Lab 2 – GPT from scratch

IASD / MASH – LLMs course

Florian Le Bronnec

November 12, 2024

Table of Contents

Table of Contents

Last time

Goal: Implement a version of the GPT model from scratch.

1. We built the **attention module**.
2. We built the **transformer block**.
   - Attention
   - Feedforward
   - LayerNorm + skip connection
3. We built the **transformer model** by stacking transformer blocks.
4. We added a linear layer to the output of the transformer model for the **vocabulary projection**.

Layer norm

- **Error in the code**: layer norm shouldn't use detach gradients!

Layer norm

- **Error in the code**: layer norm shouldn't use detach gradients!
- Where should we put the layer norm? In recent LLaMA models, **pre-layer normalization** is applied.

$$\mathbf{y} = \mathrm{SelfAttention}(\mathrm{LayerNorm}(\mathbf{x})) + \mathbf{x}$$
$$\mathbf{z} = \mathrm{FeedForward}(\mathrm{LayerNorm}(\mathbf{y})) + \mathbf{y}$$

  In contrast, the original Transformer model uses **post-layer normalization**:

$$\mathbf{y} = \mathrm{LayerNorm}(\mathrm{SelfAttention}(\mathbf{x}) + \mathbf{x})$$

- **Rule of thumb**: Apply layer normalization before each sublayer (like attention or feed-forward) in deep or autoregressive models, such as GPT-style transformers.

Attention's scaling

- **Add scaling**: $\dfrac{1}{\sqrt{d}}$ (forgotten in the previous lab).

Masks in multi-head self-attention

Goal: Ensure that the model only attends to relevant tokens by:

- Preventing access to future tokens (important for generation tasks)
- Ignoring padding tokens that don't contain information

Causal masking (masking future tokens)

- **Why?** In text generation, each position should only access past tokens to make sure predictions are causal.
- **How?** We use an upper triangular matrix to mask future tokens:

$$\text{mask}_{\text{causal}} = \mathrm{triu}(\mathbf{1}_{\text{shape}(\mathbf{s}_{ij})}, 1)$$

- **Setting scores to $-\infty$:** Scores for future tokens are set to $-\infty$. Then $a_i j$ will be zero for these tokens.

Masks in multi-head self-attention

Padding masking (ignoring padding tokens)

- **Why?** Padding tokens are added to make all inputs the same length, but they don't contain any information. The model should ignore them.
- **How?** The padding mask is calculated using `attention_mask`.
- **Setting scores to $-\infty$:** Similar to causal masking, padding tokens are set to $-\infty$ in $\mathbf{s}_{ij}$.

Combining causal and padding masks

- **Why combine both?** Combining both masks ensures that each token attends only to valid, relevant tokens, ignoring both future and padding tokens.
- **Final adjustment:** We add both masks to $\mathbf{s}_{ij}$ and scale them by $-\infty$:

$$\mathbf{s}_{ij} = \mathbf{s}_{ij} + (\text{mask}_{\text{causal}} + \text{mask}_{\text{padding}}) \times (-\infty)$$

$$\mathbf{a}_{ij} = \text{softmax}(\mathbf{s}_{ij}, \text{dim} = -1)$$

Making equal size tensors (padding)

- **Error in the code:** Typo when using a wrong variable name (use `label_ids`)

Table of Contents

Step 1: Input Representation as Word IDs

- The input sequence is represented by a list of unique IDs for each token, starting with a special BOS (beginning-of-sequence) token.

Token	Word ID
`<BOS>`	0
`Paris`	102
`is`	76
`the`	34
`capital`	159
`of`	45
`France`	203

Step 2: Mapping Word IDs to Embedding Vectors

- Each token ID is converted into a vector in the embedding space.
- Example embeddings are shown as 3-dimensional vectors.

Token	Word ID	Embedding Vector
`<BOS>`	0	`[0.1, 0.0, 0.3]`
`Paris`	102	`[0.2, 0.1, 0.4]`
`is`	76	`[0.3, 0.6, 0.5]`
`the`	34	`[0.5, 0.2, 0.7]`
`capital`	159	`[0.6, 0.4, 0.3]`
`of`	45	`[0.1, 0.8, 0.2]`
`France`	203	`[0.4, 0.9, 0.1]`

Step 3: Causal Processing with Self-Attention

- Causal processing is achieved through masked self-attention.
- Each token can only attend to itself and previous tokens.

Self-Attention with Mask

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right) V$$

where Q, K, and V are the query, key, and value matrices, d_k is the dimension of keys, and M is the causal mask.

- The causal mask M is set to $-\infty$ for future tokens, blocking attention to future positions.

Step 4: Logits and Prediction for Next Token

- After processing, each token produces a logits vector representing the probability distribution for the next word.
- Example logits for each position (showing top 3 predictions):

Token	Top Logits for Next Token
`<BOS>`	`{Paris: 0.7, London: 0.2, Berlin: 0.1}`
`Paris`	`{is: 0.6, was: 0.2, and: 0.1}`
`is`	`{the: 0.5, a: 0.3, an: 0.1}`
`the`	`{capital: 0.4, city: 0.3, center: 0.2}`
`capital`	`{of: 0.7, in: 0.2, for: 0.1}`
`of`	`{France: 0.6, Europe: 0.3, Paris: 0.1}`
`France`	`{<END>: 0.9, is: 0.05, the: 0.02}`

Step 5: Teacher Forcing and Loss Calculation

- In teacher forcing, we calculate the loss as the negative log probability of the true next token.

