# TINY NOTES ON (NOT SO TINY) LANGUAGE MODELS

Leonardo Cotta[1]

[1]Ellison Institute of Technology

*These notes were written to accompany a two-day mini course on language models taught at the Ellison Institute of Technology Centre for Doctoral Training in the Fundamentals of AI [a]. These notes are not revised, and most likely contain typos or even errors —please, open an issue on GitHub if you find any.*

[a] https://www.eitcdt.ox.ac.uk/

## 1 PREAMBLE

You might have heard: "language models are just next-word (token) predictors". In this mini-course, we'll see what a language model is and how it works. By the end, you'll be able to build your own. Where does the magic come from? Let's dive in.

## 2 MODELING LANGUAGE

Let $x$ be a string and $p(x)$ the probability of observing it. Our vocabulary $V$ is a finite set of tokens (e.g., ASCII characters), and $X$ is a discrete random variable over $V^*$. Our **goal** is then to approximate $p(x)$ given training samples $x^1, \dots, x^N$.

**Autoregressive models.** Recall that by the (Bayes) chain rule:

$$p(x) = \prod_{i=1}^{|x|} p(x_i | x_{<i}). \tag{1}$$

We can then achieve our goal by learning a parametric model $p_\theta(x) = \prod_{i=1}^{|x|} p_\theta(x_i|x_{<i})$ where we minimize

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \log p_\theta(x^n). \tag{2}$$

The beauty of autoregressive models lies not only in simplifying the joint distribution, but also in their ability to handle both unconditional generation (sample from $p_\theta(x)$) and conditional generation (sample $y$ from $p_\theta(y|x)$). This model flexibility, coupled with language's flexbile data representation, makes them a perfect model to both i) scale and ii) solve general, diverse problems.

**Shannon's Model.** In 1948, Claude Shannon introduced entropy to quantify uncertainty in probability distributions. To measure language entropy, he proposed the simplest autoregressive model: count token frequencies after each observed prefix in the corpus, normalize to get empirical $\hat{p}(y|x)$, then compute $H(\hat{p}) = -\mathbb{E}_{x\sim\hat{p}}[\log \hat{p}(x)]$ as an approximation of $H(p)$.

Shannon's model is a perfect memorization procedure, *i.e.*, a lookup table of the training distribution. This is both its strength (zero training loss) and weakness (zero generalization). It compresses the *completions* (by counting) but not the *contexts*. Every unique prefix requires a separate table entry, making the model size grow with corpus diversity.

**Neural Models.** Neural networks replace memorization with *inductive learning*: discovering patterns that compress both contexts and completions into parameters. The learning objective, under infinite data:

$$\theta^* = \arg\min_\theta \mathbb{E}_{x\sim p}[-\log p_\theta(x)] = \mathrm{KL}(p\|p_\theta) + H(p).$$

Since $H(p)$ is constant, we minimize $\mathrm{KL}(p\|p_\theta)$ —the divergence between the true and model distributions. Shannon's model achieves zero KL divergence on training data by construction, but neural networks achieve *better* test performance through (context) compression.

**Why compression enables generalization?.** Consider the rule "if context ends in token $a$, predict token $b$". Instead of storing all contexts containing $a$ (Shannon's approach), we store the rule itself —fewer parameters, more coverage. This is compression: representing the data with fewer degrees of freedom by exploiting regularities (symmetries, patterns, structure). When we compress successfully, we can extrapolate to unseen contexts that follow the same patterns.

**The inductive leap..** With finite data (and finite parameters), the model *must infer* the general underlying regularities in the distribution from observed examples. This is induction —generalizing from particular instances to general principles. Understanding this is crucial to grasping LLMs: they are not just interpolating training data, they are learning compressive representations that transfer to new contexts. The more effectively they compress during pretraining, the better they generalize during inference.

