Homebrew LLM fine-tuning using my personal chat messages

Matteo Bonacini

Abstract

Fine-tuning a Large Language Model (LLM) is a process that allows a model pre-trained on generic data to perform well when presented with task-specific data. However, acquiring good-quality data to use for fine-tuning can be quite hard. In this paper, I explore werther it is possible to fine-tune a LLM using my personal WhatsApp chat messages. I show how I trained two different models (GPT2 100M and Gemma 2.5B) using two different strategies (full fine-tuning and QLoRA) and how the results compare. I also give an idea about what the costs of fine-tuning are for small scale models. In the end, I found that Gemma yielded better results than GPT2 and I provided some example outputs to compare the two models.

CONTENTS

1 Introduction

Machine learning is nothing more than finding the minimum of some data-dependant functional. For Large Language Models (LLMs), training a model means minimizing the cross-entropy loss functional, averaged through all the examples of some training dataset. Then, after one (possibly local) minimum has been found, we can start to generate new (possibly useful) data.

Now, if we feed a model data that it is not used to seeing, it will perform poorly. This is because the new data will shift the loss functional landscape and the model will not represent a minimum anymore. This is referred to as a *domain adaptation problem*. Over the years, many different strategies have been developed to solve this problem with minimal computation costs. For LLMs, we use a process called *fine-tuning*. To better explain why we need fine-tuning, let us have a look at we can train a model that behaves like the state-of-the-art ChatGPT.

1.1 How is ChatGPT created

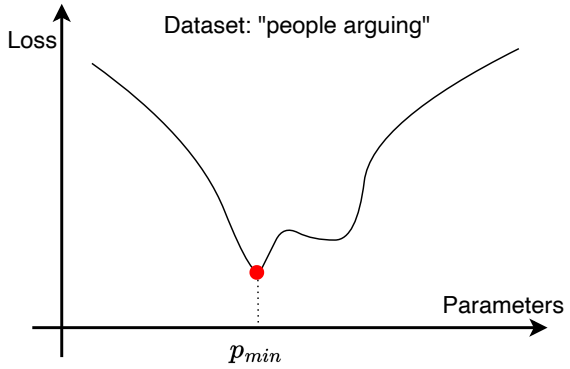
The starting point is the assumption that *the inner workings of language are universal and indipendent of what one is talking about*. If this is true, it means that we expect the cross-entropy functional to have the same, broad, funnel-shape appearence, regardless of what the training data is (as long as it is proper language). The shape of the innermost part of the funnel, however, varies depending on the specific topics that are present in the data. This behaviour is schematized in Figure 1.

The most broad recollection of text available to us humans is *the entire internet*. Thus, we start by teaching a model *how to speak* by feeding it just that. This results in a model that will try to complete whatever webpage you give them. This can be useful in itself but, for the purpose of this text, we would like to obtain something that behaves like an helpful chatbot. Thus, what we do is to take a large set of high-quality man-made examples of a conversation with a helpful chatbot, and continue training our model on that. This step is crucial because it allows our model to adapt to the new data domain.

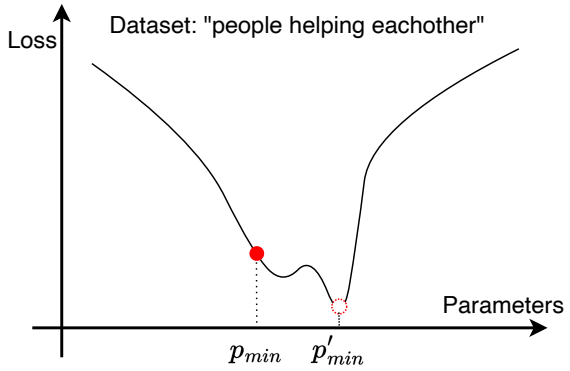
For the purposes of this text we could stop here, but I want to point out that the actual ChatGPT fine-tuning process includes two more steps, which greatly improve the quality of the resulting model. These consist in 1) building another, different model for ranking ChatGPT responses and 2) using this to perform a reinforcement-learning training round that will make ChatGPT prefer high-ranked responses over lower-ranked ones. After all these steps, we would have our final ChatGPT model.

1.2 Overview of my work

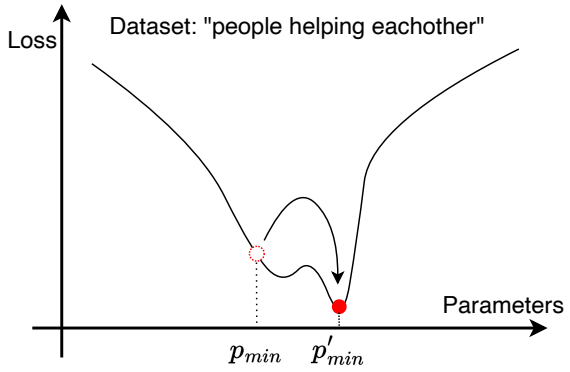
I have fine-tuned two different LLMs on my personal chat messages, using different techniques. My goals were, first, to see wether this would prove useful in any way (i.e. if it would learn to respond to messages like I would) and, second, to learn how fine-tuning is performed in a real-world environment.



(a) Result of a training of a LLM over the ”people arguing dataset”. The red dot indicates the parameter for which we have a minimum of the loss.



(b) If we keep the parameters fixed and switch the dataset, the landscape will change and we will not be at a minimum anymore.



(c) Through fine-tuning, we slightly alter the parameters in order to reach the new minimum.

Figure 1: The process of fine-tuning, visualized. The three graphs represent the loss landscape over two fictional text datasets. One point in the graphs is computed by taking the loss function, fixing its parameters and computing its average over the entire dataset. The graphs show how the broad shape of the landscape is the same, but the small details are different depending on the dataset.

There are several steps that need to be taken in order to be able to achieve any results. In the following paragraph I will explain everything I have done, starting from data acquisition up to the model training. I will give an overview of some different fine-tuning techniques and I will show how I made my hardware and software choices. In the end, I will show some example prompts, along with the output obtained from my models.

