TINY NOTES ON (NOT SO TINY) LANGUAGE MODELS

Leonardo Cotta

Ellison Institute of Technology

Notes for a two-day mini course on language models
Ellison Institute of Technology Centre for Doctoral Training
in the Fundamentals of AI

Preamble

"Language models are just next-word (token) predictors"

- What is a language model?
- How does it work?
- By the end: you'll be able to build your own
- Where does the magic come from?

Modeling Language

Let x be a string and $p(x)$ the probability of observing it.

- Vocabulary V: finite set of tokens (e.g., ASCII characters)
- X: discrete random variable over V^*
- **Goal**: approximate $p(x)$ given training i.i.d. samples $x^1, \ldots, x^N$

Issue with this?

Set of sequences of tokens

Subscript = token

Superscript = sequence of tokens

Autoregressive Models

Chain Rule

By the (Bayes) chain rule:

Makes $p(x)$ tractable

Vocabulary size

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

If i is 1, then this is empty

$\rightarrow p(x_1) = 1$

$\uparrow$ beginning of string

Learning Objective

Learn parametric model $p_\theta(x) = \prod_{i=1}^{|x|} p_\theta(x_i | x_{<i})$ by minimizing:

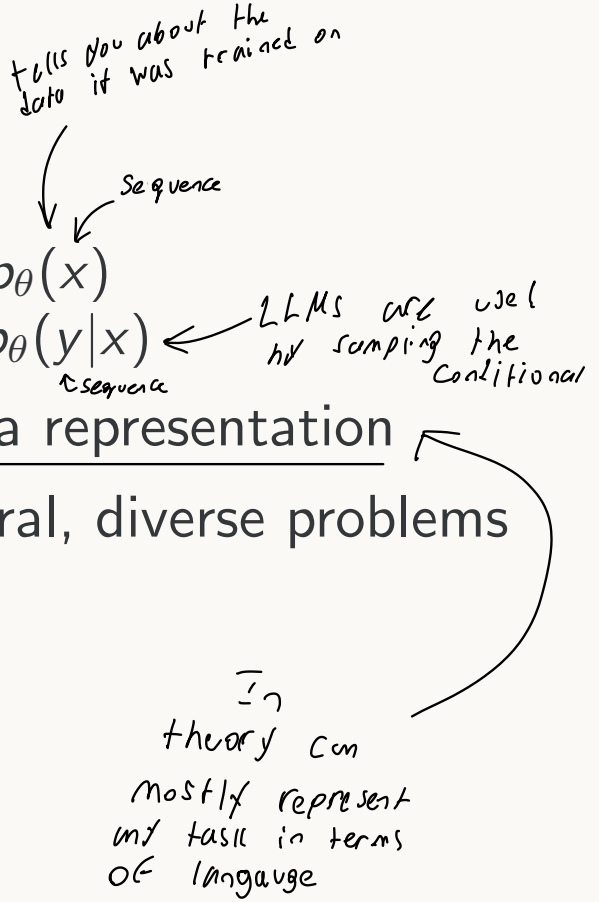
Back prop $\uparrow$

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

Why Autoregressive?

The beauty lies in:

- Simplifying the joint distribution
- Handling both:
  - Unconditional generation: sample from $p_\theta(x)$
  - Conditional generation: sample y from $p_\theta(y|x)$
- Model flexibility + language's flexible data representation
- Perfect for: i) scaling and ii) solving general, diverse problems



Shannon's Model (1948)

Claude Shannon's simple autoregressive model:

- Count token frequencies after each observed prefix
- Normalize to get empirical $\hat{p}(y|x)$
- Compute entropy: $H(\hat{p}) = -\mathbb{E}_{x\sim\hat{p}}[\log \hat{p}(x)]$

Strength: Zero training loss (perfect memorization)
Weakness: Zero generalization

Compresses *completions* but not *contexts*

Neural Models

Key Idea

Replace memorization with *inductive learning*

Learning objective (under infinite data):

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

entropy of true distribution

intrinsic uncertainty provides lower bound

Since $H(p)$ is constant, we minimize $\mathrm{KL}(p\|p_\theta)$

Neural networks achieve *better* test performance through context compression

Why Compression Enables Generalization

Consider: "if context ends in token a, predict token b"

Shannon: Store all contexts containing a
Neural: Store the rule itself

Compression: Represent data with fewer degrees of freedom by exploiting regularities (symmetries, patterns, structure)

Successful compression $\Rightarrow$ extrapolation to unseen contexts

The Inductive Leap

- With finite data and parameters, the model *must infer* general underlying regularities from observed examples
- **This is induction**: generalizing from particular instances to general principles

LLMs are not just interpolating training data—they learn compressive representations that transfer to new contexts

Pretraining is That Simple

As you increase compute, 1 parameter per 20 tokens

Remark (Pretraining)

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.

