```python
from typing import Tuple

from torch import LongTensor, cat
from transformers import PreTrainedTokenizer
from transformers.tokenization_utils_base import TruncationStrategy, BatchEncoding


class PreProcessor:
    framework: str = "pt"

    def __init__(
            self,
            tokenizer: PreTrainedTokenizer,
            max_input_len: int,
    ):
        self.tokenizer: PreTrainedTokenizer = tokenizer
        self.max_input_len: int = max_input_len

    def get_token_tensor(
            self,
            text: str,
            truncation: TruncationStrategy = TruncationStrategy.DO_NOT_TRUNCATE,
    ) -> LongTensor:
        """Complexity: O(n) where n is the number of characters in the text"""
        if len(text) == 0:
            return LongTensor([])
        # tokenizing a string is O(n) where n is the length of the string
        tokenized_text = self.tokenizer(
            text,
            return_tensors=self.framework,
            padding=False,
            add_special_tokens=False,
            truncation=truncation,
            max_length=self.max_input_len,
        )
        if isinstance(tokenized_text, dict) or isinstance(tokenized_text, BatchEncoding):
            token_tensor: LongTensor = tokenized_text["input_ids"]
            # O(1) because we are accessing a single element
            # in a dictionary and saving the reference to it.
        elif isinstance(tokenized_text, LongTensor):
            token_tensor: LongTensor = tokenized_text
            # O(1) because we are saving the reference to the tensor
        else:
            raise TypeError(
                "The tokenizer output is not one of:"
                "dict, BatchEncoding, LongTensor"
            )
        token_tensor = token_tensor.squeeze()
        # O(n) where n is the number of tokens in the text
        # because we are copying n elements from one tensor to the other
        # the number of tokens in the text
        # is always less than the number of characters in the text

        return token_tensor

    def __call__(
            self,
            prompt: str,
            prefix: str = "",
            truncation: TruncationStrategy = TruncationStrategy.DO_NOT_TRUNCATE,
            postfix: str = "",
    ) -> Tuple[LongTensor, int, int, int]:
        """A helper method for __call__ that tokenize the prompt
        all the arguments are sent directly from the __call__ method
        Returns:
            the tokenized prompt as a list of ints,
             the length of the prompt,
             the length of the prefix,
             the length of the postfix

        Complexity: O(a + b + c) where:
            'a' is the number of characters in the prefix
            'b' is the number of characters in the prompt
            'c' is the number of characters in the postfix"""

        prefix_tokens: LongTensor = self.get_token_tensor(prefix, truncation)
        # O(A) where A is the number of characters in the prefix.
        postfix_tokens: LongTensor = self.get_token_tensor(postfix, truncation)
        # O(B) where B is the number of characters in the postfix.
        prompt_tokens: LongTensor = self.get_token_tensor(prompt, truncation)
        # O(C) where C is the number of characters in the prompt.
        token_tensor: LongTensor = cat((prefix_tokens, prompt_tokens, postfix_tokens))
        # O( + b + c) where 'a' is the number of tokens in the prefix.
        # 'b' is the number of tokens in the prompt.
        # 'c' is the number of tokens in the postfix.
        # we know that the number of tokens is less than the number of characters
```

```
        # We know that the number of tokens is less than the number of characters
        return token_tensor, len(prefix_tokens), len(prompt_tokens), len(postfix_tokens)


from typing import Optional

from torch import tensor
from transformers import PreTrainedTokenizer

from src.grouped_sampling import TokenIDS


class PostProcessor:
    def __init__(
            self,
            tokenizer: PreTrainedTokenizer,
    ):
        self.tokenizer: PreTrainedTokenizer = tokenizer

    def __call__(
            self,
            token_ids: TokenIDS,
            num_new_tokens: Optional[int],
            prompt_len: int,
            return_text: bool,
            return_tensors: bool,
            return_full_text: bool,
            clean_up_tokenization_spaces: bool,
            prefix_len: int = 0,
            postfix_len: int = 0,
    ):
        """A helper method for __call__
        that converts the token ids to dictionary
        token_ids - the token ids from the _forward method
        prompt_len - the length of the tokenized prompt
        the rest of the arguments are the arguments
        from the __call__ method
        look up the documentation of the __call__ method for more info
        Complexity: O(n) where n is the number of tokens in token_ids
        """
        # define n as the length of the token_ids
        full_prompt_len = prefix_len + prompt_len + postfix_len
        if num_new_tokens is None:
            shorten_token_list = token_ids
            # O(1)
        else:
            final_num_tokens = full_prompt_len + num_new_tokens
            # O(1)
            if len(token_ids) > final_num_tokens:
                shorten_token_list = token_ids[:final_num_tokens]
                # O(final_num_tokens)
                # because we are copying final_num_tokens
                # elements from one list to the other
            else:
                shorten_token_list = token_ids
                # O(1)

        generated_tokens = shorten_token_list[full_prompt_len:]
        # O(num_new_tokens) because we are copying
        # num_new_tokens elements from one list to the other
        if return_full_text:
            prompt_tokens = shorten_token_list[prefix_len:prefix_len + prompt_len]
            # Without prefix and postfix
            # O(prompt_len) because we are copying prompt_len
            # elements from one list to the other
            final_token_list = prompt_tokens + generated_tokens
            # O(prompt_len + num_new_tokens)
            # because we are copying elements from one list to the other
        else:
            final_token_list = generated_tokens
            # O(1) because we are saving the reference to the list
        final_ans = {}
        if return_tensors:
            final_ans["generated_token_ids"] = tensor(final_token_list)
            # O(prompt_len + num_new_tokens)
            # because we are copying at maximum
            # prompt_len + num_new_tokens elements from a list to a tensor
        if return_text:
            final_ans["generated_text"] = self.tokenizer.decode(
                final_token_list,
                skip_special_tokens=True,
                clean_up_tokenization_spaces=clean_up_tokenization_spaces
            )
            # decoding is O(m)
            # where m is the number of tokens in the text
            # so the complexity of this line is O(n)
            # because final_token_list could be at most n tokens long
```

```
        return final_ans

from abc import ABC, abstractmethod
from typing import Set, Optional

from torch import Tensor

from .token_ids import TokenIDS


class RepetitionPenaltyStrategy(ABC):
    theta: float

    @staticmethod
    def get_already_generated_tokens(
            tokens: TokenIDS,
            generation_start: int
    ) -> Set[int]:
        return set(tokens[generation_start:])

    @abstractmethod
    def __call__(
            self,
            logits: Tensor,
            tokens: TokenIDS,
            generation_start: int
    ) -> Tensor:
        raise NotImplementedError

    def __str__(self) -> str:
        return f"{self.__class__.__name__}(theta={self.theta})"

    def __repr__(self) -> str:
        return str(self)


class LogitScalingRepetitionPenalty(
    RepetitionPenaltyStrategy
):
    def __init__(self, theta: float):
        if theta <= 1:
            raise ValueError("Theta must be > 1")
        self.theta = theta

    def __call__(
            self,
            logits: Tensor,
            tokens: TokenIDS,
            generation_start: int
    ) -> Tensor:
        """applies repetition penalty,
        using the repetition_penalty_theta parameter defined in the class
        the formula from the paper is:
        softmax(original_logits, T) =
            exp(original_logits_i/(T · h(i))
                  / sum(exp(original_logits_i/(T · h(i)))
        which is equivalent to:
        pi = exp((xi/h(i)) / T / sum(exp(xj/(T · h(j)))
        and to:
        penalized_logits = original_logits / h(i)
        pi = softmax(original_logits / h(i), T)
        where:
            h(i) = θ if i in generated_tokens else 1
            T is the temperature parameter of softmax
        Args:
            logits: the logits matrix of shape (group_size, vocab_size)
            tokens: the sequence of tokes from the prompt and the generated
            generation_start: The index of the first
        Returns:
            original_logits / h(i)
        complexity: O(group_size * n)
        where n is the number of tokens generated by the algorithm
        """
        generated_tokens = self.get_already_generated_tokens(
            tokens,
            generation_start
        )
        for token_id in generated_tokens:
            # len(generated_tokens) < max(n, vocab_size)
            logits[:, token_id] /= self.theta
            # O(group_size) because we are taking a slice of size group_size
        return logits


class NoRepetitionPenalty(
    RepetitionPenaltyStrategy
):
```

```
    theta = 1.0

    def __call__(
            self,
            logits: Tensor,
            tokens: TokenIDS,
            generation_start: int
    ) -> Tensor:
        return logits


DEFAULT_REPETITION_PENALTY = LogitScalingRepetitionPenalty(1.2)


def repetition_penalty_factory(
        theta: Optional[float]
) -> RepetitionPenaltyStrategy:
    if theta is None:
        return DEFAULT_REPETITION_PENALTY
    if theta == 1:
        return NoRepetitionPenalty()
    elif theta > 1:
