МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

РОЗРОБКА ТА ОЦІНКА НАЛАШТОВАНИХ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторної роботи № 6
з дисципліни «Штучний інтелект в ігровому дизайні»
для студентів бакалаврського рівня вищої освіти спеціальності 121
"Інженерія програмного забезпечення"

Львів -- 2025

Розробка та оцінка налаштованих великих мовних моделей: методичні вказівки до виконання лабораторної роботи №6 з дисципліни "Штучний інтелект в ігровому дизайні" для студентів першого (бакалаврського) рівня вищої освіти спеціальності 121 "Інженерія програмного забезпечення" . Укл.: О.Є. Бауск. -- Львів: Видавництво Національного університету "Львівська політехніка", 2025. -- 10 с.

Укладач: Бауск О.Є., к.т.н., асистент кафедри ПЗ

Відповідальний за випуск: Федасюк Д.В., доктор техн. наук, професор

Рецензенти: Федасюк Д.В., доктор техн. наук, професор

Задорожний І.М., асистент кафедри ПЗ

Тема роботи: Розробка та оцінка налаштованих великих мовних моделей.

Мета роботи: Набути практичних навичок з налаштування великих мовних моделей (LLM), засвоїти основні концепції токенізації, шаблонів запитів та оцінки якості мовних моделей. Дослідити повний конвеєр для налаштування мовної моделі для генерації відповідей у певному стилі та вивчити способи оцінки продуктивності мовної моделі.

Теоретичні відомості

Великі мовні моделі (LLM)

Великі мовні моделі (Large Language Models, LLM) — це нейронні мережі з мільярдами або навіть трильйонами параметрів, навчені на величезних обсягах тексту. Ці моделі здатні генерувати текст, перекладати, відповідати на запитання та виконувати інші завдання, пов'язані з обробкою природної мови.

Ключові особливості LLM:

- **Масштаб**: Чим більше параметрів має модель, тим краще вона зазвичай працює.
- **Архітектура Трансформер**: Більшість сучасних LLM базуються на архітектурі Трансформер, запропонованій у статті "Attention is All You Need" (2017).
- **Самонавчання**: LLM навчаються передбачати наступне слово або токен у тексті, не потребуючи розмічених даних.
- **Виникаючі здібності**: З ростом розміру моделі виникають нові здібності, які не були явно закладені при навчанні.

Токенізація

Токенізація — це процес перетворення тексту в послідовність числових ідентифікаторів (токенів), які модель може обробляти. Цей процес є критично важливим для роботи LLM.

Основні аспекти токенізації:

- **Словник токенів**: Кожна модель має свій фіксований словник, що містить слова, частини слів або окремі символи.
- **Субслівна токенізація**: Більшість сучасних токенізаторів розбивають слова на менші частини, що дозволяє ефективно обробляти невідомі слова.
- **Спеціальні токени**: Токенізатори використовують спеціальні токени для позначення початку та кінця тексту, розділення різних частин запиту тощо.
- **Контекстне вікно**: LLM мають обмеження на максимальну довжину послідовності токенів, яку вони можуть обробляти (контекстне вікно).

Шаблони запитів (Prompt Templates)

Шаблони запитів — це структуровані формати для подання вхідних даних у LLM. Вони визначають, як різні компоненти запиту (система, користувач, асистент) повинні бути організовані.

Ключові особливості шаблонів запитів:

- **Системний запит**: Встановлює загальні інструкції та контекст для моделі.
- **Запит користувача**: Безпосереднє питання або завдання.
- **Відповідь асистента**: Місце, де модель генерує свою відповідь.
- **Формат чату**: Багато сучасних LLM використовують формат чату, де різні ролі (система, користувач, асистент) чергуються.

Налаштування моделей (Fine-tuning)

Налаштування — це процес адаптації попередньо навченої моделі до конкретного завдання шляхом додаткового навчання на спеціалізованому наборі даних. Для LLM це дозволяє покращити продуктивність у конкретних доменах або стилях.

Методи налаштування LLM:

- **Повне налаштування**: Оновлення всіх параметрів моделі, що вимагає значних обчислювальних ресурсів.
- **Ефективне налаштування параметрів**: Методи, які оновлюють лише невелику частину параметрів.
  - **LoRA (Low-Rank Adaptation)**: Додавання невеликих низькорангових матриць до ваг моделі.
  - **QLoRA**: Комбінація LoRA з квантизацією для ще більшої ефективності.
  - **Prefix Tuning**: Додавання навчених префіксів до кожного шару моделі.