**Remark 1** (Pretraining is that simple)**.** *Pretraining compresses data (e.g., the Internet) into parameters. Chinchilla scaling suggests ∼1 parameter per 20 tokens. This creates a "smooth lookup table" that post-training refines for instructions, reasoning, etc.*

# 3 TOKENIZATION

**Motivation.** Efficiently represent text as sequences of tokens.

**Trade-off.** Characters give small, effective vocabulary (128 for ASCII) but long sequences; words give shorter sequences but sparse vocabulary ($|V| \approx 10^5$) with many unseen, or rarely seen tokens.

**Solution.** Byte Pair Encoding (BPE) - iteratively merge frequent character pairs. This guarantees that we achieve a desired vocabulary size, while remaining efficient, *i.e.*, tokens are commonly seen in the corpus. Note that, more generally, we can start with any base vocabulary, and we do not even need to tokenize text at all. It can learn to tokenize any sequence of symbols. We can see BPE as an improvement over an initial tokenization scheme, by compressing it even more.

Code Listing 1: Byte Pair Encoding

```
def bpe(text, num_merges): #num_merges (vocab_size)
    """Byte Pair Encoding tokenizer"""
    vocab = set(text)  # Start with characters
    tokens = list(text) # Converts text to list of characters

    for _ in range(num_merges):
        # Count adjacent pairs
        pairs = {}
        for i in range(len(tokens)-1):
            pair = (tokens[i], tokens[i+1])
            pairs[pair] = pairs.get(pair, 0) + 1

        if not pairs: break

        # Merge most frequent pair
        best_pair = max(pairs, key=pairs.get)
        new_tokens = []
        i = 0
        while i < len(tokens):
            if i < len(tokens)-1 and (tokens[i], tokens[i+1]) == best_pair:
                new_tokens.append(tokens[i] + tokens[i+1])
                i += 2
            else:
                new_tokens.append(tokens[i])
                i += 1
        tokens = new_tokens
        vocab.add(best_pair[0] + best_pair[1])

    return tokens, vocab
```

# 4 TRANSFORMERS

Process sequences up to length $\ell_{\max}$ with hidden dimension $d$.

**Token Embeddings.** Matrix $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ where row $v$ is token $v$'s embedding.

**Positional Encoding.** Captures token order (attention is permutation-equivariant). Sinusoidal encoding:

$$\mathrm{PE}[i, 2j] = \sin(i/10000^{2j/d}), \quad \mathrm{PE}[i, 2j+1] = \cos(i/10000^{2j/d}) \tag{3}$$

for position $i$ and dimension $j = 0, \ldots, d/2 - 1$.

**Attention.** Token gets query $q \in \mathbb{R}^d$; context tokens get keys $k_t$, values $v_t \in \mathbb{R}^d$.

$$\alpha_t = \frac{\exp(\langle q, k_t \rangle / \sqrt{d})}{\sum_{t'} \exp(\langle q, k_{t'} \rangle / \sqrt{d})}, \quad \mathrm{Attn}(q, K, V) = \sum_t \alpha_t v_t \tag{4}$$

**Multi-Head Attention.** Run $h$ attention heads in parallel (each with own $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$), concatenate outputs.

**Forward Pass.**

Code Listing 2: Transformer Forward Pass

```
def transformer_forward(x, E, L, h, W_Q, W_K, W_V, W_O, W_ff1, W_ff2):
    """
    x: token indices [n]
    E: embedding matrix [V, d]
    L: num layers, h: num heads
    W_Q, W_K, W_V, W_O: attention weights per layer [d, d]
    W_ff1: [d, d_ff], W_ff2: [d_ff, d]
    """
    n, d = len(x), E.shape[1]
    d_k = d // h

    # Embed tokens
    X = E[x, :]  # [n, d]

    # Add positional encoding
    PE = zeros(n, d)
    for i in range(n):
        for j in range(d//2):
            PE[i, 2*j] = sin(i / 10000**(2*j/d))
            PE[i, 2*j+1] = cos(i / 10000**(2*j/d))
    X = X + PE

    # Transformer layers
    for layer in range(L):
        # Multi-head attention
        Q = X @ W_Q[layer]  # [n, d]
        K = X @ W_K[layer]
        V = X @ W_V[layer]

        # Reshape to [n, h, d_k]
        Q = Q.reshape(n, h, d_k)
        K = K.reshape(n, h, d_k)
        V = V.reshape(n, h, d_k)

        # Attention per head
        heads = []
        for i in range(h):
            scores = (Q[:, i, :] @ K[:, i, :].T) / sqrt(d_k)  # [n, n]
            attn = softmax(scores, axis=1)
            head_out = attn @ V[:, i, :]  # [n, d_k]
            heads.append(head_out)

        H = concat(heads, axis=1)  # [n, d]
        X_attn = H @ W_O[layer]

        # Residual + LayerNorm
        X = LayerNorm(X + X_attn)

        # Feed-forward
        X_ff = relu(X @ W_ff1[layer]) @ W_ff2[layer]

        # Residual + LayerNorm
        X = LayerNorm(X + X_ff)

    # Output projection (tie weights to embeddings)
    logits = X @ E.T  # [n, V]
    return logits
```

