O'REILLY®

Transformers
The Definitive Guide
Applications Beyond NLP

Early Release
RAW & UNEDITED

Nicole Koenigstein

Transformers: The Definitive Guide

Applications Beyond NLP

Nicole Koenigstein

Beijing • Boston • Farnham • Sebastopol • Tokyo

Transformers: The Definitive Guide

by Nicole Koenigstein

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

- Acquisition Editor: Nicole Butterfield
- Development Editor: Sarah Grey
- Production Editor: Elizabeth Faerm
- Copyeditor: TO COME
- Proofreader: TO COME
- Indexer: TO COME
- Interior Designer: David Futato

- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- July 2025: First Edition

Revision History for the Early Release

- 2024-05-31: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098167011 for release details.

rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-16695-3

[LSI]

Brief Table of Contents (*Not Yet Final*)

Chapter 1. Intro to Transformers and Attention

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo is avaialble at *https://github.com/Nicolepcx/transformers-the-definitive-guide*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

Since its introduction in 2017, the transformer architecture [1] has revolutionized the field of natural language processing (NLP), marking a paradigm shift towards models capable of natural language understanding (NLU). This shift was possible because transformers process sequential data in parallel, enabling a deeper and more contextual understanding of language than was achievable with previous sequential models, like long short-term memory (LSTM) networks.

In recent years, transformers have evolved to impact a wide array of domains, including computer vision, speech recognition, reinforcement learning, and mathematical operations, moving beyond their initial usage within NLP. Their adaptability has led to significant advancements in machine translation, allowing

for context-aware translations, and in scientific research, notably in predicting protein structures with remarkable accuracy.

Among the most exciting developments are multimodal models. These models can process and interpret multiple data types simultaneously, such as text and images, demonstrating the architecture's capability to integrate and learn from various data types and opening the doors to innovative applications that could transform our interaction with technology.

I assume in this book that you have at least some familiarity with the transformer architecture. Perhaps you're read the book Natural Language Processing with Transformers, or a similar work. This chapter aims to provide you with a comprehensive understanding of the transformers architecture to set the stage for the more advanced and complex models, beyond NLP, that I will cover in later chapters. This chapter briefly reviews the main parts of the model, to refresh and solidify your knowledge. I will start off by explaining the basic transformer architecture. Then I'll cover how longer context is possible, and finish with a tour through the various types of attention mechanisms.

Transformer basics

This section explains the main architectural components of the original transformer model, such as encoder and decoder, positional embeddings, and attention mechanism.

The transformer architecture was originally developed for machine translation, a challenging sequence-to-sequence task in which the concept of tokenization plays a critical role. *Tokenization* breaks down sequences-like sentences into manageable

units, or tokens, that the transformer can effectively process. For example, in the sentence:

```
The Transformer has revolutionized NLP.
```

the word *the* represents a single word-level token.

Before we dive into the architectural components, understanding tokenization is crucial, as it facilitates the transformer's ability to interpret text. And sets the foundation for its application to other sequences.

Tokenizer: Text representation in the transformer

A *tokenizer* is used to tokenize the text. This is the first step to make natural language digestible for the model, before applying token embeddings and finally positional embeddings. The different types of tokenization are:

Character-level tokenization

Character-level tokenization splits the underlying alphabet into each existing character in the sequence. If you used character-level tokenization for

```
"The Transformer has revolutionized NLP."
```

it would yield:

```
[T, h, e ,' , … 'N, L, P, .]
```

This will lead to very long sequences, which can increase computational complexity. It can also be challenging for the model to learn long-term dependencies. Nonetheless, this can be helpful if your task requires a fine-grained understanding.

Word-level tokenization

Word-level tokenization would split the example sentence as follows:

```
[The, Transformer, … NLP, ., ]
```

that is, the sequence will be split into its words, plus punctuation. The downside is that this requires a large vocabulary, and if the language changes, this tokenization will not be able to understand new words.

Subword tokenization

Most modern LLMs use *subword tokenization*, in which the word is split into smaller parts. For instance, a subword tokenizer would split the word *hiking*, the tokenizer into:

```
[h, ik, ing ]
```

and the word *cooking* into:

```
[cook, ing]
```

So subword tokenization splits a word (or sequence) into smaller, commonly occurring chunks, like

```
[ing]
```

Single-characters words are also included.

Now that you understand the basics of tokenization, let's move on to token and positional embeddings.

Token and positional embeddings

A part of the transformer architecture that contains learnable parameters is the token and positional embeddings (PE). The token embedding is tasked with encoding each vocabulary element into a d-dimensional vector in the space of $\mathbb{R}$ of d_e ($\mathbb{R}^{d_e}$). The token embedding can mathematically be presented as follows:

Let V be the vocabulary with $|V| = N_v$, where each word w in V is assigned a unique token ID, $v \in \{1, 2, ..., N_v\}$. The token embedding is a function $e : \mathbb{N} \rightarrow \mathbb{R}^{d_e}$ that maps each token ID to a d_e-dimensional vector. This is achieved through a token embedding matrix $W_e \in \mathbb{R}^{d_e \times N_t}$, where d_e is the dimensionality of the embeddings. Here's how to do it using bidirectional encoder representations from transformers (BERT):

```
from transformers import AutoTokenizer, AutoModel

tokenizer = AutoTokenizer.from_pretrained('bert-base-
model = AutoModel.from_pretrained('bert-base-uncased'

sentence = "The Transformer has revolutionized NLP."
inputs = tokenizer(sentence, return_tensors='pt') ❷
input_ids = inputs['input_ids'] ❸

print(input_ids)
outputs = model(input_ids)
```

```
embeddings = outputs.last_hidden_state ❹
print(embeddings)
```

❶ Load tokenizer and model from Hugging Face

❷ Tokenize the sentence

❸ Get the input IDs and pass them through the model to get the embeddings

❹ Get the last hidden state, to access the embeddings of the tokens

This will result in the following output for the input IDs of the sentence:

```
tensor([[ 101, 1996, 10938, 2121, 2038, 4329, 3550, 17953, 2361, 1012, 102]])
```

For the corresponding embeddings the output is:

```
tensor([[[-0.5249, -0.2210,  0.2696,  ..., -0.4204,  0.2605,  0.6457],
         [-0.6665, -0.4994,  0.4651,  ..., -0.2517,  0.2334,  0.0176],
         [ 0.8416, -2.0561,  0.8323,  ..., -0.2709, -0.1999, -0.1918],
         ...,
         [-0.4018, -0.6402,  0.7791,  ..., -0.0290, -0.4070,  0.2974],
         [-0.3327, -0.8091, -0.0304,  ...,  0.4745,  0.3230, -0.5991],
         [ 0.4928, -0.0878, -0.0971,  ...,  0.1629, -0.7012, -0.3848]]],
       grad_fn=<NativeLayerNormBackward0>)
```

This representation lacks the position of the word in the sequence. And since the transformer does not have *recurrence*, meaning that it doesn't need to process the data sequentially as it was originally represented, you need a function to represent the position. This is why you need to add positional embeddings: without them the model treats sequences as unordered collections of words.

The positional-embedding function learns to encode a token's location within a sequence into a vector in the space $\mathbb{R}^{d_e}$. The original Transformer uses for position p_i:

$$p_{i,2t} = \sin\left(k/10000^{2t/d}\right)$$
$$p_{i,2t+1} = \cos\left(k/10000^{2t/d}\right)$$

Here $p_{i,2t}$ is the $2t^{th}$ element of the d-dimensional vector p_i. This means that the position of the first token is captured by a vector, $p[1]$, while the position of the second token is captured by a different learned vector $p[2]$, and so on.

This technique enables transformer models to understand the order of words. In the next section you'll see how the transformer uses this vector representation to understand and learn from the text.

Attention mechanism

The attention mechanism is at the core of the transformer's ability to understand and interpret text. It gives the model the ability to analyze the relevance of a word in a sequence on a token-to-token basis.

In that context, you will often hear the term *attribution matrix*, which is computed from the input embeddings. Here the term *attribution* refers to the significance between different parts of the input. The attribution matrix is computed with the Q (query) and the K (key) matrices. The resulting scores form the Q and K interaction to determine the attention weights, which are then applied to the V (Value) matrix to produce the output of the attention mechanism:

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

This attribution matrix is crucial for understanding how the model interprets and processes the corresponding input sequences. For instance, by analyzing these scores, you can gain insights into the model's decision-making process, such as which tokens it sees as more relevant than others when generating the output token. Libraries such as Captum help make this decision-making process visible.

However, despite the specific roles of Q, K and V, the initial computation for each of these matrices follows a similar process: a *linear projection* of the input embeddings. This means that for each of these matrices, the input embeddings are multiplied by a weight matrix. This process can be mathematically described as follows:

- Query matrix Q: $Q = W_q E$
- Key matrix K: $K = W_k E$
- Value matrix V: $V = W_v E$

Here E represents the input embeddings, and W_q, W_k and W_v are the weight matrices for the query, key and value projections, respectively. Take the dot product of the query and key matrices, followed by the softmax function and the scaling factor (for scaled dot attention). The result will be a matrix of scores representing self-attention, or how much focus each token should put on each other token by considering its relationship with every other element in the sequence. These scores are then used to weight the values in the V matrix, producing the final weighted-sum output of the attention mechanism:

$$Output = AttentionScores \times V$$

This dynamic process allows the model to focus on different parts of the input sequences for each input token, making it possible to understand each token's contextual relevance and information.

Multi-head attention

The attention mechanism you've seen so far represents the computation performed by a single attention head, which is the component responsible for calculating attention in the transformer. However, the original transformer, as well as state-of-the-art (SOTA) models apply multiple attention heads simultaneously. Each individual attention head has its own learnable parameters, which are then combined into a single output. This allows the model to integrate information from the same sequence and capture a variety of relationships between its words or elements. This approach enhances the model's ability to understand and represent complex dependencies in the data.

In technical terms, given input sequences A, B and C, . . ., the multi-head attention mechanism computes new representations for the elements in A by considering information from B, C, and so on. This process involves several steps: each head computes its own attention scores and output vectors based on the input, then concatenates and linearly transforms these outputs to produce the final output vector V.

This process consolidates the contextual information captured by the individual attention heads into one unified output that encapsulates all critical information across the entire input sequence. Since each attention head might focus on different relationships within the input sequence, this is crucial to the model gaining a better language understanding.

Bidirectional and unidirectional attention

As I mentioned, the first transformer model was used for machine translation. That's why it uses two distinct types of attention mechanism within the architecture: one for the encoder, and another for the decoder.

First, the encoder applies bidirectional self-attention, not just left-to-right processing, as traditional sequence processing methods do. This means it treats all tokens as context, applying attention to each token in the sequence. This gives the model a full understanding of the entire input sequence when it generates representations for each token.

The decoder's attention is masked (also called *causal attention*) to prevent the model from attending to future tokens (subsequent positions). In practice, this means that for the prediction i, the model can only attend to the position $< i$. With that method in place, the model generates each token based only on the tokens previously created, from left to right, thus preventing it from using future tokens in the sequence. This is important for all task where the model must generate one token at a time as, for instance, for translation.

Now that you understand the two distinct variations of attention used with the first transformer, let's look at the encoder and decoder.

Encoder and decoder parts

The first transformer model's architecture (Figure 1-1) was characterized by its encoder-decoder structure. Some subsequent models leverage a decoder-only framework, such as GPT, LLaMA, Mistral, and Falcon.

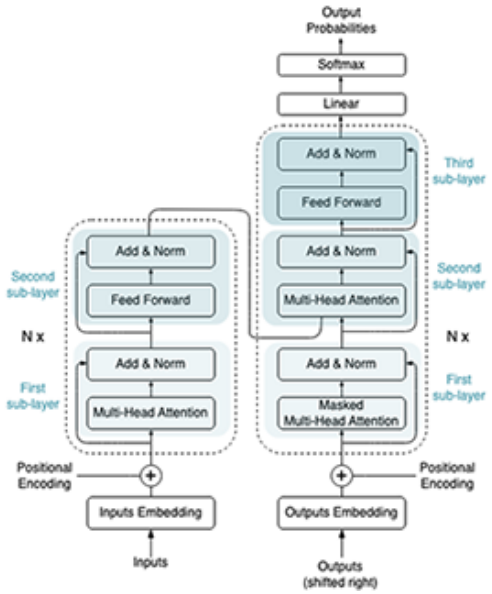

Figure 1-1. Encoder and decoder part of the Transformer architecture.

