Une orange sur la table
Transformer language models from scratch

Antoine Gourru & ChatGPT

March 2026

1 Language Modeling

1.1 The Language Modeling Objective

Let a sequence of discrete tokens be

$$(x_1, x_2, \ldots, x_T), \quad x_t \in \mathcal{V}, \tag{1}$$

where $\mathcal{V}$ is a finite vocabulary.

By the chain rule of probability, any joint distribution can be written as:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}). \tag{2}$$

Thus, language modeling reduces to learning conditional next-token probabilities:

$$P_\theta(x_t \mid x_{<t}), \tag{3}$$

parameterized by a neural network with parameters θ.

Autoregressive language models explicitly parameterize each conditional distribution $P_\theta(x_t \mid x_{<t})$ and impose a causal structure so that predictions depend only on past tokens.

1.2 Maximum Likelihood Training

Given a dataset $\mathcal{D}$ of sequences, training maximizes the log-likelihood:

$$\mathcal{L}(\theta) = \sum_{(x_1,\ldots,x_T)\in\mathcal{D}} \sum_{t=1}^{T} \log P_\theta(x_t \mid x_{<t}). \tag{4}$$

Equivalently, we minimize the negative log-likelihood (cross-entropy loss):

$$\mathcal{J}(\theta) = -\mathbb{E}_{x\sim\mathcal{D}} \sum_{t=1}^{T} \log P_\theta(x_t \mid x_{<t}). \tag{5}$$

1.3 Softmax Parameterization

Neural language models produce logits:

$$\mathbf{z}_t \in \mathbb{R}^{|\mathcal{V}|}. \tag{6}$$

Probabilities are obtained via softmax:

$$P_\theta(x_t = i \mid x_{<t}) = \frac{\exp(z_{t,i})}{\sum_{j=1}^{|\mathcal{V}|} \exp(z_{t,j})}. \tag{7}$$

1.4 Perplexity

Model quality is commonly evaluated using perplexity, defined as the exponential of the average negative log-likelihood per token:

$$\mathrm{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T} \log P_\theta(x_t \mid x_{<t})\right). \tag{8}$$

Perplexity is lower when the model assigns higher probability to the observed sequence (i.e., is less "surprised" on average). Lower perplexity indicates better predictive performance, when evaluated under the same dataset and tokenization.

1.5 Autoregressive Generation

After training, generation proceeds sequentially:

1. Initialize with prompt $(x_1, \ldots, x_k)$.
2. Sample $x_{k+1} \sim P_\theta(\cdot \mid x_{\leq k})$.
3. Repeat.

Sampling strategies include:

- Greedy decoding
- Temperature sampling
- Top-k sampling
- Nucleus (top-p) sampling

1.6 Why Attention?

Traditional language models (e.g., RNNs) compute:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, x_t), \tag{9}$$

which compresses all history into a fixed-size hidden state.

Attention-based models instead compute:

$$\mathbf{h}_t = \sum_{i<t} \alpha_{ti} \mathbf{v}_i, \tag{10}$$

where weights α_{ti} are learned dynamically.

This allows direct access to all previous tokens, enabling modeling of long-range dependencies without recurrence.

2 Tokenization and Subword Modeling

Neural language models operate on discrete symbols drawn from a finite vocabulary $\mathcal{V}$. Since natural language is composed of variable-length words and an open vocabulary, a tokenizer is required to map raw text into a sequence of discrete tokens.

2.1 The Tokenization Problem

Given raw text:

$$\texttt{"Transformers are powerful"} \tag{11}$$

the tokenizer produces:

$$(x_1, x_2, \ldots, x_T), \quad x_t \in \mathcal{V}. \tag{12}$$

The design of $\mathcal{V}$ must balance:

- Expressivity (rare words should be representable)

- Vocabulary size (softmax cost is $O(|\mathcal{V}|)$)
- Sequence length (smaller tokens $\Rightarrow$ longer sequences)

Modern language models use **subword tokenization**, which decomposes text into frequent character sequences.

2.2 Byte Pair Encoding (BPE)

Byte Pair Encoding constructs a vocabulary by iteratively merging frequent symbol pairs.

Initialization. Start with a character-level vocabulary:

$$\mathcal{V}_0 = \{\text{all characters in corpus}\}. \tag{13}$$

Represent each word as a sequence of characters plus an end-of-word marker.

