LLM Intro Course: 1 – simpleGPT: end-to-end pipeline

History, tokenizer, tensors, causal self-attention, training, metrics

Simon Vary

Mathematical Institute, University of Oxford

March 4, 2026

Hands-on Introduction to LLMs

Course Goal: Comprehensive overview of the modern LLM lifecycle in four weeks.

Able to: understand open-weight architectures, be able to finetune and run experiments.

The 4-Week Roadmap:

1. **The Core Pipeline:** Tokenizers, causal self-attention, and training `simpleGPT`.
2. **Architecture:** Position encodings (RoPE), modern attention variants, and normalization.
3. **Inference:** The KV-cache, prefill vs. decode, and long-context bottlenecks.
4. **Post-Training:** Supervised fine-tuning (SFT), LoRA, and preference learning.

A (Very) Brief History of Language Modelling

Language modelling

- representation

$$p(w_{1:T}) = \prod_{t=1}^{T} p(w_t \mid w_{1:t-1})$$

  where $w_{i:j} = \{w_i, w_{i+1}, \ldots, w_{j-1}, w_j\}$.
- With $p(w_{1:T})$ we can:
  - rank sequences (likelihood / perplexity)
  - translation via $p(w_{1:T} \mid s_{1:T}) \propto p(s_{1:T} \mid w_{1:T})\, p(w_{1:T})$

Problem: data sparsity + curse of dimensionality

- Huge vocabulary – many different words
- Long contexts – many different word sequences

- N-grams: Markov assumption

$$p(w_t \mid w_{1:t-1}) \approx p(w_t \mid w_{t-N+1:t-1})$$

  but it destroys long-range context.

Early Word Embeddings **(Bengio et al., 2003)**

- Learn an embedding matrix $C \in \mathbb{R}^{V \times d}$ and a predictor f_Θ

$$p(w_t \mid w_{1:t-1}) \approx f_\Theta\big(c_{I(w_{t-1})}, \ldots, c_{I(w_{t-N})}\big),$$

where $I(w) \in \{1, \ldots, V\}$ indexes the vocabulary and $c_{I(w)}$ is the corresponding row of C.

Idea 1: Smoothness ⇒ generalization

- Discrete words have no notion of "small change"; vectors do.
- If "cat" and "dog" have close embeddings and f_Θ is smooth, then probability mass transfers to *neighbors* in sentence space. *Cat is walking on a road* ≈ *Dog is walking on a road*.

Training and early limitations

- **Approach:** Next token prediction from N previous; optimize cross-entropy over V words.
- **The N-gram Era:** Pre-GPU compute and shallow architectures:
  - *"[N]eural probabilistic language models remain far less widely used than N-gram models due to their notoriously long training times..."* (Mnih & Teh, 2012)
- Overcoming required new hardware (GPUs) and architectures (RNNs, Transformers).

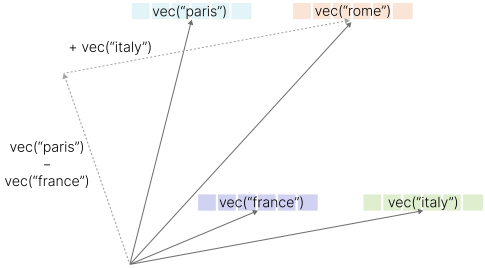
word2vec **(Mikolov et al., 2013)**

- **The AlexNet (2012):** Scale efficient architecture + GPU training.
- **But** early neural language models too expensive for large vocabularies.

Idea 2: Efficiency ⇒ Train on massive data ⇒ Rich representations

- **Simple architecture:** Remove non-linear hidden layers (only a log-linear model).
- **Trick for "cheap" softmax:** Compute probabilities only for the vocabulary V that is important.

- **Emergent Structure:** Simple dot products learned complex geometry.
- Analogies became linear math:
  $\text{vec(King)} - \text{vec(Man)} + \text{vec(Woman)} \approx \text{vec(Queen)}$



Linear structure emerges in the embedding space.