The encoder itself is composed of six identical layers, each containing two principal components: a multi-head self-attention mechanism and a point-wise fully connected feed-forward network. The term *point-wise* refers to applying the same linear transformation to each sequence element. These components are further refined with residual connections and layer normalization.

The decoder interprets the encoded information, mirroring the encoder's layered structure but introduces an essential feature: *masked multi-head self-attention*. This added feature in the decoder prevents the model from accessing subsequent positions in the sequence.

The model maintains a consistent output dimension of 512 across all sub-layers, including the embedding layers, meaning its maximum sequence length is 512 tokens. This limitation comes mostly from the specific architectural setup of the first transformer model, which made it hard to process longer sequences on the available hardware efficiently.

Enhancements in transformer design: Longer context and attention variations

Now it’s time to look into methods by which modern transformer models, like GPT-4, achieve higher levels of performance and flexibility. In particular the ability to process more information at once, through longer context windows. Attention-mechanism variations such as multi-query and flash attention also increase the efficiency and accuracy of SOTA transformer models.

Longer context windows with better performance

A model’s *context window* refers to the portion of text it can process when making predictions or generating text. A longer context window allows the model to understand more complex narratives and capture nuances better than it could using a chunked version of a text with a small context window.

However, simply extending the context length results in quadratic increases in time complexity and memory usage, which can constrain improvements. Therefore, recent enhancements, such as *rotary positional embedding* (RoPE) [2], *position interpolation* (PI) [3] and *Yet another RoPE extensioN method* (YaRN) [4], are designed to more effectively manage longer contexts during inference.

RoPE brings *absolute* and *relative* PEs together. But before I dive deeper into how RoPE works, let’s first look at the key differences between absolute and relative PEs.

- With absolute PEs, for each token embedding, the model adds information about the absolute position of the token. Absolute PEs are simpler and faster to compute.
- Relative PEs consider distances between sequence elements and can be shared across sequences, which helps the model to understand and interpret the relationships and distances between different tokens within a sequence. Relative PEs result in an increase in performance, but are computationally more complex.

RoPE combines absolute and relative positional embeddings, representing a significant advancement in the design of transformer models. These models process longer sequences of text more naturally and accurately, while maintaining efficiency.

Specifically, RoPE integrates a rotation matrix, $R_{\Theta,m}$, to encode the absolute positions of tokens, incorporating the explicit dependency of relative positions into the self-attention mechanism. To illustrate RoPE's implementation more concretely, consider a model with dimension $d = 6$, which then can be computed as follows:

$$R^6_{\Theta,m} = \begin{matrix} cosm\theta_1 & -\sin m\theta_1 & 0 & 0 & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cos m\theta_3 & -\sin m\theta_3 \\ 0 & 0 & 0 & 0 & \sin m\theta_3 & \cos m\theta_3 \end{matrix}$$

Higher dimensions are divided into $d/2$ subspaces, so the dimension number has to be even. Let's put the math into code to make the theoretical concept more clear:

```
def simple_rotary_matrix(d, m, max_len):
    assert d % 2 == 0, "Embedding dimension must be ev

    theta = 10000 ** (-2 * torch.arange(d // 2).float
    theta *= m

    cos_theta = torch.cos(theta)❸
    sin_theta = torch.sin(theta)

    R = torch.zeros((d, d)) ❹

    R[torch.arange(0, d, 2), torch.arange(0, d, 2)] =
    R[torch.arange(0, d, 2), torch.arange(1, d, 2)] =
    R[torch.arange(1, d, 2), torch.arange(0, d, 2)] =
    R[torch.arange(1, d, 2), torch.arange(1, d, 2)] =

    return R
```

❶ To ensure the dimension d is even, since this is required.

❷ Compute thetas

❸ Compute sin and cos for rotation

❹ Initialize the rotation matrix

❺ Compute the rotation matrix

To use the function, you can simply do the following:

```
d = 6 ❶
max_len = 10 ❷
```

```
R_matrix = simple_rotary_matrix(d, m=1, max_len=max_l
print(R_matrix)
```

❶ Embedding dimension d

❷ Sequence length

❸ Creates the rotation matrix

This creates the following rotary matrix:

```
tensor([[ 0.5403, -0.8415,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.8415,  0.5403,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.9989, -0.0464,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0464,  0.9989,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  1.0000, -0.0022],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0022,  1.0000]])
```

Figure 1-2 shows an illustrates the RoPE process.

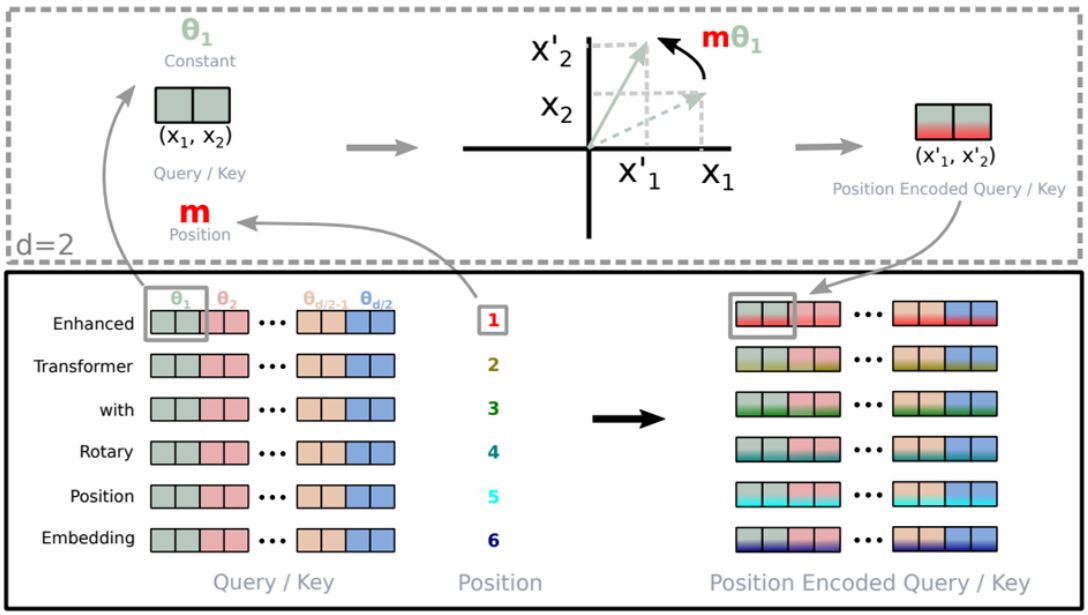

Figure 1-2. Illustration of Rotary Position Embedding(RoPE). Image adapted from: Jianlin Su et al.

To apply RoPE in the context of self-attention, define the relationship between the q_m in position m and key k_n in position n as:

$$q_m^T k_n = \left(R_{\Theta,m}^d W_q x_m\right)^T \left(R_{\Theta,m}^d W_k x_n\right) = x^T W_q R_{\Theta,n-m}^d W_k x_n$$

Here $R_{\Theta,n-m}^d = \left(R_{\Theta,m}^d\right)^T R_{\Theta,n}^d$ represents the rotary matrix adapting the relative positions.

RoPE enhances efficiency and accuracy, so it's used in SOTA models like LLaMA and, LLaMA 2. Even SOTA LLMs have a maximum number of tokens they can process at once. For instance, the LLaMA models can handle up to 2048 tokens in a single input.

This limitation becomes a problem in use cases that involve long prompts or extensive document summaries, where LLMs capable of managing more extensive contexts are desirable. However, it would take substantial computational resources to create a new LLM with an expanded context capability from the ground up. This raises an important question: Is it possible to increase the context window size of an already pre-trained LLM? The good news is: yes! PI and YaRN can extend these RoPE-based pre-trained LLMs with minimal fine-tuning. Figure 1-3 demonstrates the PI technique for a LLaMA model with a 2048 context window.

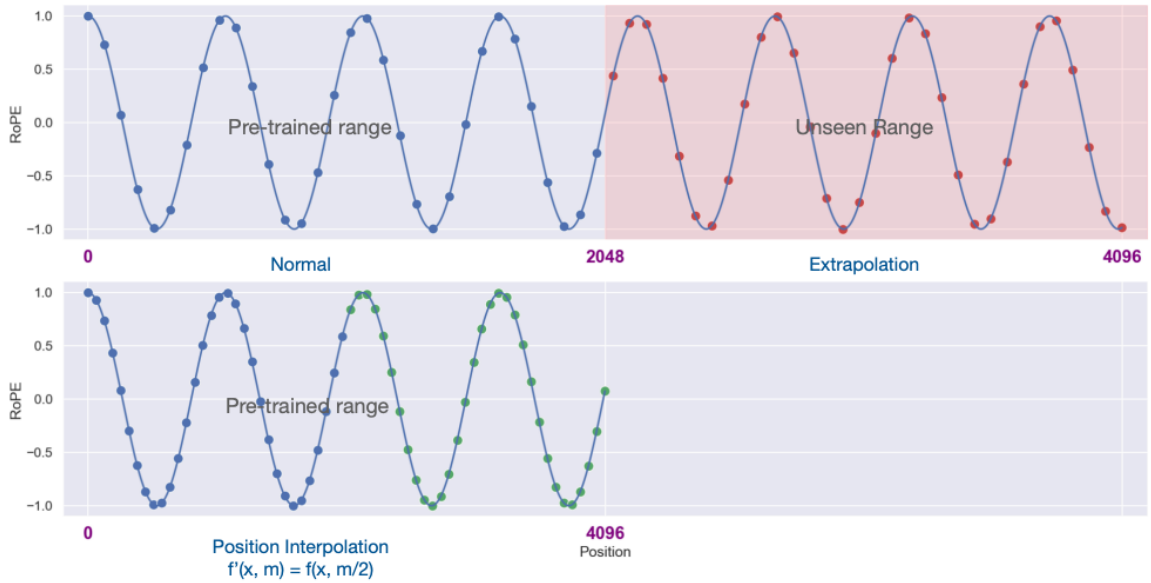

Figure 1-3. How the position interpolation (PI) method works for a LLaMA model with a 2048 context window. The blue dots stand for the training limit of LLMs; the red squares illustrate how models adapt to new positions. The blue dots and the green triangles demonstrate how PI scales down from [0, 4096] to [0, 2048] to keep them within the trained range.

Normally, LLM models use input positions (blue dots) within their trained range. For length extrapolation, models handle new positions (red squares) up to 4096. Position interpolation downscales these indices (blue dots and the green triangles) from [0, 4096] to [0, 2048], ensuring they stay within the pretrained range.

To extend the context window, PI interpolates the position indices within the pre-trained limit, with a small set of fine-tuning applied.

That is, PI extends RoPEs function f by f' as follows:

$$f'(x, m) = f\left(x, \frac{mL}{L'}\right)$$

Here $L' > L$ is a new context window beyond the pre-trained one.

Let me take a short step back and explain an important way to evaluate the performance of a model - *perplexity* (PPL). This is a measure how "surprised" or "perplexed" a model is about context. That is, perplexity measures on how well a probability model predicts a sample, with lower values indicating better predictive accuracy. Let me illustrate this with a concrete coding example:

```
from transformers import AutoModelForCausalLM, AutoTo
model = AutoModelForCausalLM.from_pretrained("tiiuae/f
tokenizer = AutoTokenizer.from_pretrained("tiiuae/fal

wiki_text = tokenizer("Apple Inc. is an American mult
                      "corporation and technology com
                      "in Cupertino, California, in S
                      return_tensors = "pt")

loss = model(input_ids = wiki_text["input_ids"],
             labels = wiki_text["input_ids"]).loss
ppl = torch.exp(loss)
print(ppl)

input_text = tokenizer("A Falcon is a generative tran
                       "model and it can't fly.", ret

loss = model(input_ids = input_text["input_ids"],
             labels = input_text["input_ids"]).loss ❶
ppl = torch.exp(loss)
print(ppl)
```

❶ Compute loss

The `wiki_text`

input yields a score of 5.08, while the

`input_text`

yields 121.19. This significantly higher perplexity score indicates that the model finds this sentence quite surprising or unlikely. This is because the model was most likely just trained on data indicating that a falcon is a bird known for its remarkable flying abilities, not a transformer model.

For evaluating LLM performance with longer context windows, you will use *sliding window perplexity*. This metric calculates perplexity over a fixed-size window of tokens, moving across the text, to better handle and evaluate large texts and datasets.

