```python
"""Sampling parameters for text generation."""

from enum import IntEnum
from functools import cached_property
from typing import Callable, List, Optional, Union

import torch

_SAMPLING_EPS = 1e-5


class SamplingType(IntEnum):
    GREEDY = 0
    RANDOM = 1
    BEAM = 2


LogitsProcessor = Callable[[List[int], torch.Tensor], torch.Tensor]
"""LogitsProcessor is a function that takes a list of previously generated
tokens and a tensor of the logits for the next token, and returns a modified
tensor of logits to sample from."""


class SamplingParams:
    """Sampling parameters for text generation.
```

```
Overall, we follow the sampling parameters from the OpenAI text completion
API (https://platform.openai.com/docs/api-reference/completions/create).
In addition, we support beam search, which is not supported by OpenAI.

Args:
    n: Number of output sequences to return for the given prompt.
    best_of: Number of output sequences that are generated from the prompt.
        From these `best_of` sequences, the top `n` sequences are returned.
        `best_of` must be greater than or equal to `n`. This is treated as
        the beam width when `use_beam_search` is True. By default, `best_of`
        is set to `n`.
    presence_penalty: Float that penalizes new tokens based on whether they
        appear in the generated text so far. Values > 0 encourage the model
        to use new tokens, while values < 0 encourage the model to repeat
        tokens.
    frequency_penalty: Float that penalizes new tokens based on their
        frequency in the generated text so far. Values > 0 encourage the
        model to use new tokens, while values < 0 encourage the model to
        repeat tokens.
    repetition_penalty: Float that penalizes new tokens based on whether
        they appear in the prompt and the generated text so far. Values > 1
        encourage the model to use new tokens, while values < 1 encourage
        the model to repeat tokens.
    temperature: Float that controls the randomness of the sampling. Lower
        values make the model more deterministic, while higher values make
        the model more random. Zero means greedy sampling.
```

top_p: Float that controls the cumulative probability of the top tokens
    to consider. Must be in (0, 1]. Set to 1 to consider all tokens.

top_k: Integer that controls the number of top tokens to consider. Set
    to -1 to consider all tokens.

min_p: Float that represents the minimum probability for a token to be
    considered, relative to the probability of the most likely token.
    Must be in [0, 1]. Set to 0 to disable this.

use_beam_search: Whether to use beam search instead of sampling.

length_penalty: Float that penalizes sequences based on their length.
    Used in beam search.

early_stopping: Controls the stopping condition for beam search. It
    accepts the following values: `True`, where the generation stops as
    soon as there are `best_of` complete candidates; `False`, where an
    heuristic is applied and the generation stops when is it very
    unlikely to find better candidates; `"never"`, where the beam search
    procedure only stops when there cannot be better candidates
    (canonical beam search algorithm).

stop: List of strings that stop the generation when they are generated.
    The returned output will not contain the stop strings.

stop_token_ids: List of tokens that stop the generation when they are
    generated. The returned output will contain the stop tokens unless
    the stop tokens are special tokens.

include_stop_str_in_output: Whether to include the stop strings in output
    text. Defaults to False.

