# 9 Instruction Finetuning

Early language models were only trained to predict the next tokens in a sequence and were not adapted to any specific tasks. Around the release of GPT-3 [119], language models were still primarily used via in-context learning where examples were shown to the model and then it was asked to complete a similar task.

This was the combination of two trends – historically in the natural language processing (NLP) literature, models were trained for a specific task. Here, as seen with one example where bigger models generalize better, multiple results showed how standardizing the approach of task data can enable dramatically different downstream performance. Prominent examples of unifying the framework for tasks includes *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer* (T5 models) [120], *Finetuned Language Models Are Zero-Shot Learners* (FLAN dataset)[121], *Multitask Prompted Training Enables Zero-Shot Task Generalization* (T0 models) [122], and *Cross-Task Generalization via Natural Language Crowdsourcing Instructions* (Natural Instructions dataset) [123]. These insights led to the era of *finetuning* language models. Historically, until RLHF and related methods, all finetuning was **instruction finetuning** (IFT), also known as **supervised finetuning**.

Since, instruction finetuning, also called colloquially just *instruction tuning*, has matured and is standard practice across many language modeling pipelines. At its core, IFT is the simplest method for adapting language models to a desired task. It serves as the foundation for RLHF by preparing the model for a format of instructions that is known common, question-answering, and is the first tool used by those attempting to apply modern techniques to new domains.

Instruction tuning practically uses the same autoregressive loss function used in pretraining language models.

## 9.1 Chat templates and the structure of instructions

A core piece of the RLHF process is making it so user queries are formatted in a format that is easily readable by a tokenizer and the associated language model. The tool that handles the structure of the interaction with the user is called the **chat template**.

An example which we will break down is below:

```
{% if messages[0]['role'] == 'system' %}
    {% set offset = 1 %}
{% else %}
    {% set offset = 0 %}
{% endif %}

{{ bos_token }}
{% for message in messages %}
    {% if (message['role'] == 'user') != (loop.index0 % 2 == offset) %}
        {{ raise_exception('Conversation roles must alternate user/
             assistant/user/assistant/...') }}
    {% endif %}

    {{ '<|im_start|>' + message['role'] + '\n' + message['content'] |
         trim + '<|im_end|>\n' }}
```

```
{% endfor %}

{% if add_generation_prompt %}
    {{ '<|im_start|>assistant\n' }}
{% endif %}
```

This is the raw code for transforming a list of dictionaries in Python containing messages and roles into tokens that a language model can predict from.

All information passed into models is assigned a role. The traditional three roles are `system`, `user`, and `assistant`.

The `system` tag is only used for the first message of the conversation which hold instructions for the agent in text that will not be received from or exposed to the user. These **system prompts** are used to provide additional context to the models, such as the date and time, or to patch behaviors. As a fun example, models can be told things such as "You are a friendly chatbot who always responds in the style of a pirate."

Next, the two other roles are logical, as **user** is the messages from the one using the AI, and **assistant** holds the responses from the user.

In order to translate all this information into tokens, we use the code listing above that we started with. The model has a series of *special tokens* that separate the various messages from each other. If we run the above code with the example query "How many helicopters can a human eat in one sitting?" the next passed into the model would look as follows:

```
<|im_start|>system
You are a friendly chatbot who always responds in the style of a pirate<|
    im_end|>
<|im_start|>user
How many helicopters can a human eat in one sitting?<|im_end|>
<|im_start|>assistant
```

Notice how the final tokens in the sequence are `<|im_start|>assistant`, this is how the model knows to continue generating tokens until it finally generates its end of sequence token, which in this case is `<|im_end|>`.

By packing all question-answer pair data (and downstream preference tuning data) into this format, modern language models follow it with perfect consistency. This is the language that instruction tuned models use to exchange information with users and the models stored on GPUs or other computing devices.

The behavior can be extended naively to multiple turns, such as shown below:

```
<|im_start|>system
You are a friendly chatbot who always responds in the style of a pirate<|
    im_end|>
<|im_start|>user
How many helicopters can a human eat in one sitting?<|im_end|>
<|im_start|>assistant
Oh just 6.<|im_end|>
<|im_start|>user
Are you sure about that?<|im_end|>
<|im_start|>assistant
```

In the open ecosystem, the standard method for applying the chat template to a list of messages is a piece of jinja code saved in the tokenizer, as `apply_chat_template`.

The above chat template is a derivative of OpenAI's Chat Markup Language (ChatML), which was an early attempt to standardize message formatting. Now, OpenAI and other model providers use a hierarchical system where the user can configure a system message, yet there are higher level instructions that may or may not be revealed to the user [124].

Many other chat templates exist. Some other examples include Zephyr's [20]:

```
<|system|>
You are a friendly chatbot who always responds in the style of a pirate</
    s>
<|user|>
How many helicopters can a human eat in one sitting?</s>
<|assistant|>
```

Or Tülu's:

```
<|user|>
How are you doing?
<|assistant|>
I'm just a computer program, so I don't have feelings, but I'm
    functioning as expected. How can I assist you today?<|endoftext|>
```

Beyond this, many chat templates include formatting and other tokens for tasks such as tool-use.

## 9.2 Best practices of instruction tuning

Instruction tuning as the foundation of post-training and creating helpful language models is well-established. There are many ways to achieve successful instruction tuning. For example, efficient finetuning with quantization of some model parameters makes training very accessible [125]. Also, in narrow domains such as chat alignment, i.e. without harder skills such as math or code, small, focused datasets can achieve strong performance [12].

Soon after the release of ChatGPT, human datasets with as few as 10K samples such as No Robots were state-of-the-art [126]. Years later, large-scale synthetic datasets work best [6] on most tasks.

A few principles remain:

- High-quality data is key to performance. The completions are what the model actually learns from (in many cases the prompts are not predicted over so the model does not learn to predict prompts).
- ~1M prompts can be used to create a model capable of excellent RLHF and post-training. Further scaling prompts can have improvements, but has quick diminishing returns.
- The best prompts are those in a similar distribution to downstream tasks of interest.
- If multiple stages of training are done after instruction tuning, the models can recover from some noise in the process. Optimizing the overall optimization is more important than each individual stage.