2 Preparing the data

Depending on which chat messages one wants to extract, doing so can be very easy or very hard. In Europe, thanks to GDPR regulations[1]it is, in general, always possible to export one's personal data from an online messaging service. The format in which the data is exported, however, is not always easy to handle programmatically. I chose to export my data from the WhatsApp Messenger application, due to the following reasons:

- It is the application where I have the most chat messages saved and
- extracting messages in a easy-to-use format from WhatsApp can be straightforward.

There are different ways of extracting chat data from WhatsApp. If one is concerned with exporting a single chat, then this can be done simply by opening it in the application and using the *Export chat* function in the context menu. If one wants to export multiple chats, however, the process becomes more involved. Since I have and Android phone with *root*[2]access, I was able to access the (unencrypted) databases that contains all the data in the application. The chat database is a single *sqlite3* file: `msgstore.db`; its location is usually `/data/data/com.whatsapp/database` (but it may vary depending on the device model and Android version). Another sqlite3 database, `wa.db`, found in the same folder as the previous one, contains all the information relevant to one's contacts. It is useful to have access to this database as well, in order to retrieve the sender name of the chat messages.

Interacting with a sqlite3 database can be done with the appropriate command-line tool[3]. The tool loads the database into memory and allows the user to interact with it through *queries* written in the `SQL` language[4].

2.1 Parsing the WhatsApp database

The database schema is proprietary and closed-source. This means that, in order to extract meaningful data, one has to do some reverse-engineering-work. One thing worthy of note is that WhatsApp records every user interaction in the same table. For example, the creation, deletion or modification of a group chat is recorded to the database as an empty message with some added special flags. Since we want to extract *clean* data, we need to construct a query that parses all the messages and returns them in order of time and chat.

Starting from scratch, I believe one would need about a day of work to construct such query. Luckily, there exist some open-source projects online that have already done the work for us. I was able to use the project `WhatsApp-Chat-Exporter` from `KnugiHK` on GitHub[5], along with some of my own python code, to parse all the messages into a more usable *JSON* format. Just for reference, the main query they used to parse the database is shown in appendix A, along with an example of the output JSON file.

2.2 Preparing the prompts

The data used for fine-tuning can be prepared in different ways, depending on the approach one wants to take. In my case, I chose to use *causal fine-tuning*, as described in [1].

Causal finetuning is just a particular approach used to mask the training inputs that tries to mimic how some real-life interactions happen in written text. In order to explain the reasoning behind it, I think it is best to make an example.

Let us take the following snippet of a chat of mine.

```
< Chat with: Crew >
Giuseppe:    What game are we
             playing tomorrow night?
Stef:        I would like to play
             Nemesis
Giuseppe:    Yes, that sounds fun
Matteo (me): Ok, Nemesis is fine
             for me too
```

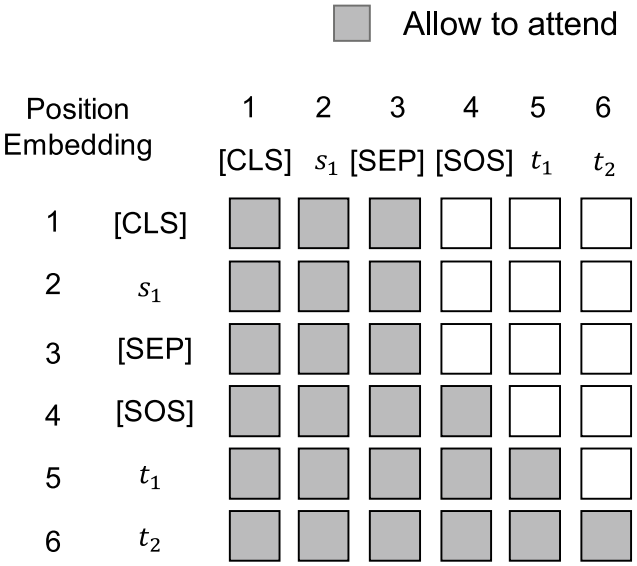
Here, the response I gave takes into account the information contained in all of the previous messages. From an attention point of view, *each one of the tokens in my response has access to the information of every token in the messages sent before mine*. While answering, however, I wrote the text sequentially, one word after another. This means that *each token in the response also has access to the previous tokens in the response (and itself)*.

We can force a transformer model to have a similar behaviour by using an attention mask, like the one shown in Figure 2. To be more precise, here is the list of steps I use to generate the input examples to the model:

- Open a chat.
- Start reading the chat messages until a message from me is found.
- Concatenate together all of the consecutive messages from me. These will be my "response".
- Now, going backwards, concatenate together all of the previous messages until block size is reached[6]. These will be the "prompt".
- Prepare the causal mask for the prompt and the response. All the tokens in the prompt will be

1. http://data.europa.eu/eli/reg/2016/679/oj
2. https://en.wikipedia.org/wiki/Rooting_(Android)
3. https://sqlite.org/about.html
4. https://www.sqlite.org/lang.html
5. https://github.com/KnugiHK/WhatsApp-Chat-Exporter/

able to attend each other; the tokens in the response will be able to attend the previous token in the response and every token in the prompt.
- The input to the model is the concatenation of prompt and response, appropriately masked. The expected output are the input tokens, shifted by one, with appended a special "end of text" token.
- Continue reading messages to generate further input examples. When the end of the chat is reached, repeat the algorithm on another chat, until there are no more chats left.



(a) Causal Fine-Tuning

Figure 2: Reprinted from [1]. Example for a causal fine-tuning mask. The `[CLS]`, `[SEP]` and `[SOS]` tokens indicate, respectively: the start of the prompt, the end of the prompt and the start of the response. The s_i and t_i tokens indicate, respectively, any tokens from the prompt and the response. A gray box indicates that the respective input token (row index) can attend to another input token (column index) inside an attention head.it

This algorithm does not guarantee that every input example will fit into block size and thus some additional checks need to be made down the line. In particular, it fails when one or more of the messages sent by me are longer than block size. These are extremely rare and can be safely discarded.

