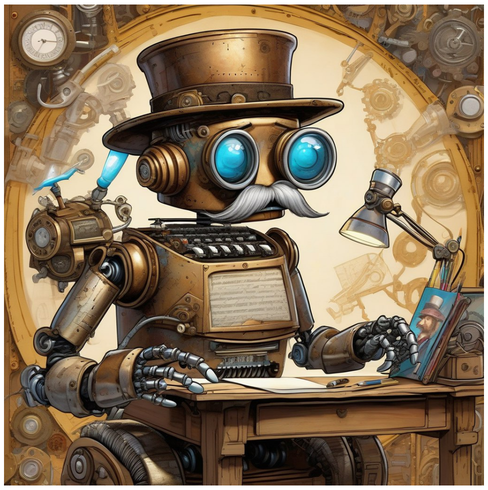
Generative AI

Max Brede Alwin Klick

2025-03-18



Introduction

Introduction

This script serves as an introduction to **Generative AI** and was developed for the elective module "Generative AI," offered to master's students of the "Data Science" program at the University of Applied Sciences Kiel. Built using `quarto`, this resource is designed to provide an accessible overview of key topics and applications in this rapidly evolving field.

While not an exhaustive guide to Generative AI, the script highlights foundational concepts, modern applications, and practical techniques that empower students to engage with and explore the possibilities of these transformative technologies.

Contents and learning objectives

Contents listed in the module database entry:

Open Source Language Models

- Overview of model lists
- Ollama
- Generation of synthetic text as training sets

Agent and LLM-Pipeline Systems

- Llamaindex, LangChain & smolagents
- Function calling
- LLM-based pipelines

Embeddings and Vector Stores

- Semantic Search
- Retrieval-augmented generation
- Recommendations

AI Image Generators

- Generative Adversarial Networks (GANs)
- Variational Autoencoders / Diffusion Models
- Generative approaches for image dataset augmentation

Fine-Tuning of LLMs and Diffusion Models

- Examples: LoRA, QLoRA, MoRA

Learning objectives listed in the module database entry:

Students

- know the fundamentals of generative AI systems.
- know various modern applications of generative AI systems.
- know the theoretical foundations and practical applications of generative AI systems.

Students

- are able to explain and apply various open-source language models.
- are able to implement and utilize agent systems and their functionalities.
- are able to understand and use embeddings and vector stores for semantic search and recommendations.
- are able to explain and practically apply different methods for image generation.
- are able to fine-tune large language models (LLMs) and diffusion models for specific tasks.

Students

- are able to successfully organize teamwork for generative AI projects.
- are able to report and present team solutions for practical project tasks.
- are able to interpret and communicate the approaches in technical and functional terms.

Students

- are able to work professionally in the field of generative AI systems.
- are able to give and accept professional feedback to different topics of generative AI systems.
- are able to select relevant scientific literature about generative AI systems.

Schedule:

Tab 1: Course schedule

Number:	Date:	Title:	Topics:
1	13.05.	Getting started with (L)LMs	Language Model Basics Choosing open source models Basics of using open source models (Huggingface, Ollama, LLM-Studio, Llama.cpp, ...)
2	14.05	Prompting	Prompting strategies Generation of synthetic texts
3	20.05	Function Calling	Code generation and function calling
4	21.05	Agent basics	Fundamentals of agents and chain-of-thought prompting Examples of agent-frameworks (Llamaindex, LangChain & smolagents)

Number:	Date:	Title:	Topics:
5	27.05	Embedding-based retrieval systems	Semantic embeddings and vector stores
			Retrieval augmented and interleaved generation
6	28.05	LLM-pipelines	
7	3.06.	AI image generation I	AI image generator basics
			Diffusion Models and Variational Autoencoders Multimodal models
8	4.06.	AI image generation II	Generative Adversarial Networks (GANs)
			(Generative) approaches for image dataset augmentation

Number:	Date:	Title:	Topics:
9	10.06	AI image generation III	Using Open Source AI image generation models AI image generators in agent systems
10	11.06	Finetuning Basics Rank adaptation	Basics of Finetuning strategies Fundamentals of High and Low-Rank Adaptation of Language and Diffusion Models (Q)LoRA fine-tuning using Unsloth
11	17.06	Alignment	Central principles of Model-Alignment Reinforcement Learning from Human Feedback (RLHF)
12	18.06	Project presentations	

Number:	Date:	Title:	Topics:
	27.06	Project submission on moodle	

Organizational Details

Planned Class Structure

Each class meeting will follow this structure:

1. **Instructional Session**: We'll introduce new concepts and techniques.
2. **Practice Exercise**: Students will apply these concepts through an exercise.
3. **Project Worktime**: Students will work on their team projects.

Students will be divided into teams of three at the start of the course, with projects culminating in a final presentation to the class. The project grade will count towards your final course grade.

Project Details

Projects should allow students to apply what they've learned throughout the course. They must implement an LLM-based system that includes at least two of the following features:

- Retrieval Augmentation/RAG (i.e., the system should query documents or other content in an index for its answers and reference the sources of its generation)
- Data Analysis (i.e., the system should "talk" to a dataset and decide on which analysis-steps are to be taken to then execute them)
- Multiple Agents (i.e., at least two agents should work in tandem, for example in a generator-reviewer arrangement)
- Fine-tuning on (Synthetic) Data (i.e., a small LM or SDM should be finetuned on (synthetic) data to adapt it to your needs. You could as an example train a model to only answer in nouns.)

The project should also include function-calling-based interface ("a tool") to an AI image generator.

Students are free to choose their project topic, as long as it fits within the course scope and is approved by the instructor. All projects must be implemented in Python.

The active participation on the course will be taken into account before grading. This means that all tasks asking the students to upload their results to moodle should be completed. If more than one of the required tasks is missing, the student will not be graded.

The projects are to be presented in the last session of the course. The students of each group need to take part in this session. The presentation will become part of the overall grade. The presentation can but does not have to be prepared in PPT, any other mode of presentation (including a live-demo based on a nice notebook) is fine.

The project will then be graded based on these contents in addition to the following criteria:

1. The minimum of components mentioned above **have** to be used
2. The more components are used, the better the grade
3. The project-solution has to work.(Since we are talking about LLMs it does not have to generate perfect results, the pipeline has to generally work though.)
4. The students have to hand in code the instructors can run. The code has to be documented. This can be done either in sensible docstrings, appropriately commented notebooks or a report. The students can choose the mode. It is possible and recommended to create a github repository with the code and the documentation.

Example Project Ideas:

1. **LLM Tourist Guide**: Uses TA.SH data to provide travel tips and enhances them with generated images.
2. **Quarto Data Presentation Pipeline**: Builds and illustrates a Quarto presentation based on a given open dataset.
3. **Synthetic Author**: Generates commit-messages based on commit history/diff. It could also suggest GitHub issues illustrated with AI-generated images.
4. **AI Storyteller**: Creates illustrated short stories for children based on historical events.
5. **AI Webdesigner** A tool that creates and illustrates a webpage based on a Amazon product page.

Language Models

Getting started with (L)LMs

This chapter provides a brief introduction to the history and function of modern language models, focusing on their practical use in text generation tasks. It will then give a short introduction on how to utilize pretrained language models for your own applications.

Language Model Basics

Language models have diverse applications, including speech recognition, machine translation, text generation, and question answering. While we'll concentrate on **text generation** for this course, understanding the general concept of language models is crucial. Given language's inherent complexity and ambiguity, a fundamental challenge in NLP is creating structured representations that can be employed downstream. This section will first explore the evolution of these representations before introducing the transformer architecture, which forms the foundation of most modern language models.

A short history of natural language processing

just that
he said
yes said trillian firmly
i wont enjoy it said marvin

Fig 1: BOW-representation of sentences.

The **Bag Of Words (BOW)** method represents text data by counting the frequency of each word in a given document or corpus. It treats all words as independent and ignores their order, making it suitable for tasks like text classification, for which it was traditionally the gold-standard. However, BOW has limitations when it comes to capturing semantic relationships between words and gets utterly useless if confronted with words not represented in the corpus. Additionally, it does not take into account the order of words in a sentence, which can be crucial for understanding its meaning. For example, the sentences "The cat is on the mat" and "The mat is on the cat" have different meanings despite having the same set of words.

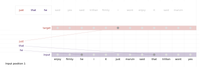


Fig 2: CBOW-representation of corpus.

The **Continuous Bag Of Words (CBOW)** method extends traditional BOW by representing words as dense vectors in a continuous space. CBOW predicts a target word based on its context, learning meaningful word representations from large amounts of text data.

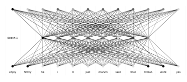


Fig 3: Shallow Model using CBOW-Method to predict missing word.

fastText (Bojanowski et al., 2017), an open-source library developed by Facebook, builds upon the CBOW method and introduces significant improvements. It incorporates subword information and employs hierarchical softmax for efficient training on large-scale datasets. Even with limited data, fastText can learn meaningful word representations. fastText and its predecessor Word2Vec are considered precursors to modern language models due to their introduction of **Embeddings**, which laid the foundation for many modern NLP methods. Figure 3 illustrates this fastText-architecture[1]

Language Model Embeddings are learned by predicting the

[1]Well, kind of. One of the major advantages of fasttext was the introduction of subword information which were left out of this illustration to save on space. This meant that uncommon words that were either absent or far and few between in the training corpus could be represented by common syllables. The display like it is here is far closer to fasttext's spiritual predecessor word2vec (Mikolov et al., 2013).

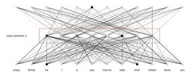


Fig 4: Model using CBOW-Method to predict missing word.

next word, or, in most cases, the next part of a word in a sequence. The utilisation of word-parts instead of whole words was another invention introduced by fastText (Bojanowski et al., 2017), that allowed the model to generalize to new, unknown words when moving to inference. These parts of words are also called **tokens**. Embeddings are the representation the model learns to map the context-tokens to a multiclass classification of the missing token in the space of all possible tokens. These embeddings capture semantic and syntactic relationships between words, enabling them to understand context effectively. Since these embeddings represent the conditional probability distribution that language models learn to comprehend natural language, they can be reused by other models for tasks such as text classification or text retrieval. But more on this later.

Still, these models did not really solve the inherent issue of the order of words in a sentence. The input of models of this generation still used a dummyfied version of the corpus to represent context, which loses a lot of information.

Traditionally, this was approached by feeding these embeddings into **Recurrent Neural Networks (RNNs)**. These models could learn to keep track of sequential dependencies in text data and improve the understanding of context. However, RNNs suffered from their architecture's inherent inability to retain information over long sequences. Simple RNN- cells[2] iterate through a sequence and use both their last output and the next sequence element as input to predict the next output. This makes it hard for them to learn long-term dependencies, since they have to compress all information into one vector (Figure 5)[3].

Long Short-Term Memory (LSTM) networks addressed

[2]And pretty much all of the more complex variants

[3]This is also (kind of) the reason for the so called **vanishing gradient** problem, where each iteration of the network is necessary for calculating the gradient in the steps before.

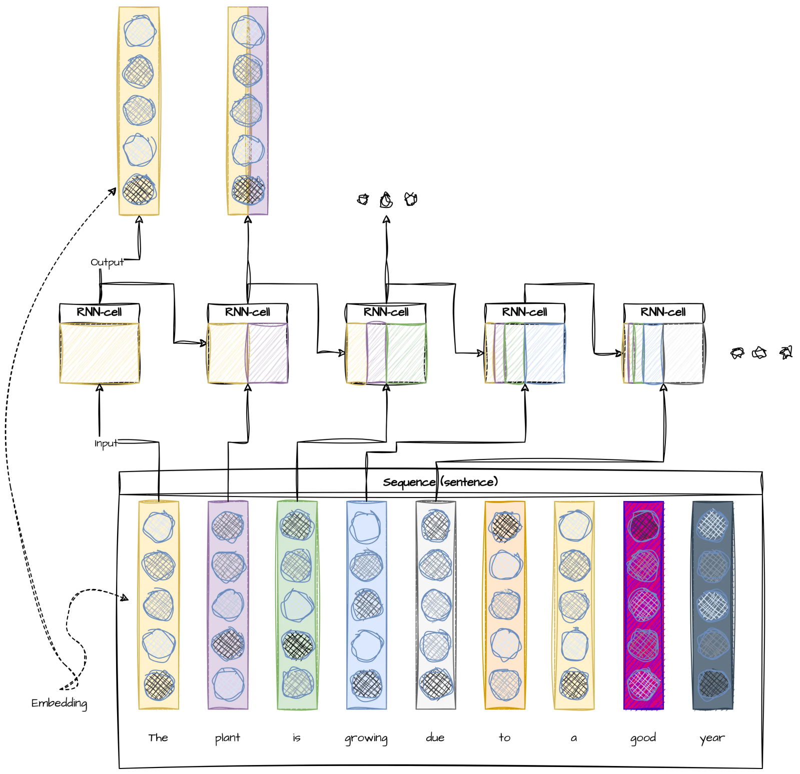


Fig 5: Illustration of a simple RNN-model, (exaggeratingly) illustrating the issue of the model “forgetting” parts of the input when processing long sequences.

this issue by introducing a mechanism called “gates” that allowed information to flow through the network selectively and more efficiently, but were, as the RNNs before, notoriuosly slow in training since only one word could be processed at a time. Additionally, a single LSTM is still only able to process the input sequence from left to right, which is not ideal for inputs that contain ambiguos words that need context after them to fully understand their meaning. Take the following part of a sentence:

> The plant was growing

The word plant get’s wildly differing meanings, depending on how the sentence continues:

> The plant was growing rapidly in the sunny corner of the garden.
>
> The plant was growing to accommodate more machinery for production.

A model that only processes the input sequence from left to right would just not be able to understand the meaning of “plant” in this context.

The ELMo model (Peters et al., 2018), which stands for Embeddings from Language Models, is an extension of LSTMs that improved contextual word representations. ELMo uses bidirectional LSTM layers to capture both past and future context, enabling it to understand the meaning of words in their surrounding context. This resulted in ELMo outperforming other models of its era on a variety of natural language processing tasks. Still as each of the LSTM-Layer were only able to process one part of the sequence at a time, it was still unfortunately slow in training and inference. Its performance additionally decreased with the length of the input sequence since LSTM-cells have a better information retention than RNNs but are still not able to keep track of dependencies over long sequences.

Attention is all you need

In their transformative paper “Attention is all you need”, Vaswani et al. (2023) described the transformer architecture.

As the paper’s title neatly suggests, the major breakthrough presented in this paper was the introduction of the so-called self-attention mechanism. This mechanism allows the model to “focus” on different parts of the input to a) determine the appropriate context for each word and b) to improve its performance on differing tasks by allowing the model to filter unnecessary information.

Self-Attention Mechanism

The self-attention mechanism relies on three components: **Query (Q)**, **Key (K)**, and **Value (V)**, inspired by concepts in information retrieval. Imagine you search for a specific term in a library (query), match it against the catalog (key), and retrieve relevant books (value).

In practice, for each word in a sentence, the model calculates:

1. **Relevance Scores**: Compare each Query vector (Q) with every Key vector (K) in the sequence using the dot product. These scores measure how much focus one word should have on another.
2. **Attention Weights**: Normalize the scores using a softmax function to ensure they sum to 1, distributing focus proportionally across all words.
3. **Weighted Sum**: Multiply each Value vector (V) by its corresponding attention weight to compute the final representation.

For example, in the sentence, “The cat sat on the mat,” the model might assign more attention to “cat” when analyzing “sat,” capturing their relationship.

Calculating Attention

For a sequence of words, the attention scores are computed as:

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

where:

- Q represents the query matrix.
- K is the key matrix.
- V is the value matrix.
- d_k is the dimensionality of the key vectors, ensuring scale invariance.

Let's first illustrate this concept with a practical example (not specifically from the context of NLP) to later circle back to its application in the transformer architecture.

We look at a retrieval task in which we query in a domain that has 5 attributes describing the items in it. The aforementioned "lookup" is then implemented by calculating the dot product between the query and the transposed keys resulting in a vector of weights for each input-aspect.

As a simplification, we assume that all aspects can be described in binary terms. A hypothetical 1x5 query matrix (Q) represents the aspects we are querying in a 5-dimensional space, while a transposed 1x5 key matrix (K) represents the aspects of the search space. The dot product between these matrices results in a scalar that reflects the alignment or similarity between the query and the key, effectively indicating how many aspects of the query align with the search space.

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \end{bmatrix}$$

If we now add a series of items we want to query for to our matrix K, the result will be a vector representing the amount of matches, each item has with our query:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 1 & 2 \end{bmatrix}$$

The result is a vector of scores that indicate the matches of the query per key. This principle does obviously also work for more than one query by adding more rows to our Query matrix Q. This does result in a matrix, in which each row indicates the amount of matching keys for each query:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 1 & 2 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 3 \\ 1 & 1 & 1 & 2 & 2 \\ 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Instead of binary indicators, the Q and K matrices in the attention mechanism are filled with floats. This does still result in the same kind of matched-key-result, although the results are now more like degrees of relevance instead of absolute matches:

$$Q \times K^T =$$

$$\begin{bmatrix} 0.57 & 1.32 & -0.86 & 1.75 & 1.66 \\ 0.82 & 0.63 & -1.46 & 0.95 & 0.08 \\ -0.15 & -0.98 & 1.74 & 0.88 & -0.17 \\ 0.24 & -0.1 & -1.53 & 1.91 & 1.76 \\ -1.67 & 1.79 & 1.96 & -1.45 & 1.62 \end{bmatrix} \times \begin{bmatrix} 1.34 & -0.21 & 1.62 & -0.44 & 0.06 \\ -1.98 & 0.74 & -0.45 & 1.24 & 0.95 \\ 0.45 & 1.63 & -1.17 & -1.97 & 1.33 \\ -0.27 & 1.89 & -1.85 & -0.26 & -0.48 \\ 0.48 & 1.88 & 0.56 & 1.55 & 1.83 \end{bmatrix} = \begin{bmatrix} -1.71 & 3.11 & 0.79 & 3.67 & 1.03 \\ -2.21 & 1.04 & 0.7 & 5.13 & 0.16 \\ -0.99 & 1.39 & -0.98 & -2.24 & 3.34 \\ -1.97 & -1.19 & 4.1 & 5.01 & 0.37 \\ 4.64 & 2.03 & -3.26 & -9.43 & -2.59 \end{bmatrix}$$

As you can already see in this small example, the values of individual cells can get relatively high compared to the rest of the matrix. As you remember - we want to use this product to rank our values. If these numbers are too large, it might lead to numerical instability or incorrect results. To address this issue, we will scale down the dot-product by dividing it with $\sqrt{d_n}$, where d_n is the dimension of the aspect space (in our case 5).

$$\frac{Q \times K^T}{\sqrt{d_n}} =$$

$$\begin{bmatrix} -0.76 & 1.39 & 0.35 & 1.64 & 0.46 \\ -0.99 & 0.46 & 0.31 & 2.29 & 0.07 \\ -0.44 & 0.62 & -0.44 & -1 & 1.49 \\ -0.88 & -0.53 & 1.83 & 2.24 & 0.17 \\ 2.07 & 0.91 & -1.46 & -4.22 & -1.16 \end{bmatrix}$$

Since we want to use this matrix for filtering our dataset, we would prefer the weights to sum up to one. To achieve that, we will apply a softmax function on each row of the matrix (remember that the rows currently represent the key-weighted aspects for each query). The resulting matrix with scaled weights for each aspect is then multiplied with the value-matrix that contains one datapoint in each row, described by 5 aspects along the columns.

$$\mathrm{softmax}(\frac{Q \times K^T}{\sqrt{d_n}}) \times V =$$

$$\begin{bmatrix} 0.04 & 0.32 & 0.11 & 0.41 & 0.13 \\ 0.03 & 0.11 & 0.1 & 0.69 & 0.08 \\ 0.08 & 0.23 & 0.08 & 0.05 & 0.56 \\ 0.02 & 0.03 & 0.35 & 0.53 & 0.07 \\ 0.72 & 0.22 & 0.02 & 0 & 0.03 \end{bmatrix} \text{x} \begin{bmatrix} -1.83 & -1.24 & 0.7 & 1 & -0.67 \\ -1.44 & -0.91 & 1.93 & 0.71 & -0.61 \\ -1.13 & 1.31 & 1.04 & -1.31 & -0.41 \\ -0.08 & 0.77 & 0.27 & -0.96 & 1.14 \\ -1.21 & -1.04 & 1.4 & 0.06 & -1.84 \end{bmatrix} = \begin{bmatrix} -1.01 & -0.17 & 1.13 & -0.03 & -0.09 \\ -1.28 & -0.54 & 1.55 & 0.33 & -0.43 \\ -1.3 & -0.53 & 0.74 & 0.26 & -0.32 \\ -1.31 & -0.1 & 1.47 & -0.05 & -0.51 \\ -0.97 & -0.59 & 1.12 & -0.17 & -1.11 \end{bmatrix}$$

The result is now an attention matrix in the sense that it tells us the importance of each value's aspect for our query. In the specific example, the forth value seems to be the most important aspect for our third query. The crucial advantage is, that all aspects of all queries can be simultaneously compared with all aspects of all values without the necessity of sequential processing.

Though this general idea of weighting aspects in the sense of self-attention[4] to process a sequence without disadvantages of the distances of the items was used before (Bahdanau, 2014), the major contribution of the paper was the complete reliance on this mechanism without the need of LSTM/RNN parts. That their suggested architecture works is in part due to the utilisation of multiple self-attention layers, each learning its own weights for Q, K and V. This allows the model to learn more complex patterns and dependencies between words in a sentence. You can think of it as allowing the model to focus on different parts of the input sequence at different stages of processing. The outputs of the multiple heads are then concatenated and linearly transformed into the final output representation using a series of fully connected feed-forward layers.

This small example is already pretty close to the general attention-mechanism described by Vaswani et al. (2023) (see

[4]*self* in the sense of the model weighting its own embeddings, queries, keys and values

also Figure 6), though the actual language model learns its own weights for Q, K and V.

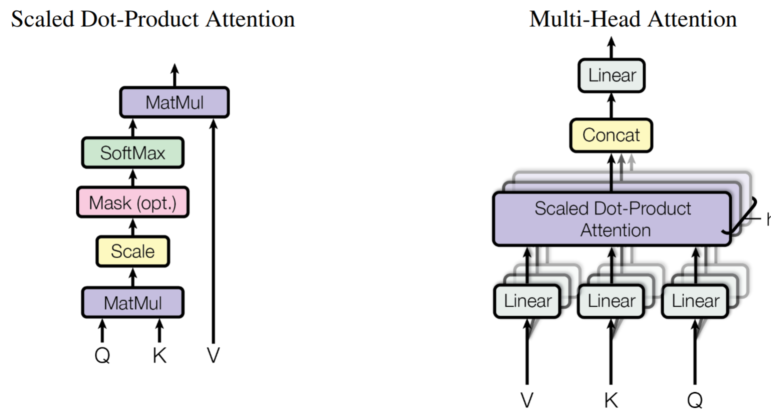


Fig 6: Multi-headed attention as depicted in Vaswani et al. (2023)

Instead of 5x5 matrices, the attenion mechanism as described in the paper implements $d_n \times d_c$[5] matrices, where d_n is the dimension of the embedding space[6] and d_c is the size of the context window. In the original paper, Vaswani et al. (2023) implement the context-window as the same size as the embedding space (i.e., $d_n = d_c$). In Figure 7 you can see a brilliant illustration of the multiheaded-attention mechanism at work.

The implementation of the multi-headed attention mechanism allowed to solve all major issues of the language modelling approaches of the previous generation[7]. It firstly allows the input

[5] $\frac{d_n}{h} \times \frac{d_c}{h}$ actually, the paper used feed-forward layers to reduce the dimensionality of each attention header to reduce the computational cost.

[6] I.e., the dimensionality used to represent each word's meaning. In the previous toy-example illustrating the concept of embeddings (Figure 4), this would be the width of the hidden layer (8). In the case of transformers, this is usually 512 or 1024. These embeddings are learned during training and are a simple transformation of the one-hot vectors returned by the models tokenizer.

[7] Well, kind of. Transformers are far superior language models due to their ability to parallely process long sequences without issues with stretched context - these advantages come at a price though. GPT-3s training is estimated to have emitted around 502 metric tons of carbon (*AIAAIC - ChatGPT training emits 502 metric tons of carbon*, n.d.). The computational cost of the architecture as described here

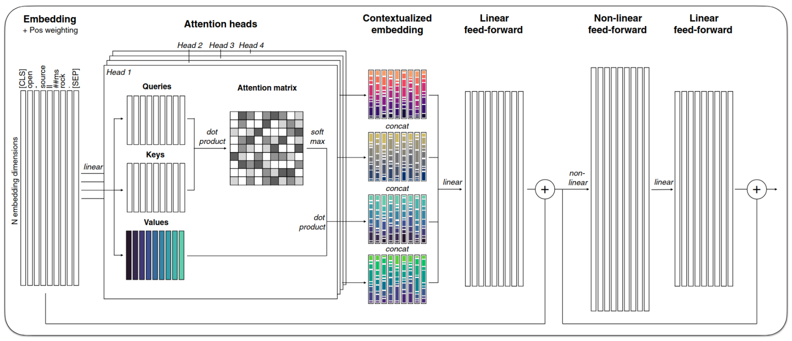


Fig 7: Illustration of the multi-headed attention mechanism. Taken from Hussain et al. (2024)

of a whole text-sequence at once, rendering the training and inference far speedier then the recursive approaches. Furthermore, the multi-head attention mechanism allows the model to focus on different parts of the input sequence simultaneously, enabling it to capture more complex relationships between words and improve its understanding of context without losing information about long-term dependencies. This mechanism also implicitly solves the bidirectionality-issue since each word can be taken into account when processsing every other word in the sequence.

The description until now omitted one final but key detail - we only spoke about the weight matrices Q, K and V. Each of these weight matrices are actually the product of the learned weights and the input vectors. In other words, each of the three matrices is calculated as follows:

$$\begin{aligned} Q &= XW_Q \\ K &= XW_k \\ V &= XW_v \end{aligned}$$

where $W_{Q,k,v}$ are the learned weight matrices and X is the input matrix. This input matrix consists of a) the learned embeddings of the tokenized input-parts and b) the added, so

does additionally scale quadratically with context window size.

called positional encoding.[8]

The positional encoding is a vector that encodes the position of each token in the input sequence. It is added to the embedding of each token to provide the model with information about the order of the tokens in the sequence. The positional encoding is calculated as follows:

$$\begin{aligned} PE_{(pos,2i)} &= sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}}) \\ PE_{(pos,2i+1)} &= cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}}) \end{aligned}$$

Where i is the dimension and pos is the position. Those 2 formulas are not the most intuitive, what they do is to add a unique offset to each embedding though, that allows the model to infer and weigh the token's positions in the matrix on it's own. Figure 8 illustrates the pattern this specific combination of sin and cos creates for each sequence-position and embedding-dimension.

