A Comprehensive Guide to Data for Large Language Model Training

This guide provides a detailed overview of the data formats, sourcing strategies, and theoretical foundations required for training and fine-tuning Large Language Models (LLMs). It covers the three primary stages of modern LLM training: Supervised Fine-Tuning (SFT), Reward Modeling (RM), and Reinforcement Learning with Human Feedback (RLHF). The guide also delves into crucial topics such as Scaling Laws, data augmentation techniques, and provides a rich set of theoretical and practical interview questions with detailed solutions.

Knowledge Section

The Three-Stage Training Paradigm of Modern LLMs

Training a state-of-the-art, instruction-following Large Language Model, such as ChatGPT or Llama, typically involves a three-stage process. Understanding the data requirements for each stage is critical.

1. **Pre-training (Optional Foundation)**: A base model is trained on a massive, general-purpose corpus of text (e.g., a large portion of the internet and books). The objective is language modeling—predicting the next word. This stage is computationally expensive and usually performed by large research labs. The subsequent stages adapt this base model.
2. **Supervised Fine-Tuning (SFT)**: The pre-trained model is taught to follow instructions and act as a helpful assistant. This is done by fine-tuning it on a smaller, high-quality dataset of instruction-response pairs.
3. **Reward Modeling (RM)**: To align the model's behavior more closely with human preferences, a separate Reward Model is trained. This model learns to score the quality of the LLM's responses.
4. **Reinforcement Learning with Human Feedback (RLHF)**: The SFT model is further fine-tuned using reinforcement learning. The Reward Model from the previous stage provides the reward signal, guiding the LLM to generate responses that humans would rate highly. Proximal Policy Optimization (PPO) is

the most common algorithm for this stage.

Stage 1: Supervised Fine-Tuning (SFT) Data Format

Supervised Fine-Tuning is the first step in adapting a base LLM to follow user instructions. The goal is to teach the model a specific input-output format and a helpful, harmless, and honest conversational style.

Concept: The model is trained on a dataset of high-quality "demonstrations" of desired behavior. These are typically structured as prompt-response pairs.

Data Format:
The data is a collection of examples, where each example contains an instruction and its desired output. While a simple `{"input": "...", "output": "..."}` format works, a more structured format is often used to handle various instruction types, especially those that require additional context. The "Alpaca" format is a popular standard:

- `instruction`: The primary task or question given to the model.
- `input`: (Optional) Additional context for the instruction. For example, if the instruction is "Summarize the following text," the text to be summarized goes here.
- `output`: The desired, high-quality response that the model should learn to generate.

Example (JSON format):

```
[
  {
    "instruction": "Translate the following English sentence to French.",
    "input": "Hello, how are you?",
    "output": "Bonjour, comment ça va ?"
  },
  {
    "instruction": "Explain the theory of relativity in simple terms.",
    "input": "",
    "output": "The theory of relativity, proposed by Albert Einstein, is based on two mai
  }
]
```

Training Objective:
The SFT stage uses a standard **Causal Language Modeling** objective. The model is trained to predict the next token in a sequence. The `instruction`, `input`, and `output` are concatenated into a single sequence. The model calculates a loss only on the `output` part of the sequence, training it to generate the desired response given the prompt. The loss function is typically **Cross-Entropy Loss**.

For a sequence of tokens $Y = (y_1, y_2, \ldots, y_k)$ (the desired output), given a context X (the instruction and input), the loss for this single example is the negative log-likelihood:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{i=1}^{k} \log P(y_i | y_1, \ldots, y_{i-1}, X; \theta)$$

Where θ represents the model's parameters.

Stage 2: Reward Modeling (RM) Data Format

After SFT, the model can follow instructions, but its responses may not always align with what humans find most helpful or safe. The Reward Model is trained to act as a proxy for human preference.

Concept: The RM is a separate language model (often initialized from the SFT model) that is trained not to generate text, but to take a prompt and a response as input and output a single scalar value representing its "goodness" or "preference score."

Data Format:
To learn human preferences, the RM is trained on a dataset of comparisons. For a single prompt, several responses are generated by the SFT model. A human labeler then ranks these responses from best to worst. For training, these rankings are broken down into pairs of `chosen` (better) and `rejected` (worse) responses.

Example (JSON format):

```
[
  {
    "prompt": "Write a short poem about a robot learning to dream.",
    "chosen": "Circuits hum a silent tune,\nBeneath a cold and metal moon.\nI process data
    "rejected": "I am a robot. I can compute things. I do not sleep. Therefore, I cannot d
  },
  {
    "prompt": "What are the main benefits of Python?",
    "chosen": "Python's main benefits include its simple, readable syntax which lowers the
    "rejected": "Python is a programming language. It is used for coding."
  }
]
```

Training Objective:
The RM is trained as a regression model to predict a score, but the loss is based on the comparison data. The model should assign a higher score to the `chosen` response than to the `rejected` one. The loss function is often a pairwise ranking loss, with the **Bradley-Terry model** being a common inspiration.

