Notes on *Build a Large Language Model (From Scratch)*

by Jacob Williams • last updated 2025-03-18

1. “Understanding large language models”

Raschka mentions that BERT fills in masked words while GPT predicts the next token, and that the former “equips BERT with strengths in text classification tasks, including sentiment prediction and document categorization”.

Raschka indicates GPT is simpler than “the original transformer architecture” because it lacks an encoder.

2. “Working with text data”

> A detailed discussion and implementation of BPE is out of the scope of this book, but in short, it builds its vocabulary by iteratively merging frequent characters into subwords and frequent subwords into words. For example, BPE starts with adding all individual single characters to its vocabulary (“a,” “b,” etc.). In the next stage, it merges character combinations that frequently occur together into subwords. For example, “d” and “e” may be merged into the subword “de,” which is common in many English words like “define,” “depend,” “made,” and “hidden.” The merges are determined by a frequency cutoff. (Raschka 34–35)

It's not clear to me why the target value contains the context minus the first word, as opposed to only containing the next word.

> For those who are familiar with one-hot encoding, the embedding layer approach described here is essentially just a more efficient way of implementing one-hot encoding followed by matrix multiplication in a fully connected layer, which is illustrated in the supplementary code on GitHub at https://mng.bz/ZEB5. Because the embedding layer is just a more efficient implementation equivalent to the one-hot encoding and matrix-multiplication approach, it can be seen as a neural network layer that can be optimized via backpropagation. (Raschka 43)

The book mentions both *absolute* and *relative* positional embeddings; the latter makes it so "the model can generalize better to sequences of varying lengths, even if it hasn't seen such lengths during training". (Raschka 45) But "OpenAI's GPT models use absolute positional embeddings that are optimized during the training process…" (Raschka 45)

3. "Coding attention mechanisms"

> In self-attention, the "self" refers to the mechanism's ability to compute attention weights by relating different positions within a single input sequence. It assesses and learns the relationships and dependencies between various parts of the input itself, such as words in a sentence or pixels in an image.
>
> This is in contrast to traditional attention mechanisms, where the focus is on the relationships between elements of two different sequences, such as in sequence-to-sequence models where the attention might be between an input sequence and an output sequence… (Raschka 56)

> The goal of self-attention is to compute a context vector for each input element that combines information from all other input elements. (Raschka 56)

The walkthrough was helpful in reminding me why there are separate 'key', 'query', and 'value' weights matrices.

I'm not sure I really understand the need for causal/masked attention—why isn't the fact that the target/y-value is the next word enough?

4. "Implementing a GPT model from scratch to generate text"

Explanation of the focus on GPT-2, with some interesting stats:

> Note that we are focusing on GPT-2 because OpenAI has made the weights of the pretrained model publicly available, which we will load into our implementation in chapter 6. GPT-3 is fundamentally the same in terms of model architecture, except that it is scaled up from 1.5 billion parameters in GPT-2 to 175 billion parameters in GPT-3, and it is trained on more data. As of this writing, the weights for GPT-3 are not publicly available. GPT-2 is also a better choice for learning how to implement LLMs, as it can be run on a single laptop computer, whereas GPT-3 requires a GPU cluster for training and inference. According to Lambda Labs (https://lambdalabs.com/), it would take 355 years to train GPT-3 on a single V100 datacenter GPU and 665 years on a consumer RTX 8000 GPU. (Raschka 94)

The chapter discusses layer normalization, to help with “problems like vanishing or exploding gradients”:

> The main idea behind layer normalization is to adjust the activations (outputs) of a neural network layer to have a mean of 0 and a variance of 1, also known as unit variance. (Raschka 99)

Normalization is done by subtracting the mean and dividing by standard deviation, then scaling and shifting by trainable parameters.

Section 4.3 discusses the Gaussian error linear unit, GELU, an alternative to ReLUs.

> …the exact version is defined as GELU(x) = , where is the cumulative distribution function of the standard Gaussian distribution. (Raschka 105)

But it’s approximated like this:

> — — (Raschka 105)

GELU is smoother than ReLU.

> Moreover, unlike ReLU, which outputs zero for any negative input, GELU allows for a small, non-zero output for negative values. This characteristic means that during the training process, neurons that receive negative input can still contribute to the learning process, albeit to a lesser extent than positive inputs. (Raschka 106)

The graph of GELU is surprising to me: the minimum output value occurs for an input somewhere between -1 and 0.

The chapter also discusses skip/shortcut connections, another tool for addressing vanishing gradients.

On transformer blocks:

> The preservation of shape throughout the transformer block architecture is not incidental but a crucial aspect of its design. This design enables its effective application across a wide range of sequence-to-sequence tasks, where each output vector directly corresponds to an input vector, maintaining a one-to-one relationship. However, the output is a context vector that encapsulates information from the entire input sequence (see chapter 3). This means that while the physical dimensions of the sequence (length and feature size) remain unchanged as it passes through the transformer block, the content of each output vector is re-encoded to integrate contextual information from across the entire input sequence. (Raschka 116)

On “weight tying”: “the original GPT-2 architecture reuses the weights from the token embedding layer in its output layer.” (Raschka 121) But this isn’t the current practice.

“5. Pretraining on unlabeled data”

On the relation of cross-entropy loss and perplexity:

> Perplexity can be calculated as `perplexity = torch.exp(loss)`, which returns `tensor(48725.8203)` when applied to the previously calculated loss.
>
> Perplexity is often considered more interpretable than the raw loss value because it signifies the effective vocabulary size about which the model is uncertain at each step. In the given example, this would translate to the model being unsure about which among 48,725 tokens in the vocabulary to generate as the next token. (Raschka 140)

The chapter estimates that llama-2-7b would cost “around $690,000” (Raschka 141)—that’s way less than I might have guessed!

Temperature scaling, at least as described here, just consists of dividing the output token probabilities by the temperature. So: “Temperatures greater than 1 result in more uniformly distributed token probabilities, and temperatures smaller than 1 will result in more confident (sharper or more peaky) distributions.” (Raschka 154)

Temperature scaling can be combined with top-k sampling to ensure you don’t pick *too* improbable tokens.

“6. Fine-tuning for classification”

This chapter replaces the model’s output head with one that just has outputs for two classes, and fine-tunes by training on just that output head and the final transformer block and layer norm layer.

“7. Fine-tuning to follow instructions”

Training went haywire when I tried using mps but was fine on cpu. Could be interesting to look into why.

Interesting thing while executing the sample code on page 244 (asking llama-3 to ‘score the model response’): on the third (but only the third) case, llama refers to it as “my own response”. This happened for me and in the book, too!

References

Raschka, Sebastian. *Build a Large Language Model (from scratch)*. Manning, 2025.