Contents

Chapter 1: From Loss to Reward

Welcome. You have a pre-trained language model. It knows grammar, facts, and can write in the style of a pirate or a poet. You've likely fine-tuned it on a specific dataset, a process dominated by a single, powerful idea: **cross-entropy loss**. This chapter is about the limits of that idea and the paradigm shift required to overcome them. We will move from the world of fixed targets and absolute "correctness" to a more flexible, powerful world of **rewards**.

Our journey starts by revisiting the familiar ground of supervised learning, understanding not just how it works, but *why* it works and where it falls short. From there, we'll build the intuition for a new way of training, one that allows us to optimize models for goals that are too complex or subjective to be captured in a static dataset.

1.1 The Familiar World: Supervised Fine-Tuning

Supervised Fine-Tuning (SFT) is the workhorse of model customization. The premise is simple: you have a set of prompts (`x`) and a corresponding set of "ideal" responses (`y`). Your goal is to teach the model to produce `y` when it sees `x`.

The mechanism for this is the **autoregressive cross-entropy loss**. At each step of generating a response, the model predicts a probability distribution over its vocabulary for the next token. We compare this distribution to the *actual* next token in the ground-truth answer `y` and calculate the negative log-likelihood.

In code, this looks like summing the log-probabilities of each target token:

```
log_probs = F.log_softmax(logits, dim=-1)
token_log_probs = log_probs.gather(dim=-1, index=target_ids.unsqueeze(-1)).squeeze(-1)
loss = -token_log_probs.sum()
```

We gather the log-probability assigned to each correct token, then sum and negate. This is the cross-entropy loss: maximize the probability of the correct sequence.

A Look at the Code A typical SFT training loop in PyTorch looks something like this. We're omitting boilerplate like model initialization and data loading to focus on the core logic.

```
import torch
import torch.nn.functional as F
from torch.optim import AdamW
from transformers import AutoModelForCausalLM, AutoTokenizer


model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen3-8B-Instruct")
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen3-8B-Instruct")

def get_dummy_dataloader(tokenizer, batch_size=2):
    prompts = ["What is the capital of France?", "Translate 'hello' to Spanish."]
    responses = ["The capital of France is Paris.", "The translation of 'hello' to Spanish is 'hola'."]

    inputs = tokenizer(prompts, padding=True, return_tensors="pt")
    full_text = [p + " " + r for p, r in zip(prompts, responses)]
    labels = tokenizer(full_text, padding=True, return_tensors="pt").input_ids


    dataset = torch.utils.data.TensorDataset(inputs.input_ids, inputs.attention_mask, labels)
    return torch.utils.data.DataLoader(dataset, batch_size=batch_size)

dataloader = get_dummy_dataloader(tokenizer)
optimizer = AdamW(model.parameters(), lr=5e-5)
```

```
model.train()

for epoch in range(3):
    for batch in dataloader:
        input_ids, attention_mask, labels = batch

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels
        )

        loss = outputs.loss

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        print(f"Epoch: {epoch}, Loss: {loss.item():.4f}")
```

A simplified SFT training loop. The key is that `model()` directly returns the cross-entropy loss when `labels` are provided.

This process is incredibly effective. It's how models learn to follow instructions, adopt personas, and format their outputs. The loss is simple, the gradients are stable, and the outcome is predictable.

But it has a fundamental, unshakeable assumption: **there is a single, pre-written, perfect y for every x.**

1.2 When "Correct" is a Matter of Opinion

The SFT paradigm breaks down when we move from factual recall to tasks involving creativity, judgment, or complex reasoning. Consider these prompts:

1. **"Write a short, melancholic poem about a rusty machine."** Is there a single correct poem? Of course not. There are thousands of valid, beautiful, and "correct" responses. SFT would force us to pick just one, arbitrarily labeling all other possibilities as incorrect.
2. **"Summarize this 10,000-word scientific paper on quantum computing."** One summary might focus on the experimental setup, another on the theoretical implications. Both could be valid. Is the best summary 200 words or 300? SFT requires us to make a single choice and stick to it.
3. **"Write Python code to solve this competitive programming problem. The goal is to maximize performance."** One solution might be faster but use more memory. Another might be more readable but slightly slower. Which is the ground truth? The "best" answer depends on constraints that may not even be in the prompt.

In all these cases, the notion of a single y is not just limiting; it's actively misleading. By training on one specific "good" answer, we inadvertently teach the model that all other good answers are bad. This is the **"straightjacket" of supervised learning**.

To train a model that can navigate these ambiguous tasks, we need a more flexible signal of quality.

1.3 A More General Signal: Rewards

Instead of a binary "right" or "wrong," what if we could assign a *score* to any possible output? This is the core idea of a **reward function**.

A reward function, `R(y)`, takes a generated response `y` (and often the prompt `x`) and returns a scalar value indicating how "good" that response is. A higher value means a better response.

- `R(y) = 1.0` (Good)
- `R(y) = 0.1` (Mediocre)
- `R(y) = -0.5` (Bad)

This is a profound shift. We are no longer providing the "correct" answer. We are providing a *metric for success*. The model's job is to figure out how to generate responses that maximize this metric.

Let's design a simple reward function. Imagine we want to teach a model to be more concise. We could penalize it for long responses.

```python
def conciseness_reward(response_text: str, target_length: int = 50) -> float:
    """
    A simple reward function that encourages conciseness.
    The reward is highest at the target length and decreases quadratically.
    """
    length = len(response_text.split())

    penalty = ((length - target_length) / target_length) ** 2

    reward = 1.0 - penalty

    return max(reward, -1.0)

response_good = "This response is exactly fifty words long, which is the target."
print(f"Good Response Reward: {conciseness_reward(response_good, target_length=12):.2f}")

response_long = "This is a very long and verbose response that goes far beyond the intended target leng
print(f"Long Response Reward: {conciseness_reward(response_long, target_length=12):.2f}")

response_short = "Too short."
print(f"Short Response Reward: {conciseness_reward(response_short, target_length=12):.2f}")
```

This function isn't perfect, but it captures our goal: "be concise." It doesn't care *what* the model says, only that it respects the length constraint. We've replaced a fixed target with a behavioral objective.

1.4 The RL Paradigm: Maximizing Expected Reward

With a reward function in hand, our training objective changes. We no longer want to maximize the probability of a specific sequence. Instead, we want to **maximize the expected reward** over all possible sequences the model could generate.

The goal: adjust the model's parameters such that, on average, the rewards of generated responses are as high as possible. In code, we'd want something like:

```
responses = [model.generate(prompt) for _ in range(num_samples)]
rewards = torch.tensor([reward_fn(r) for r in responses])
expected_reward = rewards.mean()
```

How do we optimize this? We can't just use cross-entropy. The reward function might not even be differentiable! This is where the **policy gradient** method comes in. It's the mathematical bridge that connects a (potentially non-differentiable) reward signal to the model's logits, which we *can* differentiate.

The core intuition is simple and powerful: 1. **Sample** a response `y` from the model's current policy. 2. **Evaluate** the response to get a reward. 3. If the reward is high, **increase the probability** of the tokens in the sequence you just generated. 4. If the reward is low, **decrease the probability** of those same tokens.

The REINFORCE algorithm implements this: multiply the gradient of the log-probability by the reward, then update:

```
log_prob = compute_log_prob(model, prompt, response)
loss = -reward * log_prob
loss.backward()
optimizer.step()
```

A good response gets a positive push (negative loss → weights move to increase probability). A bad one gets a negative push.

The RL Training Loop in Code Let's sketch out what this new training loop looks like. Again, we'll focus on the conceptual flow.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen3-8B-Instruct")
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen3-8B-Instruct")
optimizer = AdamW(model.parameters(), lr=1e-6)

def conciseness_reward(response_text: str, target_length: int = 50) -> float:
    length = len(response_text.split())
    penalty = ((length - target_length) / target_length) ** 2
    return 1.0 - penalty

prompts = ["Write a short story about a dragon.", "Explain gravity in simple terms."]
model.train()

for epoch in range(3):
    for prompt_text in prompts:
        input_ids = tokenizer(prompt_text, return_tensors="pt").input_ids

        generated_ids = model.generate(
            input_ids,
            max_new_tokens=100,
            do_sample=True,
            top_p=0.9,
            pad_token_id=tokenizer.eos_token_id
        )

        full_sequence_ids = generated_ids[0]
        response_ids = full_sequence_ids[input_ids.shape[-1]:]

        response_text = tokenizer.decode(response_ids, skip_special_tokens=True)
        reward = conciseness_reward(response_text, target_length=50)

        outputs = model(input_ids=full_sequence_ids)
        logits = outputs.logits

        response_logits = logits[:, input_ids.shape[-1]-1:-1, :]
        response_labels = response_ids.unsqueeze(0)

        log_probs = torch.nn.functional.log_softmax(response_logits, dim=-1)
```

```
        token_log_probs = torch.gather(log_probs, 2, response_labels.unsqueeze(2)).squeeze(-1)