This is optimal for generalization, not compression

General rule for language // compressing by 20x the size of your dataset"

Tokenization

Motivation

Efficiently represent text as sequences of tokens

Trade-off:

- **Characters**: Small vocabulary (128 for ASCII) but long sequences → Requires more memory (Compute efficiency)
- **Words**: Shorter sequences but sparse vocabulary ($|V| \approx 10^5$)
  - Reduce representation ability
  - → Also, some words seen much more than others
  - → Dist. becomes skewed

Solution: Byte Pair Encoding (BPE)

- Iteratively merge frequent character pairs
- Guarantees desired vocabulary size
- Tokens are commonly seen in corpus

Byte Pair Encoding

← BPE really dependent on delimiters (e.g. spaces)

```
def bpe(text, num_merges):
    """Byte Pair Encoding tokenizer"""
    vocab = set(text)
    tokens = list(text)

    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
```

Counts freq. of consecutive tokens

e.g. [he fox ... th

Th=2

He=1

Most freq. pair becomes next token

Merges most freq. pair and updates token list

Adds rest of the tokens to updated list

Transformers: Overview

Process sequences up to length ℓ_{max} with hidden dimension d

Key Components:

- Token embeddings
- Positional encoding
- Multi-head attention
- Feed-forward networks
- Residual connections & layer normalization

Token Embeddings & Positional Encoding

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:

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

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

Attention Mechanism

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

$$\alpha_t = \frac{\exp(\langle q, k_t \rangle / \sqrt{d})}{\sum_{t'} \exp(\langle q, k_{t'} \rangle / \sqrt{d})}$$

$$\text{Attn}(q, K, V) = \sum_t \alpha_t v_t$$

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

Transformer Forward Pass (1/2)

```
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 [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 Forward Pass (2/2)

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

        Q = Q.reshape(n, h, d_k)
        K = K.reshape(n, h, d_k)
        V = V.reshape(n, h, d_k)

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

        H = concat(heads, axis=1)
        X_attn = H @ W_O[layer]
        X = LayerNorm(X + X_attn)

        X_ff = relu(X @ W_ff1[layer]) @ W_ff2[layer]
        X = LayerNorm(X + X_ff)

    logits = X @ E.T  # [n, V]
    return logits
```

Generation

Given trained model $p_\theta(x)$, generate text autoregressively:

- Sample each token conditioned on prefix
- Append it
- Repeat

Sampling Strategies

- **Greedy decoding**: $x_t = \arg\max_v p_\theta(v|x_{<t})$
  - Deterministic but repetitive
- **Temperature sampling**: Scale logits by $T > 0$
  - $T < 1$ sharpens (more confident)
  - $T > 1$ flattens (more random)
- **Top-k sampling**: Sample from k most probable tokens
- **Top-p (nucleus) sampling**: Sample from smallest set whose cumulative probability exceeds p

Naive 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)

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

        # Sample next token
        next_token = sample(next_logits)

        # Append to sequence
        tokens.append(next_token)

        if next_token == EOS_TOKEN:
            break

    return tokens
```

Issue: At step t, recompute attention for all positions $1, \ldots, t-1$.
Cost: $O(n^2)$ 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
- $\Rightarrow$ We can cache them!

Generation with KV Caching (1/2)

```
def generate_with_kv_cache(prompt_tokens,
                           max_len, model):
    """
    prompt_tokens: initial token sequence [n]
    max_len: maximum generation length
    model: transformer with cache support
    """
    tokens = prompt_tokens.copy()

    # Initial forward pass - cache all KV pairs
    logits, kv_cache = \
        model.forward_with_cache(tokens)

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

Generation with KV Caching (2/2)

```
    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],
                kv_cache=kv_cache
            )

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

        if next_token == EOS_TOKEN:
            break

    return tokens
```

KV Caching: 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

KV Caching: Complexity & Trade-offs

Complexity

Complexity: Reduces per-step cost from $O(t^2)$ to $O(t)$

- Total generation cost: $O(n^2) \rightarrow O(n)$

Memory Trade-off

Must store $\mathbf{K}, \mathbf{V}$ for each layer and head
For model with L layers, h heads, dimension d, sequence length n:

- Memory: $O(L \cdot h \cdot n \cdot d)$
- This is why long contexts are expensive!

Post-Training

A look at the first strategies used in post-training

(In the distant past of 2022)

- First ideas: **alignment**
- Pretrained models just regurgitate compressed Internet text
- Goal: align predictions with human preferences/values
- Make models useful: answer questions, follow instructions, engage meaningfully

Reinforcement Learning from Human Feedback (RLHF)

Data

Collect human feedback: ratings, annotations, pairwise preferences

Reward Models

Train a model to predict reward scores for generated sequences given prompts

- Trained on human feedback data
- Assigns real-valued scores (not probabilities)
- Goal: guide model answers toward higher rewards

RLHF is Just Bayesian Inference

The Intuition

Update pretrained model p_θ to account for human preferences encoded in reward function r

This is Bayesian inference:

- Prior distribution: pretrained model
- Evidence: reward function
- Compute: posterior distribution

From Rewards to Distributions

Convert reward function $r(x)$ into distribution:

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

Where:

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

The KL-Regularized RL Objective

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

Can be rewritten as minimizing KL divergence to target:

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

This is *variational inference*: approximating intractable posterior $\pi^*_{\text{KL-RL}}$ with parametric model π_θ

Why This View Matters

- KL penalty isn't ad-hoc regularization
- It emerges naturally from casting alignment as posterior inference
- Prior π_0 keeps π_θ fluent and diverse
- Reward likelihood guides toward preferred behaviors
- Temperature β controls trust balance:
  - High β: stay close to π_0
  - Low β: follow reward more aggressively

Tell Us What You Think

https://forms.office.com/e/J9y561KuDV