        return LogitScalingRepetitionPenalty(theta)
    else:
        raise ValueError("theta must be greater than 1")


from enum import Enum


class GenerationType(Enum):
    """The type of generation to use"""
    GREEDY = "greedy"
    TOP_K = "top_k"
    TOP_P = "top_p"
    TREE = "tree"
    RANDOM = "random"

    def requires_softmax(self) -> bool:
        """Whether the generation type requires a softmax"""
        return self in (self.TOP_K, self.TOP_P, self.TREE, self.RANDOM)


from typing import Dict
from warnings import warn

from torch import cuda, LongTensor, ones, long, Tensor, cat, no_grad
from transformers import AutoModelForCausalLM
from torch.nn import Softmax

from .token_ids import TokenIDS
from .repetition_penalty import RepetitionPenaltyStrategy


class GroupedGenerationUtils:
    descriptive_attrs = (
        "group_size",
        "end_of_sentence_stop",
        "temp",
        "repetition_penalty_strategy",
    )

    def __init__(
            self,
            model_name: str,
            group_size: int,
            max_input_len: int,
            end_of_sentence_id: int,
            padding_id: int,
            vocab_size: int,
            repetition_penalty_strategy: RepetitionPenaltyStrategy,
            end_of_sentence_stop: bool = True,
            temp: float = 1.0,
            use_softmax: bool = True,
            **kwargs
    ):
        """initializes the model wrapper
        Args:
            model_name: the name of the model to be used
            repetition_penalty_strategy:
            the strategy to be used for the repetition penalty
            group_size: the number of next tokens to be generated
            max_input_len: the maximum length of the input to the model
            padding_id: the id of the padding token
            vocab_size: the size of the vocabulary
            end_of_sentence_stop:
            whether to stop when the end of sentence token is generated
```

```
            whether to stop when the end of sentence token is generated
            end_of_sentence_id:
            the id of the end of sequence token
            use_softmax:
            true if the model should use softmax,
            false if it should return the logits
            **kwargs: the arguments to be passed to the model
        Complexity: O(1)
        """
        self.use_softmax: bool = use_softmax
        self.end_of_sentence_id: int = end_of_sentence_id
        self.repetition_penalty_strategy: RepetitionPenaltyStrategy = repetition_penalty_strategy
        self.group_size: int = group_size
        self.max_input_len: int = max_input_len
        self.padding_id: int = padding_id
        self.end_of_sentence_stop: bool = end_of_sentence_stop
        self.model = AutoModelForCausalLM.from_pretrained(
            model_name, **kwargs
        )
        self.temp: float = temp
        self.vocab_size: int = vocab_size
        if cuda.is_available():
            self.model = self.model.cuda()

    @property
    def padding_tokens(self) -> LongTensor:
        cpu_tokens = ones(self.group_size, dtype=long) * self.padding_id
        if cuda.is_available():
            return cpu_tokens.cuda()
        return cpu_tokens

    def prepare_model_kwargs(
            self, tokens: TokenIDS
    ) -> Dict[str, LongTensor]:
        """preparing the arguments for the model call
        Args:
            tokens: the tokens to be sent to the model
        Returns:
            a dictionary of the arguments for the model call
        Complexity: O(group_size + n) where n is the number of tokens
        """
        if not isinstance(tokens, Tensor):
            tokens = LongTensor(tokens)  # O(n)
        padded_tokens: LongTensor = cat(
            (tokens, self.padding_tokens), dim=0
        ).unsqueeze(0)
        # the length of padded_tokens is n + group_size - 1
        # so creating it is O(n + group_size)
        attention_len = padded_tokens.shape[1]  # n + group_size - 1
        if attention_len > self.max_input_len:
            padded_tokens = padded_tokens[:, -self.max_input_len:]
            # O(self.max_input_len) which is constant so O(1)
            attention_len = self.max_input_len
        attention_mask: LongTensor = ones(
            [1, attention_len], dtype=long
        )
        # O(attention_len) so O(n + group_size)
        if cuda.is_available():
            padded_tokens = padded_tokens.cuda()  # O(n + group_size)
            attention_mask = attention_mask.cuda()  # O(n + group_size)
        else:
            warn("CUDA is not available, using CPU")
        return {
            "input_ids": padded_tokens,
            "attention_mask": attention_mask,
        }

    def get_logits_matrix(self, tokens: TokenIDS) -> Tensor:
        """Given a sequence of tokens,
        returns the logits matrix of shape (group_size, vocab_size)
        where logits[i] is the logits vector of the i-th next token
        complexity: O(n^2 + group_size^2) where n is the length of the tokens
        notice that the logits are not divided by the temperature in this function."""