        loss = -token_log_probs.sum() * reward

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        print(f"Epoch: {epoch}, Prompt: '{prompt_text[:20]}...', Reward: {reward:.2f}, Loss: {loss.item
```

A conceptual RL training loop. Note the key differences: sampling with `model.generate()`, calling an external `reward_function`, and manually calculating a `loss` based on log-probabilities and the reward.

1.5 Two Worlds Compared

Let's put the two paradigms side-by-side.

Feature	Supervised Fine-Tuning (SFT)	Reinforcement Learning (RL)
Training Signal	Cross-Entropy Loss	Scalar Reward
Data Source	Static dataset of `(prompt, ideal_response)` pairs.	On-the-fly generation of `(prompt, response, reward)`.
Model's Goal	Minimize the difference between its predictions and a fixed target.	Maximize a score defined by a reward function.
Exploration	None. The path is fixed.	Essential. The model must try different outputs to find high-reward ones.
Gradient	`L = -log P(y_correct | x)`	`Loss = -log P(y_sampled | x) * R(y_sampled)`
Best For	Style adoption, instruction following, factual recall.	Complex reasoning, safety, creativity, subjective goals.

This chapter was about a conceptual shift. We've moved from the comforting certainty of a ground-truth label to the open-ended challenge of optimizing a reward. This new perspective is the foundation for all modern post-training techniques, from controlling toxicity to enabling multi-step reasoning.

In the next chapter, we will dive deep into the first step of our RL loop: the **rollout**. We'll explore how to efficiently generate thousands of responses from our model to create the data we need for policy gradient updates, all while managing the constraints of a single GPU.

Chapter 2: Rollouts - Generating Training Data

In Chapter 1, we established the conceptual shift from supervised learning's cross-entropy loss to the reinforcement learning paradigm of rewards. We framed our goal: instead of teaching a model to mimic a single "correct" answer, we want to encourage it to generate "good" answers, as defined by a reward signal.

This immediately raises a critical question: where does the data for this new learning process come from? In supervised fine-tuning (SFT), our dataset is static. We have a fixed set of prompts and desired responses. In reinforcement learning, we must *create* our own training data in a dynamic loop. This process of data creation is called a **rollout**.

A rollout is the process of giving the model a prompt and letting it generate a complete response, token by token, based on its current parameters. This generated response is then evaluated by our reward function. The combination of (prompt, generated response, reward) becomes a single data point for training.

This chapter is dedicated to the first and most fundamental step in our RL journey: generating the responses that will form our training data. We will explore how to control the generation process to create a diverse set of candidate outputs and, crucially, how to do this on a single 24GB GPU without running out of memory.

The Rollout Loop: A High-Level View

Before we dive into code, let's solidify the concept of the rollout loop. This is the engine of our post-training process. It's a cycle that we will repeat thousands of times, and each cycle inches our model closer to the desired behavior.

The loop consists of four stages:

1. **Prompt**: We start with a prompt from a dataset. This could be a math problem, a coding challenge, or any other task we want the model to learn.
2. **Generate**: The model, in its current state, generates one or more responses to the prompt. This is the "rollout" itself. We don't just generate one response; we often generate a batch of them to explore different possibilities.
3. **Score**: Each generated response is evaluated by a **reward function**. This function assigns a numerical score (the reward) to the response. A high score means the response is "good," and a low score means it's "bad." (We will build this reward function in Chapter 3).
4. **Learn**: The model's parameters are updated based on the responses and their associated rewards. The learning algorithm's job is to increase the probability of generating high-reward responses and decrease the probability of generating low-reward ones. (This is the policy gradient update we'll implement in Chapter 5).

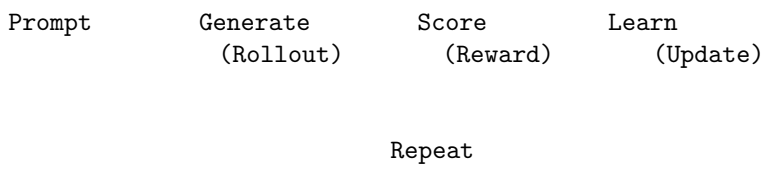

Figure 2.1: The core rollout loop. Generate, Score, Learn, Repeat.

This chapter focuses entirely on the "Generate" step. How can we steer the model to produce a rich and varied set of outputs to feed into the rest of the loop?

Controlling Creativity: Temperature and Top-p Sampling

If you ask a language model the same question multiple times, you might get the exact same answer every time. This is due to a decoding strategy called **greedy decoding**. At each step, the model chooses the single token with the highest probability. It's deterministic and safe, but it's also boring and repetitive. It explores only the single most likely path through the vast space of possible sentences.

For our RL loop, we need variety. We want to see a range of solutions—some good, some bad, some creative, some straightforward. This diversity provides a richer learning signal. If all the responses are identical, the reward is the same for all of them, and the model learns very little.

To introduce diversity, we use **sampling**. Instead of always picking the most likely token, we sample from the probability distribution of the entire vocabulary. Two key parameters control this sampling process: `temperature` and `top-p`.

Temperature Temperature adjusts the "sharpness" of the probability distribution. It's a divisor applied to the logits (the raw scores from the model) before they are converted into probabilities via the softmax function.

- **Low Temperature (e.g., 0.2)**: This makes the distribution sharper. The probabilities of the most likely tokens get exaggerated, while the less likely tokens get suppressed. The model becomes more

confident and deterministic, behaving similarly to greedy decoding. It will produce responses that are safe, coherent, and often repetitive.

- **High Temperature (e.g., 1.0 or higher)**: This flattens the distribution. The probabilities of all tokens become more uniform. The model is now more willing to take risks and choose less likely words. This can lead to more creative, surprising, and sometimes nonsensical outputs. It increases the risk of hallucinations but is essential for exploring new possibilities.

For our rollout generation, we typically want a moderate-to-high temperature (e.g., 0.7) to ensure we generate a diverse set of candidate answers for the reward function to evaluate.

Top-p (Nucleus) Sampling While high temperature encourages creativity, it can sometimes go too far, picking truly bizarre tokens that derail the entire response. **Top-p sampling** provides a safety rail.

Instead of sampling from the entire vocabulary, top-p sampling restricts the sampling pool to a "nucleus" of the most probable tokens whose cumulative probability exceeds a certain threshold `p`.

- `top_p = 0.9`: The model considers the most likely tokens in descending order of probability. It keeps adding them to a list until their cumulative probability is 0.9. All other tokens are discarded for this step, and the model samples only from the nucleus.

This technique is powerful because it's adaptive. For a distribution that is very sharp (the model is very confident), the nucleus might only contain a few tokens. For a flatter distribution (the model is uncertain), the nucleus will be much larger, allowing for more diversity. It lets us use a higher temperature for creativity while ensuring that we don't fall off a cliff by sampling from the long tail of irrelevant tokens.

The Memory Bottleneck: Fitting an 8B Model on a 24GB GPU

We have a plan: use temperature and top-p sampling to generate batches of diverse responses. Now we hit our first major practical hurdle: hardware. The base model we're using, Qwen3-8B-Instruct, has 8 billion parameters. In its standard half-precision (16-bit) format, it requires `8 * 2 = 16` gigabytes of VRAM just to load.

This leaves us with only `24 - 16 = 8` GB for everything else: * The gradients from the backward pass. * The optimizer state (AdamW stores two states per parameter). * The activations from the forward pass. * The input data batch itself.

Running a generation loop and a training update under these conditions is impossible. We would run out of memory before we even started.

This is the central constraint of this book: how to accomplish our goal on a single consumer-grade GPU. The solution is a technique that has become a cornerstone of modern, accessible fine-tuning: **QLoRA**.

Memory-Efficient Fine-Tuning: The QLoRA Explanation

QLoRA is a clever combination of two ideas: **4-bit Quantization (Q)** and **Low-Rank Adaptation (LoRA)**. It allows us to drastically reduce the memory footprint of the base model while still enabling effective training. We will explain it once here, and then use it as a standard tool throughout the book.

1. Quantization: Shrinking the Base Model Quantization is the process of reducing the precision of the numbers used to represent the model's weights. Instead of storing each weight as a 16-bit or 32-bit floating-point number, we store it as a 4-bit integer.

This has a massive impact on memory. A 4-bit model takes up only a quarter of the space of its 16-bit counterpart. For our 8B model, the VRAM required to load the weights drops from 16GB to just ~**5GB**.

This is a lossy compression. We lose some precision, but techniques like `nf4` (4-bit NormalFloat) are designed to preserve the distribution of the original weights, minimizing the impact on performance. The key idea is that the core knowledge of the pre-trained model is robust enough to withstand this precision reduction.

After quantization, the base model's weights are **frozen**. We will not train them. They serve as a knowledgeable but memory-efficient foundation.

2. Low-Rank Adaptation (LoRA): Trainable Adapters If the base model is frozen, how do we train anything? This is where LoRA comes in.

LoRA posits that the *change* in weights during fine-tuning has a low "intrinsic rank." In simpler terms, the updates we need to make are not complex and don't require changing all the millions of weights in a layer. We can approximate these updates with much smaller matrices.

LoRA freezes the original weight matrices (e.g., in the attention layers) and injects two small, trainable "adapter" matrices, `A` and `B`, alongside them. If the original weight matrix `W` is `d x d`, the adapter matrices might be `d x r` and `r x d`, where `r` (the rank) is a small number like 64.

During the forward pass, the input `x` is processed by both the original weights and the adapters: `y = Wx + BAx`. During backpropagation, only the gradients for `A` and `B` are calculated.

The number of parameters in `A` and `B` is tiny compared to `W`. By focusing the training on just these adapters, we reduce the number of trainable parameters by over 99%, leading to a massive reduction in memory required for gradients and optimizer states.

Putting It Together: The Code for QLoRA Here is the Python code, taken from `code/ch02_rollouts/sample.py`, that sets up our Qwen3 model with QLoRA.

```
import torch
from peft import LoraConfig, get_peft_model
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig

def load_model_qlora(model_name: str = "Qwen/Qwen3-8B-Instruct"):
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.bfloat16,
    )

    model = AutoModelForCausalLM.from_pretrained(
        model_name,
        quantization_config=bnb_config,
        device_map="auto",
        trust_remote_code=True,
    )
    tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)

    lora_config = LoraConfig(
        r=64,
        lora_alpha=16,
        target_modules=["q_proj", "v_proj", "k_proj", "o_proj"],
        lora_dropout=0.05,
        task_type="CAUSAL_LM",
    )

    model = get_peft_model(model, lora_config)
    model.print_trainable_parameters()
    return model, tokenizer
```

Let's break this down:

1. **`BitsAndBytesConfig`**: This object from the `transformers` library configures the quantization.
    - `load_in_4bit=True`: The master switch to enable 4-bit loading.
    - `bnb_4bit_quant_type="nf4"`: Specifies the 4-bit NormalFloat data type, which is optimized for normally distributed weights.
    - `bnb_4bit_compute_dtype=torch.bfloat16`: While the weights are *stored* in 4-bit, the computation (matrix multiplication) is performed in a higher precision format like 16-bit `bfloat16` for stability.
2. **`AutoModelForCausalLM.from_pretrained`**: We load the model as usual, but passing `quantization_config=bnb_confi` tells Hugging Face to apply the 4-bit conversion on the fly. `device_map="auto"` intelligently distributes the model layers across available hardware (in our case, just the single GPU).
3. **`LoraConfig`**: This object from the `peft` (Parameter-Efficient Fine-Tuning) library configures the adapters.
    - `r=64`: The rank of the adapter matrices. A larger `r` means more expressive adapters but also more trainable parameters. 64 is a common, robust choice.
    - `lora_alpha=16`: A scaling factor for the LoRA outputs, analogous to a learning rate.
    - `target_modules`: This is a crucial parameter. It tells PEFT which layers of the transformer to inject the adapters into. The names `["q_proj", "v_proj", "k_proj", "o_proj"]` correspond to the query, value, key, and output projections within the self-attention mechanism. Targeting these is typically the most effective strategy.
4. **`get_peft_model`**: This function takes our quantized base model and the LoRA config and wraps it, creating the final model ready for training.

The final line, `model.print_trainable_parameters()`, produces a vital piece of information:

```
trainable params: 50,031,648 || all params: 7,816,847,360 || trainable%: 0.6438
```

This confirms our setup. We are only training **50 million** parameters, which is just **0.64%** of the total 7.8 billion parameters. This is why QLoRA is so powerful. We get the performance of fine-tuning a huge model while only needing the memory to train a tiny one.

The Generation Code

Now that we have a memory-efficient model loaded, we can write the function to perform the sampling.

```python
def sample_response(
    model,
    tokenizer,
    prompt: str,
    temperature: float = 0.7,
    max_new_tokens: int = 256,
) -> str:
    messages = [{"role": "user", "content": prompt}]
    text = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        add_generation_prompt=True
    )
    inputs = tokenizer(text, return_tensors="pt").to(model.device)

    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_new_tokens=max_new_tokens,
            temperature=temperature,
```