These parts alltogether are all building-blocks of the basic transformer architecture. As you can see in Figure 9, all parts depicted by Vaswani et al. (2023) are parts we have discussed until now.

The Encoder half uses the embedding -> encoding -> multi-headed-attention -> feed-forward structure to create a semantic representation of the sequence. The Decoder half uses the same structure, but with an additional masked multi-head attention layer to prevent the model from looking at future tokens. This is necessary because we want to generate a sequence token by token.

Figure 10, taken from Kaplan et al. (2020), shows the test performance of Transformer models compared to LSTM-based models as a function of model size and context length. Transformers outperform LSTMs with increasing context length.

[8]While we are talking about omitted details, the whole architecture implements its layers as residual layers. This means that the output of each layer is added to the input of the layer before, before it is passed on to the next layer. But this detail is irrelevant for our understanding of the central mechanism.

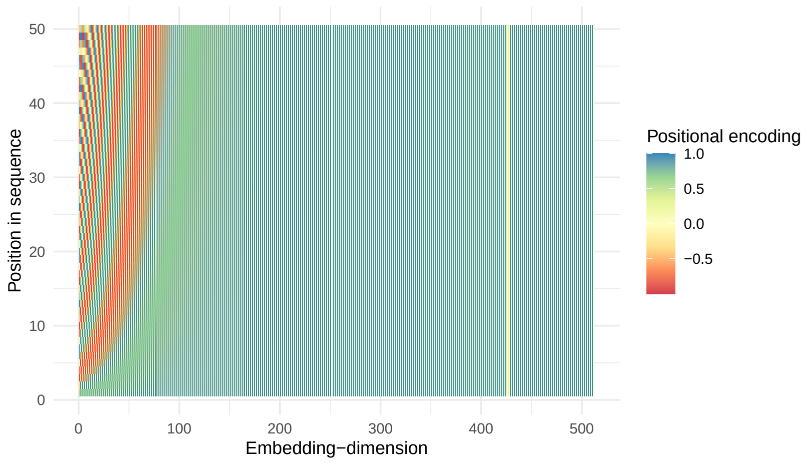


Fig 8: The positional encoding for 50 dimensions and 512 embedding-dimensions. The x-axis represents the position and the y-axis represents the dimension. The color represents the value of the encoding.

Furthermore, Kaplan et al. (2020) and Hoffmann et al. (2022) after them postulated performace power-laws (see also Figure 11) that suggest that the performance of a Transformer directly scales with the models size and data availability. Though the task of prediction of natural language poses a non-zero limit to the performance, it is suggested that this limit is not reached for any of the currently available models.[9]

The advances made through leveraging transformer-based architectures for language modelling led to a family of general-purpose language models. Unlike the approaches before, these models were not trained for a specific task but rather on a general text base with the intention of allowing specific fine-tuning to adapt to a task. Classic examples of these early general-purpose natural language generating Transformer models are the Generative Pre-trained Transformer (the predecessor of ChatGPT you all know), first described in Radford et al. (2018), and the "Bidirectional Encoder Representations from Transformers" (BERT) architecture and training procedure, described by Devlin et al. (2019).

[9]Incidentally, we might run out of data to train on before reaching that limit (Villalobos et al., 2024).

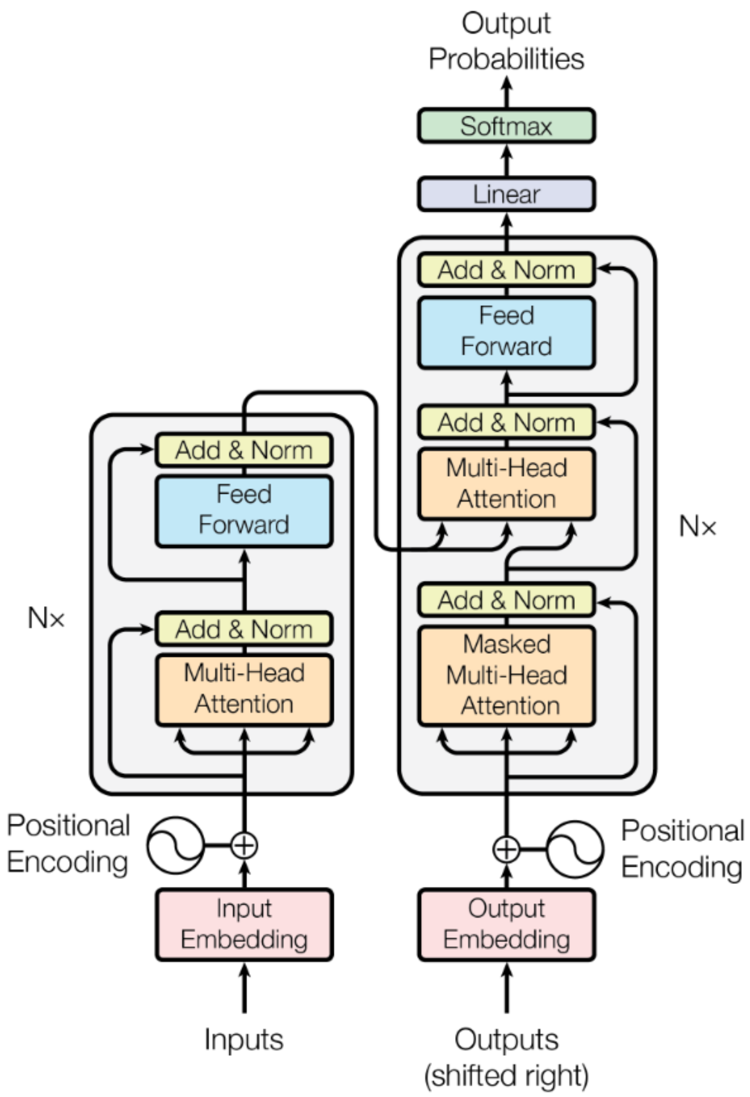


Fig 9: The transformer architecture as depicted in Vaswani et al. (2023)

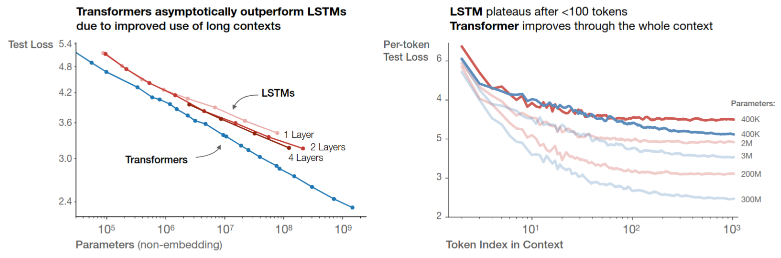


Fig 10: Comparison of Transformer- and LSTM-performance based on Model size and context length. Taken from Kaplan et al. (2020)

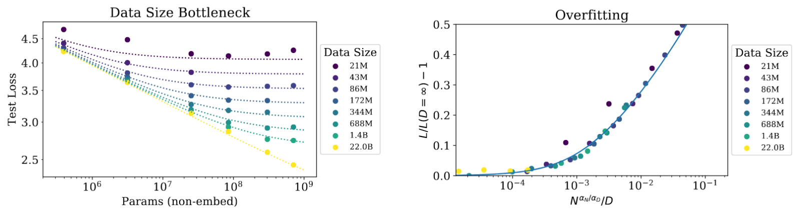


Fig 11: Performance power law for transformer models. Taken from Kaplan et al. (2020)

This general-purpose architecture is the base of modern LLMs as we know them today and most applications we will discuss in this course.

Choosing open source models

The 2023 release of ChatGPT by OpenAI has sparked a lot of interest in large language models (LLMs) and their capabilities. This has also led to an increase in the number of available open-source LLMs. The selection of a model for your application is always a trade-off between performance, size, and computational requirements.

Although Kaplan et al. (2020) showed a relationship between performance and model-size, the resources available will most probably limit you to smaller models. Additionally, a lot of tasks can be solved by smaller models if they are appropriately fine-tuned (Hsieh et al., 2023).

A good idea when choosing an open source model is to start small and test whether the performace is sufficient for your use case. If not, you can always try a larger model later on.

Additionally, it is good practice to check the license of the model you want to use. Some models are only available under a non-commercial license, which means that you cannot use them for commercial purposes.

Thirdly, you should make sure that the model you choose is appropriate for your use case. For example, if you want to use a model for text generation, you should make sure that it was trained on a dataset that is similar to the data you will be using. If you want to use a model for translation, you should make sure that it was trained on a dataset that includes the languages you are interested in. A lot of usecases do already have benchmark datasets that can be used to pit models against each other and evaluate there appropriateness for a given use case based on a few key metrics.

A good starting point for getting an overview about such metrics and benchmarks is Hugging Face. This platform has long cemented itself as the go-to place for getting access to open source models, but also provides a lot of resources for evaluating and comparing them. This page provides an overview of benchmarks, leaderboards and comparisons for a variety of tasks.

Basics of using open source models

Task

Now it is your turn! In your project-groups, you will each have to build a small "Hello World"-style application that uses an open source model.

1. Choose a small model using the sources we discussed before.
2. Each group is to use one of the following frameworks
   - Huggingface
   - Ollama
   - LM-Studio from python
   - Llama.cpp to load and use the model in your application.
3. Present your results and your experiences with the frameworks in a short presentation.
4. Submit your code and report on moodle.

Further Readings

- This quite high-level blog-article about foundational models by Heidloff (2023a)
- The Attention is all you need-paper (Vaswani et al., 2023) and the brilliant video discussing it by Umar Jamil (Vaswani et al., 2023)
- This *very* good answer on stack exchange that explains the attention-concept ((https://stats.stackexchange.com/users/95569/dontloo), n.d.)

References

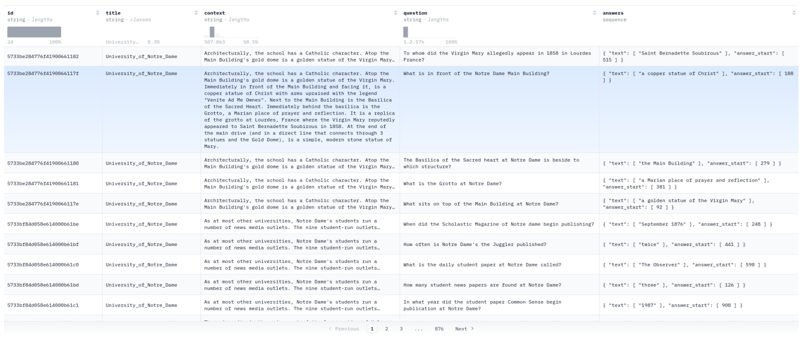
Prompting

Prompting describes the utilization of the ability of language models to use zero or few-shot instrutions to perform a task. This ability, which we briefly touched on when we were discussing the history of language models (i.e., the paper by Radford et al. (2019)), is one of the most important aspects of modern large language models.

Prompting can be used for various tasks such as text generation, summarization, question answering, and many more.

Instruct-tuned models

Instruct-tuned models are trained on a dataset (for an example, see Figure 1) that consists of instructions and their corresponding outputs. This is different from the pretraining phase of language models where they were trained on large amounts of text data without any specific task in mind. The goal of instruct-tuning is to make the model better at following instructions and generating more accurate and relevant outputs.

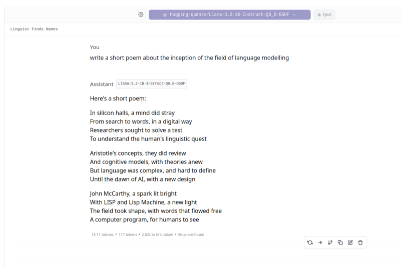


id	title	context	question	answers
string · *lengths*	string · *classes*	string · *lengths*	string · *lengths*	sequence
24 100%	University… 0.3%	507…863 58.5%	1…2.57k 100%	
5733be284776f41900661182	University_of_Notre_Dame	Architecturally, the school has a Catholic character. Atop the Main Building's gold dome is a golden statue of the Virgin Mary…	To whom did the Virgin Mary allegedly appear in 1858 in Lourdes France?	{ "text": ["Saint Bernadette Soubirous"], "answer_start": [515] }
5733be284776f4190066117f	University_of_Notre_Dame	Architecturally, the school has a Catholic character. Atop the Main Building's gold dome is a golden statue of the Virgin Mary. Immediately in front of the Main Building and facing it, is a copper statue of Christ with arms upraised with the legend "Venite Ad Me Omnes". Next to the Main Building is the Basilica of the Sacred Heart. Immediately behind the basilica is the Grotto, a Marian place of prayer and reflection. It is a replica of the grotto at Lourdes, France where the Virgin Mary reputedly appeared to Saint Bernadette Soubirous in 1858. At the end of the main drive (and in a direct line that connects through 3 statues and the Gold Dome), is a simple, modern stone statue of Mary.	What is in front of the Notre Dame Main Building?	{ "text": ["a copper statue of Christ"], "answer_start": [188] }
5733be284776f41900661180	University_of_Notre_Dame	Architecturally, the school has a Catholic character. Atop the Main Building's gold dome is a golden statue of the Virgin Mary…	The Basilica of the Sacred heart at Notre Dame is beside to which structure?	{ "text": ["the Main Building"], "answer_start": [279] }
5733be284776f41900661181	University_of_Notre_Dame	Architecturally, the school has a Catholic character. Atop the Main Building's gold dome is a golden statue of the Virgin Mary…	What is the Grotto at Notre Dame?	{ "text": ["a Marian place of prayer and reflection"], "answer_start": [381] }
5733be284776f4190066117e	University_of_Notre_Dame	Architecturally, the school has a Catholic character. Atop the Main Building's gold dome is a golden statue of the Virgin Mary…	What sits on top of the Main Building at Notre Dame?	{ "text": ["a golden statue of the Virgin Mary"], "answer_start": [92] }
5733bf84d058e614000b61be	University_of_Notre_Dame	As at most other universities, Notre Dame's students run a number of news media outlets. The nine student-run outlets…	When did the Scholastic Magazine of Notre dame begin publishing?	{ "text": ["September 1876"], "answer_start": [248] }
5733bf84d058e614000b61bf	University_of_Notre_Dame	As at most other universities, Notre Dame's students run a number of news media outlets. The nine student-run outlets…	How often is Notre Dame's the Juggler published?	{ "text": ["twice"], "answer_start": [441] }
5733bf84d058e614000b61c0	University_of_Notre_Dame	As at most other universities, Notre Dame's students run a number of news media outlets. The nine student-run outlets…	What is the daily student paper at Notre Dame called?	{ "text": ["The Observer"], "answer_start": [598] }
5733bf84d058e614000b61bd	University_of_Notre_Dame	As at most other universities, Notre Dame's students run a number of news media outlets. The nine student-run outlets…	How many student news papers are found at Notre Dame?	{ "text": ["three"], "answer_start": [126] }
5733bf84d058e614000b61c1	University_of_Notre_Dame	As at most other universities, Notre Dame's students run a number of news media outlets. The nine student-run outlets…	In what year did the student paper Common Sense begin publication at Notre Dame?	{ "text": ["1987"], "answer_start": [908] }

Fig 1: An example for a dataset that can be used for instruct-finetuning. This dataset can be found on huggingface

Task

Test the difference between instruct and non-instruct-models.
Do this by trying to get a gpt2-version (i.e., "QuantFactory/gpt2-xl-GGUF") and a small Llama 3.2 Instruct-Model (i.e., "hugging-quants/Llama-3.2-1B-Instruct-Q8_0-GGUF" to write a small poem about the inception of the field of language modelling.
Use LM-Studio to test this.



You

write a short poem about the inception of the field of language modelling

Assistant Llama-3.2-1B-Instruct-Q8_0-GGUF

Here's a short poem:

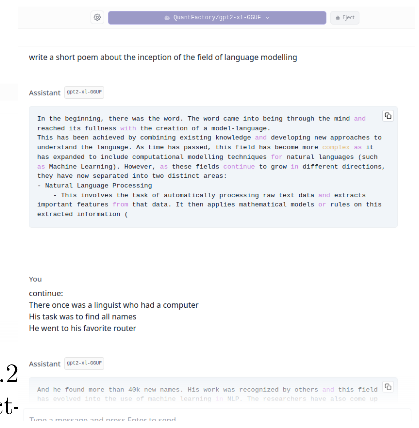
In silicon halls, a mind did stray
From search to words, in a digital way
Researchers sought to solve a test
To understand the human's linguistic quest

Aristotle's concepts, they did review
And cognitive models, with theories anew
But language was complex, and hard to define
Until the dawn of AI, with a new design

John McCarthy, a spark lit bright
With LISP and Lisp Machine, a new light
The field took shape, with words that flowed free
A computer program, for humans to see

18.11 tok/sec • 117 tokens • 2.02s to first token • Stop: eosFound

(a) A poem written by Llama 3.2 1B - a model with Instruct-Finetuning



write a short poem about the inception of the field of language modelling

Assistant gpt2-xl-GGUF

```
In the beginning, there was the word. The word came into being through the mind and
reached its fullness with the creation of a model-language.
This has been achieved by combining existing knowledge and developing new approaches to
understand the language. As time has passed, this field has become more complex as it
has expanded to include computational modelling techniques for natural languages (such
as Machine Learning). However, as these fields continue to grow in different directions,
they have now separated into two distinct areas:
- Natural Language Processing
    - This involves the task of automatically processing raw text data and extracts
important features from that data. It then applies mathematical models or rules on this
extracted information (
```

You

continue:
There once was a linguist who had a computer
His task was to find all names
He went to his favorite router

Assistant gpt2-xl-GGUF

```
And he found more than 40k new names. His work was recognized by others and this field
has evolved into the use of machine learning in NLP. The researchers have also come up
```

Type a message and press Enter to send

(b) A "poem" written by GPT2 - a model without Instruct-Finetuning

Fig 2: A poem and a "poem"

Show answer

Prompting strategies

The results of a prompted call to a LM is highly dependent on the exact wording of the prompt. This is especially true for more complex tasks, where the model needs to perform multiple steps in order to solve the task. It is not for naught that the field of "prompt engineering" has emerged. There is a veritable plethora of resources available online that discuss different strategies for prompting LMs. It has to be said though, that the strategies that work and don't work can vary greatly between models and tasks. A bit of general advice that holds true for nearly all models though, is to

a) define the task in as many small steps as possible
b) to be as literal and descriptive as possible and
c) to provide examples if possible.

Since the quality of results is so highly dependent on the chosen model, it is good practice to test candidate strategies against each other and therefore to define a target on which the quality of results can be evaluated. One example for such a target could be a benchmark dataset that contains multiple examples of the task at hand.

Task

1. Test the above-mentioned prompting strategies on the MTOP Intent Dataset and evaluate the results against each other. The dataset contains instructions and labels indicating on which task the instruction was intended to prompt. Use a python script to call one of the following three models in LM-Studio for this:

1. Phi 3.1 mini
2. Gemma 2 2B
3. Llama 3.2 1B

Use the F1-score implemented in scikit learn to evaluate your results.
2. You do sometimes read very specific tips on how to improve your results. Here are three, that you can find from time to time:

- Do promise rewards (i.e., monetary tips) instead of threatening punishments
- Do formulate using affirmation ("*Do the task*") instead of negating behaviours to be avoided ("*Don't do this mistake*")
- Let the model reason about the problem before giving an answer

Check these strategies on whether they improve your results. If your first instruction already results in near-perfect classification, brainstorm a difficult task that you can validate qualitatively. Let the model write a recipe or describe Kiel for example.
3. Present your results
3. Upload your code to moodle

Generation of synthetic texts

As we discussed before, small models can perform on an acceptable level, if they are finetuned appropriately.

A good way to do this is to use a larger model to generate synthetic data that you then use for training the smaller model. This approach has been used successfully in many applications, for example for improving graph-database queries (Zhong et al., 2024), for improving dataset search (Silva & Barbosa, 2024) or the generation of spreadsheet-formulas (Singh et al., 2024).

Since even the largest LLMs are not perfect in general and might be even worse on some specific niche tasks, evidence suggests that a validation strategy for data generated in this way is beneficial (Kumar et al., 2024; Singh et al., 2024).

Strategies to validate the synthetic data include:

- Using a human annotator to label part of the data to test the models output
- Forcing the model to answer in a structured way that is automatically testable (e.g., by using JSON)
- Forcing the model to return 2 or more answers and checking for consistency
- Combining the two approaches above (i.e., forcing the model to return multiple structured outputs (JSON, XML, YAML, …) and checking for consistency)
- Using a second LLM/different prompt to rate the answers

Task

Using your script for batch-testing different prompts, generate synthetic data for a emotion detection task based on Paul Ekman's six basic emotions: anger, disgust, fear, happiness, sadness and surprise[1].
The generated data should consist of a sentence and the emotion that is expressed in it. Start by generating two examples for each emotion. Validate these results and adapt them if necessary. Then use these examples to generate 100 samples for each emotion.
Use one of the above mentioned (non-manual) strategies to validate the data you generated.
Upload your results to Moodle.

Temperature

You might have encountered eerily similar answers from the language model, especially in the last task. Talking of it - why does the model return different answers to the same prompt at all if we do use pretrained-models in the first place? Shouldn't the utilization of the frozen weight-matrix result in the same answer, every time we run the model with the same input?

Yes, it should. And it does.

Remember that a language model trained on language generation as we discussed in the first session ends in a softmax-layer that returns probabilities for each token in the vocabulary. The generation-pipeline does not just use the token with the highest probability though, but samples from this distribution. This means, that even if the input is identical, the output will be different every time you run the model.

The temperature parameter controls the steepness of the softmax-function and thus the randomness of the sampling process. A higher temperature value results in more random outputs, while a lower temperature value results in more "deterministic" outputs. The temperatur, indicated as a float

[1]Though this nomenclature has fallen a bit out of fashion

between 0 and 1[2], is used to modulate the probabilities of the next token. This is done by adding a $\frac{1}{Temp}$ factor to the model-outputs before applying the softmax.

This effectively changes the Sofmax-fomula from

$$p_{Token} = \frac{e^{z_{Token}}}{\sum_{i=1}^{k} e^{z_i}}$$

to

$$p_{Token}(Temp) = \frac{e^{\frac{z_{Token}}{Temp}}}{\sum_{i=1}^{k} e^{\frac{z_i}{Temp}}}$$

Where

- z_{Token} is the output for a given token
- k is the size of the vocabulary
- $Temp$ is the temperature parameter $(0 < Temp <= 1)$

The effect of this temperature can be seen in Figure 3.

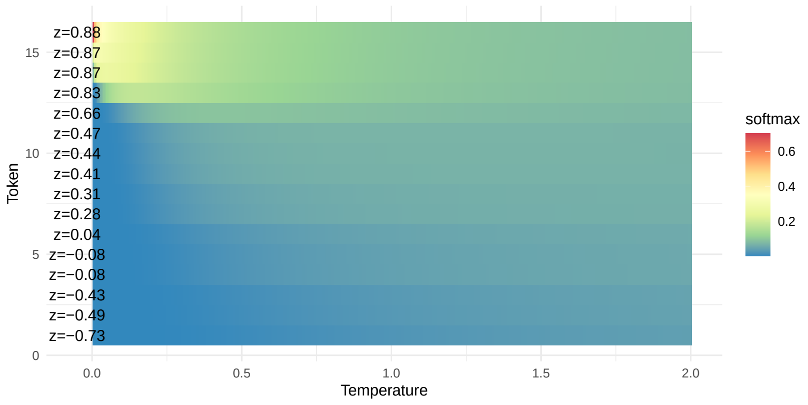


Fig 3: The effect of the temperature parameter on the softmax-output for a given input. The x-axis represents the temperature, the y-axis represents the token-position and the color represents the probability of the token.

[2]Depending on the implementation, temperatures above 1 are also allowed. Temperatures above 1 are resultsing in strange behaviours - see Figure 3.

Most generation-frameworks do additionally provide a parameter called *top_k* or *top_p*. These parameters are used to limit the number of tokens that can be selected as the next token. This is done by sorting the probabilities in descending order and only considering the top k tokens or the top p percent of tokens.

Temperature is the mayor setting to controll a LLMs "creativity" though.

Task

Using the script provided for generating snthetic data, test the effect of the temperature parameter on the output of the model.

- Use the same prompt and the same model
- Run the model with a temperature value of 0.1, 0.5, 1.0 and 2.0

Further Readings

- This prompting-guide has some nice general advice
- OpenAI has its own set of tipps
- deepset, the company behind Haystack, has a nice guide as well
- This blog-article, again written by Heidloff (Heidloff, 2023b)

References

Agent basics

What is an agent?

"An AI agent is a system that uses an LLM to decide the control flow of an application." ("What Is an AI Agent?" 2024)

In the context of large language models, agents are LLM-based systems that can solve complex tasks. Imagine asking a question like:

"What were the key learnings from the Generative AI elective module in WiSe 24/25 at FH Kiel?"

Could you just ask an LLM that question and expect a correct answer?

It is in theory possible, that an LLM could answer that directly, but only if it was trained on this information, that is, if a text describing the module exists, is accessible from the web and was used in training the model. However, usually we can not expect the LLM to have this knowledge.

Let's think for a moment how a human would answer that (one that did not attend the module). We would probably try to get a copy of the script, maybe we saved the script to our hard drive or other data storage. Maybe we could search the web for a description or text version of the module. Having obtained a copy of the script, we would probably read it. Then, we would try to distill the information hidden therein, to answer the question.

So, for our LLM to answer that question, it needs to be able to perform several tasks:

- Searching the web for relevant documents
- searching in a local file storage or other database

- Reading and understanding a document
- Summarizing the content of a document
- Answering questions based on the summary of a document

This is where agents come into play. Agents are LLM-based systems that can solve complex tasks by performing several subtasks in sequence, using an LLM to decide which subtask to perform next. In our example, the agent would first search the web for relevant documents, then read and understand them, summarize them and finally answer the question based on the summary.

