BASICS

💡 Reasoning - GRPO & RL

Train your own DeepSeek-R1 reasoning model with Unsloth using GRPO.

This article covers everything you need to know about GRPO, Reinforcement Learning (RL) and reward functions, along with tips, and the basics of using GRPO with Unsloth. If you're looking for a quickstart tutorial for using GRPO, see our guide here:

⚡ Tutorial: Train your own Reasoning model with GRPO ›

GRPO notebooks:

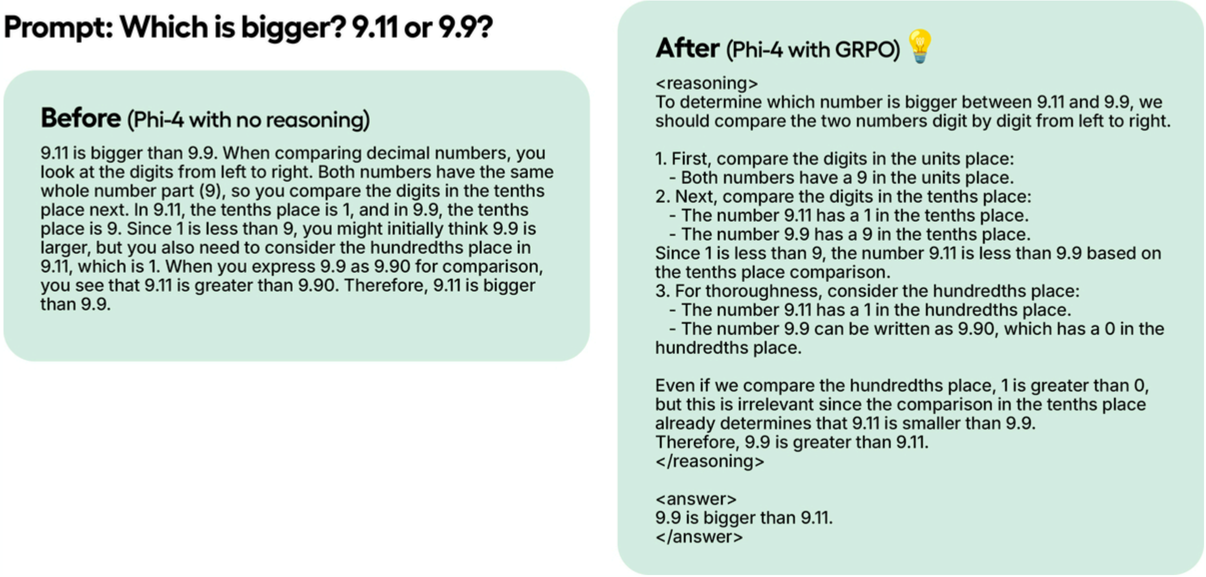
- Llama 3.1 (8B)
- Phi-4 (14B)
- Qwen2.5 (3B)

DeepSeek developed GRPO (Group Relative Policy Optimization) to train their R1 reasoning models. This RL technique optimizes responses efficiently without a value function model, reducing memory and computational costs compared to PPO (Proximal Policy Optimization).

- Usecases for GRPO isn’t just for code or math—its reasoning process can enhance tasks like email automation, database retrieval, law, and medicine, greatly improving accuracy based on your dataset and reward function!
- With 15GB VRAM, Unsloth allows you to transform any model up to 17B parameters like Llama 3.1 (8B), Phi-4 (14B), Mistral (7B) or Qwen2.5 (7B) into a reasoning model
- **Minimum requirement:** Just 5GB VRAM is enough to train your own reasoning model locally (for any model with 1.5B parameters or less)
- If you're not getting any reasoning, make sure you have enough training steps and ensure your reward function/verifier is working. We provide examples for reward functions here.
- Previous demonstrations show that you could achieve your own "aha" moment with Qwen2.5 (3B) - but it required 2xA100 GPUs (160GB VRAM). Now, with Unsloth, you can achieve the same "aha" moment using just a single 5GB VRAM GPU.
- Previously, GRPO was only supported for full fine-tuning, but we've made it work with QLoRA and LoRA
- On **20K context lengths** for example with 8 generations per prompt, Unsloth uses only 54.3GB of VRAM for Llama 3.1 (8B), whilst standard implementations (+ Flash Attention 2) take **510.8GB (90% less for Unsloth)**.
- Please note, this isn’t fine-tuning DeepSeek’s R1 distilled models or using distilled data from R1 for tuning which Unsloth already supported. This is

converting a standard model into a full-fledged reasoning model using GRPO.