Let $r_\phi(x, y)$ be the scalar score output by the reward model with parameters ϕ for a prompt x and completion y. For a pair of completions (y_c, y_r) where y_c is chosen and y_r is rejected, the loss is:

$$\mathcal{L}_{\text{RM}}(\phi) = -\mathbb{E}_{(x,y_c,y_r)\sim D}\left[\log(\sigma(r_\phi(x, y_c) - r_\phi(x, y_r)))\right]$$

Here, σ is the sigmoid function. This loss maximizes the margin between the scores of chosen and rejected responses, effectively training the RM to understand human preferences.

Stage 3: RLHF using PPO Data Format

This is the final alignment stage, where the SFT model is fine-tuned using reinforcement learning to maximize the scores given by the Reward Model.

Concept: The LLM is treated as an **agent** in an RL environment.

- **State**: The sequence of tokens generated so far, given an initial prompt.

- **Action**: Choosing the next token to add to the sequence.
- **Policy**: The LLM itself, which defines a probability distribution over the vocabulary for the next token. This is the policy π_θ^{RL}.
- **Reward**: At the end of a generated sequence, the Reward Model r_ϕ provides a score. An additional penalty term is often added to prevent the model from deviating too far from the original SFT model.

Data Format:
Unlike SFT and RM, this stage doesn't use a static dataset. Instead, it generates data on the fly in an iterative loop:

1. A prompt x is sampled from a prompt dataset (similar to the SFT/RM prompts).
2. The RL policy π_θ^{RL} generates a response y.
3. The Reward Model r_ϕ evaluates the response and returns a reward $R = r_\phi(x, y)$.
4. This `(prompt, generated_response, reward)` tuple is used to update the policy's parameters θ.

PPO Objective Function:
The Proximal Policy Optimization (PPO) algorithm is used to update the LLM's weights. Its objective function is complex but designed for stable training. It tries to maximize the reward while staying close to the original SFT policy π^{SFT}.

The objective for a given prompt x and generated response $y \sim \pi_\theta^{\text{RL}}(\cdot|x)$ is:

$$\text{Objective}(\theta) = \mathbb{E}_{(x,y)\sim D}\left[r_\phi(x, y) - \beta \cdot \text{KL}(\pi_\theta^{\text{RL}}(y|x)||\pi^{\text{SFT}}(y|x))\right]$$

Let's break this down:

- $r_\phi(x, y)$: The reward from the Reward Model. The policy is updated to maximize this term.
- $\pi_\theta^{\text{RL}}(y|x)$: The probability of generating response y with the current policy being trained.
- $\pi^{\text{SFT}}(y|x)$: The probability of generating response y with the initial SFT model (whose weights are kept frozen).
- $\text{KL}(\cdot||\cdot)$: The Kullback-Leibler (KL) divergence. This term measures how much the current policy has diverged from the original SFT policy.

- β: A hyperparameter that controls the strength of the KL penalty. A higher β forces the model to stay closer to its original SFT behavior, preventing it from "over-optimizing" for the reward model and producing gibberish.

This per-token KL penalty ensures the model doesn't forget its language modeling capabilities while learning to generate higher-quality responses.

Sourcing Datasets for LLM Training

Public Datasets for Pre-training and Fine-tuning

1. **Massive Pre-training Corpora**:
    - **Common Crawl**: A massive web crawl dataset containing petabytes of raw web page data. It's diverse but requires extensive cleaning.
    - **C4 (Colossal Cleaned Common Crawl)**: A cleaned version of Common Crawl used to train Google's T5 model.
    - **The Pile**: A large, diverse, and well-curated 825 GiB text dataset created by EleutherAI, composed of 22 smaller high-quality datasets (e.g., PubMed, arXiv, GitHub, Wikipedia).
    - **Wikipedia**: High-quality, structured encyclopedic text. Available in many languages.
    - **BookCorpus / Books3**: Datasets containing text from a large number of books, providing long-range coherence and formal language.
2. **Instruction Fine-Tuning Datasets**:
    - **Alpaca**: 52k instruction-following examples generated by `text-davinci-003` from a seed set of 175 human-written tasks.
    - **Dolly 15k**: ~15,000 instruction/response pairs created by Databricks employees, covering a range of tasks.
    - **OpenOrca**: A large dataset created by applying Orca replication methods to open-source models, containing augmented FLAN instructions.
    - **GLUE/SuperGLUE**: Benchmarks consisting of several NLP tasks (e.g., sentiment analysis, question answering) that can be formatted into instruction-tuning data.

Strategies for Domain-Specific Model Training

When building a model for a specific domain (e.g., legal, medical, finance), general-purpose datasets are insufficient.