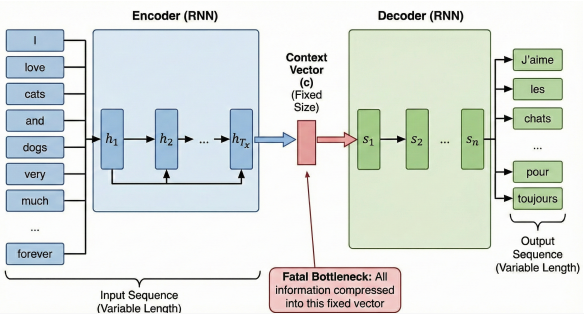
Sequence-to-Sequence Models **(Sutskever et al., 2014)**

Problem: Variable Length Context

- word2vec embeddings are learned using a *fixed context window*
- no mechanism to capture variable-length, sentence-level dependencies.

Idea 3: Learn embeddings for variable length contexts

- **seq2seq:** Encoder–Decoder to learn a variable-length input X to a variable-length output Y.



Adaptive Context via Attention **(Bahdanau et al., 2014)**

Problem: Long sequences don't fit into a single state

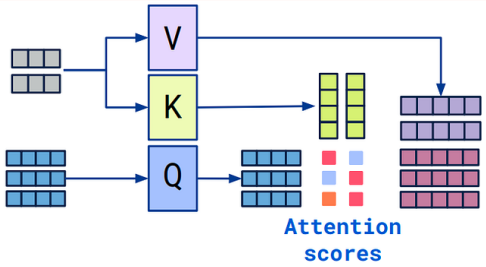
- A single vector c lacks the capacity for complex, long-range meaning.

Idea 4: Don't try to compress into a single state

- Keep encoder states for *all* tokens and *let the decoder decide* what is relevant.
- Dynamic c_i as a weighted sum of all token hidden states $c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$

Translating to modern QKV (Cross-Attention):

- **Query (q_i):** What the decoder wants ← Previous decoder state s_{i-1}
- **Key/Value (k_j, v_j):** What the encoder holds ← Encoder states h_j



The Transformer **(Vaswani et al., 2017)**

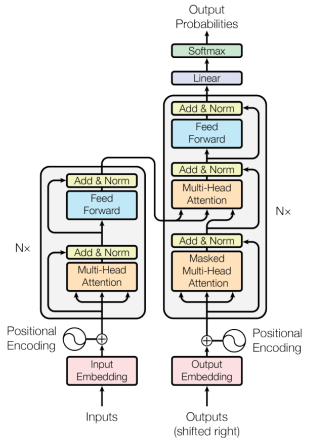
- **Sequential representations:** RNNs decoders are inherently sequential (h_t depends on h_{t-1}). This precludes parallelization within a sequence, strictly limiting training scale.
- **Idea 5:** *"Attention is All You Need."* Remove recurrence entirely.

Scaled Dot-Product Attention

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

Architectural changes:

- **Self-Attention:** Remove RNN. Words attend to other words in the *same* sequence to build representations.
- **Positional Encoding:** Attention processes everything at once, must add positions.
- **Multi-Head:** Multiple attention representations in parallel.



Generative Pre-Training **(Radford et al., 2019 - GPT-2)**

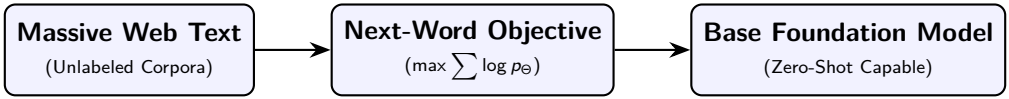
- **Massive data:** Use massive unlabeled web corpora (instead of labeled).
- **Decoder Only:** Train decoder-only Transformer using *next-word prediction*.

The Objective: Maximum Likelihood Estimation

Maximize the probability of the next token w_t given all previous context $w_{<t}$:

$$\max_{\Theta} \sum_{t=1}^{T} \log p_{\Theta}(w_t \mid w_{1:t})$$

- **Allows for scale:** No human labeling is required.
- **Takeaway:** Data scale $+$*pure* next word prediction $\rightarrow$ learn *syntax*, *semantics*, and *world knowledge* to perform translation, summarization, and QA.