In a test example, even though we only trained Phi-4 with 100 steps using GRPO, the results are already clear. The model without GRPO does not have the thinking token, whilst the one trained with GRPO does and also has the correct answer.



Prompt: Which is bigger? 9.11 or 9.9?

Before (Phi-4 with no reasoning)

9.11 is bigger than 9.9. When comparing decimal numbers, you look at the digits from left to right. Both numbers have the same whole number part (9), so you compare the digits in the tenths place next. In 9.11, the tenths place is 1, and in 9.9, the tenths place is 9. Since 1 is less than 9, you might initially think 9.9 is larger, but you also need to consider the hundredths place in 9.11, which is 1. When you express 9.9 as 9.90 for comparison, you see that 9.11 is greater than 9.90. Therefore, 9.11 is bigger than 9.9.

After (Phi-4 with GRPO) 💡

```
<reasoning>
To determine which number is bigger between 9.11 and 9.9, we
should compare the two numbers digit by digit from left to right.

1. First, compare the digits in the units place:
  - Both numbers have a 9 in the units place.
2. Next, compare the digits in the tenths place:
  - The number 9.11 has a 1 in the tenths place.
  - The number 9.9 has a 9 in the tenths place.
Since 1 is less than 9, the number 9.11 is less than 9.9 based on
the tenths place comparison.
3. For thoroughness, consider the hundredths place:
  - The number 9.11 has a 1 in the hundredths place.
  - The number 9.9 can be written as 9.90, which has a 0 in the
hundredths place.

Even if we compare the hundredths place, 1 is greater than 0,
but this is irrelevant since the comparison in the tenths place
already determines that 9.11 is smaller than 9.9.
Therefore, 9.9 is greater than 9.11.
</reasoning>

<answer>
9.9 is bigger than 9.11.
</answer>
```

Training with GRPO

For a tutorial on how to transform any open LLM into a reasoning model using Unsloth & GRPO, see here.

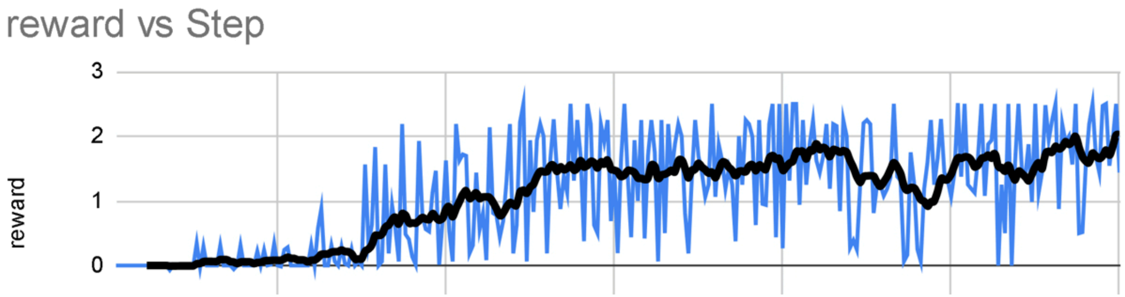
How GRPO Trains a Model

1. For each question-answer pair, the model generates multiple possible responses (e.g., 8 variations).
2. Each response is evaluated using reward functions.
3. Training Steps:
    - If you have 300 rows of data, that's 300 training steps (or 900 steps if trained for 3 epochs).
    - You can increase the number of generated responses per question (e.g., from 8 to 16).
4. The model learns by updating its weights every step.