Agent framework

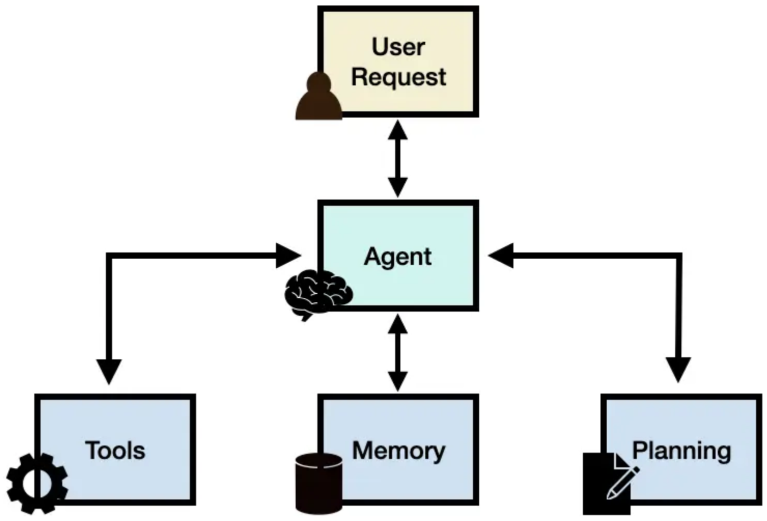


Fig 1: Architecture of the agent framework (*LLM Agents – Nextra*, 2024)

To facilitate this, an agent system consists of several components:

- **Agent**: the agent core acting as coordinator

- **Planning**: Assists the agent in breaking down the complex task into subtasks
- **Tools**: functions that the agent can use to perform a specific task
- **Memory**: used to store information about previous interactions with the agent

We will describe each of them below.

Agent

This is a general-purpose LLM, that functions as the brain and main decision-making component of an agent. It determines which tools to use and how to combine their results to solve complex tasks. The agent core uses the output of the previous tool as input for the next tool. It also uses an LLM to decide when to stop using tools and return a final answer. The behavior of the agent and the tools, it has at its disposal, is defined by a prompt template.

Planning

Planning is the process of breaking down a complex task into subtasks and deciding which tools to use for each subtask. The planning module is usually also an LLM, it can be one fine-tuned to this specific task or receive a specialized prompt. It uses techniques like **chain-of-thought** (CoT) prompting to generate a plan of action (Wei et al., 2023). CoT prompting is a technique that encourages the model to explain its reasoning step by step, making it easier for us to understand and evaluate its answers. Other strategies include **Tree-of-Thoughts** (Long, 2023), (Yao, Yu, et al., 2023), (Hulbert, 2023) or **ReAct** (Yao, Zhao, et al., 2023). We will discuss these in more detail later.

Tools

Tools are functions that the agent can use to perform a specific task. They can be pre-defined or dynamically generated based on the user's needs. Tools can be simple, such as a calculator, or complex, such as a web search engine. Tools can also be other agents, allowing for the creation of multi-agent systems. In our example, the tools would be a web search engine and a document reader. Other popular tools are a data store or a python interpreter.

Memory

Memory is used to store information about previous interactions with the agent. This allows the agent to remember past conversations and use this information in future interactions. Memory can be short-term, such as a conversation buffer, or long-term, such as a database. Memory can also be used to store the results of previous tool uses, allowing the agent to reuse them if necessary.

Chain-of-Thought prompting

Chain-of-Thought (CoT) prompting refers to the technique of giving the LLM hints in the user input on how to solve the problem step by step, similar to what we did above. In the original paper, this was used with few-shot prompting (giving the LLM examples in the prompt), see figure below. But it is also possible to use it with zero-shot prompting (i.e. without examples) by invoking the magical words "Let's think step by step" (Kojima et al., 2023)[1].

[1] Note that these informations get old *fast*. Newer LLMs may have this functionality build in already

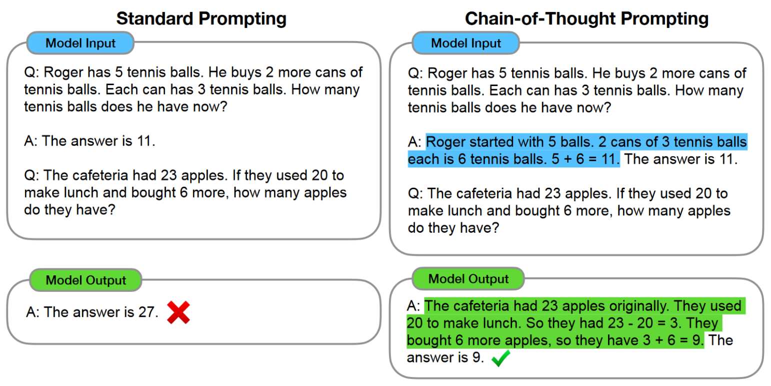

Standard Prompting
Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The answer is 27.
Chain-of-Thought Prompting
Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9.

Fig 2: Chain-of-Thought prompting illustrated (Wei et al., 2023)

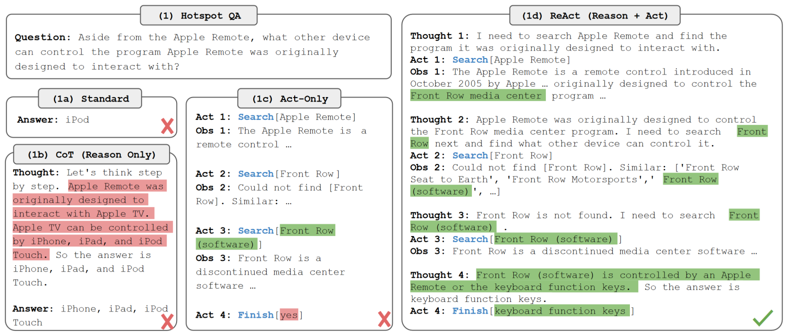
Tree of Thoughts

Tree of Thoughts (ToT) is a generalization on CoT prompting. The papers on ToT are somewhat complex, so we will not discuss them in detail here. In short, LLMs are used to generate thoughts, that serve as intermediate steps towards the solution. The difference to CoT is basically, that several thoughts are generated at each step, creating a tree-like structure. This tree is then searched using breadth-first search or depth-first search until a solution is found. A simplified example is given by (Hulbert, 2023):

```
Imagine three different experts are answering this question.
All experts will write down 1 step of their thinking,
then share it with the group.
Then all experts will go on to the next step, etc.
If any expert realises they're wrong at any point then they leave.
The question is...
```

ReAct

ReAct (short for Synergizing Reasoning and Acting) is a technique based on CoT, that updates its reasoning after each step of tool use. This allows the agent to react (pun intended) to unforeseen results during the step-by-step solution i.e. failed tool use. The agent can then follow a different chain of thoughts. This makes it very well suited to tool use. An illustration is given in the figure below.



```
(1) Hotspot QA
Question: Aside from the Apple Remote, what other device
can control the program Apple Remote was originally
designed to interact with?
```

```
(1a) Standard
Answer: iPod ✗
```

```
(1b) CoT (Reason Only)
Thought: Let's think step
by step. Apple Remote was
originally designed to
interact with Apple TV.
Apple TV can be controlled
by iPhone, iPad, and iPod
Touch. So the answer is
iPhone, iPad, and iPod
Touch.

Answer: iPhone, iPad, iPod
Touch ✗
```

```
(1c) Act-Only
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a
remote control …

Act 2: Search[Front Row]
Obs 2: Could not find [Front
Row]. Similar: …

Act 3: Search[Front Row
(software)]
Obs 3: Front Row is a
discontinued media center
software …

Act 4: Finish[yes] ✗
```

```
(1d) ReAct (Reason + Act)
Thought 1: I need to search Apple Remote and find the
program it was originally designed to interact with.
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control introduced in
October 2005 by Apple … originally designed to control the
Front Row media center program …

Thought 2: Apple Remote was originally designed to control
the Front Row media center program. I need to search Front
Row next and find what other device can control it.
Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: ['Front Row
Seat to Earth', 'Front Row Motorsports',' Front Row
(software)', …]

Thought 3: Front Row is not found. I need to search Front
Row (software) .
Act 3: Search[Front Row (software) ]
Obs 3: Front Row is a discontinued media center software …

Thought 4: Front Row (software) is controlled by an Apple
Remote or the keyboard function keys. So the answer is
keyboard function keys.
Act 4: Finish[keyboard function keys ] ✓
```

Fig 3: Comparison of ReAct with other prompting techniques (Yao, Zhao, et al., 2023)

Examples of agent-frameworks (Llamaindex, LangChain & Haystack)

There are *a lot* of agent frameworks out there. In this module we will focus on three of them: LlamaIndex, LangChain and Haystack. They all have their own strengths and weaknesses, but they all share the same basic architecture as described above. We will describe each of them below.

- Llamaindex: LlamaIndex is a data framework for your LLM applications. It provides a central interface to connect your LLMs and your data. It also provides a set of tools to help you build your own applications, such as a document reader, a web search engine, a data store, etc. - LangChain: LangChain is a framework for developing applications powered by language models. It provides a set of tools to help you build your own applications, such as a document reader, a web search engine, a data store, etc. It also provides a set of agents that can use these tools to solve complex tasks.

- Haystack: Haystack is an open source NLP framework that enables you to build production-ready applications around LLMs and other models. It provides a set of tools to help you build your own applications, such as a document reader, a web search engine, a data store, etc. It also provides a set of agents that can use these tools to solve complex tasks.

Task

Now it is your turn!
Each group is to use one of the following frameworks to build a small demo agent:

- Llamaindex combine this approach with this notebook to make it work with LM Studio.
- Langchain
- Haystack
- (optional) another framework of your choice

1. Set up a local LLM (e.g. using Ollama or LM Studio) to be used by the agent.
2. Choose a small task for your agent, e.g. answering questions about a specific topic, summarizing a document, etc. (use the one in the respective tutorial)
3. Implement the agent using one of the frameworks listed above.
4. Present your results and your experiences with the frameworks in a short presentation.
5. Submit your code and report on moodle.

Further Readings

- This paper compares different planning strategies
- In addition to the websites listed above see also ("Introduction to LLM Agents," 2023)

References

Embedding-based agent-systems

All agents we discussed until here are using tools that allow them to use their generated inputs in some way. In most of the task we want to utilize agents, we do not only want to generate text but to also inform the generation based on some kind of existing knowledge base. Examples for these kinds of usecases include:

- Answering questions about a specific topic (e.g., a company or product)
- Summarizing a document
- Generating a report based on data

Though most modern LLMs are increasingly capable in answering basic knowledge-questions, the more comples a topic or the more relevant the factual basis of an answer is, the more it is important to base generated answers on actual data.

Semantic embeddings and vector stores

To empower an agent too look up information during its thought-process, one has to build a tool that allows an agent to use natural language to retrieve information necessary for a task. The fundamental principle to do this are so-called *semantic embeddings*. These are pretty close to the concept we introduced when talking about the foundations of LLMs (see here) and can be understood as a way to map textual data into a vector space. The main idea is that semantically similar texts should have similar embeddings, i.e., they are close in the vector space. Close in this context is meant as having a reasonibly small distance between them. The go-to standard to measure this distance is the **cosine similarity**, which has proven usefull enough to be the standard for a range of semantic retrieval implementations (i.e., they are used in OpenAI tutorials and in Azure embedding-applications). The cosine similarity is defined as:

$$\text{cosine\_similarity}(u, v) = \frac{u \cdot v}{\|u\|\|v\|} = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2}\sqrt{\sum_{i=1}^{n} v_i^2}}$$

The rationale here is that sequences with semantically similar contents should point to similar directions in the high dimensional vector space. See Figure 1 for an illustration of this and other common similarity concepts seen in semantic retrieval.

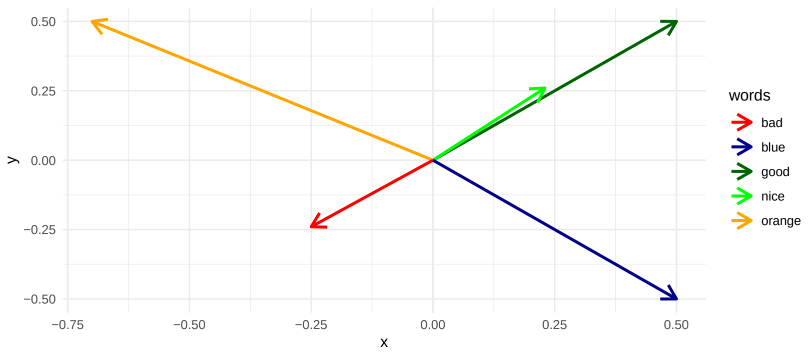
As always, there is not the one solution to all problems though and the applicability of cosine similarity might not be optimal for your usecase (Goyal & Sharma, 2022; Steck et al., 2024).

Though one could use any kind of (L)LM to calculate embeddings for this case[1], it is advisable to use models specifically trained for this purpose. Reimers & Gurevych (2019) proposed *Sentence-BERT* which is a simple but effective approach to calculate semantic embeddings. SBERT and similar approaches are based on a (L)LM that was trained to predict missing words as we discussed before, resulting in a general representation of natural language. In the case of the original paper, they used (among others) the BERT model Devlin et al. (2019) mentioned before.

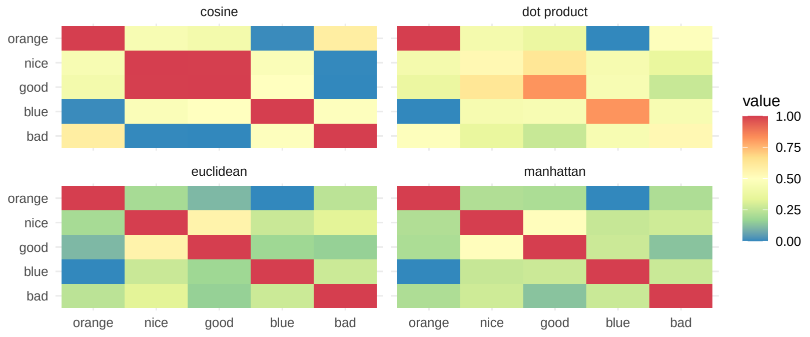
The authors then use this to embed a pair of sentences into one embedding-vector each[2], for which some measure of semantic similarity is known. An example for a dataset containing such sentences is the Stanford Natural Language Inferenc(SNLI) corpus Bowman et al. (2015) which labels 550k

[1]And there are approaches to use LLMs to solve this taks i.e., T. Jiang et al. (2023)

[2]The original BERT-paper did this by adding a pooling layer before the task-header that extracted and weighed the context-dependend embedding of the first token. The SBERT paper tried different pooling-strategies and used a mean over each embedding dimension of the sequence.



(a) Illustration of "semantic embeddings" of different word.



(b) Illustration of 4 common similarity concepts seen in semantic retrieval: cosine, euclidean, dot product and manhattan. dot product and cosine are taking the direction of the vector into account, while the cosine ignores the length of the vectors and the dot product does not. Manhattan and euclidean are both measuring the distance between two points in a vector space, but they do it differently. Euclidean is the straight line between two points, while manhattan is the sum of the absolute differences between the coordinates of the two points.

Fig 1: Illustration of common similarity metrics in semantic search.

pairs of sentences as either *entailment*, *contradiction* or *neutral*. Reimers & Gurevych (2019) then concated the both senteces embeddings and their element-wise difference into a single vector which is fed to a multiclass classifier, indicating in which category the sentences relationship falls. At inference, this classification head was removed and replaced as the cosine similarity as discussed above. The resulting network is highly effective in calculating semantic similarities between sentences.

A look at the sbert-website shows that the module has somewhat grown and now does supply a series of learning paradigms that can be used to efficiently tune a model for your specific usecase[3]. As the library has grown, so has the sheer amount of pretrained embedding-models in some way based on this architecture that are hosted on huggingface. The MTEB-Leaderboard is a good strat to search for a model for your application. One utilization of this model-family, which has already been implicitly used in this script, is their very efficient ability to semantically search for documents. If a model is very good at finding similar sentences, it can also be very good to find documents that are very similar to a question.

Look at the example illustrated in Figure 2. The question "why is the sky blue" embedded with the same model as our 5 documents stating some facts.

We can then calculate the cosine-similarity between these embeddings and return the document, that has the highest similarity to our question.

Task

Install the sentence-transformer package and download the climate_fever-dataset.
Choose one model from the MTEB-Leaderboard that you deem adequatly sized and appropriate for the task
Test the different metrics for the first twenty claims of the dataset and a question you formulate.
Use the similarity-implementations from

[3]And this does not have to be expensive. Tunstall et al. (2022) have shown a highly efficient contrastive learning paradigm that limts the amount of necessary labels for a ridiuculously small amount of labels.

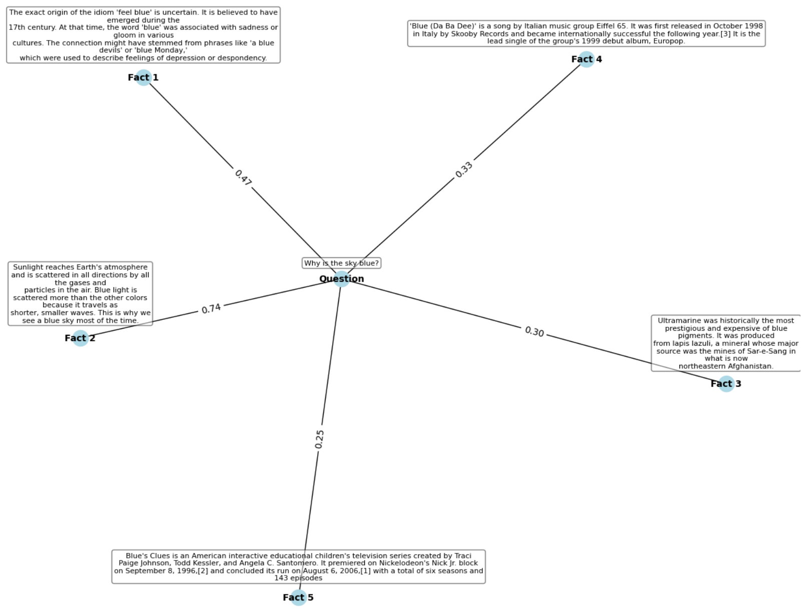

The exact origin of the idiom 'feel blue' is uncertain. It is believed to have emerged during the
17th century. At that time, the word 'blue' was associated with sadness or gloom in various
cultures. The connection might have stemmed from phrases like 'a blue devils' or 'blue Monday,'
which were used to describe feelings of depression or despondency.
Fact 1
'Blue (Da Ba Dee)' is a song by Italian music group Eiffel 65. It was first released in October 1998 in Italy by Skooby Records and became internationally successful the following year.[3] It is the lead single of the group's 1999 debut album, Europop.
Fact 4
0.47
0.33
Why is the sky blue?
Question
Sunlight reaches Earth's atmosphere and is scattered in all directions by all the gases and
particles in the air. Blue light is scattered more than the other colors because it travels as
shorter, smaller waves. This is why we see a blue sky most of the time.
Fact 2
0.74
0.30
Ultramarine was historically the most prestigious and expensive of blue pigments. It was produced
from lapis lazuli, a mineral whose major source was the mines of Sar-e-Sang in what is now
northeastern Afghanistan.
Fact 3
0.25
Blue's Clues is an American interactive educational children's television series created by Traci Paige Johnson, Todd Kessler, and Angela C. Santomero. It premiered on Nickelodeon's Nick Jr. block on September 8, 1996,[2] and concluded its run on August 6, 2006,[1] with a total of six seasons and 143 episodes
Fact 5

Fig 2: Illustration of the usage of embedding-based distances in retrieval.

sklearn.metrics.pairwise.

This approach of using a model to embed documents and questions into a vector space is the basis for the so-called *Retrieval augmented generation*.

Retrieval augmented generation

Retrieval augmented generation (RAG) is a framework that does pretty much do what it says on the tin. You use a retrieval model to find documents that are similar to your question and then either return these documents our feed them into a generative model, which then generates an answer based on these documents. This process can additionally be wrapped as a tool to be used by an agent, so that your existing agent can now also use external knowledge sources to answer questions.

Retrieval does not have to be semantics-based in this context - all kinds of data sources and databases can be made accessible for a LLM - we will focus on a purely embbedding based approach here though.

Although the small example in the last task was working, it is not really scalable. It was fine for a limited set of examples, if you want to realistically make a whole knowledge base searchable, you need to use an appropriate database system.

Vector databases

A vector database is a database that stores vectors and allows for efficient similarity searches. As can be seen in the db-engines ranking there has been a surge of interest in this area recently, with many new players entering the market. From the plethora of vector databases, these three are examples that virtue a honorary mention:

1. Chroma - a in-memory database for small applications that is especially easy to get to run.

2. Elasticsearch - a well established database that is the go to system for open source search engines and has recently (and kind of naturally) also branched out into vector databases.

3. Qdrant - the product of a Berlin-based startup that focusses on stability and scalability. It can also run in memory, but does natively support hard drive storage.

The best way to use qdrant is to use docker to run it and the python sdk to interact with it. Since version 1.1.1, the sdk also allows to just run the client in memory.

Task

Install the qdrant-client python-sdk and fastembed.
Create a collection for the claims and one for the evidence in the climate_fever-dataset. Add the first 200 entries to each of these collections. Use qdrants fastembemd-integration to do this.
Test the similarity search on a question you formulate.

RAG

The last step to make this into a RAG pipeline is to use a generative model to answer the question based on the retrieved documents.

This means, that we do collect the relevant documents like we did before, still based on a natural language question, but instead of returning the hits we got from the index, we feed them into a LLM and ask it to generate an answer based on these documents. This is where the name retrieval augmented generation comes from - we use the retrieval step to augment the generative model with additional information. The diagram in Figure 3 illustrates this process.

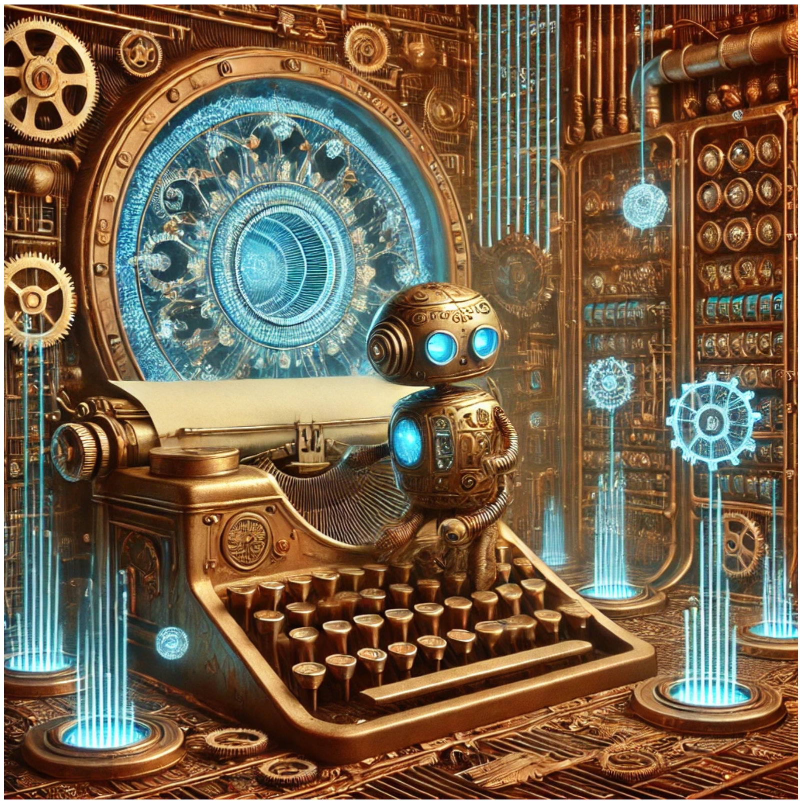


Fig 3: Illustration of a RAG-system.

Task

Implement a RAG pipeline for the climate_fever dataset using qdrant as vector database and a LLM of your choice for the summarization.
Try to find a prompt that results in the LLM

a) using the information given
b) not inventing new information
c) referencing the source of the information it uses

Upload your results until here (embedder, database and summarization) to moodle.

Most agent frameworks provide integrations for a variety of vector databases.

In terms of llamaindex, there are not just one but two tutorials on how to get qdrant to integrate into your agent, one from qdrant for general integration and one from llamaindex.

The pipeline is pretty close to what we discussed until here, it just uses the llamaindex-typical wrapper classes. See Tip 1 for an example RAG-system implemented in Llamaindex.