One downside of RoPE is that it expands token positional information into a multidimensional complex vector. It struggles with encoding high-frequency components, because its one-dimensional input limits its ability to distinguish between very similar and proximate tokens.

To address this, practitioners of *neural tangent kernel* (NTK) [5] theory developed, *NTK-aware interpolation*, adjusting the scaling of frequencies differently across dimensions to preserve high-frequency information. One of the applications of NTK theory is identifying and mitigating issues related to training neural networks, such as difficulties in learning high-frequency components or patterns in data with low *intrinsic dimensionality*, as is the case with RoPE. Intrinsic dimensionality refers to the minimum number of parameters needed to accurately describe a dataset without losing significant information, representing the dataset's inherent complexity.

However, NTK-aware interpolation can stretch some dimensions beyond their bounds, potentially degrading the model's performance. Additionally, *NTK-by-parts interpolation* and *dynamic NTK interpolation* were introduced as refined strategies, focusing on preserving relative local distances and adapting scale factors dynamically for varying sequence lengths, respectively.

Building upon these NTK-techniques, YaRN introduces a temperature t to the attention scores before the attention softmax, uniformly affecting perplexity across different data samples and token positions. This approach modifies attention weight computation and utilizes a length-scaling technique that adjusts both q_m and k_n by a constant factor, enhancing the attention mechanism without altering its underlying code. RoPE embeddings, pre-generated and reused, facilitate this process with no additional computational cost during inference or training. When combined with *NTK-by-parts interpolation*, YaRN performs effective in models like LLaMA and LLaMA 2 see Figure 1-4.

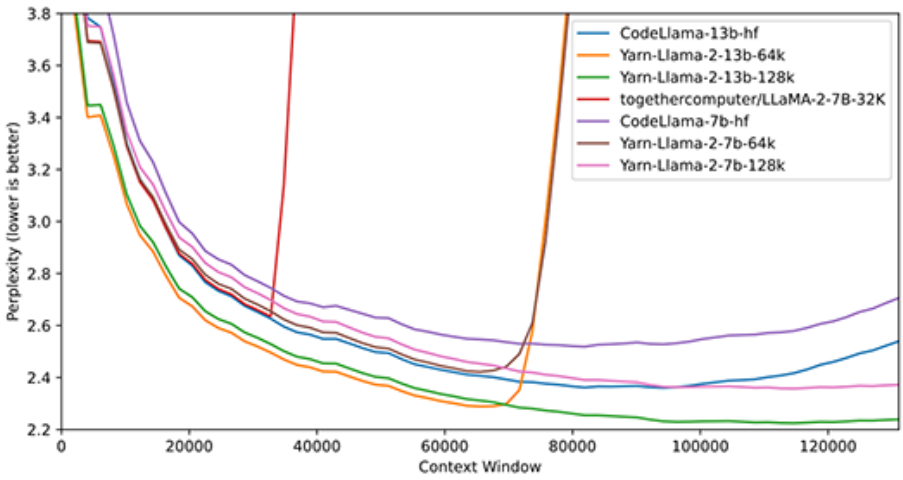

Figure 1-4. How the context window can affect the perplexity.

As you have seen the lower the perplexity score, the better the model performs. For instance, LLaMA 7b with YaRN and 128k extrapolation performs well in comparison to LLaMA 7b without YaRN.

The 7B, 13B and 70B LLaMA 2 models with improved context length are available under the LLaMA 2 license on Hugging Face. The links to each model can be found in this repository.

Next, let's move to different attention variations and how they improve the performance.

Attention mechanism variations

Today's transformers are more efficient than previous models, like LSTMs. That is, the first transformer model achieved a similar high BLEU score as LSTMs, which needed to be trained for months, after only 3.5 days of training. However, transformers can still be considered memory-hungry, since the time and memory complexity of self-attention grows quadratically with the sequence length. This section explores various improvements on the attention mechanisms used in high-performing SOTA LLMs including:

- Cross attention [6]
- Multi-query attention (MQA) [7]
- Grouped-query attention (GQA) [8]
- FlashAttention [9]
- FlashAttention-2 [10]

It is common for models to combine different attention variations: for instance, Falcon uses multi-query attention and FlashAttention.

Cross-Attention

In *cross-attention* the inputs from two different sequences are combined. Usually this means that the queries come from the decoder and the keys and the values come from the encoder. So, in essence, cross-attention enables the interaction between a set of embeddings. This is important for applications where you want to attend to a source sequence while generating a target sequence, such as translation or question-answering tasks. Let me explain the concept further with code.

```
def CrossAttention(x_1, x_2, W_query, W_key, W_value)

    scaling_factor = W_query.shape[1]**0.5

    Q = torch.einsum('bd,dk->bk', x_1, W_query)
    K = torch.einsum('bd,dk->bk', x_2, W_key)
    V = torch.einsum('bd,dv->bv', x_2, W_value)

    attn_scores = torch.einsum('bk,mk->bm', Q, K)
    attn_weights = F.softmax(attn_scores / scaling_fa

    Y = torch.einsum('bm,mv->bv', attn_weights, V)

    return Y
```

In this code, you can see that the input for Q comes from x_1 and for K and V from x_2, demonstrating the information flow between sequences. Using multiple information sources the LLM gets a more sophisticated understanding and better generation results.

Multi-Query Attention

Multi-query attention (MQA) uses only a single key-value head, whereas multi-head attention (MHA) uses h-number of heads for query, key and value heads, respectively. Thus, MQA significantly speeds up the decoder's inference time. Figure 1-5 compares the two.

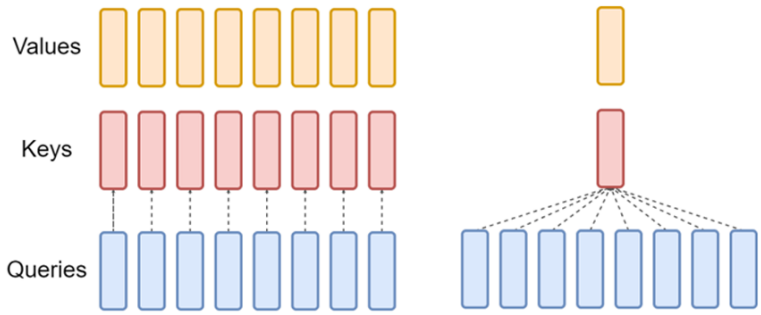

Figure 1-5. Comparison of multi-head attention (left) and multi-query attention (right). Where multi-head attention has h number of query, key and value heads, multi-query shares a single key and value head across all query heads.

To make this difference more tangible, read the following code which computes MHA. Note that there is a letter h for each Q, K and V to represent the head's dimension.

```
import torch
import torch.nn.functional as F

def MultiheadAttention(x, M, W_query, W_key, W_value,

    scaling_factor = W_key.shape[1]**0.5

    ❶
    Q = torch.einsum('d,hdk->hk', x, W_query)
    K = torch.einsum('md,hdk->hmk', M, W_key)
    V = torch.einsum('md,hdv->hmv', M, W_value)
```

```
    ❷
    attn_scores = torch.einsum('hk,hmk->hm', Q, K) / s
    ❸
    attn_weights = F.softmax(attn_scores, dim=-1)

    ❹
    o = torch.einsum('hm,hmv->hv', attn_weights, V)
    y = torch.einsum('hv,hdv->d', o, P_o)

    return y
```

❶ Weight matrices

❷ Compute attribution matrices using the scaling factor for scaled dot-product attention

❸ Apply softmax to attention scores

❹ Compute final attention weights (context vectors)

With MQA, the letter h is omitted from the K and V matrices:

```
def MultiqueryAttention(X, M, mask, W_query, W_key, W_

    scaling_factor = W_key.shape[1]**0.5

    Q = torch.einsum('bnd,hdk->bhnk', X, W_query)
    K = torch.einsum('bmd,dk->bmk', M, W_key)
    V = torch.einsum('bmd,dv->bmv', M, W_value)

    attn_scores = torch.einsum('bhnk,bmk->bhnm', Q, K
    attn_weights = F.softmax(attn_scores + mask, dim=

    O = torch.einsum('bhnm,bmv->bhnv', attn_weights, V
```

```
        Y = torch.einsum('bhnv,hdv->bnd', O, P_o)

        return Y
```

These two examples make it clear that MQA is identical to MHA, except that in MQA the different Q heads share a single set of keys and values. This modification speeds up computation in the decoder, but can lead to loss of quality, though still more performant than MHA. GQA was developed to address this.

Grouped-Query Attention

Grouped-query attention (GQA) organizes query heads into G number of groups, with each group sharing one key and one value head. Figure 1-6 compares multi-head attention (left) and grouped-query attention (right).

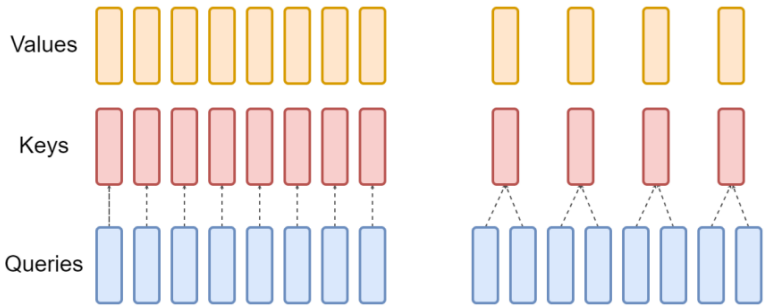

Figure 1-6. Comparison of multi-head attention (left) and grouped-query attention (right). Where multi-head attention has h number of query, key and value heads grouped-query attention instead shares one key and value head for each group of query heads, interpolating between multi-head and multi-query attention.

Comparing MHA to GQA, you can see that GQA consolidates multiple key and value heads into a single key and value head, effectively reducing the key-value size. This means significantly less data to load into memory during computation, decreasing the required bandwidth and capacity by a factor of h. The following code illustrates this setup:

```
def GroupedQueryAttention(Q, K, V, num_heads, group_s

    batch_size, seq_len, embed_dim = Q.shape
    scaling_factor = (embed_dim // num_heads) ** 0.5

    Q = rearrange(Q, 'b s (h d) -> (b h) s d', h=num_
    K = rearrange(K, 'b s (h d) -> (b h) s d', h=num_
    V = rearrange(V, 'b s (h d) -> (b h) s d', h=num_

    attn_scores = torch.einsum('bid,bjd->bij', Q, K)
    attn_weights = F.softmax(attn_scores, dim=-1)
    attn_output = torch.einsum('bij,bjd->bid', attn_w

    Y = rearrange(attn_output, '(b h) s d -> b s (h d

    return Y
```

GQA is specifically beneficial for larger models as they usually expand the number of heads. That said, employing GQA substantially reduces both memory bandwidth and capacity, while maintaining performance as models scale up.

Thus, memory bandwidth overhead from attention has less impact in larger models. This is because the key-value cache size increases linearly with the model dimension, whereas the model's floating-point operations per second (FLOPs) and parameters increase quadratically with the model dimension.

Even given these improvements, there is still room to optimize how attention leverages the GPU memory. This is where FlashAttention and FlashAttention-2 come in.

FlashAttention

FlashAttention uses *tiling* to rearrange how attention calculations are performed. By doing so, it avoids creating a $N \ x \ N$ attention matrix. Tiling involves transferring chunks of input data from GPU high bandwidth memory (HBM) and GPU on-chip SRAM (speedy cache). FlashAttention iterates over sections of the K and V matrices, transferring them to the "speedy cache". Within each section, it cycles through portions of the Q matrix, moving them to SRAM, then saves the results of the attention process back to the HBM (illustrated in Figure 1-7).

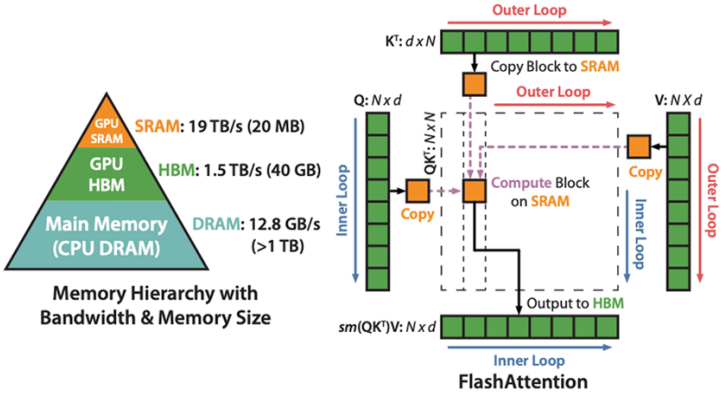

Figure 1-7. FlashAttention uses tiling to eliminate the large $N \ x \ N$ attention matrix. It works by cycling through segments of the K and V matrices in its outer loop (indicated with red arrows), loading these segments into the fast on-chip SRAM. For each segment, FlashAttention also processes chunks of the Q matrix (denoted by blue arrows), loading them into SRAM, then saving the attention output back to HBM.