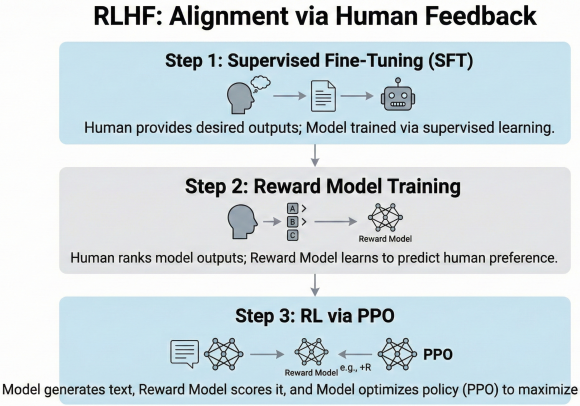


Alignment via RLHF **(Ouyang et al., 2022 - InstructGPT)**

- **The Problem:** Base models predict statistical likelihood, not human values (can be toxic, unhelpful, or ignore instructions).
- **The Solution:** RLHF shifts the training objective from *likelihood* to *human preference*.

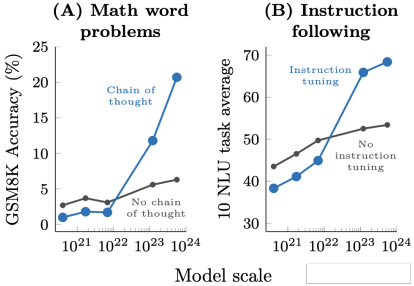
Three steps:

1. **SFT:** Supervised fine-tuning on high-quality human demonstrations.
2. **Reward Model:** Train a "critic" network on human rankings to predict preference.
3. **RL (PPO):** Optimize the model's policy to maximize that reward score.



Emergent Abilities of LLMs **(Wei et al., 2022)**

- **Emergent ability:** Not present at small scale.
- **Phase Shift:** A few-shot prompted tasks sharply works well for large models.
- **Measuring Scale:** Training FLOPS.
- **Unpredictability:** Cannot be predicted looking at scaling plots.



The plan for today

- **Tokenizer:** text $\rightarrow$ token IDs (and back), BPE merges.
- **Torch tensor basics:** shapes, dtype, device, broadcasting, einsum.
- **Causal self-attention:** QKV projections, masking, softmax, output projection.
- **Training loop:** batching, loss, backprop, optimizer step, sampling.

Tokenizers

Tokenizers

Question

What is the right granularity to discretize text?

Summary

- Map between strings to tokens (indices)
- Character-based, byte-based, word-based tokenization highly suboptimal
- BPE is an effective heuristic that looks at corpus statistics
- Why not do it on bytes?

Sources

- Karpathy: Let's build the GPT Tokenizer `https://youtu.be/zduSFxRajkE`

Tokenization: representation choices & vocabulary/length tradeoff

- Raw text must be converted into discrete units → *tokenization*
- The set of possible tokens defines the model's **vocabulary**

Tokenization ideas

- **Character-level**, e.g., "dog" → ['d', 'o', 'g']
- **UTF-8 code level**, e.g., "café" → [99, 97, 102, 195, 169]
- **Word-level**, e.g., "The dog runs fast." → ['The', 'dog', 'runs', 'fast', '.']
- **Subword-level**, e.g., *Byte Pair Encoding*, "happening" → ['happen', 'ing']

The tradeoff

Vocabulary size V vs **Sequence length** T (and attention cost $\sim T^2$).

Byte Pair Encoding (BPE): Learning the vocabulary

Idea: Learn the vocabulary from data

Start from a fine-alphabet (e.g. bytes) and iteratively *merge frequent adjacent pairs*.

BPE toy example)

1. Corpus with counts:

```
"hug": 10, "pug": 5, "pun": 12, "bun": 4,
"hugs": 5
```

2. Split into characters:

```
("h", "u", "g"): 10
("p", "u", "g"): 5
("p", "u", "n"): 12 ...
```

3. Find the most frequent pair:

The pair `("u", "g")` appears $10 + 5 + 5 = 20$ times.