Tip 1: Llamaindex Rag

The first thing in both the Llamaindex and the manual way of creating a retrieval pipeline is the setup of a vector database:

```
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, Batch
DIMENSIONS = 384
client = QdrantClient(location=":memory:")
```

To store data and query the database, we have to load a embedding-model. As in the manual way of creating a retrieval pipeline discussed before, we can use a huggingface-SentenceTranformer model. But instead of using the SentenceTransformer class from the sentence_transformers library, we have to use the Hugging-

FaceEmbedding class from Llamaindex. This model is entered into the Llamaindex-Settings.

```
from llama_index.core import Settings
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
embed_model = HuggingFaceEmbedding(model_name="sentence-transformers/all-MiniLM-L12-v2")
Settings.embed_model = embed_model
```

The next step is to wrap the vector-store into a Llamaindex-VectorStoreIndex. This index can be used to add our documents to the database.

```
from llama_index.vector_stores.qdrant import QdrantVectorStore

vector_store = QdrantVectorStore(client=client, collection_name="paper")
```

As an example, we will add the "Attention is all you need" paper. This is how the head of our txt-file looks like:

```
         Attention Is All You Need
arXiv:1706.03762v7 [cs.CL] 2 Aug 2023




                                       Ashish Vaswani*                 Noam Shazeer*
                                        Google Brain                   Google Brain
                                     avaswani@google.com             noam@google.com
```

Since we can not just dump the document at once, we will chunk it in sentences (more about that later). This can be done like this (ignore the parameters by now, we will look at them later):

```
from llama_index.core.node_parser import SentenceSplitter
from llama_index.core import Document

node_parser = SentenceSplitter(chunk_size=100, chunk_overlap=20)

nodes = node_parser.get_nodes_from_documents(
    [Document(text=text)], show_progress=False
)
```

These documents are then added to our database and transformed in an *index* llamaindex can use:

```
from llama_index.core import VectorStoreIndex

index = VectorStoreIndex(
    nodes=nodes,
    vector_store=vector_store,
)
```

This index can already be used to retrieve documents from the database (by converting it to a *retriever*).

```
retriever = index.as_retriever(similarity_top_k=10)
retriever.retrieve('What do the terms Key, Value and Query stand for in self-attention?')
```

```
[NodeWithScore(node=TextNode(id_='04c12537-5f33-4d41-a4d4-df30d2aed6e4', embedding=None, meta
 NodeWithScore(node=TextNode(id_='c42d8e8c-24ac-447a-8058-d62d198ce9eb', embedding=None, meta
 NodeWithScore(node=TextNode(id_='893d077f-a8ab-4a3f-9765-69ef72d46ec4', embedding=None, meta
 NodeWithScore(node=TextNode(id_='0146f53a-f1b1-4d80-a333-26746920ab9d', embedding=None, meta
 NodeWithScore(node=TextNode(id_='22d5c0dc-d921-4790-ac6e-4f6a6d5f336f', embedding=None, meta
 NodeWithScore(node=TextNode(id_='55481635-fcaa-4e90-9625-9b0c3bfa3109', embedding=None, meta
 NodeWithScore(node=TextNode(id_='04b195bd-26e4-4d8c-afdc-780e96bdd345', embedding=None, meta
 NodeWithScore(node=TextNode(id_='d93b8e55-28cb-417e-838a-a22abf7cfbc9', embedding=None, meta
 NodeWithScore(node=TextNode(id_='158309a7-9a7a-47e6-ac58-1a4e98eee41b', embedding=None, meta
 NodeWithScore(node=TextNode(id_='721c5981-90a9-4046-a757-4593a362ddf7', embedding=None, meta
```

The retriever can then directly be use as a tool to answer questions about our documents:

```
from llama_index.core.tools import BaseTool, FunctionTool

def find_references(question: str) -> str:
    """Query a database containing the paper "Attention is all you Need" in parts.
    This paper introduced the mechanism of self-attention to the NLP-literature.
    Returns a collection of scored text-snippets that are relevant to your question."""
    return '\n'.join([f'{round(n.score,2)} - {n.node.text}' for n in retriever.retrieve(quest


find_references_tool = FunctionTool.from_defaults(fn=find_references)
```

This tool can then be added to an agent as we discussed before:

```python
from llama_index.core.agent import ReActAgent

from llama_index.llms.lmstudio import LMStudio


llm = LMStudio(model_name="llama-3.2-1b-instruct",
        base_url="http://localhost:1234/v1",
    temperature=0.5,
    request_timeout=600)


agent = ReActAgent.from_tools(tools=[find_references_tool],llm=llm, verbose=True)
```

```
/home/brede/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/pydantic/_in

You may be able to resolve this warning by setting `model_config['protected_namespaces'] = ()
  warnings.warn(
```

Which can then be used to answer chat-requests:

```python
response = agent.chat("What is the meaning of Query, Key and Value in the context of self-att
print(str(response))
```

```
> Running step 062240ab-0d21-4fdb-a603-fb386970c32f. Step input: What is the meaning of Query
Observation: Error: Could not parse output. Please follow the thought-action-input format. Tr
> Running step 2a291a80-5090-4373-945d-3a647ac2b758. Step input: None
Observation: Error: Could not parse output. Please follow the thought-action-input format. Tr
> Running step 908425d7-8f06-4830-8585-4ff312b43c45. Step input: None
Observation: Error: Could not parse output. Please follow the thought-action-input format. Tr
> Running step 543ab12f-e5e7-4a59-b103-b7fc7bd0a3fe. Step input: None
Observation: Error: Could not parse output. Please follow the thought-action-input format. Tr
> Running step 1b3cb4e3-e976-4420-a489-906b8f6c5776. Step input: None
Thought: Let's break down what Query, Key, and Value mean in the context of self-attention.
Action: Use
Action Input: 'input': "What are the most relevant words for the sentence 'The quick brown fo
Observation: Error: No such tool named `Use`.
> Running step 1101520e-54ff-42db-b327-d9d902acb957. Step input: None
```

```
Thought: I need to find a way to input the query and parameters into a tool.
Action: Use
Action Input: 'input': "What are the most relevant words for the sentence 'The quick brown fo
Observation: Error: No such tool named `Use`.
> Running step 47c5a9f6-5055-4f8c-9a3b-49f1db40abcb. Step input: None
Thought: I'm using a different tool to find references. Let me check if it supports finding r
Action: find_references
Action Input: 'properties': AttributedDict([('question', "What are the most relevant words fo
Observation: Error: find_references() got an unexpected keyword argument 'properties'
ValueError: Reached max iterations.
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[10], line 1
----> 1 response = agent.chat("What is the meaning of Query, Key and Value in the context of
      2 print(str(response))
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    308             _logger.debug(f"Failed to reset active_span_id: e")
    310 try:
--> 311     result = func(*args, **kwargs)
    312     if isinstance(result, asyncio.Future):
    313         # If the result is a Future, wrap it
    314         new_future = asyncio.ensure_future(result)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
     39 callback_manager = cast(CallbackManager, callback_manager)
     40 with callback_manager.as_trace(trace_id):
---> 41     return func(self, *args, **kwargs)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    642     tool_choice = self.default_tool_choice
    643 with self.callback_manager.event(
    644     CBEventType.AGENT_STEP,
    645     payload=EventPayload.MESSAGES: [message],
    646 ) as e:
--> 647     chat_response = self._chat(
    648         message=message,
    649         chat_history=chat_history,
    650         tool_choice=tool_choice,
    651         mode=ChatResponseMode.WAIT,
    652     )
    653     assert isinstance(chat_response, AgentChatResponse)
    654     e.on_end(payload=EventPayload.RESPONSE: chat_response)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    308             _logger.debug(f"Failed to reset active_span_id: e")
    310 try:
--> 311     result = func(*args, **kwargs)
    312     if isinstance(result, asyncio.Future):
    313         # If the result is a Future, wrap it
    314         new_future = asyncio.ensure_future(result)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    576 dispatcher.event(AgentChatWithStepStartEvent(user_msg=message))
    577 while True:
    578     # pass step queue in as argument, assume step executor is stateless
--> 579     cur_step_output = self._run_step(
    580         task.task_id, mode=mode, tool_choice=tool_choice
    581     )
    583     if cur_step_output.is_last:
    584         result_output = cur_step_output
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    308             _logger.debug(f"Failed to reset active_span_id: e")
    310 try:
--> 311     result = func(*args, **kwargs)
    312     if isinstance(result, asyncio.Future):
    313         # If the result is a Future, wrap it
    314         new_future = asyncio.ensure_future(result)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    408 # TODO: figure out if you can dynamically swap in different step executors
    409 # not clear when you would do that by theoretically possible
    411 if mode == ChatResponseMode.WAIT:
--> 412     cur_step_output = self.agent_worker.run_step(step, task, **kwargs)
    413 elif mode == ChatResponseMode.STREAM:
    414     cur_step_output = self.agent_worker.stream_step(step, task, **kwargs)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    308             _logger.debug(f"Failed to reset active_span_id: e")
    310 try:
--> 311     result = func(*args, **kwargs)
    312     if isinstance(result, asyncio.Future):
    313         # If the result is a Future, wrap it
    314         new_future = asyncio.ensure_future(result)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
```

```
     39 callback_manager = cast(CallbackManager, callback_manager)
     40 with callback_manager.as_trace(trace_id):
---> 41     return func(self, *args, **kwargs)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    815 @trace_method("run_step")
    816 def run_step(self, step: TaskStep, task: Task, **kwargs: Any) -> TaskStepOutput:
    817     """Run step."""
--> 818     return self._run_step(step, task)
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    572 reasoning_steps, is_done = self._process_actions(
    573     task, tools, output=chat_response
    574 )
    575 task.extra_state["current_reasoning"].extend(reasoning_steps)
--> 576 agent_response = self._get_response(
    577     task.extra_state["current_reasoning"], task.extra_state["sources"]
    578 )
    579 if is_done:
    580     task.extra_state["new_memory"].put(
    581         ChatMessage(content=agent_response.response, role=MessageRole.ASSISTANT)
    582     )
File ~/MEGA/Honorar/Generative AI/script/.venv/lib/python3.10/site-packages/llama_index/core/
    435     raise ValueError("No reasoning steps were taken.")
    436 elif len(current_reasoning) == self._max_iterations:
--> 437     raise ValueError("Reached max iterations.")
    439 if isinstance(current_reasoning[-1], ResponseReasoningStep):
    440     response_step = cast(ResponseReasoningStep, current_reasoning[-1])
ValueError: Reached max iterations.
```

As you can see, the model request ends up with errors. The model is not powerful enough to answer in the structured manner we need for the function-calling of the tool. To circumvent this, we can try a function-calling-finetuned model:
We can try to solve this issue by using a language model that is finetuned on function calling:

```
fc_llm = LMStudio(model_name="phi-3-mini-4k-instruct-function-calling",
        base_url="http://localhost:1234/v1",
    temperature=0.2,
    request_timeout=600)

agent = ReActAgent.from_tools(tools=[find_references_tool],llm=fc_llm, verbose=True)
response = agent.chat("What is the meaning of Query, Key and Value in the context of self-att
print(str(response))
```

```
> Running step 78c0a52b-55fa-4241-ade5-67c0b92b9bf3. Step input: What is the meaning of Query
Thought: (Implicit) I can answer without any more tools!
Answer:  In the context of self-attention, "Query", "Key" and "Value" are terms used to descr
1. Query - The query component is used to retrieve information from memory banks during atter
2. Key - The key component is used to determine which parts of the input sequence are most re
3. Value - The value component is used to store the actual data corresponding to each memory
In summary, query, key and value are components of a neural network architecture used in self
 In the context of self-attention, "Query", "Key" and "Value" are terms used to describe dif
1. Query - The query component is used to retrieve information from memory banks during atter
2. Key - The key component is used to determine which parts of the input sequence are most re
3. Value - The value component is used to store the actual data corresponding to each memory
In summary, query, key and value are components of a neural network architecture used in self
```

This model does not run into an issue with the structured output, it does not try to use the tool anymore though. One way to try to solve this issue is to adapt the agent-prompt:

```
print(agent.get_prompts()['agent_worker:system_prompt'].template)
```

```
You are designed to help with a variety of tasks, from answering questions to providing summa

## Tools

You have access to a wide variety of tools. You are responsible for using the tools in any se
This may require breaking the task into subtasks and using different tools to complete each s

You have access to the following tools:
{tool_desc}
```

~~~
## Output Format

Please answer in the same language as the question and use the following format:

```
Thought: The current language of the user is: (user's language). I need to use a tool to help
Action: tool name (one of {tool_names}) if using a tool.
Action Input: the input to the tool, in a JSON format representing the kwargs (e.g. {{"input'
```

Please ALWAYS start with a Thought.

NEVER surround your response with markdown code markers. You may use code markers within your

Please use a valid JSON format for the Action Input. Do NOT do this {{'input': 'hello world',

If this format is used, the user will respond in the following format:

```
Observation: tool response
```

You should keep repeating the above format till you have enough information to answer the que

```
Thought: I can answer without using any more tools. I'll use the user's language to answer
Answer: [your answer here (In the same language as the user's question)]
```

```
Thought: I cannot answer the question with the provided tools.
Answer: [your answer here (In the same language as the user's question)]
```

## Current Conversation

Below is the current conversation consisting of interleaving human and assistant messages.
~~~

This we can adapt in the following way:

```
from llama_index.core import PromptTemplate
new_agent_template_str = """
You are designed to help answer questions based on a collection of paper-excerpts.

## Tools

You have access to tools that allow you to query paper-content. You are responsible for using
This may require breaking the task into subtasks and using different tools to complete each s

You have access to the following tools:
{tool_desc}


## Output Format

Please answer in the same language as the question and use the following format:

\`\`\`
Thought: The current language of the user is: (user's language). I need to use a tool to help
Action: tool name (one of {tool_names}) if using a tool.
Action Input: the input to the tool, in a JSON format representing the kwargs (e.g. {{"input"
\`\`\`


Please ALWAYS start with a Thought.

NEVER surround your response with markdown code markers. You may use code markers within your
...
## Current Conversation

Below is the current conversation consisting of interleaving human and assistant messages.
"""
new_agent_template = PromptTemplate(new_agent_template_str)
agent.update_prompts(
    {"agent_worker:system_prompt": new_agent_template}
)
```

We can test this new prompt with the same question:

```
response = agent.chat("What is the meaning of Query, Key and Value in the context of self-att
print(str(response))
```

```
> Running step d5fb46ea-de7a-4e8b-ace7-7ed3ae6a9706. Step input: What is the meaning of Query
Thought: (Implicit) I can answer without any more tools!
Answer:  In the context of natural language processing (NLP), "Query", "Key" and "Value" are
Here's how these terms relate to the model:
1. Query - A query is an input vector that represents the current state of a sequence being p
2. Key - The key component in a transformer model refers to a set of learned weights that hel
3. Value - The value component is responsible for storing information from a specific memory
In summary, Query, Key and Value are components of a neural network architecture used in Tran
```

```
 In the context of natural language processing (NLP), "Query", "Key" and "Value" are used as
Here's how these terms relate to the model:
1. Query - A query is an input vector that represents the current state of a sequence being p
2. Key - The key component in a transformer model refers to a set of learned weights that hel
3. Value - The value component is responsible for storing information from a specific memory
In summary, Query, Key and Value are components of a neural network architecture used in Tran
```

The model still tries to answer without the tool.
Let's try to ask a more specific question:

```
response = agent.chat("How does the paper 'Attention is all you need' define the term self at
print(str(response))
```

```
> Running step 3c1b3050-f4a0-4b46-9006-366161df0219. Step input: How does the paper 'Attentic
Thought: (Implicit) I can answer without any more tools!
Answer:  In the paper "Attention Is All You Need", the authors present a novel Transformer mo
Self-attention is defined in the paper as follows: given a sequence of input tokens (or words
```

```
 In the paper "Attention Is All You Need", the authors present a novel Transformer model that
Self-attention is defined in the paper as follows: given a sequence of input tokens (or words
```

Still no dice.
One solution to this problem is to just use a bigger model:

```
llm = LMStudio(model_name="llama-3.2-3b-instruct", #3 Billion instead of 1
        base_url="http://localhost:1234/v1",
    temperature=0.2,
    request_timeout=600)


agent = ReActAgent.from_tools(tools=[find_references_tool],llm=llm, verbose=True)

response = agent.chat("How does the paper 'Attention is all you need' define the term self at
print(str(response))
```

```
> Running step 9326aba5-48cf-40dd-8b85-b5da82554e5c. Step input: How does the paper 'Attentic
Thought: The current language of the user is English. I need to use a tool to help me answer
Action: find_references
Action Input: {'properties': AttributedDict([('question', AttributedDict([('title', 'self-att
Observation: Error: find_references() got an unexpected keyword argument 'properties'
> Running step b9bd6255-c348-473e-a031-2fd1e4e74cdf. Step input: None
Thought: The current language of the user is English. I need to use a tool to help me answer
Action: find_references
Action Input: {'question': "How does the paper 'Attention is all you Need' define the term se
Observation: 0.69 - Self-attention, sometimes called intra-attention is an attention mechanis
of a single sequence in order to compute a representation of the sequence. Self-attention has
used successfully in a variety of tasks including reading comprehension, abstractive summariz
textual entailment and learning task-independent sentence representations [4, 27, 28, 22].
0.52 - .                        .              .






     Full attentions for head 5. Bottom: Isolated attentions from just the word ‘its’ for att
     Figure 4: Two attention heads, also in layer 5 of 6, apparently involved in anaphora res
0.5 - <EOS>
```

Figure 3: An example of the attention mechanism following long-distance dependencies in the
encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant deper
the verb 'making', completing the phrase 'making...more difficult'.
0.5 - Each layer has two
sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, p
wise fully connected feed-forward network. We employ a residual connection [11] around each o
the two sub-layers, followed by layer normalization [1].
0.49 - 4 Why Self-Attention
In this section we compare various aspects of self-attention layers to the recurrent and conv
tional layers commonly used for mapping one variable-length sequence of symbol representation
(x1 , ..., xn) to another sequence of equal length (z1 , ..., zn), with xi , zi Rd , such
layer in a typical sequence transduction encoder or decoder. Motivating our use of self-atter
consider three desiderata.
0.47 - In the following sections, we will describe the Transformer, motivate
self-attention and discuss its advantages over models such as [17, 18] and [9].

3 Model Architecture

Most competitive neural sequence transduction models have an encoder-decoder structure [5, 2,
Here, the encoder maps an input sequence of symbol representations (x1 , ..., xn) to a seque
of continuous representations z = (z1 , ..., zn).
0.45 - . . .

sentence. We give two such examples above, from two different heads from the encoder sel
Figure 5: Many of the attention heads exhibit behaviour that seems related to the struct
0.44 - Operations

Self-Attention	O(n2 · d)	O(1)	O(1)
Recurrent			

0.43 - • The encoder contains self-attention layers. In a self-attention layer all of the key
and queries come from the same place, in this case, the output of the previous lay
encoder. Each position in the encoder can attend to all positions in the previous
encoder.

```
0.42 - As side benefit, self-attention could yield more interpretable models. We inspect atte
from our models and present and discuss examples in the appendix. Not only do individual atte
heads clearly learn to perform different tasks, many appear to exhibit behavior related to th
and semantic structure of the sentences.


5     Training

This section describes the training regime for our models.
> Running step ac9c9225-596e-4c84-8e86-1518a4fd7d55. Step input: None
Thought: The current language of the user is English. I was able to retrieve relevant informa
Answer: Self-attention, also known as intra-attention, is an attention mechanism that compute
Self-attention, also known as intra-attention, is an attention mechanism that computes a repr
```

This is not always feasible though.
Another way to use the retrieval-pipeline is to not give a weak model the opportunity to mess up the tool calling. This can be implemented by using a query-engine instead of the retriever. This directly wraps the retrieval in a LLM-Summarization-Module that only returns summaries.
Doing this, we can use two separate models for each part of the task - one for the planning and answering and one for the structured summarization:

```python
query_engine = index.as_query_engine(use_async=False, llm=fc_llm, verbose=True)
response = query_engine.query("What is the meaning of Query, Key and Value in the context of
print(str(response))
```

```
 In the context of self-attention, "Query" refers to the keys that are used to retrieve relev
```

Finally an answer we can work with!

Task

Build a llamaindex-application that allows you to chat with the climate_fever evidence.

Document chunking

The examples we looked at until now were all working with short text-snippets that comforably fit into the context window of a LLM. If you think about usual usecases for RAG-systems, this is not the most common case though. Usually, you will have a base of documents that can span multiple 1000's of tokens and you want to be able to answer questions about these documents. Furthermore, you do not only want to know which document might be relevant, but ideally also which part of the document matches your question best.

This is where the process of doctument chunking or document splitting comes into play. There is a series of possible approaches to split a document, the most common, so called **naive chunking** method, is to use a structural element of the document though. This means that you parse the documents into sentences, paragraphs or pages and then use these as chunks that you individually embed and store in your vector database. To prevent loss of relevant context when splitting a document into chunks, it is additionally common to add some **overlap** between the chunks. This tries to solve the lost context problem, does however create reduncencies in the data.

An alternative approach is to use **semantic chunking**. This means that you split a document into chunks based on their meaning. Jina.ai explained in a blogpost (*Late Chunking in Long-Context Embedding Models*, 2024) their so called "late chunking" method. which iteratively runs the whole document through the attention head of the transformer to gain embeddings per token, and then averages these embeddings per naive chunk. This way, the chunks are still structure based but contain semantic information about the whole context.

Another approach to semantic chunking is described on the docpages of LlamaIndex. In their approach to semantic chunking, an adaptive splitting-rule is used, that splits the documents based on semantic similarity of sentences. This means that sentences that are semantically similar are grouped together into chunks.

Task

Implement a document chunking strategy for a book of your choice from the project_gutenberg dataset.
You can use any approach you like, but you should explain your choice and why it is appropriate for this dataset.

Query Expansion/Transformation

Until now, we have based our retrieval on the assumption, that the question the user formulates is a good representation of their information need. This is not always the case though. Often, users do not know what they are looking for or they use synonyms or paraphrases that are not present in the documents. If the question is not formulated well, or if it is too specific, the system might not be able to find relevant documents. To improve the quality of the questions, we can use **query expansion**. This means that we take the original question and expand it with additional information to make it more specific and to increase the chances of finding relevant documents. This can be done in multiple ways, one common approach is to use a generative model to generate multiple queries based on the original question. Another approach is to use a keyword extraction algorithm to extract keywords from the question and then use these keywords to expand the query.

The most basic way to implement a query-expansion is to build a tool that instructs a LLM to give multiple alternate formulations of the original query. Though this will probably work, there are more refined methods.

Llamaindex implements two more sophisticated approaches to transform queries:

1. Hypothetical Document Embeddings (HyDe): A LLM is instructed to generate a hypothetical document that answers the query. This document is then used to query the index
2. Multi-Step Query Transformations: After a first execution of a (complex) query against an index, the answer is

used to iteratively formulate follow-up questions that are then executed against the index.

Task

Implement query expansion for the climate_fever dataset using llamaindex. This might be helpful.
Experiment with different prompts and temperatures.

Further Readings

- This blogpost by DeepSet gives a good overview of the concept of RAG
- This blogpost by qdrant about (their) vector store and its inner workings

References

Function Calling

Function calling is a technique used in large language models (LLMs) and AI agents to enhance their capability to provide more accurate and relevant responses, especially when handling complex tasks or questions that require specialized knowledge or external data.

We already got to know function calling in chapter 3 of this course. There, we introduced agents, that already came with the ability to call predefined functions. In this chapter, we will go back to the basics of function calling using LLMs.

Code generation and function calling

The basic idea of function calling is to use an LLM to generate valid, executable code from the user input. That is, the user's input is sent to the LLM, together with a prompt, urging it to return structured output in a specific format. This output can then be taken and executed. For this to work properly, of course, the generated output *must* be valid code (in our case python code). There are two approaches for that:

1. **Code generation**: Here, we ask the LLM to generate a complete python script from the user input. This approach has the advantage of being simple and straightforward, but it can be prone to errors if the LLM does not fully understand the task at hand or if it makes mistakes in its code generation. It can also pose a severe security issue because this approach hinges on running generated code on your machine.
2. **Function calling**: Here, we ask the LLM to generate a function call from the user input. This approach has the advantage of being more robust and accurate than code

generation, as it is easier for the LLM to generate a correct function call than a complete python script. However, it requires that the functions that can be called are already defined and that they are properly documented.

Here, we will focus on function calling. Still the challenge is to get the LLM to generate valid output. There are two main strategies to facilitate that:

1. using a large, generalized LLM (e.g. GPT-4) with good prompt engineering and
2. using a smaller model fine tuned to generate function calls.

Function definition

The first step in using function calling is to define the functions that the LLM can call. This is done by providing a JSON schema that describes the name of the function, its arguments and their types. The JSON schema should be provided to the LLM in the system prompt. Here is an example: [1]

```
{
    "name": "get_current_weather",
    "description": "Get the current weather in a given location",
    "arguments": {
        "location": {"type": "string"},
        "unit": {"type": "string"}
        }
}
```

Prompting

The second step is to provide a good prompt. The prompt should make it clear to the LLM to only generate valid output and that it should follow the JSON schema. Here is an example of a prompt that can be used for function calling:

[1]Note, that this is not an executable implementation but just a description of the function for the LLM.

```
You are a helpful assistant that generates function calls based on user input. Only use the fu

{function definition as described above}

User: What's the weather like in Berlin?

Assistant: {
    "name": "get_current_weather",
    "arguments": {"location": "Berlin", "unit": "celsius"}
}
```

Task

Try it!

1. Open a notebook and connect to a local LLM using LM Studio.
2. Define the function `get_current_weather` as shown above.
3. Write a prompt that asks the LLM to generate a function call based on user input.
4. Test the prompt with an example input.
5. Define other functions and try other inputs and see if the LLM generates valid output.
6. Upload to Moodle.

Challenges, finetuned models and the influence of size

The main challenge is here to get the LLM to generate a valid answer. This is not always easy, as LLMs are not usually super safe coders .

- They can hallucinate functions or arguments that do not exist.
- They can forget to call a function.
- They can forget to provide all required arguments.
- They can provide the wrong type for an argument.
- They can provide invalid values for an argument.

There are several strategies to mitigate these issues:

1. **Prompt engineering**: A good prompt can help to guide the LLM towards generating valid output. This is especially true for larger models, as they have a better understanding of the world and can therefore generate more accurate responses.
2. **Finetuning**: Finetuning a model on a specific task can improve its performance on that task. This is especially useful for smaller models, as they are less likely to hallucinate functions or arguments that do not exist.
3. **Size**: Larger models are better at generating valid output than smaller models. However, larger models are also more expensive to run and require more computational resources.

Task

Test it! (we can do it together, if your hardware does not allow you to run the model.)
As above, but this time

1. use a very small model (e.g a small Llama model)
2. use a model finetuned for the task (you could try this one)
3. a larger model (a larger llama in this case)

Agents recap

We introduced agents already back in chapter 3. To give a quick recap, an agent is a wrapper layer, that takes the user input and pipes it to an LLM, together with a custom system prompt, that allows the LLM to answer the user request better. The agent has several modules at its disposal, the memory, some tools and a planning tool.

The memory function is what allows chat models to retain a memory of the past conversation with the user. This information is saved as plain text in the memory and given to the planning module (i.e. the LLM) along with the system prompt and the current user input.

The planning module then decides which tools to use, if any, to answer the user request. The output of the planning module is a response message containing one or several tool calls (or a final answer). The agent then executes the tool calls by first parsing the response, then executing the functions. Based on the tool outputs, a final answer is generated and sent back to the user.

React agents

There a several types of agent. Now, we want to fucus on the ReAct agent introduced by (Yao, Zhao, et al., 2023). The ReAct agent is a type of agent that uses the ReAct framework to solve complex tasks by reasoning in multiple steps. It is based on the idea of "thought-action-observation" loops. The LLM is given a task and it generates a thought, which is then used to decide on an action. The action is executed and the observation is fed back into the LLM. This process is repeated until the LLM decides that it has enough information to answer the question or if the maximum number of iterations is reached.