This enhances computation speed while decreasing memory consumption from quadratic to linear, relative to the sequence length. FlashAttention avoids saving the large intermediate attention matrices in HBM, minimizing memory operations and doubling or even quadrupling processing speed. In addition, FlashAttention enables longer context windows in transformers, resulting in better perplexity scores and therefore higher quality models.

This is impressive, but there is still room for more improvement. The number of non-matmul FLOPs operations can be further reduced, as you will see in the next

section.

FlashAttention-2

I mentioned earlier that it's difficult to increase context window size in transformers. The core attention layer's runtime and memory demands grow quadratically with the input sequence length. RoPE, PI, and YaRN help improve efficiency and lower the perplexity, as you saw.

FlashAttention-2 reduces the amount of non-*matmul* FLOPs while not changing the output. Although these non-matrix multiplication FLOPs amount to only a minor portion of the total FLOPs, they are slower to execute. GPUs have specialized units that make matrix multiplication operations run up to 16 times faster than non-matrix multiplication operations. Therefore, minimizing non-matrix multiplication FLOPs and maximizing the time spent on matrix multiplication FLOPs is crucial for speeding up your computations.

FlashAttention-2 achieves this by optimizing GPU resource utilization. It minimizes shared memory access through parallel computation across different thread blocks and work partitioning among warps within a single thread block. A *warp* is a group of threads that execute computations. These adjustments contribute to a 2-3 times speedup.

This approach involves inverting the *loop hierarchy*, focusing first on row segments in the outer loop and column segments in the inner loop. This reverses the original method presented in the FlashAttention and introduces parallel processing along the sequence length dimension. Figure 1-8 illustrates this.

Figure 1-8. In the forward pass (left), the tasks (thread blocks) are distributed in parallel, with each task handling a segment of rows from the attention matrix. In the backward pass (right), each task is responsible for a segment of columns within the attention matrix.

Figure 1-9 compares the work partitioning between different warps in the forward pass in FlashAttention and FlashAttention-2. Efficiently dividing work among warps can significantly impact the performance of parallel computing tasks, including those in deep learning models like transformers.

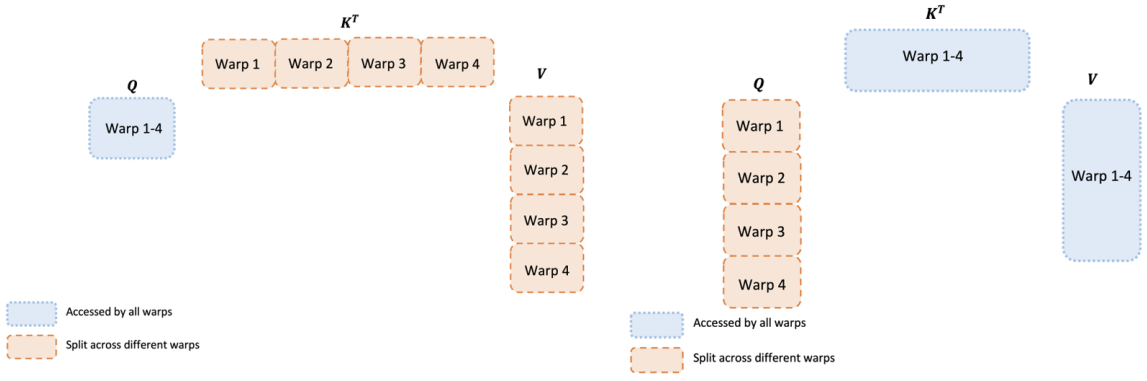

Figure 1-9. Comparison of work partitioning between different warps in the forward pass in FlashAttention (left) and FlashAttention-2 (right).

Conclusion

Let me conclude this chapter with good news. I'm sure you're keen to try these amazing improvements on the original transformer architecture, especially the last one. FlashAttention-2 is already supported in Hugging Face for many

architectures, including Falcon, LLaMA, and Mistral. Even better, I will cover some common tasks for which you can use these models in the next chapter.

Ashish Vaswani et al. “Attention Is All You Need.”, *https://arxiv.org/abs/1706.03762* (2017).

Jianlin Su et al. “RoFormer: Enhanced Transformer with Rotary Position Embedding”, *https://arxiv.org/abs/2104.09864* (2021).

Shouyuan Chen et al. “Extending Context Window of Large Language Models via Positional Interpolation”, *https://arxiv.org/abs/2306.15595* (2023).

Bowen Peng et al. “YaRN: Efficient Context Window Extension of Large Language Models”, *https://arxiv.org/abs/2309.00071* (2023).

Arthur Jacot et al. “Neural Tangent Kernel: Convergence and Generalization in Neural Networks”, *https://arxiv.org/abs/1806.07572* (2018).

Mozhdeh Gheini et al. “Cross-Attention is All You Need: Adapting Pretrained Transformers for Machine Translation”, *https://arxiv.org/abs/2104.08771* (2021).

Noam Shazeer “Fast Transformer Decoding: One Write-Head is All You Need”, *https://arxiv.org/abs/1911.02150* (2019).

Joshua Ainslie et al. “GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints”, *https://arxiv.org/abs/2305.13245* (2023).

Tri Dao et al. “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness”, *https://arxiv.org/abs/2205.14135* (2022).

Tri Dao “FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning”, *https://tridao.me/publications/flash2/flash2.pdf* (2023).

Chapter 2. Leveraging and Refining LLMs

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo is avaialble at *https://github.com/Nicolepcx/transformers-the-definitive-guide*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

In the previous chapter, you learned about breakthroughs like RoPE and FlashAttention. These innovations are at your fingertips to use with models such as LLaMA 3, and Mistral. This is exactly what you do in this chapter. You will use these models to perform common NLP tasks such as *named entity recognition* (NER), text classification, summarization and text generation. To spice things up, you will use also *LlamaIndex* to perform some of these tasks. LlamaIndex is a framework that not only simplifies but also accelerates the development of LLM applications, bringing a new level of innovation to our projects.

As you progress through the chapter, I'll explain various prompting strategies to enhance the response of your LLM model, such as chain-of-thought and three-of-thought prompting. Lastly, I show you how to refine your language model to better align with your preferences. I'll do this using techniques such as *reinforcement*

learning from human feedback (RLHF), direct *preference optimization* (DPO), and *Odds Ratio Preference Optimization* (ORPO).

Using instruction models for common NLP tasks

In this section, you will use instruction models to perform common NLP tasks. These billion-parameter models show already impressive results without further training. That said, I will not dive into fine-tuning LLMs on a dataset for specific downstream tasks. If you're interested in learning how to fine-tune models for these tasks, I recommend the following books: Natural Language Processing with Transformers or Transformers in Action, which have dedicated chapters for each task. As for specific, niche tasks or when working with highly domain-specific data, fine-tuning a model on a targeted dataset might yield better results.

LLMS VS. FOUNDATION MODELS VS. INSTRUCTION MODELS

The terms LLMs, instruction models (often referred to as chat models), and foundation models are frequently used interchangeably, yet they have distinct meanings. Instruction models are a specialized type of LLM designed to follow prompts for specific tasks. These models are particularly trained to process prompts that are structured either as direct questions or actionable tasks. On the other hand, the term foundation model denotes a broader class of generative artificial intelligence models trained on vast datasets to handle a wide array of tasks to generate outputs. These models are not limited to language alone but also include capabilities to process visual and auditory data. Therefore, while all instruction models are LLMs and fit within the broader category of foundation models, not all foundation models are restricted to language tasks.

Accessing the LLaMA 3 instruct model

If you don’t have access to the LLaMA 3 model family yet, you have to request access on Hugging Face as shown in Figure 2-1. For this, open the model page for LLaMA 3 8B Instruct and login to your Hugging Face account.

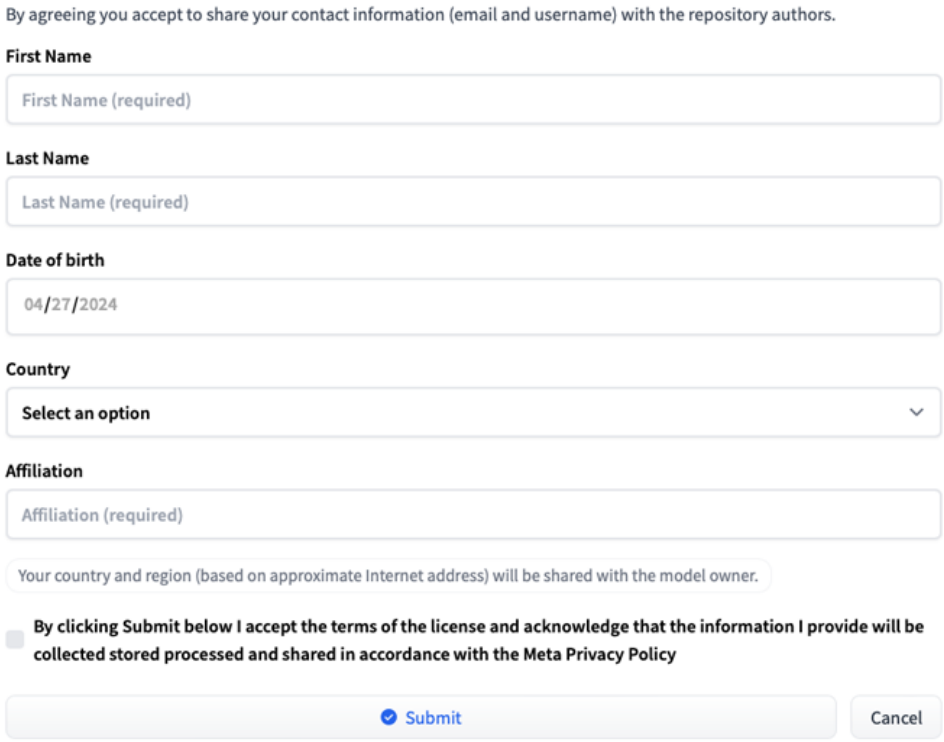

Figure 2-1. Request access for LLaMA 3 instruct model.

Next, go to the settings page of your Hugging Face account and click on “Access Tokens” as shown in Figure 2-2:

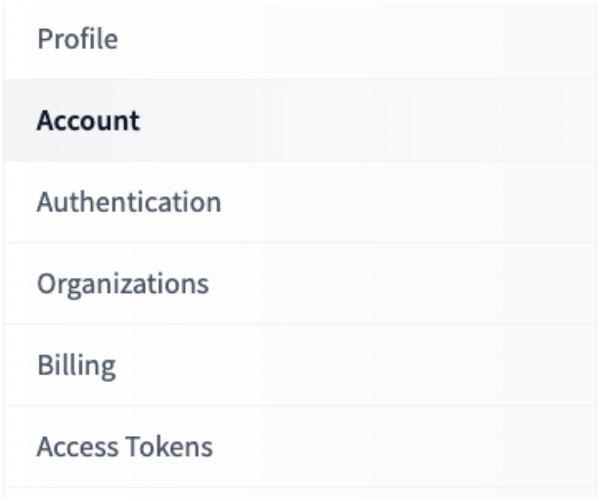

Figure 2-2. Create access token on Hugging Face.

Here you can create a “read” token for accessing the model. To load the model in your notebook follow the code in Example 2-1:

Example 2-1. Load LLaMA 3 model

```
hf_token = "your_access_token" ❶
HfFolder.save_token(hf_token)


model_id = "meta-llama/Meta-Llama-3-8B-Instruct"
tokenizer = AutoTokenizer.from_pretrained(model_id, toke

bnb_config = BitsAndBytesConfig( ❷
    load_in_4bit=True,
    load_in_8bit=False,
    bnb_4bit_use_double_quant=False,
    bnb_4bit_quant_type="fp4",
    bnb_4bit_compute_dtype=torch.bfloat16 ❸
)

model = AutoModelForCausalLM.from_pretrained( ❹
    model_id,
    device_map='auto', ❺
    quantization_config = bnb_config,
```

```
        token=True,
    )
```

❶ Hugging Face access token

❷ BitsAndBytes configuration

❸ Bfloat16 is a 16-bit floating point format designed to offer near 32-bit precision in the most significant bits.

