Viswa Kumar | Book Notes for AI Engineering by Chip Huyen

Viswa Kumar

2025-02-14

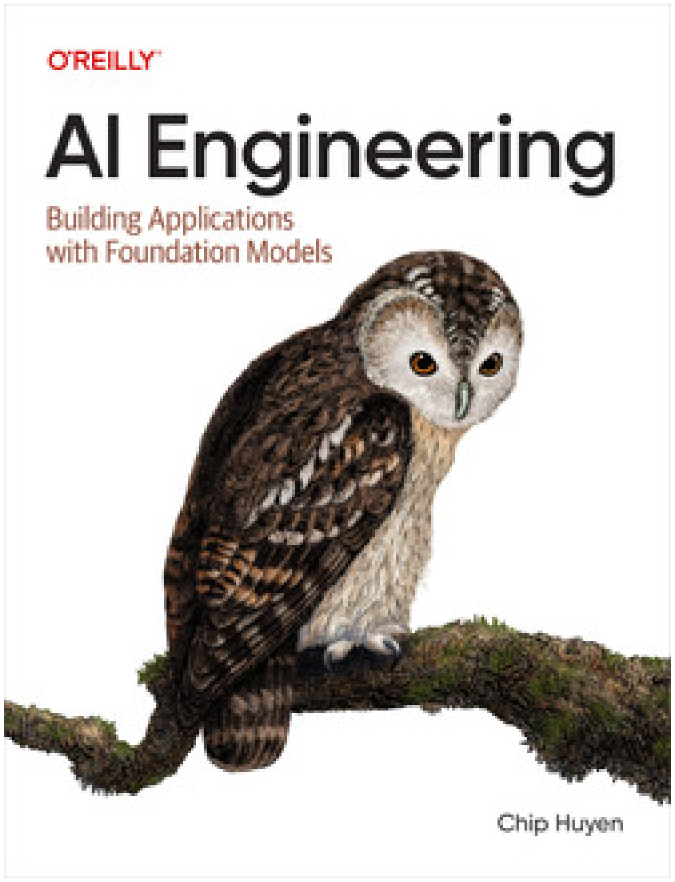
Table of contents

Preface

Welcome to my book notes website for the book **AI Engineering by *Chip Huyen***. In this website, I have documented all of my notes for this book, chapter by chapter as I read through the content.

Taking notes from the book you read, not only firms the understanding, but also serves as a quick summary / reference of the key concepts, should you wanna look into at a later point in time.

Normally I do this for my safe keeping, but now I'm trying to document my knowledge journey for the public benefit as well.



Please don't consider my notes as a *free* replacement to the actual book. I strongly encourage you to buy the actual book to compensate for all the efforts the author(s) & publishing house team put together to deliver the book.
All the actual content of the book are the IPs of the book. I'm merely recollecting, summarizing & articulating what I learnt from the book along with my own thoughts & opinions.

Chapter 1

This chapter introduces the term *AI Engineering* and attempts to define how it is different from *ML Engineering* (which has been there for many years). First the book introduces very concepts like language model, large & frontier models etc and also talks about the opportunity of AI Engineering.

There are 2 types of language models conceptually 1) Masked models - Models are trained to predict the fill in the masked words (aka fill in the blanks). These models are useful for classification, translations etc. In other words *encoder* heavy 2) Autoregressive models - Models are trained to predict the next word in the sequence. These models are useful for text generative tasks such as summarization, instruction following, chatbots etc. In other words, *decoder* heavy.

If the model accepts more than 1 data modality (forms of input), they are *Multi-modal-Model* .

Within the AI Engineering stack, there could be 3 layers (Infrastructure, Model Development & Application Development).

The rush and value proposition is on the applications built on top of foundational models and this book argues that knowing how these models are trained and the science behind those models would help make several key long term decisions in choosing, fine-tuning & tweaking the models for application needs. The moat is clearly on the application layers since only handful of large companies can stomach the investments in doing any groundbreaking research on the model creation itself and majority of the industry will add value to the application layer.

Before jumping directly to GenAI app development, it is crucial to define the success. Some possible metrics could be 1) LLM output quality 2) Time To First Token (TTFT), Time Per Output Token (TPOT), Overall Latency 3) Cost for the APIs 4) Fairness, Consistency, Accuracy etc.

Use model evaluation techniques to deal with the *probablistic* nature of uncertain LLM outputs to gauge whether you are going in right direction, is it worth to invest time and effort on this.

