id, pos_id, keys, values)
9         probs = softmax([l / temperature for l in logits]) # Eq.
10                (31)
11                token_id = random.choices(
12                    range(vocab_size),
13                    weights=[p.data for p in probs]
14                )[0]
15                if token_id == BOS:
16                    break
17                sample.append(uchars[token_id])
18    print(f"sample {sample_idx+1:2d}: {''.join(sample)}")

```

9 Connecting to Production LLMs

microGPT and a model like GPT-4 or Claude share the same fundamental algorithm. Every difference between them is a matter of scale and engineering efficiency rather than algorithmic novelty. The following table summarizes the key differences:

None of these differences alter the core loop: encode tokens $\rightarrow$ embed $\rightarrow$ transform (attention + MLP) $\rightarrow$ predict next token $\rightarrow$ compute loss $\rightarrow$ backpropagate $\rightarrow$ update parameters. A conversation with ChatGPT is, from the model's perspective, just a document completion—exactly like microGPT completing a name.

Component	microGPT	Production LLMs
Data	32K names	Trillions of tokens of internet text
Tokenizer	Character-level (27 tokens)	BPE / SentencePiece (~100K tokens)
Autograd	Scalar <code>Value</code> in Python	Tensor ops on GPU (PyTorch/JAX)
Architecture	4,192 params, 1 layer	$10^{11}+$ params, 100+ layers
Normalization	RMSNorm (no learnable params)	RMSNorm with learnable gain
Activation	ReLU	SwiGLU / GeLU
Positional	Learned absolute	RoPE (relative)
Attention	Full MHA	GQA (Grouped Query Attention)
Optimizer	Adam ($\beta_1=0.85$)	AdamW with warmup + cosine decay
Training	1 doc/step, 1000 steps	Millions of tokens/step, months of compute
Post-training	None	SFT + RLHF/RLAIF

Table 4: Comparison between microGPT and production LLMs.

10 Summary of All Equations

For reference, we collect the key equations that define the forward pass of microGPT:

Embedding (section 6.2):

$$\mathbf{x}^{(0)} = \text{RMSNorm}(W^{\text{te}}[t, :] + W^{\text{pe}}[p, :])$$

For each layer $\ell = 0, \dots, L - 1$:

Attention (section 6.3):

$$\begin{aligned} \mathbf{q} &= W_\ell^Q \cdot \text{RMSNorm}(\mathbf{x}), \quad \mathbf{k} = W_\ell^K \cdot \text{RMSNorm}(\mathbf{x}), \quad \mathbf{v} = W_\ell^V \cdot \text{RMSNorm}(\mathbf{x}) \\ \alpha_s^{(h)} &= \frac{\mathbf{q}^{(h)} \cdot \mathbf{k}_s^{(h)}}{\sqrt{d_h}}, \quad \mathbf{w}^{(h)} = \text{softmax}(\boldsymbol{\alpha}^{(h)}), \quad \mathbf{h}^{(h)} = \sum_s w_s^{(h)} \mathbf{v}_s^{(h)} \\ \mathbf{x} &\leftarrow W_\ell^O \cdot [\mathbf{h}^{(0)} \parallel \dots \parallel \mathbf{h}^{(H-1)}] + \mathbf{x}_{\text{residual}} \end{aligned}$$

MLP (section 6.5):

$$\mathbf{x} \leftarrow W_\ell^{\text{fc2}} \cdot \text{ReLU}(W_\ell^{\text{fc1}} \cdot \text{RMSNorm}(\mathbf{x})) + \mathbf{x}_{\text{residual}}$$

Output (section 6.6):

$$\mathbf{z} = W^{\text{lm}} \mathbf{x}, \quad \mathbf{p} = \text{softmax}(\mathbf{z})$$

Loss (section 7.1):

$$\mathcal{L} = -\frac{1}{n} \sum_{t=0}^{n-1} \log p_{s_{t+1}}^{(t)}$$

Acknowledgments

This document is based entirely on Andrej Karpathy’s microGPT project (MIT License), available at <https://gist.github.com/karpathy/8627fe009c40f57531cb18360106ce95>, and his accompanying blog post at <https://karpathy.github.io/2026/02/12/microgpt/>. All math-

emational formulations presented here are derived directly from the source code to ensure exact correspondence.