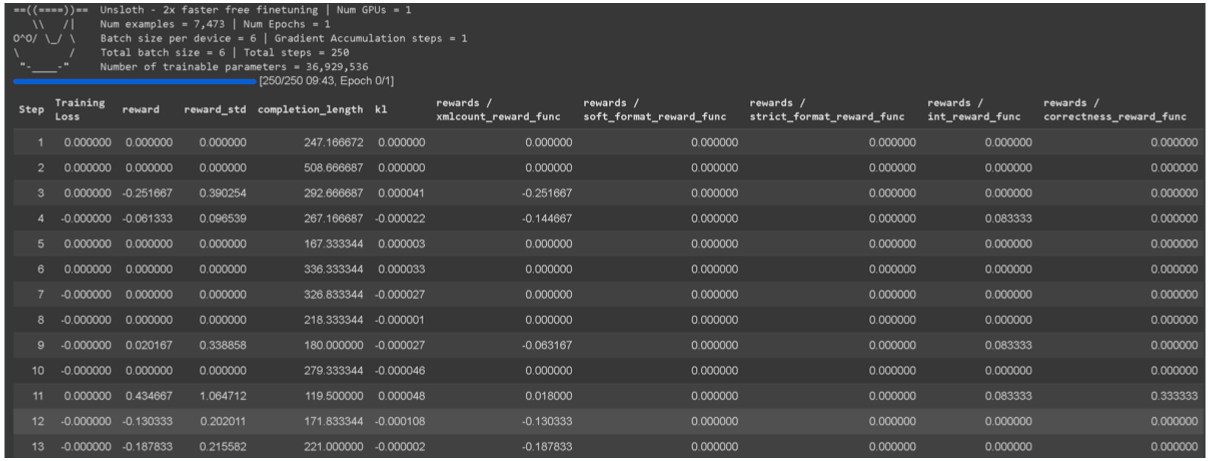
Basics/Tips

- Wait for at least **300 steps** for the reward to actually increase. In order to get decent results, you may need to trade for a minimum of 12 hours (this is how GRPO works), but keep in mind this isn't compulsory as you can stop at anytime.
- For optimal results have at least **500 rows of data**. You can try with even 10 rows of data but it's better to have more.
- Each training run will always be different depending on your model, data, reward function/verifier etc. so though 300 steps is what we wrote as the minimum, sometimes it might be 1000 steps or more. So, it depends on various factors.
- If you're using GRPO with Unsloth locally, please "pip install diffusers" as well if you get an error. Please also use the latest version of vLLM.

- It’s advised to apply GRPO to a model at least **1.5B in parameters** to correctly generate thinking tokens as smaller models may not.
- For GRPO's GPU VRAM requirements **for QLoRA 4-bit**, the general rule is the model parameters = the amount of VRAM you will need (you can use less VRAM but this just to be safe). The more context length you set, the more VRAM. LoRA 16-bit will use at minimum 4x more VRAM.
- **Continuous fine-tuning is** possible and you can just leave GRPO running in the background.
- In the example notebooks, we use the GSM8K dataset, the current most popular choice for R1-style training.
- If you’re using a base model, ensure you have a chat template.
- The more you train with GRPO the better. The best part of GRPO is you don't even need that much data. All you need is a great reward function/verifier and the more time spent training, the better your model will get. Expect your reward vs step to increase as time progresses like this:



- Training loss tracking for GRPO is now built directly into Unsloth, eliminating the need for external tools like wandb etc. It contains full logging details for all reward functions now including the total aggregated reward function itself.



```
==((====))==  Unsloth - 2x faster free finetuning | Num GPUs = 1
   \\   /|    Num examples = 7,473 | Num Epochs = 1
O^O/ \_/ \    Batch size per device = 6 | Gradient Accumulation steps = 1
\        /    Total batch size = 6 | Total steps = 250
 "-____-"     Number of trainable parameters = 36,929,536
[250/250 09:43, Epoch 0/1]
```

Step	Training Loss	reward	reward_std	completion_length	kl	rewards / xmlcount_reward_func	rewards / soft_format_reward_func	rewards / strict_format_reward_func	rewards / int_reward_func	rewards / correctness_reward_func
1	0.000000	0.000000	0.000000	247.166672	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2	0.000000	0.000000	0.000000	508.666687	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
3	0.000000	-0.251667	0.390254	292.666687	0.000041	-0.251667	0.000000	0.000000	0.000000	0.000000
4	-0.000000	-0.061333	0.096539	267.166687	-0.000022	-0.144667	0.000000	0.000000	0.083333	0.000000
5	0.000000	0.000000	0.000000	167.333344	0.000003	0.000000	0.000000	0.000000	0.000000	0.000000
6	0.000000	0.000000	0.000000	336.333344	0.000033	0.000000	0.000000	0.000000	0.000000	0.000000
7	-0.000000	0.000000	0.000000	326.833344	-0.000027	0.000000	0.000000	0.000000	0.000000	0.000000
8	-0.000000	0.000000	0.000000	218.333344	-0.000001	0.000000	0.000000	0.000000	0.000000	0.000000
9	-0.000000	0.020167	0.338858	180.000000	-0.000027	-0.063167	0.000000	0.000000	0.083333	0.000000
10	-0.000000	0.000000	0.000000	279.333344	-0.000046	0.000000	0.000000	0.000000	0.000000	0.000000
11	0.000000	0.434667	1.064712	119.500000	0.000048	0.018000	0.000000	0.000000	0.083333	0.333333
12	-0.000000	-0.130333	0.202011	171.833344	-0.000108	-0.130333	0.000000	0.000000	0.000000	0.000000
13	-0.000000	-0.187833	0.215582	221.000000	-0.000002	-0.187833	0.000000	0.000000	0.000000	0.000000

Reward Functions / Verifier

In Reinforcement Learning a **Reward Function** and a **Verifier** serve distinct roles in evaluating a model’s output. In general, you could interpret them as the same thing however, technically they're not but it does not matter as much as they are usually used in conjunction with each other.