Merge Step. At each iteration:

1. Count all adjacent symbol pairs in the corpus.
2. Find the most frequent pair (a, b).
3. Merge it into a new symbol ab.
4. Add ab to the vocabulary.

After K merges:

$$|\mathcal{V}| = |\mathcal{V}_0| + K. \tag{14}$$

Objective Interpretation. BPE greedily compresses the corpus by maximizing frequency of merged pairs. It approximates minimizing description length of the dataset.

2.3 Unigram Language Model Tokenization

An alternative is the Unigram LM tokenizer.

Instead of merges, it assumes a probabilistic model over subwords:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(s_t), \tag{15}$$

where s_t are subword units.

Training proceeds by:

1. Initializing a large candidate vocabulary.
2. Estimating subword probabilities via EM.
3. Iteratively pruning low-probability tokens.

This directly optimizes likelihood of the corpus under a segmentation model.

2.4 Training Objective

Given a corpus $\mathcal{C}$, tokenizer training solves:

$$\max_{\mathcal{V}} \sum_{x \in \mathcal{C}} \log P(x \mid \mathcal{V}), \tag{16}$$

subject to $|\mathcal{V}| \leq V_{max}$.

For BPE, this is approximated greedily. For Unigram LM, it is optimized via EM.

2.5 Byte-Level Tokenization

Modern LLMs often operate at the byte level.

Let text be encoded as UTF-8 bytes:

$$b_1, b_2, \dots, b_n, \quad b_i \in \{0, \dots, 255\}. \tag{17}$$

Advantages:

- No unknown tokens
- Language-independent
- Robust to arbitrary input

Subword merges are then learned over byte sequences.

2.6 Trade-offs

Let:

- $|\mathcal{V}|$ = vocabulary size
- L = sequence length
- d = model dimension

The output softmax cost is:

$$O(L|\mathcal{V}|). \tag{18}$$

Smaller vocabulary $\Rightarrow$ longer sequences (larger L). Larger vocabulary $\Rightarrow$ heavier softmax computation.
Modern models typically use:

$$|\mathcal{V}| \in [32\text{k}, 200\text{k}]. \tag{19}$$

2.7 Tokenizer as a Learned Compression Scheme

Tokenization can be viewed as learning a compression map:

$$\text{Text} \to \text{Discrete Codes}. \tag{20}$$

Good tokenizers:

- Reduce sequence entropy
- Preserve semantic boundaries
- Avoid excessive fragmentation

The tokenizer therefore defines the discrete modeling problem that the Transformer ultimately solves.

3 The Transformer Architecture from First Principles

The Transformer is a sequence-to-sequence architecture built entirely from attention mechanisms and position-wise feed-forward networks, without recurrence or convolution.

Let an input sequence of tokens be:

$$(x_1, x_2, \dots, x_L), \quad x_i \in \mathcal{V} \tag{21}$$

where $\mathcal{V}$ is a vocabulary.

3.1 Token Embeddings

Each token is mapped to a continuous vector space:

$$\mathbf{e}_i = E[x_i], \quad E \in \mathbb{R}^{|\mathcal{V}| \times d_{model}}. \tag{22}$$

This yields an embedding matrix:

$$\mathbf{X} \in \mathbb{R}^{L \times d_{model}}. \tag{23}$$

3.2 Positional Encoding

Since the model contains no recurrence, position information must be injected.

The original Transformer uses sinusoidal positional encodings:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \tag{24}$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right). \tag{25}$$

The initial hidden states become:

$$\mathbf{H}^{(0)} = \mathbf{X} + \mathbf{PE}. \tag{26}$$

3.3 Scaled Dot-Product Attention

The fundamental operation of the Transformer is attention.

Given queries Q, keys K, and values V:

$$Q \in \mathbb{R}^{L\times d_k}, \quad K \in \mathbb{R}^{L\times d_k}, \quad V \in \mathbb{R}^{L\times d_v}, \tag{27}$$

attention is defined as:

$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V. \tag{28}$$

The scaling factor $\frac{1}{\sqrt{d_k}}$ prevents the dot products from growing large in magnitude, which would push the softmax into regions with extremely small gradients.

