TECHNION – ISRAEL INSTITUTE OF TECHNOLOGY

NSSL – Networked Software Systems Lab

Personal Chatbot

GitHub

Final Report

Submitted by:

Eden Ninary Jonathan Bleicher

Supervised by:

Hovav Gazit

Table of content

1.Introduction

1.1 Project Goal

The main goal of the project is to create a personal model that can communicate with other people through Whatsapp messages in the same tone and manner that we would. For this we used a pre-trained Mistral-7B model architecture which we will expand on in further detail later on in this report.
The chosen method for fine tuning was using QLORA which is a 4-bit quantisation along with LORA to allow us to train far fewer parameters than in the original model. This allowed us to get training done in a relatively short period of time, and with very limited GPU memory, while still achieving impressive results.
We created python scripts that processed out data and allowed for clear and easy communication with the model.

1.2 Motivation

Our motivation for this project stems from our curiosity about large language models (LLMs) and how to make them more personal and engaging. We wanted to explore how to create a chatbot that feels like a genuine conversation with an individual. By working with the Mistral-7B architecture and employing techniques like QLORA for 4-bit quantization, we sought to understand how to fine-tune these models effectively while managing limited resources. Our goal was to learn how to customize a model for specific, personalized responses and assess the feasibility of using practical advancements like quantization.

2. Method

2.1 Hugging Face

Hugging Face is an open-source platform dedicated to Natural Language Processing and machine learning, offering a suite of easy-to-use tools for tasks such as text generation, classification, translation, and various other ML applications. By democratizing AI, it enables easy access to state-of-the-art models like BERT, GPT, and Mistral, making advanced technology available to a wider audience. The platform features a large hub of curated datasets for training and evaluating ML models.
Libraries we used in this project:

- Transformers - Provided us with the pre-trained model, the tokenizer, and the trainer function.

```
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

tokenizer = AutoTokenizer.from_pretrained("yam-peleg/Hebrew-Mistral-7B",padding_side="left",
        add_eos_token=True,
        add_bos_token=True,)

model = AutoModelForCausalLM.from_pretrained("yam-peleg/Hebrew-Mistral-7B", quantization_config =
                                              BitsAndBytesConfig(load_in_4bit=True,
                                             bnb_4bit_use_double_quant=True,
                                             bnb_4bit_quant_type='nf4',
                                             bnb_4bit_compute_dtype=torch.bfloat16))


training_args = transformers.TrainingArguments(
                output_dir='./trials_output',
                evaluation_strategy='epoch',
                warmup_steps=2,
                learning_rate=2.5e-5,
                per_device_train_batch_size=2,
                per_device_eval_batch_size=2,
                weight_decay=0.001,
                save_total_limit=3,
                num_train_epochs=3,
                gradient_accumulation_steps=4,
                gradient_checkpointing=True,
                optim="paged_adamw_8bit",
                logging_dir="./logs",
                logging_steps=50,
                bf16=True
            )

data_collector = transformers.DataCollatorForLanguageModeling(tokenizer, mlm=False)
trainer =transformers.Trainer(
    model=model_change,
    args=training_args,
    train_dataset=tokenise_train_dataset,
    eval_dataset=tokenise_eval_dataset,
    data_collator=data_collector
)
```

The code in the image above shows how we loaded the model and the tokenizer with the desired parameters and features. It also shows how we set up all the arguments for the trainer so that it uses the training parameters during its training.

- Parameter Efficient Fine-Tuning (PEFT) - designed to minimize the number of parameters for optimization. We used this library to access LoRA which we will expand on later.

```
# LORA
LORA_R = 32
LORA_ALPHA = 64
LORA_DROPOUT = 0.07
LORA_TARGET_MODULES = "q_proj,k_proj,v_proj,o_proj,gate_proj,up_proj,down_proj,lm_head"

peft_config = LoraConfig(
    lora_alpha=LORA_ALPHA,
    lora_dropout=LORA_DROPOUT,
    r=LORA_R,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=LORA_TARGET_MODULES.split(","),
)

model_change = get_peft_model(model, peft_config)
```

The code in the image above shows exactly how LoRA is configured and used in our project.

- Datasets - streamlines the processes of loading, preprocessing, and managing datasets. This is especially useful for splitting the data into training, validation, and test sections.

```
from datasets import load_dataset

data_set = load_dataset('json', data_files=
'/home/joned/Desktop/Chat_bot_proj/project_v1/src/data_preprocessing/final_outputs/formatted_data.jsonl',
split='train')

eval_dataset = data_set.train_test_split(test_size=0.2)
```

The code above shows how we used the datasets library to load our data from the json file and then split it into 80 percent train and 20% train/validation.

- Accelerate - simplifies the process of training by distributing the training and the models across different types of available hardware.

```
from accelerate import Accelerator, FullyShardedDataParallelPlugin

fsdp_plugin = FullyShardedDataParallelPlugin(
    state_dict_config=FullStateDictConfig(offload_to_cpu=True, rank0_only=False),
    optim_state_dict_config=FullOptimStateDictConfig(offload_to_cpu=True, rank0_only=False),
)

accelerator = Accelerator(fsdp_plugin=fsdp_plugin)
model_change = accelerator.prepare_model(model_change)
```

The code above shows how we used accelerate to delegate the work between the GPU and the CPU for maximum efficiency.