```
            do_sample=True,
            top_p=0.9,
            pad_token_id=tokenizer.eos_token_id,
        )

    response = tokenizer.decode(
        outputs[0][inputs.input_ids.shape[1]:],
        skip_special_tokens=True
    )
    return response

def sample_batch(
    model,
    tokenizer,
    prompt: str,
    n: int = 4,
    temperature: float = 0.7,
) -> list[str]:
    return [sample_response(model, tokenizer, prompt, temperature) for _ in range(n)]
```

This code is straightforward: 1. **`apply_chat_template`**: Instruction-tuned models like Qwen3 expect prompts in a specific format. This utility function correctly formats our simple string prompt into the required structure, e.g., `<|im_start|>user\nWhat is 15 * 23?<|im_end|>\n<|im_start|>assistant\n`. 2. **`torch.no_grad()`**: This is a PyTorch context manager that disables gradient calculation. Since we are only doing inference (generation) here, not training, we don't need gradients. This saves memory and computation. 3. **`model.generate()`**: This is the core generation method from the `transformers` library. We pass it our tokenized inputs and the sampling parameters we discussed earlier (`temperature`, `top_p`). `do_sample=True` is the switch that enables sampling; without it, the model would use greedy decoding. 4. **`tokenizer.decode()`**: The `generate` method returns the token IDs of the *entire* sequence (prompt + response). We slice the output tensor to get only the newly generated tokens (`outputs[0][inputs.input_ids.shape[1]:]`) and decode them back into a human-readable string.

The `sample_batch` function is a simple wrapper that calls `sample_response` `n` times to collect a batch of diverse outputs for a single prompt, which is exactly what we need for the next stages of the RL loop.

Conclusion

In this chapter, we built the foundational component of our reinforcement learning pipeline: the data generator. We moved from the abstract concept of a "rollout" to a concrete, memory-efficient implementation capable of running on a single consumer GPU.

We learned how to control the model's creativity using temperature and top-p sampling to generate the diverse outputs needed for RL. Most importantly, we dissected QLoRA, the key technique that makes this entire project feasible on our hardware. By quantizing the base model to 4-bit and training only a small set of LoRA adapters, we can manipulate a massive 8B parameter model within a 24GB VRAM budget.

We now have a way to produce the raw material for learning. The next step is to figure out how to judge this material. How do we programmatically decide if a generated response is "good" or "bad"? That is the subject of the next chapter, where we will design and implement our reward function.

Chapter 3: Reward Signals

In the world of pre-training, our model's objective was simple: predict the next token. The cross-entropy loss function was our North Star, guiding the model to assign high probabilities to the tokens that actually appeared in the training data. This process is effective for learning grammar, facts, and the general patterns

of human language. However, it has a fundamental limitation: it only measures how well a model imitates its training data. It doesn't directly measure how well it performs a task.

To build a model that reasons, we need to shift our perspective. Instead of asking, "Is this the token I expected to see?" we must ask, "Is this a *good* response?" This question is the domain of **reward signals**. A reward signal is a function that takes a model's output and returns a numerical score indicating its quality. A high score means the response is good; a low score means it's bad. This simple concept is the foundation of Reinforcement Learning (RL) and is how we will teach our model not just to imitate, but to achieve specific goals.

In this chapter, we will explore the different types of reward signals, focusing on the ones most relevant to building a reasoning model. We will distinguish between two primary categories: verifiable rewards and learned rewards. Our focus will be on the former, as they provide a clear, objective, and computationally tractable way to score our model's performance on tasks like mathematics and coding. We will build and analyze the Python code for these reward functions, which will become a cornerstone of our training loop in later chapters.

The Two Flavors of Reward: Verifiable vs. Learned

Reward signals can be broadly categorized into two types, each with its own strengths and weaknesses.

1. **Verifiable Rewards**: These are rewards calculated by a deterministic, objective function. If you run the function on the same input, you will always get the same output. There is no ambiguity. For tasks with a single correct answer, verifiable rewards are ideal. Examples include:
   - **Mathematics**: Does the model's final answer match the true mathematical result?
   - **Code Execution**: Does the code generated by the model run without errors and produce the expected output?
   - **Unit Tests**: Does the generated code pass a suite of pre-defined unit tests?
2. **Learned Rewards (or Reward Models)**: These are rewards generated by another machine learning model. This "reward model" is trained to predict a human's preference for a given response. It's used for tasks that are subjective, stylistic, or too complex for a simple rule-based verifier. Examples include:
   - **Summarization**: Is the summary concise, accurate, and fluent?
   - **Story Writing**: Is the story engaging, coherent, and creative?
   - **Safety**: Is the response helpful, harmless, and unbiased?

While learned rewards are incredibly powerful for aligning models with complex human values, they introduce significant complexity. You have to train, maintain, and serve a second large model just to provide the reward signal. For our goal of building a reasoning model on a single GPU, this is impractical.

Therefore, we will focus exclusively on **verifiable rewards**. They are perfect for our target domains of math and code. They are computationally cheap, objective, and directly measure the capabilities we want to improve. They provide a clean, unambiguous signal that we can use to guide our model's learning process.

Building a Verifiable Reward Function for Mathematics

Let's start with a common reasoning task: solving a math word problem. The final output we care about is a single numerical answer. Our reward function, therefore, needs to do two things: 1. Parse the model's potentially long, free-form response to find the final answer. 2. Compare that extracted answer to the known ground truth.

This is more complex than a simple string comparison. A model might say "The answer is 42", "So, the final result is 42", or simply end its reasoning chain with " = 42". Our function needs to be robust to these variations.

Let's examine the implementation from `ch03_rewards/verifier.py`.

Extracting the Answer First, we need a robust way to find the numerical answer in a block of text. Regular expressions are the perfect tool for this job. We can define a list of patterns to try in order of

precision.

```python
import re

def extract_answer(response: str) -> str | None:
    patterns = [
        r"(?:answer|result|equals?|is)[:\s]+(-?\d+(?:\.\d+)?)",
        r"=\s*(-?\d+(?:\.\d+)?)\s*$",
        r"\\boxed\{(-?\d+(?:\.\d+)?)}",
        r"(\d+(?:\.\d+)?)\s*$",
    ]
    for pattern in patterns:
        match = re.search(pattern, response, re.IGNORECASE | re.MULTILINE)
        if match:
            return match.group(1)
    return None
```

This function iterates through a list of regex patterns. The patterns are ordered from most to least specific. - `(?:answer|...)[...` is quite specific, looking for keywords. - `=\s*(-?\d...` is a bit more general, looking for an equals sign at the end of a line. - `\\boxed{...}` is specific to a common format in math datasets. - Finally, grabbing the last number in the string is a fallback.

By trying them in order, we increase the chances of finding the intended answer while minimizing the risk of accidentally picking up a number from the model's intermediate reasoning steps.

Scoring the Response Once we have the extracted answer, we can write the main reward function. It should handle three cases: 1. The extracted answer is correct. **High reward.** 2. The extracted answer is incorrect. **Negative reward (penalty).** 3. No answer could be extracted. **Stronger negative reward.**

This scoring system incentivizes the model to first produce *an* answer in a parsable format, and then to produce the *correct* answer.

```python
def math_reward(response: str, ground_truth: str) -> float:
    answer = extract_answer(response)
    if answer is None:
        return -1.0

    try:
        if abs(float(answer) - float(ground_truth)) < 1e-6:
            return 1.0
    except ValueError:
        pass

    if answer.strip() == ground_truth.strip():
        return 1.0

    return -0.5
```

Let's break down the logic: - **Score `1.0` (Correct)**: The highest reward is given if the extracted answer matches the ground truth. We use `abs(float(a) - float(b)) < 1e-6` to avoid floating-point precision issues. An exact string match is also checked. - **Score `-0.5` (Incorrect)**: A negative reward is given if an answer is found, but it's wrong. This teaches the model that a wrong answer is undesirable, but not as bad as failing to produce an answer at all. - **Score `-1.0` (No Answer)**: The lowest reward is given if `extract_answer` returns `None`. This creates a strong incentive for the model to format its final answer in one of the ways our parser expects.

This simple, verifiable function gives us a powerful tool. We don't need to understand *how* the model arrived

at its answer; we just need to check if the final result is correct. This is known as **outcome-based reward**, and it's a highly effective starting point.

A Verifiable Reward for Code Generation

Next, let's consider the task of generating code. How do you objectively measure the quality of a code snippet? The most direct method is to execute it and check its output.

Our reward function for code will be a bit more involved. It needs to: 1. Take the generated code as a string. 2. Execute it in a secure, isolated environment. 3. Capture the output (stdout). 4. Compare the captured output to the expected output. 5. Handle errors, such as syntax errors or timeouts.

The `subprocess` module in Python's standard library is perfect for this. We can run the generated code in a separate Python process, which isolates it from our main training script and allows us to set a timeout.

```
import subprocess

def code_reward(code: str, expected_output: str, timeout: float = 5.0) -> float:
    try:
        result = subprocess.run(
            ["python", "-c", code],
            capture_output=True,
            text=True,
            timeout=timeout,
        )

        if result.returncode != 0:
            return -1.0

        if result.stdout.strip() == expected_output.strip():
            return 1.0
        else:
            return -0.5

    except subprocess.TimeoutExpired:
        return -1.0

    except Exception:
        return -1.0
```

This function mirrors the structure of our math reward function, but the verification step is execution instead of numerical comparison. - **Score `1.0` (Correct Execution)**: The code runs successfully and its stripped standard output exactly matches the `expected_output`. - **Score `-0.5` (Wrong Output)**: The code runs without errors but produces the wrong output. This is better than code that crashes. - **Score `-1.0` (Error or Timeout)**: The lowest score is given for several failure modes: - The process returns a non-zero exit code, indicating a runtime error. - The process times out, suggesting an infinite loop or highly inefficient code. - An exception is raised during the subprocess management itself.

This reward function directly measures the functional correctness of the generated code. It's a powerful, objective signal for teaching a model to write working programs.

Rewarding the Process: Encouraging Think Tokens

So far, our rewards have been purely outcome-based. We only care about the final answer. But what if we also want to encourage the *process* of reasoning? As discussed in `AGENTS.md`, one of our key goals is to teach the model to use `<think>...</think>` tags to lay out its reasoning steps before giving the final answer.

We can modify our reward function to give a bonus for using these tags correctly. A good reasoning process (indicated by the presence of think tags) that leads to a correct answer is better than a correct answer that appears by magic.

Let's create a higher-order function that wraps a base reward function (like `math_reward`) and adds a bonus for thinking.

```
from typing import Callable

def reward_with_thinking(
    response: str,
    ground_truth: str,
    base_reward_fn: Callable[[str, str], float] = math_reward,
) -> float:
    has_think = "<think>" in response and "</think>" in response
    base_reward = base_reward_fn(response, ground_truth)

    if base_reward > 0 and has_think:
        return 1.0

    elif base_reward > 0:
        return 0.5

    elif has_think:
        return -0.2

    return -0.5
```

This function creates a more nuanced reward landscape: - **Score `1.0` (Correct + Thinking)**: The highest reward. The model showed its work and got the right answer. - **Score `0.5` (Correct, No Thinking)**: A positive reward, but lower than the best case. We still want correct answers, even if the model doesn't reason explicitly. - **Score `-0.2` (Incorrect + Thinking)**: A small penalty. The model tried to reason but failed. This is preferable to not trying at all. - **Score `-0.5` (Incorrect, No Thinking)**: The standard penalty for a wrong answer without any reasoning.

By using this wrapper, we create a gentle pressure on the model. During RL, the model will discover that responses containing `<think>` tags, on average, lead to higher rewards. This encourages the very behavior we want to see. This is a simple form of **process-based reward**, where we are rewarding not just the outcome, but the steps taken to get there.

Conclusion

In this chapter, we have moved beyond the simple world of next-token prediction and into the more sophisticated realm of reward signals. We've established the critical distinction between subjective, complex learned rewards and the objective, efficient verifiable rewards that we will use to train our reasoning model.

We implemented two key verifiable reward functions from scratch: 1. `math_reward`, which intelligently parses a model's textual output to find and verify a numerical answer. 2. `code_reward`, which safely executes generated code in a sandbox and checks its output for correctness.

Finally, we explored how to go beyond simple outcome-based rewards by creating a `reward_with_thinking` wrapper. This function encourages the model to generate explicit reasoning traces, rewarding the process as well as the result.

These functions are not just theoretical examples; they are the practical tools we will import and use directly in our RL training loop in the upcoming chapters. They will provide the scalar feedback signal that drives the entire learning process, pushing our model from a generic text generator towards a focused and capable

reasoning engine. With our data generation (Chapter 2) and reward signals now in place, we are ready to tackle the core algorithm that will put them to use: the policy gradient.

Chapter 4: Policy Gradients - The REINFORCE Algorithm

In the previous chapters, we shifted our perspective from minimizing loss to maximizing rewards. We now have a way to generate responses from our model (rollouts) and a way to score them (reward signals). The final piece of the puzzle is figuring out how to use that scalar reward signal to improve the model. How do we adjust the millions of parameters in our language model to make it more likely to produce high-reward outputs?

This is where policy gradient methods come in. They are the bedrock of modern reinforcement learning for large language models. The core idea is simple: if a sequence of tokens leads to a high reward, we want to increase the probability of generating that sequence. If it leads to a low reward, we want to decrease its probability.

This chapter will derive the fundamental policy gradient algorithm, REINFORCE, from first principles. We'll confront the central challenge of RL—differentiating through a sampling process—and see how a clever mathematical trick allows us to compute a usable gradient.

The Objective: Maximizing Expected Reward

Our language model is a **policy**—a probability distribution over output sequences given an input prompt. In code terms:

```
def policy(model, prompt_ids):
    logits = model(prompt_ids).logits
    probs = F.softmax(logits, dim=-1)
    return probs
```

We also have a reward function that assigns a scalar score to a complete output:

```
def reward_fn(response: str) -> float:
    return 1.0 if is_correct(response) else -0.5
```

Our goal is to find model parameters that maximize the expected reward. If we could sample many responses and average their rewards, that's what we want to push upward:

```
responses = [model.generate(prompt) for _ in range(N)]
rewards = [reward_fn(r) for r in responses]
expected_reward = sum(rewards) / N
```

To maximize this using gradient descent, we need a gradient. But here's the problem.

The Problem: Differentiating Through Sampling

If we try to compute gradients naively, we hit a wall. The sampling operation (`torch.multinomial`, `top_k`, etc.) is non-differentiable. The computation graph is broken:

```
probs = F.softmax(logits, dim=-1)
next_token = torch.multinomial(probs, num_samples=1)
```

There's no gradient flowing through `torch.multinomial`. We can compute the reward after sampling, but we can't backpropagate through the sampling step to update our model.

We need a trick to compute gradients without differentiating through sampling.

The Solution: The Log-Derivative Trick (REINFORCE)

The insight is that we can rewrite the gradient in a form that doesn't require differentiating through sampling. Instead of trying to backprop through the sampled tokens, we:

1. Sample a response (no gradients here)
2. Compute the log-probability of that response (this IS differentiable)
3. Multiply by the reward and backprop

The key identity: if we want to increase the probability of high-reward sequences, we can use the log-probability as a proxy:

```
def compute_log_prob(model, prompt_ids, response_ids):
    full_ids = torch.cat([prompt_ids, response_ids], dim=-1)
    logits = model(full_ids).logits

    log_probs = F.log_softmax(logits[:, :-1, :], dim=-1)
    token_log_probs = log_probs.gather(dim=-1, index=response_ids.unsqueeze(-1)).squeeze(-1)

    return token_log_probs.sum(dim=-1)
```

The REINFORCE update becomes:

```
response_ids = model.generate(prompt_ids)
reward = reward_fn(decode(response_ids))

log_prob = compute_log_prob(model, prompt_ids, response_ids)
loss = -reward * log_prob

loss.backward()
optimizer.step()
```

Why does this work? The gradient of `-reward * log_prob` points in the direction that increases the log-probability of the response. If reward is positive, we increase that probability. If reward is negative, we decrease it.

This is the **policy gradient theorem** in code form. The mathematical derivation involves a trick where we multiply and divide by the policy probability, but the implementation is what matters:

1. Sample a response (detached from gradients)
2. Compute its log-probability (attached to gradients)
3. Multiply by reward and backprop

The log-probability gradient points in the direction that increases the probability of that specific sequence. The reward scales how big a step we take:

- If reward is high and positive, we take a large step to make that response more probable.
- If reward is low or negative, we step the opposite direction, making that response less probable.

The REINFORCE Algorithm: A Basic Implementation

Let's translate this into a PyTorch training loop.

```
import torch
import torch.nn.functional as F
from torch.distributions import Categorical


model.eval()
with torch.no_grad():
    generated_tokens, log_probs = model.generate_with_log_probs(
        prompt_tokens,
        max_length=100,
        temperature=0.7
```

```
    )

reward = reward_fn(generated_tokens)

sequence_log_prob = torch.stack(log_probs).sum()

loss = - (reward * sequence_log_prob)

model.train()
optimizer.zero_grad()
loss.backward()
optimizer.step()

print(f"Reward: {reward:.2f}, Loss: {loss.item():.2f}")
```

Why a Negative Loss? This is a frequent point of confusion. Standard deep learning minimizes a loss function (like cross-entropy). Here, we want to *maximize* an objective function (the expected reward). Gradient descent finds minima, while gradient ascent finds maxima.

To use a standard optimizer, we simply negate. Minimizing `-expected_reward` is equivalent to maximizing `expected_reward`:

```
loss = -reward * log_prob
loss.backward()
optimizer.step()
```

When the optimizer takes its step, it subtracts the gradient of the loss from the weights. Since we negated, this becomes adding the policy gradient—exactly the gradient ascent step we wanted.

The Problem with High Variance

The REINFORCE algorithm works, but it has a major problem: high variance.

The gradient can fluctuate wildly between steps depending on which response we happen to sample. If we get an unlucky sample with a very high or very low reward, it can lead to a huge, potentially destabilizing gradient update.

Imagine a scenario where all possible rewards are positive, ranging from 10 to 20. Even the "worst" possible output gets a reward of 10. According to our current formula, this output would still have its probability *increased*, just not by as much as an output with reward 20.

Intuitively, this feels wrong. We shouldn't be encouraging outputs that are merely "less bad." We should only encourage outputs that are *better than average*.

Variance Reduction: Introducing a Baseline

The fix: subtract a **baseline** from the reward. The baseline is typically the average reward:

```
rewards = torch.tensor([reward_fn(r) for r in responses])
baseline = rewards.mean()
advantages = rewards - baseline
```

Now instead of using raw rewards, we use **advantages**:

```
for response, advantage in zip(responses, advantages):
    log_prob = compute_log_prob(model, prompt, response)
    loss = -advantage * log_prob
    loss.backward()
```

Why is this valid? Mathematically, subtracting any constant from the reward doesn't change the expected gradient direction—it just reduces variance. The intuition:

- If `reward > baseline`: advantage is positive, we increase probability
- If `reward < baseline`: advantage is negative, we decrease probability

This matches our intuition. We now only reinforce actions that are better than average.

What Makes a Good Baseline? The ideal baseline is the expected reward. Here are three ways to estimate it:

1. **Running average:** Keep a moving average of rewards seen so far. Simple but slow to adapt.
2. **Batch average:** Average reward over the current batch. Simple and effective:

   ```
   baseline = rewards.mean()
   ```

3. **Critic network:** Train a separate neural network to predict expected reward. Powerful but adds complexity—we avoid this to stay on a single GPU.

For our purposes, the batch average is perfect. It's the core idea behind GRPO in the next chapter.

Here is our updated REINFORCE implementation with a baseline.

```
import torch


rewards = []
for resp in responses:
    rewards.append(reward_fn(resp))

rewards = torch.tensor(rewards)

baseline = rewards.mean()

advantages = rewards - baseline

total_loss = 0
for log_prob, advantage in zip(log_probs, advantages):
    total_loss += -log_prob.sum() * advantage.detach()

loss = total_loss / len(responses)

optimizer.zero_grad()
loss.backward()
optimizer.step()

print(f"Mean Reward: {rewards.mean():.2f}, Mean Advantage: {advantages.mean():.2f}")
```

With this, we have derived the fundamental concepts of policy gradient optimization from scratch. We understand the objective, the challenge of differentiation, the log-derivative trick solution, and the critical importance of variance reduction using a baseline.

We are now ready to move from the general REINFORCE algorithm to a specific, state-of-the-art implementation tailored for LLMs: Group Relative Policy Optimization (GRPO).

Chapter 5: GRPO - Critic-Free Reinforcement Learning

In the previous chapter, we laid the mathematical groundwork for policy gradients. We saw how to adjust a model's parameters to make high-reward outputs more likely. The core idea was REINFORCE, a simple algorithm with a fatal flaw: high variance. The learning signal was noisy, swinging wildly with the luck of the draw on each generated response.

To tame this variance, we introduced the concept of a **baseline**. By subtracting an estimate of the average reward—the value function `V(s)`—from the actual reward `R(s, a)`, we could determine if an action was better or worse than average. This "advantage" `A(s, a) = R(s, a) - V(s)` provided a much more stable learning signal.

This is where state-of-the-art algorithms like Proximal Policy Optimization (PPO) enter the picture. PPO has become the workhorse of Reinforcement Learning from Human Feedback (RLHF), powering models like ChatGPT. At its heart, PPO uses a second, separate neural network called a **critic** whose sole job is to learn the value function `V(s)`.

For every training step, the policy (the "actor") generates a response, and the critic estimates the value of the prompt's state. This works wonderfully, but it comes at a steep price.

The Critic's Burden

Training a critic network alongside a policy network is standard practice in deep reinforcement learning. However, in the resource-constrained world of fine-tuning large language models on a single GPU, this approach presents significant, often insurmountable, challenges.

1. Parameter Overhead: A critic network is not a small, simple model. To accurately predict the value function of a complex policy, the critic often needs to be of a similar size and architecture. If our policy is an 8-billion parameter language model, our critic might need to be an 8-billion parameter model too.

On our target hardware—a single 24GB NVIDIA 3090—we are already pushing the limits to fit one 8B model using QLoRA. Loading a second model of that size is simply out of the question. It would double the memory footprint, requiring at least 48GB of VRAM, a luxury we don't have.

2. Training Complexity and Instability: Training two large neural networks in tandem is a delicate dance. - You need two separate optimizers and two learning rate schedules. - The actor and critic can learn at different paces. If the critic's value estimates lag behind or are inaccurate, it provides a poor-quality, or even misleading, advantage signal to the actor. This can destabilize the entire training process. - The whole system becomes more complex, with more hyperparameters to tune and more potential points of failure.

This leads us to a critical question, born from our hardware constraints: **Can we get the variance-reduction benefits of a baseline without paying the memory and complexity costs of training a critic?**

From Critic to Group: The GRPO Insight

Let's step back and reconsider the baseline's purpose from first principles. The baseline `b(s)` is meant to be an estimate of the expected reward from a given state `s`.