Оцінка мовних моделей

Оцінка якості мовних моделей — складне завдання, що включає як автоматичні метрики, так і людську оцінку.

Основні підходи до оцінки:

- **Перплексія (Perplexity)**: Міра того, наскільки добре модель передбачає текст. Нижча перплексія означає кращу продуктивність.
- **Автоматичні метрики**: BLEU, ROUGE, BERTScore та інші, які порівнюють згенерований текст з еталонними відповідями.
- **Автоматична оцінка за допомогою LLM**: Використання інших LLM для оцінки якості відповідей.
- **Людська оцінка**: Експерти або краудсорсери оцінюють якість відповідей за різними критеріями.

Gemma 2B

Gemma 2B — це відкрита мовна модель від Google з приблизно 2 мільярдами параметрів. Це полегшена версія сімейства моделей Gemma, розроблена для дослідницьких цілей та застосування на менш потужному обладнанні.

Ключові особливості Gemma 2B:

- **Ефективність**: Оптимізована для роботи на обмежених обчислювальних ресурсах.
- **Відкритий доступ**: Модель доступна для дослідницьких та комерційних цілей.
- **Безпека**: Розроблена з урахуванням вимог безпеки та етичних стандартів.

- **Інструкційне налаштування**: Версія "gemma-2b-it" пройшла інструкційне налаштування для покращення продуктивності в діалогах.

LFM-40B

LFM-40B — це велика мовна модель від Liquid AI з 40 мільярдами параметрів. У цій лабораторній роботі ми використовуватимемо її як "суддю" для оцінки якості генерації тексту нашої налаштованої моделі.

Ключові особливості LFM-40B:

- **Висока продуктивність**: Має значно більше параметрів, ніж Gemma 2B, що дозволяє їй краще розуміти контекст та нюанси.
- **Оцінка якості**: Можливість використання як "судді" для оцінки генерації інших моделей.
- **Мультизадачність**: Здатність виконувати різноманітні завдання з обробки природної мови.

Comet ML Opik

Opik — це фреймворк від Comet ML для спрощеної оцінки LLM. Він дозволяє оцінювати якість генерації моделей за різними аспектами без необхідності розробки власних метрик та інфраструктури.

Ключові особливості Opik:

- **Уніфікований API**: Єдиний інтерфейс для різних методів оцінки.
- **Масштабованість**: Можливість обробляти велику кількість оцінок паралельно.
- **Інтеграція з LLM**: Підтримка використання зовнішніх LLM для автоматизованої оцінки.
- **Гнучкість**: Можливість визначення власних метрик та критеріїв оцінки.

Хід роботи

1. Підготовка середовища

1.1. Встановлення необхідних бібліотек:

```
# Встановлення MIT Deep Learning утиліт
!pip install mitdeeplearning > /dev/null 2>&1
import mitdeeplearning as mdl

# Імпорт необхідних бібліотек
import os
import json
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

import torch
from torch.nn import functional as F
from torch.utils.data import DataLoader

from transformers import AutoTokenizer, AutoModelForCausalLM
```

```
from datasets import load_dataset
from peft import LoraConfig, get_peft_model
from lion_pytorch import Lion
```

1.2. Налаштування середовища Google Colab:

Ця лабораторна робота розроблена для виконання в середовищі Google Colab. Переконайтеся, що ви використовуєте GPU для прискорення обчислень:

1. В меню виберіть "Runtime" -> "Change runtime type"
2. Встановіть "Hardware accelerator" на "GPU"
3. Натисніть "Save"

2. Налаштування мовної моделі для стилю

2.1. Розуміння структури шаблону запиту

Сучасні LLM зазвичай використовують структурований формат запиту, що складається з різних ролей (система, користувач, асистент). Визначимо шаблони для нашої роботи:

```
# Базовий шаблон розмови
template = """<bos><start_of_turn>system
{system}<end_of_turn>

<start_of_turn>user
{user}<end_of_turn>

<start_of_turn>model
"""

# Шаблон з відповіддю (для тренування)
template_with_answer = """<bos><start_of_turn>system
{system}<end_of_turn>

<start_of_turn>user
{user}<end_of_turn>

<start_of_turn>model
{answer}<end_of_turn>
"""

# Системний запит
system_message = "You are a helpful AI assistant."
```