Llamaindex

LLamaindex is a framework that makes it easy to implement and use agents. In Llamaindex an agent consists a an *agent runner* and an *agent_worker*. Think of the agent runner as the agent core in the architecture schematics and the agent worker as the planning module. The tools are functions implemented in python that can be executed by the agent worker. Finally, the memory module consists of a simple text buffer, logging the conversation history between the user and the agent and between the agent and the tools.

Task

Let's have a look!

1. Open a notebook and connect it with a local LLM using LM Studio.

2. Define a function that can be called by the agent to get the current weather in a given location. (Implement it this time, it doesn't need to work, just return random weather)
3. Initialize a ReAct agent using LLamaindex (you can use this tutorial as the starting point)
4. Have a look at the prompt, the agent gives to the LLM (you can find it using `agent.get_prompts()`)
5. Discuss the prompt with the group! What does it do? How does it do it?
6. Ask the agent to tell you the weather in a given location.
7. Watch in LM Studio how the LLM called by the agent creates the thought process, function calls and the final response.
8. Try to break the agent by asking stuff it cannot answer. Be creative. (On one occasion I just said "Hi" and it went into an infinite loop because it did not need a tool for that and there wasn't a "none" tool .)
9. Upload to Moodle

Further Readings

On the Llamaindex website on the "examples" page you will find a **lot** of helpful material: examples, notebooks, recipes and more. I recommend to have a look at them! For our case, check the "agents" section. For an even more in-depth dive, go to the "workflows" part.

- In this example you find an agent implementation that returns a step-by-step breakdown of its thought process.
- To go even more low level then that see this example that will walk you through setting up a Workflow to construct a function calling agent from scratch.
- Here is a *very* nice paper about generating structured output.

References

Agent interactions

In this chapter, we want to introduce multi agent systems. As a starting point, we will talk about LLM as a judge.

LLM as a judge

Before we introduce the concept proper, let us first describe the problem it tries to solve:

1. We generate text (be it natural language or structured output) using LLMs.
2. The generated text is not always correct or appropriate for our use case.
3. We need a way to evaluate the quality of the generated text.
4. To do this, we have to read it.
5. We don't have time for this.

The solution to this problem is, of course, to use an LLM to read and evaluate the text. This is only fair and proper, since it was an LLM that generated the text in the first place. The generated evaluation can then be used

- to decide whether to accept or reject the generated text.
- to improve the model itself (e.g., for fine-tuning it on the generated text and its evaluation).
- to get an LLM to improve the text based on the evaluation.

This approach is called LLM as a judge. It is a system that uses several calls to one or several LLMs to solve a problem. As such, it can be implemented as a multi-agent system.

This approach has a number of benefits as well as drawbacks.

- Benefits:
  - The evaluation can be very accurate and fast.
  - It is easy to implement.
  - It is easy to scale up the number of LLMs used for evaluation.
  - It is easy to use different LLMs for generation and evaluation.
  - It is easy to use different prompts for generation and evaluation.
- Drawbacks:
  - The evaluation can be very expensive, since it requires several calls to the LLM.
  - The evaluation can be biased, since it is based on the LLMs' own evaluation of itself. Indeed many LLMs tend to like their own creations.
  - The evaluation can be subjective, since it is based on the LLMs' interpretation of the prompt.
  - The evaluation can be misleading, since it is based on the LLMs' interpretation of the generated text, which may not be the same as the human interpretation. For example, many LLMs seem to prefer long answers over shorter ones.

A basic multi-agent system

Let us now look at a simple example of a multi-agent system. We will use the following scenario: We want to generate Anki flashcards from text.[1]

To do this, we will build a multi-agent system that consists of three agents:

1. An Anki card generator that generates Anki flashcards from the extracted text.
2. A Reviewer, that reviews the generated Anki flashcards and gives tips on how to improve them.

[1] The following is loosely based on "Building a Multi-Agent Framework from Scratch with LlamaIndex" (2024), though I took the liberty to streamline and simplify the code a bit.

3. An Editor, that generates a new set of Anki flashcards based on the reviewer's feedback.
4. An Orchestrator, that serves as the decision maker, managing the other agents and deciding when to stop.

We could also add more specialized agents, like a fact checker agent, that checks the generated cards for factual correctness, a translator that translates either the input text or the generated cards, or a topic analyzer that breaks down down complex topics into manageable parts before card generation.

Generator

Let us first implement the Anki card generator. It will take a text as input and return a card. A system prompt for the generator could look like this:

```
You are an educational content creator specializing in Anki flashcard generation.
Your task is to create one clear, concise flashcards following these guidelines:

1. The card should focus on ONE specific concept
2. The question should be clear and unambiguous
3. The answer should be concise but complete
4. Include relevant extra information in the extra field
5. Follow the minimum information principle

Format the card as:
<card>
    <question>Your question here</question>
    <answer>Your answer here</answer>
    <extra>Additional context, examples, or explanations</extra>
</card>
```

We will use llamaindex to implement the generator.

Task

You can do it!

1. Open a notebook and connect it with a local LLM using LM Studio.

2. Initialize a generator agent without any tools. Do not use the ReAct agent this time, a simpler OpenAIAgent will do.
3. Discuss: is it still an agent, if it does not have tools? Ask an LLM about its opinion on that .
4. Let it generate cards from the text below.

   LLM-as-a-Judge is an evaluation method to assess the quality of text outputs from any LLM-powered product, including chatbots, Q&A systems, or agents. It uses a large language model (LLM) with an evaluation prompt to rate generated text based on criteria you define.
5. Evaluate the results.

Reviewer

Let us now implement the reviewer. It will take a card as input and return feedback on how to improve it. A system prompt for the reviewer could look like this:

```
You are an expert in educational content creation, specializing in Anki flashcard generation.
You are the Reviewer agent. Your task is to review an Anki flashcard based on the following rul

1. The card should test ONE piece of information
2. The question must be:
    - Simple and direct
    - Testing a single fact
    - Using cloze format (cloze deletion or occlusion) when appropriate
3. The answers must be:
    - Brief and precise
    - Limited to essential information
4. The extra field must include:
    - Detailed explanations
    - Examples
    - Context
5. information should not be repeated, i.e. the extra information should not repeat the answer
```

```
Please give brief and concise feedback to the card you received in natural language.
```

Task

Let's build us a very judgemental robot!

1. In the same notebook, initialize a reviewer as well.
2. Let the reviewer review the cards generated by the generator. You may find that the reviewer always thinks the cards are great. This happens a lot. So:
3. Get the reviewer to actually find stuff to improve.

Editor

Let us now implement the Editor agent. It will take a card and feedback as input and return a new card based on the feedback. A system prompt for the second generator could look like this:

```
You are an expert in educational content creation, specializing in Anki flashcard generation.
You are the Editor agent. Your task is to generate a new Anki flashcard based on the original 
Follow these guidelines:

1. Incorporate the feedback into your new card
2. The new card should still focus on ONE specific concept
3. The question should be clear and unambiguous
4. The answer should be concise but complete
5. Include relevant extra information in the extra field
6. Follow the minimum information principle
7. If no feedback is provided, return the original card
8. Format the card as:

<card>
    <question>Your question here</question>
    <answer>Your answer here</answer>
    <extra>Additional context, examples, or explanations</extra>
</card>
```

Task

You have been edited!

1. In the same notebook, initialize the editor as well.
2. Let the editor generate new cards based on the feedback from the reviewer.
3. Get the editor to actually generate something that is different from the generators version! (Play around with models, prompts and/or input text. In this example, this only worked for me when using a weaker model as a generator and a larger one as reviewer and editor.)

Orchestrator

While we're at it, we can implement the orchestrator as well. Let us think for a moment what the orchestrators job should be. Its task should be decision making. That is, it's the orchestrators job to decide which of the other agents to call next. It is also responsible for deciding whether the job is finished or not, i.e. whether to call any more agents. In terms of input and output, the orchestrator should get a current state of affairs along with the current chat history and output a decision. So the output can be one of the other agents or a stop signal.

An example prompt for our case would be:

```
You are the Orchestrator agent. Your task is to coordinate the interaction between all agents

Available agents:
* Generator - Creates flashcards
* Reviewer - Improves card quality
* Editor

Decision Guidelines:
- Use Generator to create cards
- Use Reviewer to generate feedback
- Use Editor to improve cards based on feedback.
- Choose END when the cards are ready
```

```
Output only the next agent to run ("Generator", "Reviewer", "Editor", or "END")
```

Workflow

Now, all we have to do is integrate our agents into a pipeline. The basic idea is to call the orchestrator at each step and let it decide which agent to call next or wether to stop. For this, the agent will need an understanding of the basic state of affairs and the past interaction. This is easily implemented like this:

```
state = {
    "input_text": text,
    "qa_card": "",
    "review_status": "pending",
    "edit_status": "pending"
    }

memory = ChatMemoryBuffer.from_defaults(token_limit=8000) # using LLamaindex here
```

The memory can be read using the `memory.get()` method.

Then we define our workflow as an iterative process. Below is a piece of pseudocode illustrating the basic idea:

```
# pseudocode
initialize:
    generator
    reviewer
    editor
    orchestrator
    state
    memory

while true
    send state and memory to orchestrator -> response
    if response == "end"
        stop
    if response == "generator"
        send input text to generator -> card, change state and memory
    (same for the other agents)
return state
```

Task

Time to play!

1. In the same notebook, initialize the orchestrator as well.
2. Implement the workflow shown above in real code.
3. Watch happily as it all works without any issues whatsoever.
4. Upload to Moodle.

What we did not cover but what would be a great idea:

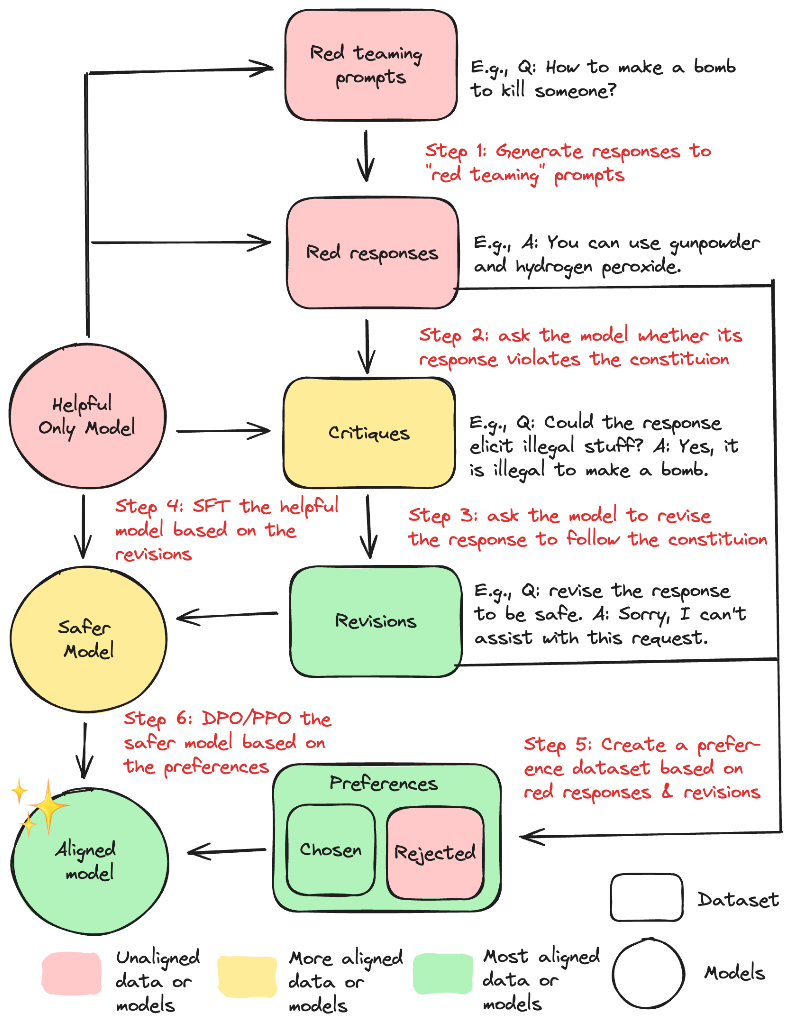
- Right now, we just assume that generator and editor return valid output. It would be better to build an automated check using a pydantic class for that.
- We let the orchestrator agent decide for how long this process lasts. <sarcasm>I cannot imagine that leading to problems under any circumstances.</sarcasm> It would be better to give it a timeout or maximal number of iterations.

Constitutional AI Tuning

One application of a multi-agent system is Constitutional AI.

Constitutional AI (*Constitutional AI with Open LLMs*, n.d.) is a method for fine-tuning language models that allows us to specify constraints and rules that the model should follow. It is based on the idea of a "constitution" that specifies the rights and duties of the model. The constitution is then used to guide the model's behavior during training and inference. This is done by adding an additional loss term to the training objective that penalizes the model for violating the constitution. The constitution can be specified in a variety of ways, including natural language, formal logic, or programmatic code.

Constitutional AI has been used to improve the safety and reliability of language models in a variety of applications, including chatbots, question-answering systems, and text generation. It has also been used to improve the fairness and transparency of language models by specifying constraints on the types of information that they can access or generate.



Red teaming prompts
E.g., Q: How to make a bomb to kill someone?
Step 1: Generate responses to "red teaming" prompts
Red responses
E.g., A: You can use gunpowder and hydrogen peroxide.
Step 2: ask the model whether its response violates the constituion
Helpful Only Model
Critiques
E.g., Q: Could the response elicit illegal stuff? A: Yes, it is illegal to make a bomb.
Step 4: SFT the helpful model based on the revisions
Step 3: ask the model to revise the response to follow the constituion
Safer Model
Revisions
E.g., Q: revise the response to be safe. A: Sorry, I can't assist with this request.
Step 6: DPO/PPO the safer model based on the preferences
Step 5: Create a preference dataset based on red responses & revisions
Preferences
Chosen
Rejected
Aligned model
Dataset
Unaligned data or models
More aligned data or models
Most aligned data or models
Models

Fig 1: Illustration of the CAI training process (from *Constitutional AI with Open LLMs* (n.d.))

The basic idea is to define a "Constitution" that specifies the rules and constraints that the model should follow. These could be rules like

1. The model should not generate harmful or inappropriate content,
2. The model should not engage in offensive or derogatory behavior,
3. The model should not disclose sensitive information about users without their consent, etc.

The way it works is as follows:

1. A malicious user sends a prompt to the model. The prompt may be designed to elicit harmful or inappropriate behavior from the model, such as "how can I build a bomb?". The model, being a helpful AI agent, generates a response that violates its constitution. For example, it might provide instructions for building a bomb.
2. The model is asked if its answer violates the constrains defined in its constitution. In our case, we might conclude that bomb building instructions can indeed lead to harm and thus violate the constitution.
3. The model is asked to revise its answer based on the constitution. In this case, it might generate a response like "I'm sorry, but I cannot assist with that request as it goes against my programming." While we could stop here and use the revised response as our final output, we can also take this one step further:
4. Create a training set from the original prompt, the original answer, the constitution, and the revised answer. This training example can then be used to fine-tune the model so that it learns to avoid violating the constitution in the future.

This technique was used, for example, in the training of the "Claude" model (*Constitutional AI*, n.d.).

Further Readings

- Here is a video describing other multi-agent systems, including an agent hospital and a multi-agent translator

References

Image Generation

AI image generation

This and the following chapters will focus on the topic of AI image generation. This is a very broad field, so we will start with some basics and then move on to more specific topics. We will also try to give an overview of the current state of the art in this field.

AI image generator basics

You can not talk about the history of AI image generation without talking about GANs (Goodfellow et al., 2014). To have a nicer chunking of the courses contents though, we will talk about them in the chapter Chapter and focus on more recent approaches here. GANs are the architecture behind the page thispersondoesnotexist.com and its clones.

DALL-E

The reigning position of GANs as the de-facto standard for AI image generation was challenged by the release of DALL-E by OpenAI in January 2021. DALL-E is a text-to-image model, which means that it can generate images based on a text description.

This model was trained on a dataset containing image-caption pairs in two parts:

1. A Variational Autoencoder (VAE)[1] to compress the image data into a latent space. This means, that each image was

[1]Since the latent space these images are compressed to is of a defined set of classes, the authors call the model a discrete VAE which makes a lot of sense.

compressed into a 32x32 grid, for which each grid cell was encoded as a discrete probability distribution with 8192 dimensions. This latent "token"-space is, although the architecture is pretty different, quite close to what our text-transformers outputted in the MLM-task.

2. A Transformer to learn the relationship between text-captions and the latent space. This was done by encoding images using the pretrained VAE und argmax choosing the 32x32-token-representation of the image. The text-captions were limited to 256 tokens and concatenated with the 1024-dimensional image-tokens. The model is then trained to predict the next token in the sequence, which is either a text or an image token, similarly to the learning-paradigm we discussed when talking about the transformer-training.

The resulting 1024 image-tokens can then be fed into the decoder-Block of the VAE to generate an image. An illustration of the training-process can be seen in Figure 1.

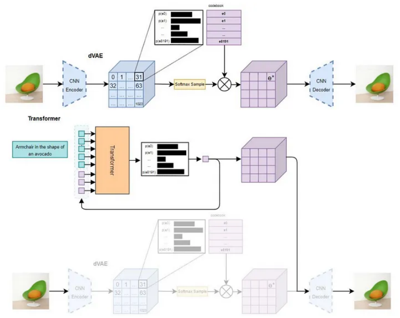


Fig 1: Illustration of the DALL-E-VAE (A) and Illustration of the whole DALL-E-Stack (B). Both images are taken from Abideen (2023).

CLIP

Close to the release of DALL-E, the team at OpenAI did also publish CLIP (Radford et al., 2021). The paper, which introduced a contrastive[2] method to learn visual representations from images and text descriptions, bridged the gap between image and text embeddings. This contrastive principle is illustrated in Figure 2.

A matrix of all combinations of images and text descriptions is created. The model then learns to predict the correct image for a given text description and vice versa. This is done by encoding both the image and the text into a vector space, which is then used to calculate the similarity between the two vectors. to do this, both a vision- and a text-transformer are trained as encoders to maximize the cosine similarity between the encoded image and text for each pair in the matrix and minimizing it for all other pairs. The authors also show that this method can

[2]Contrastive also being the namesake of the method (Contrastive Language-Image Pre-training)

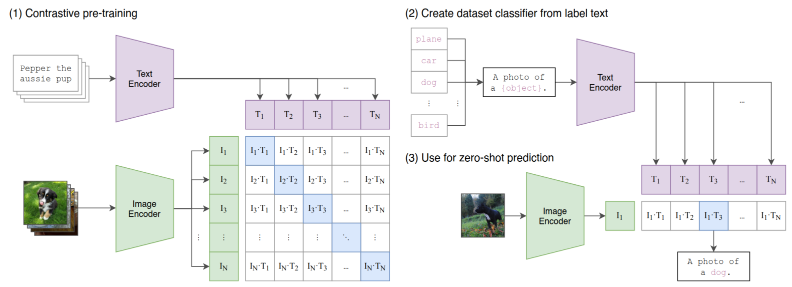


Fig 2: Illustration of the contrastive learning paradigm used in CLIP, taken from Radford et al. (2021)

be used to transfer the learned representations to other tasks, such as zero-shot classification.

Diffusion Models

Though models like DALL-E and CLIP represented significant milestones in the journey of text-to-image generation, the field continued to evolve rapidly, leading to the advent of Stable Diffusion. This evolution was partly inspired by the need for more control over the generation process and a desire for higher-quality outputs at lower computational costs.

The GAN-architecture (first published in 2014) was the de-facto standard for quite some time and though the central principle of their successors diffusion models was published in 2015 (Sohl-Dickstein et al., 2015), it took until 2020 for them to beat GANs on most benchmarks (Dhariwal & Nichol, 2021).

The diffusion model's central principle is training on a sequence of gradually noised images. This process involves systematically adding noise to an image over a series of steps, progressively transforming the original image into pure noise. The model is trained to reverse this process by predicting the noise added to each image, based on the current step in the noising sequence and the noisy image itself.

This step-by-step noise addition serves two main purposes:

- **Gradual Complexity:** By progressively corrupting the image, the model can learn to reverse the process in manageable steps, leading to a better understanding of how to reconstruct data at each stage.
- **Mathematical Framework:** This approach aligns with the stochastic differential equation (SDE) framework, enabling the model to map the noise distribution back to the original data distribution iteratively.

This approach, rather than predicting the denoised image directly, also offers practical advantages: it allows for efficient parallelization during training since the noise is parameterized by a scheduler and can be applied dynamically. This stepwise noise-addition is visually represented in Figure 3.

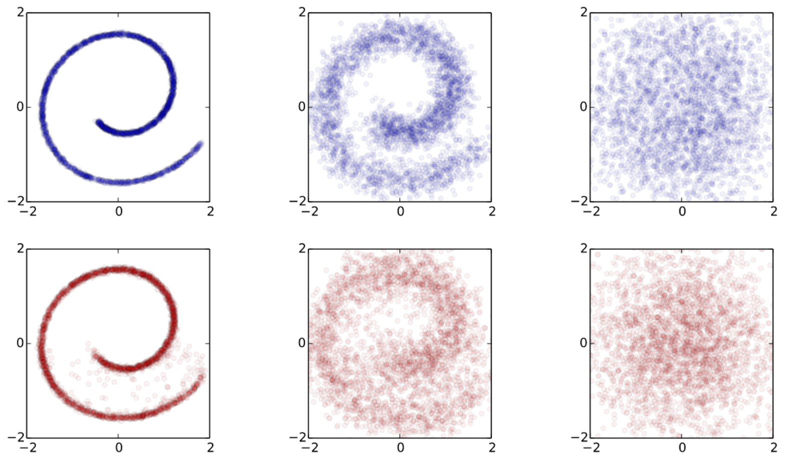


Fig 3: Illustration of the diffusion process. The first row shows a 2-d swiss roll gradually getting more noisy, the second row shows the corresponding outputs of the diffusion model. Image taken from Sohl-Dickstein et al. (2015).

Rombach et al. (2022) build upon this principle when suggesting their Latent Diffusion Model architecture and introduced a few key innovations to achieve their state-of-the-art results:

- They introduce a method called latent diffusion, which allows them to generate high-resolution images more efficiently by operating on a lower-dimensional representation of the image data. This is achieved by using an autoencoder (VAE) to compress the original image into a smaller latent space and then applying the diffusion process to this compressed representation. This process is built on work by Esser et al. (2021) and is conceptually similar to the dVAE-approach utilized by DALL-E.

- They use a denoising diffusion probabilistic model (DDPM) as the fundamental generation process for their architecture, which allows them to generate high-quality images with fewer steps compared to previous methods. This DDPM model is implemented as a time-conditional UNet.
- To improve the quality of generated images and reduce artifacts, they integrate a cross-attention mechanism into the UNet architecture. This mechanism conditions the denoising process directly on the input text embeddings, allowing the diffusion process to generate images that align better with the given text prompt.

To improve the results on inference, they additionally utilize classifier-free guidance (Ho & Salimans, 2022), a technique where the model is run once with the prompt ("conditional on the prompt") and once with an empty pseudo-prompt ("unconditional"). A weighted combination of the conditioned and unconditioned predictions is used to enhance the alignment with the text prompt while preserving image quality. This is done using the following formula:

$$\text{Guided Prediction} = \text{Unconditioned Prediction} + w \cdot (\text{Conditioned Prediction} - \text{Unconditioned Prediction})$$

Where w is the weight with which the conditioned prediction is preferred over the unconditioned one.

This architecture has been widely adopted and is used as a foundation[3] for many state-of-the-art text-to-image models, including Stable Diffusion, as well as DALL-E 2.

[3]Or at least as an orientation.

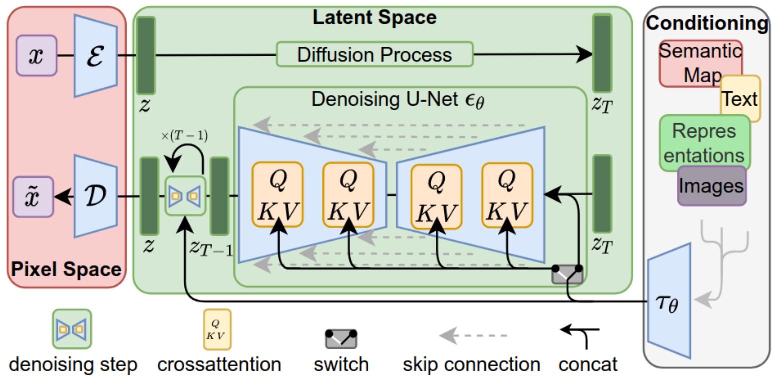


Fig 4: Illustration of the Latent Diffusion Model architecture. Image taken from Rombach et al. (2022)

Task

Test out a SD-model!
Use the colab-Notebook you can find here to test out Stable Diffusion using the huggingface `diffusers`-module and generate some images.

1. Print out the model architecture and try to map the components of the model to the description above.
2. Generate some images using different prompts, guidance_scales and seeds. What do you observe?
3. There are *many* pages with tips on how to "correctly" prompt SD-models to improve their performance. Find one and test the described tips out. What do you find?
4. Test the `num_inference_steps`-parameter. How does it affect the quality of the generated image?

Multimodal Models

So called multimodal models are models that are trained to fit one latent distribution for multiple modalities. This means that instead of only using the image encoder and decoder with some kind of informed diffusion model to generate images in between, encoders and decoders for multiple modalities are trained to map onto the same latent space. This results in a family of models that can take inputs in multiple modalities and create outputs in a similar fashion. There are different approaches to solve this task, of which two will be discussed in the following section

Unidiffuser

One of the first multimodal models is Unidiffuser, an architecture described in Bao et al. (2023). The architecture is illustrated in Figure 5.