        # define n as the number of tokens in tokens
        model_kwargs = self.prepare_model_kwargs(tokens)  # O(n + group_size)
        with no_grad():
            # The time complexity of causal language model`s __call__ function
            # is O(n^2) where n is the length of the inputs
            outputs = self.model(
                **model_kwargs
            )
            # the length of all the inputs is n + group_size - 1
            # so the complexity of this line is O((n + group_size - 1)^2)
            # which is O(n^2 + group_size^2 + group_size * n)
            # we now that if a > b and a, b > 1 then a^2 > ab
            # so the complexity is O(n^2 + group_size^2)
        unscaled_relevant_logits: Tensor
```

```
        unscaled_relevant_logits = outputs.logits[0, -self.group_size:, :self.vocab_size]
        # The shape of unscaled_relevant_logits is (group_size, vocab_size)
        # So the complexity of this line should be
        # O(group_size) because we are coping group_size * vocab_size
        # elements from one tensor to the another
        return unscaled_relevant_logits

    def get_prob_mat(
            self, tokens: TokenIDS, generation_start: int
    ) -> Tensor:
        """Returns the probability matrix
         as a list of lists of floats
         Time complexity: O(n^2 + group_size^2)
         where n is the number of tokens"""
        unscaled_relevant_logits = self.get_logits_matrix(tokens)
        # O(n^2 + group_size^2)
        # unscaled_relevant_logits is a tensor of shape (group_size, vocab_size)
        if not self.end_of_sentence_stop:
            unscaled_relevant_logits[:, self.end_of_sentence_id] = -float('inf')
            # setting a vector of size vocab_size
            # so the complexity is O(group_size)
            # setting the logits to -inf so the probability will be 0
        penalized_logits = self.repetition_penalty_strategy(
            unscaled_relevant_logits, tokens, generation_start
        )
        # O(group_size * n)
        # where n is the number of tokens generated by the algorithm
        prob_tensor = self.logits_to_probs(penalized_logits)  # O(group_size)
        # We are doing a softmax operator
        # of group_size different vectors of size vocab_size
        # The complexity of the softmax for each vector is
        # O(1) because the size of the vector size is constant
        # the complexity of this line is O(group_size)
        # because we are doing group_size softmax operations
        return prob_tensor

    def logits_to_probs(self, penalized_logits: Tensor) -> Tensor:
        """Gets the logits matrix and returns the probability matrix
        Time complexity: O(group_size)"""
        if self.use_softmax:
            # if the generation type
            # is not greedy then we need to apply softmax
            # to the penalized logits
            if self.temp != 1.0:
                penalized_logits /= self.temp
                # O(group_size * vocab_size)
                # because we are dividing a matrix of size (group_size, vocab_size)
            return Softmax(dim=1)(penalized_logits)
            # O(group_size * vocab_size) so the complexity is O(group_size)
        return penalized_logits

    def __str__(self):
        return f"GroupedGenerationUtils({self.as_dict()})"

    def __repr__(self):
        return str(self)

    def as_dict(self):
        return {attr_name: getattr(self, attr_name)
                for attr_name in self.descriptive_attrs}

from __future__ import annotations

from abc import ABC, abstractmethod
from collections.abc import Callable
from typing import Optional, List, Union, Dict, Any

from torch import LongTensor
from transformers import (
    AutoTokenizer,
    AutoConfig,
    PreTrainedTokenizer,
)
from transformers.tokenization_utils_base import TruncationStrategy

from .generation_type import GenerationType
from .generation_utils import GroupedGenerationUtils
from .postprocessor import PostProcessor
from .preprocessor import PreProcessor
from .repetition_penalty import RepetitionPenaltyStrategy, DEFAULT_REPETITION_PENALTY
from .completion_dict import CompletionDict
from .token_ids import TokenIDS

MAX_MODEL_INPUT_SIZE = 32768


def remove_nones(d: Dict[str, Any]) -> Dict[str, Any]:
```

```
    """Returns a copy of a dictionary with all the not None values"""
    return {key: d[key] for key in d.keys() if d[key] is not None}


def get_padding_id(tokenizer: PreTrainedTokenizer):
    padding_id = tokenizer.pad_token_id
    if not isinstance(padding_id, int):
        padding_id = tokenizer.unk_token_id
    if not isinstance(padding_id, int):
        padding_id = tokenizer.mask_token_id
    if not isinstance(padding_id, int):
        raise RuntimeError(f"padding_id is {padding_id} and its type is {type(padding_id)}")
    return padding_id


class GroupedGenerationPipeLine(Callable, ABC):
    """An abstract base class for
    A callable object that given a func_prompt
     and length of wanted answer,
    generates text
    the text generator has a model,
    and some parameters
    (Defined in the subclasses)"""

    framework: str = "pt"
    descriptive_attrs = (
        "model_name",
        "generation_type",
        "answer_length_multiplier",
        "wrapped_model",
    )

    def __init__(
            self,
            model_name: str,
            group_size: int,
            temp: Optional[float] = None,
            end_of_sentence_stop: Optional[bool] = None,