❹ Load model with given specifications

❺ Automatically maps to GPU, if available

NOTE

I will only show important steps, the accompanying notebooks in the books repo will cover all details.

This code loads LLaMA 3 in 4 bit *quantization* with *BitsAndBytes*. BitsAndBytes is a library which enables you to load LLMs with k-bit quantization for PyTorch. Quantization reduces the precision in a model by converting, for instance, 32 bit floating-point to 16 bit floating-point, this decreases model size and speeds up computation. To use the model for text generation follow Example 2-2:

Example 2-2. Simple text generation

```
prompt = """Nicole lives in Zurich, Switzerland and is a
          Her personal interests include"""

model_input = tokenizer(prompt, return_tensors="pt").to(
```

```
model.eval()
with torch.no_grad(): ❶
    output_ids = model.generate(model_input["input_ids"]
    response = tokenizer.decode(output_ids, skip_special
    print(response)
```

❶ Generate text without tracking, calculating, or updating the gradients.

This gives you the following output:

Example 2-3.

```
Nicole lives in Zurich, Switzerland and is a Data Scient
Her personal interests include 3D printing, DIY electror
```

The code from listing Example 2-2 is okay if you simply want to try out the model. However, if you want to use a chat model for your application you have to specify prompt templates.

Text Generation

To use LLaMA 3 for text generation, you have to set up the LLaMA 3 template, as shown in Example 2-4:

Example 2-4. Chat template for LLaMA 3 using Hugging Face chat template

```
messages = [
    {
        "role": "system",
        "content": "Tell me five facts about the statue
    },
    {"role": "user", "content": "You are an expert in Hi
 ]
```

```
chat_template = tokenizer.apply_chat_template(messages,
```

This will create the correct chat template for LLaMA 3:

Example 2-5.

```
<|begin_of_text|><|start_header_id|>system<|end_header_i

liberty.<|eot_id|><|start_header_id|>user<|end_header_id

<|eot_id|>
```

To now generate text with your messages, you have to apply your chat template, and generate the output as shown in Example 2-6

Example 2-6. Generate text with chat template

```
prompt = tokenizer.apply_chat_template(
messages,
add_generation_prompt=True,
return_tensors="pt"
).to(model.device)

terminators = [ ❶
tokenizer.eos_token_id,
tokenizer.convert_tokens_to_ids("<|eot_id|>")
]

outputs = model.generate(
prompt,
max_new_tokens=512,
eos_token_id=terminators,
do_sample=True,
temperature=0.2, ❷
```

```
    top_p=0.8, ❸
    )
    response = outputs[0][prompt.shape[-1]:]
    print(tokenizer.decode(response, skip_special_tokens=Tru
```

❶ Model specific end of sentence tokens.

❷ The parameter called “temperature” is used to control the randomness of the sampling process. A high temperature leads to more randomness, while a low temperature makes the output closer to greedy search.

❸ Top-p sampling, chooses the smallest set of top words such that their total probability exceeds a certain threshold (p), and then samples the next word from this set.

LLaMA 3 generates the following output:

Example 2-7.

```
The Statue of Liberty! A iconic symbol of freedom and de
are five fascinating facts about this beloved landmark:

1. *Gift from France*: The Statue of Liberty was a gift
people of France to the people of the United States. It
by French sculptor Frédéric Auguste Bartholdi and built
Eiffel. The statue was dedicated on October 28, 1886.
2. *Colossal Size*: The Statue of Liberty stands 151 fee
including the pedestal. The statue itself is 111 feet ta
one of the largest statues in the world at the time of i
construction.
3. *Broken Chains*: The statue depicts Libertas, the Rom
of freedom, holding a torch above her head and broken ch
feet. The broken chains represent the abolition of slave
idea of freedom from oppression.
4. *Seven Points of the Crown*: The Statue of Liberty's
```

```
made up of seven points, representing the seven seas and
The crown is also adorned with a tablet inscribed with t
American Declaration of Independence, July 4, 1776.
5. *Restoration and Reopening*: After being closed to th
over 9 years, the Statue of Liberty underwent a major re
project from 1984 to 1986. The statue reopened on July 4
new pedestal museum and a renovated interior.

These are just a few of the many fascinating facts about
Liberty. This iconic landmark continues to inspire and e
visitors from around the world, serving as a powerful sy
freedom, democracy, and the American dream.
```

Text Summarization

In this section, you will use Mistral and LLaMA 3 together with LamaIndex to perform text summarization. To use LLaMA 3, for text summarization, you now have to use the LlamaIndex specific prompt template for chat models:

Example 2-8.

```
system_prompt = """<|begin_of_text|><|start_header_id|>s
You are a helpful, respectful, and honest assistant.
<|eot_id|><|start_header_id|>user<|end_header_id|>
"""

# This will wrap the default prompts that are internal t
query_wrapper_prompt = PromptTemplate(
"{query_str}<|eot_id|><|start_header_id|>assistant<|end_
```

Additionally, you need to change, how you load and create your model:

Example 2-9.

```
model_id = "meta-llama/Meta-Llama-3-8B-Instruct"
tokenizer = AutoTokenizer.from_pretrained(model_id, toke

stopping_ids = [
tokenizer.eos_token_id,
tokenizer.convert_tokens_to_ids("<|eot_id|>"),
]

bnb_config = BitsAndBytesConfig(
load_in_4bit=True,
load_in_8bit=False, # You can optionally load it in 8bit
bnb_4bit_use_double_quant=False,
bnb_4bit_quant_type="fp4",
bnb_4bit_compute_dtype=torch.bfloat16
)

llm = HuggingFaceLLM(
model_name=model_id,
max_new_tokens=512,
model_kwargs={
"token": hf_token,
"quantization_config": bnb_config
},
generate_kwargs={
"do_sample": True,
"temperature": 0.6,
"top_p": 0.9,
},
system_prompt=system_prompt,
query_wrapper_prompt=query_wrapper_prompt,
tokenizer_name=model_id,
tokenizer_kwargs={"token": hf_token},
stopping_ids=stopping_ids,
device_map="auto",
)
```

Now, to summarize text from a document, you can use the *simple directory reader* from LlamaIndex as shown here:

Example 2-10. Access documents with LlamaIndex

```
documents = SimpleDirectoryReader('./data').load_data()
```

This will load your documents from the data directory. To query the model with the summarization task, you need to create embeddings first:

Example 2-11. Setup embeddings model

```
embed_model = HuggingFaceEmbedding(
model_name="hkunlp/instructor-large"
) ❶
Settings.llm = llm
Settings.embed_model = embed_model ❷
Settings.num_output = 256
Settings.context_window = 4096
Settings.chunk_size = 512
Settings.chunk_overlap = 64

vector_index = VectorStoreIndex.from_documents(documents
```

❶ Create an embedding model

❷ Apply embedding model

With the embeddings in place, you can ask LLaMA 3 to provide you a summary of the text. How you can do this, is shown in Example 2-12

Example 2-12. Use LlamaIndex for text summarization

```
print(
vector_index.as_query_engine(
llm=llm,
```

```
).query("Provide a short summary of the patient record c
)
```

I instructed the model to summarize a patient record, this is the resulting model output:

Example 2-13.

```
Here is a short summary of the patient record of Pamela

Pamela Rogers, a 56-year-old woman, was admitted to the
department with a chief complaint of chest pains. She re
experiencing dull and aching chest pain, which radiates
accompanied by shortness of breath. The pain occurs appr
a week, usually after working in her garden or engaging
activity. The patient has a history of hypertension, dia
ago, but has never been told she has heart problems. She
smoke or have diabetes. Her physical examination reveale
signs, no abnormal findings on her skin, HEENT, and neur
examination, and a grade 2/6 systolic decrescendo murmur
right intercostal space.
```

However, the patient record is not that long and LLaMA 3 models only have a context window of 8k tokens. So, if you want to summarize longer documents, you need a model with a larger context window, or you would have to extend the context window of LLaMA 3. However, to simplify things here, change the model to be Mistral 7B version 0.2, which has a context window of 32k tokens. For that, you only have to change the model id:

Example 2-14.

```
model_id = "mistralai/Mistral-7B-Instruct-v0.2"
```

Now you use Mistral 7B version 0.2 for your summarization task. In addition, to use it correctly, you have to modify the prompt template as follows:

Example 2-15.

```
system_prompt = """<|prompter|>\n"""

# This will wrap the default prompts that are internal t
query_wrapper_prompt = PromptTemplate("{query_str}\n<|as
```

Now you can acces a shared txt file via my GoogleDrive account to summarize the first two chapters from another book I have written. This document has 16917 tokens, which are clearly more tokens than the 8k tokens the LLaMA 3 model can handle. Accessing the file and getting the embeddings stays all the same as shown in Example 2-10 and Example 2-11, you just have to adjust the query:

Example 2-16.

```
query_engine = vector_index.as_query_engine()
print(query_engine.query(
"""Could you summarize the given context? Return your re
the key points of the text and does not miss anything im
```

Mistral generates the following response:

Example 2-17.

```
The text discusses the Transformer model, its components
advantages over traditional sequence-to-sequence models.
that Transformers employ attention and multi-head attent
to navigate through a sentence, focusing on key parts. T
mechanism allows the model to focus on important informa
```

```
multi-head attention helps identify numerous links betwe
sequence. The text also notes that Transformers have rev
NLP by making it possible to train models in a matter of
outperforming state-of-the-art networks. However, the te
acknowledges the limitations of extremely large language
including decreased effectiveness in certain specialized
computational demands.
```

Named Entity Recognition

In this section, you'll use LLaMA 3 and LlamaINdex to perform Named Entity Recognition (NER), as shown in Example 2-18:

Example 2-18. NER with LLaMA 2 and LangChain

```
input_text = """
Tim Cook is CEO of Apple. Apple is an American multinati
corporation and technology company headquartered in Cupe
California, in Silicon Valley.
"""

text = f"Find all entities in the following \n\n {input_

messages = [
ChatMessage(role="system", content="You are and entities
ChatMessage(role="user", content=text),
]
response = llm.chat(messages)
```

This prints out the following text:

Example 2-19.

```
Here are the entities found in the text:

Tim Cook (Person)
Apple (Organization)
California (Location)
Cupertino (Location)
Silicon Valley (Location)
```

Text Classification

In this section, you use LangChain's news loader (which helps you download web articles) to perform sentiment classification. LangChain is a similar framework like LlamaIndex. All you have to do, is to provide the links you want to analyze as shows in Example 2-20:

Example 2-20. Simple text generation

```
news = NewsURLLoader(urls=[
    'https://finance.yahoo.com/news/nvidias-
     gtc-was-ceo-jensen-huangs-big-moment-173007915.html
    'https://finance.yahoo.com/news/stock-market-today-
     sp-500-hits-fresh-record-as-all-eyes-turn-to-fed-de
    'https://finance.yahoo.com/video/doj-accuses-apple-i
    'https://finance.yahoo.com/m/4205eaa9-f620-3a0b-
     a81a-0e82c7c9fd0b/magnificent-seven-stocks-to.html'
]).load()
```

You could, of course, also use a scraper to provide the links to the articles you want to analyze. To organize the gathered data for further processing, you can store the article headline and content in a DataFrame:

Example 2-21.

```
news_content = [c.page_content for c in news]
news_headline = [c.metadata["title"] for c in news]

news_df = pd.DataFrame({"News_Headline": news_headline})
news_df["News_Text"] =  news_content
```

This will result in the following DataFrame:

	News_Headline	News_Text
0	Nvidia's GTC was CEO Jensen Huang's big moment	Nvidia CEO Jensen Huang might as well have pit...
1	Stock market today: S&P 500 hits fresh record ...	Wall Street closed the trading session on a hi...
2	DOJ accuses Apple of iPhone monopoly in landma...	The US Justice Department and 16 state and dis...
3	Magnificent Seven Stocks To Buy And Watch: Nvi...	Reuters\n\n"New evidence has emerged in recent...

Now you can use the prompt template and utility function from Example 2-22 to perform sentiment classification on the news data.