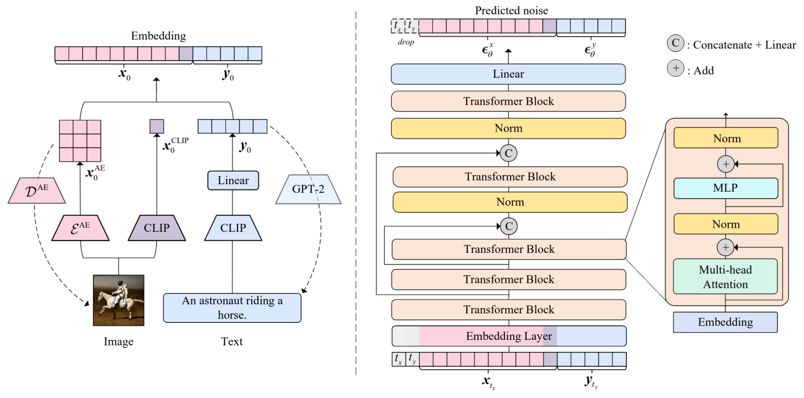


Fig 5: Illustration of the Unidiffuser architecture. Image taken from Bao et al. (2023)

The model is based on a transformer-encoder and decoder that are trained to map inputs of multiple modalities onto the same latent space. In the text-image implementation, there are two encoders and two decoders. The image encoder consists of two parts. One is the VAE-encoder from Stable Diffusion, which maps the input image into a lower dimensional representation. This is appended by the CLIP-image-embedder described in Radford et al. (2021). The text gets also encoded by the CLIP-trained model used in Stable Diffusion.

For image-decoding, the Stable Diffusion VAE-decoder is used to map the latent space back into an image. For text-decoding, a GPT-2-based (Radford et al., 2019)model is finetuned to take the latent space embeddings as a prefix-embedding and to autoregressively generate text. During finetuning, the CLIP-embeddings were held constant and only the GPT-2-parameters were updated. This means that the already defined latent space learned by the CLIP-model is used to map the GPT-2 decoder onto it.

These embeddings are then used to train a U-ViT (Bao et al., 2022) model, which takes the concated time-step-tokens, noised text- and image-embeddings as input-tokens and outputs the estimated noise-vector for the denoising process.

Task

Use the same colab-notebook as before to test out Unidiffuser using the huggingface `diffusers`-module and generate some images and text.
Try the tips you tested on the basic SD-model and test whether the model accurately generates descriptions for your generated images.
Present your results of both tasks to the course and upload your adapted notebook to moodle.

Llama 3.2

Llama 3.2 introduced image-understanding to the Llama-model family. Similarly to the decoder-training in the unidiffuser case, this was done by mapping existing embeddings onto a new latent space. Instead of finetuning a part of the model on a constant other embedding though, Meta describes a slightly different approach in their launch-blogpost ("Llama 3.2," n.d.).

They describe a procedure in which they use both a pretrained image encoder as well as a fixed pretrained language model. The embeddings of both models are aligned using a special third adapter model, that builds on multiple cross-attention layers to map the encoded image onto the language models text-embedding space. The encoder and adapter were then trained using image-text pairs to correctly generate the text-labels for the images.

Further Reading

- This blogpost about the reparametrization trick
- This Medium-article about how the first DALL-E worked
- The tutorial-paper by Doersch (2021) about the intuition and mathematics of VAEs
- Computerphile did some very nice videos about SD and CLIP

References

AI image generation II

In AI Image Generation I we mentioned GANs without going into details. In this chapter, we will take a closer look at them. We will also briefly touch on image augmentation.

Generative Adversarial Nets (GAN)

Generative Adversarial Nets, as first proposed by Goodfellow et al. (2014), are a class of generative models that can be used to generate new data samples from a given dataset. They consist of two components: a generator and a discriminator. The generator takes random noise as input and tries to produce realistic-looking data samples, while the discriminator takes data samples as input and tries to distinguish between real and fake samples. The generator and discriminator are trained simultaneously in a game-theoretic framework, with the goal of minimizing the difference between the distribution of real and fake samples. To use the authors own words:

> "The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistiguishable from the genuine articles."

While the rest of the paper goes into mathematical depth and is not really recommendable for the casual reader, the basic concept behind it is surprisingly simple. The following figure illustrates the concept:

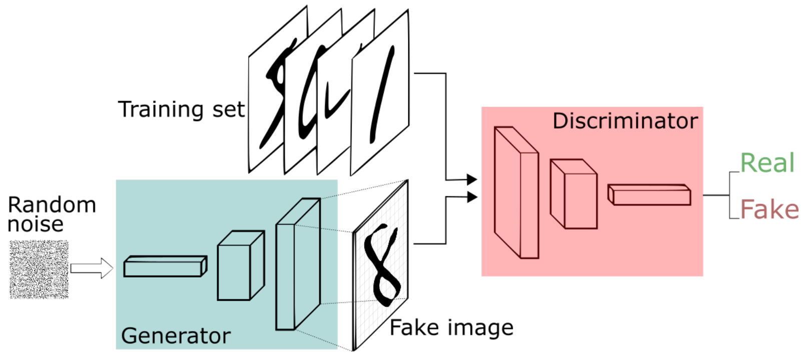


Fig 1: GAN architecture shown here for a model trained on the MNIST dataset, from *PyTorch GAN Basic Tutorial for Beginner* (n.d.)

The generator is usually fed with noise, that is then transformed into a latent space, comparable with the embeddings, we talked about earlier. This latent vector is then passed through the generator network to generate an image. So, in this framework, the generator is analogue to the decoder of a VAE and the discriminator is analogue to the encoder, transferring the input data into a latent space and then using a classification head to decide whether the input is real or fake.

Usually, GANs make heavy use of convolutional neural networks (CNN) in both the generator and discriminator part, but in principle they can use any architecture. Additionally, while they were developed in the context of image generation, they are not limited to this domain and have been used for text generation as well.

Challenges

While GANs have shown promising results in various applications, they also come with their own set of challenges. Some of these include:

- They tend to be unstable in training, often requiring careful tuning of hyperparameters and training techniques to achieve good performance. One possible solution is to first train on smaller images and then later in the training process scale up the size of the images.

- If the discriminator is too bad early on, a situation can emerge where, by accident, one or a few classes of possible generated output perform better than others. This can lead to **mode collapse**, where the generator only produces samples from this class and ignores all other classes. In the example of the MNIST dataset, it could learn to only produce images of the number 5. In the original paper, this is referred to as the "helvetica scenario".[1] To avoid mode collapse, often the discriminator is trained more often then the generator to make it better. However, this can lead to the following problem.
- The generator and discriminator can get stuck in a state where the generator produces low-quality samples that are easily distinguishable from real data, while the discriminator becomes too good at distinguishing between real and fake samples. In this case, it will be very hard for the generator to improve its performance over time. This is often referred to as vanishing gradients. To avoid this, techniques like Wasserstein GANs (WGAN) have been proposed, which use a different loss function that can help stabilize training and prevent mode collapse.
- They can be computationally expensive to train, especially when dealing with high-dimensional data such as images.

Variants of GANs

There are many variants of GANs that have been proposed in the literature to address some of these challenges and improve their performance. Some examples include:

- Deep Convolutional Generative Adversarial Networks (DCGAN), which use convolutional layers in both the generator and discriminator to generate high-quality images.

[1]Apparently, this is a reference to a british parody science show, see here.

- Wasserstein GAN (WGAN), which uses the Wasserstein distance as a loss function instead of the traditional cross-entropy loss to improve stability and convergence properties.
- StyleGAN, which uses a novel architecture that allows for fine-grained control over the style and content of generated images. It also introduces a new technique called style mixing, which allows for the creation of new styles by combining existing ones.
- BigGAN, which uses a large batch size and spectral normalization to improve stability and convergence properties.
- Progressive Growing GAN (PGGAN), which gradually increases the resolution of generated images over time to improve quality and stability.
- CycleGAN, which uses a cycle consistency loss to enable unsupervised image-to-image translation between two domains without the need for paired data.
- StarGAN, which enables unsupervised image-to-image translation between multiple domains by learning a single mapping function that can transform images from one domain to any other domain.

Task

Train one yourself!
(or, at least, try it. GANs are notoriously bad to train, also there are hardware concerns.)

- implement a simple GAN architecture in pytorch (you can use this noteook on kaggle) and train it on the MNIST dataset
- Have a look at this GAN zoo implemented in pytorch. Find one that might be interesting for your use case.
- (optional) train that one on this or another dataset (see the pytorch vision dataset page for datasets already implemented in pytorch.)

No need to upload to Moodle this time.

(Generative) approaches for image dataset augmentation

Image augmentation is used to generate more training images a limited number of original training images. This can help improve the performance of machine learning models by increasing the size and diversity of the training data, which can help prevent overfitting and improve generalization. Image augmentation techniques can be applied during the preprocessing stage of the machine learning pipeline, before the data is fed into a model for training.

Classical image augmentation

There are many different image augmentation techniques that can be used to generate new training images from existing ones. Some common techniques include:

- Random cropping and resizing: This involves randomly selecting a region of an image and resizing it to a fixed size, which can help improve the robustness of models to variations in object scale and position.
- Flipping and rotation: These simple transformations can help increase the amount of training data by creating new images that are similar but not identical to the original ones.
- Color jittering: This involves randomly adjusting the brightness, contrast, saturation, or hue of an image, which can help improve the robustness of models to variations in lighting and color.
- Elastic transformations: These involve applying a series of small, random deformations to an image, which can help increase the amount of training data by creating new images that are similar but not identical to the original ones.
- Cutout: This involves randomly masking out a region of an image with a fixed size and filling it with a constant value (e.g., black or white), which can help improve the robustness of models to occlusions and other types of noise.

- Mixup: This involves combining two images in a weighted manner, along with their corresponding labels, to create a new image and label pair. This can help increase the amount of training data by creating new examples that are intermediate between existing ones.

Most of these are already implemented in pytorch's torchvision.transforms.v2 module.

Task

Let's have a look!

- Have a look at the datasets in the pytorch vision dataset page and find one that might be interesting for you.
- Load that dataset with pytorch's `DataLoader` class, apply some transformations to it using the torchvision.transforms.v2 module and visualize some of the results.

Generative image augmentation

GANs can be used for image augmentation as well (D. Liu & Hu, n.d.). They can generate new images that are similar to the original ones but not identical, which can help increase the size and diversity of the training data. GANs can be trained on a dataset of real images, and then used to generate new images by sampling from the latent space of the generator network. The generated images can then be added to the training set to improve the performance of machine learning models.

There are, of course, techniques other than GANs to augment existing image datasets using generative models. One example are diffusers, we talked about last time. Trabucco et al. (2023) make the case for using these for data augmentation.

In the following, we will introduce some types of image augmentation using diffusers, without claiming this to be an exhaustive list.

Inpainting

When using inpainting, a section of the image is masked and then the model is prompted to fill the gap. This is most often used to remove unwanted content from images.

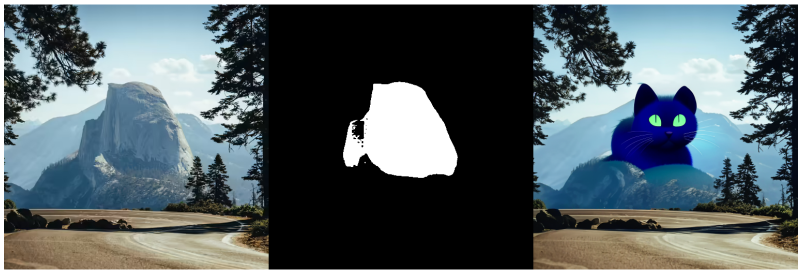


Fig 2: Inpainting, from the example on huggingface.

Image to image

In this case, an image is given to the model in addition to the prompt, conditioning the model to generate a specific output. This can be used to generate images from sketches or change the artistic style of a painting.

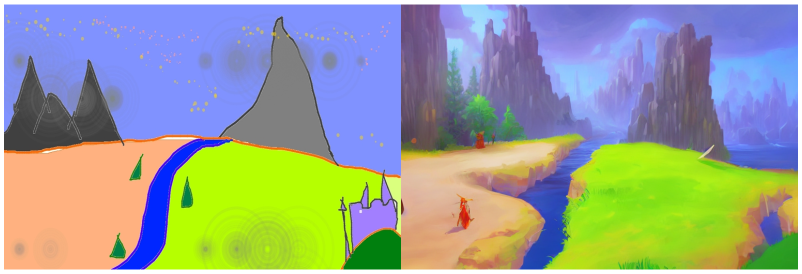


Fig 3: Image to image generation, from huggingface.

Another way of generating an image from another image is to first generate a description from an image and then using it as a prompt to generate another image. Hopefully, the second image will be similar to the initial image.

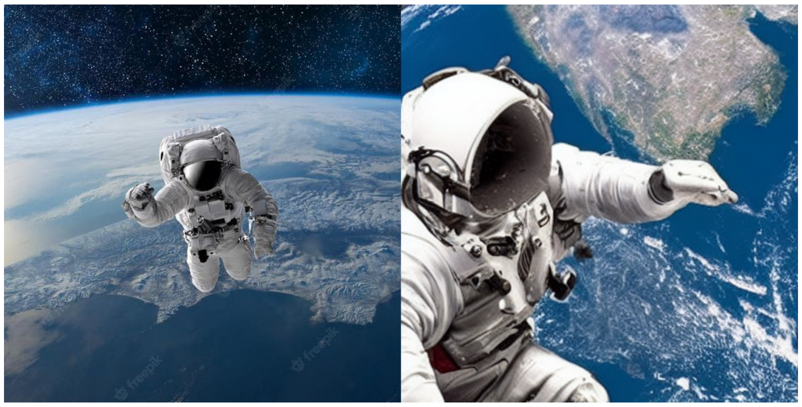


Fig 4: Image to text to image generation, also from huggingface.

Image variation

There is also a version of stable diffusion on huggingface that is finetuned on image variation. At first glance the result is underwhelming, but give it a shot!

ControlNet

Another type of image-to-image generation is ControlNet. Here, you would typically give the model a prompt and in addition an sketch, human pose or canny edge to condition the model. In the example given below, a canny sketch is made from a painting, then a new painting is generated based on the canny sketch and a prompt detailing the desired image (in this case “Mona Lisa”)

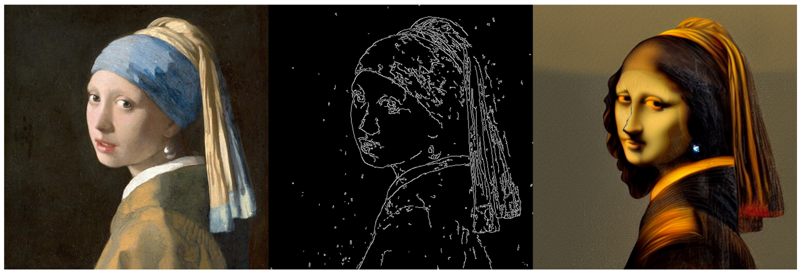


Fig 5: Example for ControlNet, again from huggingface.

Task

Give it a go!

- Open a notebook, locally or on google colab.
- Test the generative image augmentation techniques and models introduced above.
- Upload your notebook to Moodle.

References

AI image generation III

Basics of using Open Source AI image generation models

One of the challenges of using image generation models is the required computational power and the fine-tuning effort needed to obtain high quality images. This can be a significant barrier for individuals or smaller organizations that may not have access to large computing resources. We will cover finetuning next time. This time we want to focus on using image generation models locally.

For large language models, we used mainly LM Studio to run the models on our laptops. Image generation models, however, do not run in LM Studio as of 2024. Additionally, there is no real equivalent for image generation models. There is, however, a tool that makes running image generation models locally more convenient: AUTOMATIC1111's Stable Diffusion web UI.

Task

Let's have a look!

- Install AUTOMATIC1111's Stable Diffusion web UI on your laptop using these instructions.
- Start the server, open the webUI.
- Start generating images.
- Change some of the settings and see what happens.
- What does the `Sampling steps` parameter do?

This tool surely does make image generation more convenient. Most of the time, however, we do not want to deal with a web UI, but with an API endpoint. Fortunately, A1111's webUI also has an API mode, which is quite easy to use and supports all features of the web UI (and some more). We are mostly interested in the `txt2img` API endpoint, which allows us to generate images from a text prompt. Let's have a look at how this works:

Task

- Open the documentation of the API.
- Run the web UI in API mode.
- in a notebook, run an example call to the `txt2img` endpoint.

We now know how to easily generate images using a local model. The next steps would be to try different models, and to add Lora (or other) adapters to them.

AI image generators in agent systems or pipelines

In this section we want to explore the use of AI image generators as components in an agent system or a pipeline. An example for this might be a system that takes a few keywords, generates a text from it and then uses a language model to generate an image generation prompt based on this text. This prompt is used to generate an image. The final image is then sent to some quality assurance system to check if the output matches the input (or at least makes sense).

We covered agent systems extensively already. This time we want to focus on building a language model pipeline instead. In this section, we will:

- generate or retrieve a text based on some input keywords.
- use this text as context for generating an image generation prompt.
- generate an image from the prompt.
- implement quality assurance by comparing the original text embedding with the generated image embedding.

Most agent frameworks we already introduced support building pipelines in addition to agents. See for example this tutorial on how to implement query pipelines in llamaindex or this documentation for pipelines in haystack. To get a full understanding of the basic principles, it is most educational to implement a pipeline from scratch.

Text generation or retrieval

The pipeline we are about to build starts with some input given by the user. In previous chapters we covered several ways of doing this. You could:

- use a local LLM to generate the text for you.
- use a retrieval function from a vector store or other text database.
- combine both approaches in a RAG system.

Task

Let's get started!

- Open a notebook and implement a simple text generation or retrieval function.
- Get a text from an input.

Image generation

The next step is to to generate an images that fits the text. While we could just send the full text to the image generator and let it do its thing, a better approach is to generate a special prompt for the image generator. This prompt is then used to generate an image.

Task

- In your notebook, implement a call to an LLM that generates in image generation prompt from your text.

- Also implement a call to an image generator.
- Connect to an LLM (if not already done so) and to an image generation model.
- Generate an image for your text.

Quality assurance

Now that we have the image, we want to assure that it fits the text. There are several ways of doing this. We could, for instance, evaluate text and images manually (or, rather, by eyeballing it). This works well for small amounts of images. However, it is not scalable for larger amounts.

One way of automating the examination is to check, if the image matches the text semantically, i.e. in meaning. One could translate the image back to text, using an image-to-text model. This description of the image can then be compared to the original text using embeddings and a suitable distance metric, e.g. cosine. Or we could embed both image and text using a multi-modal model and calculate the distance directly. On both cases, we need a predefined criterion, i.e. a fixed distance, that has to be reached to accept the image as good enough. Alternatively, we could generate several images and just chose the best matching one.

Task

Let's have a look!

- In your notebook, implement a function that displays text and image for manual inspection.
- Implement an automated similarity rater for text and images. You can use CLIP for that task.

Pipeline

Finally, we can wrap everything in a pipeline. The pseudocode below shows the general principle. this is shown here for generating a number of images and picking the best matching one, but it can easily be converted to generate images until a predefined matching criterion is matched.

```
## pseudocode
define pipeline(user_input):
    get_text(user_input) -> text
    generate_image_prompt(text) -> image_prompt
    for in in range 5:
        generate_image(image_prompt) -> image
        rate_image(image) -> rate_value
    find_best_rated_image(images, rate_values) -> best_image
    return best_image
```

Task

Let's finalize

- In your notebook, implement the pipeline outlined above.
- Make a few test runs.
- Upload your notebook to Moodle.

Finetuning

Finetuning Approaches

Finetuning in terms of generative models means the general concept taking a pre-trained, "foundational" model and updating its parameters using new data. This data is usually much smaller than the data used to train the original model. The goal is to adapt the model to the new data while preserving as much of the knowledge it has already learned from the original training data. We have already seen an example of a finetuning approach when we were talking about instruct-tuned models Section . These models are based on plain MLM-trained language models, that are then trained on new data that is presented in a Instruct - Response format. The result of this specific example of finetuning was a model that, instead of just completing a text, answered in the format present in the finetuning data.

Though the central concept of finetuning is always the same, i.e., updating the parameters of a pre-trained model using new data, there are many different ways to do this. The following sections will give an overview of some of the most common approaches.

Full Finetuning

Full finetuning is the simplest approach to finetuning. As the name says, it is based on completely updating the parameters of the pre-trained model using new data. This means that all weights of the model are updated during training using regular gradient descent or a variant thereof. The main advantage of this approach is that it is very simple and easy to implement. Complete (few-shot) fine-tuning has also shown to perform better in the domain of finetuning and in Out-of-domain tasks when compared to Few-Shot-Prompt-approaches Mosbach et al. (2023). However, it also has some disadvantages. Firstly, it can be computationally expensive as it requires training all parameters of the model.

Secondly, it can lead to catastrophic forgetting, i.e., the model forgets what it has learned during pre-training when adapting to new data (Luo et al., 2024).

Parameter-Efficient Finetuning (PEFT)

Another approach to finetuning is to not update all a models parameters but to (partially) freeze them and only update a small subset of the parameters or to train an adaptor module that can be added to the model. This approach is called parameter-efficient fine-tuning (PEFT). The main advantage of PEFT is that it is much more computationally efficient than full finetuning as it only requires updating a small subset of the parameters. We will look at three different approaches to PEFT:

1. Prompt-based Finetuning (Prefix-tuning and Prompt tuning)
2. Adapter-based finetuning (Low-Rank Adaptation and its relatives)
3. (IA)^3 (Infused Adapter by Inhibiting and Amplifying Inner Activations)

Prompt-based Finetuning

Prompt-based finetuning is a family of methods that use so called "soft-prompts" to guide a models generation. The general concept is pretty close to prompting as we discussed it in Chapter . The main difference is that instead of engineering a prompt constructed from discrete tokens that results in opportune results, we let standard optimization procedures find a continuos embedding-vector in a pre-trained LMs embedding-space. Prefix-Tuning, Prompt Tuning and P-tuning are three different approaches to prompt-based finetuning - all utilizing some implementation of this soft-prompt concept.

Prefix tuning

Prefix-Tuning (Li & Liang, 2021) is a method of adapting a language model to a specific down-stream task by adding a continuous prefix vector to the input embeddings. This is done by learning a continuos matrix with a set amount of columns (i.e., tokens) and the frozen models embeddings-dimensionality[1] that is prepended to the input of each transformer layer (i.e., the encoder and the decoder-stack). The principle is illustrated in Figure 1.

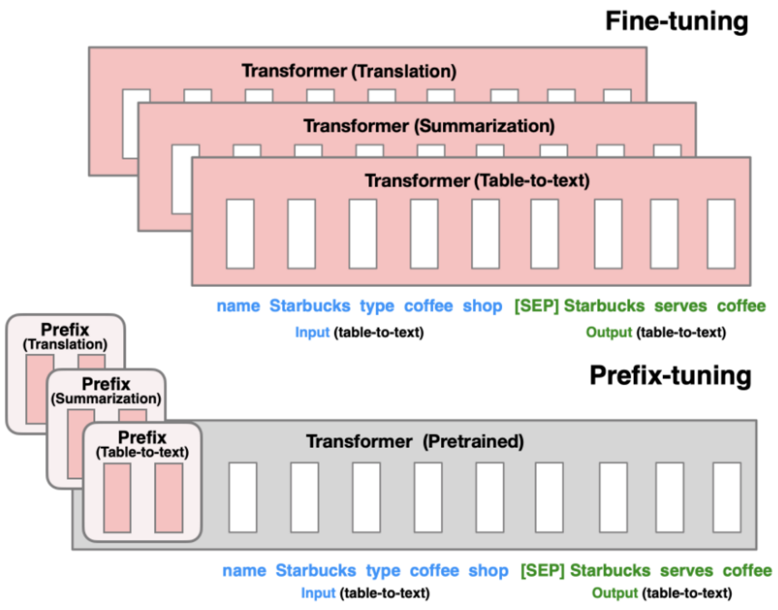

Fine-tuning
Transformer (Translation)
Transformer (Summarization)
Transformer (Table-to-text)
name Starbucks type coffee shop [SEP] Starbucks serves coffee
Input (table-to-text)
Output (table-to-text)
Prefix (Translation)
Prefix (Summarization)
Prefix (Table-to-text)
Prefix-tuning
Transformer (Pretrained)
name Starbucks type coffee shop [SEP] Starbucks serves coffee
Input (table-to-text)
Output (table-to-text)

Fig 1: Illustration of Prefix-tuning. A continuous prefix vector is learned and concatenated to the input embeddings before they are fed into the transformer layers. From Li & Liang (2021)

This vector can then be used to guide the model during inference. The main advantages of this method are

[1]Since directly learning the prefix-weights proved to result in unstable performance, the authors did not directly train prefix-vectors but a MLP scaling up from a smaller dimensionality to the embedding size. Since the rest of the proxy-model is discarded after training though, the method can be treated as the same principle.

a) a small number of parameters that need to be learned and
b) the ability to quickly adapt to different tasks by simply switching out the prefix vector.

Since the learned prefix-weights have to be prepended to each input though, one has to have access to the models internal representation during inference (at least for encoder-decoder-stacks). This is not always possible, especially when using black-box models like large language models that are hosted on a remote server.

Prompt-Tuning

Prompt-tuning (Lester et al., 2021) is a method that is conceptually very similar to prefix-tuning, but avoids the need for accessing the internal representation of the model during inference by using what the authors call "soft prompts". Again, instead of prompting using discrete tokens, continuous "special tokens" are learned that are concatenated to the input embeddings. The main contribution of Prompt-Tuning over Prefix-Tuning is a) that they showed that inputting the soft-prompts to the encoder alone suffices and more importantly b) that the performance of models fine-tuned in this manner is comparable to full finetuning, at least for larger LLMs (Figure 2).