2.2 Preprocess

The data was extracted from our personal Whatsapp conversation and it contained a lot of relevant artifacts such as messages and emojis, as well as irrelevant artifacts such as pictures, links, voice memos, and even code snippets.
We first ran the data through a python script that cleaned out most of the irrelevant artifacts and then through another script that structured the data in the form of a template that communicates to the model effectively. The structure is as follows:

<s> - start of speaking
</s> - end of speaking
<person1> - one speaker
<person2> - another speaker

```
{"input": "<s><person1>היום להפגש לך בא מתי?\nאתה לי ויש
הפתרון</s><s><person2>15?\nלך שנוחה שעה לך יש אם מתי לי משנה לא בעיקרון
יותר</s><s><person1>משנה לא לי גם\nמגיעה אתה מאיפה ?\nמגיע
*?</s><s><person2>ליד פה משכיר שאני מהדירה\nרחוק לא
חח</s><s><person1>נווש או נשר?</s><s><person2>נשר\nהליכה דק 10 8 ממש
המכינה של מהבניין</s><s><person1>סבבי\n15 אז</s><s><person2>סבבה"}
```

It is a bit difficult to read because of the Hebrew changing directions, but essentially the structure tells the model that in a given line of data, there are two speakers and it tells the model when one speaker starts, finishes, and the other begins and finishes as well.

We organized our data into blocks of inputs, with each block representing a conversation. The conversations end after an hour of silence between the two people. This structure was chosen to provide the model with contextual understanding of the conversations and enable it to learn both our conversational intuition and broader context of word usage. The separation into two people also allows the model to learn the mannerisms and nuance that each person has.

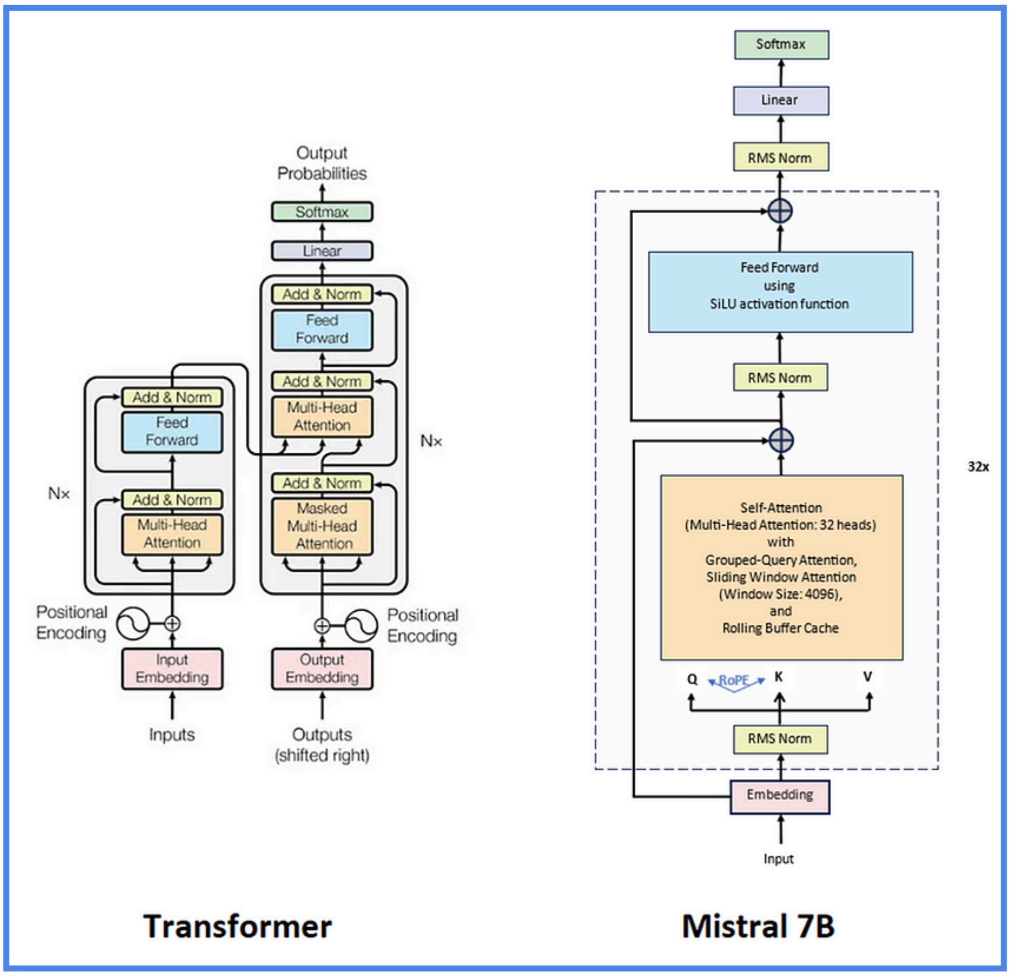
2.3 Tokenization

Tokenization is the process of converting raw text into smaller, more manageable units called tokens. These tokens typically represent words, subwords, or even characters, depending on the type of tokenization used. It is necessary for transforming text into a form that deep learning models which typically work with numeral input, can process.