Example 2-22. Sentiment classification with LangChain

```
def get_sentiment(text):
# Formatted instruction string with dynamic text input
instruction = f"""Classify the text into neutral, negati
Answer only with the word. Text: {text}
"""

    messages = [
        ChatMessage(role="system", content="You are an e
        ChatMessage(role="user", content=instruction),
    ]

    response = llm.chat(messages)
    sentiment = extract_sentiment(response)

    return sentiment
```

```
news_df['Sentiment'] = news_df['News_Headline'].apply(ge
❶
```

❶ Apply function over each row in the DataFrame.

This results in the following DataFrame:

	News_Headline	News_Text	Sentiment
0	Nvidia's GTC was CEO Jensen Huang's big moment	Nvidia CEO Jensen Huang might as well have pit...	Positive
1	Stock market today: S&P 500 hits fresh record ...	Wall Street closed the trading session on a hi...	Positive
2	DOJ accuses Apple of iPhone monopoly in landma...	The US Justice Department and 16 state and dis...	Negative
3	Magnificent Seven Stocks To Buy And Watch: Nvi...	Reuters\n\nThe U.S. has filed a lawsuit agains...	Positive

The great thing about using a chat model to perform sentiment classification, is, that you can ask the model to reason. So, if you want the model to explain why it classified the text as *neutral*, *negative* or *positive* you simply alter your instruction:

Example 2-23.

```
instruction = f"""Classify the text into neutral, negati
a reasoning why you classified it as neutral, negative,
For instance if you find words like 'big moment', this i
Text: {sent_headline}
Reason: """

messages = [
ChatMessage(role="system", content="You are an expert in
ChatMessage(role="user", content=instruction),
]

response = llm.chat(messages)
```

For the first news headline about Nvidia’s CTC event it responds as follows:

Example 2-24.

```
I would classify the text as positive.
Reasoning: The phrase "big moment" is a positive phrase,
occasion was significant and notable. Additionally, the
Huang's name suggests that the text is highlighting his
which is often associated with positive connotations. Th
specifically mentions Nvidia's GTC (GPU Technology Confe
the positive tone, as it implies a sense of accomplishme
industry.
```

Getting the model’s explanations for its classifications offers valuable insights beyond the mere classification labels. This approach, applicable across various domains, enables a deeper understanding of the model’s reasoning processes. Whether for enhancing content moderation, improving customer feedback analysis, or refining market sentiment evaluations.

Now that you’ve seen how to use instruction models with basic prompting techniques, the next section will dive deeper into various enhanced prompting strategies. These refined approaches optimize how you instruct your model and are designed to yield more precise outputs.

Advanced prompting techniques

In all previous examples you prompted the chat model to answer your question without providing an example or guidance, this is known as *zero-shot prompting*. The next stage is to provide the model with some examples; this is called *few-shot*

prompting. For instance, for the sentiment classification task from the previous section you would alter the prompt as follows:

```
instruction = f"""Classify the text into neutral, negati
with only one word: Positive, Negative, or Neutral.
Text: {text}
Sentiment:
Examples: The movie was great! Sentiment: Positive;
Steve didn't like the cake. Sentiment: Negative
Reason: """

messages = [
ChatMessage(role="system", content="You are an expert ir
ChatMessage(role="user", content=instruction),
]
```

In this example of few-shot prompting, you provide the model two examples for classifying sentiment. This approach works well for a wide range of tasks, but if you want to empower the LLM to undertake reasoning tasks, you have to employ a more advanced prompting technique.

In contrast to zero-shot and few-shot prompting, where you ask the models to directly produce the final answer, you can encourage the LLM to generate intermediate reasoning steps before providing the final answer to a problem. Doing so gives you the ability to break down complex problems into manageable, intermediate steps. This section will cover various such techniques, such as chain-of-thought and three-of-thought prompting.

Chain-of-thought prompting

Chain-of-thought prompting (CoT) [1] enhances the reasoning capabilities of LLMs, especially in multi-step reasoning tasks. Inspired by the human thought process, this

technique first solves intermediate steps before getting to the final answer. Figure 2-3 shows examples of how you alter the prompt for CoT.

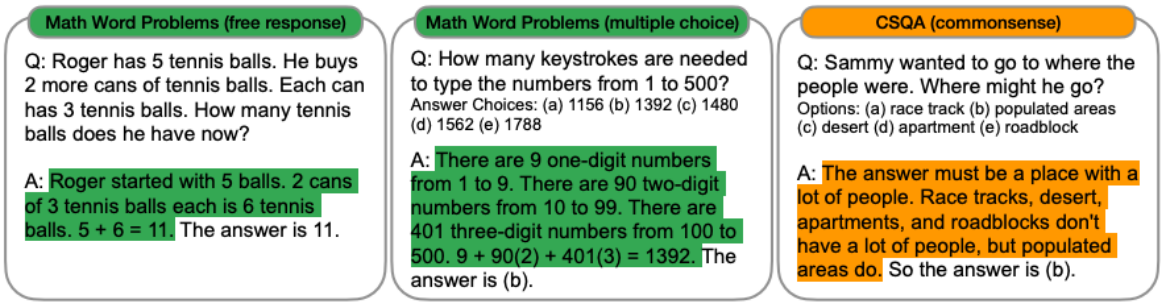

Figure 2-3. Examples for arithmetic and commonsense reasoning benchmarks. The highlighted statements show how you can phrase an explicit CoT example for the model for each shown task.

To use this with the template you applied in the previous section, you could do the following:

Example 2-25.

```
question = """Q 1: Roger has 5 tennis balls. He buys 2 m
Each can has 3 tennis balls. How many tennis balls does
A 1: The answer is 11.
Q 2: The cafeteria had 23 apples.
If they used 20 to make lunch and bought 6 more, how mar
A:"""

messages = [
ChatMessage(role="system", content="You are a mathematic
ChatMessage(role="user", content=question),
]
response = llm.chat(messages)
print(response)
```

This will result in the following model output:

Example 2-26.

```
Let's solve the problem!

The cafeteria had 23 apples initially. They used 20 to m
they have 23 - 20 = 3 apples left.

Then, they bought 6 more apples. To find the total numbe
they have now, we add the remaining apples to the new or

3 (remaining apples) + 6 (new apples) = 9 apples

So, the cafeteria has 9 apples now.
```

CoT prompting can be enhanced through a technique known as *self-consistency*. Figure 2-4 demonstrates this process.

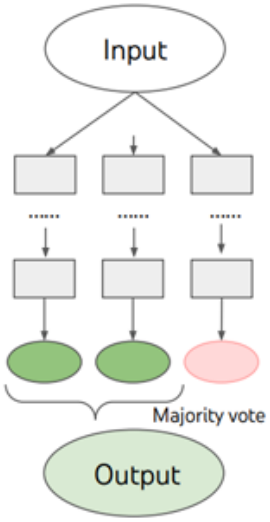

Figure 2-4. Self consistency with chain of thought

Essentially, this technique involves presenting the same prompt to the model multiple times and then determining the final result based on the majority response.

Zero-shot CoT prompting

Another way to enhance CoT is *zero-shot CoT* prompting. This allows LLMs to generate more accurate answers to questions by appending the words:

```
Let’s think step by step.
```

to the end of a question. This straightforward prompt enables the LLM to produce a chain of thought, allowing it to generate a more precise response. Example 2-27 shows how you would question the model.

Example 2-27. Simple text generation

```
question = """Q: Roger has 5 tennis balls. He buys 2 mor
Each can has 3 tennis balls. How many tennis balls does
A: The answer is 11.

Q: The cafeteria had 23 apples.
If they used 20 to make lunch and bought 6 more, how mar
A: Let's think step by step."""

messages = [
ChatMessage(role="system", content="You are a mathematic
ChatMessage(role="user", content=question),
]
response = llm.chat(messages)
```

The model answers as follows:

Example 2-28.

```
Let's break it down!

The cafeteria started with 23 apples. They used 20 for l
have:

23 - 20 = 3 apples left
```

```
Then, they bought 6 more apples. So, the new total is:

3 (remaining apples) + 6 (new apples) = 9 apples

The cafeteria now has 9 apples.
```

However, sometimes even that might not be enough to get the correct answer.

Tree of thought prompting

If you have more complex tasks where initial decisions play a pivotal role or a more exploratory and strategic approach is needed, you can use *tree of thoughts prompting* (ToT) [2]. An overview how this technique works is shown in Figure 2-5.

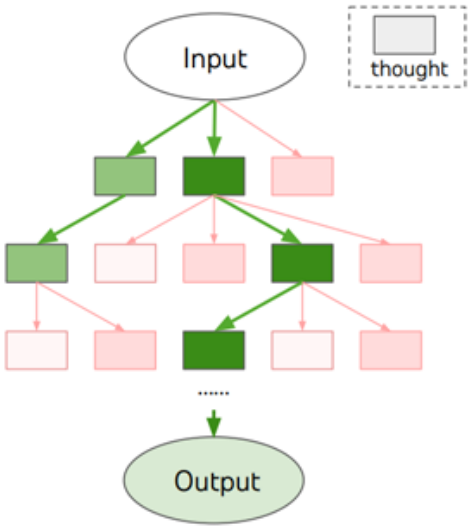

Figure 2-5. Tree of thought prompting

ToT uses the LLMs ability to evaluate coherent language sequences in combination with search algorithms for data structures such as depth-first search (DFS) or breadth-first search (BFS).

DFS VS. BFS

DFS and BFS are widely used algorithms in computer science for searching within tree or graph data structures. Starting from the root, BFS examines all nodes present at the current level of the tree prior to progressing to the next level, whereas DFS dives as deep as possible down each branch before moving to another.

ToT expands upon the CoT prompting method by allowing the LLM to investigate coherent text segments that act as an intermediary step in problem-solving. Figure 2-6 shows an example for a math problem.

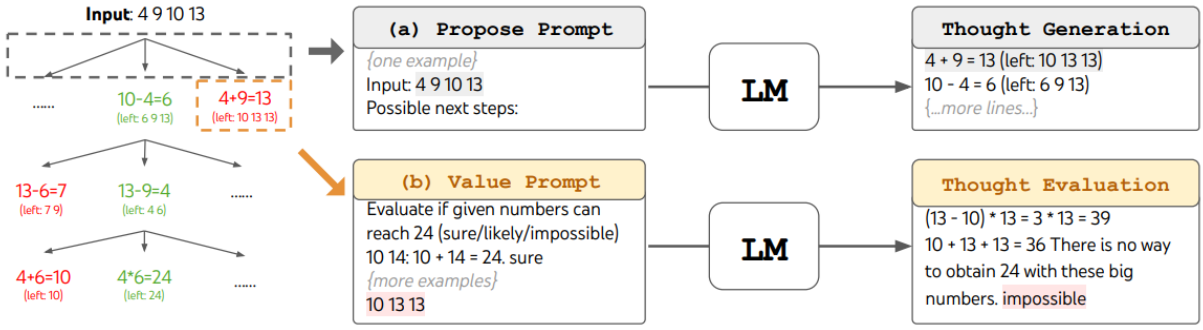

Figure 2-6. Thought generation (a) and evaluation of the thought (b).

Example 2-29 demonstrates how you can use ToT solving *Game of 24* tasks with ChatGPT. Game of 24 is a math reasoning contest where participants aim to manipulate four integer numbers using basic arithmetic operations (addition, subtraction, multiplication, division) to achieve a result of 24.

Example 2-29. ToT prompting

```
os.environ['OPENAI_API_KEY'] = 'your-api-key'

args = argparse.Namespace(backend='gpt-4', temperature=0
naive_run=False, prompt_sample=None, method_generate='pr
method_evaluate='value', method_select='greedy', n_gener
```

```
  n_evaluate_sample=3, n_select_sample=5)

  task = Game24Task()

  ys, infos = solve(args, task, 999)
  print(ys[0])
```

This will produce the following output:

Example 2-30.

```
  ['13 - 9 = 4 (left: 4 4 10)\n10 - 4 = 6 (left: 4 6)\n4 *
  \nAnswer: 4 * (10 - (13 - 9)) = 24\n', '10 - 4 = 6 (left
  (left: 4 6)\n4 * 6 = 24 (left: 24)\nAnswer: (10 - 4) * (
  '13 / 4 = 3.25 (left: 3.25 9 10)\n10 / 3.25 = 3.08 (appr
  \n9 - 3.08 = 5.92 (left: 5.92)\n5.92 * 2 = 11.84 (left:
  '13 / 4 = 3.25 (left: 3.25 9 10)\n10 / 3.25 = 3.08 (appr
  (left: 3.08 9)\n9 - 3.08 = 5.92 (left: 5.92)\n5.92 + 2 =
  (left: 7.92 8 8 14)\n', '13 / 4 = 3.25 (left: 3.25 9 10)
  n10 / 3.25 = 3.08 (approx) (left: 3.08 9)\n9 - 3.08 = 5.
  (left: 5.92)\n14 - 5.92 = 8.08 (left: 2 8 8 8.08)\n']
  13 - 9 = 4 (left: 4 4 10) \\
  10 - 4 = 6 (left: 4 6) \\
  4 * 6 = 24 (left: 24) \\
  Answer: 4 * (10 - (13 - 9)) = 24
```

ToT is a good choice for tasks (such as solving mathematical problems) that involve analytical reasoning, or for coding.