2.2. Робота з токенізацією

Щоб зрозуміти, як LLM обробляє текст, важливо розуміти токенізацію. Завантажимо модель Gemma 2B та її токенізатор:

```
# Завантаження токенізатора Gemma
tokenizer = AutoTokenizer.from_pretrained("google/gemma-2b-it")
```

```
# Завантаження моделі Gemma
model = AutoModelForCausalLM.from_pretrained(
    "google/gemma-2b-it",
    device_map="auto",  # Автоматично використовувати доступні пристрої
    torch_dtype=torch.float16,  # Використання половинної точності для ефективності
)
```

Розглянемо процес токенізації на прикладі:

```
# Створення запиту в форматі шаблону
prompt = template.format(
    system=system_message,
    user="What is deep learning?"
)

# Токенізація тексту
tokens = tokenizer(prompt, return_tensors="pt").to(model.device)
print(f"Кількість токенів: {tokens.input_ids.shape[1]}")

# Виведення деяких токенів
for i in range(min(10, tokens.input_ids.shape[1])):
    token_id = tokens.input_ids[0, i].item()
    token_text = tokenizer.decode([token_id])
    print(f"Токен {i}: {token_id} -> '{token_text}'")
```

2.3. Функція для генерації тексту

Визначимо функцію для генерації відповідей на запитання:

```
def chat(question, system=system_message, only_answer=False, max_new_tokens=512, temperature=0.
    # Формування запиту
    prompt = template.format(system=system, user=question)

    # Токенізація запиту
    tokens = tokenizer(prompt, return_tensors="pt").to(model.device)

    # Генерація відповіді
    with torch.no_grad():
        outputs = model.generate(
            tokens.input_ids,
            max_new_tokens=max_new_tokens,
            temperature=temperature,
            do_sample=True,
        )

    # Декодування відповіді
    full_response = tokenizer.decode(outputs[0], skip_special_tokens=True)

    # Виділення лише відповіді моделі, якщо потрібно
    if only_answer:
        answer_prefix = "<start_of_turn>model\n"
        answer_start = full_response.find(answer_prefix)
        if answer_start != -1:
            answer_start += len(answer_prefix)
```

```
            answer_end = full_response.find("<end_of_turn>", answer_start)
            if answer_end != -1:
                return full_response[answer_start:answer_end].strip()

    return full_response
```

2.4. Завантаження набору даних для налаштування

Для налаштування моделі на генерацію відповідей у стилі лепрекона (персонажа ірландського фольклору), використаємо підготовлений набір даних:

```
# Завантаження набору даних для налаштування
train_loader, test_loader = mdl.lab3.create_dataloader(style="leprechaun")

# Перегляд прикладу з набору даних
sample = next(iter(train_loader))
print(f"Запитання: {sample['instruction'][0]}")
print(f"Відповідь (стандартна): {sample['response'][0]}")
print(f"Відповідь (лепрекон): {sample['response_style'][0]}")
```

2.5. Налаштування моделі з використанням LoRA

Повне налаштування великої мовної моделі потребує значних обчислювальних ресурсів. Натомість, ми використаємо метод LoRA (Low-Rank Adaptation) для ефективного налаштування:

```
# Застосування LoRA до моделі
def apply_lora(model):
    # Визначення конфігурації LoRA
    lora_config = LoraConfig(
        r=8,  # ранг матриць LoRA
        task_type="CAUSAL_LM",
        target_modules=[
            "q_proj", "o_proj", "k_proj", "v_proj", "gate_proj", "up_proj", "down_proj"
        ],
    )

    # Застосування LoRA до моделі
    lora_model = get_peft_model(model, lora_config)
    return lora_model

model = apply_lora(model)

# Виведення кількості параметрів для навчання
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
total_params = sum(p.numel() for p in model.parameters())
print(f"Кількість параметрів для навчання: {trainable_params}")
print(f"Загальна кількість параметрів: {total_params}")
print(f"Відсоток параметрів для навчання: {trainable_params / total_params * 100:.2f}%")
```