The process of tokenzing occurs in the pre-training when the original model is built. The list of tokens also known as the vocabulary is built by separating the data into words based on punctuation and spaces (this is known as word-level tokenization), or separating the data into subwords and characters which help it break down unusual and rare words. The tokenizer then counts the frequencies of the words in the vocabulary and that way chooses which ones to assign a number to and keep and which ones to discard. The end result is a list called a vocabulary that contains all the most frequently used words/subwords/characters in the data. Each token in the vocabulary is assigned a number and from that point on the model works only with the numbers.

2.4 Model

We chose the Mistral-7B model for our project, which is pre-trained specifically for the Hebrew language. This model is built on the Transformer architecture and utilizes an Attention Mechanism. It consists of 32 Transformer layers, each serving as both an Encoder and Decoder, making it well-suited for handling large language models (LLMs). The model is optimized for tasks involving Hebrew text, with a deep architecture that captures complex language patterns. This is a solid foundation for our goal of training the model on our own conversation data, which includes nuanced contexts within both the language and the conversations themselves.



This image above compares the architecture of the standard Transformer model (on the left) with the architecture of the Mistral-7B model (on the right). The Transformer model on the left is a standard implementation of the attention-based neural network, which processes input sequentially through layers of attention and feed-forward networks. The Mistral-7B model on the right is an optimized and more scalable version of the Transformer, with more attention heads, efficient memory management, advanced normalization techniques, and enhanced handling of long sequences. These optimizations allow Mistral to process large amounts of data efficiently and scale well with increased model size.

Transformer:

- **Input embedding** - The input tokens are first transformed into embeddings. These embeddings represent the input words in a high-dimensional space where semantically similar words have similar vectors.
- **Positional Encoding** - Positional encoding adds information about the position of each word in the sequence to the embedding.
- **Multi-Head Attention** - Attention mechanism allows the model to focus on different parts of the input sequence when producing each word. The multi-head variant allows the model to learn different relationships between words through multiple "attention heads."
- **Add & Norm** - This step involves skip connections (adding the original input to the output of the attention layer) followed by a normalization step to stabilize training. Skip connections help to mitigate the problem of vanishing gradients by providing an alternative path for the gradient to flow during backpropagation. Summarized in one word, bypass.
- **Feed Forward** - This layer applies a fully connected neural network to each position separately and identically. It adds non-linearity and helps the model learn complex patterns.
- **Nx Blocks** - The structure shown (Multi-Head Attention + Feed Forward) is repeated N times, where "N" is the number of layers in the Transformer model. For large models, this number can be quite high.
- **Masked Multihead Attention** - In decoder models (for tasks like text generation), masked attention ensures that the model cannot "see" future tokens, only the ones it has already processed. This masking is essential for autoregressive models that are trained to predict the next token in a sequence.
- **Output Embedding** - The outputs are shifted by one position so that the model predicts the next word in the sequence rather than the current word.
- **Linear + Softmax** - A linear transformation followed by a softmax function is applied to produce probabilities over the vocabulary for the next token in the sequence.

Mistral 7B:

- **RMS Norm** - Instead of the traditional LayerNorm used in many Transformer models, Mistral uses RMS Norm (Root Mean Square Layer Normalization). RMSNorm normalizes the activations without subtracting the mean, making it computationally more efficient.

- **Feed Forward with SiLU** - The feed-forward layer here uses the SiLU (Sigmoid Linear Unit) activation function, which is a smoother and more flexible activation function compared to the commonly used ReLU. This activation helps with better gradient flow during training.
- **Self Attention (with specialized techniques)**:
    - **Multi-Head Attention with 32 heads** - Mistral uses a large number of attention heads (32 in this case), which allows the model to capture various intricate relationships between different parts of the input sequence.
    - **Grouped-Query Attention** - This technique helps to reduce the computational cost of the attention mechanism by grouping query representations together. It helps the model scale more efficiently. In attention, a "query" can be thought of as a piece of information that asks how much attention one part of the input (like a word in a sentence) should give to other parts of the input.
    - **Sliding Window Attention (Window Size: 4096)** - This technique restricts attention to a local window of tokens, which is useful for handling longer sequences. The model focuses on a "sliding window" of tokens rather than attending to the entire input sequence at once.
    - **Rolling Buffer Cache** - This is an optimization for efficiently storing past tokens, allowing the model to look back at previously processed tokens without recomputing their embeddings, which is particularly helpful for long contexts.

