4.1 Supervised fine-tuning and RL fine-tuning

Author: Polina Tsvilodub

This sheet provides an overview of different flavours of fine-tuning of LLMs and their respective use cases. Particular focus is provided for *RL based* fine-tuning, and specifically, *RLHF* (reinforcement learning from human feedback).

The key learning goals for this sheet are:

- be able to identify the type of fine-tuning used for a particular model and explain it conceptually
- gain a practical understanding of RLHF components, in particular:
  - creation and use of a reward model
  - training steps for the policy.

Flavours of fine-tuning

In sheet 2.5, we already brushed over the distinction between pretrained and fine-tuned models. In particular, fine-tuning was introduced as the training LMs on *task-specific* datasets, e.g., for movie review generation or classification. For classification, LMs are usually fine-tuned with a classification head (see sheet 3.2 for a recap). However, from here on, we will focus on fine-tuning of model for *generative* tasks, i.e., simply fine-tuning LMs for next-word prediction on a specific task.

We have seen fine-tuning of a generative LM for question answering in homework 1. Here, the specific task the model is supposed to learn is constituted by the specific dataset and the formatting of the questions and the answers. The model was trained on examples of questions with correct answers; i.e., this is *supervised fine-tuning* where the model was shown what the desired behavior is (i.e., which next tokens it is supposed to predict).

For state-of-the-art LLMs, it has been identified that there is a particular kind of task that *general-purpose LLMs*, and in particular *assistants*, should be able to complete: namely, **instruction-following**. That is, rather than creating LMs that can only do QA on a specific dataset with a particular formatting, the community started to build *instruction-tuned* LLMs, which generate sensible outputs given user instructions ranging from "Provide ten recipes with tofu in a bullet list format" to "Summarize the following scientific paper". This has also been achieved with supervised fine-tuning, where the example input and output pairs in the dataset consist of example instructions, and their respective completions.

Exercise 4.1.1: Supervised-finetuning

1. Take a look at this dataset. For which kind of fine-tuning is it intended? What kinds of examples are there?
2. Consider the following use cases for which you want to build an LM: (a) an assistant tutor model for students for different subjects, (b) a model for answering highly specific questions about a medical knowledge base, (c) a model intended for writing abstracts of

scientific papers. What kind of fine-tuning set up would you consider (i.e., what kind of fine-tuning dataset would you ideally choose)?

3. Please come up with a prompt for testing whether an LM can follow instructions. Use the following code to test the instruction-following performance of an instruction-tuned model (Phi-3) and a simple small LM (GPT-2). Feel free to play around with the decoding parameters! (**WARNING**: the instruction-tuned model is a large model, so it can take a moment to load on Colab. Please also be aware of it if you execute the notebook locally!)

```python
# import packages

from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

tokenizer_instruct = AutoTokenizer.from_pretrained("microsoft/Phi-3-
mini-4k-instruct")
model_instruct = AutoModelForCausalLM.from_pretrained("microsoft/Phi-
3-mini-4k-instruct")

tokenizer_lm = AutoTokenizer.from_pretrained("gpt2-small")
model_lm = AutoModelForCausalLM.from_pretrained("gpt2-small")

instruction_text = #### YOUR TEXT HERE ####
input_ids_instruct = tokenizer_instruct.encode(instruction_text,
return_tensors="pt").to(device)
input_ids_lm = tokenizer_lm.encode(instruction_text,
return_tensors="pt").to(device)

prediction_instruct = model_instruct.generate(
    input_ids_instruct
)
print("Instruction-tuned model's prediction: ",
tokenizer_instruct.decode(prediction_instruct[0],
skip_special_tokens=True))

prediction_lm = model_lm.generate(
    input_ids_lm
)
print("GPT-2's prediction: ", tokenizer_lm.decode(prediction_lm[0],
skip_special_tokens=True))
```

The distinctions above focused on distinctions in the *content* of the fine-tuning, i.e., the content of the input-output demonstrations in the datasets used for the supervised fine-tuning.

Additionally, the lecture introduced different *methods* of efficient supervised fine-tuning, which is especially important for large LMs that take a lot of resources to train. The QA fine-tuning that we did in homework 1 was naive fine-tuning. That is, during the fine-tuning, all parameters were updated. However, as explained in the lecture, the more common state-of-the-art approach to fine-tuning is *parameter-efficient*, i.e., only a *selected* subset of the pretrained model parameters, or a *small set of new parameters* is updated.

