

ニューラルネットワークの理解

Zhiwei:

もちろん、コードの説明をします。どのコードについて説明してほしいか教えていただけますか？コードブロックや具体的なコードの内容を共有していただければ、詳しく解説いたします。

```
self.weights = [w - (eta / len(mini_batch)) * nw
                for w, nw in zip(self.weights, nabla_w)]
self.biases = [b - (eta / len(mini_batch)) * nb
               for b, nb in zip(self.biases, nabla_b)]
```

このコードは、ニューラルネットワークの重み (weights) とバイアス (biases) を更新する部分です。各重みとバイアスは、勾配降下法に基づいて、ミニバッチの平均勾配 (nabla_w と nabla_b) に学習率 (eta) を掛けた値を引くことで更新されます。`len(mini_batch)` はミニバッチのサイズで、これで割ることでミニバッチ全体の平均勾配を計算しています。

ChatGPT:

もちろんです！提供されたコードは、機械学習アルゴリズムの一部、特にニューラルネットワークのトレーニングプロセス中に重みとバイアスを更新するための確率的勾配降下法 (SGD) の実装のようです。ステップバイステップで分解してみましょう：

1. `self.weights` と `self.biases`: これらの変数は、おそらくニューラルネットワーククラスの一部であり、ネットワークの重みとバイアスを表しています。ニューラルネットワークにおいて、重みは異なる層のニューロン間の接続の強さを調整するパラメータであり、バイアスはニューロンの活性化閾値を制御するパラメータです。
2. `nabla_w` と `nabla_b`: これらの変数は、それぞれ重みとバイアスに対するコスト関数の勾配を保持していると考えられます。トレーニングプロセス中に、勾配が計算され、コスト関数が重みとバイアスに関してどのように変化するかを決定します。この情報は、コストを最小化するためにパラメータを更新するために使用されます。
3. `eta`: これは学習率で、パラメータ更新時に勾配の方向に進むステップサイズを制御します。学習率が高いと、訓練中の収束が速くなる可能性がありますが、不安定になることもあります。学習率が低いと安定性が向上するかもしれません、訓練時間が長くなる可能性があります。
4. `mini_batch`: これは、確率的勾配降下法 (SGD) の各イテレーションで使用されるトレーニングデータのサブセットです。ミニバッチ勾配降下法は、SGD の一種で、トレーニングデ

ータセット全体を小さなバッチに分割し、各ミニバッチを使用してパラメータの更新を行います。

それでは、コード自体を見てみましょう:

```
self.weights = [w - (eta / len(mini_batch)) * nw
                for w, nw in zip(self.weights, nabla_w)]
```

このコードは、ニューラルネットワークの重みを更新する部分です。各重み w に対して、勾配 $nabla_w$ に学習率 η とミニバッチのサイズ $\text{len}(\text{mini_batch})$ でスケーリングした値を引いています。これにより、重みが勾配降下法に従って更新されます。

この行は、ニューラルネットワークの重みを更新します。リスト内包表記を使用して、各重み w とそれに対応する勾配 nw をループ処理します。各重みについて、元の重みから勾配の一部（学習率 η とミニバッチのサイズでスケーリングされたもの）を引きます。これが、コスト関数を最小化するために重みが更新される仕組みです。

```
self.biases = [b - (eta / len(mini_batch)) * nb
               for b, nb in zip(self.biases, nabla_b)]
```

このコードは、ニューラルネットワークのバイアスを更新する部分です。具体的には、各バイアス b に対して、学習率 η とミニバッチのサイズ $\text{len}(\text{mini_batch})$ でスケーリングされた勾配 nb を引いています。この操作により、バイアスが勾配降下法に基づいて更新されます。

この行は、ニューラルネットワークのバイアスを同様の方法で更新します。リスト内包表記を使用して、各バイアス b とそれに対応する勾配 nb をループします。各バイアスについて、元のバイアスから勾配の一部（学習率 η とミニバッチのサイズでスケーリングされたもの）を引きます。