Key Differences:

- **Normalization** - Mistral uses RMS Norm instead of LayerNorm for better efficiency.
- **Attention Mechanism** - Mistral has more attention heads (32 vs. typical lower counts) and optimizes the attention process with techniques like Grouped-Query Attention, Sliding Window Attention, and Rolling Buffer Cache.
- **Positional Encoding** - Mistral uses **RoPE** (Rotary Position Embedding) instead of traditional positional encodings.
- **Activation Function** - Mistral uses **SiLU** instead of ReLU in the feed-forward layers, allowing for smoother activation.

2.5 Fine-Tuning

Fine-tuning adapts a pre-trained model, like the Mistral-7B model we chose, to perform well on a specific task or dataset. We began with the pre-trained model, which had already learned general language patterns in both English and Hebrew, and fine-tuned it on our task-specific dataset to enhance its performance in that domain. During this fine-tuning process, we updated the model's parameters with the new data, leveraging its existing knowledge of the Hebrew language while focusing on the new task. To carry out this process, we utilized the Hugging Face library, which provided a structured framework for the training stage and allowed us to fine-tune various training parameters, such as LoRA, learning rate, and dropout.

As seen in the previous sections, these models are extremely complex and have been carefully optimized. These models were created by researchers and industry leaders with extensive compute resources and research capabilities. Fine-tuning allows individuals to harness much of that power and create something more personally useful in a small use case. If we would have designed and trained a model from scratch for this project we likely would not have been able to achieve any results.

2.6 LoRA

Low-Rank Adaptation (LoRA) is a technique for fine-tuning large language models in a parameter-efficient manner. It works by adding low-rank decomposition matrices to the model's weights during training, enabling significant reductions in the number of trainable parameters while retaining model performance. This approach allows models to be adapted to specific tasks with fewer resources and less computational overhead, making it practical for deploying large models in resource-constrained environments. The original paper introducing LoRA is titled **"LoRA: Low-Rank Adaptation of Large Language Models"** by Edward Hu, Yelong Shen, Aditi Raghunathan, and others, published in 2021.

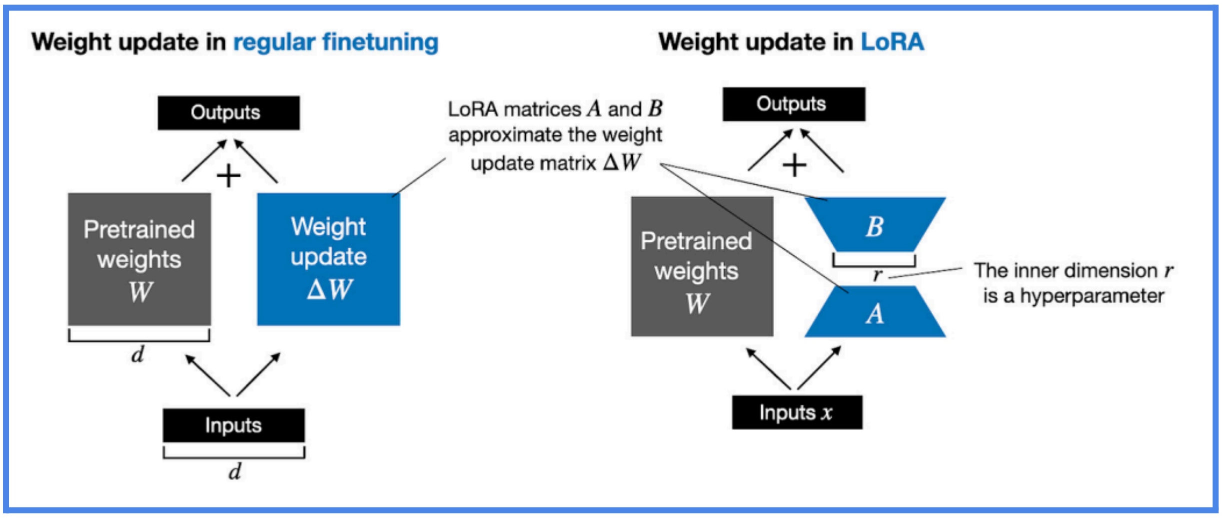
Given a weight matrix $W \in \mathbb{R}^{d_i \times d_o}$, in regular training or fine-tuning using gradient descent, the weights are updated according to:

$$W_{\text{updated}} = W + \Delta W.$$

LoRA proposes an alternative method for this update, by learning a low-rank approximation of ΔW:

$$\Delta W \approx AB,$$

where $A \in \mathbb{R}^{d_i \times r}$, $B \in \mathbb{R}^{r \times d_o}$ and $r << d_i, d_o$ is the *rank* (very similar to SVD--Singular Value Decomposition).
In practice, we usually keep W frozen and only learn A and B.