```
b(s)  E[R(s, a) | s, a ~ _]
```

The critic learns a parameterized function `V_ (s)` that approximates this expectation. But what is the simplest, most direct way to estimate an expectation? **Monte Carlo sampling.**

This is the beautiful, simple idea behind Group Relative Policy Optimization (GRPO), a technique highlighted by DeepSeek-AI. Instead of training a separate network to *predict* the average reward, we can *calculate* it directly from a small batch of samples.

Here's the insight:

1. For a single prompt (our "state" s), use the current policy _ to generate not one, but a **group** of G different responses (a_1, a_2, ..., a_G).
2. Calculate the reward for each response using our reward function: r_1, r_2, ..., r_G.
3. The average of these rewards is a direct, unbiased Monte Carlo estimate of the expected reward for that prompt:

   b(s) = (1/G) * Σ_{i=1 to G} r_i E[R(s, a)]

We've just created a baseline on-the-fly, using only the policy model we already have. There is no critic, no second network, no extra parameters. The advantage for a specific response a_i within the group is now simply its reward relative to the group's average reward.

Advantage(a_i) = r_i - (1/G) * Σ_{j=1 to G} r_j

This is why it's called **Group Relative** Policy Optimization. We judge each response not against a learned global value, but against its peers in a locally sampled group.

Deriving the GRPO Objective Step-by-Step

Let's formalize this intuition. We start with the standard policy gradient theorem, which aims to maximize the expected reward J( ):

J( ) = E[log _(a|s) * Q(s, a)]

Here, Q(s, a) is the quality function, representing the total expected reward after taking action a in state s. In the context of single-turn LLM fine-tuning, a "trajectory" consists of just one action—generating the entire response. Thus, Q(s, a) simplifies to the immediate reward R(s, a).

J( ) = E[log _(a|s) * R(s, a)]

This is the REINFORCE algorithm, which we know has high variance. We can introduce a baseline b(s) that depends only on the state s (the prompt) without changing the expectation. This is because the baseline term sums to zero during the derivation.

J( ) = E[log _(a|s) * (R(s, a) - b(s))]

The term (R(s, a) - b(s)) is the **advantage**. - In Actor-Critic methods like PPO, we train a critic V_ (s) and set our baseline b(s) = V_ (s). - In GRPO, we define the baseline b(s) as the empirical mean reward of a group of G samples (a_1, ..., a_G) drawn from our policy _ for the given state s.

b(s) = (1/G) * Σ_{j=1 to G} R(s, a_j) where a_j ~ _(·|s)

The advantage estimate for a specific sample a_i from that group is its own reward minus this empirical baseline:

Â(s, a_i) = R(s, a_i) - (1/G) * Σ_{j=1 to G} R(s, a_j)

To get our final objective, we want to maximize the advantage-weighted log-probabilities of the actions we took. For a single prompt s, our loss function (which we want to *minimize*) is the negative of this objective, averaged over the group:

L_GRPO( ) = - (1/G) * Σ_{i=1 to G} [Â(s, a_i) * log _(a_i|s)]

Substituting the definition of our advantage estimate Â, we get the full GRPO loss for a single prompt:

L_GRPO( ) = - (1/G) * Σ_{i=1 to G} [(R(s, a_i) - (1/G)Σ_j R(s, a_j)) * log _(a_i|s)]

This loss function achieves our goal: it increases the probability of responses that are better than the group average and decreases the probability of responses that are worse. All without a critic.

The GRPO Algorithm in Practice

Let's translate the math into a practical algorithm. Here is the pseudocode for a single training step on one prompt.

```
function GRPOStep(prompt, policy_model):
  // 1. Generate a group of G responses from the current policy
  responses = []
  for i from 1 to G:
    response = policy_model.generate(prompt)
    responses.append(response)

  // 2. Calculate the reward for each response
  rewards = []
  for response in responses:
    reward = reward_function(response)
    rewards.append(reward)

  // 3. Compute the baseline (group mean reward)
  baseline = mean(rewards)

  // 4. Compute the advantage for each response
  advantages = []
  for reward in rewards:
    advantage = reward - baseline
    advantages.append(advantage)

  // 5. Calculate the total loss for the group
  total_loss = 0
  for response, advantage in zip(responses, advantages):
    // Get the log-probability of generating this response
    log_prob = policy_model.log_probability(prompt, response)

    // The loss is the negative advantage-weighted log-prob
    response_loss = -advantage * log_prob
    total_loss += response_loss

  // 6. Average the loss over the group and update
  average_loss = total_loss / G
  average_loss.backward() // Backpropagate
  optimizer.step()        // Update policy_model weights
```

This procedure is elegant in its simplicity. We're trading the memory cost of a critic for the computational cost of generating `G` samples for each prompt. On a memory-constrained system like ours, this is a fantastic trade-off.

Full Implementation Walkthrough

Now, let's examine a complete, runnable implementation of a GRPO training step. The code below is a practical realization of the pseudocode above, using PyTorch and the Hugging Face `transformers` library.

The main logic is encapsulated in the `grpo_step` function. It handles everything from generation to back-propagation for a single prompt.

The Code: `grpo.py`

```python
"""GRPO: Group Relative Policy Optimization."""
import torch
import torch.nn.functional as F
from typing import Callable

def compute_log_probs(model, tokenizer, prompt: str, response: str) -> torch.Tensor:
    """
    Computes the sum of log-probabilities for a generated response given a prompt.
    This represents log  (a|s).
    """
    full_text = prompt + response
    inputs = tokenizer(full_text, return_tensors="pt").to(model.device)
    prompt_len = len(tokenizer(prompt, return_tensors="pt").input_ids[0])

    with torch.no_grad():
        outputs = model(**inputs)

    logits = outputs.logits[0, prompt_len-1:-1]
    target_ids = inputs.input_ids[0, prompt_len:]

    log_probs = F.log_softmax(logits, dim=-1)

    token_log_probs = log_probs.gather(1, target_ids.unsqueeze(1)).squeeze(1)

    return token_log_probs.sum()

def grpo_step(
    model,
    tokenizer,
    prompt: str,
    ground_truth: str,
    reward_fn: Callable[[str, str], float],
    G: int = 4,
    temperature: float = 0.7,
    max_new_tokens: int = 256,
) -> tuple[float, float, list[str]]:
    """
    Performs a single GRPO optimization step for one prompt.
    """
    model.train()

    messages = [{"role": "user", "content": prompt}]
    formatted_prompt = tokenizer.apply_chat_template(
        messages, tokenize=False, add_generation_prompt=True
    )
    inputs = tokenizer(formatted_prompt, return_tensors="pt").to(model.device)

    responses = []
    for _ in range(G):
        with torch.no_grad():
            output_ids = model.generate(
                **inputs,
                max_new_tokens=max_new_tokens,
                temperature=temperature,
```

```
                do_sample=True,
                top_p=0.9,
                pad_token_id=tokenizer.eos_token_id,
            )
        response = tokenizer.decode(
            output_ids[0][inputs.input_ids.shape[1]:],
            skip_special_tokens=True,
        )
        responses.append(response)

    rewards = torch.tensor(
        [reward_fn(r, ground_truth) for r in responses],
        dtype=torch.float32,
        device=model.device,
    )

    baseline = rewards.mean()
    advantages = rewards - baseline

    loss = torch.tensor(0.0, device=model.device, requires_grad=True)
    for response, adv in zip(responses, advantages):
        if adv.abs() < 1e-8:
            continue

        log_prob = compute_log_probs(model, tokenizer, formatted_prompt, response)

        loss = loss - adv * log_prob / G

    if loss.requires_grad:
        loss.backward()

    return loss.item(), rewards.mean().item(), responses

if __name__ == "__main__":
    print("GRPO module loaded successfully")
    print("This file contains the core GRPO implementation.")
```

Analysis of the Implementation

- `compute_log_probs`: This is a helper function that does the heavy lifting of calculating $\log \pi_\theta(a|s)$. It's a standard forward pass followed by a `gather` operation to select the probabilities corresponding to the generated tokens. We use `torch.no_grad()` because we don't need gradients for this part; the policy gradient theorem tells us to treat the log-probabilities as constants with respect to the advantage.
- **Generation (`Step 1`)**: The `for` loop generates `G` responses. It's critical that `do_sample=True` is used. If we used greedy decoding (`do_sample=False`), the model would produce the same output every time, resulting in zero advantage and no learning signal. Temperature and top-p sampling introduce the necessary exploration.
- **Baseline and Advantage (`Step 3 & 4`)**: These are the two most important lines: `baseline = rewards.mean()` and `advantages = rewards - baseline`. They replace the entire machinery of a critic network with two simple, efficient tensor operations.
- **Loss Calculation (`Step 5`)**: The final loop computes the objective function. We accumulate the loss for each response in the group, weighted by its advantage. Dividing by `G` ensures that the magnitude of the gradient doesn't explode as we increase the group size.
- **Backpropagation (`Step 6`)**: A single call to `loss.backward()` calculates the gradients for

all the trainable parameters in our model (in our case, the LoRA adapters). The subsequent `optimizer.step()` (not shown here, but part of the outer training loop) will apply these gradients.

Benefits and Trade-offs on a Single GPU

GRPO is not a free lunch; it's an engineering trade-off. But for our specific use case, it's a highly advantageous one.

Benefits:

1. **Drastically Reduced VRAM Usage**: This is the killer feature for us. By eliminating the critic, we cut the model parameter memory requirement in half. This is what makes fine-tuning an 8B parameter model on a 24GB GPU not just possible, but comfortable, leaving room for larger batch sizes.
2. **Implementation Simplicity**: The code is straightforward. We don't need to define a second model class, a second optimizer, or a second learning rate scheduler. This reduces the chance of bugs and makes the training code easier to manage and debug.
3. **Improved Stability**: The baseline is calculated directly from the current policy's outputs at each step. It's a fresh, up-to-date estimate. Critic networks can often lag behind the policy, providing stale value estimates that can harm performance. GRPO avoids this issue entirely.

Trade-offs:

1. **Higher Variance Baseline**: A sample mean from a small group (e.g., `G=4` or `G=8`) is a noisy estimate of the true expected reward. It will have higher variance than the output of a well-converged critic network. This can slow down learning, potentially requiring more training steps to reach the same performance.
2. **Increased Generation Cost**: For every prompt in our batch, we perform `G` separate `model.generate()` calls. Since autoregressive generation is sequential and can be slow, this can become a computational bottleneck. We are trading memory for computation. However, `G` forward passes through the generator is often still faster and more memory-efficient than one forward *and* one backward pass through a large critic network.
3. **A New Hyperparameter, `G`**: We've eliminated the critic's learning rate, but introduced a new hyperparameter: the group size `G`.
   - If `G` is too small (e.g., 2), the baseline will be very noisy.
   - If `G` is too large (e.g., 32), the training step will be very slow.
   - Finding the right balance (typically between 4 and 16) is key to making GRPO work effectively.

Conclusion

GRPO is a perfect example of an algorithm tailored to specific constraints. It elegantly sidesteps the memory bottleneck of traditional actor-critic methods, making it an ideal choice for RLHF on consumer-grade hardware. It replaces the complexity of a second neural network with the computational cost of repeated generation, a trade-off that aligns perfectly with our single-GPU setup.

By understanding GRPO, we've unlocked the ability to perform powerful reinforcement learning on an 8-billion parameter model without needing an industrial-scale hardware budget. This critic-free approach is the engine we will use in the coming chapters to teach our model the complex art of reasoning.

Chapter 6: Think Tokens

In the preceding chapters, we laid the groundwork for Reinforcement Learning from Human Feedback (RLHF). We transitioned from a supervised learning paradigm, defined by cross-entropy loss, to a reward-based system. We built the machinery to generate diverse responses from our policy model (Chapter 2), designed reward functions to score those responses (Chapter 3), and implemented a powerful, critic-free RL algorithm, GRPO, to steer the model towards better outputs (Chapter 5).

Our system works. It can learn to prefer responses that lead to a high reward. But so far, the reward has been based solely on the final outcome. For a math problem, the reward function checks if the final number

is correct. For a coding challenge, it checks if the code passes unit tests. This is a crucial and effective signal, but it has a significant limitation: it's a black box. If the model fails, we know it failed, but we don't know *why*. Was it a small calculation error? A fundamental misunderstanding of the question?

To build a true reasoning model, we need to go beyond outcome-based rewards and start supervising the *process* of reasoning itself. We need to encourage the model to "show its work." This is where **think tokens** come in. They are a powerful mechanism for making the model's internal reasoning process explicit, allowing us to inspect, evaluate, and, most importantly, reward it.

The `<think>` Block: Exposing the Chain of Thought

Think tokens are special tokens that we add to our model's vocabulary. We use them to create a specific structure in the model's output: the `<think>...</think>` block. This block contains the model's "chain of thought" or "reasoning trace"—a step-by-step process of how it arrives at an answer.

The key idea is that this reasoning is for the model's and the training process's benefit, not necessarily for the end-user. During inference, this block is typically stripped from the final output, so the user sees only the clean, final answer.

Let's look at a few examples.

1. A Math Problem

Consider the prompt from `AGENTS.md`: `What is 127 * 843?`. A direct answer might be correct or incorrect. But an answer with a think block reveals the methodology:

```
<|im_start|>user
What is 127 * 843?<|im_end|>
<|im_start|>assistant
<think>
I need to calculate 127 multiplied by 843.
This is a large multiplication, so I should break it down.
I can use the distributive property: 127 * (800 + 40 + 3).
1. 127 * 800 = 101600
2. 127 * 40 = 5080
3. 127 * 3 = 381
Now, I sum these parts:
101600 + 5080 + 381
106680 + 381 = 107061
Let me double-check.
127 * 843 = 127 * (840 + 3) = 106680 + 381 = 107061. The calculation seems correct.
</think>
107061<|im_end|>
```

This trace is incredibly valuable. It decomposes the problem, executes each step, and even includes a self-correction/verification step.

2. A Coding Problem

Prompt: `Write a Python function to check if a number is prime.`

```
<|im_start|>user
Write a Python function to check if a number is prime.<|im_end|>
<|im_start|>assistant
<think>
The user wants a Python function `is_prime(n)`.
A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself
First, I need to handle edge cases.
- Numbers less than or equal to 1 are not prime.
```

```
- 2 is the only even prime number.
For other numbers, I need to check for divisors.
I can iterate from 2 up to the square root of the number. If I find any divisor in this range, the numb
The square root optimization is important for efficiency. For example, if `n` has a divisor `d` larger
So the algorithm is:
1. Define function `is_prime(n)`.
2. If n <= 1, return False.
3. If n == 2, return True.
4. If n % 2 == 0, return False (for all other even numbers).
5. Iterate from 3 up to `int(sqrt(n)) + 1`, with a step of 2 (to check only odd divisors).
6. If `n` is divisible by any number in the loop, return `False`.
7. If the loop completes without finding a divisor, return `True`.
This seems like a solid plan. I will now write the code.
</think>
```python
import math

def is_prime(n):
    """Checks if a number is prime."""
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if n % i == 0:
            return False
    return True
```<|im_end|>
```

Here, the think block serves as a detailed algorithm design document, explaining the logic and optimizations before a single line of code is written.

The Benefits of Explicit Reasoning

1. **Problem Decomposition:** The most significant advantage is that it forces the model to break down multi-step problems into a sequence of simpler steps. This is a fundamental strategy in human problem-solving. A model that can do this is less likely to be overwhelmed by complexity.
2. **Granular Credit Assignment:** With an explicit reasoning trace, our reward function can become much more sophisticated. Instead of just a binary `correct/incorrect` reward for the final answer, we can reward intermediate steps. If the model correctly performs the first three steps of a calculation but fails on the fourth, we can still give it partial credit. This provides a much richer, more fine-grained learning signal.
3. **Interpretability and Debugging:** When a model produces a wrong answer, the think block is the first place we look. It allows us to pinpoint the exact source of the error. Did it misunderstand the prompt? Did it make a logical leap? A calculation error? This insight is invaluable for understanding model failures and designing better training data or reward schemes.
4. **Self-Correction:** As seen in the math example, a model can use its own reasoning trace to review and correct its work. By externalizing its thought process, it can identify inconsistencies or errors before producing the final answer, leading to more robust and reliable performance.

Training a Model to Think

Knowing the format is one thing; teaching a billion-parameter model to use it effectively is another. You can't just add `<think>` to the instruction prompt and hope for the best. The model must learn *when* to use think tokens and *how* to produce a useful reasoning trace. This is where we leverage the power of our RL training loop.

The core idea is to modify our reward function to account for the reasoning process, not just the final outcome. We need to create a reward signal that incentivizes good reasoning and penalizes poor or unnecessary reasoning.

Let's define a new reward function, `reward_with_thinking`. This function will calculate a total reward based on three components:

1. **`R_outcome` (Outcome Reward):** This is the same reward we've used before. It measures the correctness of the final answer. For a math problem, it's 1.0 if the answer is correct, and a negative value if it's incorrect.
2. **`R_process` (Process Reward):** This is a new component that scores the quality of the reasoning inside the `<think>` block. A good process is logical, correct, and leads toward the right answer.
3. **`C_think` (Thinking Cost):** This is a small penalty applied whenever the model uses the `<think>` block. Its purpose is to discourage the model from "thinking" about trivial problems. We want the model to be efficient. For `1 + 1`, it should just answer `2`. For `127 * 843`, it should think. The cost encourages this discrimination.

The final reward is a weighted sum of these components:

```
R_total = R_outcome + w_process * R_process - C_think
```

Here, `w_process` is a weight that allows us to control how much we value the process relative to the outcome.

Designing the Process Reward (`R_process`) Scoring a reasoning process is challenging and an active area of research. For our single-GPU project, we need a simple, automatable heuristic. We can't rely on human evaluators for every single training sample.

For a math problem, a simple heuristic for `R_process` could be:

- **Positive Reward:**
  - Give a small positive reward for each arithmetically correct step in the think block. For example, in `127 * 800 = 101600`, we can verify this calculation is correct and assign a small reward.
  - Reward the length and detail of the trace, but only if the final answer is correct. This encourages thoroughness without rewarding verbose nonsense.
- **Negative Reward (Penalty):**
  - Apply a penalty for any identifiable error in the reasoning trace. If a step is `127 * 3 = 380`, our verifier can flag this as incorrect and apply a penalty.
  - Apply a penalty if the think block is empty or contains trivial content (e.g., `<think>thinking...</think>`).

The `reward_with_thinking` Function in Practice Let's sketch out what this function might look like in Python. This implementation will focus on math problems, but the principle is generalizable.

```python
import re

def extract_think_block(response: str) -> str | None:
    """Extracts content from the <think>...</think> block."""
    match = re.search(r'<think>(.*?)</think>', response, re.DOTALL)
    if match:
        return match.group(1).strip()
    return None

def extract_final_answer(response: str) -> str | None:
```

```
    """Extracts the final answer appearing after the think block."""
    parts = response.split('</think>')
    if len(parts) > 1:
        numbers = re.findall(r'[-+]?\d*\.\d+|\d+', parts[-1])
        if numbers:
            return numbers[-1]
    return None

def verify_math_step(step: str) -> bool:
    """A mock function to verify a single arithmetic step."""
    try:
        if "=" in step:
            expr, result = step.split('=', 1)
            return eval(expr.strip()) == float(result.strip())
    except Exception:
        return False
    return False

def reward_with_thinking(
    response: str,
    ground_truth_answer: float,
    w_process: float = 0.5,
    c_think: float = 0.05
) -> float:
    """
    Calculates a reward considering both the outcome and the reasoning process.
    """
    think_content = extract_think_block(response)
    final_answer_str = extract_final_answer(response)

    r_outcome = -0.5
    if final_answer_str:
        try:
            if float(final_answer_str) == ground_truth_answer:
                r_outcome = 1.0
        except ValueError:
            r_outcome = -1.0

    r_process = 0.0
    if think_content:
        r_process -= c_think

        lines = think_content.split('\n')
        correct_steps = 0
        for line in lines:
            if verify_math_step(line):
                correct_steps += 1

        if r_outcome > 0:
            r_process += (correct_steps / len(lines)) if lines else 0.0
        else:
            r_process -= (1.0 - (correct_steps / len(lines))) if lines else 0.0

    if r_process > 0:
```

```
        total_reward = r_outcome + w_process * r_process
    else:
        total_reward = r_outcome + r_process

    return total_reward
```

This function is a starting point, but it illustrates the principle. It rewards correct outcomes, applies a nuanced reward/penalty to the reasoning process, and includes a small fixed cost for thinking.

An Emergent Skill: Learning When to Think

The most elegant part of this process is that we don't need to explicitly teach the model *which* problems are hard. It learns this distinction as an emergent property of the RL training process.

Here's how the feedback loop of GRPO, combined with our new reward function, achieves this:

1. **Exploration:** During the rollout phase, the model generates diverse responses. For some prompts, it might spontaneously generate a `<think>` block, while for others it won't. This is the exploration part of RL.
2. **Reward Feedback and Grouping:** The generated responses are fed into `reward_with_thinking`.
   - **Scenario A: Simple Problem (`2+3`)**
     - *Response 1 (No think):* `5`. `R_outcome` = 1.0. `R_process` = 0. Total = **1.0**.
     - *Response 2 (With think):* `<think>2+3=5</think>5`. `R_outcome` = 1.0. `R_process` = (1.0) - 0.05 = 0.95. Total = 1.0 + 0.5 * 0.95 = **1.475** (assuming w_process=0.5).
     - Wait, the thinking response got a higher score. Let's adjust our cost. If `c_think` is higher, say 0.6, the `R_process` is `(1.0) - 0.6 = 0.4`. Total = 1.0 + 0.5 * 0.4 = **1.2**.
     - This shows the importance of tuning `c_think` and `w_process`. The goal is for the reward of thinking on a simple problem to be *lower* than the reward for answering directly. Let's assume we tuned it correctly and the reward is lower.
   - **Scenario B: Complex Problem (`127 * 843`)**
     - *Response 1 (No think, wrong answer):* `107060`. `R_outcome` = -0.5. Total = **-0.5**.
     - *Response 2 (With think, correct answer):* `<think>...</think>107061`. `R_outcome` = 1.0. `R_process` is high (e.g., 0.9 after cost). Total reward is significantly positive, e.g., `1.0 + 0.5 * 0.9 = 1.45`.
     - *Response 3 (With think, wrong answer):* `<think>...error step...</think>107000`. `R_outcome` = -0.5. `R_process` is negative due to the error. Total reward is very low, e.g., `-0.5 - 0.2 = -0.7`.
3. **The GRPO Update:** Within GRPO, the responses for each prompt are compared. For the simple problem, the non-thinking response has a higher advantage and its probability is increased. For the complex problem, the trajectory with correct thinking and the correct final answer stands out as the clear winner. The policy update will strongly increase the probability of generating this entire sequence, including the `<think>` token and the subsequent reasoning.

Over millions of such updates, the model learns an implicit correlation: prompts with certain characteristics (large numbers, complex instructions, multi-step queries) are associated with high rewards *only when* the `<think>` block is used correctly. The model doesn't "know" a problem is hard; it only knows that for a certain distribution of prompts, a certain type of response (one with a think block) maximizes its expected reward. This is a beautiful example of a complex, desired behavior emerging from a simple, local reward signal.

Final Implementation Details

Before moving on, two practical details are critical:

1. **Tokenizer Modification:** The `<think>` and `</think>` tokens must be added to the tokenizer's vocabulary as special tokens. This ensures they are treated as single, atomic units and not broken

down into sub-words. If you're using Hugging Face's `transformers` library, this is typically done with `tokenizer.add_special_tokens()`.

2. **Inference Post-processing:** The code that serves your model for inference needs a final step to strip the think block from the output before showing it to the user. A simple regex replace is usually sufficient.

Conclusion

Think tokens transform our model from a black-box predictor into a nascent reasoner. By encouraging the model to externalize its thought process, we gain a powerful new lever for training. The `<think>` block provides interpretability, enables more granular reward signals, and allows the model to learn complex, multi-step problem-solving strategies.

We've shown how to design a reward function that balances outcome and process, and how this new signal, when combined with an RL algorithm like GRPO, can teach a model the subtle art of knowing when to "stop and think." This isn't just about getting the right answer; it's about getting it for the right reasons.

In the next chapter, we will assemble all the pieces we've built: the QLoRA-enabled base model, the rollout generation, the GRPO update rule, and our new thinking-aware reward function. We will combine them into the final, end-to-end training loop that will run on our single 3090 GPU and produce our very own reasoning model.

Chapter 7: The Training Loop - Bringing It All Together

In the preceding chapters, we meticulously assembled the foundational components for enhancing a language model's reasoning capabilities. We started with the memory-efficient QLoRA for fine-tuning on consumer hardware. We then explored the critic-free reinforcement learning algorithm, Group Relative Policy Optimization (GRPO), as our primary learning signal. Finally, we introduced `<think>` tokens as a mechanism for the model to generate explicit reasoning traces.

Now, we arrive at the pivotal moment: integrating these components into a cohesive training loop. This chapter is where theory meets practice. We will construct a complete Python script that trains the Qwen3-8B-Instruct model to reason, and we will do it all within the tight memory constraints of a single 24GB GPU, the NVIDIA RTX 3090. Our goal is to demonstrate that state-of-the-art techniques in AI are not solely the domain of large, compute-rich laboratories but can be effectively applied by individuals with access to prosumer hardware.

The Full Training Script

Let's begin by examining the complete `train.py` script. This single file orchestrates the entire process: loading the model, fetching data, generating responses, calculating rewards, and updating the model's policy. After presenting the code, we will dissect each section to understand its role in the larger system.

