**Training FAQ**

**What Metrics Should I Look at?**

When performing classical supervised fine-tuning of language models, the loss (especially the validation loss) serves as a good indicator of the training progress. However, in Reinforcement Learning (RL), the loss becomes less informative about the model’s performance, and its value may fluctuate while the actual performance improves.

To address this, we recommend focusing on two key metrics first:

**Mean Reward**: The primary goal is to maximize the reward achieved by the model during RL training. **Objective KL Divergence**: KL divergence (Kullback-Leibler divergence) measures the dissimilarity between two probability distributions. In the context of RL training, we use it to quantify the difference between the current model and a reference model. Ideally, we want to keep the KL divergence between 0 and 10 to ensure the model’s generated text remains close to what the reference model produces.

However, there are more metrics that can be useful for debugging, checkout the [logging section](https://huggingface.co/docs/trl/logging).

**Why Do We Use a Reference Model, and What’s the Purpose of KL Divergence?**

When training RL models, optimizing solely for reward may lead to unexpected behaviors, where the model exploits the environment in ways that don’t align with good language generation. In the case of RLHF, we use a reward model trained to predict whether a generated text is highly ranked by humans.

However, the RL model being optimized against the reward model may learn patterns that yield high reward but do not represent good language. This can result in extreme cases where the model generates texts with excessive exclamation marks or emojis to maximize the reward. In some worst-case scenarios, the model may generate patterns completely unrelated to natural language yet receive high rewards, similar to adversarial attacks.

**Figure:** Samples without a KL penalty from <https://arxiv.org/pdf/1909.08593.pdf>.

To address this issue, we add a penalty to the reward function based on the KL divergence between the current model and the reference model. By doing this, we encourage the model to stay close to what the reference model generates.

**What Is the Concern with Negative KL Divergence?**

If you generate text by purely sampling from the model distribution things work fine in general. But when you use the generate method there are a few caveats because it does not always purely sample depending on the settings which can cause KL-divergence to go negative. Essentially when the active model achieves log\_p\_token\_active < log\_p\_token\_ref we get negative KL-div. This can happen in a several cases:

* **top-k sampling**: the model can smooth out the probability distribution causing the top-k tokens having a smaller probability than those of the reference model but they still are selected
* **min\_length**: this ignores the EOS token until min\_length is reached. thus the model can assign a very high log prob to the EOS token and very low prob to all others until min\_length is reached
* **batched generation**: finished sequences in a batch are padded until all generations are finished. The model can learn to assign very low probabilities to the padding tokens unless they are properly masked or removed.

These are just a few examples. Why is negative KL an issue? The total reward R is computed R = r - beta \* KL so if the model can learn how to drive KL-divergence negative it effectively gets a positive reward. In many cases it can be much easier to exploit such a bug in the generation than actually learning the reward function. In addition the KL can become arbitrarily small thus the actual reward can be very small compared to it.

So how should you generate text for PPO training? Let’s have a look!

**How to generate text for training?**

In order to avoid the KL issues described above we recommend to use the following settings:

Copied

generation\_kwargs = {

"min\_length": -1, *# don't ignore the EOS token (see above)*

"top\_k": 0.0, *# no top-k sampling*

"top\_p": 1.0, *# no nucleus sampling*

"do\_sample": True, *# yes, we want to sample*

"pad\_token\_id": tokenizer.eos\_token\_id, *# most decoder models don't have a padding token - use EOS token instead*

"max\_new\_tokens": 32, *# specify how many tokens you want to generate at most*

}

With these settings we usually don’t encounter any issues. You can also experiments with other settings but if you encounter issues with negative KL-divergence try to go back to these and see if they persist.

**How can debug your own use-case?**

Debugging the RL pipeline can be challenging due to its complexity. Here are some tips and suggestions to make the process easier:

* **Start from a working example**: Begin with a working example from the trl repository and gradually modify it to fit your specific use-case. Changing everything at once can make it difficult to identify the source of potential issues. For example, you can start by replacing the model in the example and once you figure out the best hyperparameters try to switch to your dataset and reward model. If you change everything at once you won’t know where a potential problem comes from.
* **Start small, scale later**: Training large models can be very slow and take several hours or days until you see any improvement. For debugging this is not a convenient timescale so try to use small model variants during the development phase and scale up once that works. That being said you sometimes have to be careful as small models might not have the capacity to solve a complicated task either.
* **Start simple**: Try to start with a minimal example and build complexity from there. Your use-case might require for example a complicated reward function consisting of many different rewards - try to use one signal first and see if you can optimize that and then add more complexity after that.
* **Inspect the generations**: It’s always a good idea to inspect what the model is generating. Maybe there is a big in your post-processing or your prompt. Due to bad settings you might cut-off generations too soon. These things are very hard to see on the metrics but very obvious if you look at the generations.
* **Inspect the reward model**: If you reward is not improving over time maybe there’s an issue with the reward model. You can look at extreme cases to see if it does what it should: e.g. in the sentiment case you can check if simple positive and negative examples really get different rewards. And you can look at the distribution of your dataset. Finally, maybe the reward is dominated by the query which the model can’t affect so you might need to normalize this (e.g. reward of query+response minus reward of the query).

These are just a few tips that we find helpful - if you have more useful tricks feel free to open a PR to add them as well!