Basically an enterprise will jump into AI only for 3 reasons:

1) Without AI, the company's main business will be replaced. Eg anything that deals heavily with text content - content creation, copyright etc

2) Without AI the company will miss lot of value creation. Eg anything that can be automated, more productivity with less investment
3) Without AI, the company will be left behind while others are bathing in shiny light. Eg anything that is more of a GPT wrapper

On the Model development, there are 3 stages

- Pre-Training - activities including gathering training data, cleaning, tokenziation, embeddings and generating a foundational model to predict the next token effectively
- Post-Training / Fine-Tuning - activities that further train the pre-trained model to produce outputs that are aligned to human preference

For the ML engineering tasks, several training algo exists and for each task specific labelled dataset is used. ML engineering for the most part is SFT with labelled data. AI Engg on the other hand deals with unstructured data and often self supervised fine tuning is employed. Training to predict the next token is often the only training algorithm that is employed in AI Engg. Hence for AI Engg, the knowledge of ML algos are nice to have but not strictly needed. However the skill to optimize the inference for the hardware is much needed in AI Engg. In case of ML engg, feature engineering esp with tabular data is needed whereas in AI Engg, data cleaning, de-duplication, tokenization, context retrieval etc are more needed.

Another important distinction is *mindshift* . Incase of ML engineering, one would go from Data -> Model -> Product. Incase of AI Engg, we go from Product -> Data -> Model. i.e we use existing models and try to productize . If that is not solving the problem, we augment the model with external KB using RAG etc and finally resort to fine tune the model with custom data, thereby creating a specialized model when needed.

Chapter 2

This chapter starts with Training data. For GPT like LLMs, internet is the training data. It is impossible to curate a special training data for LLMs. Hence the approach was to use whatever that was available. Google's C4 dataset (Colossal Clean Common Crawl) was a refined dataset from Common Crawl dataset. The dataset is already skewed, biased and unfairly composed of dominating subjects. English language is unfairly represented more than other worldly languages, so does with technology than other useful domains. So naturally the LLMs are more general than specifically trained for a special purpose domain.

In future, model trainers would have a special arrangement with publishing houses to seek quality data for pre-training needs. Moving forward, original content would be even more valuable.

Many model architectures are available but the one that is popular is Transformer. The journey started with Sequence-to-Sequence, followed by RNNs and then took a turn in Transformer. Now we have space based models like Mixture of Experts (MoEs) (*aka Sparse Models*), hybrid architectures like Mamba, Jamba etc.

Transformer contains Encoder & Decoder modules. Encoder tries to capture different dimensions of input token (embedding) into a vector representations. Decoder tries to probabilistically predict output tokens based on those internal vector representations. What made transformer unique is the attention mechanism.

Attention is basically a technique where, when the decoder is trying to predict the output token, it helps the decoder to help select certain dimensions of the input tokens such that the output token is making sense when in the context with input token. i.e While predicting the output token w.r.t the set of input token, which of the input tokens should be taken into account? That question is answered by this attention mechanism.

At a low level, this is done using the KQV matrices. K stands for Key - for which attention is needed, Q are the queries i.e for the token pointed by K, what are the tokens that needs to queried *(studied or should be taken for attention)* and finally V stands for Values i.e for the tokens pointed by Q, the corresponding Weights are taken as Values. This values are then multiplied (*looked at*) by the weights of the actual token (pointed by K) and that is the learning for the Kth token.

Model designers will need to set some values to decide on the model architecture `d_model` - model's hidden size `d_ff` - feedforward dimension `V` - Model's vocab size - list of total words / tokens that the model may learn `C` - Position indices to track - this is related to attention i.e the position of the token as part of the attention mechanism

Vocab size - the total size of the tokens used in training Context length - the total length of the model's memory.

All this put together will determine the model's total transformer, output and other blocks, the total parameter size and the total amount tokens that needs to be used for training.

Higher the model size (params), 20x times the token is needed for training. It is not a wrong thing to train a higher size model with lower tokens but it simply a wasted effort since the training performance can be achieved with lower size model itself. Hence if you end up having a higher size model, make sure you train that model with 20x tokens. This is termed as *Scaling Law*

FLOPS - Floating Point Operations FLOP/s - Number of FLOPS per second. Although it is not clear how one could estimate the number of FLOPS from the model size / architecture that was decided above. Perhaps the number of transformer blocks and the associated matmul operations could be used to estimate the FLOPS?

Parameters are the basically the variables that the model learns during the training. Weights and Biases. Hyperparameters are the varialble that one can control. Vocab Size, model dimensions etc. Setting the hyperparam heavily influence the model performance and hence setting this right is important and you won't get too many changes since the training run is costly. Hence there is a new field of research that extrapolate the performance of small models to tune the hyperparameters of large models.

In summary, three numbers signal a model's scale: - Number of parameters, which is a proxy for the model's learning capacity. - Number of tokens a model was trained on, which is a proxy for how much a model learned. - Number of FLOPs, which is a proxy for the training cost.

