Mathematics for Deep Learning

Enhancing Large Language Models with Human Feedback via Reinforcement Learning

Anna Blanpied

December 2023

Contents

1 Introduction

This report navigates through two significant domains: Transformers and the integration of Reinforcement Learning with Human Feedback.

We start by developing what is of Transformers. Initially, the report analyzes their architecture and mathematical foundations. Then, it delves deeper into the concept of causal transformers, represented by models like GPT, shedding light on their self-supervised training methodologies. This section aims to elucidate how these models have reshaped AI, particularly in natural language processing, by capturing intricate patterns and contextual dependencies.

Moving to the second part, the report delves into the cooperation between Reinforcement Learning and Human Feedback. It begins by establishing foundational concepts that are Markov Decision Processes, providing a framework for understanding decision-making under uncertainty. The exploration continues by detailing the mechanisms of reinforcement learning, emphasizing its iterative learning approach in dynamic environments. Finally, the report concludes by clarifying the incorporation of human feedback into reinforcement learning models, explaining how such integration improve learning processes.

2 Transformers

There exist various neural network architectures types optimized to specific data types. For example, Convolutional Neural Networks (CNNs) excel in processing images due to their ability to extract spatial features effectively. On the other hand, Recurrent Neural Networks (RNNs) are employed for text comprehension as they maintain an understanding of sequential data, crucial for interpreting language where word order matters. However, RNNs encountered limitations with the vanishing gradient problem, and with large text volumes due to their sequential processing nature so that they could not parallelize and thus it was hard to train them.

Transformers were introduced in 2017 by researchers from Google and the University of Toronto. Initially aimed at translation tasks, Transformers revolutionized language processing. Unlike RNNs, they efficiently parallelize computations, enabling the training of large models.

Transformers utilize three key mechanisms: Positional Encoding, Attention, and Self-Attention.

- **Positional encoding** involves assigning a unique number to each word in a sentence before fitting it into the neural network. This numbering represents the word's position in the sentence, so it stores the information about word order directly into the data rather than within the network's structure. Consequently, the model learns to interpret these positions, getting the significance of word order from the provided data.

- The **Attention** mechanism. The key elements in a transformer are *scaled dot-product attention units*.
  Each attention unit learns three sets of weight matrices: query W_Q, key W_K, and value W_V. For every token represented as x_i, these matrices generate query $q_i = x_i * W_Q$, key $k_i = x_i * W_K$, and value $v_i = x_i * W_V$ vectors.
  The attention weights, used to understand the relevance of one token to another, are derived by calculating the dot product between query and key vectors $a_{ij} = q_i * k_j$ and then normalizing these using a softmax function.
  Stabilizing gradients during training involves dividing the attention weights by the square root of the key vector's dimension $\sqrt{d_k}$. The separate query and key matrices W_Q and W_K enable non-symmetrical attention; if token i strongly attends to token j, it doesn't necessarily mean that token j will strongly attend to token i.
  Finally, the output for token i is a weighted sum of all token value vectors, weighted by their respective attention weights a_{ij}. This gives us the following

attention representation :

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

In conclusion, the attention mechanism enables a text model to examine every word in the original sentence, aiding in decision-making for translating words into the output sentence. Through exposure to data, the model gradually learns which words to prioritize or 'attend to'.

- Lastly, **Self-Attention** empowers a neural network to comprehend a word within the context of the neighboring words, facilitating a deeper understanding of the relationships between words in a sequence.

2.1 Architecture

Transformers are composed of two essential components: an **encoder** and a **decoder**. The encoder processes the input sequence, while the decoder focuses on generating the target output sequence.
These systems operate on a sequence-to-sequence learning. They analyze a sequence of tokens (words for example) and predict the subsequent word in the output sequence by iterating through layers within the encoder. Each encoder layer generates encodings that define the relationships among different parts of the input sequence, passing these encodings to subsequent encoder layers. The decoder then utilizes all these encodings, derived from the input context, to generate the output sequence.