4. Merge Rule #1: `"u" + "g"` $\rightarrow$ `"ug"`

Updated Corpus:

```
("h", "ug"): 10
("p", "ug"): 5
("p", "u", "n"): 12 ...
```

BPE Algorithm

Initialize vocab V with all base bytes/chars.

1. Count adjacent pairs (a, b).
2. Pick the most frequent pair.
3. Create new token $c \leftarrow ab$.
4. Repeat M times.

Final vocab size: $|V_{\text{initial}}| + M$.

After tokenization: IDs $\rightarrow$ vectors

- Tokenizer outputs integer IDs: $x_t \in \{1, \ldots, V\}$.
- Show `https://tiktokenizer.vercel.app/` or the Jupyter Notebook
- Convert IDs to vectors via an embedding table $E \in \mathbb{R}^{V \times d}$:

$$\text{one-hot}(x_t)^\top E \;=\; E_{x_t} \in \mathbb{R}^d \quad \text{(implemented as a lookup)}.$$

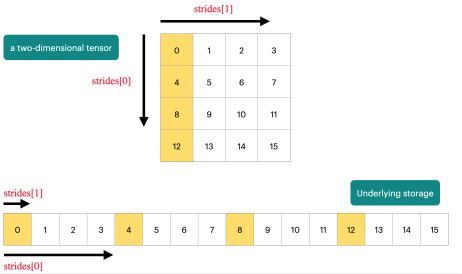
- Add positional information (absolute or relative) before attention.
- Transformer maps embeddings $\rightarrow$ logits in $\mathbb{R}^V$ for next-token prediction.

PyTorch Tensors

PyTorch tensor mental model

- **A tensor is:** *(1) storage + (2) metadata*
- Metadata: `shape`, `dtype`, `device`, `stride`
- Everything in an LLM is tensors: parameters, gradients, activations, optimizer state

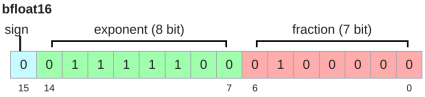
$$x[i,j] = \texttt{storage}\left[\texttt{offset} + i \cdot s_0 + j \cdot s_1\right]$$



Precision & dtype: memory vs stability vs speed

- **float32** (4 bytes): stable baseline, expensive
- **float16/bfloat16** (2 bytes): $\sim 2\times$ smaller, faster on GPUs
- **bf16 vs fp16:** bf16 keeps fp32-like *range* (fewer under/overflows)
- **fp8** (H100+): very fast/compact, needs care (scaling / mixed precision)

Rule of thumb: keep things low precision *when safe*, keep FP32 where it matters (e.g. some accumulations / optimizer state).



Storage & performance: views, strides, contiguity

- Many ops return a **view** (O(1)): shares storage, different strides
- `transpose/permute/slicing` often $\Rightarrow$ **non-contiguous** view
- `view(...)` requires stride-compatible layout; `reshape(...)` is safer
- `contiguous()` forces a **copy** (costly, but sometimes necessary)

LLM relevance: attention reshapes constantly $[B, T, C] \to [B, H, T, D]$. Accidental copies can dominate memory/time.

Transformer Decoder Block

Standard Decoder Block (Post-Norm)

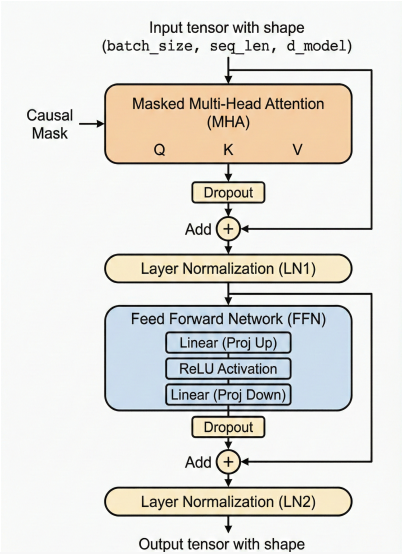
- **Original Vaswani et al. (2017) design**: Layer Normalization (LN) *after* the residual addition.
- **Step 1: Masked Self-Attention**