1. **Collect Domain-Specific Text**: Gather text from domain-specific sources like:
    - **Medical**: PubMed abstracts, medical textbooks, clinical trial reports.
    - **Legal**: Court filings, legal opinions, contracts, legislation.
    - **Finance**: Financial reports (10-K, 10-Q), analyst reports, market news.
    - **Scientific**: arXiv pre-prints, academic journals.
2. **Crawl Domain-Specific Websites**: Use web scraping tools (like Scrapy or Beautiful Soup) to collect text from targeted forums, expert blogs, and industry websites.
3. **Use Internal Company Data**: Leverage proprietary data like customer support chats, internal documentation, and technical reports. Ensure privacy and compliance are handled correctly.
4. **Social Media**: Collect data from platforms like Twitter, Reddit, or Stack Exchange where professionals in a specific domain discuss relevant topics.

Data Augmentation for Text

When high-quality data is scarce, data augmentation can create new training examples from existing ones.

1. **Easy Data Augmentation (EDA)**: A set of simple yet effective techniques:
    - **Synonym Replacement (SR)**: Replace a few non-stop words with their synonyms.
    - **Random Insertion (RI)**: Insert synonyms of random words at random positions.
    - **Random Swap (RS)**: Randomly swap the positions of two words in the sentence.
    - **Random Deletion (RD)**: Randomly delete words from the sentence.
2. **AEDA (An Easier Data Augmentation)**: A very simple technique that involves inserting random punctuation marks (e.g., `.`, `;`, `?`, `!`) into the original text. This can improve model robustness.

3. **Back Translation**: Translate a sentence into another language (e.g., English -> German) and then translate it back to the original language (German -> English). The resulting sentence is often a paraphrase of the original. Using multiple intermediate languages can generate diverse samples.
4. **Masked Language Model (MLM) Augmentation**: Use a pre-trained masked language model like BERT or RoBERTa.
    - Mask some tokens in a sentence (e.g., `The [MASK] brown fox...`).
    - Have the MLM predict the masked tokens to generate variations of the sentence.
    - Heuristics can be used to mask less important words (e.g., based on TF-IDF or part-of-speech tags) to better preserve the original meaning.
5. **LLM-based Generation (Self-Instruct)**: Use a powerful existing LLM (like GPT-4) to generate new instruction-tuning data. This involves providing the LLM with a few seed examples and asking it to generate more diverse and complex examples in the same format.

Data Scaling and Scaling Laws

How much data is needed for fine-tuning?
This is highly task-dependent.

- **Simple style/format adaptation**: A few hundred to a few thousand high-quality examples may suffice.
- **Learning new, complex skills**: May require tens of thousands of examples.
- **Domain adaptation**: The amount depends on how different the new domain is from the model's pre-training data.

Scaling Laws for Pre-training:
Groundbreaking research, particularly from DeepMind's **Chinchilla** paper (Hoffmann et al., 2022), has refined our understanding of how to scale LLMs.

- **Previous Belief**: It was thought that model performance scaled primarily with model size (number of parameters). For a given compute budget, the best strategy was to train the largest possible model on a reasonably large dataset.

- **Chinchilla's Finding**: For optimal performance under a fixed compute budget, **both model size and the number of training tokens must be scaled equally**. The paper found that for every doubling of model size, the training dataset size should also be doubled.
- **Practical Rule of Thumb**: The Chinchilla study suggested an optimal ratio of approximately **20 tokens of training data for every parameter in the model**. For a 7 billion parameter model (like Llama 7B), this implies training on roughly 140 billion tokens for optimal performance. This is a guideline for pre-training, not fine-tuning.

This discovery means that many previous large models were "undertrained." A smaller model trained on more data (like Chinchilla 70B) could outperform a much larger model trained on less data (like Gopher 280B).

Interview Questions

Theoretical Questions

Question 1: Explain the three main stages of training an instruction-following LLM like ChatGPT or Llama 3.

Answer:
The training process involves three key stages designed to build a powerful base model and then align it with human intent and preferences.

1. **Supervised Fine-Tuning (SFT):** This is the first alignment stage. You start with a pre-trained base LLM (e.g., Llama 3 8B). The goal is to teach it to follow instructions and respond in a conversational format. This is done by training it on a high-quality dataset of instruction-response pairs (e.g., "Instruction: Explain gravity", "Response: Gravity is the force..."). The model is trained using a standard cross-entropy loss to predict the response tokens given the instruction. This stage teaches the model *what* to do.
2. **Reward Modeling (RM):** After SFT, the model is helpful but may not always generate the *best* or safest responses. The RM stage aims to capture human preferences. A separate model, the Reward Model, is trained. Its training data consists of a prompt and several responses generated by the SFT model, which

are then ranked by human labelers. The data is structured as pairs of (prompt, chosen_response, rejected_response). The RM is trained to assign a higher scalar score to the 'chosen' response than the 'rejected' one, using a pairwise ranking loss. This model learns *what humans prefer*.

