Direct Preference Optimization for Diffusion Language Models

Chi tiết mở rộng

1. Giới thiệu và Động lực

Bối cảnh

Large Language Diffusion Models (LLDMs) đang nổi lên như một giải pháp thay thế đầy hứa hẹn cho mô hình tự hồi quy truyền thống. Tuy nhiên, việc tối ưu hóa các mô hình này gặp phải thách thức đáng kể, đặc biệt là khi áp dụng các kỹ thuật Reinforcement Learning (RL) do bản chất không phải Markov của quá trình khuếch tán. Trong khi đó, Direct Preference Optimization (DPO) đã chứng minh hiệu quả trong việc tối ưu hóa mô hình ngôn ngữ tự hồi quy mà không cần các phương pháp RL phức tạp.

Vấn đề nghiên cứu

Làm thế nào để điều chỉnh DPO cho mô hình khuếch tán ngôn ngữ để cải thiện khả năng reasoning trong điều kiện tài nguyên hạn chế (GPU RTX 4080)?

Tầm quan trọng

- Giải quyết trực tiếp thách thức áp dụng RL cho mô hình khuếch tán
- Cung cấp phương pháp tối ưu hóa hiệu quả về mặt tính toán
- Cải thiện khả năng reasoning của LLDMs
- Mở rộng khả năng tiếp cận của nghiên cứu LLDMs cho cộng đồng rộng hơn

2. Nền tảng kỹ thuật

Direct Preference Optimization (DPO) truyền thống

DPO là một phương pháp tối ưu hóa mô hình ngôn ngữ dựa trên sở thích của con người mà không cần mô hình hóa phần thưởng rõ ràng. Phương pháp này chuyển đổi dữ liệu sở thích thành một hàm mất mát có thể tối ưu hóa trực tiếp.

Trong DPO truyền thống cho mô hình tự hồi quy, hàm mất mát được định nghĩa như sau:

```
L_DPO(θ) = -E_{(x,y_w,y_l)~D} [log σ(β(log p_θ(y_w|x) - log p_θ(y_l|x) - log p_θ_ref(y_w|x) + log p_θ_ref(y_l|x)))]
```

Trong đó: - θ là tham số mô hình - (x,y_w,y_l) là bộ ba gồm đầu vào x, đầu ra được ưa thích y_w và đầu ra không được ưa thích y_l - p_θ(y|x) là xác suất của đầu ra y cho đầu vào x theo mô hình với tham số θ - p_θ_ref(y|x) là xác suất theo mô hình tham chiếu - β là tham số nhiệt độ - σ là hàm logistic

Thách thức khi áp dụng cho mô hình khuếch tán

1. **Tính toán xác suất**: Trong mô hình khuếch tán, việc tính toán xác suất p_θ(y|x) phức tạp hơn nhiều so với mô hình tự hồi quy
2. **Quy trình nhiều bước**: Quá trình khuếch tán liên quan đến nhiều bước, làm phức tạp việc quy trách nhiệm
3. **Yêu cầu bộ nhớ**: Tính toán gradient qua nhiều bước khuếch tán đòi hỏi bộ nhớ đáng kể
4. **Bản chất không phải Markov**: Mỗi bước khuếch tán phụ thuộc vào toàn bộ lịch sử trước đó

3. Phương pháp Diffusion-DPO

Khung lý thuyết

Chúng tôi đề xuất một khung lý thuyết mới để điều chỉnh DPO cho mô hình khuếch tán ngôn ngữ. Khung này dựa trên các nguyên tắc sau:

1. **Biểu diễn sở thích trong không gian khuếch tán**: Phát triển phương pháp để biểu diễn sở thích của con người trong bối cảnh quá trình khuếch tán nhiều bước

2. **Xác suất khuếch tán**: Định nghĩa lại cách tính xác suất p_θ(y|x) thông qua quá trình khuếch tán

3. **Tối ưu hóa bước-cụ thể**: Phát triển kỹ thuật để tối ưu hóa các tham số cụ thể cho từng bước khuếch tán

4. **Cân bằng khả năng và sở thích**: Thiết kế hàm mất mát cân bằng giữa khả năng sinh và sự phù hợp với sở thích