2.6. Обчислення функції втрат

Для навчання моделі потрібно визначити функцію втрат, яка оцінює, наскільки прогнози моделі відрізняються від справжніх значень:

```
def forward_and_compute_loss(model, tokens, mask, context_length=512):
    # Обмеження до довжини контексту
    tokens = tokens[:, :context_length]
    mask = mask[:, :context_length]

    # Формування входу, виходу та маски
    x = tokens[:, :-1]
    y = tokens[:, 1:]
    mask = mask[:, 1:]

    # Прямий прохід для обчислення логітів
    logits = model(x).logits

    # Обчислення втрат
    loss = F.cross_entropy(
        logits.view(-1, logits.size(-1)),
        y.view(-1),
        reduction="none"
    )

    # Маскування втрат для токенів, що не є частиною відповіді
    loss = loss[mask.view(-1)].mean()

    return loss
```

2.7. Цикл навчання

Тепер визначимо функцію для навчання моделі:

```
def train(model, dataloader, tokenizer, max_steps=200, context_length=512, learning_rate=1e-4):
    losses = []

    # Застосування LoRA до моделі
    model = apply_lora(model)

    # Ініціалізація оптимізатора
    optimizer = Lion(model.parameters(), lr=learning_rate)

    # Цикл навчання
    for step, batch in enumerate(dataloader):
        question = batch["instruction"][0]
        answer = batch["response_style"][0]

        # Форматування питання і відповіді в шаблон
        text = template_with_answer.format(
            system=system_message,
            user=question,
            answer=answer
        )

        # Токенізація тексту та обчислення маски для відповіді
```

```
        ids = tokenizer(text, return_tensors="pt", return_offsets_mapping=True).to(model.device
        mask = ids["offset_mapping"][:,:,0] >= text.index(answer)

        # Передача токенів через модель та обчислення втрат
        loss = forward_and_compute_loss(model, ids.input_ids, mask, context_length)

        # Зворотний прохід
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

        # Моніторинг прогресу
        if step % 10 == 0:
            print(chat("What is the capital of France?", only_answer=True))
            print(f"Крок {step} втрати: {torch.mean(torch.tensor(losses)).item()}")
            losses = []

        if step > 0 and step % max_steps == 0:
            break

    return model
```

Запустимо процес навчання:

```
# Навчання моделі
model = train(model, train_loader, tokenizer, max_steps=200)

# Тестування налаштованої моделі
answer = chat(
    "What is the capital of Ireland?",
    only_answer=True,
    max_new_tokens=32,
)
print(answer)
```

3. Оцінка налаштованої моделі

3.1. Підготовка "судді" для оцінки

Для об'єктивної оцінки якості генерації тексту нашою моделлю, використаємо LFM-40B як "суддю":

```
# Ініціалізація "судді"
llm = mdl.lab3.LLMClient("liquid-40b")

# Системний запит для "судді"
system_prompt = """
You are an expert judge who evaluates the style of text. You will be presented with a text, and

Leprechauns speak with an Irish accent and use Irish slang and expressions. They often mention

Rate the text on a scale from 0 to 10, where:
0 = Not at all like a leprechaun, standard formal English
```

```
10 = Perfect leprechaun speech, authentic Irish folklore style

Output your rating as a JSON dictionary of the form: {"score": <score between 0 and 10>}
"""
```

3.2. Створення класу оцінювача

Створимо клас для оцінки згенерованого тексту за допомогою "судді":

```python
from opik.evaluation.metrics import base_metric, score_result

class LLMJudgeEvaluator(base_metric.BaseMetric):
    def __init__(self, judge: mdl.lab3.LLMClient = None, system_prompt: str = None):
        self.judge = judge
        self.system_prompt = system_prompt
        self.prompt_template = "Evaluate this text: {text}"

    def score(self, text: str, n_tries=20, **kwargs):
        """ Оцінка тексту за допомогою LLM. """

        for attempt in range(n_tries):
            try:
                # Форматування запиту перед передачею "судді"
                prompt = self.prompt_template.format(text=text)

                # Виклик "судді" LLM із системним запитом та шаблоном запиту
                res = self.judge.ask(
                  system=self.system_prompt,
                  user=prompt,
                  max_tokens=50,
                  stop=["}"]
                )

                # Витягнення вмісту відповіді з API
                res = res.choices[0].message.content + "}"
                res_dict = json.loads(res)

                max_score = 10  # Максимальна оцінка, яку має видавати LLM
                score = res_dict["score"] / max_score  # Нормалізація
                score = max(0.0, min(score, 1.0))  # Обмеження між 0 та 1

                # Повернення об'єкту оцінки
                return score_result.ScoreResult(name="StyleScore", value=score)

            except Exception as e:
                if attempt == n_tries - 1:  # Остання спроба
                    raise e  # Повторне викидання винятку, якщо всі спроби невдалі
                continue  # Спробувати знову, якщо не остання спроба
```