There is also some cleanup that needs to be done afterwards. We need to discard every example that contains any media (i.e. images and audio messages) and also any example that contains too many system messages (that is, messages that are send by WhatsApp itself and are not written by a human). I also chose to discard all the examples which contains messages before the year 2021 (which add up to about two thirds of the total messages), as my language has definitely changed with time. Another thing that we can check down the line is the delay between two messages in the response. If it is too high (e.g. more than one hour) it probably means that, while the first message is a response to the previous conversation, the second message is the start of a new conversation by me, and it should be removed by the response.

The full data-processing workflow is shown in appendix A.

2.3 Putting some numbers into perspective

It can be interesting to look at how much data can one expect to obtain in this way. In my case, the WhatsApp database file took `2.0GB` of space. Most of that, however, is just redundant information. All of the messages concatenated and saved in text form only take up `95.1MB` of space. Using the algorithm described above I was able to obtain a total of `43443` example inputs.

This is very little data compared to the amount that is usually used for pretraining an LLM (for reference, GPT2 was trained on `40GB` of data [2]); it is plenty[7], however, to perform fine-tuning.

Running the code used to prepare the examples takes some time. Starting from the raw WhatsApp database, it takes about 45 minutes to get to the final input examples. The data, however, needs to be processed only once and can then be reused for different fine-tuning runs.

3 Exploring different fine-tuning strategies

In this section I will give an overview of some common fine-tuning approaches. In my work I only implemented the first two methods (full fine-tuning and QLoRA). I also wanted to give an overview on RAG (Retrieval Augmented Generation), as it is somewhat related to the first two methods.

3.1 Full finetuning

Full finetuning is the most straightforward approach. It consists in:

1) Taking a pre-trained LLM and
2) continue training that network on the data we want to fine-tune on.

Here, the training process used for fine-tuning is the same one that gets used for pre-training. That is, we compute the gradient of the loss function with respect to some batch of data and use that to update the model

6. When concatenating messages for the response, count how many tokens are used. Then, when concatenating messages for the prompt, add to the count. Once the count reaches block size, we stop.

7. The number of data points required for fine-tuning varies greatly depending on the model used and on the type of fine-tuning one is going to do. A full-fine-tuning run requires prompts in the order of 100k-10k [3]. Few-shot fine-tuning requires a lot less data [4]. OpenAI recommends 50-100 prompts when fine-tuning using their cloud services.

parameters. The only difference is that, here, we use a *lower* learning rate than what was used in the pre-training step. This is crucial, as during fine-tuning we have access to a smaller quantity of data (usually by some orders of magnitude) and we want to both

- avoid overfitting and
- avoid *catastrophic forgetting*.

Catastrophic forgetting is akin to overfitting and consists in the model *forgetting* what it had learned in the pre-training stage. This would result in a model that knows very well how to complete the data it had seen during fine-tuning, but that does not know how to use natural language at all.

Full fine-tuning is commonly regarded as the most powerful strategy [5], but it has one main drawback. That is, since we need to keep track of the gradient through the whole network, it means that the memory and computing power requirements are *very high*. This starts to become a problem as the parameter count in a network becomes higher and higher. As an example, fine-tuning the biggest llama2 model (70 billion parameters) would take $70E9 * (8 + 2)bytes = 700GB$ of memory (the optimizer uses 8 bytes per parameter and storing one parameter in floating point format takes 2 bytes). There are techniques to reduce this number by splitting the memory usage across different GPUs and RAM, such as DeepSpeed and ZeRO [6], [7], but the hardware requirements would still be beyond what any normal person could achieve (without resorting to more expensive cloud resources). This means that full-finetuning at home is limited to very small models.

Some variants of full-finetuning are being developed, such as DEFT (Data Efficient Fine Tuning) [3] and PEFT (Parameter Efficient fine tuning) [8]. DEFT aims to reduce training times by using only a smaller, representative subset of the training data. In their test case, the authors of [3] were able to show that their model could reach the same fine-tuning result using 70% less data. PEFT, in the other hand, aims to reduce training times by reducing the number of parameters that are altered in the fine-tuning process. There exist a series of different PEFT techniques, which I will not discuss here [8].

3.2 (Q)LoRA

LoRA (Low-Rank Adaptation) [9] is a clever approach that leverages the findings of [10], [11] to allow for a fine-tuning process that both produces very good results and is very efficient in terms of memory usage and parameter count.

There are two main ideas that make up LoRA. The first one is that, after the fine-tuning process, we want to save the weight matrix of the new network W' as the sum of the weight matrix of the old network W_0 and some delta:

$$W' = W_0 + \Delta W.$$

The second idea is that we assume that we can encode the ΔW matrix using the product of two low-rank matrices A and B:

$$\Delta W = AB.$$

This assumption is justified by previous works [10], [11] which have shown that *the intrinsic dimensionality of a problem can be orders of magnitude lower* than the dimensionality of the parameter space of a neural network that can solve said problem.

We can apply these ideas to any subset of the weight matrix of any neural network. In particular, any matrix multiplication layer, becomes:

$$y = W_0 x \qquad \text{(before finetuning)}$$
$$y = (W_0 + \Delta W)x = (W_0 + AB)x \qquad \text{(after finetuning)}.$$

To better understand the dimensionality of the matrices involved, please refer to Figure 3.

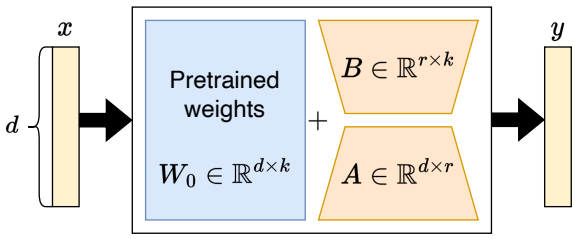


Figure 3: Schematic representation of a LoRA adapter. During inference, the W_0 and AB matrices get added together to generate the output. At the beginning of training, the matrix A is initialized as gaussian and the matrix B is set to zero (so that $\Delta W = 0$). In a real-world application, many of these adapters are added at different matrix multiplication layers of a single neural network.