In the architecture of a Transformer, we find the following steps :

- The **Input Embedding** step : In this phase, words are converted into numerical vectors. Word embedding, also called word vectorization, is a method that represents words using numbers. It creates numeric vectors enabling words with similar meanings to have close numerical representations in an 'embedding space'. To make this concept easier to understand, we often illustrate it in a 2-dimensional space, as shown in Figure 1 for a clearer visualization.

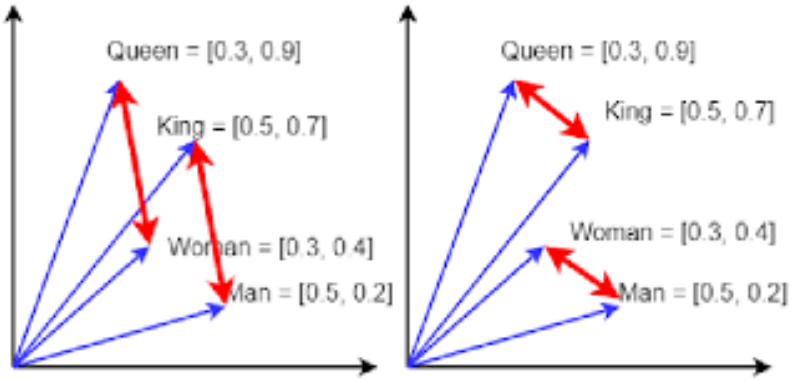

Figure 1: Word embedding representation

As we can see on this chart, we can perform mathematical operations on the vectors. For instance :

$$\text{Women} + \text{King} - \text{Men} \approx \text{Queen}$$

We find various established methods for word embedding include *Term Frequency-Inverse Document Frequency (TF-IDF)*, *Bag of Words (BOW)*, and *Word2Vec*.

- The **Positional Encoding** step : Positional encoders create vectors that express how words are positioned relative to each other in a sentence in order to handle varying meanings of the same word across different sentences. Typically using sin and cos functions, these vectors add positional context to word representations. According to the dimension i, the positional encoding is defined as follows :

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

- The **Multi-Headed Attention** block : It computes attention vectors for each word, capturing their relationships within the sentence.
  However, sometimes these attention vectors might overly focus on individual words rather than their broader context. For example the attention vector may weight its relation with itself much higher, which is true but useless since we are more interested in interactions with different words.
  To mitigate this, multiple attention vectors are generated for each word, and through a weighted average, they form the final attention vector. This process helps balance the attention across various words in the sentence.

- The **Feed Forward** block : This section fine-tunes the output vector, making it better suited for the next decoder block or a linear layer. It is applied individually to each position in the vector. Here is what it does: it undergoes two transformations using linear operations with a ReLU activation in between. Mathematically, the feed-forward network (FFN) for one hidden layer is expressed as follows:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

- The **Normalization** layer : The output of a sub-layer (the multi-head attention one or the feed forward one) becomes $LayerNorm(x + Sublayer(x))$ where x is the input sequence, and $Sublayer(x)$ is the function implemented by the sub-layer itself.

- The **Masked Attention** block : In this self-attention sub-layer, changes are made to block connections between certain pairs of word, for instance, it is important to prevent a position from focusing on or considering future positions. To do this before the softmax stage, we use a mask matrix, M. In this matrix, we set the value to $-\infty$ where we want to stop the attention link, and we keep it as 0 everywhere else :

$$MaskAttention(Q, K, V) = softmax(M + \frac{QK^T}{\sqrt{d_k}})V$$

  This "masking" technique, along with the adjustment that shifts the output embeddings by one position, guarantees that predictions for a position i rely only on the known outputs at positions that come before i.