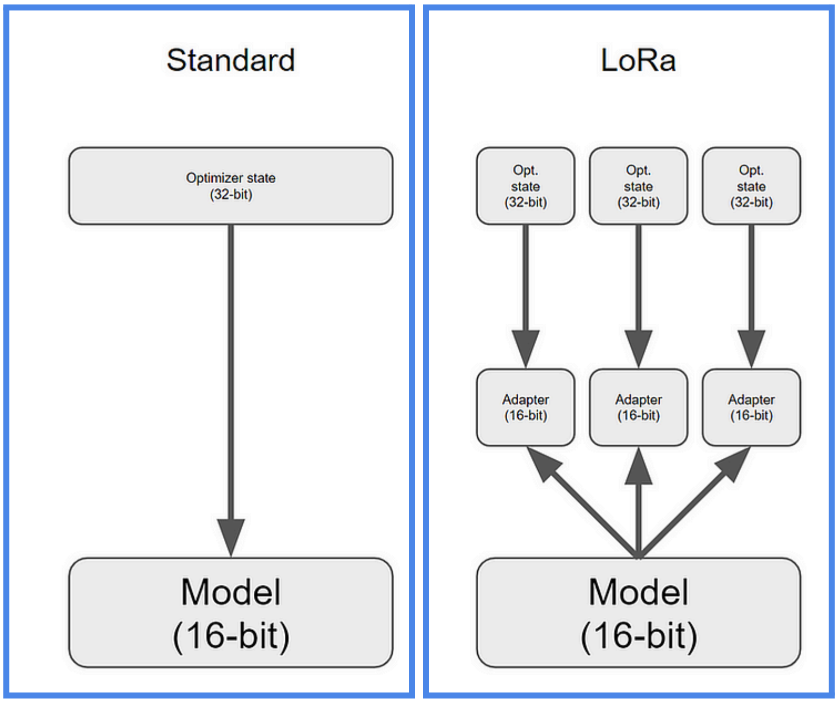
The images above were taken from tutorial 9 of the Deep Learning course at the Technion (046211). The first image describes LoRA mathematically. The second image demonstrates how the LoRA wight update depends on the "r" parameter and therefore significantly lowers the amount of parameters to update.

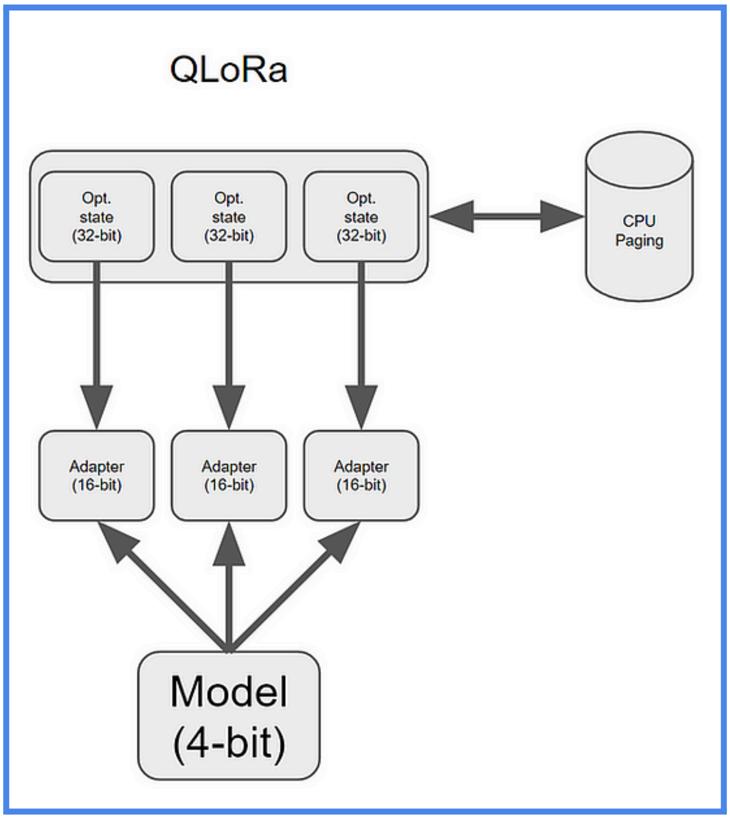
In our model the final number of parameters we needed to train was 86,065,184 which is only 1.1339% of our original pre-trained model, making the fine-tuning process more efficient and manageable on limited hardware. Together, these techniques helped us efficiently train the large model despite hardware constraints.

2.7 QLoRA

Quantization reduces the precision of a model's weights to save memory and computational resources. In QLoRA, quantization typically involves converting weights to lower-bit representations, such as 4-bit integers. This process significantly reduces the memory footprint and allows large language models to be fine-tuned or even run on smaller hardware. Despite this reduction in precision, carefully designed quantization methods can maintain the model's performance while enabling more efficient training and inference.

In this project, the size of our model is far too large for our available GPU resources. We had to make use of quantization to "shrink" the model as well as LoRA to limit the number of parameters being trained. We chose a 4-bit quantization which reduces the precision of model weights and activations from 32-bit floating-point to 4-bit integers, decreasing memory usage and speeding up computations while potentially sacrificing some accuracy.





In the figure above we can see the comparison between a standard model, a LoRA model, and a QLoRA Model.