            repetition_penalty_strategy: RepetitionPenaltyStrategy = DEFAULT_REPETITION_PENALTY,
            answer_length_multiplier: float = 16,
    ):
        """Model name: the name of the model
        used for loading from hugging face hub
        group size: int
        the number of tokens to be predicted at each model call
        temp: float
        temperature parameter for the softmax function
        answer_length_multiplier: int
            if the answer length is not given,
            the maximum answer length is set to:
            the length of the prompt * answer_length_multiplier
        repetition_penalty_strategy: RepetitionPenaltyStrategy
            The strategy for the repetition penalty
        """
        self.model_name: str = model_name
        tokenizer = AutoTokenizer.from_pretrained(model_name)
        end_of_sentence_id = tokenizer.eos_token_id
        end_of_sentence_stop = end_of_sentence_stop and end_of_sentence_id is not None
        max_input_len = tokenizer.model_max_length
        max_len_is_huge = max_input_len > MAX_MODEL_INPUT_SIZE
        if max_len_is_huge or max_input_len is None:
            config = AutoConfig.from_pretrained(model_name)
            max_input_len = config.max_position_embeddings
            max_len_is_still_huge = max_input_len > MAX_MODEL_INPUT_SIZE
            if max_len_is_still_huge or max_input_len is None:
                raise ValueError(
                    "The maximum length of the model is too big"
                )
        self.pre_processing_strategy: PreProcessor = PreProcessor(
            tokenizer=tokenizer,
            max_input_len=max_input_len,
        )
        self.post_processing_strategy: PostProcessor = PostProcessor(
            tokenizer=tokenizer,
        )
        wrapped_model_kwargs: Dict[str, Any] = {
            "model_name": model_name,
            "group_size": group_size,
            "max_input_len": max_input_len,
            "end_of_sentence_id": end_of_sentence_id,
            "end_of_sentence_stop": end_of_sentence_stop,
            "repetition_penalty_strategy": repetition_penalty_strategy,
            "padding_id": get_padding_id(tokenizer),
            "temp": temp,
            "use_softmax": self.generation_type.requires_softmax(),
            "vocab_size": tokenizer.vocab_size,
```

```
        }
        self.wrapped_model: GroupedGenerationUtils = GroupedGenerationUtils(**remove_nones(wrapped_model_kwargs))
        self.answer_length_multiplier: float = answer_length_multiplier

    @property
    @abstractmethod
    def generation_type(self) -> GenerationType:
        """A method that chooses the generation type
        Returns:
            a GenerationType object"""
        raise NotImplementedError

    @abstractmethod
    def _forward(
            self,
            tokenized_prompt: LongTensor,
            num_new_tokens: Optional[int] = None,
            num_return_sequences: int = 1,
    ) -> List[TokenIDS]:
        """A helper method for __call__ that generates the new tokens
        Has a unique implementation for each subclass
        Args:
            tokenized_prompt: List[int]
                the tokenized prompt from the preprocess method
            num_new_tokens: int - the number of new tokens to generate
                from the __call__ method
        Returns:
            the prompt + generated text as a list/tuple of ints"""
        pass

    def __call__(
            self,
            prompt_s: Union[str, List[str]],
            max_new_tokens: Optional[int] = None,
            return_tensors: bool = False,
            return_text: bool = True,
            return_full_text: bool = True,
            clean_up_tokenization_spaces: bool = False,
            prefix: str = "",
            num_return_sequences: int = 1,
            truncation: TruncationStrategy = TruncationStrategy.DO_NOT_TRUNCATE,
            postfix: str = "",
    ) -> CompletionDict | List[CompletionDict] | List[List[CompletionDict]]:
        """The function that outside code should call to generate text
        Args:
            prompt_s: str or list of str - the prompt(s) to start the generation from
                (the text given by the user)
                if many prompts are given as a list,
                the function process each one independently and returns them as a list.
                (the same as calling [__call__(prompt, *args, **kwargs)
                 for prompt in prompts])])
            max_new_tokens: Optional[int] > 0
             - the number of tokens to generate
                if None, the function will generate tokens
                 until one of them is the end of sentence token
            return_tensors: bool - whether to return the generated token ids
            return_text: bool - whether to return the generated string
            return_full_text: bool - whether to return the full text
                (prompt + generated text)
                (if false, it will return only the generated text)
            clean_up_tokenization_spaces: bool
             - whether to clean up tokenization spaces
                This parameter is forwarded to the decode function of the AutoTokenizer class
            prefix (`str`, defaults to an empty string):
                Prefix added to prompt.
            num_return_sequences (`int`, defaults to 1):
                The number of independently generated answers to return for each prompt.
                For GroupedSamplingPipeLine:
                    each answer will be generated with different seed.