QLoRA [12] builds on top of LoRA by adding *quantization* to the models during training and storage. "Quantizing" a model means reducing the number of bits that are used to store the model parameters. This results in a way lower memory and storage usage, but it comes with a degradation of performance. The authors of [12] presented some clever ways of quantizing the data that result in minimal performance degradation.

When combined, LoRA and QLoRA drop the memory requirements for training a LLM by a factor of ten. This is a very good compromise to the slightly lower performance that they provide.

3.3 RAG

The RAG (Retrieval Augmented Generation) approach is commonly used in cases where the parameter-based knowledge of a LLM is not sufficient [13]. With RAG, an additional *information retrieval* component gets added in between a user prompt and a LLM response. This component works by vectorizing the initial prompt and using a series of algorithms to fetch relevant documents from a vector database. All of the fetched data gets bundled together with the original user prompt and then

gets fed to the LLM. Commonly, RAG implementations use a Dense Passage Retriever (DPR) [14] to fetch data from the vector database.

The RAG approach works very well in use-cases where we need to be able to obtain answers from an LLM that are backed by a factual source. As a matter of fact, a RAG enables a LLM to cite the precise source of any piece of information it outputs. Moreover, a RAG enables us to update the information sources over time, without having the need to re-train the LLM. Finally, a RAG drops the limit on how much information a LLM can memorize [15] and allows us to use lower-parameter-count models.

Another upside is that, in general, we do not need to alter the weights of a pre-trained network to use a RAG. The only components that gets trained is the retriever, which size is generally a lot smaller compared to the full LLM model. This, however, can also be seen as a limitation, as the performance of a RAG system will depend on how good its retrieval system is.

4 An overview of my training setup

In choosing my training setup I had to take three factors into consideration:

1) Data privacy
2) Training cost
3) Training speed

Factor 1) boils down to this: *I am dealing with strictly personal data and I want to make sure that it remains private*. This means that, if I were to use a cloud-based training solution, I would need to take all the necessary precautions to make sure that my data is not exposed to the internet. A well-built cloud infrastructure can be as secure–if not more–as a hard disk stored in one's personal computer, but it requires a time overhead for its setup. Keeping the data locally is the most straightforward way to ensure privacy, but it locks down the hardware choice (that is, you can only use the hardware that you have at home).

Factors 2) and 3) are strictly correlated: the more money one spends, the faster the training is. It is also worth to consider that, in order to train some of the larger models, very high-end GPUs are required, and this usually means that one must resort to a cloud provider.

With these considerations in mind, let us *delve*[8] into the choices that I made.

4.1 Hardware choices

In the end, I decided that I wanted to keep *all* of my data on my computer. This meant that I had to use my own GPU (a consumer-grade `RTX 2070`) for the training process. The main limiting factor of this card is that it has only `8GB` of video memory, which limits greately the size of the models I was able to train. We can compare it with the state-of-the art GPUs used to train larger models (Table 1).

8. The text in this article is all hand-written. My own writing style, however, may have been influenced by me having read other articles that were written with the help of ChatGPT

Model	RTX 2070	A100
VRAM	8GB	40GB/80GB
CUDA core count	2304	6912
Card cost (2024)	200€	$\sim$ 10k€
Running cost	$\sim$.02 €/h	2.5/6.5 €/h

Table 1: Comparison between my gpu and state-of-the-art machine learning GPUs. The training speed is roughly proportional to the number of CUDA cores. The amount of VRAM does influence training speed but, most importantly, enables a card to train larger models.

The hardware implicitly limits the size of the models I am able to train. Using `fp16` training, AdamW optimizer and a batch size of 8, I expect (at most) to be able to train a $100M$ parameter model (without (Q)LoRA):

$$\begin{aligned}[2\ bytes/parameter & & (fp16)+\\ 8\ bytes/parameter & & (AdamW)]\times\\ 8 & & (batchsize)\times\\ 100M & & (parameters) =\\ 8GB.\end{aligned}$$

$100M$ parameters are very little for a LLM. I could also train larger models by reducing the batch size, but this would slow down training. For the purposes of this work I chose to stay close to the 100M parameter mark.

4.2 Software choices

There are many different choices one can make in terms of software. Most of the open-source machine learning code available comes in the form of python libraries (PyTorch [16], TensorFlow/Keras [17]). These libraries are often used in conjuction with other libraries to enable faster developement. The state-of-the-art platform for in-house generative AI training is Huggingface[9]. They provide streamlined libraries for training, storing and sharing various models. In particular, their Transformers[10] library can be used to train and run LLMs.

I did some performance testing and I found out that the Transformers implementation of GPT2 runs about three times slower than an equivalent native implementation in PyTorch. I do not know why this happens, but it appears to be an incompatibility issue between different python libraries.

In the end, my software choices were dictated by the models I wanted to train. I chose to train two different models using two different fine-tuning strategies. First, I trained the smallest GPT2 implementation[11] using full-fine-tuning. Then, I trained Google's Gemma 2.5B model [18], quantized to 4 bits, using QLoRA. Both models had a similar number of training parameters, slightly above $100M$. The models were

9. https://huggingface.co/
10. https://huggingface.co/docs/transformers/index
11. I used an "italian" version of GPT2, since most of my chats are in italian. The model I used was proposed in this paper [19] and was obtained by stripping the embedding layers in OpenAI's GPT2 model and retraining them with italian text.

Model	GPT2-italian	Gemma-2.5b-q4
Software	nanoGPT	unsloth
Parameter count	117M	39M
Training speed	2.5it/s	2.1it/s
Disk space	476MB	1.94GB + 241MB

Table 2: Comparison between the two different models I have trained. The training speed is measured in (unbatched) iterations per second.

trained using, respectively, Karpathy's `nanoGPT`[12] and the `unsloth` library from UnslothAI[13].

