Lctr 1: Transformers and Applications

Jianhua Liu

Fall 2024

Contents

1.1 Intro to the chapter

(Pages 2 and 3.)

Purpose of the chapter:

- Unveil the incredible power of the deceivingly simple O(1) time complexity of transformer models that changed everything.

- Build a notebook in Python and PyTorch to see how transformers hijacked hardware accelerators.
- Discover how one token (a minimal part of a word) led to the AI revolution we are experiencing.
- Discover how a hardly known transformer algorithm in 2017 rose to dominate so many domains.
- Introduce Foundation Models, which can do nearly everything in AI!
- Watch how ChatGPT explains, analyzes, writes a classification program in a Python notebook, and displays a decision tree.
- Introduce the role of an AI professional in the ever-changing job market. We will begin to tackle the problem of choosing the right resources.

This chapter covers the following topics:

- How one O(1) invention changed the course of AI history
- How transformer models hijacked hardware accelerators
- How one token overthrew hundreds of AI applications
- The multiple facets of a transformer model
- Generative AI versus discriminative AI
- Unsupervised and self-supervised learning versus supervised learning
- General-purpose models versus task-specific models
- How ChatGPT has changed the meaning of automation
- Watch ChatGPT create and document a classification program
- The role of AI professionals
- Seamless transformer APIs
- Choosing a transformer model

1.2 Intro to applications of Transformers

(Pages 1 and 2 and additional notes.)

Transformers can support industrialized, homogenized foundation Large Language Models (LLMs) designed for parallel computing that can handle emergent tasks. There are a few terms used here:

- Foundation models: They are transformer-based LLMs that have been trained on vast amounts of data and can perform various tasks without further fine-tuning. These models are designed to be general-purpose AI engines that can learn from and generate human-like text.
- Homogenization: The same model can be used across many domains with the same fundamental architecture. Foundation Models can learn new skills through data faster and better than any other model.
- Emergence: Transformer models that qualify as Foundation Models can perform tasks they were not trained for. They are large models trained on supercomputers. They are not trained to learn specific tasks like many other models. Foundation Models learn how to understand sequences.

ChatGPT popularized the usage of transformer architectures that have become general-purpose technologies.

Applications are burgeoning everywhere, and there are many big boys and new companies: Google Cloud AI, Amazon Web Services (AWS), Microsoft Azure, OpenAI, Google Workspace, Microsoft 365, Google Colab Copilot, GitHub Copilot, Hugging Face, Meta, and many others.

The functionality of transformer models has pervaded every aspect of our workspaces with Generative AI for text, Generative AI for images, discriminative AI, task specific-models, unsupervised learning, supervised learning, prompt design, prompt engineering, text-to-code, code-to-text, and more. Sometimes, a GPT-like model will encompass all these concepts!

The societal impact is tremendous.

- Developing an application has become an educational exercise in many cases.
- A project manager can now go to OpenAI's cloud platform, sign up, obtain an API key, and get to work in a few minutes. Users can then enter a text, specify the NLP task as Google Workspace or Microsoft 365, and obtain a response created by a Google Vertex AI or a ChatGPT transformer model.
- Users can go to Google's Gen App Builder and build applications without programming or machine learning knowledge.

The numbers are dizzying. Bommasani et al. (2023) created a Foundation Model ecosystem that lists 128 Foundation Models 70 applications, and 64 datasets. The paper also mentions Hugging Face's 150,000+ models and 20,000+ datasets! The list is growing weekly and will spread to every activity in society.

1.3 What AI professionals can do?

(See Page 2 and Pages 28 and 29 of the book.)

AI professionals must adapt to the new era of increasingly automated tasks, including transformer model implementations. If we list transformer NLP tasks that an AI specialist will have to do, from top to bottom, it appears that some high-level tasks require little to no development from an AI specialist.

1.3.1 AI development ecosystem with the *embedded transformers*

The recent evolution of NLP AI can be termed as "embedded transformers" in assistants, copilots, and everyday software, which is disrupting the AI development ecosystem:

- LLMs are currently embedded in several Microsoft Azure applications with GitHub Copilot, among other services and software.
- The embedded transformers are not accessible directly but provide automatic development support, such as automatic code generation. They also provide summarization, question and answering capabilities, and many other tasks in a growing number of applications.