The following code provides an simple example of vanilla selective fine-tuning where only the last transformer block and the last layer (i.e., LM head) of GPT-2 would be finetuned, and all other layers are frozen (❄). Concretely, this means that we don't want to compute gradients of parameters that are frozen, and we do not want to change their values. Usually parameters are (un)frozen by-layer / component.

Of course, the same approach can be used for (un)freezing any other subset of layers, in any other `transformers` model. For this, it is useful to know how to inspect and access different components of a pretrained model, as was briefly shown in sheet 3.1.

> Optionally, you can reuse the code from the homework to fine-tune this partially frozen model on the QA task from the homework. Do your results change?

```
gpt2_model = AutoModelForCausalLM.from_pretrained("gpt2")

# first we inspect the model's configuration
print(gpt2_model)

# first, we can inspect the model's configuration and named parameters
for name, _ in gpt2_model.named_parameters():
    print(name)

# next, we define which layers NOT to freeze
# (of course, we can do this vice versa and define which layers to
freeze)

layers_to_unfreeze = ["transformer.h.11", "transformer.ln_f.weight",
"transformer.ln_f.bias"]

# iterate over model's parameters
# note that, by default, in train mode, all parameters are set to
require grad = True (i.e., unfrozen)
for name, param in gpt2_model.named_parameters():
     # check that these parameters are not in the layers_to_unfreeze
list
     if all([not name.startswith(n) for n in layers_to_unfreeze]):
        # if not, freeze these parameters
        param.requires_grad = False

# now we check how many parameters are trainable
params = [p for p in gpt2_model.parameters() if p.requires_grad]
print(f'The model has {sum(p.numel() for p in params):,} trainable
parameters')
```

> Exercise 4.1.2: PEFT

1. Compare the number above with the number of trainable parameters in the original model. What changed? How do you expect this to affect fine-tuning results?
2. Suppose that we wanted to use rank $r = 4$ LoRA for fine-tuning the decoder self-attention block of GPT-2. How many parameters would the lower rank matrices A and B have (see slides 27-29 for reference)?

Outlook: PEFT in practice

Below, more optional resources for the other types of fine-tuning introduced in the (LoRA, QLoRA) can be found.

- Blog post on QLoRA with `transformers`
- Overview blogpost on PEFT with `transformers`
- PEFT package based on transformers
- QLoRA paper

RL fine-tuning

Reinforcement learning is often introduced a separate type of machine learning, in addition to supervised and unsupervised learning. Reinforcement learning broadly defines the field of study and the methods for training agents to learn to take actions that (optimally) achieve a goal, based on experience in the environment. It can be seen as the computational formalization of trial-and-error learning.

The key difference to supervised learning is that the agent (the terms "model", "LM" and "agent" will be used interchangeably in this section) learns which actions are useful for achieving the goal itself, rather than being shown the "ground truth" optimal actions as in supervised learning. The task of the developer is, therefore, to correctly specify the goal, and the agent will "discover" a way to achieve it. In the formal framework which underpins RL (namely, MDPs), the goal is represented via the *reward function*. This function assigns high rewards to desired outcomes, and low rewards to undesired ones, therefore implicitly representing a goal. It is important to note that, in general, the approach of RL allows to specify what the developer want the agent to learn to do (i.e., the goal), but *not necessarily how*, exactly. Correct specification of the goal is a far from trivial task and a lot of current research goes into understanding how to specify these goals in the field of *alignment* (more on this in future sessions).

Using RL for fine-tuning LLMs is one of the main methodological innovations that seems to have led to the impressive performance of SOTA LLMs. In particular, RL allows to fine-tune LLMs towards human preferences and commericial usability, because its mechanics lend itself to training a model based on a more abstract signal whether an output is good or bad (i.e., let it discover how to generate output that recevies a "good" reward!), rather than based on particular demonstrations. This is especially useful because the objectives of fine-tuning of SOTA LLMs often include aspects that are very difficult to specify via specific demonstrations, like being *helpful, honest, harmless* (Bai et al., 2022). To this end, instead of using supervised learning, *human feedback* can be used as a reward signal to fine-tune the model. This is why the fine-tuning technique in question is called Reinforcement Learning from Human Feedback (RLHF).

Below, practical aspects of the core components of RLHF are discussed. These core components are (see this figure for an overview):

- the policy (i.e., the backbone LLM),
- the supervised fine-tuning data (SFT) and training,
- the reward modeling data and the resulting reward model,
- and, finally the RL training objective (commonly, the PPO algorithm) and the dataset for fine-tuning.

As with standard LM training, there are packages which implement some of the steps required for training for us. We will look at the `transformers` based package `trl`.