Table 2 offers a comparison between the models and software I used for training.

5 Results and Conclusions

Unfortunately, I cannot publish the final trained models online, as they contain mostly private data. However, all of the code that I wrote is available on GitHub upon request.

5.1 Empirical results

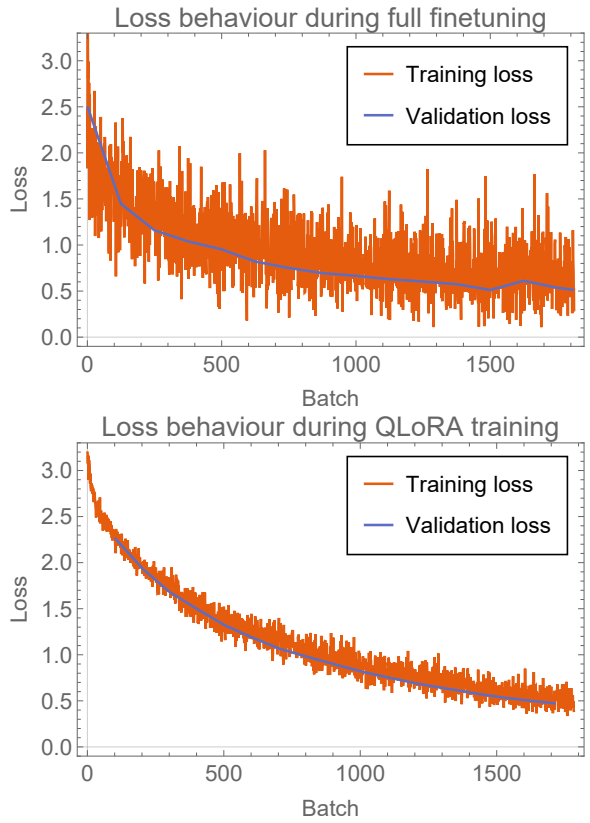


Figure 4: Loss behaviour during fine-tuning of different models.

I did some test runs (which I have not recorded) to tune the hyperparameters. Then, I did two training runs: one for the full finetuning of GPT2 and one for QLoRA tuning of Gemma. I chose the number of training samples so that the two models would take roughly the same time to train over one epoch. The samples are split 80/20 between the training and evaluation dataset. All the parameters that I used for training are shown in Table 3. The loss behaviour during the runs is shown in Figure 4.

The graphs of Figure 4 suggest that there may be two problems in the way I did the training. First, we can see that the loss function presents very large oscillations in the full fine-tuning run. This suggest that the learning rate may be too high and that some additional hyperparameter tuning may be needed. I was not able to fix this issue as training takes a lot of time and so does hyperparameter tuning. The second issue lies in the fact that, in both runs, the validation loss matches the training loss *too closely*. This suggests that the splitting of the data may have been done incorrectly. I suspect that this might be caused by the same text being repeated in different training examples[14].

Run	Full	QLoRA
Learning rate	1E-5	1E-4
Effective batch size	64	63
Number of samples	87k	112k
Block size	1024	
Dropout	.1	0
Weight decay	.01	
Optimizer	AdamW	
Learning rate decay	Linear	
Training epochs	1	
Gradient clipping	1	No
Data type	fp16	
LoRA rank	/	32
LoRA alpha	/	16

Table 3: Hyperparameters used for training.

Qualitatively, the results are acceptable (see appendix B for some example outputs). However, it is easy to see that both models may suffer from overfitting problems and the responses are not really human-like. Still, they did manage to learn the prompt structure well (both models do not appear to generate responses that are not from me and they know that the chat need to end with an "end of sentence" token). Between the two models, GPT2 got the *worst* results. The far better quality of the Gemma base model outweighed the loss in quality that comes with QLoRA training. I believe that the quality of the results could be improved further by tuning the hyperparameters and by increasing the base model size (Gemma 7b would probably yield way better results).

5.2 The cost of fine tuning

One final thing that I would like to focus on is how much it costs to do a fine-tuning run with the methods that I have described above. I also would like to provide

12. https://github.com/karpathy/nanoGPT
13. https://github.com/unslothai/unsloth

14. I feed the network the most messages I can fit in context size. This means that some training examples might contain also messages that are present in different examples. I have split the data randomly, and this means that some examples in the validation dataset may contain messages that the network has already seen in the training dataset.

some insight in what are the costs that one might incur into if they wanted to train some even bigger models.

In order to provide a baseline, I have measured the power consumption of my graphics card during training (Figure 5). I have not measured the power draw of the rest of the system[15], but we can estimate a higher bound of about $50W$. Table 4 shows the cost calculation for the training process and gives an estimate of how much the same training would cost if done using a cloud-based service.

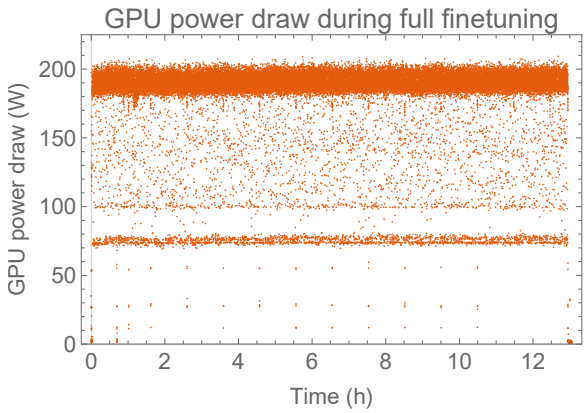
When moving to bigger models, the memory available to the GPU is a limiting factor. With only 8GB I was not able to train any model bigger than 2.5 billion parameters (even with a batch size of 1 and QLoRA, it would take too much memory). ZeRO optimizer offloading [7] would help with the memory issues, but it would slow down the training by an order of magnitude. The solution to this problem would be to either use cloud services or to buy a different graphics card. Cloud services are cheap when compared to the cost of buying a new graphics card, but they can get expensive if one needs to do many hyperparameter-tuning runs. On the other hand, RTX3090 or RTX4090 cards from Nvidia retail for 1000€ to 2000€ and offer about ten times the performance of a RTX2070 for double the power draw.