                For GroupedTreePipeLine:
                    the num_return_sequences with the highest scores will be returned.
            truncation: TruncationStrategy
             - whether to truncate the prompt
            postfix: str
             - a postfix to add to the prompt
        Returns:
            Each result comes as a dictionary with the following keys:
            - "generated_text"
            (`str`, present when `return_text=True`)
             -- The generated text.
            - "generated_token_ids" (
            `torch.tensor`, present when `return_tensors=True`)
             -- The token
              ids of the generated text.
            """

        if max_new_tokens is None and \
```

```
                not self.wrapped_model.end_of_sentence_stop:
            raise ValueError(
                "max_new_tokens must be given if end_of_sentence_stop is False"
            )
        if isinstance(prompt_s, list):
            return [self.__call__(
                prompt_s=prompt,
                max_new_tokens=max_new_tokens,
                return_text=return_text,
                return_tensors=return_tensors,
                return_full_text=return_full_text,
                clean_up_tokenization_spaces=clean_up_tokenization_spaces,
                truncation=truncation,
                postfix=postfix,
            ) for prompt in prompt_s]
        tokens: LongTensor
        prefix_len: int
        postfix_len: int
        prompt_len: int

        tokens, prefix_len, prompt_len, postfix_len = self.pre_processing_strategy(
            prompt=prompt_s,
            prefix=prefix,
            truncation=truncation,
            postfix=postfix
        )
        # O(len(prompt) + len(prefix) + len(postfix))

        if max_new_tokens is None:
            max_new_tokens = int(prompt_len * self.answer_length_multiplier)
            # O(1)
        tokenized_answers: List[TokenIDS]
        tokenized_answers = self._forward(
            tokens,
            max_new_tokens,
            num_return_sequences
        )

        if num_return_sequences > 1:
            # O(sum(len(tokenized_answer) for tokenized_answer in tokenized_answers))
            return [self.post_processing_strategy(
                token_ids=tokenized_answer,
                num_new_tokens=max_new_tokens,
                prompt_len=prompt_len,
                return_text=return_text,
                return_tensors=return_tensors,
                return_full_text=return_full_text,
                clean_up_tokenization_spaces=clean_up_tokenization_spaces,
                prefix_len=prefix_len,
                postfix_len=postfix_len,
            ) for tokenized_answer in tokenized_answers]
        else:
            # O(len(tokenized_answers[0]))
            return self.post_processing_strategy(
                token_ids=tokenized_answers[0],
                num_new_tokens=max_new_tokens,
                prompt_len=prompt_len,
                return_text=return_text,
                return_tensors=return_tensors,
                return_full_text=return_full_text,
                clean_up_tokenization_spaces=clean_up_tokenization_spaces,
                prefix_len=prefix_len,
                postfix_len=postfix_len,
            )

    def __repr__(self):
        attrs_description = ", ".join(
            f"{attr}={getattr(self, attr)}" for attr in self.descriptive_attrs
        )
        return f"{self.__class__.__name__}: " + attrs_description

    def __str__(self):
        return repr(self)

    def as_dict(self) -> Dict[str, Any]:
        """Returns a dictionary representation
         of the generator
        such that it can be saved and loaded
         using the from_dict method"""
        return {
            key: getattr(self, key)
            for key in self.descriptive_attrs
        }

    @classmethod
    def from_dict(cls, my_dict: Dict[str, Any]):
        """Creates an GroupedGenerationPipeLine from a dictionary
```

```
        Created an GroupedGenerationPipeline from a dictionary
        The dictionary should have the same format
         as the dictionary returned by the as_dict method"""
        if "generation_type" in my_dict.keys():
            my_dict.pop("generation_type")
        wrapped_model: GroupedGenerationUtils = my_dict.pop("wrapped_model")
        wrapped_model_dict = wrapped_model.as_dict()
        my_dict.update(wrapped_model_dict)
        return cls(**my_dict)

import heapq
from collections.abc import Iterator
from random import seed
from typing import Callable, List, Dict, Optional, Any

from torch import Tensor, zeros, argmax, multinomial, manual_seed

from .generation_type import GenerationType
from .base_pipeline import GroupedGenerationPipeLine


class ChangingSeed(Iterator):
    """Context manager for changing the seed of the random module.
    How to use:
    with ChangingSeed(first_seed, number_of_different_seeds) as changing_seed:
        for _ in changing_seed:
            # do something with random module"""
    def __init__(self, default_seed: int, max_num_calls: int):
        self.default_seed: int = default_seed
        self.curr_seed: int = self.default_seed
        self.max_num_calls: int = max_num_calls
        self.curr_num_calls: int = 0

    def __enter__(self):
        self.curr_num_calls = 0
        self.curr_seed = self.default_seed
        return self

    def __exit__(self, *args):
        self.curr_seed = self.default_seed
        seed(self.default_seed)
        manual_seed(self.default_seed)

    def __iter__(self):
        self.curr_seed = self.default_seed
        return self

    def __next__(self):
        self.curr_seed += 1
        seed(self.curr_seed)
        manual_seed(self.curr_seed)
        self.curr_num_calls += 1
        if self.curr_num_calls > self.max_num_calls:
            raise StopIteration


class TokenProb:
    """Class for storing the probability of a token and the token itself.