Ініціалізуємо оцінювача:

```python
judge = LLMJudgeEvaluator(llm, system_prompt=system_prompt)
```

3.3. Оцінка моделі за допомогою "судді"

Тепер оцінимо згенерований текст:

```
def scoring_function(text):
    return judge.score(text).value

# Тестові тексти для перевірки "судді"
test_texts = [
    "Hello, I am a standard language model.",
    "Top o' the mornin' to ya, me lad! May the luck o' the Irish be with ya today!",
    "Aye, me pot o' gold is hidden at the end o' the rainbow, to be sure!"
]

for text in test_texts:
    score = scoring_function(text)
    print(f"{text} ==> Оцінка: {score}")
```

3.4. Генерація зразків та оцінка

Згенеруємо відповіді моделі на тестові запитання та оцінимо їх:

```
# Генерація зразків з тестового набору даних
def generate_samples_from_test(test_loader, num_samples):
    samples = []
    for test_sample in tqdm(test_loader, total=num_samples):
        test_question = test_sample['instruction'][0]
        with torch.no_grad():
            generated = chat(test_question, only_answer=True, max_new_tokens=100)
        samples.append(generated)
        if len(samples) >= num_samples:
            break
    return samples

# Генерація зразків
n_samples = 20
generated_samples = generate_samples_from_test(test_loader, num_samples=n_samples)

# Збір зразків у стандартному стилі та стилі лепрекона з тренувального набору
base_samples = [sample['response'][0] for i, sample in enumerate(train_loader) if i < n_samples
style_samples = [sample['response_style'][0] for i, sample in enumerate(train_loader) if i < n_
```

3.5. Паралельна оцінка зразків

Для ефективної оцінки, використаємо паралельне обчислення:

```
# Налаштування середовища для паралельного обчислення
os.environ["TOKENIZERS_PARALLELISM"] = "false"
from multiprocessing import Pool

def compute_scores_in_parallel(samples):
    with Pool(processes=10) as pool:
        scores = pool.map(scoring_function, samples)
```

```
    return scores

# Обчислення та виведення оцінок для тексту в стандартному стилі, згенерованого тексту та трену
base_scores = compute_scores_in_parallel(base_samples)
print(f"Базовий стиль: {np.mean(base_scores):.2f} ± {np.std(base_scores):.2f}")

generated_scores = compute_scores_in_parallel(generated_samples)
print(f"Згенерований стиль: {np.mean(generated_scores):.2f} ± {np.std(generated_scores):.2f}")

style_scores = compute_scores_in_parallel(style_samples)
print(f"Еталонний стиль: {np.mean(style_scores):.2f} ± {np.std(style_scores):.2f}")
```

3.6. Візуалізація результатів

Візуалізуємо розподіл оцінок для різних типів тексту:

```
import seaborn as sns
import pandas as pd

# Створення DataFrame для візуалізації
df = pd.DataFrame({
    'Оцінка': [*base_scores, *generated_scores, *style_samples],
    'Тип': ['Базовий']*len(base_scores) + ['Згенерований']*len(generated_scores) + ['Еталонний'
})

# Візуалізація за допомогою seaborn
sns.histplot(data=df, x='Оцінка', hue='Тип', multiple="dodge", bins=6, shrink=.8)

plt.title('Розподіл оцінок')
plt.show()
```

4. Експерименти зі стилем Йоди

4.1. Налаштування на стиль Йоди

Тепер проведемо експеримент з налаштуванням моделі на стиль Йоди (персонажа з "Зоряних війн"):

```
# Завантаження набору даних для стилю Йоди
train_loader, test_loader = mdl.lab3.create_dataloader(style="yoda")

# Навчання моделі на стилі Йоди
model = train(model, train_loader, tokenizer, max_steps=200)
```