要約すると、これらのコード行は、確率的勾配降下法（SGD）を使用してニューラルネットワークの重みとバイアスを更新します。このプロセスは、トレーニングフェーズ中に各ミニバッチに対して繰り返され、ネットワークのパラメータを最適化し、与えられたタスクでのパフォーマンスを向上させます。

Zhiwei: SpaCy の言語モデルとは何ですか？

ChatGPT: SpaCy は、Explosion AI によって開発されたオープンソースの自然言語処理（NLP）ライブラリです。効率的で高速、かつ本番環境での使用に適しているように設計されており、さまざま

まな NLP タスクにおいて人気のある選択肢となっています。SpaCy は、テキストを処理し理解するための事前訓練済みの言語モデルを提供しており、開発者は品詞タグ付け、固有表現認識、文の解析などのさまざまな NLP タスクを実行することができます。

SpaCy 言語モデルの主な特徴は以下の通りです：

1. トークン化: SpaCy は、さまざまな言語規則や複雑さを考慮して、テキストを個々の単語やトークンに分割することができます。
2. 品詞タグ付け: このライブラリは、各トークンに品詞タグを割り当てることができ、それが名詞、動詞、形容詞などであるかを示します。
3. 固有表現抽出 (NER): SpaCy は、人名、組織名、場所、日付などのエンティティを識別し、カテゴリ分けすることができます。
4. 依存関係解析: 文の文法的構造を分析し、単語間の関係を決定することができます。
5. レンマ化 (Lemmatization) : SpaCy は、単語をその基本形または語幹形（レンマ）に変換することができます。
6. 単語ベクトル: このライブラリは、単語の意味を捉えた密な数値表現である単語ベクトル（単語埋め込みとも呼ばれる）を提供します。

SpaCy は複数の言語をサポートしており、さまざまな言語向けの事前学習済みモデルが提供されています。これにより、情報抽出、感情分析、チャットボット、言語翻訳など、さまざまな自然言語処理 (NLP) アプリケーションで広く使用されています。さらに、必要に応じて開発者が特定のデータセットでカスタムモデルをトレーニングすることも可能です。

Zhiwei: SpaCy の言語モデルとは何ですか？

ChatGPT: SpaCy は、Explosion AI によって開発されたオープンソースの自然言語処理 (NLP) ライブラリです。効率的で高速、かつ使いやすいように設計されており、さまざまな NLP タスクにおいて人気のある選択肢となっています。SpaCy は、複数の言語に対応した事前学習済みの言語モデルを提供しており、ユーザーはトークン化、品詞タグ付け、固有表現認識、依存関係解析などのタスクを実行することができます。

SpaCy の速度と効率の核心は、最適化された Cython の実装とデータ構造にあります。これにより、特に本番環境や実世界のアプリケーションに適しています。このライブラリは Python で書かれており、英語、ドイツ語、フランス語、スペイン語など、さまざまな言語の事前訓練済みモデルが付属しています。

SpaCy の言語モデルは、深層学習技術を活用して自然言語データの基礎となるパターンを学習します。これらのモデルは大規模なテキストコーパスで訓練され、学習された表現は様々な NLP タスクに利用できます。事前訓練されたモデルは、特定のデータに基づいてさらに微調整することで、ドメイン固有のタスクにおける性能を向上させることができます。

そのアーキテクチャと使いやすさから、SpaCy は NLP コミュニティの開発者や研究者の間で人気のある選択肢となっています。情報抽出、テキスト分類、固有表現認識、品詞タグ付け、構文解析などのタスクによく使用されます。さらに、SpaCy のモジュール設計により、ユーザーは既存の NLP ワークフローやパイプラインに簡単に統合することができます。