1.3.2 Basic requirements for becoming an AI professional

- The skill set of an LLM AI professional requires adaptability, cross-disciplinary knowledge, and above all, flexibility.
- This book will provide the AI specialist with a variety of transformer ecosystems to adapt to the new paradigms of the market.
  - These opposing and often conflicting strategies leave us with various possible implementations.
  - Working through these problems will be a great training of basic skills.
- The future of AI professionals will expand into many specializations.

1.3.3 Specific work of an AI specialist

An AI specialist can be an AI guru, providing design ideas, explanations, and implementations.

The pragmatic definition of what Transformers represent for an AI specialist will vary with the ecosystem.

- API: Using APIs, such as the OpenAI API, does not require an AI developer. A web designer can create a form, and a linguist or Subject Matter Expert (SME) can prepare the prompt input texts. The primary role of an AI specialist will require linguistic skills to show, not just tell, the LLMs and their successors how to accomplish a task using examples. This new task is named prompt engineering.
- Library: There are many libraries that require limited development to start with ready-to-use models. An AI professional with linguistics and NLP skills can work on the datasets and the outputs. The AI professional can also use various toolkits to build tailored models for a project.
- Training and fine-tuning: Some Hugging Face functionalities require limited development, as they provide both APIs and libraries. However, sometimes, we still have to get our hands dirty. In that case, training and fine-tuning the models and even finding the correct hyperparameters will require the expertise of an AI specialist.
- Development-level skills: In some projects, the tokenizers and the datasets do not match. Or the embeddings of a model might not fit a project. In this case, an AI developer working with a linguist, for example, can play a crucial role. Therefore, computational linguistics training can come in very handy at this level.

1.3.4 The future of AI professionals

The societal impact of Foundation Models should not be underestimated. Prompt engineering has become a skill required for AI professionals. (Many tools, such as LangChain will be able to help to do this systematically.)

- An AI professional will be involved in machine-to-machine algorithms using classical AI, IoT, edge computing, and more.
- An AI specialist will also design and develop fascinating connections between bots, robots, servers, and all types of connected devices using classical algorithms.

Here are some domains an AI professional can invest in:

- AI specialists have specialized knowledge or skills in one of the fields of AI. One of the interesting fields is fine-tuning, maintaining, and supporting AI systems.
- AI architects design architectures and provide solutions for scaling and other information for deployment and production.
- AI experts have authoritative knowledge and skills in AI. An AI expert can contribute to the field with research, papers, and innovative solutions.
- AI analysts focus on data, big data, and model architecture to provide information for other teams.
- AI researchers focus on research in a university or private company.
- AI engineers design AI, build systems, and implement AI models.
- AI managers leverage the critical skills of project management, product management, and resource management (human, machine, and software).
- Many other fields will appear showing that there are many opportunities in AI to develop assets, such as datasets, models, and applications.

1.4 What tools and resources should we use?

1.4.1 Good suggestions from the book

(Pages 30 and 31 of the book.)

Suggestions for choosing the ecosystems:

- If you only focus on the solution you like, you will most likely sink with the ship at some point.
- Focus on the systems you need, not the ones you like.
- Learn enough **transformer ecosystems** and be flexible and adapt to any situation you face in an NLP project.

Suggestions for making decisions based on three simple principles:

- Understand the task(s) the AI model is expected to do at an Subject Matter Expert level. Write it down on paper without ChatGPT or any help. Then, write a detailed flow chart.
- Build a prototype dataset that can be duplicated to whatever scale you need to test the hardware/software performances.
- Try one to a few NLP models to see which one fits your needs: open source or not, API or not, local installation or cloud platform, and the other constraints of your project.

Take your time. Use science to guide you, not opinions.

1.4.2 Tool suggestions from the book

See pages 31 to 34 of the book.

1.4.3 Tools beyond the book

Beyond the book contents, we will discuss the tools briefly and demonstrate the usage of the following tools that can chat with local docs and runs locally without needing GPU:

- GPT4ALL
- Open WebUI with Ollama

See llm-tools–mr-4-various-tools.