3. **Reinforcement Learning with Human Feedback (RLHF):** This is the final and most complex stage. The SFT model is further fine-tuned using reinforcement learning, with the trained Reward Model acting as the reward function. The LLM (the 'policy') generates a response to a prompt. This response is then scored by the Reward Model. The reward signal is used to update the LLM's weights via an algorithm like PPO (Proximal Policy Optimization). A key component is a KL-divergence penalty that prevents the LLM from deviating too much from its original STFT capabilities, ensuring it remains a coherent language model. This stage fine-tunes the model's behavior to actively produce responses that maximize human preference scores.

Question 2: What is the purpose of the Reward Model in RLHF? Derive and explain the loss function used to train it.

Answer:
The purpose of the Reward Model (RM) is to create a computational proxy for human preferences. Since it's infeasible to have a human evaluate every single output during the RL fine-tuning phase, we instead train a model to automate this evaluation. The RM takes a prompt and a candidate response and outputs a single scalar score representing how "good" that response is, according to the preferences learned from human-ranked data.

The RM is trained on a dataset of comparisons, $D = \{(x, y_c, y_r)\}$, where for a prompt x, y_c is the response a human preferred (chosen) over y_r (rejected).

Derivation of the Pairwise Ranking Loss:
We want the model's score for the chosen response, $r_\phi(x, y_c)$, to be higher than the score for the rejected response, $r_\phi(x, y_r)$. We can model the probability that y_c is preferred over y_r using a logistic function, inspired by the Bradley-Terry model for pairwise comparisons.

1. Let the underlying "true" scores be $s_c = r_\phi(x, y_c)$ and $s_r = r_\phi(x, y_r)$.
2. The difference in scores is $\Delta s = s_c - s_r$.
3. We model the probability of y_c being preferred as the sigmoid of this difference:

$$P(y_c \succ y_r|x) = \sigma(s_c - s_r) = \frac{1}{1 + e^{-(s_c - s_r)}}$$

4. The goal of training is to maximize the likelihood of observing the human-provided preferences over the entire dataset. This is equivalent to minimizing the negative log-likelihood of this probability.
5. The loss function $\mathcal{L}_{\text{RM}}$ for a single data point (x, y_c, y_r) is the negative log of the probability above:

$$\mathcal{L}_{\text{RM}} = -\log(P(y_c \succ y_r|x)) = -\log(\sigma(s_c - s_r))$$

6. Substituting the definitions, the final loss function averaged over the dataset is:

$$\mathcal{L}_{\text{RM}}(\phi) = -\mathbb{E}_{(x,y_c,y_r)\sim D}\left[\log(\sigma(r_\phi(x, y_c) - r_\phi(x, y_r)))\right]$$

This loss function effectively pushes the score of chosen responses up and the score of rejected responses down, teaching the model to distinguish between high-quality and low-quality outputs.

Question 3: Write down and explain the objective function for PPO in the context of RLHF. What is the critical role of the KL-divergence term?

Answer:
The objective function in Proximal Policy Optimization (PPO) for RLHF is designed to maximize the reward from the Reward Model while ensuring the language model remains stable and doesn't deviate catastrophically from its well-behaved SFT state.

The objective for a prompt x and a response y generated by the policy π_θ^{RL} is:

$$\text{Objective}(\theta) = \mathbb{E}_{(x,y)\sim D_{RL}}\left[\underbrace{r_\phi(x, y)}_{\text{Reward Term}} - \underbrace{\beta \cdot \text{KL}(\pi_\theta^{\text{RL}}(y|x)||\pi^{\text{SFT}}(y|x))}_{\text{KL Penalty Term}}\right]$$

Let's break down each component:

- $\mathbb{E}_{(x,y)\sim D_{RL}}$: This indicates we are taking the expectation over prompts x from a distribution and responses y generated by our current policy π_θ^{RL}.
- $r_\phi(x, y)$: This is the **Reward Term**. It is the scalar score from our pre-trained Reward Model. The optimization process seeks to adjust the policy π_θ^{RL} to generate responses that maximize this reward.

- $\mathrm{KL}(\pi_\theta^{\mathrm{RL}}(y|x)||\pi^{\mathrm{SFT}}(y|x))$: This is the **KL Divergence** between the probability distribution of the current policy and the original SFT policy (π^{SFT}, which is kept frozen). The KL divergence measures how different two probability distributions are. In this context, it's calculated on a per-token basis.
- β: This is a hyperparameter that controls the **strength of the KL penalty**.

The Critical Role of the KL-Divergence Term:
The KL penalty term is arguably the most crucial component for successful RLHF. Its role is to act as a **regularizer** or a **constraint**.