Hàm mất mát Diffusion-DPO

Chúng tôi định nghĩa lại hàm mất mát DPO cho bối cảnh khuếch tán:

$$L_{DiffDPO}(\theta) = -E_{(x,y_w,y_l)\sim D} [\log \sigma(\beta(S_\theta(x,y_w) - S_\theta(x,y_l) - S_{\theta_{ref}}(x,y_w) + S_{\theta_{ref}}(x,y_l)))]$$

Trong đó: - $S_\theta(x,y)$ là điểm số khuếch tán, được định nghĩa là tổng log xác suất qua tất cả các bước khuếch tán - Các thành phần khác tương tự như trong DPO truyền thống

Tính toán điểm số khuếch tán

Điểm số khuếch tán $S_\theta(x,y)$ được tính như sau:

$$S_\theta(x,y) = \Sigma_{t=1}^{T} \log p_\theta(y_t \mid y_{t+1}, x)$$

Trong đó: - T là tổng số bước khuếch tán - y_t là trạng thái tại bước khuếch tán t - $p_\theta(y_t \mid y_{t+1}, x)$ là xác suất chuyển đổi từ y_{t+1} sang y_t

Tối ưu hóa hiệu quả về bộ nhớ

Để giải quyết vấn đề bộ nhớ, chúng tôi đề xuất:

1. **Gradient checkpointing**: Lưu trữ chỉ một tập con các trạng thái trung gian và tính toán lại khi cần thiết
2. **Xấp xỉ điểm số**: Sử dụng một tập con các bước khuếch tán để ước tính điểm số tổng thể
3. **Phân đoạn tính toán**: Chia nhỏ quá trình tính toán thành các phần có thể quản lý được

4. Kiến trúc và triển khai

Kiến trúc tổng thể

Hệ thống Diffusion-DPO bao gồm các thành phần sau:

1. **Mô hình khuếch tán cơ sở**: Mô hình LLDM đã được pre-trained
2. **Mô hình tham chiếu**: Bản sao đóng băng của mô hình cơ sở
3. **Bộ mã hóa sở thích**: Module chuyên biệt để mã hóa sở thích
4. **Bộ tối ưu hóa nhận biết tài nguyên**: Điều chỉnh quá trình tối ưu hóa dựa trên tài nguyên có sẵn

Chi tiết triển khai

Mã giả cho thuật toán đào tạo

```
def train_diffusion_dpo(base_model, preference_dataset, config):
    # Tạo mô hình tham chiếu
    reference_model = copy.deepcopy(base_model)
    reference_model.requires_grad_(False)

    # Khởi tạo bộ tối ưu hóa
    optimizer = create_optimizer(base_model, config)

    # Khởi tạo bộ giám sát tài nguyên
    resource_monitor = ResourceMonitor(gpu_type="RTX 4080")

    for epoch in range(config.max_epochs):
        for batch in preference_dataset:
            inputs, preferred, dispreferred = batch

            # Điều chỉnh số bước khuếch tán dựa trên tài nguyên
            num_steps = resource_monitor.get_optimal_steps()

            # Tính điểm số khuếch tán cho các đầu ra được ưa thích
            preferred_score = compute_diffusion_score(
                base_model, inputs, preferred, num_steps)

            # Tính điểm số khuếch tán cho các đầu ra không được ưa thích
            dispreferred_score = compute_diffusion_score(
                base_model, inputs, dispreferred, num_steps)

            # Tính điểm số tham chiếu (không gradient)
            with torch.no_grad():
                ref_preferred_score = compute_diffusion_score(
                    reference_model, inputs, preferred, num_steps)
                ref_dispreferred_score = compute_diffusion_score(
                    reference_model, inputs, dispreferred, num_steps)

            # Tính logits
            logits = preferred_score - dispreferred_score - ref_preferred_score +
ref_dispreferred_score

            # Tính mất mát DPO
            loss = -F.logsigmoid(config.beta * logits).mean()

            # Tối ưu hóa với gradient checkpointing
            with torch.cuda.amp.autocast(enabled=config.use_mixed_precision):
                loss.backward()

            # Cập nhật tham số
            optimizer.step()
            optimizer.zero_grad()
```

```
        # Cập nhật bộ giám sát tài nguyên
        resource_monitor.update()

        # Điều chỉnh cấu hình nếu cần
        if resource_monitor.is_memory_constrained():
            adjust_training_config(config)
```