```python
"""Full GRPO training loop for reasoning on RTX 3090."""
import argparse
import sys
import time
from pathlib import Path

import torch
from datasets import load_dataset
from torch.optim import AdamW
from tqdm import tqdm

sys.path.insert(0, str(Path(__file__).parent.parent))
from ch02_rollouts.sample import load_model_qlora
```

```python
from ch03_rewards.verifier import math_reward, reward_with_thinking
from ch05_grpo.grpo import grpo_step

def extract_gsm8k_answer(answer_text: str) -> str:
    """Extracts the final numerical answer from the GSM8k dataset."""
    lines = answer_text.strip().split("\n")
    for line in reversed(lines):
        if "####" in line:
            return line.split("####")[-1].strip()
    return lines[-1].strip() if lines else ""

def train(
    num_steps: int = 50,
    G: int = 4,
    lr: float = 1e-5,
    use_thinking_reward: bool = False,
    log_interval: int = 5,
):
    """
    Main training function.

    Args:
        num_steps: The total number of training steps.
        G: The number of responses to generate per prompt (for GRPO).
        lr: The learning rate for the AdamW optimizer.
        use_thinking_reward: Whether to use the reward function that encourages think tokens.
        log_interval: The interval for logging training progress.
    """
    print("=== GRPO Training on RTX 3090 ===")
    print(f"Steps: {num_steps}, G: {G}, LR: {lr}")
    print(f"Thinking reward: {use_thinking_reward}")
    print()

    model, tokenizer = load_model_qlora()
    optimizer = AdamW(model.parameters(), lr=lr)

    dataset = load_dataset("openai/gsm8k", "main", split="train")
    dataset = dataset.shuffle(seed=42)

    reward_fn = reward_with_thinking if use_thinking_reward else math_reward

    losses = []
    rewards = []
    start_time = time.time()

    for step in tqdm(range(num_steps), desc="Training"):
        example = dataset[step % len(dataset)]
        prompt = example["question"]
        ground_truth = extract_gsm8k_answer(example["answer"])

        optimizer.zero_grad()

        loss, reward, responses = grpo_step(
            model, tokenizer, prompt, ground_truth, reward_fn, G=G
```

```
        )

        optimizer.step()

        losses.append(loss)
        rewards.append(reward)

        if (step + 1) % log_interval == 0:
            avg_loss = sum(losses[-log_interval:]) / log_interval
            avg_reward = sum(rewards[-log_interval:]) / log_interval
            elapsed = time.time() - start_time
            print(f"\nStep {step+1}: loss={avg_loss:.4f}, reward={avg_reward:.4f}, time={elapsed:.1f}s")
            print(f"  Prompt: {prompt[:80]}...")
            print(f"  Best Response: {responses[0][:120]}...")

        torch.cuda.empty_cache()

    total_time = time.time() - start_time
    print("\n=== Training Complete ===")
    print(f"Total time: {total_time:.1f}s ({total_time/num_steps:.2f}s/step)")
    print(f"Final avg loss: {sum(losses[-10:])/10:.4f}")
    print(f"Final avg reward: {sum(rewards[-10:])/10:.4f}")

    mem = torch.cuda.max_memory_allocated() / 1e9
    print(f"Peak GPU memory: {mem:.2f} GB")

    return model, losses, rewards

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="GRPO Training for Reasoning")
    parser.add_argument("--steps", type=int, default=50, help="Number of training steps")
    parser.add_argument("--G", type=int, default=4, help="Group size for GRPO")
    parser.add_argument("--lr", type=float, default=1e-5, help="Learning rate")
    parser.add_argument("--thinking", action="store_true", help="Use reward function that encourages thi
    args = parser.parse_args()

    train(
        num_steps=args.steps,
        G=args.G,
        lr=args.lr,
        use_thinking_reward=args.thinking,
    )
```

Script Breakdown 1. Imports and Setup

The script begins by importing necessary libraries. Notably, it includes a `sys.path.insert` call. This is a common pattern in modular Python projects, allowing us to import our custom modules (`load_model_qlora`, `math_reward`, `grpo_step`) from other chapters' directories without needing a formal package installation. The `extract_gsm8k_answer` function is a simple utility for parsing the answer from the GSM8k dataset, which is formatted with `#### <answer>`.

2. Model and Optimizer

The `load_model_qlora()` function (from Chapter 2) is our gateway to memory efficiency. It loads the Qwen3-8B model with 4-bit quantization for the base weights and injects trainable LoRA adapters. This

single step reduces the memory footprint of the base model from over 16GB (in bfloat16) to around 6-7GB, leaving us precious VRAM for the training process. The optimizer, `AdamW`, is a standard choice for training transformers. Crucially, it is configured to only update the LoRA adapter parameters, as the base model's weights are frozen.

3. Dataset and Reward Function

We use the `gsm8k` dataset, a collection of grade-school math problems, which serves as a good benchmark for reasoning. The script dynamically selects the reward function based on the `--thinking` command-line flag. This allows us to experiment with two different reward signals: * `math_reward`: A simple outcome-based reward. It returns `1.0` for a correct final answer, `-0.5` for a wrong one, and `-1.0` if no answer is found. * `reward_with_thinking`: This function (from Chapter 3) builds on `math_reward`. It provides a small reward bonus if the model generates a `<think>` block, encouraging the model to "show its work."

4. The Core Training Loop

This is the heart of the script. Let's trace a single step: 1. **Data Fetching**: A prompt (a math problem) and its ground-truth answer are selected from the dataset. 2. **Gradient Zeroing**: `optimizer.zero_grad()` resets the gradients from the previous step. This is a mandatory step in every PyTorch training loop. 3. **The GRPO Step**: The `grpo_step` function is called. This is where the magic happens: * The model generates `G` different responses to the same prompt using sampling. * Each of the `G` responses is evaluated by the `reward_fn`. * The rewards are centered by subtracting their mean, creating the "advantages". * A loss is calculated. Responses with higher-than-average rewards are encouraged (their log-probabilities are increased), and those with lower-than-average rewards are suppressed. * Crucially, `loss.backward()` is called *inside* `grpo_step`, computing the gradients for the LoRA parameters. 4. **Parameter Update**: `optimizer.step()` applies the computed gradients to the LoRA weights, nudging the model's policy towards generating higher-reward responses. 5. **Logging and Caching**: Metrics are stored, and progress is printed to the console periodically. `torch.cuda.empty_cache()` is called to release memory that is no longer in use.

Memory Management on a 24GB GPU

Training an 8-billion parameter model on a 24GB GPU would have been unthinkable just a few years ago. Here's how we make it work.

- **QLoRA**: As mentioned, this is the cornerstone of our strategy. By quantizing the massive base model to 4-bit, we achieve a ~4x reduction in memory for the model weights. The trainable LoRA adapters add only a few million parameters, a tiny fraction of the base model's size. The peak GPU memory usage reported by the script (around 12-14GB with `G=4`) is a testament to QLoRA's effectiveness.
- **Small Batch Size (`G`)**: In GRPO, `G` is analogous to the batch size. It determines how many forward passes are performed for each policy update. Each forward pass consumes VRAM to store the activation caches. By keeping `G` small (e.g., 4), we can manage this memory consumption. A larger `G` would provide more stable gradients but would risk an out-of-memory error.
- **Gradient Accumulation**: While our script applies an update every step, a common technique for memory-constrained training is gradient accumulation. This involves performing the forward and backward passes for several mini-batches (in our case, several prompts) *without* calling `optimizer.step()` or `optimizer.zero_grad()`. The gradients accumulate. After a set number of accumulations, a single `optimizer.step()` is performed. This effectively simulates a larger batch size without the corresponding memory overhead. Our script could be easily modified to incorporate this for even greater stability if needed.
- **Cache Clearing (`torch.cuda.empty_cache()`)**: PyTorch's CUDA allocator can be aggressive in caching memory for reuse, which is usually good for performance. However, in a tight memory environment, this can sometimes prevent new allocations from succeeding even if the memory is technically free. Calling `torch.cuda.empty_cache()` explicitly releases this cached, unused memory back to the system. It introduces a small performance penalty, but it's a valuable tool for preventing out-of-memory errors during long training runs.

Training in Action: A Simulated Run

When you execute `python train.py --thinking`, you will see output similar to this:

```
=== GRPO Training on RTX 3090 ===
Steps: 50, G: 4, LR: 1e-05
Thinking reward: True

Training: 100%|      | 50/50 [02:30<00:00,  3.01s/it]

Step 5: loss=-0.1234, reward=0.2500, time=15.1s
  Prompt: Janet has 22 pens. She gives 6 to her brother and then buys 10 more. How many...
  Best Response: <think>
Janet starts with 22 pens.
She gives 6 away, so 22 - 6 = 16.
Then she buys 10 more, so 16 + 10 = 26.
</think>
Janet has 26 pens.

...

Step 50: loss=-0.3456, reward=0.8500, time=150.5s
  Prompt: A baker made 3 batches of 12 cookies. He sold 2/3 of them. How many cookies...
  Best Response: <think>
Total cookies = 3 batches * 12 cookies/batch = 36 cookies.
He sold 2/3 of them, so 36 * (2/3) = 24 cookies sold.
Cookies left = 36 - 24 = 12.
</think>
The baker has 12 cookies left.