1. **Preventing Policy Collapse ("Reward Hacking"):** Without this term, the model could learn to "hack" the reward model. It might find some strange, nonsensical sequence of tokens that the RM gives a high score to but is completely incoherent as human language. The KL penalty forces the model to generate text that is still probable under the original, well-behaved SFT model, thus preserving language quality.
2. **Ensuring Training Stability:** Reinforcement learning can be notoriously unstable. The KL term provides a stable anchor to the SFT model, preventing the policy from making excessively large updates and moving into bad regions of the policy space from which it cannot recover.
3. **Retaining General Capabilities:** The SFT model was trained on a wide variety of instructions. The RLHF process fine-tunes it on a narrower set of preferences. The KL penalty ensures that the model doesn't "forget" its general capabilities (a phenomenon known as catastrophic forgetting) while optimizing for the specific reward function.

In essence, the KL term ensures the model **improves its alignment without sacrificing its fundamental language abilities.**

Question 4: You are tasked with fine-tuning an LLM for a medical Q&A bot. How would you approach creating the SFT dataset? What are the key considerations?

Answer:
Creating a high-quality SFT dataset for a medical Q&A bot is a sensitive task that requires a meticulous approach. Here's how I would handle it:

Dataset Sourcing and Creation:

1. **Expert-Driven Data Generation:** The core of the dataset should be created by medical professionals (doctors, nurses, researchers). They would write question-answer pairs that are accurate, safe, and reflect real-world queries.
2. **Leverage Existing Medical FAQs:** Scrape and curate Q&A sections from reputable medical websites like the Mayo Clinic, a, WebMD, and government health sites (CDC, NHS). These must be reviewed and edited by experts to ensure quality and consistency.
3. **Synthesize Data from Medical Literature:** Use sources like PubMed abstracts, medical textbooks, and clinical guidelines to formulate questions and extract/summarize answers. This provides the model with domain-specific terminology and knowledge.
4. **Negative and Ambiguous Examples:** It's crucial to include examples of how the model *should not* behave.
    - **Refusal to Answer:** For questions outside its scope or that require a real doctor's diagnosis (e.g., "Do I have cancer based on this mole?"), the desired output should be a polite refusal and a strong recommendation to consult a healthcare professional.
    - **Clarification Questions:** For ambiguous queries (e.g., "Is this pill good?"), the model should be trained to ask for more context (e.g., "Could you please specify the name of the pill and the condition you are asking about?").

Key Considerations:

1. **Accuracy and Safety (Highest Priority):** Medical information must be factually correct and up-to-date. Every single data point must be verified by a medical expert. Incorrect information can have severe consequences. The bot must be explicitly trained to state that it is not a replacement for a doctor.
2. **Clarity and Simplicity:** Answers should be written in clear, simple language that a layperson can understand, avoiding overly technical jargon where possible, or explaining it if necessary.
3. **Data Privacy:** If using any real-world data (e.g., from patient forums), all Personally Identifiable Information (PII) must be rigorously anonymized.
4. **Ethical Guardrails:** The dataset must include examples that train the model to be empathetic, unbiased, and to handle sensitive topics with care.
5. **Structured Format:** I would use a consistent JSON format like

`{"instruction": "...", "input": "...", "output": "..."}` to maintain consistency and facilitate training. For example:

```
{
  "instruction": "What are the common side effects of Metformin?",
  "input": "",
  "output": "Metformin is a common medication for type 2 diabetes. Common side
}
```

The process would be iterative: create a seed dataset, fine-tune a model, test its outputs, identify weaknesses, and then augment the dataset with new examples that address those weaknesses.

Question 5: What are Scaling Laws, and how did the Chinchilla paper change our understanding of them? What is the practical implication for training a new foundation model?

Answer:
Scaling Laws in the context of deep learning are empirical principles that describe how a model's performance (typically measured by its loss on a held-out test set) predictably improves as we increase the scale of three key factors: **model size** (number of parameters, N), **dataset size** (number of training tokens, D), and the amount of **compute** used for training (C).

Previous Understanding (Pre-Chinchilla):
Before the Chinchilla paper, the prevailing wisdom, largely influenced by papers like Kaplan et al. (2020) from OpenAI, was that model size was the most important factor. The scaling laws suggested that for a given compute budget, the best performance was achieved by training the largest model possible, even if it meant training it on a relatively smaller dataset for fewer steps. The community was focused on a "bigger is better" approach for model parameters.

The Chinchilla Paper's Contribution (Hoffmann et al., 2022):
The DeepMind team conducted a rigorous study to model the optimal allocation of a fixed compute budget. Their key finding overturned the previous consensus:

For optimal performance, model size and dataset size must be scaled in roughly equal proportion.