Policy

As the policy, a pretrained, sufficintly large LM is usually chosen. For instance, Llama offers a suite of models where both the initial LM, i.e., the *base* model and the resulting fine-tuned model is provided. For Llama-2 (Touvron et al. (2022)), the paper provides some information about the details of the training and the architecture of the models.

Under state-of-the-art models, usually LMs of at least >=1B parameters are used as policies for further fine-tuning. The intuitive reason is that RLHF is mostly used for creating more general-purpose assistants rather than task-specific models, and therefore, the initial model should have rather large capacity and perform well on a wide range of tasks (which, as we know, tends to come with scale). In other words, the base model should be good -- otherwise: "Garbage in, garbage out" (i.e., it might be very difficult if not impossible to fix a bad base model through fine-tuning).

Of course, it is absolutely possible to use RLHF for task-specific fine-tuning, e.g., early work fine-tuned a model for summarization.

Supervised fine-tuning

This step is a "standard" supervised fine-tuning (SFT) step that is performed as we have discussed above. Some RL-tuned models are closed source, so it is unknown whether any of the PEFT techniques is used; others are open-source and might report their fine-tuning approach.

While the specific methodological details might vary, the conceptual point behind SFT is two-fold: (1) models are often instruction-tuned for turning into actually useful assistants and (2) (more importantly) the model is "nudged" towards outputting human-demonstrated (and therefore, human-preferred) texts for the ultimate goal of the fine-tuning (e.g., being helpful, harmless, honest). This makes subsequent RL-tuning more efficient. Intuitively, this is because the space of actions (i.e., any possible completion, given a prompt!) which the agent has to explore is quite giagantic, and the agent might make quite a lot of errors before "stumbling" upon high-reward actions. Through SFT, the agent is already biased towards the higher-reward space of actions. Therefore, the SFT dataset usually consists of examples of target outputs written by human annotators. This step is also often called *behavioral cloning*.

Exercise 4.1.3: Supervised fine-tuning for RL

1. What would an SFT dataset look like for fine-tuning a model for summarization?
2. What apects do you think are important to keep in mind when performing SFT? (Think: what sorts of examples should human annotators see? What should they be instructed to do? Feel free to also use information from the OpenAI blogpost for inspiration)
3. Below is an example of using the `trl` library for the SFT step. Please go through the code and make sure you understand it. Look at the docs for the training arguments class which is used by default by the SFTTrainer here. By default, for how many epochs is the model trained?

```python
# uncomment and run on Colab / install the package in your enironment
if you haven't yet
# !pip install trl

from datasets import load_dataset
from trl import SFTTrainer

# load dataset
# you can inspect it to get a sense of its contents and formatting
dataset = load_dataset("CarperAI/openai_summarize_tldr",
split="valid")
# load base model
model = AutoModelForCausalLM.from_pretrained("facebook/opt-350m")

# define a function that formats the prompts
# NOTE: the formatting of examples is extremely important for
successful training and deployment for a given task
# for instance, the use of special tokens and prompt formatting should
be consistent with the task, the model (if it already uses special
tokens)
# and should be used in consistent ways in further training

def formatting_prompts_func(example):
    output_texts = []
    # iterate over batch
    for i in range(len(example['prompt'])):
        # retrieve both inputs and target labels
        text = f"{example['prompt'][i]}{example['label'][i]}"
        output_texts.append(text)
    return output_texts

trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    formatting_func=formatting_prompts_func,
    max_seq_length=256,
    dataset_batch_size=8,
)
trainer.train()

# If one wants to pass custom training arguments, an example of doing
so is here:
https://github.com/huggingface/trl/blob/main/examples/scripts/sft.py

# NOTE: if you experience errors running this, setting the version of
accelerate to pip install accelerate==0.27.2 might help
```

Reward modeling

A core component of RL fine-tuning is the reward model. There are various ways of obtaining a reward model. The perhaps most intuitive one, as discussed in the lecture, is obtaining human feedback as the reward signal. Since having humans give online feedback to an LM for thousands of samples is costly and cumbersome, instead, a reward model is trained on human preferences. For training such a reward model, human annotations are collected. As discussed in the lecture, these annotations can take different forms.