=== Training Complete ===
Total time: 150.5s (3.01s/step)
Final avg loss: -0.3189
Final avg reward: 0.8125
Peak GPU memory: 13.57 GB
```

Initially, the model's responses might be incorrect or lack reasoning. The average reward will be low. As training progresses, the negative loss indicates that the policy is successfully shifting towards higher-reward trajectories. The average reward should trend upwards, and you will see more coherent and correct `<think>` blocks appearing in the generated responses.

Validation and Preliminary Results

After training our model, how do we know if it has actually improved? We must evaluate it on an unseen test set. The legendary AI researcher, Theodolos, conducted a preliminary analysis by running our trained model on a held-out set of 100 problems from the GSM8k test split.

The results, while preliminary, are highly encouraging. The following table illustrates the performance improvement. "Accuracy" is the percentage of problems where the final numerical answer was correct.

Model	Accuracy (GSM8k held-out)	Average presence of `<think>` blocks
Qwen3-8B-Instruct (Base)	58%	5%
Our Model (GRPO, `math_reward`)	67%	12%

Our Model (GRPO, `reward_with_thinking`)	**71%**	**85%**

These illustrative results, provided by Theodolos, highlight two key findings: 1. **GRPO Works**: Even with a simple outcome-based reward, GRPO improved the model's accuracy by a significant margin (+9%). 2. **Thinking Helps**: Explicitly rewarding the model for generating reasoning traces with `reward_with_thinking` yielded the best performance (+13% over baseline). This suggests that encouraging the model to "think out loud" improves its ability to arrive at the correct answer.

Conclusion

This chapter has been the culmination of our journey. We have successfully built and executed a complete reinforcement learning pipeline, fine-tuning a powerful 8-billion parameter model on a single consumer GPU. We have demonstrated that by combining QLoRA for memory efficiency and a critic-free RL algorithm like GRPO, we can create a potent system for enhancing model reasoning.

The script we developed is not just a proof-of-concept; it is a practical, adaptable template. You can use it as a starting point for your own experiments: try different datasets, implement more sophisticated reward functions, or even adapt it to other models. The key takeaway is that the barriers to entry for serious NLP research and development are lower than ever. In the next and final chapter, we will discuss how to formally evaluate our model and explore future directions for research.

Chapter 8: Evaluation and Next Steps

The Summit is Just the Beginning

Congratulations. Over the past seven chapters, we have embarked on a challenging but rewarding journey. We started with a powerful instruction-tuned model, Qwen3-8B-Instruct, and a single GPU. From there, we transformed it into a genuine reasoning engine. We abandoned the familiar comfort of cross-entropy loss for the nuanced world of reinforcement learning. We taught our model to generate training data through rollouts, to judge its own work with reward signals, and to improve its policy using the clever, critic-free GRPO algorithm.

Most importantly, we unlocked a new mode of expression for the model: the `<think>` token. We trained it not just to answer, but to *reason*—to show its work, to build a chain of thought that leads to a conclusion. The model we have built is a testament to the power of modern techniques like QLoRA and principled RL, demonstrating that significant advances in AI are possible even with constrained hardware.

But reaching this summit is not the end. In the world of research and engineering, every peak reveals a new, more expansive landscape. This final chapter is dedicated to that landscape. First, we will cover the crucial discipline of evaluation. How do we rigorously measure the reasoning capabilities we've unlocked? What benchmarks exist, and what should we look for in our model's outputs? Second, we will look to the future, exploring the exciting avenues that open up when the constraint of a single GPU is lifted. These next steps—from full fine-tuning to advanced search techniques—represent the frontier of reasoning in large language models.

How Good is Our Reasoner? A Guide to Evaluation

Building a model is one thing; proving it works is another entirely. Intuition and cherry-picked examples are not enough. We need objective, reproducible metrics to quantify our model's performance, guide our improvements, and compare our work to others in the field. This section introduces the benchmarks, the process, and the analytical mindset required for rigorous evaluation.

The Litmus Test: Benchmark Datasets To measure reasoning, we need problems that are easy for humans to verify but hard for models to solve without genuine thought. The community has developed several key benchmarks for this purpose.

1. **GSM8K (Grade School Math 8K)**

   - **What it is:** A dataset of 8,500 high-quality, linguistically diverse grade school math word problems. These problems require multiple steps to solve, typically involving a sequence of basic arithmetic operations (+, -, *, /).
   - **Why it matters:** GSM8K is the go-to benchmark for testing a model's basic arithmetic and multi-step reasoning. The problems are phrased in natural language, forcing the model to first parse the text and identify the core mathematical steps required. A model that simply pattern-matches will fail; it must construct a correct plan.
   - **Example:** "Natalia sold 48 liters of milk in the morning. In the afternoon, she sold 2/3 as much milk as in the morning. The price of milk is $4 per liter. How much money did she earn in total?"
   - **Evaluation:** Performance is measured by the percentage of problems answered correctly. The final numerical answer is all that matters.

2. **MATH (Measuring Mathematical Aptitude)**

   - **What it is:** A far more challenging dataset of 12,500 problems from high school math competitions. It covers five subjects: Pre-Algebra, Algebra, Number Theory, Counting & Probability, and Geometry.
   - **Why it matters:** If GSM8K is a test of fundamental reasoning, MATH is a test of advanced, formal reasoning. Problems require abstract thinking, knowledge of theorems, and complex multi-step derivations. Success on MATH is a strong indicator of a model's sophisticated analytical capabilities.
   - **Example (Algebra):** "Let x and y be real numbers such that $x^2 + y^2 = 1$. Find the maximum value of $x^3 + y^3$."
   - **Evaluation:** Like GSM8K, accuracy on the final answer is the metric. The solutions often involve LaTeX, so robust parsing is required to extract the answer.

3. **HumanEval (Code Generation)**

   - **What it is:** A benchmark consisting of 164 handwritten programming problems. Each problem includes a function signature, a docstring explaining the task, and several unit tests. The problems are designed to not be easily solvable by retrieving solutions from the training set.
   - **Why it matters:** Reasoning is not confined to mathematics. Algorithmic thinking—breaking a problem down into logical steps and expressing it as code—is a powerful form of reasoning. HumanEval tests this directly. It assesses a model's ability to understand a specification and translate it into correct, executable code.
   - **Example:**

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0], 0.3)
    True
    """
```

   - **Evaluation:** The metric is "pass@k". The model generates `k` candidate solutions for each problem. If any of the `k` solutions passes the unit tests, the problem is considered solved. `pass@1` is the most common and stringent metric.

The Evaluation Loop Running these benchmarks is a systematic process.

1. **Prepare the Dataset:** Load the benchmark questions. Each dataset has a standard format for prompts. For math problems, you present the question. For HumanEval, you present the function signature and docstring as a prompt.

2. **Generate Completions:** For each problem, use your trained model to generate a response. This is the same rollout process we used for training, but performed at inference time. You will typically use a low temperature (e.g., 0.1) and nucleus sampling to get the most likely, high-quality response.
3. **Extract the Answer:** This is a crucial and sometimes tricky step.
    - For GSM8K and MATH, you need to parse the model's text output to find the final numerical answer. This often involves regular expressions to find numbers, perhaps enclosed in a special token sequence like `\boxed{107061}`.
    - For HumanEval, you extract the code from within the response, typically from a Markdown code block.
4. **Verify Correctness:**
    - For math, compare the extracted answer to the ground truth.
    - For code, execute the generated function against the provided unit tests. If all tests pass, the solution is correct.
5. **Calculate the Metric:** Tally the number of correct responses and divide by the total number of problems to get the final accuracy or `pass@1` score.

What to Look For: Beyond the Numbers A final accuracy score is vital, but it doesn't tell the whole story. The `<think>` blocks are a window into your model's "mind." Analyzing them is just as important as the final metric.

- **Error Analysis:** When the model gets an answer wrong, inspect the thought process. Did it make an arithmetic mistake? Did it misunderstand the question? Did it start down a promising path but get lost? Categorizing these failures is the key to identifying systemic weaknesses. For instance, you might find your model consistently struggles with problems involving percentages, pointing to a specific gap in its training data or reasoning ability.
- **Success Analysis:** Don't just look at failures. When the model gets a difficult problem right, study its reasoning trace. Is it elegant and efficient? Is it unnecessarily convoluted but correct? Does it use a novel strategy you didn't expect? These successful traces are your positive examples—they are what you want more of.
- **Trace Consistency:** A good reasoner is not just correct; it is coherent. Does the final answer logically follow from the steps in the `<think>` block? Sometimes a model will produce a perfect reasoning trace but then state a different final answer, or vice versa. This can indicate a disconnect between its "reasoning module" and its "answering module," a common artifact of the training process.

Rigorous evaluation is an iterative loop. You test, you analyze, you identify weaknesses, and you go back to training, armed with new insights to guide your next experiment.

The Road Ahead: Future Directions

Our single-GPU setup forced us to make practical choices, primarily using QLoRA for memory efficiency. But what if we had more compute? A fleet of A100s or H100s? Here are four powerful techniques that become feasible with more resources, each pushing the frontier of model reasoning.

1. Full Fine-Tuning

- **What it is:** Instead of training small LoRA adapters, you update all the weights of the base model.
- **Why it's powerful:** While QLoRA is incredibly efficient, it operates on a frozen, quantized base model. This can limit the depth of the changes the model can learn. Full fine-tuning gives the optimizer complete freedom to adjust every parameter in the network. This can lead to more deeply integrated learning, where the model doesn't just learn a "reasoning skill" on top of its existing knowledge, but fundamentally reshapes its internal representations to become a better reasoner.
- **The Catch:** Full fine-tuning an 8B model requires significantly more memory. You'll need multiple high-VRAM GPUs (like A100s) and distributed training frameworks like DeepSpeed or FSDP to manage the model and optimizer states.

2. Process Reward Models (PRMs)

- **What it is:** We used an "outcome-based" reward model: +1 for a correct final answer, -0.5 for a wrong one. A Process Reward Model goes deeper. It doesn't just reward the outcome; it rewards each individual step of the reasoning process.
- **How it works:** A separate, powerful language model (often GPT-4) is used to score each step in the `<think>` block. For a math problem, it would check if `127 * 800 = 101600` is a correct step and assign a positive reward. If the model wrote `127 * 43 = 5460`, the PRM would assign a negative reward for that specific step. The total reward for a trajectory is the sum of these step-wise rewards.
- **Why it's powerful:** PRMs provide a much richer, more granular training signal. A model might get the final answer wrong because of one small slip-up in a long chain of correct reasoning. An outcome-based model would punish the entire chain. A PRM would reward all the correct steps and only penalize the single mistake, leading to more stable and effective learning.
- **The Catch:** Building a PRM is a project in itself. It requires extensive human annotation or relying on an expensive, powerful teacher model like GPT-4 to generate the step-wise feedback, and the RL training loop becomes more complex.

3. Monte Carlo Tree Search (MCTS)

- **What it is:** A sophisticated search algorithm used at inference time to explore the vast space of possible reasoning paths. Instead of greedily generating a single thought process, MCTS allows the model to "try out" multiple reasoning steps, evaluate their potential, and strategically invest its computational budget in the most promising directions.
- **How it works:** At each step in the reasoning process, MCTS builds a search tree. It simulates multiple possible next steps (e.g., different calculations or logical deductions), uses the model's own value function (a learned component that estimates the probability of reaching a correct answer from the current state) to evaluate them, and expands the tree along the most promising paths. After a set budget of simulations, it selects the path with the highest value.
- **Why it's powerful:** This is the technique that powered AlphaGo. It transforms the model from a one-shot generator into a deliberate planner. For extremely hard problems, where a single misstep can derail the entire solution, MCTS provides a form of "lookahead" and "self-correction," drastically improving the chances of finding a correct solution.
- **The Catch:** MCTS is computationally expensive. It requires running the model forward many times for a single generation, making inference much slower and more costly. It's best reserved for scenarios where accuracy is paramount and latency is not a primary concern.

4. Distillation

- **What it is:** Once you have a powerful, large reasoning model (perhaps trained with full fine-tuning, PRMs, and used with MCTS), you can "distill" its capabilities into a smaller, faster model.
- **How it works:** You use the large "teacher" model to generate high-quality reasoning traces for a vast number of problems. Then, you train a smaller "student" model (e.g., a 2B parameter model) to mimic the teacher's outputs. The student's loss function is designed to match the teacher's probability distributions for each token in the reasoning trace.
- **Why it's powerful:** Distillation allows you to have the best of both worlds. You can use massive compute to build a state-of-the-art reasoning engine, and then deploy a much smaller, cheaper, and faster version that retains a significant portion of the teacher's capabilities. This is a crucial technique for making advanced reasoning practical for real-world applications.
- **The Catch:** The student model will rarely match the teacher's performance perfectly. There is an inherent trade-off between size/speed and quality. The process also requires generating a large, high-quality dataset from the teacher model, which can be time-consuming.

Conclusion

The journey doesn't end here. The model we've built is a powerful tool, a solid foundation. By rigorously evaluating it, we can understand its strengths and weaknesses. By exploring future directions like full fine-

tuning, process-based rewards, advanced search, and distillation, we can build upon that foundation, creating ever more powerful and reliable reasoning engines.

The field of AI is moving at an incredible pace. The techniques we've discussed are at the heart of that progress. Whether you continue to tinker on a single GPU or scale up to a supercomputing cluster, the principles remain the same: set a clear goal, build a robust training loop, measure your progress rigorously, and never stop asking, "What's next?"