They discovered that previous large models like Gopher (280B parameters) were severely "undertrained." They were trained on far too little data for their size. The Chinchilla paper demonstrated this by training a much smaller model, "Chinchilla" (70B parameters), on significantly more data (1.4 trillion tokens, compared to Gopher's 300 billion). The result was that the 70B Chinchilla model substantially outperformed the 280B Gopher model on a wide range of benchmarks.

Practical Implication:
The main practical implication is a fundamental shift in strategy for training new foundation models: **Don't just build a bigger model; get more high-quality data.**

For anyone planning to train a new LLM from scratch with a fixed compute budget (which is always the case), the strategy should be:

1. Estimate your total compute budget.
2. Instead of pouring it all into the largest possible parameter count, use the Chinchilla scaling laws to find the optimal balance between model size and data size.
3. This means investing significantly more resources into **data sourcing, cleaning, and curation** to build a massive, high-quality training corpus. The Chinchilla paper's rule of thumb suggests aiming for approximately **20 tokens per model parameter**.

This finding has made the "data-centric" approach to AI even more critical for LLMs, proving that the quality and quantity of data are just as important as the model's architecture and size.

Practical & Coding Questions

Question 1: Implement a simple PyTorch `Dataset` class for Supervised Fine-Tuning (SFT) that handles JSON data with "instruction", "input", and "output" fields.

Answer:
This code provides a PyTorch `Dataset` class that can load a JSON file, process the structured prompts, and prepare them for a tokenizer.

```python
import json
import torch
from torch.utils.data import Dataset
from typing import List, Dict

# Assume we have a tokenizer from a library like Hugging Face Transformers
# This is a dummy tokenizer for demonstration purposes.
class DummyTokenizer:
    def __init__(self):
        self.bos_token = "<bos>"
        self.eos_token = "<eos>"

    def encode(self, text: str) -> List[int]:
        # A real tokenizer would convert text to token IDs
        print(f"Tokenizing: {text}")
        return [ord(c) for c in text] # Simple char-to-int conversion

# SFT Dataset Implementation
class SFTDataset(Dataset):
    """
    A PyTorch Dataset for Supervised Fine-Tuning.
    It loads data from a JSON file and formats it into a single sequence.
    """
    def __init__(self, json_path: str, tokenizer):
        super().__init__()

        # Load the dataset from the JSON file
        try:
            with open(json_path, 'r', encoding='utf-8') as f:
                self.data = json.load(f)
        except Exception as e:
            print(f"Error loading or parsing JSON file: {e}")
            self.data = []

        self.tokenizer = tokenizer

    def __len__(self) -> int:
        return len(self.data)
```

```python
    def __getitem__(self, idx: int) -> Dict[str, List[int]]:
        """
        Formats and tokenizes a single data point.
        """
        item = self.data[idx]

        # Construct the prompt and full sequence
        instruction = item.get("instruction", "")
        context_input = item.get("input", "")
        output = item.get("output", "")

        # Format the prompt based on whether context_input is present
        if context_input:
            prompt_text = f"Instruction: {instruction}\nInput: {context_input}\nOutput: "
        else:
            prompt_text = f"Instruction: {instruction}\nOutput: "

        full_text = f"{self.tokenizer.bos_token}{prompt_text}{output}{self.tokenizer.eos_

        # Tokenize the entire sequence
        tokenized_full_text = self.tokenizer.encode(full_text)

        # In a real implementation, you would also create labels.
        # The loss is typically calculated only on the 'output' part.
        # This requires tokenizing the prompt separately to know where the output begins.
        tokenized_prompt = self.tokenizer.encode(f"{self.tokenizer.bos_token}{prompt_text}

        # Labels are the same as input_ids, but prompt tokens are masked with -100
        labels = list(tokenized_full_text)
        prompt_len = len(tokenized_prompt)
        labels[:prompt_len] = [-100] * prompt_len

        return {
            "input_ids": tokenized_full_text,
            "labels": labels
        }


# --- Example Usage ---
```

```
# 1. Create a dummy JSON file
dummy_data = [
    {
        "instruction": "What is the capital of France?",
        "input": "",
        "output": "Paris"
    },
    {
        "instruction": "Summarize the following text.",
        "input": "The sun is a star at the center of the Solar System.",
        "output": "The sun is the central star of our Solar System."
    }
]
json_path = "sft_data.json"
with open(json_path, 'w') as f:
    json.dump(dummy_data, f)

# 2. Instantiate the tokenizer and dataset
tokenizer = DummyTokenizer()
dataset = SFTDataset(json_path=json_path, tokenizer=tokenizer)

# 3. Access a data point
print(f"Dataset size: {len(dataset)}")
sample = dataset[1]
print("\nSample from dataset:")
print(f"input_ids: {sample['input_ids']}")
print(f"labels:    {sample['labels']}")
```

Question 2: Using only NumPy, implement the loss function for a Reward Model based on the Bradley-Terry model. Your function should take the predicted scores for a 'chosen' and 'rejected' response and compute the loss.