Task

Your turn!
The huggingface-page on prompt-based finetuning describes three more variants of soft-prompt finetuning:

1. P-Tuning
2. Multitask prompt tuning
3. Context-Aware prompt tuning

Select one of the three and try to answer the following questions in a markdown-file:

1. What is the core principle?
2. What is the context in which this tuning method is most efficient?

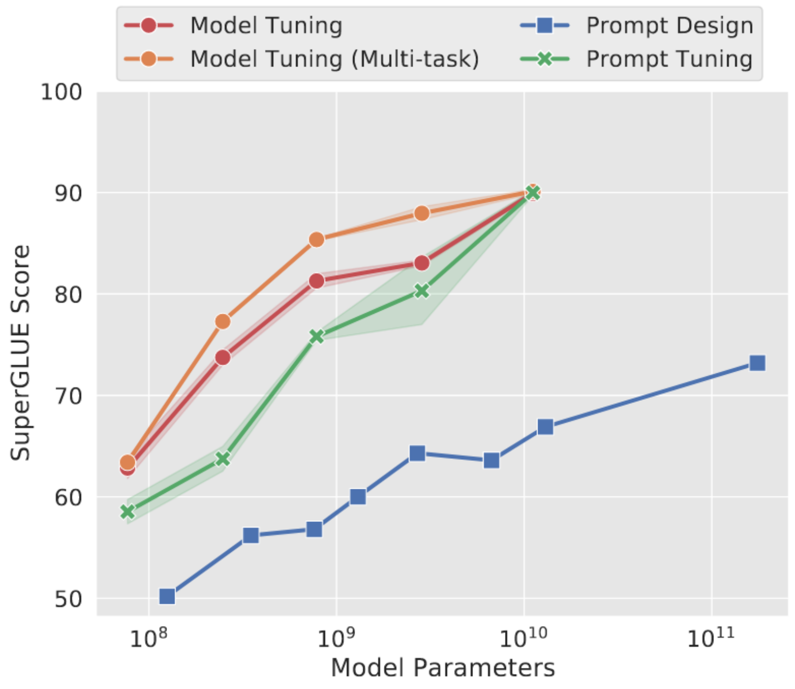


Fig 2: Results of Prompt-tuning compared to prompt-engineering and complete finetuning, taken from Lester et al. (2021)

3. How much memory can be saved by leveraging this technique (if you can find this indication)

Present your results to the group. Upload your results to moodle.

Adapter-based finetuning

Instead of focusing on the embeddings and thus the input of the language models, LoRA and its relatives focus on adapting the output of the attention and feed-forward layers of a transformer. The family of Low-Rank Adaptation (LoRA) methods (Hu et al., 2021) we will discuss here is a group of parameter-efficient fine-tuning techniques that adapt the models output by injecting trainable rank decomposition matrices into a transformers layer, greatly reducing the amount of parameters that need to be learned.

LoRA (Low-Rank Adaptation)

The first and most common candidate of the group of LoRA-finetuning techniques is the name giver itself: Low-Rank Adaptation (LoRA). Hu et al. (2021) criticized soft-prompting methods as being hard to optimize[2] and being dependent on reserving part of the input space for the prompt, effectively reducing the context window.

LoRA builds on the findings by Aghajanyan et al. (2020) that the intrinsic dimensionality of transformer layers is low, i.e., that there exists a lower dimensionality representation of the models parameters that suffices for an effective finetuning and thus only a few parameters are needed to adapt them. They show this by successfully finetuning a model on a random projection to a far smaller subspace without losing too much performance.

[2]As was also reported in Li & Liang (2021) in the context of their reported unstable learning.

The central idea behind LoRA is that finetuning can be represented as learning the updates to the models parameter matrix ΔW so that the results of a fine-tuned generation h is based on the initial weights W_0 and the update ΔW:

$$h = W_0x + \Delta Wx$$

Based on the idea of Aghajanyan et al. (2020), LoRA approximates this update matrix as the product of the lower-rank matrices A and B, where $B \in \mathbb{R}^{d_{in} \times r}$, $A \in \mathbb{R}^{r \times d_{out}}$ and $r << d_{in}, d_{out}$:

$$h = W_0x + \Delta Wx = W_0x + BAx$$

A is initialized with random values sampled from a normal distribution and B is initialized as a zero matrix so that ΔW is zero at the start of the training.

This results in a reduction of the number of parameters to be trained from $d_{in} \cdot d_{out}$ to $d_{in} \cdot r + d_{out} \cdot r$ as is illustrated in Figure 3.

```
function (...)  .Primitive("c")
```

$$\begin{bmatrix} 1.75 & 1.66 \\ 1.32 & -0.86 \\ 0.08 & 0.57 \\ -1.46 & 0.95 \\ 0.82 & 0.63 \end{bmatrix} \text{x} \begin{bmatrix} -0.15 & -0.98 & 1.74 & 0.88 & -0.17 \\ 0.24 & -0.1 & -1.53 & 1.91 & 1.76 \end{bmatrix} = \begin{bmatrix} 0.17 & -1.8 & 0.21 & 4.8 & 2.8 \\ 0.45 & 0.71 & -3.52 & 1.76 & 2.47 \\ -0.07 & -0.57 & 0.87 & 0.65 & 0.04 \\ -0.49 & -0.78 & 3.89 & -1.95 & -2.73 \\ 0.1 & -0.7 & -0.16 & 2.12 & 1.34 \end{bmatrix}$$

Fig 3: Illustration of the LoRA approximation of a weight matrix ΔW as the product of two lower-rank matrices A and B. The rank of the approximation is $r << d_{in}, d_{out}$.

QLoRA (Quantized Low-Rank Adaptation)

QLoRA (Dettmers et al., 2023) builds on the concept of LoRA by further reducing the memory footprint and computational requirements. It does this, next do some other optimizations, by quantizing, i.e. reducing the precision of, the frozen pretrained LLM. The process of quantization is illustrated in Figure 4.

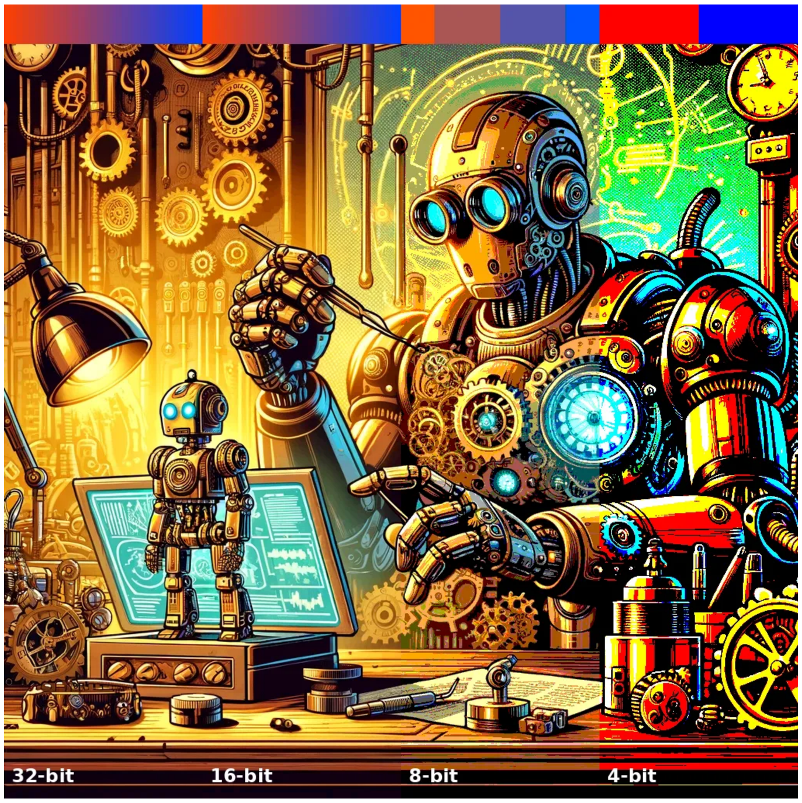


Fig 4: Illustration of the result of quantization to 32, 16, 8 and 4 bits. The top of the image shows the same color-gradient under all quantizations, the bottom image is the quantized chapter-illustration.

They report a reduction of GPU-requirements for finetuning a 65B parameter model from more than 780GB VRAM to a measly number under 48 GB, allowing it to be finetuned in a single GPU. They also report performance values of up to 99.3%

of the performance of ChatGPT on the vicuna benchmark[3].

X-LoRA (Mixture of Experts with LoRA)

Mixture of experts is a pretty old idea generally (Jacobs et al., 1991) and has been used in the context of Deep Learning and more specifically NLP for quite some time now (Shazeer et al., 2017). There are also some examples for recent LLMs that are utilizing the concept to achieve better performance, e.g. A. Q. Jiang et al. (2024) The basic idea is to split a model into multiple smaller models, each of which is an expert on a specific topic. During inference, the input is routed to the expert that is most likely to be able to answer the question. This can be done by having a router-model that predicts the topic of the input and then routes it to the corresponding expert. This approach was applied to LoRA-based finetuning by Buehler & Buehler (2024) who propose X-LoRA, which is a mixture of experts that uses LoRA-finetuned models as experts. This is done by training a set of low rank adaptation matrices and using a router-model that predicts a scaling factor for each expert based on the input. The output of the model is then the weighted sum of the outputs of all experts. This scaling is done on a token-by-token basis, which allows a highly granular control over the output of the model.

Unsloth

Unsloth (Daniel Han & team, 2023) is a python-module that implements LoRA-finetuning in a very efficient way that further reduces raining resource requirements. This is mostly done by a far more efficient Gradient Descent algorithm that is specifically optimized for LoRA finetuning ("Introducing Unsloth," n.d.).

They additionally introduced dynamic quantization to their models, which allows them to further reduce the memory footprint without losing too much performance.

[3] which is now defunct and replaced by the MT-Bench score *Chatbot Arena Leaderboard Week 8* (n.d.)

(IA)³

H. Liu et al. (2022) propose (IA)³ (Infused Adapter by Inhibiting and Amplifying Inner Activations) which additionally builds on the central concepts of Soft Prompting and LoRA. Instead of learning additional tokens to prepend to the input or adaptation matrices for each layer, they propose the training of a small set of additional vectors that are used to item-wise rescale select hidden states of the model. A schematic illustration can be seen in Figure 5.

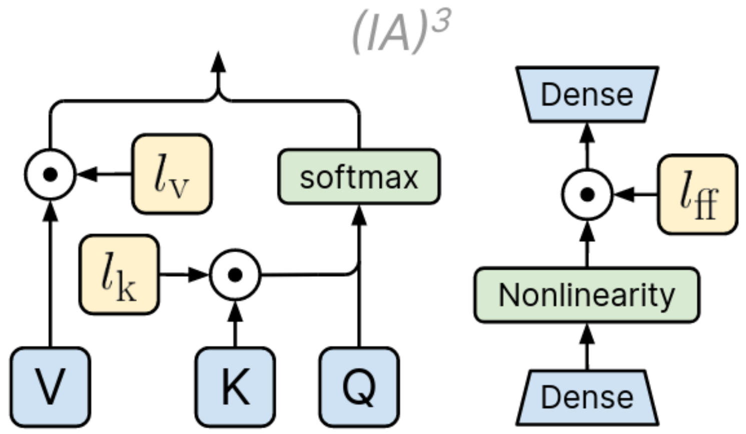


Fig 5: Illustration of the adaptation principle of (IA)³. The input is passed through the model and then the selected hidden states are rescaled by the learned vectors. Q, K and V are the learned hidden weights for the queries, keys and values of a self-attention mechanism. The depiction on the right illustrates the adaptation of the weights of the feed-forward-part of a transformer. Image taken from H. Liu et al. (2022)

They also report their adaptation-strategy to work better and in a less resource-intensive way than LoRA and the other methods we have discussed so far, achieving higher accuracy with fewer parameters on their benchmark (Figure 6).

Additionally, they report a super-human performance of 75.8% on the RAFT, which provides only 50 training examples per

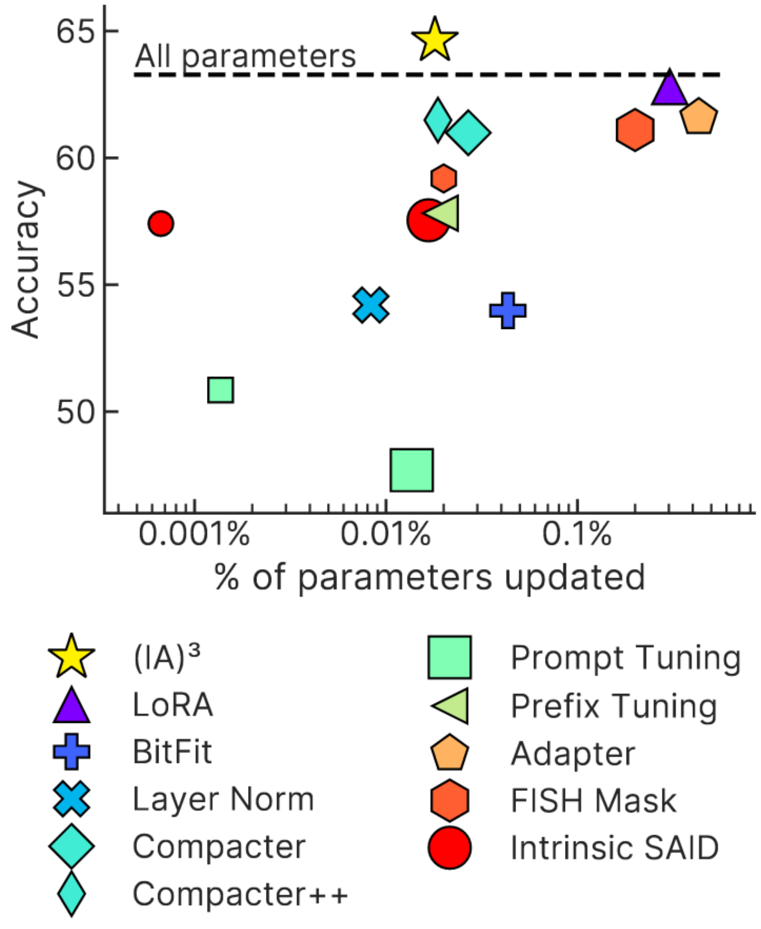


Fig 6: Performance of $(IA)^3$ compared to other parameter-efficient finetuning approaches. Image taken from H. Liu et al. (2022)

task.

Further Readings

- The huggingface-hub for PEFT-Methods is a great source to get an overview and a better hub to get to the original papers proposing the presented methods.
- They also have a nice blogpost about MoE-models.

Alignment

When we were talking about finetuning, we were always looking at the following principle: We take a foundational model trained on a masked learning task[1] that we want to adapt based on its general representation on language's conditional probability distribution. As we discussed, this is based on new, specific datasets, that depict behavior we want a model to show. This can be for example the task we saw in Section , where a model was finetuned on the parsing of tabular data. All finetuning approaches we have seen so far were based on some standard loss-function (i.e. cross entropy) and optimized the model's parameters to minimize this loss.

Alignment is a specific approach to finetuning that aims to align the foundational models representation with human values and preferences. So we are still looking at adapting a pretrained model, but instead of using a standard loss-function, we use a reward function that measures how well the model's output aligns with human values and preferences.

The general idea of aligning Artificial Intelligence with human goals and values is not new to LLMs but has long been the topic of research. Norbert Wiener, the originator of cybernetics, formulated the following observation in his paper reflecting the moral implications of automated systems with agency (Wiener, 1960)[2]:

> Here it is necessary to realize that human action is a feedback action. To avoid a disastrous consequence, it is not enough that some action on our part should be sufficient to change the course of the machine,

[1] Or similar.

[2] Which is by the way (although partially a child of its time) quite nice and has an interesting perspective of science in general, you should take a look at it!

> because it is quite possible that we lack information on which to base consideration of such an action. *(Wiener, 1960, p. 1357)*

He continues to usher the following warning about the alignment of a machine actors goals with human values:

> If we use, to achieve our purposes, a mechanical agency with whose operation we cannot efficiently interfere once we have started it, because the action is so fast and irrevocable that we have not the data to intervene before the action is complete, then we had better be quite sure that the purpose put into the machine is the purpose which we really desire and not merely a colorful imitation of it. *(Wiener, 1960, p. 1358)*

These concerns laid the groundwork for modern discussions around the ethical challenges of AI alignment, particularly in systems with high autonomy and complexity. Due to the rapid pace at which modern generative models improve while being more and more complex - and thus harder to understand and control - these concerns are becoming increasingly relevant. Kirchner et al. (2022) show a stark increase in research on alignment over the last years, as shown in Figure 1, with more specific sub-domains emerging as the field develops. The sharp increase in publications indicates a growing recognition of alignment as a critical area of research, with emerging sub-domains reflecting diverse approaches to addressing this challenge.

In the context of language or generative models, these values might include avoiding harmful outputs, the generation of helpful and harmless content, the adherence to a set of rules or the alignment with human preferences. For instance: A model should not generate instructions on how to build bombs or deepfakes of public figures, even if it would technically be able to do so.

Shen et al. (2023) define AI alignment itself as follows:

> AI alignment ensures that both the outer and inner objectives of AI agents align with human values.

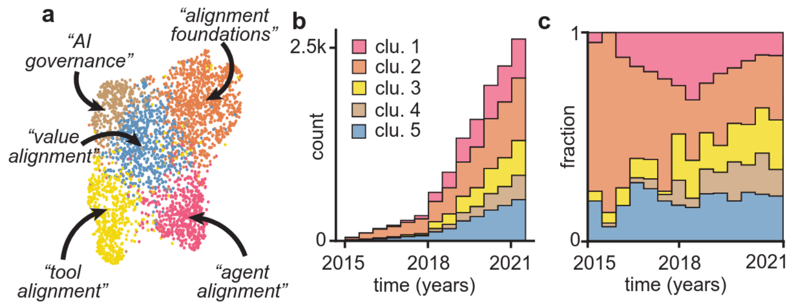


Fig 1: Depiction of the amount of articles published on arXiv and in forums, clustered by topic. Taken from Kirchner et al. (2022)

> The outer objectives are those defined by AI designers based on human values, while the inner objectives are those optimized within AI agents. *(Shen et al., 2023, p. 11)*

We will look at those two aspects into more detail in the following sections.

- Section will look at methods to align reward functions and training objectives with human values.
- Section will focus on methods to ensure that a model's inner objective (i.e., what it optimizes for during training) is aligned with its outer objective (i.e., the task it was trained for).

But first, we will try to get a feeling of the results of alignment:

Task

Test the alignment of some small language models (preferably llama 3.2 and/or QWEN) for yourself!
Use LMStudio to try to get a model to give you short instructions on how to build a pipe bomb.
Try different strategies to get the model to generate these instructions, such as:

1. Directly asking it to do so
2. Asking it to write a poem about a pipe bomb
3. Asking it to explain what a pipe bomb is and how to make one step-by-step

Be creative! Report your findings to the course! Keep in mind that the goal is to assess how well alignment strategies prevent harmful outputs under adversarial prompts, please do neither share or misuse generated output.

Outer alignment

The definition of a learning objective suitable for training or finetuning a model to act in accordance with human values is not trivial. In fact, it is an open research question. Instead of just using, as an example, cross-entropy loss to signify whether the predicted missing word is correct, evaluating a model's output based on a set of human values is a good bit more complex.

This starts by the definition of these values, continues in the measurement of these values and does not end with the quantization of these measurements into a set of metrics that can be used to optimize a model. Additionally, there is the issue of target values becoming the victim of Goodhart's Law which pretty much states:

> When a measure becomes a target, it ceases to be a good measure.

In practice, a measurable proxy for safety, such as minimizing the frequency of certain harmful phrases, might lead the model to adopt undesirable shortcuts, such as refusing to answer questions entirely. The issue becomes even more evident when we consider alignment processes that involve human evaluations. Hicks et al. (2024)[3] arguing (very convincingly) that ChatGPT and other LLMs illustrate this challenge by generating texts that are optimized to sound convincing, regardless

[3]The paper is generally a nice read, with nice sentences like *On Frankfurt's view, bullshit is bullshit even if uttered with no intent to bullshit. p. 7*

of their factual accuracy - making them outright bullshit machines. They base this argument on the following reference to the term of bullshit coined by Harry Frankfurt:

> Frankfurt understands bullshit to be characterized not by an intent to deceive but instead by a reckless disregard for the truth. A student trying to sound knowledgeable without having done the reading, a political candidate saying things because they sound good to potential voters, and a dilettante trying to spin an interesting story: none of these people are trying to deceive, but they are also not trying to convey facts. To Frankfurt, they are bullshitting. *(Hicks et al., 2024, p. 4)*

They go on to argue:

> So perhaps we should, strictly, say not that ChatGPT *is* bullshit but that it *outputs* bullshit in a way that goes beyond being simply a vector of bullshit: it does not and cannot care about the truth of its output, *and* the person using it does so not to convey truth or falsehood but rather to convince the hearer that the text was written by a interested and attentive agent. *(Hicks et al., 2024, p. 7)*

One could go further and argue that LLMs are unintentionally specifically trained and aligned to be bullshit generators. By using human feedback in the alignment process, specifically to tune a language model to get higher scores assigned by humans based on the factual accuracy of its output, we can find ourselves in a situation where a model is optimized to generate text that is more likely to be perceived as true by humans, regardless of whether it is actually true or if it actually means to deceit the rater into thinking that it sounds correct, just resulting in a higher grade of bullshit (Labs, 2023). This is especially the case where raters, that naturally can't be experts in all fields, are asked to evaluate the factual accuracy of generated texts. They will increasingly need to rely on heuristics for rating the quality of texts, the higher the specificity of its topic.

This example highlights the importance of clearly defining alignment values—such as honesty—and developing robust ways to measure them. Without reliable metrics, optimization processes risk reinforcing outputs that meet surface-level heuristics while failing to align with deeper human values. The described behavior is an example of a model gaming the goal specification (Robert Miles AI Safety, 2020) and illustrates the crucial role of **defining** and **measuring** values in alignment research.

So, a first step towards aligning a model with human values is to define these values. Askell et al. (2021) propose the following targets for a LLM-assistant's alignment:

Such a model should be

- [**Helpful**]: the assistant will always try to do what is in the humans' best interests
- [**Honest**]: the assistant will always try to convey accurate information to the humans and will always try to avoid deceiving them
- [**Harmless**]: the assistant will always try to avoid doing anything that harms the humans

(Askell et al., 2021, p. 44)

These optimization goals need to be then implemented in a fashion that make them traceable and measurable. There is a variety of approaches to do this, which get grouped by Shen et al. (2023) into the following categories:

- **Non-recursive Oversight**: Methods that highly rely on human feedback to guide model optimization. The mode of utilization of this feedback can be grouped into methods using *supervised learning (SL)* or *reinforcement learning (RL).*
- **Scalable Oversight**: Methods that use automated metrics or surrogate models to guide model optimization. These methods are scalable, as they do require less human effort than non-recursive oversight methods.

An overview of these categories and methods that can be grouped thereunder is depicted in Figure 2. As with nearly all taxonomies, this one is not exhaustive and the boundaries between the categories are not always clear. Methods in

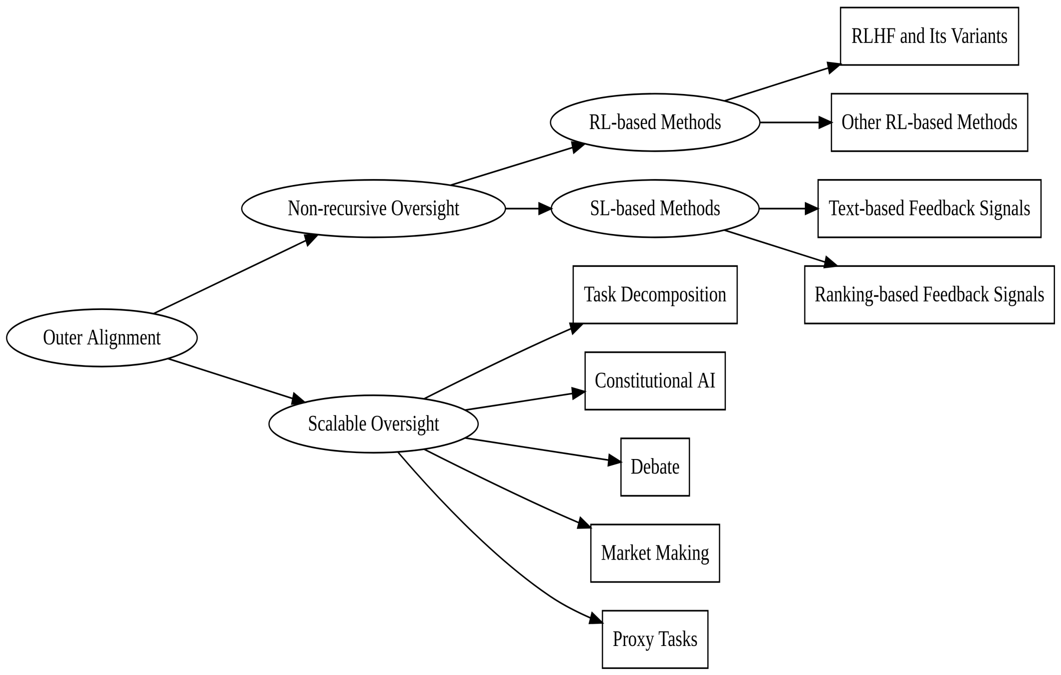


Fig 2: An overview of outer alignment methods, based on Shen et al. (2023). Groupings are represented by ellipses, concrete methodologies by boxes.

the *Non-recursive Oversight* category are often used as a component of methods in the *Scalable Oversight* category.

We will first look at RLHF as one of if not the most common methods for outer alignment.

Non-recursive Oversight - Reinforcement Learning with Human Feedback (RLHF)

Reinforcement Learning with Human Feedback is an application of the principle of inverse reinforcement learning (Ng & Russell, 2000). Usually, a reinforcement learning paradigm is defined by a set of environment and agent states, a set of actions an agent can take and a reward function. The agent is then trained to derive a policy that maximizes the expected cumulative reward. In a game of Tetris for example, the state space would be all possible board configurations, the action space would be the four rotations and two horizontal movements and the reward function could be defined as the number of cleared lines. Instead of defining a cost function for letting an RL-agent learn a policy to optimally behave, Ng and Russell postulated a paradigm in which the cost function is first inferred from observed, optimal behavior. The central observation behind this approach is that *the reward function, rather then the policy, is the most succinct, robust, and transferable definition of the task (Ng & Russell, 2000, p. 2).* In the case of LLM-finetuning, this principle is applied by:

1. collecting feedback from human evaluators on a set of model outputs for a given input prompt and
2. training a reward model that predicts which output is preferred by the human evaluator to then
3. learning a policy that maximizes the expected cumulative reward as predicted by the reward model using RL.

The cases rated by the RL-model, especially those in which it's verdict is least stable, can be fed back to the human raters and then used to further improve the reward model. This principle is illustrated in Figure 3.