3.4 Multi-Head Attention

Instead of performing a single attention operation, the Transformer uses H parallel attention heads.

For head h:

$$Q_h = \mathbf{H}W_h^Q, \tag{29}$$

$$K_h = \mathbf{H}W_h^K, \tag{30}$$

$$V_h = \mathbf{H}W_h^V. \tag{31}$$

Each head computes:

$$\mathbf{O}_h = \mathrm{Attention}(Q_h, K_h, V_h). \tag{32}$$

Outputs are concatenated and projected:

$$\mathrm{MHA}(\mathbf{H}) = \mathrm{Concat}(\mathbf{O}_1, \dots, \mathbf{O}_H)W^O. \tag{33}$$

This allows the model to attend to information from different representation subspaces simultaneously.

3.5 Position-wise Feed-Forward Network

Each position is processed independently by a two-layer MLP:

$$\mathrm{FFN}(\mathbf{x}) = \mathrm{ReLU}(\mathbf{x}W_1 + b_1)W_2 + b_2. \tag{34}$$

This increases representational capacity and mixes features within each token.

3.6 Residual Connections and Layer Normalization

Each sublayer is wrapped with residual connections:

$$\mathbf{H}' = \mathrm{LayerNorm}(\mathbf{H} + \mathrm{MHA}(\mathbf{H})), \tag{35}$$

$$\mathbf{H}^{(l+1)} = \mathrm{LayerNorm}(\mathbf{H}' + \mathrm{FFN}(\mathbf{H}')). \tag{36}$$

Residual connections enable stable gradient flow, while LayerNorm stabilizes activation statistics.

3.7 Stacking Layers

A Transformer encoder stacks N identical layers:

$$\mathbf{H}^{(l+1)} = \text{TransformerLayer}(\mathbf{H}^{(l)}), \quad l = 0, \dots, N-1. \tag{37}$$

3.8 Decoder Architecture (Original Transformer)

The original Transformer is encoder–decoder.

The decoder contains:

1. Masked self-attention
2. Encoder–decoder cross-attention
3. Feed-forward network

Masked attention ensures autoregressive behavior:

$$\text{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}}\right), \tag{38}$$

where $M_{ij} = -\infty$ if $j > i$.

3.9 Output Projection

The final hidden states are projected to vocabulary logits:

$$\mathbf{z}_i = \mathbf{h}_i W_{out}, \tag{39}$$

$$P(x_{i+1} \mid x_{\le i}) = \text{softmax}(\mathbf{z}_i). \tag{40}$$

3.10 Computational Complexity

For sequence length L:

- Attention: $O(L^2 d)$
- Feed-forward: $O(Ld^2)$

The quadratic term in L is the main scalability bottleneck.

3.11 Summary

A Transformer layer consists of:

1. Multi-Head Attention
2. Residual + LayerNorm
3. Feed-Forward Network
4. Residual + LayerNorm

Stacking these layers yields a deep attention-based architecture capable of modeling long-range dependencies without recurrence.

4 Modern Modifications to the Original Transformer

This section summarizes key architectural improvements over the original *Attention Is All You Need* Transformer.

4.1 Pre-LayerNorm instead of Post-LayerNorm

Original (Post-LN). The original Transformer applied Layer Normalization after each residual block:

$$\mathbf{x}_{l+1} = \text{LayerNorm}\big(\mathbf{x}_l + \mathcal{F}(\mathbf{x}_l)\big), \tag{41}$$

where $\mathcal{F}$ denotes either the attention or feed-forward sublayer.

While functional for shallow networks, Post-LN suffers from unstable gradient flow for deep Transformers, as gradients must pass through the normalization operator at every layer.

Modern (Pre-LN). Modern architectures instead normalize before the sublayer:

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \mathcal{F}\big(\text{LayerNorm}(\mathbf{x}_l)\big). \tag{42}$$

This improves gradient propagation since the residual path becomes an identity mapping:

$$\frac{\partial \mathbf{x}_{l+1}}{\partial \mathbf{x}_l} \approx I + \frac{\partial \mathcal{F}}{\partial \mathbf{x}_l}. \tag{43}$$

This simple reordering enables stable training at 100+ layers.