Answer:

This implementation directly computes the pairwise ranking loss used for training Reward Models.

```python
import numpy as np

def sigmoid(x: np.ndarray) -> np.ndarray:
    """Computes the sigmoid function."""
    return 1 / (1 + np.exp(-x))

def reward_model_loss(chosen_scores: np.ndarray, rejected_scores: np.ndarray) -> float:
    """
    Calculates the pairwise ranking loss for a batch of reward model predictions.

    Args:
        chosen_scores (np.ndarray): A 1D array of scalar scores for the chosen responses.
        rejected_scores (np.ndarray): A 1D array of scalar scores for the rejected respons

    Returns:
        float: The mean loss for the batch.
    """
    if chosen_scores.shape != rejected_scores.shape:
        raise ValueError("Shapes of chosen_scores and rejected_scores must be the same.")

    # The loss is -log(sigmoid(chosen_score - rejected_score))
    # We compute this for the entire batch and then take the mean.

    # Calculate the difference in scores
    score_diff = chosen_scores - rejected_scores

    # Calculate the log-probability for each pair
    # log(sigmoid(x)) can be computed more stably as -log(1 + exp(-x))
    log_probs = -np.log(sigmoid(score_diff))

    # The final loss is the mean over the batch
    loss = np.mean(log_probs)

    return loss

# --- Example Usage ---
# Let's assume a batch of 4 pairs of responses
# A good reward model should give higher scores to 'chosen' responses
# Batch 1: Good separation
```

```
chosen_scores_1 = np.array([2.5, 3.1, 1.8, 4.0])
rejected_scores_1 = np.array([1.0, 2.0, 0.5, 2.1])

# Batch 2: Poor separation (higher loss expected)
chosen_scores_2 = np.array([1.5, 2.1, 0.8, 3.0])
rejected_scores_2 = np.array([1.8, 2.0, 1.5, 3.3])


loss1 = reward_model_loss(chosen_scores_1, rejected_scores_1)
loss2 = reward_model_loss(chosen_scores_2, rejected_scores_2)

print(f"Scores 1 (Good Separation):")
print(f"  Chosen:   {chosen_scores_1}")
print(f"  Rejected: {rejected_scores_1}")
print(f"  Calculated Loss: {loss1:.4f}") # Expect a low loss

print("\nScores 2 (Poor Separation):")
print(f"  Chosen:   {chosen_scores_2}")
print(f"  Rejected: {rejected_scores_2}")
print(f"  Calculated Loss: {loss2:.4f}") # Expect a higher loss
```

Question 3: Write a Python function that performs "Easy Data Augmentation (EDA)" on a given sentence. Implement two of the four techniques: Synonym Replacement and Random Deletion.

Answer:

This code provides a simple, self-contained implementation of two EDA techniques.

```python
import random
from typing import List

# A simple synonym dictionary for demonstration purposes
SYNONYMS = {
    "quick": ["fast", "swift", "rapid"],
    "brown": ["dark", "ebony"],
    "fox": ["canine", "vulpine"],
    "jumps": ["leaps", "hops", "vaults"],
    "lazy": ["idle", "inactive", "slothful"],
    "dog": ["hound", "mutt", "pooch"],
    "amazing": ["incredible", "fantastic", "wonderful"],
    "story": ["tale", "narrative", "account"]
}

def synonym_replacement(words: List[str], n: int) -> List[str]:
    """
    Replaces n words in the sentence with their synonyms.
    """
    new_words = words.copy()
    # Find words that have synonyms in our dictionary
    replaceable_indices = [i for i, word in enumerate(new_words) if word.lower() in SYNONY

    # Shuffle and pick n words to replace
    random.shuffle(replaceable_indices)
    num_to_replace = min(n, len(replaceable_indices))

    for i in range(num_to_replace):
        idx = replaceable_indices[i]
        word_to_replace = new_words[idx].lower()
        synonym = random.choice(SYNONYMS[word_to_replace])
        new_words[idx] = synonym

    return new_words

def random_deletion(words: List[str], p: float) -> List[str]:
    """
    Randomly deletes each word in the sentence with probability p.
    Ensures that the sentence is not empty after deletion.
```

```
    """
    if len(words) <= 1:
        return words

    new_words = [word for word in words if random.uniform(0, 1) > p]

    # If all words were deleted, return a single random word
    if not new_words:
        return [random.choice(words)]

    return new_words

def perform_eda(sentence: str, alpha_sr: float = 0.1, alpha_rd: float = 0.1) -> Dict[str,
    """
    Performs EDA (Synonym Replacement and Random Deletion) on a sentence.

    Args:
        sentence (str): The input sentence.
        alpha_sr (float): The percentage of words to replace with synonyms.
        alpha_rd (float): The probability of deleting each word.

    Returns:
        A dictionary containing the augmented sentences.
    """
    words = sentence.split()
    num_words = len(words)

    # Synonym Replacement
    n_sr = max(1, int(alpha_sr * num_words))
    augmented_sr_words = synonym_replacement(words, n_sr)

    # Random Deletion
    augmented_rd_words = random_deletion(words, alpha_rd)

    return {
        "original": sentence,
        "synonym_replacement": " ".join(augmented_sr_words),
        "random_deletion": " ".join(augmented_rd_words)
    }
```

```python
# --- Example Usage ---
sentence1 = "The quick brown fox jumps over the lazy dog"
sentence2 = "This is an amazing story about a brave hero"

augmented1 = perform_eda(sentence1, alpha_sr=0.2, alpha_rd=0.2)
augmented2 = perform_eda(sentence2, alpha_sr=0.3, alpha_rd=0.1)

print("--- Augmenting Sentence 1 ---")
for key, value in augmented1.items():
    print(f"{key.replace('_', ' ').title():<25}: {value}")

print("\n--- Augmenting Sentence 2 ---")
for key, value in augmented2.items():
    print(f"{key.replace('_', ' ').title():<25}: {value}")
```