Token	Should predict	Logits (Top Scores)	Loss
`<BOS>`	`Paris`	`{Paris: 0.7, London: 0.2}`	$-\log(0.7)$
`Paris`	`is`	`{is: 0.6, was: 0.2}`	$-\log(0.6)$
`is`	`the`	`{the: 0.5, a: 0.3}`	$-\log(0.5)$
`the`	`capital`	`{capital: 0.4, city: 0.3}`	$-\log(0.4)$
`capital`	`of`	`{of: 0.7, in: 0.2}`	$-\log(0.7)$
`of`	`France`	`{France: 0.6, Europe: 0.3}`	$-\log(0.6)$
`France`	`<END>`	`{<END>: 0.9, is: 0.05}`	$-\log(0.9)$

Summary: From Input to Teacher Forcing in LLMs

- Convert tokens to IDs with BOS.
- Map IDs to embeddings.
- Process embeddings causally through masked self-attention.
- Generate logits for each position.
- Apply teacher forcing, compute log probability, and backpropagate.

Self-Attention Recap

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

where M is the causal mask to restrict attention to preceding tokens only.

Table of Contents

Introduction to Inference

- Inference is the process by which a model generates tokens to extend an input sequence.
- This is done iteratively, selecting one token at a time and appending it to the sequence.
- Two common methods are:
  - Greedy Decoding
  - Ancestral Sampling

Inference in classification

At train time

Classification models are trained with the MLE objective, i.e., they maximize $\mathrm{P}_\theta(y = \text{class} \mid x)$.

At inference time

For an input x, the model gives a probability distribution over the classes $\mathrm{P}_\theta(\cdot \mid x)$.
Then you fix a decision rule, usually:

$$\hat{y} = \arg\max_y \mathrm{P}_\theta(y \mid x). \tag{1}$$

$\Longrightarrow$ For classification models, i.e., **encoders** in NLP, we do the same.

Inference in generation

At train time

For generative models, we maximize instead the factorized density: $\prod_{i=1}^{L} \mathrm{P}_\theta(y_i = w_i \mid w_{<i})$.

At inference time

Is the following decision rule still a good choice?

$$\hat{y} = \arg\max_{y} \mathrm{P}_\theta(y \mid x) = \arg\max_{y_1,\ldots,y_L} \prod_{i=1}^{L} \mathrm{P}_\theta(y_i \mid y_{<i}).$$

No! We are taking the arg max over V^L combinations, highly **intractable**.

Inference in generation

Workaround: approximate $\arg\max_{y_1,\ldots,y_L} \prod_{i=1}^{L} \mathrm{P}_\theta(y_i \mid y_{<i})$ with a **greedy algorithm**.

Simply:

- $\hat{y}_1 = \arg\max_{y_1} \mathrm{P}_\theta(y_1)$,
- $\hat{y}_2 = \arg\max_{y_2} \mathrm{P}_\theta(y_2 \mid \hat{y}_1)$,
- $\vdots$
- $\hat{y}_i = \arg\max_{y_i} \mathrm{P}_\theta(y_i \mid \hat{y}_{<i})$.

$\Longrightarrow$ At each step, the $\arg\max$ is only performed over V possibilities.

Greedy Decoding

Definition

Greedy decoding selects the token with the highest probability at each step, creating a deterministic sequence.

- **Step 1:** Start with an input sequence, e.g., `"Paris is the"`.
- **Step 2:** The model predicts probabilities for the next token:

$$\{\texttt{capital} : 0.6, \texttt{city} : 0.3, \texttt{center} : 0.1\}$$

- **Step 3:** Select the token with the highest probability (`"capital"`).
- **Step 4:** Append the selected token to the sequence:

`"Paris is the capital"`

- **Step 5:** Repeat Steps 2-4 until reaching the end token (`<END>`) or desired length.

Greedy Decoding - Iterative Growth Example

- Initial sequence: `"Paris is the"`
- **Iteration 1:** Predict `"capital"`

  `"Paris is the capital"`

- **Iteration 2:** Predict `"of"`

  `"Paris is the capital of"`

- **Iteration 3:** Predict `"France"`

  `"Paris is the capital of France"`

- Sequence reaches an end token (`<END>`), completing generation.

Ancestral Sampling

Definition

Ancestral sampling selects tokens based on their probabilities, allowing randomness and creating diverse possible outputs.

- **Step 1:** Start with an input sequence, e.g., `"Paris is the"`.
- **Step 2:** The model predicts probabilities for the next token:

$$\{\texttt{capital} : 0.6, \texttt{city} : 0.3, \texttt{center} : 0.1\}$$

- **Step 3:** Sample a token based on these probabilities, e.g., `"city"` (with 30% chance).
- **Step 4:** Append the sampled token to the sequence:

`"Paris is the city"`

- **Step 5:** Repeat Steps 2-4 until reaching <END> or desired length.

Ancestral Sampling - Iterative Growth Example

- Initial sequence: `"Paris is the"`
- **Iteration 1:** Sample `"city"`

  `"Paris is the city"`

- **Iteration 2:** Sample `"of"`

  `"Paris is the city of"`

- **Iteration 3:** Sample `"dreams"`

  `"Paris is the city of dreams"`

- Ancestral sampling introduces variety, so the final output could vary on each run.

Comparison of Greedy Decoding and Ancestral Sampling

- **Greedy Decoding:**
  - Deterministic, choosing only the highest-probability token at each step.
  - Suitable for generating the most probable and consistent sequence.
- **Ancestral Sampling:**
  - Stochastic, selecting tokens based on probabilities, allowing randomness.
  - Produces varied and potentially more creative sequences.

Summary

- Inference in LLMs can be done using methods like greedy decoding and ancestral sampling.
- **Greedy decoding** is deterministic and produces an approximated most likely sequence.
- **Beam search** is a generalization of greedy decoding that considers multiple likely sequences.
- **Ancestral sampling** is stochastic.