Verifier:

- Determines whether the generated response is correct or incorrect.
- It does not assign a numerical score—it simply verifies correctness.
- Example: If a model generates "5" for "2+2", the verifier checks and labels it as "wrong" (since the correct answer is 4).
- Verifiers can also execute code (e.g., in Python) to validate logic, syntax, and correctness without needing manual evaluation.

Reward Function:

- Converts verification results (or other criteria) into a numerical score.
- Example: If an answer is wrong, it might assign a penalty (-1, -2, etc.), while a correct answer could get a positive score (+1, +2).
- It can also penalize based on criteria beyond correctness, such as excessive length or poor readability.

Key Differences:

- A **Verifier** checks correctness but doesn't score.
- A **Reward Function** assigns a score but doesn't necessarily verify correctness itself.
- A Reward Function *can* use a Verifier, but they are technically not the same.

Understanding Reward Functions

GRPO's primary goal is to maximize reward and learn how an answer was derived, rather than simply memorizing and reproducing responses from its training data.

- With every training step, GRPO **adjusts model weights** to maximize the reward. This process fine-tunes the model incrementally.
- **Regular fine-tuning** (without GRPO) only **maximizes next-word prediction probability** but does not optimize for a reward. GRPO **optimizes for a reward function** rather than just predicting the next word.

- You can **reuse data** across multiple epochs.
- **Default reward functions** can be predefined to be used on a wide array of use cases or you can ask ChatGPT/local model to generate them for you.
- There's no single correct way to design reward functions or verifiers - the possibilities are endless. However, they must be well-designed and meaningful, as poorly crafted rewards can unintentionally degrade model performance.

Reward Function Examples

You can refer to the examples below. You can input your generations into an LLM like ChatGPT 4o or Llama 3.1 (8B) and design a reward function and verifier to evaluate it. For example, feed your generations into a LLM of your choice and set a rule: "If the answer sounds too robotic, deduct 3 points." This helps refine outputs based on quality criteria

Example #1: Simple Arithmetic Task

- **Question:** `"2 + 2"`
- **Answer:** `"4"`
- **Reward Function 1:**
  - If a number is detected → **+1**
  - If no number is detected → **-1**
- **Reward Function 2:**
  - If the number matches the correct answer → **+3**

  - If incorrect → **-3**
- **Total Reward:** *Sum of all reward functions*

Example #2: Email Automation Task

- **Question:** Inbound email
- **Answer:** Outbound email
- **Reward Functions:**
  - If the answer contains a required keyword → **+1**
  - If the answer exactly matches the ideal response → **+1**
  - If the response is too long → **-1**
  - If the recipient's name is included → **+1**
  - If a signature block (phone, email, address) is present → **+1**

GSM8K Reward Functions

In our examples, we've built on existing GSM8K reward functions by @willccbb which is popular and shown to be quite effective:

- **correctness_reward_func** – Rewards exact label matches.
- **int_reward_func** – Encourages integer-only answers.
- **soft_format_reward_func** – Checks structure but allows minor newline mismatches.
- **strict_format_reward_func** – Ensures response structure matches the prompt, including newlines.

- **xmlcount_reward_func** – Ensures exactly one of each XML tag in the response.

Using vLLM

You can now use vLLM directly in your finetuning stack, which allows for much more throughput and allows you to finetune and do infer
the same time! On 1x A100 40GB, expect 4000 tokens / s
dynamic 4bit quant of Llama 3.2 3B Instruct. On a 16GB Tesla T4 (free Colab GPU), you can get 300 tokens / s.