Thread of thought prompting

Thread of thought prompting (ThoT) [3] draws inspiration from human cognitive processes once more. ThoT mimics the flow of thoughts that individuals preserve as they navigate through extensive information. This process enables the focused

retrieval of important information while disregarding irrelevant details. Figure 2-7 compares the input and prompting flow of CoT and ThoT.

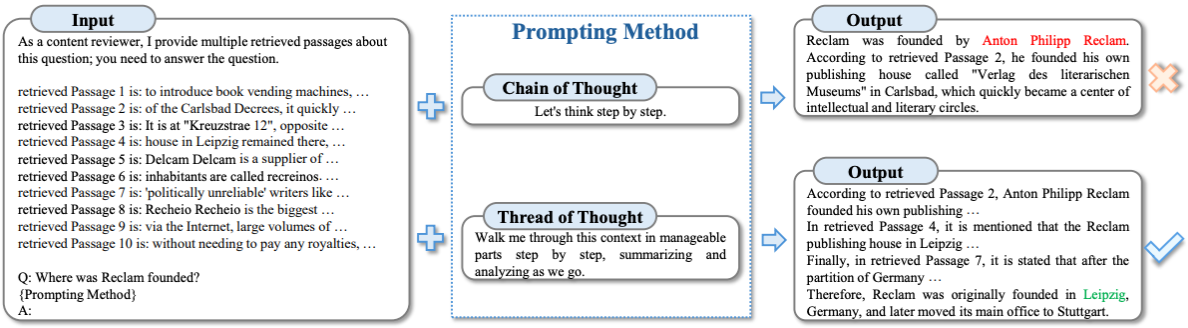

Figure 2-7. Thread of thought prompting helps LLMs to navigate chaotic information.

ThoT methodical examines and summarizes segments of information to enhance the LLMs understanding across several paragraphs. This helps the LLM to navigate information which appears relevant but is actually irrelevant. To prompt the model with ThoT you begin your prompt with:

```
Walk me through this context in manageable parts step by step, summarizing and analyzing as we go.
```

After this prompt, you let the model refine its conclusion by combining the initial prompted text with the model's response and a conclusion marker such as:

```
Therefore, the answer:
```

This simple approach offers a straightforward way to enhance the reasoning capabilities of LLMs for large and chaotic texts.

Aligning your model with your preferences

The prompting methods from the previous section are known as *in context learning*. This means you don't update the weights of your model. However, in some cases, you have to update your model weights to align your model with your preferences. *Supervised fine-tuning* (SFT), *reinforcement learning from human feedback* (RLHF) [4], and *direct preference optimization* (DPO) [5] are common methods for preference alignment. Moreover, I will cover a monolithic preference alignment method, called *Odds Ratio Preference Optimization* (ORPO) [6]. ORPO introduces a penalty during SFT and therefore eliminates the need for an additional preference alignment phase.

Supervised Fine-Tuning

SFT is a foundational step in model alignment, primarily focused on adapting the unsupervised, pre-trained language models to specific tasks or preferences through exposure to a labeled dataset. In general, training a chat model follows this structure:

- Pre-training: The model learns (unsupervised) general language patterns and capabilities from a vast and diverse dataset.
- SFT: Tailors the model to perform well on specific types of tasks it will face after deployment, based on a curated dataset that exemplifies these tasks.
- Alignment phase: Adjusts the model's output to align closely with human preferences, ensuring that the model not only knows how to respond but does so in a way that meets user expectations and ethical guidelines.

The goal of SFT is to refine the general-purpose model, that understands language, but isn't yet specialized for specific tasks, like engaging in dialogue as a chatbot or instruct model. SFT adjusts the model's parameters (weights) to reduce errors specifically for the target task, making it more effective at handling types of input it will encounter in its designated role. This is achieved by adjusting the internal

parameters to reduce the loss function, effectively making the predictions closely align with the ground truth provided by the training data.

During SFT, adjustments to the model's weights are governed by *gradient descent* algorithms, focusing on minimizing the divergence between the predicted outputs and the actual data:

$$\theta_{new} = \theta_{old} - \eta \cdot \nabla_{\theta} \mathscr{L}_{SFT}$$

Here, η represents the learning rate, a hyperparameter that determines the step size during the weight update phase. The gradient of the loss function with respect to the model parameters $\nabla_{\theta} \mathscr{L}_{SFT}$, guids the update direction.

Using SFT for your LLM is extremely easy, you can just leverage Hugging Face's trl library for that. This library is specifically tailored to support SFT and alignment tasks. This class uses system instructions, questions, and answers, forming the basis of the prompt structure. You can use the class as shown here:

Example 2-31.

```
training_args = TrainingArguments(
                per_device_train_batch_size = 2,
                gradient_accumulation_steps = 4,
                warmup_steps = 100,
                max_steps = 600,
                learning_rate = 2e-4,
                fp16 = not torch.cuda.is_bf16_supported(
                bf16 = torch.cuda.is_bf16_supported(),
                optim = "adamw_8bit",
                weight_decay = 0.01,
)

sft_trainer = SFTTrainer(
    model = model,
    train_dataset = formatted_dataset,
```

```
        dataset_text_field = "text",
        max_seq_length = 4096,
        args = training_args
    )
```

By refining the model in this manner, SFT aims to produce a model that not only performs well on general tasks but excels at domain-specific tasks.

Reinforcement learning from human feedback

Reinforcement learning from human feedback usually starts with fine-tuning a pre-trained language model with SFT on a high-quality dataset tailored to specific tasks, such as dialogues. In the second phase, the SFT model, π^{SFT}, is prompted with prompts x to produce answer pairs $(y_1, y_2) \sim \pi^{\text{SFT}}(y \mid x)$. Human labelers review the generated answers and label them according to their preferences, represented as $y_w \succ y_l \mid x$ out of (y_1, y_2). Here y_w and y_l represents the preferred and dispreferred prompt, respectively.

In addition to your pre-trained model, you need a *reward model*, $r^*(x, y)$. The idea of the reward model is, that it can generate a *scalar reward*, $r_\varphi(x, y)$, which is important for *reinforcement learning* (RL) to work. The reward model is usually a language model fine-tuned on prompt-generation pairs. These prompt-pairs are based on the initial language model's output and human annotators which rank the generated text. The *policy*, which is represented by a language model, takes a prompt as input and generates a text sequence or, more technically, *probability distributions* over possible text outputs.

The *reward function* integrates several components into a single RLHF process, by receiving a prompt x from the dataset D of human comparisons. The fine-tuned policy

generates text y. This output, concatenated with the original prompt, is evaluated by the preference model, yielding a *scalar preferability score* r_φ. Figure 2-8 shows this interaction in a high level overview.

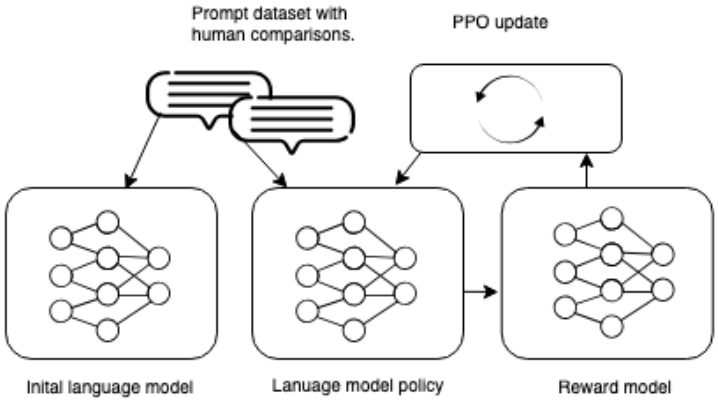

Figure 2-8. Simplified overview of a language model traing with RLHF.

Additionally, the RL policy's per-token probability distributions calculates a penalty for deviations. This penalty is often defined as a scaled *Kullback–Leibler* (KL) *divergence*, r_{KL}, which helps to prevent the RL policy to diverge too much from the pre-trained model's behaviour. This divergence control is critical for maintaining the coherence of generated text; without it, optimization might prioritize gibberish text, just to maximize the reward. Specifically, the loss function can be expressed as binary classification problem with negative log-likelihood:

$$\mathscr{L}_R(r_\varphi, \mathscr{D}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathscr{D}}[\log\, \sigma(r_\varphi(x, y_w) - r_\varphi(x, y_l))]$$

Here, $r_\varphi\,(x, y)$ represents the scalar output of the reward model for prompt x and generated text y. D is the dataset of human comparisons used to train the reward model. y_w is the preferred text of pair (y_1,y_2), wheras y_l is the less preferred text by human labelers and σ the *logistic* (sigmoid) function.

The KL divergence, $\mathbb{D}_{\mathrm{KL}}$, quantifies the difference between the RL policy's output distributions and those of the SFT model, π^{SFT}. The overall reward, communicated to the RL update mechanism, is formulated as $r = r_{\varphi} - \beta\mathbb{D}_{\mathrm{KL}}$, where β is the scaling factor that balances the importance of preference alignment versus deviation penalty between the reference policy, π_{ref} and the SFT model π^{SFT}. Therefore, the optimization can be formulated as:

$$\max_{\pi_\theta} \mathbb{E}_{x\sim\mathscr{D},y\sim\pi_\theta(y|x)}[r_\varphi\left(x,y\right)] - \beta\mathbb{D}_{\mathrm{KL}}[\pi_\theta\left(y \mid x\right) \parallel \pi_{\mathrm{ref}}\left(y \mid x\right)]$$

Proximal policy optimization (PPO) [7] maximizes the reward within the current batch. PPO, an *on-policy* algorithm, restricts updates to the current batch of prompt-generation pairs. An on-policy algorithm directly learns from and optimizes the decision-making policy it is currently using to make actions. This restriction helps to control that gradient updates do not happen to quickly, and therefore, stabilizing the learning process.

Direct preference optimization

RLHF can be complex to implement due to its need for extensive hyperparameter tuning. This complexity mainly stems from the instability of the PPO algorithm used within RLHF. PPO's instability arises from its reliance on successive small updates, which must balance between making sufficient progress and maintaining policy performance. This delicate balance can lead to variance in training outcomes, as even minor changes in hyperparameters or training data can disproportionately affect the learning trajectory.

DPO presents a more streamlined approach by optimizing the LLM's policy directly, without the requirement for a reward model. DPO streamlines this process by integrating human preferences directly into the optimization mechanism, bypassing the

step of modeling preferences as an independent reward function. Figure 2-9 compares RLHF with DPO.

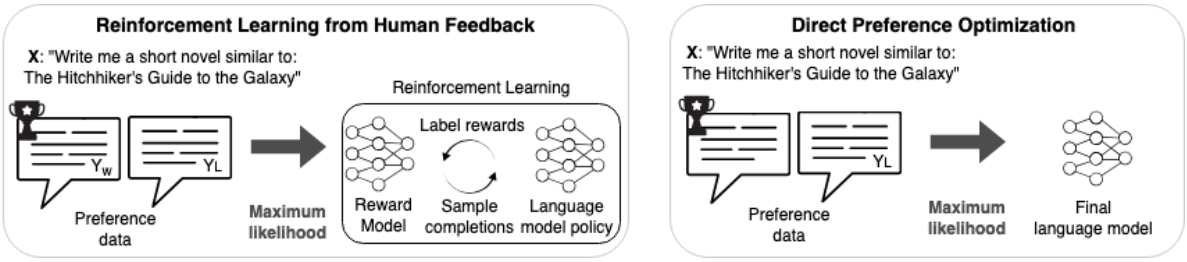

Figure 2-9. DPO directly optimizes based on chosen or rejected samples.