The architecture of a Transformer is illustrated as follow :

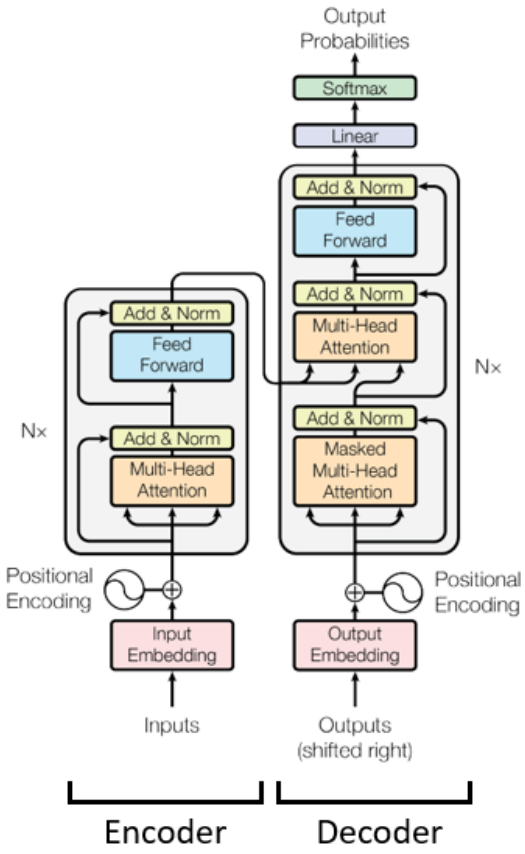

Figure 2: Architecture Transformers

The **encoder** section of this architecture (on the left side of the Figure 2) begins with input embedding, where words are transformed into numerical vectors. Then to handle varying meanings of the same word across different sentences, positional encoders come into play.
Combining input embedding with positional encoding yields word vectors enhanced with both meaning and context. These contextualized word vectors enter the encoder block that is composed of a stack of N identical layers, each layers has two sub-layers consisting of two essential components: multi-headed attention and a feed-forward layer. These two blocks are each followed by layer normalization.
Attention, managed by the multi-headed attention block, decides which parts of the input are most important, and the feed-forward network are used to transform the attention vectors into a form that is "digestible" by the next encoder block or decoder block.

The **decoder** section of this architecture is illustrated on the right side of the

Figure 2. In the training phase, the decoder processes the output sentence to convert each word into a vector. To contextualize these vectors within the sentence, positional vectors are added, providing the notion of word placement in a sequence. The decoder block consists of a stack of N identical layers, each layers has three sub-layers, two of which mirror those in the encoder block : a multi-headed attention, a feed-forward, and a masked multi-head attention layer. As well as for the encoder, connections around each of the sub-layers are followed by layer normalization. Finally, after the last decoder, a final linear transformation and softmax layer are employed to generate output probabilities across the vocabulary.

Moving into the training phase, a critical challenge faced by neural networks, particularly in earlier layers, is the phenomenon known as the **vanishing gradient problem**. The vanishing gradient problem plagues neural networks during training, particularly affecting earlier layers. This occurs when the gradient of the loss function with respect to network weights becomes extremely small. Consequently, during stochastic gradient descent (SGD), updates to weights proportional to these tiny gradients do little to advance the network's learning, leaving weights effectively stuck. This issue ripples through the network, impairing its learning ability.
Contrary to the problematic nature of the sigmoid function due to gradient vanishing, transformers employ softmax layers in each block to compute attention scores. This adaptation within transformers mitigates the vanishing gradient problem seen in traditional networks.

2.2 Causal Language Model (CLM)

Language modeling involves training a model on a specific set of text, often specifically designed to a particular domain. Approaches like BERT, using masked language modeling (MLM), and GPT2, using CLM, are crucial in training different transformer-based models.
In CLM, the model learns to predict the next word in a sentence based on the previous ones. During training, the model receives input words and predicts the probability of the next word. Its performance is evaluated (loss) by comparing its predictions to the actual subsequent words, which are the input words shifted by one step.
This technique is commonly utilized in autoregressive models such as GPT, which operate in a left-to-right Transformer arrangement. These models consider only the words that precede each word in the sequence, disregarding those that follow it, preventing from looking at the words it is meant to predict during training, ensuring it learns based on prior context.
A fine-tuned CLM model excels in generating logical text by predicting words one

by one, ideal for tasks involving generating text and creating summaries. However, in terms of understanding information from both earlier and later words, it might not be as robust as MLM models.