Zhiwei: 説明していただけますか？

```
class Llama:

    @staticmethod
    def build(
        ckpt_dir: str,
        tokenizer_path: str,
        max_seq_len: int,
        max_batch_size: int,
        model_parallel_size: Optional[int] = None,
    ) -> "Llama":
        if not torch.distributed.is_initialized():
            torch.distributed.init_process_group("nccl")
        if not model_parallel_is_initialized():
            if model_parallel_size is None:
                model_parallel_size = int(os.environ.get("WORLD_SIZE", 1))
            initialize_model_parallel(model_parallel_size)

        local_rank = int(os.environ.get("LOCAL_RANK", 0))
        torch.cuda.set_device(local_rank)

        # すべてのプロセスでシードは同じでなければならない
        torch.manual_seed(1)

        if local_rank > 0:
            sys.stdout = open(os.devnull, "w")
```

```

start_time = time.time()
checkpoints = sorted(Path(ckpt_dir).glob("*.pth"))
assert len(checkpoints) > 0, f"no checkpoint files found in {ckpt_dir}"
assert model_parallel_size == len(
    checkpoints
), f"Loading a checkpoint for MP={len(checkpoints)} but world size is {model_parallel_size}"
ckpt_path = checkpoints[get_model_parallel_rank()]
checkpoint = torch.load(ckpt_path, map_location="cpu")
with open(Path(ckpt_dir) / "params.json", "r") as f:
    params = json.loads(f.read())

```

コマンドを実行し、エラーログを取得します。

```
% torchrun --nproc_per_node 1 example_text_completion.py \
--ckpt_dir llama-2-7b/ \
--tokenizer_path tokenizer.model \
--max_seq_len 128 --max_batch_size 4
```

注: WindowsまたはMacOSでは現在リダイレクトはサポートされていません。

Traceback (most recent call last):

```

File "/Users/lzwjava/projects/llama/example_text_completion.py", line 55, in <module>
    fire.Fire(main)

File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/fire/core.py", line 141, in Fire
    component_trace = _Fire(component, args, parsed_flag_args, context, name)
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/fire/core.py", line 475, in _Fire
    component, remaining_args = _CallAndUpdateTrace(
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/fire/core.py", line 691, in _CallAndUpdate
    component = fn(*varargs, **kwargs)
File "/Users/lzwjava/projects/llama/example_text_completion.py", line 18, in main
    generator = Llama.build(
File "/Users/lzwjava/projects/llama/llama/generation.py", line 62, in build
    torch.distributed.init_process_group("nccl")
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/distributed_c10d.py", line 1
    default_pg = _new_process_group_helper(
File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/distributed_c10d.py", line 1
    raise RuntimeError("Distributed package doesn't have NCCL \"built in\"")
RuntimeError: Distributed package doesn't have NCCL built in
ERROR:torch.distributed.elastic.multiprocessing.api:failed (exitcode: 1) local_rank: 0 (pid: 23273) of 1
```

```
Traceback (most recent call last):
  File "/Users/lzwjava/anaconda3/bin/torchrun", line 33, in <module>
    sys.exit(load_entry_point('torch==2.0.1', 'console_scripts', 'torchrun')())
  File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/elastic/multiprocessing.py", line 18, in wrapper
    return f(*args, **kwargs)
  File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/run.py", line 794, in main
    run(args)
  File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/run.py", line 785, in run
    elastic_launch(
  File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/launcher/api.py", line 100, in elastic_launch
    return launch_agent(self._config, self._entrypoint, list(args))
  File "/Users/lzwjava/anaconda3/lib/python3.10/site-packages/torch/distributed/launcher/api.py", line 80, in launch_agent
    raise ChildFailedError(
torch.distributed.elastic.multiprocessing.errors.ChildFailedError:
=====
example_text_completion.py が失敗しました
-----
失敗内容:
<NO_OTHER_FAILURES>
-----
最初に観測された失敗の根本原因:
[0]:
時間      : 2023-08-03_15:35:37
ホスト      : 1.0.0.127.in-addr.arpa
ランク      : 0 (local_rank: 0)
終了コード: 1 (pid: 23273)
エラーファイル: <N/A>
トレースバック : トレースバックを有効にするには、以下を参照してください: https://pytorch.org/docs/stable/elastic.html
=====
```

エラースタックトレースを学びましょう。

Pytorch における Backend.NCCL の意味について ChatGPT に聞いてみましょう。

Zhiwei:

コード:

```

import torch
from transformers import LlamaTokenizer, LlamaForCausalLM

model_path = '.'

# model_path = 'openlm-research/open_llama_7b'

tokenizer = LlamaTokenizer.from_pretrained(model_path)
model = LlamaForCausalLM.from_pretrained(
    model_path, torch_dtype=torch.float16, device_map='auto',
)

prompt = 'Q: What is the largest animal?\nA:'
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
generation_output = model.generate(
    input_ids=input_ids, max_new_tokens=32
)
print(tokenizer.decode(generation_output[0]))

```

このコードは、指定されたプロンプトに対してモデルが生成するテキストを取得し、それをデコードして表示するものです。具体的には、以下の手順で動作します：

1. prompt に質問とその回答の開始部分を設定します。
2. tokenizer を使用して、プロンプトをモデルが理解できる形式の入力 ID に変換します。
3. model.generate メソッドを使用して、入力 ID に基づいて新しいトークンを最大 32 個生成します。
4. 生成された出力を tokenizer.decode でデコードし、人間が読める形式のテキストに変換します。
5. 最後に、生成されたテキストを表示します。

このコードは、特に自然言語処理タスクにおいて、モデルがどのように質問に答えるかを確認するために使用されます。

エラー:

```

/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/generation/utils.py:1445: UserWarning
  warnings.warn(
Traceback (most recent call last):
  File "/home/lzw/Projects/open_llama_3b/run.py", line 17, in <module>

```

```
generation_output = model.generate(
    ~~~~~
File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/utils/_contextlib.py", line 1
    return func(*args, **kwargs)
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/generation/utils.py", l
    return self.greedy_search(
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/generation/utils.py", l
    outputs = self(
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/modules/module.py", line 1
    return forward_call(*args, **kwargs)
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/models/llama/modeling_
    outputs = self.model(
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/modules/module.py", line 1
    return forward_call(*args, **kwargs)
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/transformers/models/llama/modeling_
    inputs_embeds = self.embed_tokens(input_ids)
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/modules/module.py", line 1
    return forward_call(*args, **kwargs)
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/modules/sparse.py", line 1
    return F.embedding(
    ~~~~~

File "/home/lzw/anaconda3/envs/llama/lib/python3.11/site-packages/torch/nn/functional.py", line 2210,
    return torch.embedding(weight, input, padding_idx, scale_grad_by_freq, sparse)
    ~~~~~

RuntimeError: すべてのテンソルが同じデバイス上にあることが期待されますが、少なくとも2つのデバイス、cuda:0とcpuが
```

修正方法は？

ChatGPT:

エラーメッセージが示しているのは、入力テンソル（`input_ids`）とモデル（`LlamaForCausalLM`）のデバイスタイプが一致していないということです。入力テンソルは CPU（`cpu`）上にあり、モデルは GPU（`cuda`）上にあります。この問題を解決するには、モデルを使用してテキストを生成する前に、入力テンソルがモデルと同じデバイス上にあることを確認する必要があります。

以下は、問題を修正するための変更後のコードです：

```
import torch
from transformers import LlamaTokenizer, LlamaForCausalLM

model_path = '.'

# model_path = 'openlm-research/open_llama_7b'

tokenizer = LlamaTokenizer.from_pretrained(model_path)
model = LlamaForCausalLM.from_pretrained(
    model_path, torch_dtype=torch.float16, device_map='auto',
)

prompt = 'Q: What is the largest animal?\nA:'
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
```

input_ids テンソルをモデルと同じデバイスに移動する

```
input_ids = input_ids.to(model.device)

generation_output = model.generate(
    input_ids=input_ids, max_new_tokens=32
)
print(tokenizer.decode(generation_output[0]))
```

このコードでは、`input_ids` テンソルをトークナイザーから取得した後、`input_ids = input_ids.to(model.device)` という行を追加しました。この行は、テキスト生成を行う前に、`input_ids` テンソルをモデルと同じデバイス（CPU または GPU）に移動させます。これにより、「すべてのテンソルが同じデバイス上にあることが期待されます」というエラーが解消されるはずです。

GPU 対応のマシンでこのコードを実行し、モデルが GPU 上にある場合、並列処理能力を活用するために、モデルと入力データの両方を GPU 上に保持するのが一般的に効率的です。