Run	Full	QLoRA
Average power draw(W)	179.5	180.7
Energy cost (€/kWh)	.2790	
Training time (h)	13.05	14.72
Training cost (local)	.65€	.74€
Estimate training cost (cloud)	2.5€	

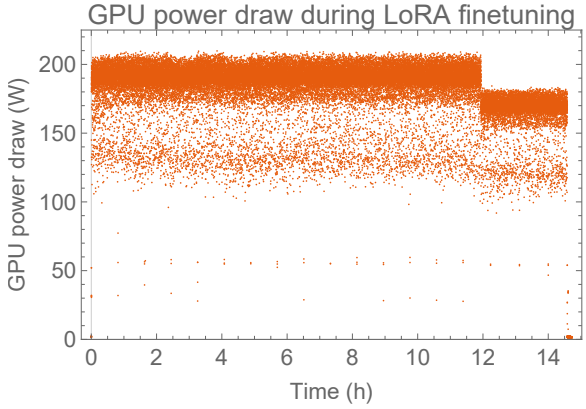
Table 4: Cost of doing a fine-tuning run. The estimate cost is computed by assuming a card that is 15 times faster than mine at a cost of 2.5€/h. These values are taken from Table 1

5.3 Conclusions

In this work I have shown that it is possible to fine-tune a Large Language Model on one's own personal chat messages. I have compared two different fine-tuning strategies and I have shown that using a better base model yields better results, even when full fine-tuning is not used. I have highlighted how much it costs to train a model at small scale and I have shown that it can be feasible to achieve better results with a reasonable budget by using a bigger model and cloud training.



(a) Power draw during the full fine-tuning run, using Karpathy's nanoGPT implementation and the smallest GPT2 model. The average power consumption was of $179.5W$. Here, we can see that the systems spends some time in the $75W$ region. This happens because the batch size I used barely exceeded the GPU memory and the system had to waste some time to transfer a small portion of the data to and from RAM at each training step. This did not result in a noticeable decrease in training speed.



(b) Power draw during the QLoRA run, using unsloth's implementation and the Gemma-2.5b model. The average power consumption was of $180.7W$. The dip in power consumption in the last two hours of training is due to the card thermal throttling (that last part of the training happened during daytime).

Figure 5: Power consumption of my GPU (RTX 2070) during training. The data points are taken at $1s$ intervals using the GPU-Z application. The periodic dips in power draw happen during the evaluation steps of the training process due to the system having to unload training data from memory and load in evaluation data.

15. Usually, when training at larger scales with multiple GPUs, the power consumption of the rest of the computer becomes negligible.

REFERENCES

[1] Hangbo Bao, Li Dong, Wenhui Wang, Nan Yang, Songhao Piao, and Furu Wei. Fine-tuning pretrained transformer encoders for sequence-to-sequence learning - International Journal of Machine Learning and Cybernetics, 2023.

[2] Alec Radford, Jeff Wu, Rewon Child, D. Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

[3] Devleena Das and Vivek Khetan. Deft: Data efficient fine-tuning for pre-trained language models via unsupervised core-set selection, 2024.

[4] Barry Z, Daniel Chang, Emma Qian, and Michael Agaby. Our humble attempt at "how much data do you need to fine-tune", Sep 2023.

[5] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources, 2024.

[6] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, and Yuxiong He. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

[7] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[8] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey, 2024.

[9] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[10] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes, 2018.

[11] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning, 2020.

[12] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[14] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering, 2020.

[15] Adam Roberts, Colin Raffel, and Noam Shazeer. How much knowledge can you pack into the parameters of a language model?, 2020.

[16] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[17] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[18] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. Gemma: Open models based on gemini research and technology, 2024.

[19] Wietse de Vries and Malvina Nissim. As good as new. how to successfully recycle english gpt-2 to make models for other languages, 2020.

APPENDIX A
PARSING THE WHATSAPP DATABASE

A.1 The Big query

The following SQL code is the main query used to parse the WhatsApp database.

```
SELECT jid_global.raw_string as key_remote_jid,
       message._id,
       message.from_me as key_from_me,
       message.timestamp,
       message.text_data as data,
       message.status,
       message_future.version as edit_version,
       message_thumbnail.thumbnail as thumb_image,
       message_media.file_path as remote_resource,
       message_location.latitude,
       message_location.longitude,
       message_quoted.key_id as quoted,
       message.key_id,
       message_quoted.text_data as quoted_data,
       message.message_type as media_wa_type,
       jid_group.raw_string as group_sender_jid,
       chat.subject as chat_subject,
       missed_call_logs.video_call,
       message.sender_jid_row_id,
       message_system.action_type,
       message_system_group.is_me_joined,
       jid_old.raw_string as old_jid,
       jid_new.raw_string as new_jid,
       jid_global.type as jid_type,
       group_concat(receipt_user.receipt_timestamp) as receipt_timestamp,
       group_concat(message.received_timestamp) as received_timestamp,
       group_concat(receipt_user.read_timestamp) as read_timestamp,
       group_concat(receipt_user.played_timestamp) as played_timestamp
FROM message
    LEFT JOIN message_quoted
        ON message_quoted.message_row_id = message._id
    LEFT JOIN message_location
        ON message_location.message_row_id = message._id
    LEFT JOIN message_media
        ON message_media.message_row_id = message._id
    LEFT JOIN message_thumbnail
        ON message_thumbnail.message_row_id = message._id
    LEFT JOIN message_future
        ON message_future.message_row_id = message._id
    LEFT JOIN chat
        ON chat._id = message.chat_row_id
    INNER JOIN jid jid_global
        ON jid_global._id = chat.jid_row_id
    LEFT JOIN jid jid_group
        ON jid_group._id = message.sender_jid_row_id
    LEFT JOIN missed_call_logs
        ON message._id = missed_call_logs.message_row_id
    LEFT JOIN message_system
        ON message_system.message_row_id = message._id
    LEFT JOIN message_system_group
        ON message_system_group.message_row_id = message._id
    LEFT JOIN message_system_number_change
        ON message_system_number_change.message_row_id = message._id
    LEFT JOIN jid jid_old
        ON jid_old._id = message_system_number_change.old_jid_row_id
    LEFT JOIN jid jid_new
        ON jid_new._id = message_system_number_change.new_jid_row_id
    LEFT JOIN receipt_user
        ON receipt_user.message_row_id = message._id
WHERE key_remote_jid <> '-1'
GROUP BY message._id;
```