At the heart of DPO is its strategy of directly modifying the language model's parameters to prioritize responses that are more favorable based on direct input. This is accomplished through an optimization process constrained by the KL divergence. The KL-constrained reward maximization quantifies the disparity between the LLM's response probability distribution and a target distribution reflecting human preferences. The reward function can be express in terms of the optimal policy π_r, the reference policy π_{ref} and an unknown partial function $Z(\cdot)$:

$$r\left(x, y\right) = \beta \log \frac{\pi_r\left(y \mid x\right)}{\pi_{\text{ref}}\left(y \mid x\right)} + \beta \log Z\left(x\right)$$

This equation can be applied to your r^* and optimal model π^*, taking the difference of the rewards between the two, considering the Bradley-Terry model [8], the RLHF policy π^* can be expressed as:

$$p^*(y_1 \succ y_2 \mid x) = \frac{1}{1+ \exp\left(\beta \log \frac{\pi^*(y_2|x)}{\pi_{\text{ref}}\left(y_2|x\right)} - \beta \log \frac{\pi^*(y_1|x)}{\pi_{\text{ref}}\left(y_1|x\right)}\right)}$$

Here $p^*(y_1 \succ y_2 \mid x) = \sigma\big(r^*(x, y_1) - r^*(x, y_2)\big)$ is the derived difference from the Bradley-Terry model.

Considering this reparameterization you can express the probability of human preference as maximum likelihood objective for the now parameterized policy π_θ:

$$\mathscr{L}_{\mathrm{DPO}}(\pi_\theta; \pi_{\mathrm{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathscr{D}}\left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\mathrm{ref}}(y_w \mid x)} - \beta \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\mathrm{ref}}(y_l \mid x)}\right)\right]$$

Minimizing this divergence allows DPO to closely align the model's outputs with preferred outcomes, effectively converting the optimization task into a classification task where responses are classified as preferred or not.

To use DPO for your LLM, you can again use the trl library from Hugging Face. Here, you load your data and model as you would do with every other fine-tuning task and create training arguments as you would do with fine-tuning. The difference is that you now use the DPO Trainer class:

Example 2-32.

```
training_args = TrainingArguments(
                per_device_train_batch_size = 2,
                gradient_accumulation_steps = 4,
                warmup_ratio = 0.1,
                num_train_epochs = 2,
                learning_rate = 5e-6,
                fp16 = not torch.cuda.is_bf16_supported(
                bf16 = torch.cuda.is_bf16_supported(),
                optim = "adamw_8bit",
                weight_decay = 0.0,
                output_dir = "outputs",
)

dpo_trainer = DPOTrainer( ❶
    model=model,
    ref_model=None,
    args=training_args,
    beta=0.1,
    train_dataset=transformed_datasets["train_prefs"],
```

```
        eval_dataset=transformed_datasets["test_prefs"],
        tokenizer=tokenizer,
        max_length=1024,
        max_prompt_length=512,
    )
```

❶ Use DPO Trainer class instead of usual Trainer class.

And in addition to using the Trainer class, you need to have a dataset with *chosen* and *rejected* columns:

Example 2-33.

```
DatasetDict({
    train_prefs: Dataset({
        features: ['prompt', 'chosen', 'rejected'],
        num_rows: 30
    })
    test_prefs: Dataset({
        features: ['prompt', 'chosen', 'rejected'],
        num_rows: 1
    })
})
```

The DPO Trainer class additionally allows you to use two enhanced methods for more robust DPO training. The first method, referred to as ΨPO [9], acknowledges that traditional DPO is prone to overfitting coming from its specific use of KL divergence in the loss function, particularly when dealing with deterministic preferences or limited datasets. The overfitting comes from an inadequate regularization, which can cause the policy to become overly deterministic and overlook other potentially useful actions. To address this, ΨPO adds a regularization term to the DPO loss function, which results in a more robust KL-regularization against deterministic preferences

from annotators. To leverage this enhanced method, you simply integrate the loss into the DPO Trainer class:

Example 2-34. DPO with ΨPO loss

```
dpo_trainer = DPOTrainer(
    model=model,
    ref_model=None,
    args=training_args,
    beta=0.1,
    train_dataset=transformed_datasets["train_prefs"],
    eval_dataset=transformed_datasets["test_prefs"],
    tokenizer=tokenizer,
    max_length=1024,
    max_prompt_length=512,
    loss_type="ipo" ❶
)
```

❶ Use ΨPO loss for DPO.

The other method, *Kahneman-Tversky Optimization* (KTO) [10] reduces your DPO to a singleton feedback, rather than an explicit feedback. That is, you define the loss function based on individual examples such as *good* or *bad*. To use this method, just add the following loss function to your DPO Trainer:

Example 2-35.

```
    loss_type="kto_pair"
```

However, RLHF and DPO still depend on a preference-aligned phase, after the SFT phase. In the next section, I will introduce a technique, which eliminates this additional preference alignment phase.

Odds Ration Performance Optimization

SFT is particularly important in RLHF for maintaining policy continuity, as it is used to stabilize the update of active policies, ensuring that newer policies do not deviate significantly from older, established ones. Beyond its use in RL, SFT is vital for achieving convergence to desired results in other alignment methods, such as DPO. The role of SFT is to adapt pre-trained language models to specific domains by enhancing the probabilities of relevant tokens. However, this comes with an undesirable side effect. With SFT, you inadvertently increase the likelihood of generating non-preferred model outputs, as illustrated in Figure 2-10:

Figure 2-10. Log probabilities during fine-tuning indicate that both chosen and rejected responses are likely, even though only chosen ones guide the training.

To address this issue, a monolithic preference alignment method named Odd Ration Performance Optimization (ORPO) can be used. ORPO dynamically penalizes the

non-preferred response for each query without the need for creating sets of rejected tokens. The objective function is as follows:

$$\mathscr{L}_{ORPO} = \mathbb{E}_{(x,y_w,y_l)}[\mathscr{L}_{SFT} + \lambda \cdot \mathscr{L}_{OR}]$$

Here $\mathscr{L}_{SFT}$ follows the causal language modeling negative log-likelihood loss function to maximize the likelihood to generate reference tokens. $\mathscr{L}_{OR}$ maximizes the odds ratio to generate favored responses y_w over unfavored y_l:

$$\mathscr{L}_{OR} = -\log \sigma\left(\log \frac{\text{odds}_\theta\left(y_w \mid x\right)}{\text{odds}_\theta\left(y_l \mid x\right)}\right)$$

By weighting $\mathscr{L}_{SFT}$ and $\mathscr{L}_{OR}$ with λ, the pre-trained language model is customized to focus on a particular subset of the targeted domain while discouraging outputs from the sets of unfavorable responses. To use ORPO with Hugging Face follow the code from Example 2-36:

Example 2-36. ORPO Trainer setup

```
training_args = ORPOConfig(
                output_dir='./results',
                per_device_train_batch_size = 2,
                gradient_accumulation_steps = 4,
                num_train_epochs = 2,
                warmup_ratio = 0.1,
                remove_unused_columns=False,
                learning_rate=9e-1,
                evaluation_strategy="steps",
                beta=0.1, ❶
            )


orpo_trainer = ORPOTrainer( ❷
                model=model,
                args=training_args,
                tokenizer=tokenizer,
```

```
                train_dataset=transformed_datasets["trai
                eval_dataset=transformed_datasets["test_
            )
```

❶ Hyperparamter needed for ORPO

❷ Initialize ORPO Trainer

Figure 2-11 shows the AlpacaEval 2.0 score comparison of ORPO, DPO and RLHF. AlpacaEval evaluates models based on their responses to single-turn prompts.

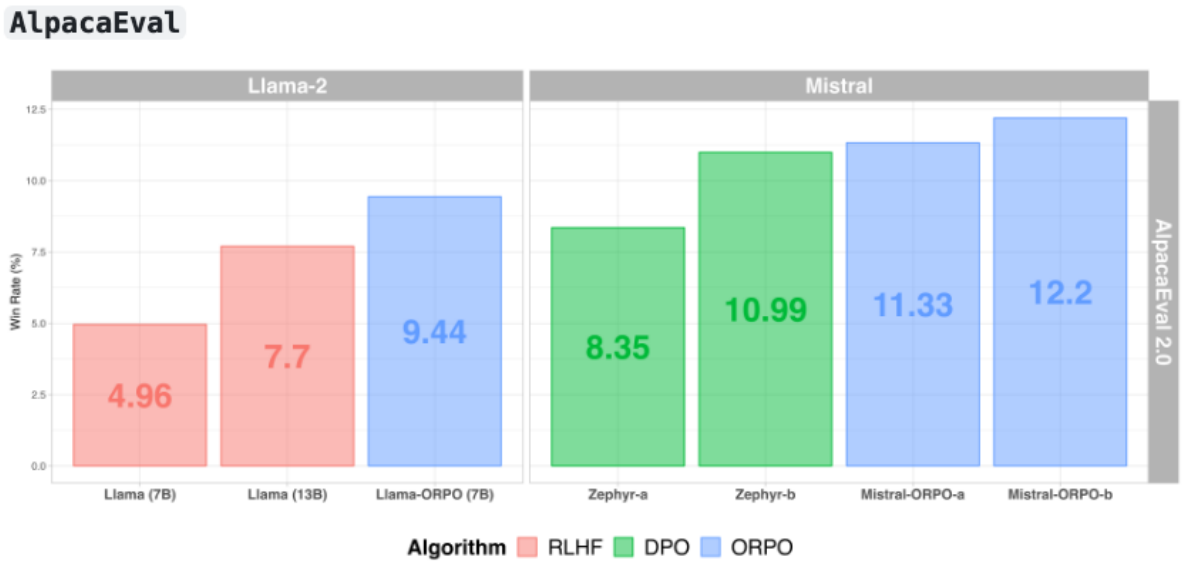

Figure 2-11. Different alignment methods evaluated with AlpacaEval 2.0.

While ORPO is computationally efficient, RLHF or DPO might still yield better results, therefore, I recommend you evaluate the different alignment methods based on your use case and available dataset.

Conclusion

In this chapter you learned how straightforward it is to use an instruct model for tasks such as text summarization using LangChain. This establishes the base for you, to use these LLMs for your applications.

In addition, you've seen different prompting techniques, commonly known as in-context learning, to guide your language models outputs based on the problem at hand, such as a mathematical problem.

In the last section you gained an understanding, how you can align your LLM with human preferences. There are different methods to chose from such as RLHF, DPO and, ORPO. Each comes with its own strengths and weaknesses, while RLHF is still the most chosen method in the industry. In the next chapter, you will take a closer look into reinforcement learning by using it with reinforcement learning transformers.

Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models", *https://arxiv.org/abs/2201.11903* (2023).

Shunyu Yao et al. "Tree of Thoughts: Deliberate Problem Solving with Large Language Models.", *https://arxiv.org/abs/2305.10601* (2023).

Yucheng Zhou et al. "Thread of Thought Unraveling Chaotic Contexts", *https://arxiv.org/abs/2311.08734* (2023).

Long Ouyang et al. "Training language models to follow instructions with human feedback", *https://arxiv.org/abs/2203.02155* (2022).

Rafael Rafailov et al. "Direct Preference Optimization: Your Language Model is Secretly a Reward Model.", *https://arxiv.org/abs/2305.18290* (2023).

Jiwoo Hong et al. “ORPO: Monolithic Preference Optimization without Reference Model”, *https://arxiv.org/abs/2403.07691* (2024).

John Schulman et al. “Proximal Policy Optimization Algorithms”, *https://arxiv.org/abs/1707.06347* (2017).

Bradley, Ralph Allan, and Milton E. Terry. “Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons.” Biometrika 39, no. 3/4 (1952): 324–45. *https://doi.org/10.2307/2334029*.

Mohammad Gheshlaghi Azar et al. “A General Theoretical Paradigm to Understand Learning from Human Preferences”, *https://arxiv.org/abs/2310.12036* (2023).

Kawin Ethayarajh et al. “KTO: Model Alignment as Prospect Theoretic Optimization”, *https://arxiv.org/abs/2402.01306* (2024).

About the Author

Nicole Koenigstein is a distinguished data scientist and quantitative researcher, currently working as Chief Data Scientist and Head of AI and Quantitative Research for Wyden Capital, an algorithmic-based investment company, and as Head of AI and Quantitative Research at quantmate, an innovative FinTech startup focused on alternative data in predictive modeling. Alongside her roles in these organizations, she serves as an AI consultant across a broad spectrum of AI applications, ranging from natural language processing, time series data, image classification and segmentation, to anomaly detection, and beyond. She leads workshops and guides companies from the conceptual stages of AI implementation through to final deployment.