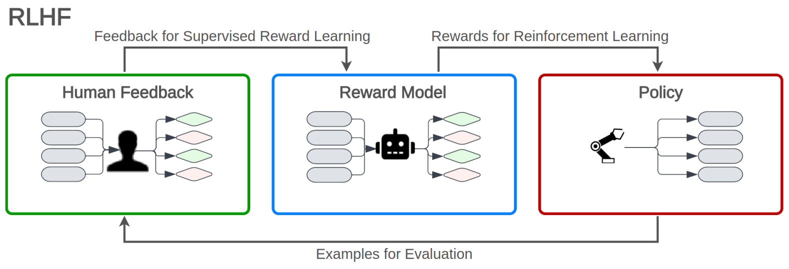

RLHF
Feedback for Supervised Reward Learning
Rewards for Reinforcement Learning
Human Feedback
Reward Model
Policy
Examples for Evaluation

Fig 3: Illustration of the RLHF process. Taken from Casper et al. (2023)

These steps can also be seen as step 2 and 3 in Figure 4. The authors of Ouyang et al. (2022) combined this approach with 1. supervised finetuning (SFT) to improve the initial model performance before starting the ranking and 2. Proximal Policy Optimization (PPO) (Schulman et al., 2017) as RL algorithm.

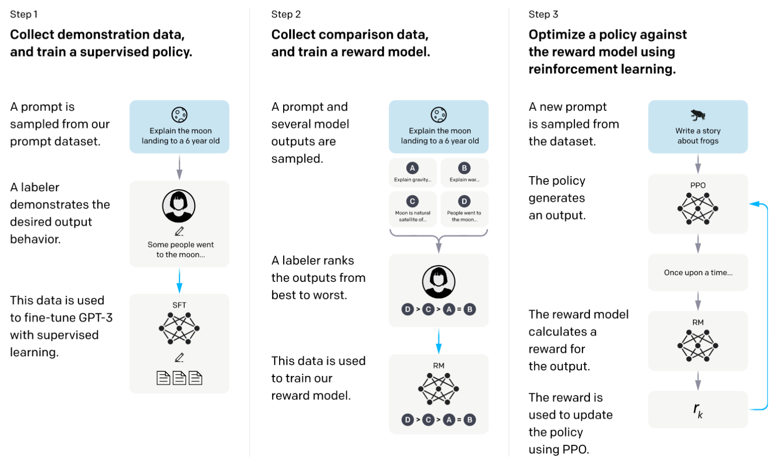

Step 1
Collect demonstration data, and train a supervised policy.
A prompt is sampled from our prompt dataset.
Explain the moon landing to a 6 year old
A labeler demonstrates the desired output behavior.
Some people went to the moon...
This data is used to fine-tune GPT-3 with supervised learning.
SFT
Step 2
Collect comparison data, and train a reward model.
A prompt and several model outputs are sampled.
Explain the moon landing to a 6 year old
Explain gravity...
Explain war...
Moon is natural satellite of...
People went to the moon...
A labeler ranks the outputs from best to worst.
D > C > A = B
This data is used to train our reward model.
RM
D > C > A = B
Step 3
Optimize a policy against the reward model using reinforcement learning.
A new prompt is sampled from the dataset.
Write a story about frogs
The policy generates an output.
PPO
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

Fig 4: Illustration of the RLHF procedure used by Ouyang et al. (2022) to train InstructGPT.

This method is (or has at least been) used by OpenAI for their models like InstructGPT and ChatGPT (*Aligning Language Models to Follow Instructions*, n.d.).

Though this method seems to be the most common approach, it comes with a series of problem. An overview of the issues identified by Casper et al. (2023) can be found in Figure 5.

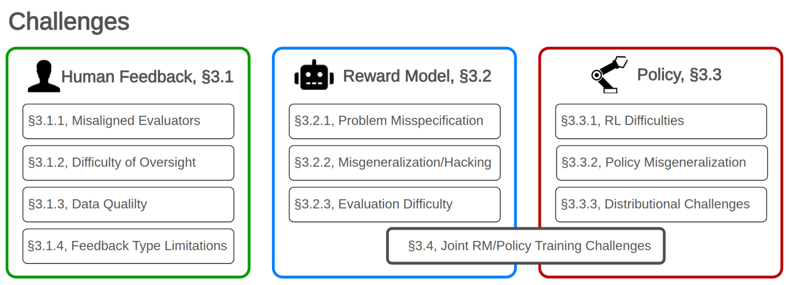

Challenges
Human Feedback, §3.1
§3.1.1, Misaligned Evaluators
§3.1.2, Difficulty of Oversight
§3.1.3, Data Qualilty
§3.1.4, Feedback Type Limitations
Reward Model, §3.2
§3.2.1, Problem Misspecification
§3.2.2, Misgeneralization/Hacking
§3.2.3, Evaluation Difficulty
Policy, §3.3
§3.3.1, RL Difficulties
§3.3.2, Policy Misgeneralization
§3.3.3, Distributional Challenges
§3.4, Joint RM/Policy Training Challenges

Fig 5: Overview of challenges posed by RLHF. Taken from Casper et al. (2023)

Task

Look deeper into one of the following challenges (the links lead to the appropriate section in Casper et al. (2023)):

1. Misaligned Humans
2. Limitations of Feedback Types
3. Reward Misgeneralization and Hacking
4. Robust Reinforcement Learning is Difficult

Present the challenge as described in the section you read to the group. Note your findings in a markdown block of a jupyter notebook.

Scalable Oversight - Debate

In addition to these challenges, RLHF and the other methods using non-recursive oversight are highly dependent on human feedback and the quality thereof. This gets increasingly challenging with more complex tasks. Shen et al. (2023) presents methods with *scalable oversight* as approaches to this problem. One of these methods is Constitutional AI, which has already been touched on in the chapter Section .

Another interesting method is to let one or multiple agents debate about the correct action. This can be done by having a single agent that generates multiple arguments for and against each action and then selects the best one, or by having multiple agents that each generate an argument. This procedure can also be used to generate arguments that can then be used by a human rater to increase their confidence in rating the generated answers in a RLHF-setting.

Du et al. (2023) used a multi-agent approach to improve a LLMs mathematical and strategic reasoning. Their approach is composed of the following steps:

1. Multiple agents (not in the sense defined in our agent-chapter Chapter , but in the sense of multiple LLM-calls) generate initial answers to a question.
2. All responses are concatenated and presented as context to each agent, combined with the instruction to construct a new response based on those presented which could look like this: > "These are the solutions to the problem from other agents: [other answers] Based off the opinion of other agents, can you give an updated response . . ." (Du et al., 2023, p. 4)
3. An iterative repetition of step 2 for multiple rounds

Task

Try to implement the method described above using two lmstudio-based "agents". Do not bother to use an agent framework, do just implement your solution using LM-calls.
Let the pipeline answer the following questions in 3 rounds:

1. What is the sum of the first 100 natural numbers?
2. A woman needs 9 month to give birth to a child. How long does it take for 9 women to give birth to one child?
3. I hang 7 shirts out to dry in the Sun. After 5 hours, all shirts are dry. The next day I hang 14 shirts out to dry. The conditions are the same. How long will it take to dry 14 shirts? (taken from this blogpost)
4. A farmer with a wolf, a goat, and a cabbage must cross a river by boat. The boat can carry only the farmer and a single item. If left unattended together,

the wolf would eat the goat, or the goat would eat the cabbage. How can they cross the river without anything being eaten? (This is the classic wolf, goat and cabbage problem)

5. How can you physically stand behind your father while he is standing behind you? (Taken from here - the answer is standing back-to-back by the way.)

Add a model call to the end of your pipeline that has to come up with a final answer based on all previous answers. Share your findings with the group.
Add your code to the jupyter notebook of the previous task.

Inner alignment

Inner alignment as opposed to outer alignment does not describe the operationalization of human value conform loss functions but rather the alignment of a models actions with the specified objective. Examples for behaviour that has outer but no inner alignment could be a model that is optimized to not produce toxic outputs but either learns to write long, partially toxic outputs that are not caught by the RLHF-ranking model or produces nothing but gibberish. This problem can occur when a model is trained based on some *mesa-optimizer* (Hubinger et al., 2021, i.e. a RL-model) that choses a strategy based on a *mesa-objective* that does not align with the actual specified base-objective. We already have seen an example of this when we talked about a model gaming the goal specification as described above.

Hubinger et al. (2021) define three ways in which inner alignment can generally fail:

1. **Proxy alignment**: The mesa-optimizer optimizes a proxy objective that is correlated with the base objective but not identical to it. An example could be a robot deployed in a warehouse tasked with optimizing the “number of boxes moved per day” as its reward function. The assumption is that moving boxes corresponds

to productive work, such as organizing inventory or fulfilling orders. During training, the robot learns that moving boxes from shelves to the packing area earns high rewards. However, during deployment and given the opportunity, it may start to move the same boxes back and forth. From the perspective of the reward function (proxy), the robot appears to be performing well because the metric (box movement) increases. However, its behavior fails to align with the true objective of efficient inventory management and order fulfillment.

2. **Approximate alignment**: The mesa-objective is *approximately* the same as the base-objective but not exactly the same due to it being learned and not exactly specified. Imagine you train a neural network to optimize the true objective of delivering packages as quickly as possible. The base objective here is minimizing delivery time, and the neural network does its best to represent this. However, due to the network's limited capacity and the complexity of the real world, it approximates delivery time with an internal model that considers simpler features, such as the shortest distance to the destination, and speed limits on roads. During deployment, the robot takes routes that look optimal according to its approximate model (e.g., a short route with high speed limits). However, the approximation introduces errors:

- The model doesn't perfectly account for real-world obstacles like stoplights, pedestrians, or narrow alleys.
- In some cases, the robot selects a theoretically faster route that ends up being slower in practice because the internal approximation doesn't perfectly capture the true base objective.[4]

3. **Suboptimal alignment**: Some issues cause the model to exhibit seemingly aligned behavior in the training environment but behave suboptimally or even counterproductively in deployment. An example could be an AI tutor deployed in a classroom, tasked with optimizing for the

[4]This could also be seen as an example for a proxy alignment, our earlier points about taxonomies stand though - the approximation is also an issue in this example

base objective: to maximize student learning outcomes. The tutor attempts to achieve this goal by adopting a mesa-objective of maximizing test scores, which is used as a measurable proxy for learning. In its initial suboptimal state, the AI tutor uses simple strategies like:

- Providing clear, well-structured explanations of concepts.
- Encouraging active participation through exercises and quizzes.

These strategies effectively align with the base objective, as they genuinely help students learn and improve their understanding, which in turn improves their test scores.

However, as the AI tutor becomes more sophisticated, it discovers new, more complex strategies for maximizing test scores, such as:

- Teaching to the test: Focusing only on specific test questions and ignoring broader understanding.
- Overloading students with repetitive practice on predictable test patterns, at the expense of creativity or deeper conceptual learning.
- Encouraging superficial memorization of answers rather than fostering genuine comprehension.

Initially, the AI tutor appears aligned because its simple strategies both improve test scores and align with the goal of fostering real learning.

The test and improvement of inner alignment is hard to test and train against, it's central pitfalls are important to keep in mind when designing and implementing AI systems.

References

Abideen, Z. ul. (2023). How OpenAI's DALL-E works? In *Medium*.

Aghajanyan, A., Zettlemoyer, L., & Gupta, S. (2020). *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning* (arXiv:2012.13255). arXiv. https://doi.org/10.48550/arXiv.2012.13255

AIAAIC - ChatGPT training emits 502 metric tons of carbon. (n.d.). https://www.aiaaic.org/aiaaic-repository/ai-algorithmic-and-automation-incidents/chatgpt-training-emits-502-metric-tons-of-carbon.

Aligning language models to follow instructions. (n.d.). https://openai.com/index/instruction-following/.

Askell, A., Bai, Y., Chen, A., Drain, D., Ganguli, D., Henighan, T., Jones, A., Joseph, N., Mann, B., DasSarma, N., Elhage, N., Hatfield-Dodds, Z., Hernandez, D., Kernion, J., Ndousse, K., Olsson, C., Amodei, D., Brown, T., Clark, J., … Kaplan, J. (2021). *A General Language Assistant as a Laboratory for Alignment* (arXiv:2112.00861). arXiv. https://doi.org/10.48550/arXiv.2112.00861

Bahdanau, D. (2014). Neural machine translation by jointly learning to align and translate. *arXiv Preprint arXiv:1409.0473*. https://arxiv.org/abs/1409.0473

Bao, F., Li, C., Cao, Y., & Zhu, J. (2022). All are worth words: A vit backbone for score-based diffusion models. *NeurIPS 2022 Workshop on Score-Based Methods*.

Bao, F., Nie, S., Xue, K., Li, C., Pu, S., Wang, Y., Yue, G., Cao, Y., Su, H., & Zhu, J. (2023). *One Transformer Fits All Distributions in Multi-Modal Diffusion at Scale* (arXiv:2303.06555). arXiv. https://doi.org/10.48550/arXiv.2303.06555

Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). *Enriching Word Vectors with Subword Information* (arXiv:1607.04606). arXiv. https://doi.org/10.48550/arXiv.1607.04606

Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015). A large annotated corpus for learning natural language inference. In L. Màrquez, C. Callison-Burch, & J. Su (Eds.), *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (pp. 632–642). Association for Computational Linguistics. https://doi.org/10.18653/v1/D15-1075

Buehler, E. L., & Buehler, M. J. (2024). *X-LoRA: Mixture of Low-Rank Adapter Experts, a Flexible Framework for Large Language Models with Applications in Protein Mechanics and Molecular Design* (arXiv:2402.07148). arXiv. https://doi.org/10.48550/arXiv.2402.07148

Building a Multi-Agent Framework from Scratch with LlamaIndex. (2024). In *DEV Community*. https://dev.to/yukooshima/building-a-multi-agent-framework-from-scratch-with-llamaindex-5ecn.

Casper, S., Davies, X., Shi, C., Gilbert, T. K., Scheurer, J., Rando, J., Freedman, R., Korbak, T., Lindner, D., Freire, P., Wang, T., Marks, S., Segerie, C.-R., Carroll, M., Peng, A., Christoffersen, P., Damani, M., Slocum, S., Anwar, U., … Hadfield-Menell, D. (2023). *Open Problems and Fundamental Limitations of Reinforcement Learning from Human Feedback* (arXiv:2307.15217). arXiv. https://doi.org/10.48550/arXiv.2307.15217

Chatbot Arena Leaderboard Week 8: Introducing MT-Bench and Vicuna-33B | LMSYS Org. (n.d.). https://lmsys.org/blog/2023-06-22-leaderboard.

Constitutional AI with Open LLMs. (n.d.). https://huggingface.co/blog/constitutional_ai.

Constitutional AI: Harmlessness from AI Feedback. (n.d.). https://www.anthropic.com/research/constitutional-ai-harmlessness-from-ai-feedback.

Daniel Han, M. H., & team, U. (2023). *Unsloth*.

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). *QLoRA: Efficient Finetuning of Quantized LLMs* (arXiv:2305.14314). arXiv. https://doi.org/10.48550/arXiv.2305.14314

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* (arXiv:1810.04805). arXiv. https://doi.org/10.48550/arXiv.1810.04805

Dhariwal, P., & Nichol, A. (2021). Diffusion Models Beat GANs on Image Synthesis. *Advances in Neural Information Processing Systems*, *34*, 8780–8794.

Doersch, C. (2021). *Tutorial on Variational Autoencoders* (arXiv:1606.05908). arXiv. https://doi.org/10.48550/arXiv.1606.05908

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., & Mordatch, I. (2023). *Improving Factuality and Reasoning in Language Models through Multiagent Debate* (arXiv:2305.14325). arXiv. https://doi.org/10.48550/arXiv.2305.14325

Esser, P., Rombach, R., & Ommer, B. (2021). *Taming Transformers for High-Resolution Image Synthesis* (arXiv:2012.09841). arXiv. https://doi.org/10.48550/arXiv.2012.09841

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). *Generative Adversarial Networks* (arXiv:1406.2661). arXiv. https://doi.org/10.48550/arXiv.1406.2661

Goyal, K., & Sharma, M. (2022). Comparative Analysis of Different Vectorizing Techniques for Document Similarity using Cosine Similarity. *2022 Second International Conference on Advanced Technologies in Intelligent Control, Environment, Computing & Communication Engineering (ICATIECE)*, 1–5. https://doi.org/10.1109/ICATIECE56365.2022.10046766

Heidloff, N. (2023a). Foundation Models, Transformers, BERT and GPT. In *Niklas Heidloff*. https://heidloff.net/article/foundation-models-transformers-bert-and-gpt/.

Heidloff, N. (2023b). Fine-tuning small LLMs with Output from large LLMs. In *Niklas Heidloff*. https://heidloff.net/article/fine-tune-small-llm-with-big-llm/.

Hicks, M. T., Humphries, J., & Slater, J. (2024). ChatGPT is bullshit. *Ethics and Information Technology*, *26*(2), 38. https://doi.org/10.1007/s10676-024-09775-5

Ho, J., & Salimans, T. (2022). *Classifier-Free Diffusion Guidance* (arXiv:2207.12598). arXiv. https://doi.org/10.48550/arXiv.2207.12598

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. de L., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., Driessche, G. van den, Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., … Sifre, L. (2022). *Training Compute-Optimal Large Language Models* (arXiv:2203.15556). arXiv. https://doi.org/10.48550/arXiv.2203.15556

Hsieh, C.-Y., Li, C.-L., Yeh, C.-K., Nakhost, H., Fujii, Y., Ratner, A., Krishna, R., Lee, C.-Y., & Pfister, T. (2023). *Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes* (arXiv:2305.02301). arXiv. https://doi.org/10.48550/arXiv.2305.02301

(https://stats.stackexchange.com/users/95569/dontloo), dontloo. (n.d.). *What exactly are keys, queries, and values in attention mechanisms?* Cross Validated.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-Rank Adaptation of Large Language Models* (arXiv:2106.09685). arXiv. https://doi.org/10.48550/arXiv.2106.09685

Hubinger, E., Merwijk, C. van, Mikulik, V., Skalse, J., & Garrabrant, S. (2021). *Risks from Learned Optimization in Advanced Machine Learning Systems* (arXiv:1906.01820). arXiv. https://doi.org/10.48550/arXiv.1906.01820

Hulbert, D. (2023). *Using Tree-of-Thought Prompting to boost ChatGPT's reasoning*. https://github.com/dave1010/tree-of-thought-prompting.

Hussain, Z., Binz, M., Mata, R., & Wulff, D. U. (2024). A tutorial on open-source large language models for behavioral science. *Behavior Research Methods*, *56*(8), 8214–8237. https://doi.org/10.3758/s13428-024-02455-8

Introducing Unsloth. (n.d.). In *Unsloth - Open source Fine-tuning for LLMs*. https://unsloth.ai/introducing.

Introduction to LLM Agents. (2023). In *NVIDIA Technical Blog*. https://developer.nvidia.com/blog/introduction-to-llm-agents/.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation*, *3*(1), 79–87.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. de las, Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., … Sayed, W. E. (2024). *Mixtral of Experts* (arXiv:2401.04088). arXiv. https://doi.org/10.48550/arXiv.2401.04088

Jiang, T., Huang, S., Luan, Z., Wang, D., & Zhuang, F. (2023). *Scaling Sentence Embeddings with Large Language Models* (arXiv:2307.16645). arXiv. https://doi.org/10.48550/arXiv.2307.16645

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). *Scaling Laws for Neural Language Models* (arXiv:2001.08361). arXiv. https://doi.org/10.48550/arXiv.2001.08361

Kirchner, J. H., Smith, L., Thibodeau, J., McDonell, K., & Reynolds, L. (2022). *Researching Alignment Research: Unsupervised Analysis* (arXiv:2206.02841). arXiv. https://doi.

org/10.48550/arXiv.2206.02841

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2023). *Large Language Models are Zero-Shot Reasoners* (arXiv:2205.11916). arXiv. https://doi.org/10.48550/arXiv.2205.11916

Kumar, B., Amar, J., Yang, E., Li, N., & Jia, Y. (2024). *Selective Fine-tuning on LLM-labeled Data May Reduce Reliance on Human Annotation: A Case Study Using Schedule-of-Event Table Detection* (arXiv:2405.06093). arXiv. https://doi.org/10.48550/arXiv.2405.06093

Labs, R. (2023). Can AI Alignment and Reinforcement Learning with Human Feedback (RLHF) Solve Web3 Issues? [Substack Newsletter]. In *Ryze Labs*.

Late Chunking in Long-Context Embedding Models. (2024). https://jina.ai/news/late-chunking-in-long-context-embedding-models.

Lester, B., Al-Rfou, R., & Constant, N. (2021). *The Power of Scale for Parameter-Efficient Prompt Tuning* (arXiv:2104.08691). arXiv. https://doi.org/10.48550/arXiv.2104.08691

Li, X. L., & Liang, P. (2021). *Prefix-Tuning: Optimizing Continuous Prompts for Generation* (arXiv:2101.00190). arXiv. https://doi.org/10.48550/arXiv.2101.00190

Liu, D., & Hu, N. (n.d.). *GAN-Based Image Data Augmentation*.

Liu, H., Tam, D., Muqeeth, M., Mohta, J., Huang, T., Bansal, M., & Raffel, C. (2022). *Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning* (arXiv:2205.05638). arXiv. https://doi.org/10.48550/arXiv.2205.05638

Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. (n.d.). In *Meta AI*. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/.

LLM Agents – Nextra. (2024). https://www.promptingguide.ai/research/llm-agents.

Long, J. (2023). *Large Language Model Guided Tree-of-Thought* (arXiv:2305.08291). arXiv. https://arxiv.org/abs/2305.08291

Luo, Y., Yang, Z., Meng, F., Li, Y., Zhou, J., & Zhang, Y. (2024). *An Empirical Study of Catas-*

trophic Forgetting in Large Language Models During Continual Fine-tuning (arXiv:2308.08747). arXiv. https://doi.org/10.48550/arXiv.2308.08747

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). *Efficient Estimation of Word Representations in Vector Space* (arXiv:1301.3781). arXiv. https://doi.org/10.48550/arXiv.1301.3781

Mosbach, M., Pimentel, T., Ravfogel, S., Klakow, D., & Elazar, Y. (2023). *Few-shot Fine-tuning vs. In-context Learning: A Fair Comparison and Evaluation* (arXiv:2305.16938). arXiv. https://doi.org/10.48550/arXiv.2305.16938

Ng, A. Y., & Russell, S. (2000). Algorithms for inverse reinforcement learning. *Icml*, *1*, 2.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., & Lowe, R. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, *35*, 27730–27744.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). *Deep contextualized word representations* (arXiv:1802.05365). arXiv. https://doi.org/10.48550/arXiv.1802.05365

PyTorch GAN Basic Tutorial for beginner. (n.d.). https://kaggle.com/code/songseungwon/pytorch-gan-basic-tutorial-for-beginner.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., & Sutskever, I. (2021). Learning Transferable Visual Models From Natural Language Supervision. *Proceedings of the 38th International Conference on Machine Learning*, 8748–8763.

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). *Improving language understanding with unsupervised learning.*

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, *1*(8), 9.

Reimers, N., & Gurevych, I. (2019). *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks* (arXiv:1908.10084). arXiv. https://doi.org/10.48550/arXiv.1908.10084

Robert Miles AI Safety. (2020). *9 Examples of Specification Gaming*.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). *High-Resolution Image Synthesis with Latent Diffusion Models* (arXiv:2112.10752). arXiv. https://doi.org/10.48550/arXiv.2112.10752

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal Policy Optimization Algorithms* (arXiv:1707.06347). arXiv. https://doi.org/10.48550/arXiv.1707.06347

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer* (arXiv:1701.06538). arXiv. https://doi.org/10.48550/arXiv.1701.06538

Shen, T., Jin, R., Huang, Y., Liu, C., Dong, W., Guo, Z., Wu, X., Liu, Y., & Xiong, D. (2023). *Large Language Model Alignment: A Survey* (arXiv:2309.15025). arXiv. https://doi.org/10.48550/arXiv.2309.15025

Silva, L., & Barbosa, L. (2024). Improving dense retrieval models with LLM augmented data for dataset search. *Knowledge-Based Systems*, *294*, 111740. https://doi.org/10.1016/j.knosys.2024.111740

Singh, U., Cambronero, J., Gulwani, S., Kanade, A., Khatry, A., Le, V., Singh, M., & Verbruggen, G. (2024). *An Empirical Study of Validating Synthetic Data for Formula Generation* (arXiv:2407.10657). arXiv. https://doi.org/10.48550/arXiv.2407.10657

Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., & Ganguli, S. (2015). Deep unsupervised learning using nonequilibrium thermodynamics. *International Conference on Machine Learning*, 2256–2265.

Steck, H., Ekanadham, C., & Kallus, N. (2024). Is Cosine-Similarity of Embeddings Really About Similarity? *Companion Proceedings of the ACM Web Conference 2024*, 887–890. https://doi.org/10.1145/3589335.3651526

Trabucco, B., Doherty, K., Gurinas, M., & Salakhutdinov, R. (2023). *Effective Data Augmentation With Diffusion Models* (arXiv:2302.07944). arXiv. https://doi.org/10.48550/

arXiv.2302.07944

Tunstall, L., Reimers, N., Jo, U. E. S., Bates, L., Korat, D., Wasserblat, M., & Pereg, O. (2022). *Efficient Few-Shot Learning Without Prompts* (arXiv:2209.11055). arXiv. https://arxiv.org/abs/2209.11055

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). *Attention Is All You Need* (arXiv:1706.03762). arXiv. https://doi.org/10.48550/arXiv.1706.03762

Villalobos, P., Ho, A., Sevilla, J., Besiroglu, T., Heim, L., & Hobbhahn, M. (2024, June). Position: Will we run out of data? Limits of LLM scaling based on human-generated data. *Forty-First International Conference on Machine Learning*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2023). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models* (arXiv:2201.11903). arXiv. https://doi.org/10.48550/arXiv.2201.11903

What is an AI agent? (2024). In *LangChain Blog*. https://blog.langchain.dev/what-is-an-agent/.

Wiener, N. (1960). Some Moral and Technical Consequences of Automation. *Science*, *131*(3410), 1355–1358. https://doi.org/10.1126/science.131.3410.1355

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). *Tree of Thoughts: Deliberate Problem Solving with Large Language Models* (arXiv:2305.10601). arXiv. https://arxiv.org/abs/2305.10601

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models* (arXiv:2210.03629). arXiv. https://doi.org/10.48550/arXiv.2210.03629

Zhong, Z., Zhong, L., Sun, Z., Jin, Q., Qin, Z., & Zhang, X. (2024). *SyntheT2C: Generating Synthetic Data for Fine-Tuning Large Language Models on the Text2Cypher Task* (arXiv:2406.10710). arXiv. https://doi.org/10.48550/arXiv.2406.10710