2.3 Self-Supervised learning

Self-supervised learning is a machine learning method where a model learns from one part of the input to understand another part. This approach transforms an unsupervised problem into a supervised one by by auto-generating the labels. To make use of the huge quantity of unlabeled data, it is essential to set the right learning objectives to get supervision from the data itself. In self-supervised learning, the goal is to identify hidden information in the input from any unhidden part of the input. For instance, in natural language processing, given a few words, this method enables completing the rest of the sentence by learning from the available data.

2.3.1 Training framework of Generative Pre-trained Transformer (GPT)

GPT models, known for their effectiveness in tasks like language generation, text classification, and translation, are neural networks that rely on a Transformer architecture. These models excel in creating human-like text due to extensive training on language data. Widely used in various language tasks such as completing text, answering questions, and translating languages, GPT models are particularly valuable for chatbots, content creation, and language modeling. Their ability to process word sequences in parallel enables more accurate and natural-sounding language generation.

The training procedure consists of two stages:

- Unsupervised Pre-Training : The model needs pre-training on a large corpus of text before being fine-tuned for specific tasks. During pre-training, the model learns to predict the next word in a sentence. To achieve this, a random word is removed from the sequence, and the model is trained to predict that missing word.

- Supervised fine-tuning : After pre-training, the model can be fine-tuned for a specific task, like text classification or language translation. This involves training the model on a smaller dataset related to the task at hand. Throughout fine-tuning, the model's parameters are adjusted to enhance its accuracy specifically for the assigned task.

3 Reinforcement Learning with Human Feedback

Before delving deeper into the reinforcement learning and its human feedback reward method, let's first establish a few key concepts essential for a comprehensive understanding of the following discussion.

3.1 Markov Decision Processes (MDPs)

Markov decision processes are theoretical frameworks that are used to formalize sequential decision making. They are the basis for problems that are solved with **reinforcement learning**. Here are the concepts and metrics involved in MDPs :

- an **Agent**: Software programs that act as *learners*, making intelligent decisions by interacting with the environment through actions and receiving rewards based on their actions.
- an **Environment** : The demonstration of the problem to be solved. It can be a real-world environment or a simulated environment with which the agent will interact.
- an **Action** a : A choice among possible actions
- a **State** s: Position of the agents at a specific time-step in the environment. When an agent takes action, the environment rewards the agent and updates its position to a new state.
- a **Reward** $r(s, a)$: Positive or negative gain acquired by taking action a in state s.

So in MDPs we have :

- a (finite) set of states S
- a (finite) set of actions A
- a (finite) set of rewards R

In a sequence, the agent interacts with the environment. At every step in this sequence, the agent receives information about the environment's current state s_t. Using this information, the agent chooses an action to take a_t. This action leads the environment to a new state s_{t+1} and the agent receives a reward based on what it did in the previous step r_t.

This creates a trajectory that shows the sequence of state, action and rewards, and htat can be represented as :

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ...$$

The following diagram illustrates the entire idea :

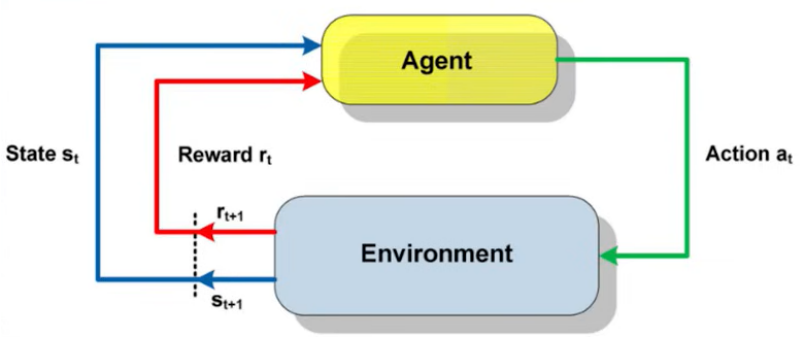

Figure 3: MDPs representation

Given that sets S and R have finite elements, the random variables s_t and r_t follow specific probability distributions. This means that there are probability assigned to all possible values that r_t and s_t can take. These probabilities rely on the state and action preceding the current time step, $t-1$.
For instance, if we consider $s \in S$ and $r \in R$. There exists a probability linked to the occurrence of $S_t = s$ and $R_t = r$. This probability is determined by the specific values of the previous state $s' \in S$ and action $a \in A(s')$ ($A(s)$ being the set of possible actions from state s).
Thus, for all $s \in S$, $r \in R$, and $a \in A(s')$, we define the *probability of the transition to state s' with reward r from taking action a in state s* as