    Used to store the probabilities of the next tokens in the sampling generator.
    Is useful because it supports the < and > operators, which are used in the
    heapq module
    The < and > are the opposite of each other because the heapq module is only supporting minimum heaps
    and I need a maximum heap"""
    __slots__ = ['token_id', 'prob']

    def __init__(self, token_id: int, prob: Tensor):
        self.token_id: int = token_id
        self.prob: Tensor = prob

    def __lt__(self, other: "TokenProb"):
        """Overrides the < operator
        Comparison is done by the probability"""
        return self.prob > other.prob

    def __gt__(self, other: "TokenProb"):
        """Overrides the > operator
        Comparison is done by the probability"""
        return self.prob < other.prob


class GroupedSamplingPipeLine(GroupedGenerationPipeLine):
    """A GroupedGenerationPipeLine that generates text
    using random sampling
    with top-k or top-p filtering."""
    default_seed: int = 0
    seed(default_seed)
    manual_seed(default_seed)
```

```
    unique_attrs = "top_k", "top_p"

    def __init__(self, top_k: Optional[int] = None,
                 top_p: Optional[float] = None, *args, **kwargs):
        self.top_p: Optional[float] = top_p
        self.top_k: Optional[int] = top_k
        super().__init__(*args, **kwargs)

    def __setattr__(self, key, value):
        super().__setattr__(key, value)
        if key == "default_seed":
            seed(value)
            manual_seed(value)

    @property
    def generation_type(self) -> GenerationType:
        if self.top_k is None and self.top_p is None:
            return GenerationType.RANDOM
        if self.top_k == 1 or self.top_p == 0.0:
            return GenerationType.GREEDY
        if self.top_k is not None:
            return GenerationType.TOP_K
        if self.top_p is not None:
            if self.top_p < 1.0:
                return GenerationType.TOP_P
            return GenerationType.RANDOM
        raise RuntimeError("Uncovered case in generation_type property")

    @property
    def sampling_func(self) -> Callable[[Tensor], int]:
        gen_type_to_filter_method: Dict[GenerationType, Callable[[Tensor, ], int]] = {
            GenerationType.TOP_K: self.top_k_sampling,
            GenerationType.TOP_P: self.top_p_sampling,
            GenerationType.GREEDY: GroupedSamplingPipeLine.highest_prob_token,
            GenerationType.RANDOM: GroupedSamplingPipeLine.unfiltered_sampling,
        }
        return gen_type_to_filter_method[self.generation_type]

    @staticmethod
    def unfiltered_sampling(prob_vec: Tensor) -> int:
        """A sampling function that doesn't filter any tokens.
        returns a random token id sampled from the probability vector"""
        return multinomial(prob_vec, 1).item()

    @staticmethod
    def highest_prob_token(prob_vec: Tensor) -> int:
        """Gets a probability vector of shape (vocab_size,)
        returns the token id with the highest probability"""
        return argmax(prob_vec).item()

    def top_p_sampling(self, prob_vec: Tensor) -> int:
        """Gets a probability vector of shape (vocab_size,)
        computes a probability vector with the top p tokens
        such that their sum in the original vector is <= self.top_p.
        and samples from that vector.
        If token with the highest probability
        have a probability higher than top_p, it will be sampled"""
        prob_sum: float = 0.0
        converted_probs = [
            TokenProb(i, prob) for i, prob in enumerate(prob_vec)
        ]
        heapq.heapify(converted_probs)
        new_probs = zeros(prob_vec.shape, dtype=float)
        while prob_sum < self.top_p and len(converted_probs) > 0:
            curr_token_prob: TokenProb = heapq.heappop(converted_probs)
            token_id = curr_token_prob.token_id
            if curr_token_prob.prob <= 0.0:
                break
            if curr_token_prob.prob > 1:
                raise ValueError(
                    f"Probability of token {token_id} "
                    f"in the vector {prob_vec} "
                    f"is {curr_token_prob.prob}"
                    f" which is higher than 1")
            prob_sum += curr_token_prob.prob
            new_probs[token_id] = curr_token_prob.prob
        if prob_sum == 0.0:
            return converted_probs[0].token_id
        return GroupedSamplingPipeLine.unfiltered_sampling(new_probs)

    def top_k_sampling(self, prob_vec: Tensor) -> int:
        """Gets a token id: probability mapping
        returns the TOP_K tokens
        with the highest probability.