Tính toán điểm số khuếch tán

```
def compute_diffusion_score(model, inputs, outputs, num_steps):
    # Khởi tạo với đầu ra nhiễu
    x_T = add_noise(outputs, noise_level=1.0)

    # Lưu trữ các trạng thái trung gian cho gradient checkpointing
    intermediate_states = []

    # Quá trình khử nhiễu
    x_t = x_T
    total_score = 0

    for t in reversed(range(num_steps)):
        # Lưu trạng thái hiện tại nếu cần thiết cho gradient checkpointing
        if t % config.checkpoint_interval == 0:
            intermediate_states.append(x_t.detach().clone())

        # Tính tỷ lệ nhiễu
        noise_level = t / num_steps

        # Dự đoán
        pred = model(inputs, x_t, noise_level)

        # Tính điểm số cho bước hiện tại
        step_score = compute_step_score(pred, outputs, noise_level)
        total_score += step_score

        # Cập nhật x_t
        x_t = denoise_step(x_t, pred, noise_level)

    return total_score
```

Điều chỉnh cấu hình dựa trên tài nguyên

```
def adjust_training_config(config):
    # Giảm kích thước batch nếu cần
    if memory_usage > 0.95 * total_memory:
        config.batch_size = max(1, config.batch_size // 2)
        print(f"Adjusted batch size to {config.batch_size}")
```

```
    # Điều chỉnh số bước khuếch tán
    if memory_usage > 0.9 * total_memory:
        config.num_steps = max(10, config.num_steps - 5)
        print(f"Adjusted diffusion steps to {config.num_steps}")

    # Bật mixed precision nếu cần
    if memory_usage > 0.85 * total_memory and not config.use_mixed_precision:
        config.use_mixed_precision = True
        print("Enabled mixed precision training")
```

5. Dữ liệu sở thích cho nhiệm vụ reasoning

Thiết kế dữ liệu sở thích

Chúng tôi đề xuất một phương pháp có hệ thống để tạo dữ liệu sở thích tập trung vào khả năng reasoning:

1. **Phân loại nhiệm vụ reasoning**:
2. Suy luận toán học (ví dụ: GSM8K, MATH)
3. Suy luận logic (ví dụ: LogiQA, ReClor)
4. Suy luận thường thức (ví dụ: PIQA, HellaSwag)
5. Suy luận đa bước (ví dụ: HotpotQA)
6. **Tạo cặp sở thích**:
7. **Đầu ra chính xác vs. sai**: Cặp gồm đầu ra đúng và đầu ra có lỗi reasoning
8. **Đầu ra đầy đủ vs. không đầy đủ**: Cặp gồm chuỗi reasoning đầy đủ và chuỗi bị cắt ngắn
9. **Đầu ra rõ ràng vs. mơ hồ**: Cặp gồm reasoning rõ ràng và reasoning mơ hồ
10. **Đầu ra hiệu quả vs. không hiệu quả**: Cặp gồm reasoning ngắn gọn và reasoning dài dòng
11. **Phương pháp thu thập**:
12. Sử dụng mô hình ngôn ngữ lớn để tạo các đầu ra đa dạng
13. Thu thập đánh giá từ chuyên gia cho các cặp đầu ra
14. Tự động tạo biến thể của đầu ra chính xác với lỗi reasoning cụ thể

Ví dụ dữ liệu sở thích

Ví dụ cho nhiệm vụ suy luận toán học (GSM8K):

Đầu vào:

```
Một cửa hàng bán 5 loại bánh khác nhau. Mỗi loại có 8 chiếc. Nếu 3 khách hàng mua tổng cộng 10 chiếc bánh, thì số bánh còn lại trong cửa hàng là bao nhiêu?
```

Đầu ra được ưa thích (reasoning đầy đủ và chính xác):

```
Số bánh ban đầu = 5 loại × 8 chiếc = 40 chiếc
Số bánh đã bán = 10 chiếc
Số bánh còn lại = 40 - 10 = 30 chiếc
```

Đầu ra không được ưa thích (reasoning không đầy đủ):

```
5 × 8 = 40
40 - 10 = 30
```

Đầu ra không được ưa thích (reasoning sai):

```
Số bánh ban đầu = 5 loại × 8 chiếc = 40 chiếc
Số bánh mỗi khách hàng mua = 10 ÷ 3 ≈ 3.33 chiếc
Số bánh còn lại = 40 - (3 × 3.33) = 40 - 10 = 30 chiếc
```