1.5 How Constant Time Complexity O(1) Changed Our Lives Forever

1.5.1 A brief journey from recurrent to attention

1.5.1.1 A first encountering with recurrent and attention layers

Recurrent layers are the basic building blocks of an RNN:

- Process input sequences one element at a time.
- Use internal memory to keep track of the hidden states obtained from previous elements of the sequence.
- Compute output based on current input, hidden state, and weights.
- Complexity for sequential operations: O(n), where n is the sequence length.

Attention Layers are the basic building blocks of a transformer:

- Process entire sequences simultaneously (not one element at a time).
- Use the concepts of query, key, and value, to obtain the attention values.
- Compute weighted sum of input elements based on attention weights.
- Complexity for sequential operations: O(1) since we only need to do pairwise operations.

1.5.1.2 From recurrent to attention

(Pages 15 and 16.)

See book.

1.5.2 O(1) attention conquers O(n) recurrent methods

(Pages 3 to 5.)

To navigate the complex landscape of transformer-based technologies, it's essential to grasp the fundamental principles rather than getting lost in the sheer volume of options. For this, we can use the following questions to guide us.

- How could this deceivingly simple $O(1)$ time complexity class forever change AI and our everyday lives?
- How could $O(1)$ explain the profound architectural changes that made ChatGPT so powerful and stunned the world?
- How can something as simple as $O(1)$ allow systems like ChatGPT to spread to every domain and hundreds of tasks?

This section will provide a significant answer to those questions before we move on to see how one token (a minimal piece of a word) started an AI revolution that is raging around the world, triggering automation never seen before.

To achieve that goal, in this section, we will use science and technology to understand how all of this started. First, we will examine $O(1)$ and then the complexity of a layer of a transformer through a Python and PyTorch notebook.

Let's first get the core concepts and terminology straight for O(1).

As we will see later, the transformer is based on the **attention** mechanism, and the traditional RNN on the **recurrent** method, both are used for sequence processing. The former depends only on the tokens in the current context window, and the latter depends on all the tokens in the past.

They are not as clean-cut as the author claimed as $O(1)$ and $O(n)$. Anyway, we just assume it is correct and move on.

In Chapter 2, Getting Started with the Architecture of the Transformer Model, we will explore the architecture of the transformer model.

In this section and chapter, we will first focus on what led to the industrialization of AI through transformers and the ChatGPT frenzy: the exponential increase of hardware efficiency with self-attention. We will discover how attention leverages hardware and opens the door to incredible new ways of machine learning.

Consider the following sentence which has $n = 11$ words: "Jay likes oranges in the morning but not in the evening."

The problem of language understanding can be summed up in one word: context, without which a word can rarely be defined.

Let's begin with a conceptual approach of an attention layer.

1.5.2.1 The attention layer

This section of the book can be misleading. Here, we make a few notes:

- The dimensions here are just for illustrative purposes; they are not the real dimensions of the embedding vector, where the relationship between different words are in expressed across multiple directions.
- When we discuss the pairwise operations, we will have n^2 pairs if we have n words in a sentence, so $O(1)$ does not make a lot of sense.
- Even if we consider the relationship between one word with the other words, it is still n pairs, which should be $O(n)$, especially when n increases.
- d in the book is the dimension of the embeddings, which will be discussed later.
- The $O(1)$ concept comes from the authors of the Transformer paper. What the authors really meant with $O(1)$ was that when finding the pairwise operation, the **sequential computation complexity** is $O(1)$. Accordingly, the maximum path length of the computation is $O(1)$.
- As we will only calculate **correlations between the embeddings**, the calculation of each pair is $O(d)$, and hence the total calculation is then $O(n^2 \times d)$.

1.5.2.2 The recurrent layer

This section of the book can be misleading. Here, we make a few notes:

- Instead of n^2 pairs, we only have to calculate n operations.
- Each operation is $O(d^2)$, as we are not confined to calculate *correlations between the embeddings*.
- The total operations will then be $O(n \times d^2)$.
- As we will see later, we usually have $n > d$, so the total number of calculations of a recurrent layer is no more than that of an attention layer.
- Yet, in contrast to the **sequential computation complexity** of $O(1)$ for the attention layer, the *sequential computation complexity* of a recurrent layer is $O(n)$ as it will need to wait to the very end of the sentence before the computation is done. Accordingly, the maximum path length of the computation is $O(n)$.