Zhiwei: 以下のコードを説明していただけますか？

```
def generate(
    self,
    prompt_tokens: List[List[int]],
    max_gen_len: int,
    temperature: float = 0.6,
    top_p: float = 0.9,
    logprobs: bool = False,
    echo: bool = False,
) -> Tuple[List[List[int]], Optional[List[List[float]]]]:
    params = self.model.params
    bsz = len(prompt_tokens)
    assert bsz <= params.max_batch_size, (bsz, params.max_batch_size)

    min_prompt_len = min(len(t) for t in prompt_tokens)
    max_prompt_len = max(len(t) for t in prompt_tokens)
    assert max_prompt_len <= params.max_seq_len
    total_len = min(params.max_seq_len, max_gen_len + max_prompt_len)
```

上記のコードは、プロンプトトークンの最小長と最大長を計算し、最大長が指定されたシーケンス長を超えていないことを確認します。その後、生成するトークンの最大長とプロンプトの最大長を加えた値と、指定されたシーケンス長のうち小さい方を `total_len` として設定します。

```
pad_id = self.tokenizer.pad_id
tokens = torch.full((bsz, total_len), pad_id, dtype=torch.long, device="cuda")
for k, t in enumerate(prompt_tokens):
    tokens[k, :len(t)] = torch.tensor(t, dtype=torch.long, device="cuda")
if logprobs:
    token_logprobs = torch.zeros_like(tokens, dtype=torch.float)

prev_pos = 0
eos_reached = torch.tensor([False] * bsz, device="cuda")
input_text_mask = tokens != pad_id
for cur_pos in range(min_prompt_len, total_len):
```

```

logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)

if logprobs:
    token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy(
        input=logits.transpose(1, 2),
        target=tokens[:, prev_pos + 1 : cur_pos + 1],
        reduction="none",
        ignore_index=pad_id,
    )

if temperature > 0:
    probs = torch.softmax(logits[:, -1] / temperature, dim=-1)
    next_token = sample_top_p(probs, top_p)
else:
    next_token = torch.argmax(logits[:, -1], dim=-1)

```

このコードは、テキスト生成モデルの推論ステップを表しています。以下に各ステップの説明を日本語で示します。

1. `prev_pos = 0`
前回の位置を 0 に初期化します。
2. `eos_reached = torch.tensor([False] * bsz, device="cuda")`
バッチ内の各シーケンスが終了記号 (EOS) に到達したかどうかを示すテンソルを初期化します。初期状態ではすべて False です。
3. `input_text_mask = tokens != pad_id`
パディングトークン (`pad_id`) でないトークンをマスクとして保持します。
4. `for cur_pos in range(min_prompt_len, total_len):`
最小プロンプト長から総長までの範囲でループを開始します。
5. `logits = self.model.forward(tokens[:, prev_pos:cur_pos], prev_pos)`
モデルに現在のトークンシーケンスを入力し、次のトークンのロジットを計算します。
6. `if logprobs:`
ログ確率を計算する場合、以下の処理を行います。
7. `token_logprobs[:, prev_pos + 1 : cur_pos + 1] = -F.cross_entropy(...)`
クロスエントロピー損失を使用して、各トークンのログ確率を計算し、`token_logprobs` に格納します。

8. if temperature > 0:
温度パラメータが0より大きい場合、以下の処理を行います。
9. probs = torch.softmax(logits[:, -1] / temperature, dim=-1)
ロジットを温度でスケーリングし、ソフトマックス関数を適用して確率分布を計算します。
10. next_token = sample_top_p(probs, top_p)
トップ-pサンプリングを使用して次のトークンを選択します。
11. else:
温度が0以下の場合、以下の処理を行います。
12. next_token = torch.argmax(logits[:, -1], dim=-1)
ロジットの最大値に対応するトークンを次のトークンとして選択します。