Question 4: Using Matplotlib, create a visualization to explain the concept of Scaling Laws. Plot hypothetical model loss against the number of model parameters for different dataset sizes.

Answer:

This code generates a plot that visually represents the core idea of Scaling Laws, especially the insight from the Chinchilla paper: for a fixed number of parameters, more data leads to lower loss.

```python
import numpy as np
import matplotlib.pyplot as plt

def hypothetical_loss(params, tokens, p_scale=1e10, t_scale=2e11):
    """
    A hypothetical loss function inspired by scaling law formulas.
    Loss decreases as params and tokens increase.
    This is a simplified model for visualization purposes.
    L(P, T) = A/P^a + B/T^b + L_inf
    """
    A = 3.0
    B = 2.5
    alpha = 0.35
    beta = 0.45
    L_inf = 1.6 # Irreducible loss

    param_term = A / ((params / p_scale)**alpha)
    token_term = B / ((tokens / t_scale)**beta)

    return param_term + token_term + L_inf

# Define a range of model sizes (parameters)
param_counts = np.logspace(8, 12, 50) # 100 Million to 1 Trillion parameters

# Define three different dataset sizes (tokens)
small_dataset_tokens = 50e9   # 50 Billion tokens
medium_dataset_tokens = 300e9 # 300 Billion tokens (e.g., Gopher scale)
large_dataset_tokens = 1.5e12 # 1.5 Trillion tokens (e.g., Chinchilla scale)

# Calculate the loss for each scenario
loss_small_data = hypothetical_loss(param_counts, small_dataset_tokens)
loss_medium_data = hypothetical_loss(param_counts, medium_dataset_tokens)
loss_large_data = hypothetical_loss(param_counts, large_dataset_tokens)

# Create the plot
plt.style.use('seaborn-v0_8-whitegrid')
fig, ax = plt.subplots(figsize=(12, 7))

ax.plot(param_counts, loss_small_data, label='Small Dataset (50B tokens)', color='red', li
```

```
ax.plot(param_counts, loss_medium_data, label='Medium Dataset (300B tokens)', color='orang
ax.plot(param_counts, loss_large_data, label='Large Dataset (1.5T tokens)', color='green'

# Formatting the plot
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_title('Visualizing LLM Scaling Laws', fontsize=16, fontweight='bold')
ax.set_xlabel('Number of Model Parameters (N)', fontsize=12)
ax.set_ylabel('Hypothetical Test Loss (Log Scale)', fontsize=12)
ax.legend(fontsize=11)
ax.grid(True, which="both", ls="-", alpha=0.5)

# Add annotations to explain the key insight
ax.annotate(
    'For a fixed model size (e.g., 100B params),\nmore data leads to lower loss.',
    xy=(1e11, hypothetical_loss(1e11, medium_dataset_tokens)),
    xytext=(1e9, 2.2),
    arrowprops=dict(facecolor='black', shrink=0.05, width=1, headwidth=8),
    fontsize=11,
    bbox=dict(boxstyle="round,pad=0.3", fc="ivory", ec="black", lw=1)
)
ax.annotate(
    'Chinchilla Insight:\nA smaller model with more data\ncan outperform a larger model\nw
    xy=(7e10, hypothetical_loss(7e10, large_dataset_tokens)),
    xytext=(5e10, 5),
    arrowprops=dict(facecolor='black', shrink=0.05, width=1, headwidth=8),
    fontsize=11,
    bbox=dict(boxstyle="round,pad=0.3", fc="ivory", ec="black", lw=1)
)


plt.show()
```