Approach to Implementing Speculative Decoding

I followed the hints provided that helped me step-by-step implement Speculative Decoding.

- **Initialize the Target and Draft Language Model:** Initializing the draft model and the target model follows the standard syntax of using the `from_pretrained` method from the `AutoModelForCausalLM` class. The same goes for the tokenizer.
- **Generate Speculative Tokens with the Draft Model:** I simply use the initialized draft model to generate the next token using `input_ids`, `attention_masks`, and hyperparameters like `num_speculative_tokens`.
- **Efficiently Verify Draft Tokens with the Target Model:** I concatenate the `input_ids` and `draft_tokens`, generate the corresponding new attention mask, and then simply call the target model for the output logits (before softmax predictions). Here, I use KV caching to speed up inference (more on it later). This avoids recomputing the KV values for each token. Since the sequence length and `num_speculative_tokens` param for the test cases were small, this can efficiently work in Colab memory. This is easily done using the `use_cache=True` parameter in models from the Transformers library. Once I get the logits, I apply argmax to get the next predicted token (this is greedy) and match my predictions from the target model with the draft tokens to see which ones are accepted.
- **Implement the Main Speculative Decoding Algorithm:** Here, I maintain a loop with a check on total generated tokens and <EOS> token. At each iteration, I generate draft tokens and call the verify method to compare the generated tokens through the target model. Then, I check if the draft tokens are accepted or not. If yes, I modify the `input_ids` and the `attention_mask` to include them and continue the process.

Challenges Encountered and How I Addressed Them

Yes, the main challenge was making the `verify_tokens_vectorized` method work. There were several nuances to this. One of the areas I got stuck on was using incorrect slices here: `logits[:, og_shape_ip:, :]`. After rigorous debugging, I realized it should be `logits[:, og_shape_ip-1:-1, :]` because we are predicting the next token at each iteration. This shift in indexing fixed the alignment between the target model's predictions and the draft tokens, ensuring accurate verification.

Optimizations Implemented

I mainly played around with two areas for optimization. One is the **precision**, and the other is **cache usage**. I added two flags to the class: one for precision and another for cache. This helped me play around with different kinds of precision **(FP16, FP32, BF16)** and see the impact caching has. Using low FP resulted in a lower draft token acceptance rate, but I had to play with a bunch of values to find the right balance between speed and accuracy, especially for the bonus part.

Performance results and analysis

```
target_model_name = "EleutherAI/pythia-1.4b-deduped"
draft_model_name = "EleutherAI/pythia-160m-deduped"
```

Using <KV-Caching & FP16> V/S <no caching & FP16>

I achieved a ***> 90%*** **token acceptance** rate for all three prompts with FP16 and Caching enabled. Prompt 1: 90.00%, Prompt 2: 97.78%, Prompt 3: 91.11%.

- **Faster Decoding Time:** With caching, the average speculative decoding time dropped compared to no caching. For Prompt 1, it went from 1.60s to 1.51s (5.6% faster). For Prompt 2, it dropped from 1.48s to 1.32s (10.8% faster). For Prompt 3, it went from 1.47s to 1.34s (8.8% faster). Caching doesn't recompute KV values for the whole sequence every time. This saves a lot of time, especially since only the new speculative tokens need processing.
- **Better Speedup and Latency Reduction:** Speedup against the baseline improved with caching. Prompt 1 went from 1.23x to 1.47x, Prompt 2 from 1.31x to 1.67x, and Prompt 3 from 1.32x to 1.49x. That's a noticeable jump—up to 27% better speedup for Prompt 2. Latency reduction also got better: Prompt 1 from 18.74% to 32.07%, Prompt 2 from 23.94% to 40.13%, and Prompt 3 from 24.08% to 32.90%.
- **No Change in Acceptance Rate:** Caching speeds up the target model's forward pass but doesn't affect how well the draft model's tokens match the target's predictions.

Using <FP32 & caching>: The results were less impressive compared to FP16. For Prompt 1, speculative decoding averaged 37.92 tokens per second vs. 38.73 for the baseline, a 0.97x speedup compared to the baseline, with a -2.97% latency increase. Prompt 2 managed 41.80 tokens per second vs. 38.66, a modest 1.12x speedup and 10.35% latency reduction. Prompt 3 hit 39.02 tokens per second vs. 39.81, a 1.08x speedup and 7.00% latency reduction. Acceptance rates stayed the same (90-97%), so the draft model wasn't the issue—FP32 just slowed things down compared to FP16.

Bonus Implementation: For the bonus question, I tried the GPT family of models to check the impact of speculative decoding.

```
target_model_name = "gpt2-xl"
draft_model_name = "gpt2"
```

Using <KV-Caching & FP16> V/S <KV-Caching & FP32>

For gpt2-xl and gpt2 with caching, FP32, and FP16 both do better than the baseline. FP16 is way faster (47.00-59.22 tokens/sec), with 1.77x-2.34x speed boosts and 43-57% less wait time, beating FP32's times (1.60-2.03 sec vs. 10.27-11.36 sec). FP16 wins for speed. With this pair, I achieved **>1.7x speedup** over baseline for all three prompts for both FP16 and FP32.

Draft token acceptance rates were high in both setups, but FP32 edged out FP32 marginally. FP32 hit 94.44% (Prompt 1), 81.00% (Prompt 2), and 72.73% (Prompt 3), while FP16 matched 94.44% (Prompt 1) but dropped to 76.36% (Prompt 2) and 72.73% (Prompt 3)—a small dip for Prompt 2 (81.00% vs. 76.36%). This suggests FP32 might keep the draft model slightly more aligned with the target, though the difference is tiny.