このコードは、テキスト生成モデルがどのようにして次のトークンを選択し、生成を進めるかを示しています。

```
next_token = next_token.reshape(-1)
# プロンプトが既に生成されている場合のみトークンを置き換える
next_token = torch.where(
    input_text_mask[:, cur_pos], tokens[:, cur_pos], next_token
)
tokens[:, cur_pos] = next_token
eos_reached |= (~input_text_mask[:, cur_pos]) & (
    next_token == self.tokenizer.eos_id
)
prev_pos = cur_pos
if all(eos_reached):
    break

if logprobs:
    token_logprobs = token_logprobs.tolist()
out_tokens, out_logprobs = [], []
for i, toks in enumerate(tokens.tolist()):
    # 最大生成長に切り詰める
    start = 0 if echo else len(prompt_tokens[i])
    toks = toks[start : len(prompt_tokens[i]) + max_gen_len]
    probs = None
```

```

if logprobs:
    probs = token_logprobs[i][start : len(prompt_tokens[i]) + max_gen_len]
    # EOS トークンがあればそこで切り詰める

    if self.tokenizer.eos_id in toks:
        eos_idx = toks.index(self.tokenizer.eos_id)
        toks = toks[:eos_idx]
        probs = probs[:eos_idx] if logprobs else None
    out_tokens.append(toks)
    out_logprobs.append(probs)

return (out_tokens, out_logprobs if logprobs else None)

```

ChatGPT: もちろんです！ 提供されたコードは、`generate` というメソッドで、プロンプト（入力テキスト）を受け取り、言語モデルを使用してテキストを生成します。このメソッドは、特定のモデルとトークナイザーと連携するように設計されており、貪欲サンプリングまたはトップ-pサンプリングを使用してテキスト生成を行います。以下に、このメソッドの動作の詳細を説明します。

1. メソッドのパラメータ:

- `prompt_tokens`: トークン化されたプロンプトのリスト。各プロンプトは整数のリスト（トークン ID）として表現されます。
- `max_gen_len`: 生成される出力の最大長（トークン単位）。
- `temperature`: 生成のランダム性を制御するパラメータ。高い値（例: 1.0）は出力をよりランダムにし、低い値（例: 0.6）はより決定論的にします。
- `top_p`: 生成されるテキストの多様性を制御するパラメータ。高い値（例: 0.9）はより多様なサンプリングを可能にします。
- `logprobs`: `True` の場合、生成されたトークンの対数確率も返されます。
- `echo`: `True` の場合、生成された出力にはプロンプト自体も含まれます。

2. **初期化:** このメソッドは、`params`、`bsz`（バッチサイズ）、`min_prompt_len`、`max_prompt_len` などの変数を初期化します。また、プロンプトを含む生成された出力の総長さである `total_len` を計算します。

3. **トークン生成:** このメソッドは、出力テキストのトークンを生成します。プロンプトの終わりから始まり、前に進む形で生成シーケンス内の位置を反復処理するループを使用します。

- 各位置のロジットは、部分シーケンスをモデルのフォワード関数に渡すことで取得されます。

- `temperature` が 0 より大きい場合、ロジットは指定された温度値を使用してソフトマックスで確率に変換されます。`temperature` が 0 の場合、`torch.argmax` を使用して最も高いロジットを持つトークンが選択されます。
- 次のトークンは、生成された確率または `argmax` に基づいて決定され、`tokens` テンソルの対応する位置に配置されます。
- EOS (シーケンス終了) トークンが検出され、すべてのシーケンスが EOS トークンに到達した場合、生成プロセスが停止されます。
- 生成されたトークンは `tokens` テンソルに保持されます。

4. 生成された出力の処理:

- 生成されたトークンは、指定された最大長 (`max_gen_len`) に合わせてトリミングされます。
 - `echo` が `True` の場合、プロンプトが生成された出力に含まれます。
 - `eos` トークンが検出された場合、生成されたテキストはその時点で切り捨てられます。
5. 戻り値: このメソッドは、生成されたトークンを含むタプルを返します。もし `logprobs` が `True` の場合、生成されたトークンの対数確率も含まれます。

このメソッドは、プロンプトによって提供されるコンテキストと、以前に生成されたトークンに基づいて、トークンを反復的に予測することでテキストを生成します。生成されるテキストのランダム性と多様性を制御するために、指定された生成パラメータを使用します。このメソッドは、特定のモデルアーキテクチャとトークナイザーと共に使用するように設計されており、テキスト生成のためにそれらのプロパティとメソッドに依存しています。