4.2 RoPE / ALiBi instead of Sinusoidal Positional Encoding

Original sinusoidal encoding. The original model added fixed positional encodings:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \tag{44}$$

$$\text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right), \tag{45}$$

which are added to token embeddings:

$$\mathbf{h}_0 = \mathbf{E}_{token} + \mathbf{PE}. \tag{46}$$

Rotary Positional Embeddings (RoPE). RoPE encodes position by rotating query and key vectors in complex space. For a 2D pair (x_1, x_2):

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \tag{47}$$

where θ depends on token position.

Attention becomes:

$$\text{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{(R\mathbf{Q})(R\mathbf{K})^\top}{\sqrt{d}}\right)\mathbf{V}, \tag{48}$$

where R applies position-dependent rotations.

This yields relative position awareness and better extrapolation.

ALiBi. ALiBi instead adds a linear bias to attention scores:

$$\text{score}_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d}} + m_h(i - j), \tag{49}$$

where m_h is a head-specific slope. This enables length extrapolation without modifying embeddings.

4.3 SwiGLU instead of ReLU

Original Feed-Forward (ReLU).

$$\text{FFN}(\mathbf{x}) = \text{ReLU}(\mathbf{x}W_1 + b_1)W_2 + b_2. \tag{50}$$

Modern Gated FFN (SwiGLU). Modern LLMs use a gated activation:

$$\mathrm{SwiGLU}(\mathbf{x}) = (\mathbf{x}W_1 \odot \mathrm{SiLU}(\mathbf{x}W_2))\, W_3. \tag{51}$$

Here,

$$\mathrm{SiLU}(x) = x\sigma(x). \tag{52}$$

The gating mechanism improves expressivity and parameter efficiency, leading to better scaling behavior.

4.4 Multi-Query Attention (MQA)

Standard Multi-Head Attention. For H heads:

$$\mathbf{Q}_h = \mathbf{X}W_h^Q, \quad \mathbf{K}_h = \mathbf{X}W_h^K, \quad \mathbf{V}_h = \mathbf{X}W_h^V. \tag{53}$$

Each head has separate Q, K, V projections, giving memory cost:

$$O(H \cdot L \cdot d_k) \tag{54}$$

for key/value caching.

Multi-Query Attention. MQA shares keys and values across heads:

$$\mathbf{Q}_h = \mathbf{X}W_h^Q, \quad \mathbf{K} = \mathbf{X}W^K, \quad \mathbf{V} = \mathbf{X}W^V. \tag{55}$$

Memory cost reduces to:

$$O(L \cdot d_k). \tag{56}$$

This dramatically reduces KV-cache size during autoregressive inference.

4.5 FlashAttention

Standard attention computes:

$$\mathbf{A} = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right), \quad \mathbf{O} = \mathbf{A}\mathbf{V}. \tag{57}$$

Naively, this requires materializing the $L \times L$ attention matrix, with $O(L^2)$ memory.

FlashAttention idea. FlashAttention reorders computation to avoid storing $\mathbf{A}$. It tiles the computation:

1. Load a block of Q, K, V into fast SRAM.
2. Compute partial attention scores.
3. Apply numerically stable online softmax:

$$m_i = \max_j s_{ij}, \tag{58}$$

$$\ell_i = \sum_j e^{s_{ij} - m_i}, \tag{59}$$

4. Accumulate output without materializing $\mathbf{A}$.

This reduces memory from $O(L^2)$ to $O(Ld)$ while preserving exact attention.

4.6 Summary

Modern Transformers retain the core attention mechanism:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V, \tag{60}$$

but incorporate architectural and systems-level improvements that enable stable training at massive scale and efficient inference.