# 5 GENERATION

Given a trained model $p_\theta(x)$, we generate text autoregressively: sample each token conditioned on the prefix, append it, and repeat.

**Sampling Strategies. Greedy decoding**: $x_t = \arg\max_v p_\theta(v|x_{<t})$. Deterministic but repetitive.

**Temperature sampling**: Scale logits by $T > 0$ before softmax. $T < 1$ sharpens distribution (more confident), $T > 1$ flattens it (more random).

**Top-k sampling**: Sample from the $k$ most probable tokens. Prevents low-probability tokens.

**Top-p (nucleus) sampling**: Sample from smallest set of tokens whose cumulative probability exceeds $p$. Adapts to the distribution shape.

**Naive Generation.**

Code Listing 3: Naive Autoregressive Generation

```
def generate_naive(prompt_tokens, max_len, model):
    """
    prompt_tokens: initial token sequence [n]
    max_len: maximum generation length
    model: transformer model
    """
    tokens = prompt_tokens.copy()
    for _ in range(max_len):
        # Run full forward pass on ALL tokens
        logits = model.forward(tokens)  # [len(tokens), V]

        # Take logits for last position
        next_logits = logits[-1, :]  # [V]

        # Sample next token
        next_token = sample(next_logits)  # e.g., top-p sampling

        # Append to sequence
        tokens.append(next_token)

        if next_token == EOS_TOKEN:
            break

    return tokens
```

**Issue**: At step $t$, we recompute attention for all positions $1, \ldots, t-1$, even though their key/value vectors were already computed at step $t-1$. For a sequence of length $n$, this costs $O(n^2)$ operations total.

**KV Caching.** Key insight: In autoregressive generation, position $i < t$ only attends to positions $\leq i$. Its key and value vectors never change as we generate new tokens. We can cache them!

Code Listing 4: Generation with KV Caching

```
def generate_with_kv_cache(prompt_tokens, max_len, model):
    """
    prompt_tokens: initial token sequence [n]
    max_len: maximum generation length
    model: transformer model with cache support
    """
    tokens = prompt_tokens.copy()
    # Initial forward pass - compute and cache all KV pairs
    logits, kv_cache = model.forward_with_cache(tokens)  # [n, V], cache
    next_token = sample(logits[-1, :])
    tokens.append(next_token)
```

```
    for _ in range(max_len - 1):
        # Forward pass only on NEW token, reuse cached KV pairs
        logits, kv_cache = model.forward_with_cache(
            [next_token],      # only new token [1]
            kv_cache=kv_cache  # reuse previous keys/values
        )

        next_token = sample(logits[-1, :])
        tokens.append(next_token)

        if next_token == EOS_TOKEN:
            break

    return tokens
```

**Implementation**: For each layer and head, maintain cached $\mathbf{K}, \mathbf{V}$ matrices. When processing token $t$:

1. Compute new query $q_t$, key $k_t$, value $v_t$
2. Append $k_t$ to cached keys: $\mathbf{K} \leftarrow [\mathbf{K}; k_t]$
3. Append $v_t$ to cached values: $\mathbf{V} \leftarrow [\mathbf{V}; v_t]$
4. Compute attention: $\text{Attn}(q_t, \mathbf{K}, \mathbf{V})$ using full cached context

**Complexity**: Reduces per-step cost from $O(t^2)$ to $O(t)$. Total generation cost drops from $O(n^2)$ to $O(n)$.
**Memory trade-off**: Must store $\mathbf{K}, \mathbf{V}$ for each layer and head. For model with $L$ layers, $h$ heads, dimension $d$, and sequence length $n$: memory is $O(L \cdot h \cdot n \cdot d)$. This is why long contexts are expensive!

# 6 POST-TRAINING

Here we will take a look at the first strategies used in post-training[1]. The first ideas behind post-training are usually referred to as alignment. They are motivated by the fact that the model's predictions are just regurgitating compressed, interpolated, Internet text, as we have seen in the previous section. This is not very useful to human applications, where we want it to answer questions, follow instructions, and engage in meaningful conversations. Therefore, the word "alignment" usually refers to aligning the model's predictions with human preferences or values. How do we do that? Let's take a look at a very simple and yet effective strategy: Reinforcement Learning from Human Feedback (RLHF).

**Data.** Collect human feedback in the form of ratings, annotations, or other forms of human input, such as pairwise preferences. This feedback can be collected through surveys, crowdsourcing platforms, or other methods.

**Reward Models.** Train a separate model to predict a reward score for generated sequences, given certain prompts. The reward model is trained on the human feedback data, such as human ratings or annotations. This reward model is simply assinging a real-valued score, not necessarily a probability distribution, for to a model answer, given a prompt. Our goal is then to use this reward model to guide, *i.e.*, align, the models' answers towards higher rewards.

**RLHF is just Bayesian Inference.** The intuition is simple: we want to update our pretrained model $p_\theta$ to account for human preferences encoded in a reward function $r$. This is a Bayesian inference problem —we have a prior distribution (the pretrained model) and evidence (the reward function), and we want to compute the posterior.

[1] In the distant past of 2022

A reward function $r(x)$ assigns scores to sequences, but to do Bayesian inference, we need a distribution. We convert $r$ into a distribution by exponentiating and normalizing:

$$\pi^*_{\text{KL-RL}}(x) = \frac{1}{Z}\pi_0(x)\exp(r(x)/\beta), \tag{5}$$

where $\pi_0 := p_\theta$is the prior (pretrained model), $\exp(r(x)/\beta)$ is the likelihood (evidence from the reward, scaled by temperature $\beta$), and $Z$ is the partition function ensuring normalization. This $\pi^*_{\text{KL-RL}}$ is the target distribution that balances the prior and the reward.

The KL-regularized RL objective

$$J_{\text{KL-RL}}(\theta) = \mathbb{E}_{x\sim\pi_\theta}[r(x)] - \beta \cdot \text{KL}(\pi_\theta \| \pi_0), \tag{6}$$

can be rewritten as minimizing the KL divergence to the target:

$$J_{\text{KL-RL}}(\theta) \propto -\text{KL}(\pi_\theta \| \pi^*_{\text{KL-RL}}). \tag{7}$$

This is *variational inference*: approximating the intractable posterior $\pi^*_{\text{KL-RL}}$ with our parametric model $\pi_\theta$. Equivalently, $J_{\text{KL-RL}}(\theta)$ is the evidence lower bound (ELBO) on the log-likelihood that $\pi_\theta$ is optimal under $r$ given prior $\pi_0$.

**Why This View Matters.** This Bayesian perspective explains the KL penalty term: it's not an ad-hoc regularizer, but emerges naturally from casting alignment as posterior inference. The prior $\pi_0$ keeps $\pi_\theta$ fluent and diverse, while the reward likelihood guides it toward preferred behaviors. The temperature $\beta$ controls how much we trust the reward versus the prior —high $\beta$ means we stay close to $\pi_0$, low $\beta$ means we follow the reward more aggressively.

# REFERENCES