        this is the bottleneck of the sampling generator."""
        top_k_keys: List[int] = heapq.nlargest(
            self.top_k,
```

```
                _
            range(prob_vec.shape[0]),
            key=lambda x: prob_vec[x]
        )
        prob_sum = sum(prob_vec[token_id] for token_id in top_k_keys)
        new_probs = zeros(prob_vec.shape, dtype=float)
        for token_id in top_k_keys:
            new_probs[token_id] = prob_vec[token_id] / prob_sum
        return GroupedSamplingPipeLine.unfiltered_sampling(new_probs)

    def generate_group(self, prob_mat: Tensor) -> List[int]:
        """Generates a group of tokens
         using the choice_function.
         Complexity: O(group_size)"""
        prob_mat.cpu()
        # coping a tensor of size (group_size, vocab_size)
        # so the complexity is O(group_size)
        # (vocab_size is constant)
        new_group: List[int] = [
            self.sampling_func(prob_vec)
            for prob_vec in prob_mat
        ]
        # the complexity of the loop is O(group_size)
        # because self.sampling_func gets a tensor
        # of constant size (vocab_size,)
        # and therefore must be O(1) in complexity
        # and the loop has group_size iterations.
        del prob_mat
        for i, token_id in enumerate(new_group):
            if token_id == self.wrapped_model.end_of_sentence_id:
                return new_group[:i + 1]
                # return the group until the end of sentence token included
                # the complexity of this line is O(group_size)
                # because it is coping a list with maximum size of group_size
        return new_group

    def _forward(
            self,
            tokenized_prompt: Tensor,
            num_new_tokens: Optional[int] = None,
            num_return_sequences: int = 1,
    ) -> List[List[int]]:
        """Complexity:
            O(num_return_sequences * (
                ((n ^ 3) / group_size) +
                ((n * l ^ 2) / group_size) +
                group_size +
                n)
            )
        where l is the number of tokens in the prompt
        and n is the number of new tokens to generate"""
        # let's define l = len(tokenized_prompt), n = num_new_tokens
        answers: List[List[int]] = []
        curr_token_list: List[int] = tokenized_prompt.tolist()
        # coping a tensor of size lso O(l)
        if num_new_tokens is None:
            raise RuntimeError("num_new_tokens is None")
        for _ in ChangingSeed(
                default_seed=self.default_seed,
                max_num_calls=num_return_sequences):
            # num_return_sequences iterations
            for _ in range(num_new_tokens // self.wrapped_model.group_size):
                # and each iteration is
                # O(n ^ 2 + l ^ 2 + group_size ^ 2 + group_size)
                # so the complexity of the loop is
                # O((n ^ 3) / group_size + (n * l ^ 2) / group_size + group_size + n)
                prob_mat: Tensor = self.wrapped_model.get_prob_mat(
                    curr_token_list, len(tokenized_prompt)
                )
                # complexity: O(group_size ^ 2 + len(curr_token_list) ^ 2)
                # len(curr_token_list) <= n + l
                # so the complexity is
                # O(group_size ^ 2 + (n + l) ^ 2) is equals to
                # O(n ^ 2 + nl + l ^ 2 + group_size ^ 2)
                # but nl <= max(n^2, l^2) so the complexity
                # is O(n ^ 2 + l ^ 2 + group_size ^ 2)
                new_tokens = self.generate_group(prob_mat)
                # complexity: O(group_size)
                # len(curr_token_list) <= n + l
                # so the complexity is O(group_size * (n + l + group_size))
                # len(new_tokens) = group_size
                if self.wrapped_model.end_of_sentence_id in new_tokens:
                    # the check is O(group_size)
                    end_of_sentence_index = new_tokens.index(
                        self.wrapped_model.end_of_sentence_id)
                    # O(group_size) because len(new_tokens) <= group_size
                    new_tokens = new_tokens[:end_of_sentence_index]
                    # O(group_size) because end_of_sentence_index < group_size
```

```
                    # O(group_size) because end_of_sentence_index < group_size
                curr_token_list.extend(new_tokens)
                # O(group_size) because len(new_tokens) <= group_size
            answers.append(curr_token_list)
            # O(1)
        return answers

    def __repr__(self):
        super_representation = super().__repr__()
        unique_representation = '/n'.join(
            f"{unique_attr_name}={getattr(self, unique_attr_name)}"
            for unique_attr_name in self.unique_attrs)
        return super_representation + unique_representation

    def as_dict(self) -> Dict[str, Any]:
        super_dict = super(GroupedSamplingPipeLine, self).as_dict()
        super_dict.update(
            {unique_attr: self.__getattribute__(unique_attr)
             for unique_attr in self.unique_attrs}
        )
        return super_dict
```