6. Tối ưu hóa cho GPU RTX 4080

Phân tích tài nguyên

RTX 4080 có khoảng 16GB VRAM, tạo ra các ràng buộc đáng kể cho việc fine-tuning LLDMs. Chúng tôi đề xuất các kỹ thuật sau để tối ưu hóa việc sử dụng tài nguyên:

Kỹ thuật tối ưu hóa bộ nhớ

1. **Gradient checkpointing**: Lưu trữ chỉ một tập con các trạng thái trung gian và tính toán lại khi cần thiết
2. **Mixed precision training**: Sử dụng FP16/BF16 cho hầu hết các phép tính
3. **Gradient accumulation**: Tích lũy gradient qua nhiều mini-batch trước khi cập nhật
4. **Selective layer freezing**: Đóng băng các lớp thấp hơn của mô hình để giảm yêu cầu bộ nhớ
5. **Offloading**: Di chuyển các tham số không hoạt động sang CPU khi không sử dụng

Tối ưu hóa tốc độ tính toán

1. **Kernel fusion**: Kết hợp các phép tính để giảm chi phí đồng bộ hóa

2. **Flash attention**: Sử dụng các thuật toán attention hiệu quả về bộ nhớ
3. **Adaptive step selection**: Điều chỉnh số bước khuếch tán dựa trên độ phức tạp của đầu vào
4. **Batch size optimization**: Tìm kích thước batch tối ưu cho hiệu suất

Cấu hình đề xuất cho RTX 4080

```
config = {
    # Cấu hình mô hình
    "model_size": "3B",  # Phù hợp với RTX 4080
    "hidden_dim": 2048,

    # Cấu hình tối ưu hóa
    "batch_size": 4,  # Bắt đầu nhỏ, điều chỉnh dựa trên sử dụng bộ nhớ
    "gradient_accumulation_steps": 8,  # Tích lũy qua 8 bước
    "mixed_precision": True,  # Sử dụng FP16/BF16
    "gradient_checkpointing": True,

    # Cấu hình khuếch tán
    "initial_diffusion_steps": 20,  # Bắt đầu với 20 bước, điều chỉnh động
    "min_diffusion_steps": 10,
    "checkpoint_interval": 5,  # Lưu trạng thái mỗi 5 bước

    # Cấu hình DPO
    "beta": 0.1,  # Tham số nhiệt độ cho DPO
    "reference_free": False,  # Sử dụng mô hình tham chiếu

    # Cấu hình tài nguyên
    "max_memory_usage": 0.9,  # Sử dụng tối đa 90% VRAM
    "enable_cpu_offloading": True
}
```

7. Thử nghiệm và đánh giá

Thiết lập thử nghiệm

Chúng tôi đề xuất một kế hoạch thử nghiệm toàn diện để đánh giá hiệu quả của Diffusion-DPO:

1. **Mô hình**:
2. LLDM cơ sở (1B, 3B tham số)
3. Mô hình fine-tuned với Diffusion-DPO

4. Mô hình fine-tuned với phương pháp truyền thống

5. **Tập dữ liệu**:

6. GSM8K (suy luận toán học)
7. LogiQA (suy luận logic)
8. HotpotQA (suy luận đa bước)
9. MATH (suy luận toán học nâng cao)
10. **Baseline**:
11. Fine-tuning tiêu chuẩn
12. RL truyền thống (nếu khả thi)
13. DPO cho mô hình tự hồi quy

Phương pháp đánh giá

Chúng tôi sẽ đánh giá các mô hình dựa trên các tiêu chí sau:

1. **Hiệu suất reasoning**:
2. Độ chính xác trên các tập dữ liệu benchmark
3. Chất lượng chuỗi reasoning (đánh giá bởi con người)
4. Khả năng giải quyết vấn đề phức tạp
5. **Hiệu quả tính toán**:
6. Thời gian đào tạo
7. Sử dụng bộ nhớ
8. Số bước khuếch tán cần thiết
9. **Khả năng mở rộng**:
10. Hiệu suất trên các kích thước mô hình khác nhau
11. Khả năng chuyển giao sang các miền mới

Phân tích ablation

Chúng tôi sẽ tiến hành các nghiên cứu ablation để đánh giá đóng góp của từng thành phần:

1. **Tác động của dữ liệu sở thích**: So sánh hiệu suất với các loại dữ liệu sở thích khác nhau
2. **Tác động của số bước khuếch tán**: Đánh giá hiệu suất với số bước khác nhau
3. **Tác động của tham số β**: Phân tích độ nhạy với tham số nhiệt độ
4. **Tác động của kỹ thuật tối ưu hóa bộ nhớ**: Đánh giá hiệu quả của các kỹ thuật khác nhau

8. Kế hoạch triển khai

Lộ trình nghiên cứu

Chúng tôi đề xuất lộ trình 6 tháng cho nghiên cứu này:

1. **Tháng 1**: Phát triển khung lý thuyết và thu thập dữ liệu sở thích
2. Xây dựng nền tảng toán học cho Diffusion-DPO
3. Thiết kế và thu thập dữ liệu sở thích cho nhiệm vụ reasoning

4. Chuẩn bị cơ sở hạ tầng thử nghiệm

5. **Tháng 2**: Triển khai thuật toán Diffusion-DPO cơ bản

6. Phát triển mã nguồn cho Diffusion-DPO
7. Triển khai kỹ thuật tối ưu hóa bộ nhớ

8. Thử nghiệm ban đầu trên tập dữ liệu nhỏ

9. **Tháng 3**: Tiến hành thử nghiệm sơ bộ và tinh chỉnh phương pháp

10. Đánh giá hiệu suất ban đầu
11. Tinh chỉnh thuật toán dựa trên kết quả

12. Mở rộng thử nghiệm sang nhiều tập dữ liệu hơn

13. **Tháng 4**: Tiến hành đánh giá toàn diện và nghiên cứu ablation

14. Thực hiện thử nghiệm đầy đủ trên tất cả các tập dữ liệu
15. Tiến hành nghiên cứu ablation

16. Thu thập kết quả và phân tích

17. **Tháng 5**: Phân tích kết quả và chuẩn bị bản thảo

18. Phân tích kết quả thử nghiệm
19. Viết bản thảo

20. Chuẩn bị mã nguồn để phát hành

21. **Tháng 6**: Hoàn thiện paper và chuẩn bị mã nguồn để phát hành mã nguồn mở

22. Hoàn thiện bản thảo
23. Chuẩn bị mã nguồn để phát hành
24. Tạo tài liệu và hướng dẫn sử dụng

Yêu cầu tài nguyên

- **Phần cứng**: GPU RTX 4080 (đã có)
- **Phần mềm**: PyTorch, Hugging Face Transformers, Accelerate
- **Dữ liệu**: Tập dữ liệu benchmark và dữ liệu sở thích tự thu thập

9. Tác động và ý nghĩa

Đóng góp kỹ thuật

- Thuật toán mới cho tối ưu hóa sở thích trong mô hình khuếch tán
- Khung toán học cho học tập sở thích trong bối cảnh không phải Markov
- Triển khai hiệu quả về tài nguyên tương thích với phần cứng tiêu dùng
- Mã nguồn mở và tập dữ liệu sở thích

Tác động thực tế

- Cho phép fine-tuning dựa trên sở thích mà không cần phức tạp của RL
- Cải thiện khả năng reasoning của mô hình khuếch tán ngôn ngữ
- Giảm yêu cầu tính toán cho các kỹ thuật alignment
- Làm cho tối ưu hóa dựa trên sở thích trở nên dễ tiếp cận hơn với cộng đồng rộng hơn

Ý nghĩa rộng hơn

- Thúc đẩy kỹ thuật alignment cho hệ thống AI dựa trên khuếch tán
- Cung cấp các giải pháp thay thế cho RL để alignment mô hình
- Cho phép ứng dụng mới của LLDMs đã alignment trong môi trường tài nguyên hạn chế

10. Kết luận

Nghiên cứu đề xuất giải quyết một thách thức quan trọng trong việc tối ưu hóa Large Language Diffusion Models cho các nhiệm vụ reasoning mà không dựa vào phương pháp RL truyền thống. Bằng cách điều chỉnh Direct Preference Optimization cho bối cảnh khuếch tán, công trình này nhằm cải thiện alignment của mô hình với sở thích của con người trong khi làm việc trong các ràng buộc của phần cứng tiêu dùng. Phương pháp này giải quyết trực tiếp các hạn chế của việc áp dụng RL cho quá trình khuếch tán không phải Markov và cung cấp một con đường hiệu quả hơn để cải thiện khả năng reasoning trong các mô hình này.