ignore_eos: Whether to ignore the EOS token and continue generating
    tokens after the EOS token is generated.

```
    max_tokens: Maximum number of tokens to generate per output sequence.
    logprobs: Number of log probabilities to return per output token.
        Note that the implementation follows the OpenAI API: The return
        result includes the log probabilities on the `logprobs` most likely
        tokens, as well the chosen tokens. The API will always return the
        log probability of the sampled token, so there  may be up to
        `logprobs+1` elements in the response.
    prompt_logprobs: Number of log probabilities to return per prompt token.
    skip_special_tokens: Whether to skip special tokens in the output.
    spaces_between_special_tokens: Whether to add spaces between special
        tokens in the output.  Defaults to True.
    logits_processors: List of functions that modify logits based on
        previously generated tokens.
"""

def __init__(
    self,
    n: int = 1,
    best_of: Optional[int] = None,
    presence_penalty: float = 0.0,
    frequency_penalty: float = 0.0,
    repetition_penalty: float = 1.0,
    temperature: float = 1.0,
    top_p: float = 1.0,
    top_k: int = -1,
    min_p: float = 0.0,
    use_beam_search: bool = False,
    length_penalty: float = 1.0,
    early_stopping: Union[bool, str] = False,
    stop: Union[str, List[str], None] = None,
    stop_token_ids: Optional[List[int]] = None,
    include_stop_str_in_output: bool = False,
    ignore_eos: bool = False,
    max_tokens: Optional[int] = 16,
    logprobs: Optional[int] = None,
    prompt_logprobs: Optional[int] = None,
    skip_special_tokens: bool = True,
    spaces_between_special_tokens: bool = True,
    logits_processors: Optional[List[LogitsProcessor]] = None,
) -> None:
    self.n = n
    self.best_of = best_of if best_of is not None else n
    self.presence_penalty = presence_penalty
    self.frequency_penalty = frequency_penalty
    self.repetition_penalty = repetition_penalty
    self.temperature = temperature
    self.top_p = top_p
    self.top_k = top_k
    self.min_p = min_p
    self.use_beam_search = use_beam_search
    self.length_penalty = length_penalty
    self.early_stopping = early_stopping
```

```
if stop is None:
    self.stop = []
elif isinstance(stop, str):
    self.stop = [stop]
else:
    self.stop = list(stop)
if stop_token_ids is None:
    self.stop_token_ids = []
else:
    self.stop_token_ids = list(stop_token_ids)
self.ignore_eos = ignore_eos
self.max_tokens = max_tokens
self.logprobs = logprobs
self.prompt_logprobs = prompt_logprobs
self.skip_special_tokens = skip_special_tokens
self.spaces_between_special_tokens = (
    spaces_between_special_tokens
)
self.logits_processors = logits_processors
self.include_stop_str_in_output = include_stop_str_in_output
self._verify_args()
if self.use_beam_search:
    self._verify_beam_search()
else:
    self._verify_non_beam_search()
    if self.temperature < _SAMPLING_EPS:
            # Zero temperature means greedy sampling.
            self.top_p = 1.0
            self.top_k = -1
            self.min_p = 0.0
            self._verify_greedy_sampling()

def _verify_args(self) -> None:
    if self.n < 1:
        raise ValueError(f"n must be at least 1, got {self.n}.")
    if self.best_of < self.n:
        raise ValueError(
            "best_of must be greater than or equal to n, "
            f"got n={self.n} and best_of={self.best_of}."
        )
    if not -2.0 <= self.presence_penalty <= 2.0:
        raise ValueError(
            "presence_penalty must be in [-2, 2], got "
            f"{self.presence_penalty}."
        )
    if not -2.0 <= self.frequency_penalty <= 2.0:
        raise ValueError(
            "frequency_penalty must be in [-2, 2], got "
            f"{self.frequency_penalty}."
        )
    if not 0.0 < self.repetition_penalty <= 2.0:
        raise ValueError(
        "repetition_penalty must be in (0, 2], got "
        f"{self.repetition_penalty}."
    )
if self.temperature < 0.0:
    raise ValueError(
        "temperature must be non-negative, got"
        f" {self.temperature}."
    )
if not 0.0 < self.top_p <= 1.0:
    raise ValueError(
        f"top_p must be in (0, 1], got {self.top_p}."
    )
if self.top_k < -1 or self.top_k == 0:
    raise ValueError(
        "top_k must be -1 (disable), or at least 1, "
        f"got {self.top_k}."
    )
if not 0.0 <= self.min_p <= 1.0:
    raise ValueError(
        f"min_p must be in [0, 1], got {self.min_p}."
    )
if self.max_tokens is not None and self.max_tokens < 1:
    raise ValueError(
        "max_tokens must be at least 1, got"
        f" {self.max_tokens}."
    )
    if self.logprobs is not None and self.logprobs < 0:
        raise ValueError(
            f"logprobs must be non-negative, got {self.logprobs}."
        )
    if (
        self.prompt_logprobs is not None
        and self.prompt_logprobs < 0
    ):
        raise ValueError(
            "prompt_logprobs must be non-negative, got "
            f"{self.prompt_logprobs}."
        )

def _verify_beam_search(self) -> None:
    if self.best_of == 1:
        raise ValueError(
            "best_of must be greater than 1 when using beam "
            f"search. Got {self.best_of}."
        )
    if self.temperature > _SAMPLING_EPS:
        raise ValueError(
            "temperature must be 0 when using beam search."
        )
    if self.top_p < 1.0 - _SAMPLING_EPS:
        raise ValueError(
            "top_p must be 1 when using beam search."
        )
    if self.top_k != -1:
        raise ValueError(
            "top_k must be -1 when using beam search."
        )
    if self.early_stopping not in [True, False, "never"]:
        raise ValueError(
            "early_stopping must be True, False, or 'never', "
            f"got {self.early_stopping}."
        )

def _verify_non_beam_search(self) -> None:
    if self.early_stopping is not False:
        raise ValueError(
            "early_stopping is not effective and must be "
            "False when not using beam search."
        )
    if (
        self.length_penalty < 1.0 - _SAMPLING_EPS
        or self.length_penalty > 1.0 + _SAMPLING_EPS
    ):
        raise ValueError(
            "length_penalty is not effective and must be the "
            "default value of 1.0 when not using beam search."
        )
```

```
def _verify_greedy_sampling(self) -> None:
    if self.best_of > 1:
        raise ValueError(
            "best_of must be 1 when using greedy sampling."
            f"Got {self.best_of}."
        )

@cached_property
def sampling_type(self) -> SamplingType:
    if self.use_beam_search:
        return SamplingType.BEAM
    if self.temperature < _SAMPLING_EPS:
        return SamplingType.GREEDY
    return SamplingType.RANDOM

def __repr__(self) -> str:
    return (
        f"SamplingParams(n={self.n}, "
        f"best_of={self.best_of}, "
        f"presence_penalty={self.presence_penalty}, "
        f"frequency_penalty={self.frequency_penalty}, "
        f"repetition_penalty={self.repetition_penalty}, "
        f"temperature={self.temperature}, "
        f"top_p={self.top_p}, "
        f"top_k={self.top_k}, "
        f"min_p={self.min_p}, "
        f"use_beam_search={self.use_beam_search}, "
        f"length_penalty={self.length_penalty}, "
        f"early_stopping={self.early_stopping}, "
        f"stop={self.stop}, "
        f"stop_token_ids={self.stop_token_ids}, "
        f"include_stop_str_in_output={self.include_stop_str_in_output}, "
        f"ignore_eos={self.ignore_eos}, "
        f"max_tokens={self.max_tokens}, "
        f"logprobs={self.logprobs}, "
        f"prompt_logprobs={self.prompt_logprobs}, "
        f"skip_special_tokens={self.skip_special_tokens}, "
        "spaces_between_special_tokens="
        f"{self.spaces_between_special_tokens})"
    )
```