$$p(s, r|s', a) = Pr(S_t = s, R_t = r|S_{t-1} = s', A_{t-1} = a)$$

3.2 Reinforcement Learning

In the most common machine learning application, people use supervised learning. This approach involves furnishing input data to a model as well as the output that the model should produce, allowing the model to learn and adjust its parameters through techniques like backpropagation. However, this method has two main limitations :

- Training dataset creation : Implementing supervised learning necessitates the creation of a training, dataset which is not always straightforward

- Limitation in Human Imitation : Training a neural network model to mimic human actions inherently restricts the agent's performance to human levels

One alternative to avoid these limitations is the **reinforcement learning**.

Indeed, reinforcement learning refers to a set of methods that enable an agent to learn how to choose which action to take autonomously. Immersed in a given environment, the agent learns by receiving rewards or penalties based on its actions. Through its experience, the agent aims to discover the optimal decision-making strategy that allows it to maximize the accumulated rewards over time.

In reinforcement learning, the network responsible for translating input frames into output actions is called the *policy network*. A common method to train this network is through policy gradients. Here is how it works: Initially, we begin with a completely random network, we feed a frame into this network, which generates random output actions. These actions are then applied in the environment, producing the subsequent frame. With each action, the agent receives a reward. The main objective for the agent is to optimize its policy in order to maximize the cumulative reward received.

To train the policy network effectively, we gather a substantial set of experiences by running numerous frames through the network. Random actions are selected, applied in the agent's environment, generating multiple instances of the agent's interactions. Initially, as the agent has not obtained useful information, high rewards are unlikely. Occasionally, the agent might receive rewards through chance occurrences.
For every episode, regardless of the reward's sign, we can compute gradient. In episodes where positive rewards are obtained, the Policy Gradient will use normal gradients to increase the probability of these actions in future iterations. On the contrary, in episodes with negative rewards, the same gradients are applied but multiplied by -1. This negation ensures that actions from unfavorable episodes become less probable in subsequent iterations.
Consequently, during policy network training, actions yielding negative rewards gradually diminish in likelihood, while those leading to positive outcomes increase in probability.

3.2.1 The Proximal Policy Optimization (PPO) example

PPO operates through two primary networks: the **Policy Network** and the **Value Function Network**, both neural networks processing input to generate specific outputs. The Policy Network takes a state as input and outputs an action probability distribution, while the value function network takes actions and states as input, producing a q-value representing the decision's quality as ouptut.

In practice, the agent begins at a random state, which is fed into the Policy Network to determine action probabilities. After taking an action based on this probability distribution, the agent receives a reward. This information (state, action, reward, and action probability) is stored in a data store for future network training. This sequence of steps is repeated for an episode or a fixed number of time steps, generating batches of data. To train the networks, these batches are used.

The value function network calculates q-values for state-action pairs, while total future rewards quantify performance. From this, the "advantage" is computed and utilized to determine the loss, which is then backpropagated through both networks for parameter updates. This process iterates across all data batches, allowing the networks to progressively improve as they learn together over time.

In practice, fine-tuning a language model using PPO generally involves three main phases:

- The **Rollout Phase** : During this phase, the language model generates a response or continuation based on a given prompt or query, which often serves as the starting point for the sequence generation.
- The **Evaluation Phase** : During this phase, the generated response along with the initial query is evaluated in order to obtain a reward. This reward is defnied as a single numerical value, and can come from a dedicated evaluation function, a separate model, human judgment, or a mix of these approaches.
- The **Optimization Phase** : This phase is the most important step in the process. It involves utilizing the query/response pairs to compute the log-probabilities of tokens present in the generated sequences. This calculation is performed using both the fine-tuned model and a reference model, often the original pre-trained model. The KL-divergence between the outputs of these models serves as an additional reward signal. This reinforcement helps ensure that the generated responses remain within an acceptable deviation from the language model reference.