Below, we will look at an example based on simple binary comparisons, where human annotators were shown two outputs of the LM sampled for the same input x and had to indicate which of them they preferred (e.g., output y_1 over y_2). Specifically, the reward model is usually initialized from a pretrained LLM (maybe even the same one as the base LM of the policy), and fine-tuned with a specific *head* to output numerical scores. It is usually trained to maximize the difference in predicted scores scores r between the preferred and the rejected answer in a pair. Formally, the model is trained with the following loss:

$$L(\theta) = -\frac{1}{N} E_{(x, y_1, y_2) \sim D}\left[log\left(\sigma\left(r_\theta(x, y_1) - r_\theta(x, y_2)\right)\right)\right]$$

This form of human annotations has proven especially useful for eliciting human intuitions for such difficult-to-capture concepts like "helpfulness". It is much easier for humans to decide which of two options they prefer than, e.g., to consistently assign scalar 1-10 scores to outputs.

Exercise 4.1.4: Reward modeling

1. Consider this well-known dataset from Anthropic for training helpful and harmless assistants. Pick a specific sample. For this sample, which text corresponds to x, y_1, y_2?
2. An example of a model trained with the approach above can be found here. Feel free to explore the repository, if you want. We will not test this model since it is quite large.

However, there are also alternative ways of providing rewards. For instance, some work has used other models to provide feedback, an approach that is called RLAIF (RL from AI feedback). Alternatively, more task-specific reward models can be constructed. For instance, if one wants to use RL-tuning for training a summarization model, the score for evaluating summaries (ROUGE) can be used as the numerical reward. If one wants to train a model for generating positive-sentiment texts only, one can train a reward model on labelled sentiment classification data like, e.g., the IMDB dataset.

Exercise 4.1.5: Task-specific reward modeling

1. The code below provides an example loading a trained reward model. This model was trained on movie reviews, in particular to assign high scores to positive reviews and low scores to negative reviews, as described above. Please look at the code and make sure to understand it. Test it on a few of your own intuitive examples; do the scores (ordinally) correspond to your intuition?

```
from datasets import load_dataset
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
)
import torch

# Load reward model and its tokenizer
reward_tokenizer = AutoTokenizer.from_pretrained("lvwerra/distilbert-
imdb")
reward_model =
AutoModelForSequenceClassification.from_pretrained("lvwerra/distilbert
-imdb")

# Run an example from the IMDB train split to see how the reward model
works
positive_sentence = "" #### YOUR EXAMPLE HERE ####
negative_sentence = "" #### YOUR EXAMPLE HERE ####

input_pos = reward_tokenizer(positive_sentence, return_tensors='pt')
input_neg = reward_tokenizer(negative_sentence, return_tensors='pt')

reward_pos = reward_model(**input_pos)
reward_neg = reward_model(**input_neg)

print("Reward for positive sentence: ", reward_pos)
print("Reward for negative sentence: ", reward_neg)
```

PPO training

Once the SFT model and the reward model are available, the final step is to fine-tune the SFT model with RL. For this, a dataset to fine-tune on is needed again. Depedning on the model, sometimes the same dataset as for SFT used, sometimes a similar dataset with other inputs is used. Note that now the dataset doesn't need any labels, but only the inputs (i.e., initial "states") based on which the model will generate predictions (i.e., actions) which, in turn, will be assigned rewards (by the reward model which represents the environment).

There are different algorithms for training the policy. One currently common choice is the Proximal Policy Optimization (PPO). Its components were developed in order to stabilize policy updates and speed up convergence. However, there is a core shared idea between PPO and other algorithms in the category of *policy-gradient* methods. Specifically, the theorem behind this class of methods shows that the policy can be trained with a loss based on "trials and errors" (i.e., based on sampling; this objective has been shown to update the policy in such a way that, by following it, the agent is expected to receive higher rewards). Specifically, for a training step, we can sample actions, retrieve their log probability under the current policy, get rewards for these actions, and calculate weight updates based on the product of log probability and the reward.

Note that choosing *hyperparameters* for successful RL fine-tuning is very important. While there are no proven results, the community best practice seems to indicate that the batch size should be relatively large, and the training should be quite short (e.g., only one epoch) to avoid undesired effects. Some of these details can be found, e.g., in the Llama-2 report.

Exercise 4.1.6: RL training

1. Suppose you train an LM for summarization and have a suitable reward model and dataset of input texts. Consider the last sentence of the explanation above. Please provide a specific details of such a training step with this summarization example (in words).
2. An example of using the `trl` library for training a model to generate positive model reviews (using the reward model above!) with PPO can be found here. Please look at the code and try to understand all of it! Please ask questions or research if anything is unclear, this approach might be relevant for your own future exercise ;)

Optional outlook

RL fine-tuning is an active area of research, so there are many developments and new methods. Below are some optional resources if you want to know more.

- Rafailov et al. (2023) Direct Preference Optimization: Your Language Model is Secretly a Reward Model
- Gao et al. (2022) Scaling Laws for Reward Model Overoptimization