Standard - *This is the baseline approach, where the entire LLM is trained from scratch.*

LoRA - *This technique introduces adapters to the LLM, which are trained separately from the main model. This allows for more efficient training and fine-tuning.*

QLoRA - *This is an extension of LoRA that further reduces the memory footprint of the model by quantizing the weights to 4-bit precision. This makes it suitable for deployment on devices with limited resources.*

3. Experiment And Results

3.1 Final model parameters

Training parameters:
The following parameters were found after several iterations of trial and error. The LoRA R and LoRA Alpha were chosen to be as large as possible given the memory limitation. We chose a low number of epochs because we noticed that we had enough data for the model to finish its training after around 2 epochs. More epochs caused the eval loss to grow.

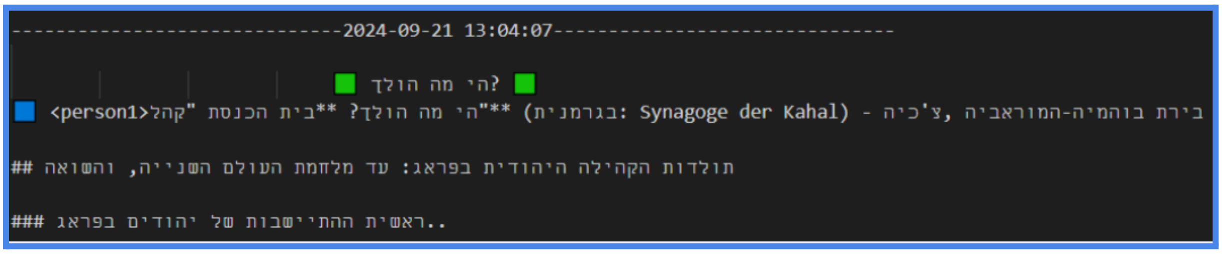
Parameter	Value	Description
LoRA R	32	The rank of the matrices.
LoRA Alpha	64	The scaling factor for the LoRA updates.
LoRA Dropout	0.07	The proportion of randomly deactivated LoRA parameters during each training step.
LoRA Target Modules	q_proj,k_proj,v_proj,o_proj,gate_proj,up_proj,down_proj,lm_head	These are the specific parts of the model that will be fine-tuned with LoRA.
Warm up steps	2	The number of initial steps during which the learning rate gradually increases from a small value to the target learning rate.
Learning rate	2.5e-5	The step size at each iteration of the optimizer during training
Weight decay	0.001	A penalty that is applied to the magnitude of the model's weights, encouraging them to be smaller.
Epochs	3	The number of full passes over the training dataset.
Optimizer	AdamW	The optimization algorithm used for updating the model's parameters.

Output generation parameters:
These parameters are extremely sensitive and are critical for the model's ability to communicate correctly. We found these values by generating outputs in a loop that scans through possible values and saves the best ones. The repetition penalty is especially crucial. When it is too low the model just repeats the input over and over again. When it is too high the model regurgitates uncontrollable nonsense that has nothing to do with the input.

Parameter	Value	Description
Temperature	0.5	Controls the randomness or creativity of the model's output
Repetition Penalty	1.025	Discourages the model from generating the same token repeatedly during text generation

3.2 Conversations with the model BEFORE training

Before the training on our data, the model was trained to know the Hebrew language. It knows almost nothing beyond that.

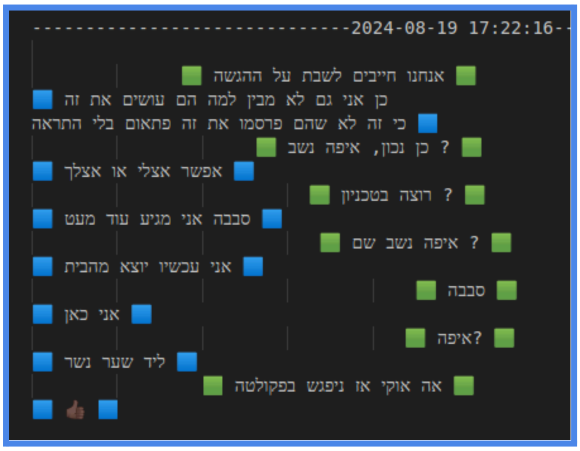


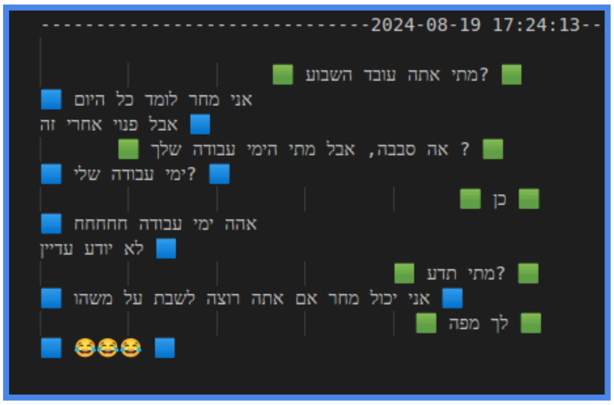
In the two conversations above it is difficult to infer exactly why the pre-trained model responded the way that it did, but it is clear that it responded in hebrew since it was provoked in hebrew, it probably regurgitated something similar to the information on the internet that it was trained on, and it did not at all take the sentiment of the question into account.