A.2 The JSON output

The following JSON shows the structure of the parsed database.

```
{
    (...)

    "chat_id": {
        "name": "Name of the chat",
        "events": {
            (...)

            "message_id": {
                "from_me": true,
                "timestamp": 1460534446.381,
                "time": "10:00",
                "media": false,
                "meta": true,
                "data": null,
                "sender": null,
                "reply": null,
                "quoted_data": null,
                "caption": null,
                "sticker": false
            },

            (...)
        }.
    },

    (...)
}
```

A.3 Data processing workflow

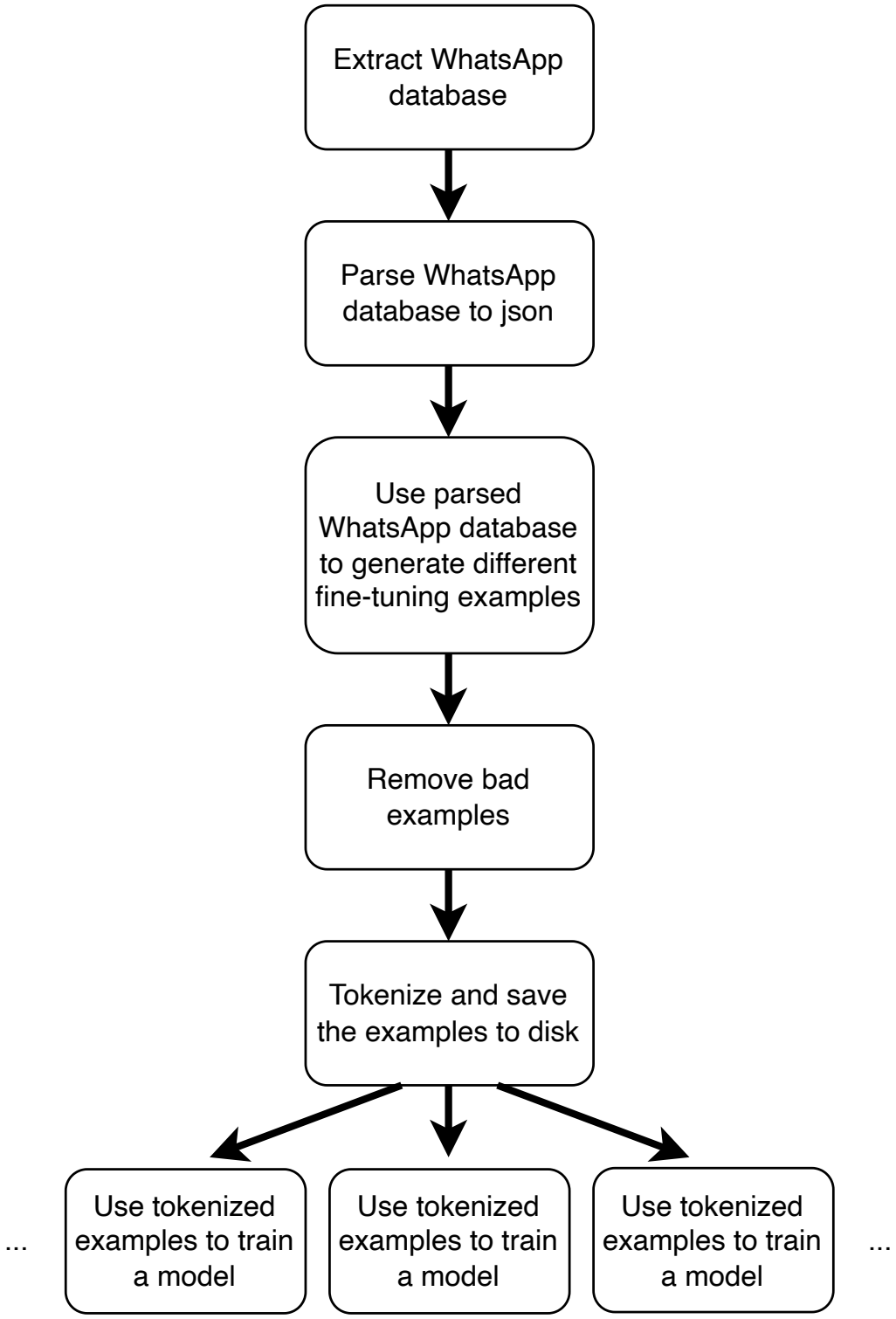


Figure 6: Data processing workflow

APPENDIX B
SOME EXAMPLE RESPONSES

These are some example prompts and responses that the models generated. The prompts are taken from my actual WhatsApp chats (they are taken from messages that I had not yet received when I created the training dataset). I cropped the responses from GPT2 as they are very long and they contain mostly gibberish. It seems that the model learned which words appear most frequently in my chats and used them out of context. When generating the responses, I have forced both models to respond by appending the keyword `<Risposta a:` to the end of the prompt; this worked very well.