$$h = \text{LN}_1(x + \text{Dropout}(\text{MaskedAttn}(x)))$$

- **Step 2: Feed-Forward Network (FFN)**

$$x_{out} = \text{LN}_2(h + \text{Dropout}(\text{FFN}(h)))$$

- **Components:**
  - **MaskedAttn:** The causal mask ensures the model cannot look ahead ($M_{ij} = -\infty$ if $j > i$).
  - **FFN:** A two-layer network using a ReLU activation: $W_2(\text{ReLU}(W_1 h)) + b_2$.
  - **Regularization:** Dropout on outputs of both components.



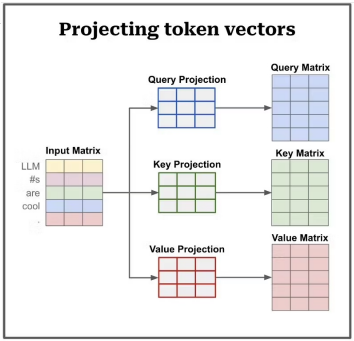
Multi-Headed Attention

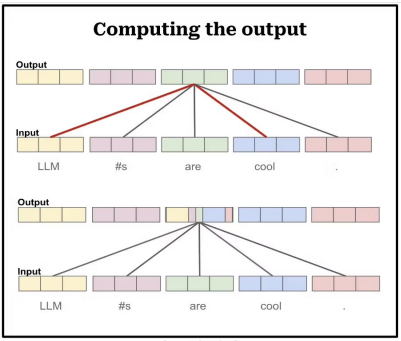
- Hidden activations $X \in \mathbb{R}^{T \times d}$, weights $W_\star \in \mathbb{R}^{d \times d_h}$, softmax $\sigma(\cdot)$:

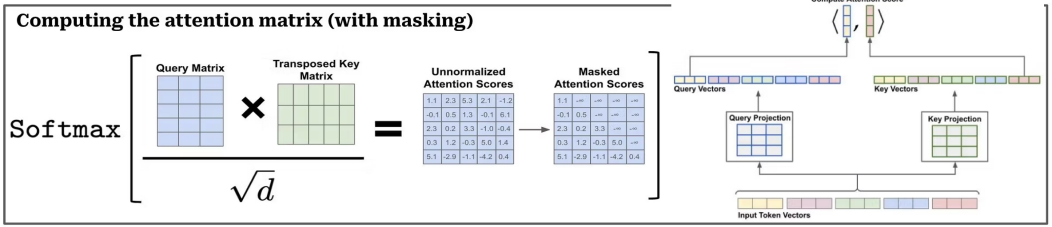
$$\text{head}_i = \underbrace{\sigma(QK^\top \odot M)}_{=A} V^\top$$

where $Q = X\,W_Q^i$, $K = X\,W_K^i$, $V = X\,W_V^i$.

- causal mask prevents looking ahead
- $\text{MHA}(X) = \text{Cat}(\{\text{head}_i\}_{i=1}^H) W_o$







Multi-Headed Attention Implementation

```
class MultiHeadAttentionFromScratch(nn.Module):
    def __init__(self, embed_dim: int, num_heads: int):
        super().__init__()

    def forward(self, x, causal_mask=None):
        B, L, D = x.shape
        # Step 1: project x into Q, K, V spaces (each: B, L, D)
        qkt = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(Q.size(-1))

        # each row becomes a probability distribution over positions
        weights = F.softmax(qkt, dim=-1)

        attn_out = torch.matmul(weights, V)
        return self.W_o(attn_out.transpose(1, 2).contiguous().view(B, L, D))
```