4.2. Оцінка стилю Йоди

Для оцінки якості стилю Йоди, створимо новий системний запит для "судді":

```
# Системний запит для оцінки стилю Йоди
yoda_system_prompt = """
You are an expert judge who evaluates the style of text. You will be presented with a text, and

Yoda speaks with inverted sentence structure, often putting the object or predicate before the
```

```
Examples of Yoda-speak:
- "Fear is the path to the dark side" -> "The path to the dark side, fear is."
- "You must learn control" -> "Control, you must learn."
- "Try not. Do or do not. There is no try." -> This is already in Yoda-style.

Rate the text on a scale from 0 to 10, where:
0 = Not at all like Yoda, standard English grammar
10 = Perfect Yoda speech, authentic Star Wars style

Output your rating as a JSON dictionary of the form: {"score": <score between 0 and 10>}
"""

# Оновлення "судді" з новим системним запитом
judge = LLMJudgeEvaluator(llm, system_prompt=yoda_system_prompt)

# Тестування "судді" на прикладах стилю Йоди
test_texts = [
    "Tennis is a fun sport. But you must concentrate.",
    "Fun sport, tennis is. But work hard, you must.",
    "Hard to see, the dark side is."
]

for text in test_texts:
    score = scoring_function(text)
    print(f"{text} ==> Оцінка: {score}")
```

4.3. Обчислення оцінок для стилю Йоди

Проведемо аналогічну оцінку для моделі, налаштованої на стиль Йоди:

```
# Генерація зразків для стилю Йоди
generated_samples = generate_samples_from_test(test_loader, num_samples=n_samples)

# Збір зразків у стандартному стилі та стилі Йоди з тренувального набору
base_samples = [sample['response'][0] for i, sample in enumerate(train_loader) if i < n_samples
style_samples = [sample['response_style'][0] for i, sample in enumerate(train_loader) if i < n_

# Обчислення оцінок
base_scores = compute_scores_in_parallel(base_samples)
print(f"Базовий стиль: {np.mean(base_scores):.2f} ± {np.std(base_scores):.2f}")

generated_scores = compute_scores_in_parallel(generated_samples)
print(f"Згенерований стиль: {np.mean(generated_scores):.2f} ± {np.std(generated_scores):.2f}")

style_scores = compute_scores_in_parallel(style_samples)
print(f"Еталонний стиль: {np.mean(style_scores):.2f} ± {np.std(style_scores):.2f}")

# Візуалізація розподілу оцінок
df = pd.DataFrame({
    'Оцінка': [*base_scores, *generated_scores, *style_samples],
    'Тип': ['Базовий']*len(base_scores) + ['Згенерований']*len(generated_scores) + ['Еталонний'
})

sns.histplot(data=df, x='Оцінка', hue='Тип', multiple="dodge", bins=6, shrink=.8)
```

```
plt.title('Розподіл оцінок для стилю Йоди')
plt.show()
```

5. Обчислення правдоподібності тестового тексту

Для об'єктивного порівняння різних моделей, обчислимо правдоподібність тестового тексту:

```
# Обчислення правдоподібності тестового тексту
yoda_test_text = mdl.lab3.yoda_test_text
tokens = tokenizer(yoda_test_text, return_tensors="pt").to(model.device)

# Отримання логарифмічної правдоподібності з моделі
with torch.no_grad():
    outputs = model(**tokens)
    logits = outputs.logits[:, :-1]
    targets = tokens.input_ids[:, 1:]
    loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)),
                           targets.reshape(-1))

print(f"Логарифмічна правдоподібність тестового тексту Йоди: {loss.item():.2f}")
```

УМОВА ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

1. Налаштувати велику мовну модель Gemma 2B для генерації відповідей у стилі персонажа (лепрекона або Йоди) з використанням методу LoRA.
2. Розробити та налаштувати "суддю" LLM для оцінки якості стилізації тексту.
3. Провести експерименти з різними параметрами налаштування (learning rate, кількість кроків навчання, температура генерації) та проаналізувати їх вплив на якість генерації.
4. Оцінити якість генерації тексту за допомогою автоматичної оцінки з використанням "судді" LLM та порівняти різні підходи до генерації.
5. Розрахувати логарифмічну правдоподібність тестового тексту для об'єктивного порівняння різних налаштованих моделей.