1.5.2.3 Advantages of the attention layer

- As hinted by the above discussions and will be shown later, the beauty of the attention mechanism is that we can do parallel processing on it while the recurrent layer can only be done in a series way. With parallel processing, the transformers can be trained using an array of GPUs to improve the training speed.
- Of course, the transformer architecture has other advantages over other architectures, including recurrent-based RNNs.

1.5.3 The magic of the computational time complexity of an attention layer

(Pages 5 to 9 of the book.)

Here, we duplicate the IPython notebook named `0-1_and_Accelerators.ipynb` below with minor changes:

This notebook illustrates the complexity for self-attention and recurrent layers. The functions are not the actual algorithms of Transformers and RNNs. They simply show how the complexity of the layers varies with a CPU and a GPU. **Run the notebook cell by cell and make sure to choose the right type of processor. If you don't have access to a GPU or a TPU, you can read the notebook.**

Self-attention layers benefit from matrix multiplications whereas recurrent layers are mostly sequential.

Reference for complexity: Attention is All You Need, Vaswani et al.(2017), page 6:

1.5.3.1 The CPU version

```python
# Comparing the computational time between:
# self attention = O(n^2 * d) and recurrent = O(n * d^2)
import numpy as np
import time

# define the sequence length and representation dimensionality
n = 512
d = 512

# define the inputs
input_seq = np.random.rand(n, d)

# simulation of self-attention layer O(n^2*d)
start_time = time.time()
for i in range(n):
    for j in range(n):
        _ = np.dot(input_seq[i], input_seq[j])
attention_time = time.time() - start_time
print(f"Self-attention computation time: {attention_time} seconds")

# simulation of recurrent layer O(n*d^2)
start_time = time.time()
hidden_state = np.zeros(d)
for i in range(n):
    for j in range(d):
        for k in range(d):
            hidden_state[j] += input_seq[i, j] * hidden_state[k]
recurrent_time = time.time() - start_time
print(f"Recurrent layer computation time: {recurrent_time} seconds")

# Calculate the total time
total = attention_time + recurrent_time

# Calculate the percentage of at
percentage_at = round((attention_time / total) * 100, 2)

# Output the result
print("The percentage of 'computational time for attention' in the " \
        + f"sum of 'attention' and 'recurrent' is {percentage_at}%")
```

Output on Surface Book 3:

```
Self-attention computation time: 0.3444480895996094 seconds
Recurrent layer computation time: 63.41322684288025 seconds
The percentage of 'computational time for attention' in the sum of 'attention'
    and 'recurrent' is 0.54%
```

Output on Dell XPS:

```
Self-attention computation time: 0.13318800926208496 seconds
Recurrent layer computation time: 27.33692693710327 seconds
The percentage of 'computational time for attention' in the sum of 'attention'
    and 'recurrent' is 0.48%
```

1.5.3.2 The GPU version

```
# PyTorch version
import torch
import time

# define the sequence length and representation dimensionality
n = 512
d = 512

# Use GPU if available, otherwise stick with cpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

# define the inputs
input_seq = torch.rand(n, d, device=device)

# simulation of self-attention layer O(n^2*d)
start_time = time.time()
_ = torch.mm(input_seq, input_seq.t())

attention_time = time.time() - start_time
print(f"Self-attention computation time: {attention_time} seconds")

# simulation of recurrent layer O(n*d^2)
start_time = time.time()
hidden_state = torch.zeros(d, device=device)
for i in range(n):
    for j in range(d):
        for k in range(d):
            hidden_state[j] += input_seq[i, j] * hidden_state[k]
    recurrent_time = time.time() - start_time
    if recurrent_time > attention_time * 10000:
        break

recurrent_time = time.time() - start_time
print(f"Recurrent layer computation time: {recurrent_time} seconds")

# Calculate the total time
total = attention_time + recurrent_time

# Calculate the percentage of at
percentage_at = round((attention_time / total) * 100, 2)
```

```

# Output the result
print("The percentage of 'computational time for attention' in the " \
        + f"sum of 'attention' and 'recurrent' is {percentage_at}%")
```

Output on a Dell XPS GPU PC:

```
cuda
Self-attention computation time: 0.00023889541625976562 seconds
Recurrent layer computation time: 2.497342348098755 seconds
The percentage of 'computational time for attention' in the sum of 'attention'
    and 'recurrent' is 0.01%
```

1.6 Transformer-based foundation models

(Pages 21 to 27.)