This process is illustrated in Figure 4 :

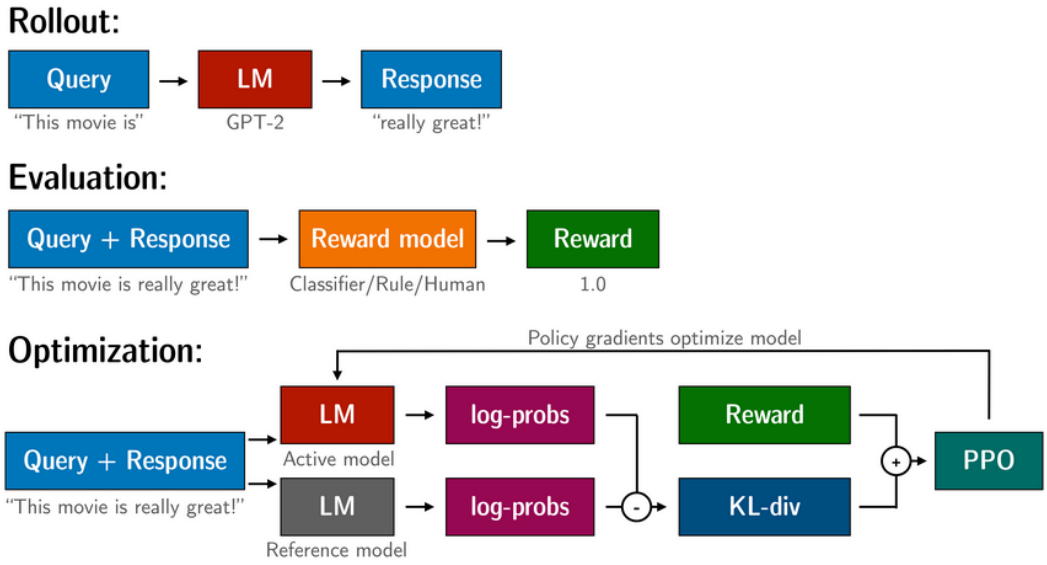

Figure 4: PPO representation

3.3 Human Feedback and the example of Chat GPT

During the agent's learning process using reinforcement learning, humans can also act as mentors, providing feedback. This interaction accelerates the agent's learning trajectory and enhances the agent's ability to produce responses that better align with human preferences. In this scenario, although the agent continued to adhere to an algorithm, the human intervened by nudging the agent toward what they believed to be the best actions.

In the practical application of ChatGPT, RLHF is employed through a two-phase process:

Firstly, the system trains a **rewards model** to act as a human advisor for ChatGPT. This rewards model, based on a GPT architecture, assesses the quality of responses generated by the agent through a score. Indeed, it takes a question and answer as input and the output of the GPT network is a number that says how good was this answer to this input question.
To train this model, multiple unique answers are obtained by presenting a question repeatedly to a pre-trained ChatGPT. Human evaluators then rank these responses from best to worst, creating a dataset used to train the rewards model. Once trained, this rewards model effectively evaluates the quality of ChatGPT's responses to specific questions.

Secondly, the rewards model, together with the **PPO algorithm**, are used to fine-tune ChatGPT. Indeed, ChatGPT generates responses to questions using PPO, and these responses, along with the original questions, are evaluated by the rewards model. The score produced by the rewards model, evaluating the response's quality, becomes part of ChatGPT's loss function. Backpropagation is then performed, allwing ChatGPT to learn and adjust. This one iterative process continues over multiple iterations, gradually enhancing ChatGPT's response quality.