A Reference Python Implementation

```python
# -*- coding: utf-8 -*-

# Language Modeling

## 1. Bullding Vocab Space


text = "<start> Une orange sur la table , \n ta robe sur le tapis \n et toi dans mon lit . \n doux present du present , \n fraicheur de la nuit , \n chaleur de ma vie <end>"

import numpy as np

tokenized = text.split(" ")

vocab = list(set(tokenized))
v = len(vocab)
word2idx = {w: i for i, w in enumerate(vocab)}
idx2word  = {v: k for k, v in word2idx.items()}

ids = [word2idx.get(w) for w in tokenized]

length = len(ids)

matrix = np.zeros((length,v))

matrix[np.arange(length), ids] = 1

matrix

## 1. A simple model


import torch.nn as nn
import torch

target = torch.Tensor(matrix[1:]).unsqueeze(0)
input = torch.Tensor(matrix[:-1]).unsqueeze(0)

def generate_sequence(model, word2idx, idx2word, start_token='<start>', max_length=20):
    model.eval()
    sequence_ids = [word2idx[start_token]]

    with torch.no_grad():
        for _ in range(max_length - 1):
            last_word_id = sequence_ids[-1]

            input_tensor = torch.zeros(1, 1, len(word2idx))
            input_tensor[0, 0, last_word_id] = 1

            output = model(input_tensor)

            probabilities = torch.softmax(output[0, -1, :], dim=-1)

            next_word_id = torch.argmax(probabilities).item()

            sequence_ids.append(next_word_id)

            if idx2word.get(next_word_id) == '<end>':
                break
    generated_words = [idx2word.get(idx, '<unk>') for idx in sequence_ids]
    return ' '.join(generated_words)

class SimpleMLP(nn.Module):
    def __init__(self, input_size, hidden_size):
        super().__init__()
        self.embedding = nn.Linear(input_size, hidden_size, bias = False)
        self.fc1 = nn.Linear(hidden_size, hidden_size)
        self.relu = nn.ReLU()
        self.decode = nn.Linear(hidden_size, input_size, bias = False)
        self.decode.weight = nn.Parameter(self.embedding.weight.transpose(0, 1))

    def forward(self, x):
        x = self.embedding(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.decode(x)
        return x

model = SimpleMLP(input_size=v, hidden_size=128)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

for epoch in range(100):
    optimizer.zero_grad()
    output = model(input)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    if epoch % 10 == 0:
      print(f"Epoch {epoch+1}, Loss: {loss.item()}")

generated_text = generate_sequence(model, word2idx, idx2word, start_token='<
    start>', max_length=50)
print("Generated Text:", generated_text)

## 2. A less shitty Model : basic transformer

import math

def create_positional(max_L,emb):
        posit = torch.zeros(max_L,emb)
        for i in range(emb):
            if (i % 2) == 0:
                posit[0,i] = 0
            if (i % 2) == 1:
                posit[0,i] = 1
        for k in range(1,max_L):
            posit[k,0] = 0
            for i in range(emb):
                if (i % 2) == 0:
                    posit[k,i] = math.sin(k/(math.pow(k,(i/emb))))
                if (i % 2) == 1:
                    posit[k,i] = math.cos(k/(math.pow(k,((i-1)/emb))))
        return posit

class SimpleTransformer(nn.Module):
    def __init__(self, max_L, input_size, hidden_size):
        super().__init__()

        self.util_emb = math.pow(hidden_size,1/2)
        self.positional = create_positional(max_L,hidden_size)
```

```python

        self.embedding = nn.Linear(input_size, hidden_size, bias = False)

        self.key = torch.nn.Linear(hidden_size,hidden_size)
        self.query = torch.nn.Linear(hidden_size,hidden_size)
        self.value = torch.nn.Linear(hidden_size,hidden_size)
        self.proj = torch.nn.Linear(hidden_size,hidden_size)

        self.decode = nn.Linear(hidden_size, input_size, bias = False)
        self.decode.weight = nn.Parameter(self.embedding.weight.transpose(0, 1))

    def forward(self, x):
        x = self.embedding(x)

        pos = self.positional[:x.shape[1],:x.shape[2]]
        x += pos

        key = self.key(x)
        value = self.value(x)
        query = self.query(x)

        attention = (query @ key.transpose(-1, -2)) / self.util_emb
        mask = torch.tril(torch.ones(attention.shape[1], attention.shape[2]))

        attention = attention.masked_fill(mask == 0, float('-inf'))
        attention = torch.softmax(attention, dim=-1)

        x = x + self.proj(attention @ value)

        x = self.decode(x)

        return x


max_L = 100
model = SimpleTransformer(max_L, v,2)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

for epoch in range(100):
    optimizer.zero_grad()
    output = model(input)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    if epoch % 10 == 0:
      print(f"Epoch {epoch+1}, Loss: {loss.item()}")

generated_text = generate_sequence(model, word2idx, idx2word, start_token='<
    start>', max_length=50)
print("Generated Text:", generated_text)
```