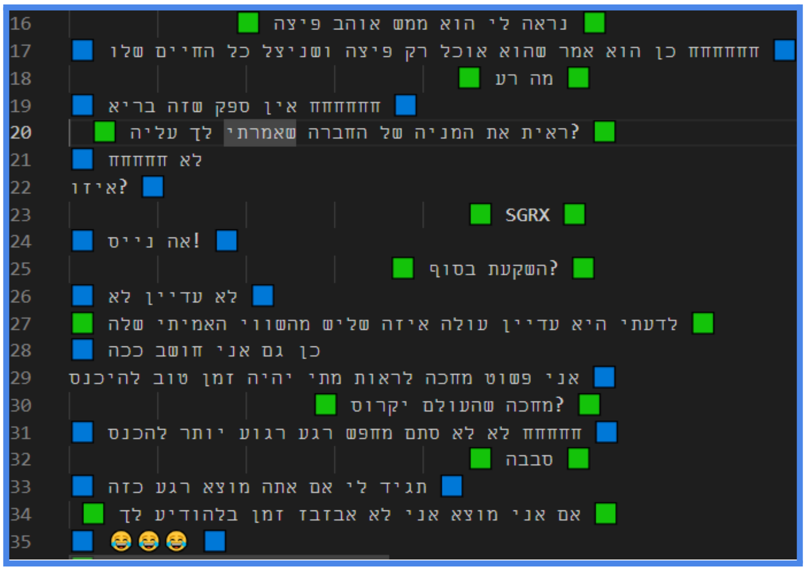
3.3 Conversations with the model AFTER training

In the following images of the conversations you can see that the model responds to the context and it knows information that we would know. It also picks up on the nuances such as use of emojis and in the correct place.

Conversations with Eden:

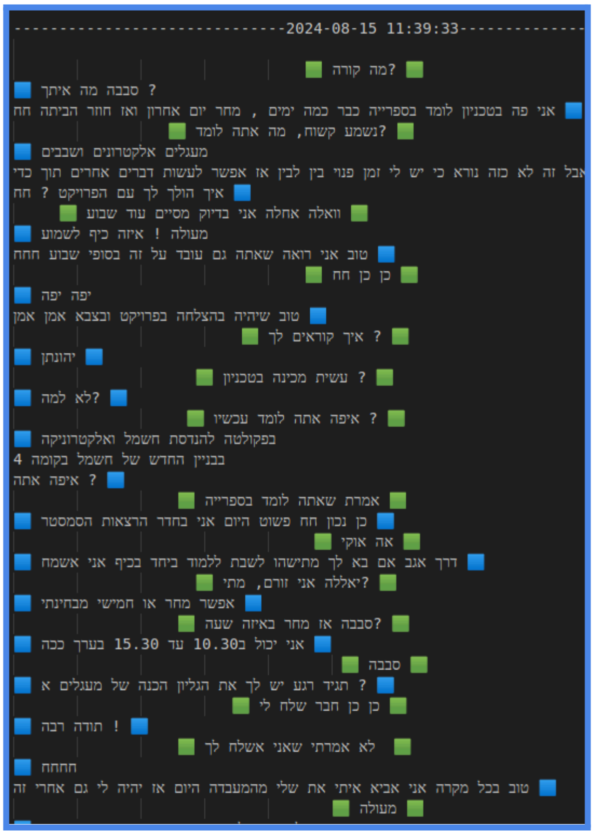


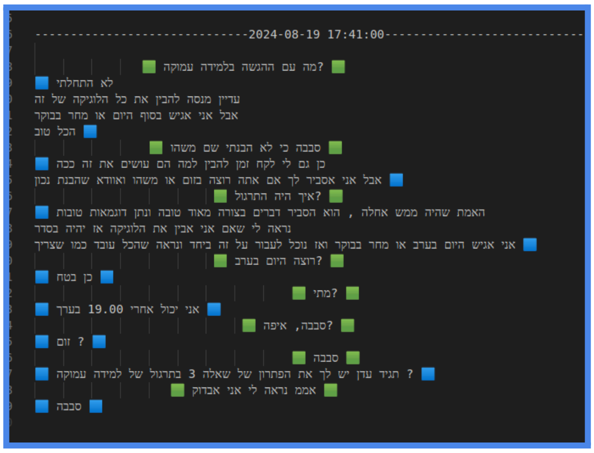


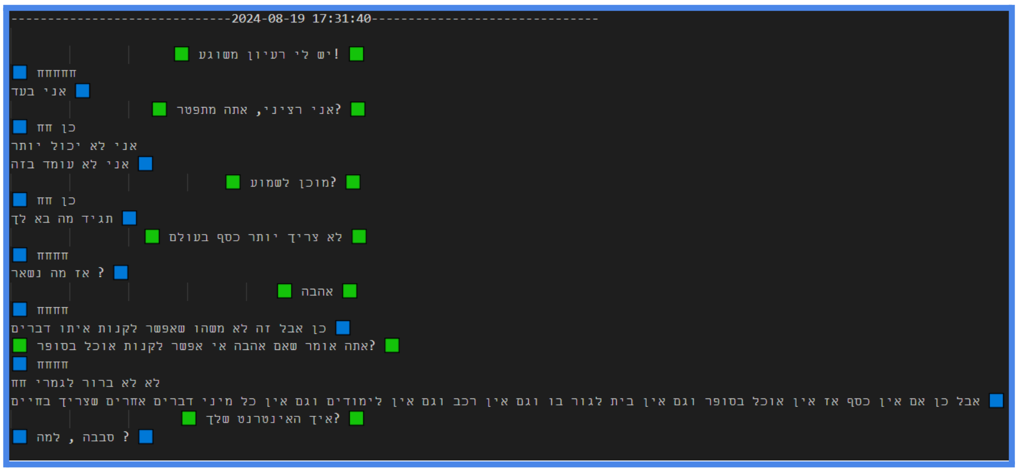


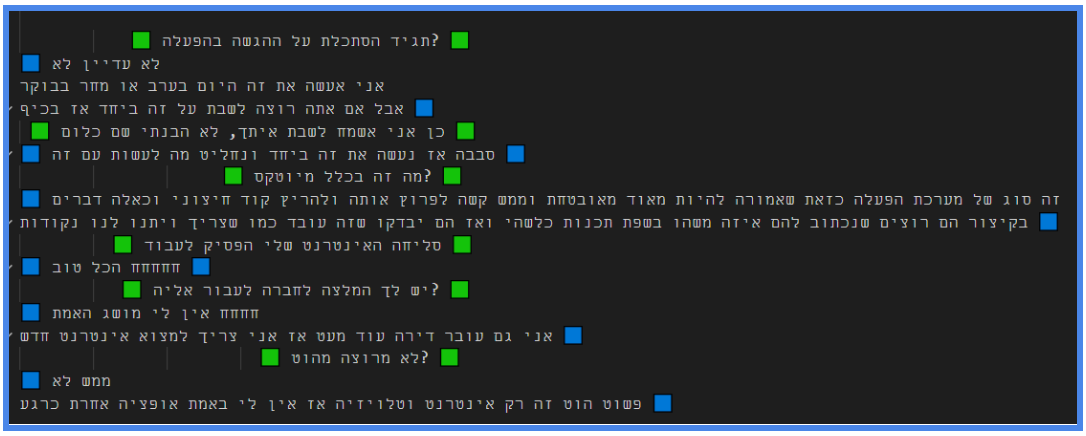
In the figures above we can see a coherent conversation where the model learned how to answer, build full sentences and when to stop generating and pass the turn onwards. The model also has information about location that it learned from our conversation and can connect between the context of arranging a meeting and a specific location. We can also see how the history we defined into the model comes in hand, the model can "remember" older parts from the conversation that we had. Another thing to note is that the use of emojis is intuitive, and that Eden tends to use more emojis which the model was able to pick up on.

Conversation with Jonathan:









In the figures above we can see that the model knows about courses in the Technion and the connection between learning them and lectures and the faculty. It is notable that the model will ask questions relevant to the situation and keep in mind things that were previously spoken about. Another notable feat is that the model is not passive and will provoke questions.

4. Conclusion And Future Work

4.1 Conclusion

This project leaned very heavily on other people's work that was done over many years. The models and the optimizations are objectively where most of the work went into. The innovation that we brought to this project was how to pick and choose different models, methods, libraries, and parameters, to channel all of the previous hard work into a very personalized use case that nobody else would have been able to work on. We created a codebase that allows anyone to import their Whatsapp data into and it will create their own chatbot that will behave like them (or of the person of their choosing from their contacts). Other LLMs today are very knowledgeable and can provide you with all the information on the internet, but they cannot sound like you.

Evaluating the success of a project like this is very tricky. It is very difficult to assess if an LLM is performing "well" or not. There is no numerical value that we can assign to its responses that can tell us if the model answered correctly or not. For this reason we were only able to assess it intuitively and by speaking to it and judging ourselves if we liked the responses or not. We spoke to the model extensively about our studies, about locations at the Technion, and about different courses, lectures and classes. We noticed that for the most part it was very on topic. The answers that it gave were very relevant to the questions. The context was almost always taken into account. In early iterations of the project the model was very passive and did not really generate much conversation, however, by the end the model was asking us questions, telling jokes, and really made great conversation. The model picked up on little nuances such as emoji use, laughing, and responding correctly to sarcasm. We were extremely satisfied with the end result.

4.2 Future Work

Multiple chats:
For future work, we recommend using multiple conversations from Whatsapp as opposed to just one conversation like we did. In doing so, you can train your chatbot to have a more holistic personality as well as to be more knowledgeable about you in general (since it gets exposed to more data and data of different kinds).
Exploring different models:
We used Mistral 7B for this project however there are more models out there that might perform better so that is a possibility worth exploring. There is the Mistral 11B that we chose not to use because of hardware constraints, but might be able to do a better job if trained correctly.