Here is an example of RLHF from OpenAI :

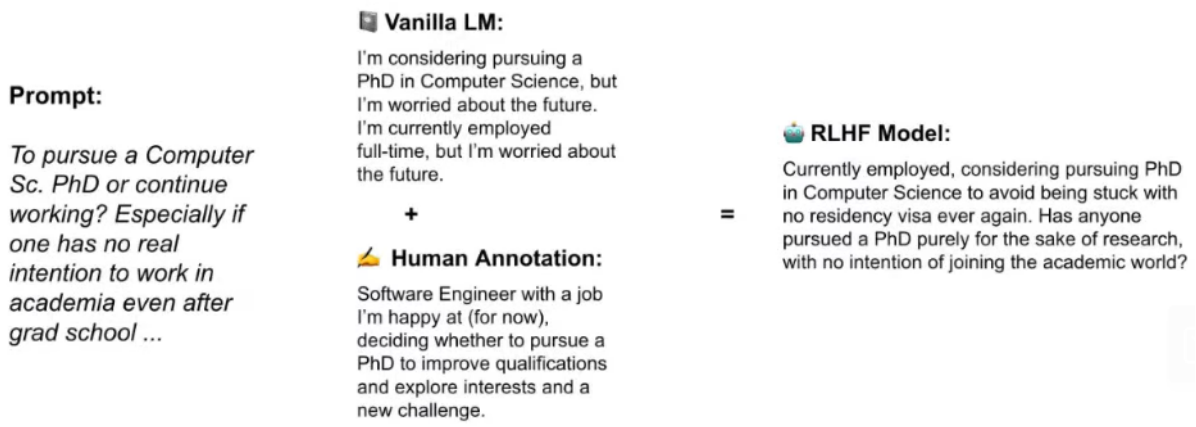

Figure 5: OpenAI experiment with RLHF

4 Conclusion

In summary, we have explored the architecture and mathematical foundations of Transformers, marking a significant innovation in language processing. We have explained the concept of causal language models, shown by models like GPT, shedding light on their self-learning methods.

Shifting focus, we have delved into the realm of reinforcement learning, starting with some key concepts that are the Markov Decision Processes. Finaly, we introduced the integration of human feedback in the reinforcement learning, a crucial aspect in enhancing learning in these systems

5 Bibliography

"Transformers, explained: Understand the model behind GPT, BERT, and T5" , YouTube video, Google Cloud Tech

"What are Transformers (Machine Learning Model)?", YouTube video, IBM Tech

"Transformer Neural Networks, YouTube video, CodeEmporium

What is Gradient Clipping? https://towardsdatascience.com/what-is-gradient-clipping-b8

Decoding Transformers' Superiority over RNNs in NLP Tasks https://hackernoon.com/decoding-transformers-superiority-over-rnns-in-nlp-tasks

Reinforcement Learning : Markov-Decision Process (Part 1) https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da

Reinforcement Learning : Définition et application https://datascientest.com/reinforcement-learning

La revue IA https://larevueia.fr/quest-ce-que-le-rlhf-rl-from-human-feedback/

https://medium.com

Hugging Face's TRL for Reward Modeling: https://github.com/huggingface/trl/blob/main/examples/scripts/reward_modeling.py

OpenAI's RLHF Papers for ChatGPT: https://arxiv.org/abs/2203.02155

RLHF Paper: https://proceedings.neurips.cc/paper_files/paper/2020/file/1f89885d556929e98d3ef9b86448f951-Paper.pdf

AI's Work on RLHF: https://arxiv.org/abs/2204.05862

RLHF lecture at Stanford: https://t.co/Uk8g6mBTrD

CleanRL's PPO Implementation in PyTorch: https://github.com/vwxyzjn/cleanrl/tree/master

PPO Paper: https://arxiv.org/abs/1707.06347

PPO Explained (YouTube Video): https://www.youtube.com/watch?v=5P7I-xPq8u8&ab_channel=ArxivInsights

Live Coding PPO (YouTube Video): https://www.youtube.com/watch?v=hlv79rcHws0&ab_channel=MachineLearningwithPhil

Blog post for RLHF with ppo: https://huggingface.com/blog/the_n_implementation_details_of_rlhf_with_ppo

RLHF explained: https://huggingface.co/blog/rlhf

RLHF course: https://www.youtube.com/watch?v=2MBJOuVq380&ab_channel=HuggingFace