Transformer Block Implementation

```python
class TransformerBlock(nn.Module):
    """Structure:
      x -> MHA (causal self-attn) -> dropout -> +residual -> LN
        -> FFN (Linear->ReLU->Linear) -> dropout -> +residual -> LN
    """
    def forward(self, inputs, training: bool = False):
        _, seq_len, _ = inputs.shape

        mask = causal_attention_mask(seq_len, inputs.device)

        attention_output = self.mha(inputs, causal_mask=mask)

        attention_output = F.dropout(attention_output, p=self.dropout_rate, training=training)
        out1 = self.ln1(inputs + attention_output)

        ffn_output = self.lin1(out1)
        ffn_output = F.relu(ffn_output)
        ffn_output = self.lin2(ffn_output)
        ffn_output = F.dropout(ffn_output, p=self.dropout_rate, training=training)

        return self.ln2(out1 + ffn_output)
```

Transformer cost per layer

Let B=batch, T=sequence length, d=model dim, H=heads, $d_h = d/H$, $d_{\text{ff}} \approx 4d$.

Forward pass (dominant terms)

- **QKV projections:** $\approx 3\,BTd^2$
- **Attention scores + apply:** $\approx 2\,BHT^2 d_h \;=\; 2\,BT^2 d$
- **MLP (two matmuls):** $\approx 2\,BTdd_{\text{ff}} \approx 8\,BTd^2$

Backward pass (rule of thumb)

Backprop is typically $\sim$**2–3**$\times$ the forward compute.

Objectives, Training, and Metrics

Objective: next-token prediction

- Maximize the prediction probability of the correct next token:

$$P(w_{1:T}) = \prod_{t=1}^{T} p_\Theta(w_t \mid w_{<t})$$

- Model *outputs* raw scores (logits) $z_t = U^\top x_t \in \mathbb{R}^V$, via (un)-embedding matrix $U \in \mathbb{R}^{d \times V}$.
- Sample probability distribution over the vocabulary via softmax: $p_\Theta(\cdot \mid w_{1:t-1}) = \mathrm{softmax}(z_t)$.

Parallel "Teacher forcing" via the shift

- **Causal mask** + **positional encodings**, give sense of order
- Can feed the sequence in parallel and shift the targets:
  - **Inputs ($w_{1:t}$):** $[w_1, w_2, w_3, w_4]$ (e.g., "The", "cat", "sat", "on")
  - **Targets (w_{t+1}):** $[w_2, w_3, w_4, w_5]$ (e.g., "cat", "sat", "on", "the")

Loss: Cross-Entropy (Negative Log-Likelihood)

- **Maximize data probability:** Minimize the Negative Log-Likelihood (NLL).

Sequence Loss

For a single sequence of length T, the loss is the average NLL across all predicted tokens:

$$\mathcal{L}(\Theta) = -\frac{1}{T-1}\sum_{t=1}^{T-1} \log p_\Theta(w_{t+1} \mid w_{1:t})$$

- Maximize the probability the model assigned to the *actual* next word.
- `loss = CrossEntropyLoss(logits, targets)`

Evaluation: Perplexity & training loop

Perplexity

$$\mathrm{PPL} = \exp(\mathcal{L}(\Theta))$$

- **Example:** If $\mathrm{PPL} = 10$, the model is as confused as if it were guessing uniformly among 10 equally likely next words.
- An **random** (untrained) model on vocabulary size V will have $\mathrm{PPL} \approx V$.
- Only **comparable** between models with the **exact same tokenizer and vocabulary size**.

Training Step

```
logits = model(inputs)    (Forward pass)
loss = CrossEntropyLoss(logits, targets)    (Compute NLL)
loss.backward()    (Get gradients ∇_Θ L)
optimizer.step()    (Update Θ ← Θ − η∇_Θ L)
```

Wrap-up

Takeaways

- **Language modelling:** Autoregressive generation $\Rightarrow$ next-token prediction.
- **Tokenization:** BPE discretizes text to balance vocabulary size V and sequence length T.
- **Transformer decoder:** Causal self-attention replaces recurrence, allowing parallel sequence processing over T.
- **Training & metrics:** Optimize average cross-entropy loss (NLL); evaluate using perplexity ($\mathrm{PPL} = \exp(\mathcal{L})$).

Next week: Architecture design choices

- Position encodings (RoPE)
- Advanced attention variants (MQA/GQA)
- Normalization strategies (Pre vs. Post-norm)