1.6.1 Definitions of foundation models

Foundation models are large language models with billions of parameters that have been trained on clusters of GPUs on billions of recorded data. These models can perform a wide range of **emergent** tasks without further retraining or fine-tuning.

1.6.2 From general purpose to specific tasks

Foundation models not only can perform general purpose tasks, they can perform specific tasks such as generating code.

See the book for an example of generating the code by ChatGPT using the following requests/prompts:

```
1. Provide a sklearn classification of Iris with some kind of matplotlib graph
    to
describe the results, and don't use OpenAI APIs.
2. Write a detailed explanation for a beginner in Python for this code
```

1.7 From one token to AI revolution—using a pipeline based on a foundation model

(Pages 17 to 19 of the book.)

1.7.1 What is a token?

Most times, the input to the foundation models, in terms of requests or prompts, is a sequence of text. The text has to be converted to **tokens**.

In LLMs, a token refers to the smallest unit of text that can be processed by the model. Tokens are like individual words, but not necessarily whole words. Tokens can be:

- Whole words: A single word, such as “hello” or “dog”.
- Subwords: Part of a word, often represented using special characters (e.g., ## for spaces). For example: “eating” can be decomposed into two tokens: “eat” and “ing”, and the second is a subword token.

- Characters: Individual letters or symbols within a word, such as a, b, or !.

Note that

- We need to use a tokenizer to convert a text into a sequence of tokens.
- There are many different tokenizers, as we will discuss later.
- Each LLM has its accompanying tokenizer. When we write code for NLP, we need to use the correct tokenizer.

1.7.2 A pipeline with an LLM application

An LLM runs on a pipeline, which works like a series of function calls. Note that later, we will learning a special pipeline called Hugging Face `pipeline`. Page 18 of the book provides a figure to illustrate a pipeline for text generation, which is detailed in the following paragraphs.

1.7.3 Producing a token from input and other generated tokens

In a real case of LLM application, the output token will be fed back to the original input sequence to form a new input sequence to the model, and the output token can be expressed as

$$t_i = f(t_1, t_2, \cdots, t_{i-1})$$

where:

- t_i is the ith token of the model.
- f is the model and the output controller that infers the next token.

1.7.4 From token to different models and tasks

There are a number of different models and tasks based on tokens.

1.7.4.1 Discriminative vs generative

Traditionally, discriminative and generative models are two fundamental approaches in machine learning.

- A discriminative model, also known as a conditional model, is a machine learning algorithm that learns the boundaries between different classes or categories in a dataset. These models are used in supervised machine learning applications like classification and regression.
- A generative model is a type of machine learning model that aims to learn the underlying patterns or distributions of data in order to generate new, similar data.

In the context of LLM, the term “Generative AI” has often been used to describe GPT- like models, which can be used to generate text-based contents.

1.7.4.2 Supervised vs unsupervised

Traditionally, supervised models are trained based on labeled pairs, and unsupervised models do not need to use the labeled pairs. An example of the former is the image classification, and an example of the latter is the K-Means clustering approach.

The training of LLMs is a combination of both the supervised and unsupervised approach. In the first stage of learning, we just need to use text paragraphs; we can use the first few words to predict the next word, which can be seen as a both supervised and unsupervised. These LLMs are called **base** LLMs.

In the later stages, the base LLMs will go through supervised training for Q&A and chat, and these models are called **instruct** or **chat** models. Sometimes, an instruct model can also be used to support chatting.

1.7.4.3 Task-specific vs specific task models

Task-specific models are often opposed to general-purpose models, such as GPT-like LLMs. Task-specific models are trained to perform very well on specific tasks. This is accurate to some extent for some tasks. However, LLMs trained to be general-purpose systems can surprisingly perform well on specific tasks, as will be shown later.

1.7.4.4 Interaction vs automation

For LLM-based applications, we often use **interaction**-based approach to get the feeling of the capabilities of the LLMs and then move on to write code to **automate** the tasks.