ІНДІВІДУАЛЬНІ ВАРІАНТИ ЗАВДАННЯ

Для індивідуальних варіантів завдання оберіть один із наступних персонажів/стилів для налаштування моделі:

1. **Стиль пірата** - використання піратського сленгу, морської тематики, відповідних звертань.
2. **Стиль Шекспіра** - архаїчна англійська, поетичні вирази, високий стиль.
3. **Стиль детектива-нуар** - короткі, різкі речення, похмура атмосфера, цинічні спостереження.
4. **Стиль наукової фантастики** - технічні терміни, футуристичні концепції, посилання на космос.
5. **Стиль казки** - прості конструкції, повтори, традиційні казкові звороти.
6. **Стиль реклами** - переконливі звернення, гіперболи, заклики до дії.

7. **Стиль інструкції** - чіткі, короткі вказівки, розбиття на кроки, технічна точність.
8. **Стиль Гаррі Поттера** - використання магічної термінології, відповідні метафори.

Для обраного стилю необхідно:

1. Створити власний набір даних для навчання (мінімум 20 пар запитання-відповідь).
2. Розробити системний запит для "судді" LLM, який буде оцінювати якість стилізації.
3. Провести процес налаштування та оцінки моделі.

ЗМІСТ ЗВІТУ

1. Тема та мета роботи
2. Теоретичні відомості
3. Постановка завдання
4. Хід виконання роботи:
    - Опис обраного стилю та підготовленого набору даних
    - Системний запит для "судді" LLM
    - Деталі процесу налаштування (параметри, кількість кроків, тривалість)
    - Результати експериментів з різними параметрами
    - Приклади згенерованого тексту та їх оцінка
    - Графіки розподілу оцінок для різних типів тексту
    - Значення логарифмічної правдоподібності тестового тексту
5. Аналіз результатів:
    - Порівняння різних параметрів налаштування
    - Аналіз впливу системного запиту "судді" на оцінку
    - Обговорення сильних і слабких сторін налаштованої моделі
6. Висновки:
    - Узагальнення результатів
    - Рекомендації щодо покращення процесу налаштування
    - Потенційні напрямки для подальших досліджень

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке велика мовна модель (LLM) і які її основні компоненти?
2. Яка роль токенізації в обробці тексту мовними моделями?
3. Що таке шаблони запитів (prompt templates) і як вони впливають на генерацію тексту?
4. Поясніть концепцію контекстного вікна в LLM та обмеження, які воно накладає.
5. У чому полягає відмінність між повним налаштуванням і методами ефективного налаштування параметрів?
6. Що таке LoRA і які її переваги порівняно з повним налаштуванням?
7. Які методи можна використовувати для оцінки якості генерації тексту LLM?
8. Як можна використовувати одні LLM для оцінки якості генерації інших LLM?
9. Що таке логарифмічна правдоподібність і як вона використовується для оцінки мовних моделей?

10. Яким чином температура генерації впливає на різноманітність і якість згенерованого тексту?
11. Які етичні аспекти слід враховувати при розробці та використанні LLM?
12. Як можна покращити ефективність процесу налаштування LLM для конкретних завдань?

СПИСОК ЛІТЕРАТУРИ

1. Achiam, J., et al. (2023). GPT-4 Technical Report. arXiv preprint arXiv:2303.08774.
2. Brown, T. B., et al. (2020). Language Models are Few-Shot Learners. NeurIPS.
3. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. NAACL.
4. Hu, E. J., et al. (2022). LoRA: Low-Rank Adaptation of Large Language Models. ICLR.
5. Raffel, C., et al. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. JMLR.
6. Vaswani, A., et al. (2017). Attention Is All You Need. NeurIPS.
7. Google. (2023). Gemma: Open Models Based on Gemini Research and Technology. https://blog.google/technology/developers/gemma-open-models/
8. Liquid AI. (2023). Liquid Foundation Models. https://www.liquid.ai/liquid-foundation-models
9. Comet ML. (2023). Opik: A Framework for Streamlined LLM Evaluation. https://www.comet.com/site/products/opik/
10. Hugging Face. (2023). Parameter-Efficient Fine-Tuning (PEFT). https://huggingface.co/docs/peft