Post Training :

SFT - Supervised finetuning - Fine tune the pre-trained model with a labelled dataset Preference Finetuning - Further funetune the model to align the output responses to human preferences. This includes - RLHF : reinforcement learning with human feedback (llama2) - DPO : direct preference optimization (llama3)

Typical training pipleline

low quality data -> self supervised finetuning -> pretrained model -> SFT with labelled data -> SFT Model -> comparison data with RLHF -> Reward model -> Prompt engg with reward model -> final model

RLHF / Preference Fine tuning

RLHF uses human labellers / annotaters to reward comparison data. Humans cannot provide numeric rewards for a given prompt consistently. But they can pick a best response from given choices. This method is working but very slow and often expensive.

Sampling

This is also called as *Un Embedding Layer* . Sampling is basically choosing the best output from available samples. When the model predicts the next token, it produces a *logit* vector. In case of classification task, the logic vector simply 2 dimensions (yes/no) (spam/not spam) . Each element of the vector contains a probability of that class. i.e yes 90*, no 10% etc. In reality the logic vector contains the learned weights of the corresponding output token, which then sent via *Softmax* layer to convert that weight to probability.

For language model, the probabilistic vector works differently. In this case, the *logic* vector contains the vocab size dimension. i.e if the vocab size is 50,256, there would 50,2056 elements in the vector with a probability for each token. i.e `logit[245] = 0.10` would simply mean 245th token (could something like `t`)'s probability is 10%.

In language model, simply the most probable token won't be useful. Because if that's the case, you will always be seeing same output without any context based on the probability of the token patterns seen during the training phase. Instead for the language models, the probability is used as the probability of selecting that output token. i.e For example if the logit of `a` is `0.9` and `t` is `0.1`, then the output `t` will be chosen for 10% of the time and the output `a` will be chosen for 90% of the time, so on so forth.

And because there are too many logit elements on the logit vector due to sheer size of voab size, instead of doing probabilities on the *softmax* on the vector, it is done as *logprobs* i.e output probabilities are converted into logarithmic scale and then the probabilities are applied.

There are several other strategies used to sample the output token from those probabilities

1) Temperature - higher the value, more rare tokens are selected than more obvious tokens. This basically achieved by dividing the probability value by this temperature (`Xi/T`) so that it elevates the probability of less value tokens.
2) Top-k - Instead of taking the entire vocab dimension of the logit vector and computing the probabilities, select the top K elements and then do the probability calculation, thereby reducing the compute load.
3) Top-p - Also known as nucleus sampling, instead of selecting the top K samples, select the list of tokens that cumulatively satisfies the top p. If p is 90% (0.9) and if token `a` is 89% and `t` is 1%, then only these 2 tokens are selected.
4) Stopping Condition - Give a total token count to stop sampling or use a special token like eos, stop word to stop sampling.

Test Time Compute

The process of selecting the whole output for a completion task. The more the compute, the more token it can generate. Generating multiple responses for a single query often improves overall model performance but comes with the inference cost. This is the `choices []` seen at the OpenAI completions API response. OpenAI found that the model performance plateaued at 400 outputs mark. i.e if the number of outputs is > 400, it doesn't contribute to the model performance. Also to ensure accuracy, these multiple outputs can used to select the most consistent output as the accurate output.

Structured Output

Structured output can be achieved through 1) prompt engineering with examples - this often works but not always guaranteed 2) post processing - certain mistakes can be handled such as missing a {} or json output cutoff due to context length etc. 3) constrained sampling - Need model knowledge and the ability to influence the sampling using filter of accepted tokens 4) fine tuning - most expensive but most reliable way to train a model to output structured outputs

Probabilistic Nature of LLM

2 main problems with this nature

1) Inconsistent / Indeterministic output for the sample input / slightly different input
    1) Use caching as interim solution
    2) Use prompt engineering and memory systems
2) Hallucination - where the model generates its own facts not grounded in truth. This probably happens due to the fact that the model cannot distinguish between the training data and the data that it generates.
    1) How a model learns to produce its own data? 2 school of thoughts
        1) It happens during RLHF if the human annotaters are training the model with knowledge that the model doesn't know during training. This teaches the model that it is ok to generate new facts that are not seen in training
        2) The model knows that it generating hallucinating response but it still do it because it was told not to do so. Some try to mitigate by adding "Answer Truthfully" "if you are unsure say I don't know" in system prompts.

The two hypotheses discussed complement each other. The self-delusion hypothesis focuses on how self-supervision causes hallucinations, whereas the mismatched internal knowledge hypothesis focuses on how supervision causes hallucinations.

It is very hard to detect hallucinations in a generic fashion unless we know for sure the output is not in any training data.

Research tasks:

- ☐ How to determine the context length based on the model size / architecture?
- ☐ How to estimate FLOPS from the model size / architecture?
- ☐ How is the test time compute difference from prompting the LLM with same prompt again & again.

Chapter 3

Coming Soon...

Chapter 4

Coming Soon...

Chapter 5

Coming Soon...

Chapter 6

Coming Soon...

Chapter 7

Coming Soon...

Chapter 8

Coming Soon...

Chapter 9

Coming Soon...

Chapter 10

Coming Soon...

Book Summary

Coming soon...