B.1 Prompt 1:

```
<bos><Chat con: Giuseppe>
Giuseppe: Comunque se porti il mazzo
Giuseppe: Portiamo tutti i mazzi
Giuseppe: e giochiamo
```

Response from Gemma:

```
<Risposta a: Giuseppe>
Matteo Bonacini: Ok
Matteo Bonacini: Ma te hai preso il mazzo da 24?
<eos>
```

Response from GPT2:

```
<Risposta a: Giuseppe>
Matteo Bonacini: ^^^
Matteo Bonacini: Mi sembra un buon modo per risolverla
Matteo Bonacini: Perché ho il mazzo anche io, se riesco io la pizzetta
Matteo Bonacini: Ma secondo me il consumo di carta lo prendi comunque
Matteo Bonacini: Perchè se io prendo carta da matita non so se riesco a paccare il
mazzo da solo
(cropped)
<eos>
```

Ground truth:

```
<Risposta a: Giuseppe>
Matteo Bonacini: Ok
Matteo Bonacini: Adesso cerco tutte le mie carte
<eos>
```

B.2 Prompt 2:

```
<bos><Chat con: Applied Physics>
+39 366 420 7296: If anyone had his grade registered for the exam of Software and
Computing for Applied Physics, can you please contact me and let me know? I've
already sent my project long ago and neither of the professors has answered my email
+39 366 420 7296: I'm not the only one with this issue by the way...
```

Response from Gemma:

```
<Risposta a: Applied Physics>
Matteo Bonacini: No
Matteo Bonacini: I don't know who you are but you're not my professor
<eos>
```

Response from GPT2:

```
<Risposta a: Applied Physics>
Matteo Bonacini: Ah ok
Matteo Bonacini: Lol
Matteo Bonacini: Ah
Matteo Bonacini: Ah, te pareva
Matteo Bonacini: La prossima volta in aula studio sì
Matteo Bonacini: Però sono molto più vecchia di quello allora
Matteo Bonacini: E c'è roba qua già molto più vecchia di quello dove avevamo messo
gli esercizi
```

```
<eos>
```

Ground truth:

```
<Risposta a: Applied Physics>
Matteo Bonacini: Yeah they explicitly told us that this was gonna be an issue
<eos>
```

Here we can see one very important difference between the two models: Gemma was trained to be multilingual, and it correctly responds in English to an English message. The version of GPT2 that I am using was trained only on Italian text, and it responds in Italian.

B.3 Prompt 3:

```
<bos><Chat con: Irnerio Gangsters>
[FF] Giuseppe: @[FF] Marta pizza Irnerio mo fa le pizze senza lattosio
[FF] Marta: Lo soolll
[FF] Marta: L'ho già mangiate
[FF] Simone Coli: Hanno già aperto la pizzeria?
```

Response from Gemma:

```
<Risposta a: Irnerio Gangsters>
Matteo Bonacini: Ma non è più la pizzeria che c'è in via Irnerio?
Matteo Bonacini: Non so se è la stessa proprietà
<eos>
```

Response from GPT2:

```
<Risposta a: Irnerio Gangsters>
Matteo Bonacini: Si
Matteo Bonacini: Ma la marta non ha nemmeno guardato cosa gli manca
Matteo Bonacini: E non sai manco cosa ha scritto lei
Matteo Bonacini: Perché ha detto "dopo cena se volete aggiungere un tot vi dico che
vi metto nel gruppo"
Matteo Bonacini: E comunque, voi due siete voi che avete messo i due criteri e ci
avete messo tre/sei>"
(cropped)
<eos>
```

Ground truth:

```
<Risposta a: Irnerio Gangsters>
Matteo Bonacini: non apriranno mai la pizzeria
<eos>
```

We can see that Gemma has some kind of understanding of what is being talked about. It rarely predicts the exact response that I would give, but this is understandable as it obviously cannot read my mind. GPT2, on the other hand, does not appear to follow the structure of a regular conversation.

B.4 Prompt 4:

```
<bos><Chat con: Frookes>
Frookes: Domanda
Frookes: Se ti proponessi di fare 4h di workshop divise in 2h/die su due giorni
Frookes: A novembre
Frookes: E forse riusciamo a pagarti 100€ (totale, lordo)
Frookes: (Puoi anche riproporre roba già fatta tipo il cuore se non è troppo
overkill per l'audience tipo cioè gente che di base parte che non sa programmare)
Frookes: Di certo invece posso offrirti un apericena/festa (anche a Sara se la vuoi
portare) e farti conoscere la Prof. Elisa Araldi che fa un sacco di roba figa di
bioinformatica
```

Response from Gemma:

```
<Risposta a: Frookes>
Matteo Bonacini: Ok ci sta
Matteo Bonacini: Però mi chiedo
Matteo Bonacini: Tu, per quanto sia un corso di laurea, non hai ancora studiato le
stringhe?
```

```
Matteo Bonacini: Perché se tu ti presentassi a me e se io mi assicuro di capire che
non le sto studiando, allora ci si può presenterne anche a me
<eos>
```

Response from GPT2:

```
<Risposta a: Frookes>
Matteo Bonacini: Nope
Matteo Bonacini: Che in realtà non ce lo faccio ahah
Matteo Bonacini: Ma se vuoi posso anche chiederglielo a me
Matteo Bonacini: Ceh tipo lunedì e martedì giusto?
Matteo Bonacini: Che magari dopo la laurea ho tempo libero
Matteo Bonacini: (secondo me se lo facciamo venerdì o sabato mi fanno bene)
(cropped)
<eos>
```

Ground truth:

```
<Risposta a: Frookes>
Matteo Bonacini: Allora mi piacerebbe un sacco, io però a novembre sarò a Parigi in erasmus
Matteo Bonacini: Quindi non penso di riuscire
<eos>
```

Here Gemma started hallucinating after the first two messages. However, the model still managed to capture the positivity with which I responded to that message. GPT2, on the other hand, captured the fact that I barely have any free time because of university (lol). Sadly, none of the models could have known that I would not be available this November.