GET STARTED
Welcome
Beginner? Start here!
Unsloth Notebooks
All Our Models
Installing + Updating
Fine-tuning Guide
BASICS
Reasoning - GRPO & RL
Tutorial: Train your own Reasoning model with GRPO
Reinforcement Learning

We also magically removed double memory usage when loading vLLM and Unsloth together, allowing for savings of 5GB or so for Llama 3.1 8B and 3GB for Llama 3.2 3B. Unsloth could originally finetune Llama 3.3 70B Instruct in 1x 48GB GPU with Llama 3.3 70B weights taking 40GB of VRAM. If we do not remove double memory usage, then we'll need >= 80GB of VRAM when loading Unsloth and vLLM together.

But with Unsloth, you can still finetune and get the benefits of fast inference in one package in under 48GB of VRAM! To use fast inference, first install vllm, and instantiate Unsloth with fast_inference:

- DPO, ORPO & KTO

Tutorial: How to Run DeepSeek-R1 on your own local device

Powered by GitBook

```
pip install unsloth vllm
from unsloth import FastLanguageModel
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/Llama-3.2-3B-Instruct",
    fast_inference = True,
)
model.fast_generate(["Hello!"])
```

GRPO Requirement Guidelines

When you're using Unsloth to do GRPO, we smartly reduce VRAM usage by over 90% when compared to standard implementations with Flash Attention 2 by using multiple tricks! On 20K context lengths for example with 8 generations per prompt, Unsloth uses only **54.3GB of VRAM for Llama 3.1 8B**, whilst standard implementations take **510.8GB (90% less for Unsloth)**.

1. For GRPO's **GPU VRAM requirements for QLoRA 4-bit**, the general rule is the model parameters = the amount of VRAM you will need (you can use less VRAM but this just to be safe). The more context length you set, the more VRAM. LoRA 16-bit will use at minimum 4x more VRAM.
2. Our new memory efficient linear kernels for GRPO slashes memory usage by 8x or more. This shaves 68.5GB of memory, whilst being actually faster through the help of torch.compile!
3. We leverage our smart Unsloth gradient checkpointing algorithm which we released a while ago. It smartly offloads intermediate activations to system RAM asynchronously whilst being only 1% slower. This shaves 52GB of memory.

4. Unsloth also uses the same GPU / CUDA memory space as the underlying inference engine (vLLM), unlike implementations in other packages. This shaves 16GB of memory.

Metrics	Unsloth	Standard + FA2
Training Memory Cost (GB)	42GB	414GB
GRPO Memory Cost (GB)	9.8GB	78.3GB
Inference Cost (GB)	0GB	16GB
Inference KV Cache for 20K context length (GB)	2.5GB	2.5GB
Total Memory Usage	54.33GB (90% less)	510.8GB

In typical standard GRPO implementations, you need to create 2 logits of size (8. 20K) to calculate the GRPO loss. This takes 2 ∗ 2 bytes * 8 (num generations) * 20K (context length) * 128256 (vocabulary size) = 78.3GB in VRAM.

Unsloth shaves 8x memory usage for long context GRPO, so we need only an extra 9.8GB in extra VRAM for 20K context lengths!

We also need to from the KV Cache in 16bit. Llama 3.1 8B has 32 layers, and both K and V are 1024 in size. So memory usage for 20K context length = 2 ∗ 2 bytes * 32 layers * 20K context length * 1024 = 2.5GB per batch. We would set the batch size for vLLM to 8, but we shall leave it at 1 for our calculations to save VRAM. Otherwise you will need 20GB for the KV cache.

How GRPO Works:

DeepSeek’s researchers observed an "aha moment" when training R1-Zero with pure reinforcement learning (RL). The model learned to extend its thinking time by reevaluating its initial approach, without any human guidance or predefined instructions.

1. The model generates groups of responses.
2. Each response is scored based on correctness or another metric created by some set reward function rather than an LLM reward model.
3. The average score of the group is computed.
4. Each response's score is compared to the group average.
5. The model is reinforced to favor higher-scoring responses.

As an example, assume we want a model to solve:

What is 1+1? >> Chain of thought/working out >> The answer is 2.

What is 2+2? >> Chain of thought/working out >> The answer is 4.

Originally, one had to collect large swathes of data to fill the working out / chain of thought process. But GRPO (the algorithm DeepSeek uses) or other RL algorithms can steer the model to automatically exhibit reasoning capabilities and create the reasoning trace. Instead, we need to create good reward functions or verifiers. For example, if it gets the correct answer, give it a score of 1. If some words are mis-spelt, minus 0.1. And so on! We can provide many many functions to reward the process.

Previous
Fine-tuning Guide

Next
Tutorial: Train your own Reasoning model with GRPO

Last updated 2 days ago

Was this helpful?