



Bryn: 3d, this is the big one. I want to do a research pass. Package 3d as a question, explain what we're trying to do and detail the data we have to do it with. Include data types. I'll pass that upstairs to research who will give us the precise details we need.

Gemini: Okay, let's package Step 3d as a clear research request.

Research Request: Step 3d - Single-Pass Diffusion Repair

Project Context:

We are building the "Positronic Brain," a continuous inference system featuring "Halo Weave" context management. The system maintains coherence over long interactions by selectively managing context based on token "brightness" (derived from attention scores). When tokens fall below a certain brightness threshold, they are culled (removed conceptually).

Step 3d Goal:

This step aims to "repair" the context *after* tokens have been identified for culling in the previous steps (3a, 3b). For each culled token position, we need to use a diffusion-style model (likely a Masked Language Model like DistilBERT or a specialized diffusion model like MDLM) to suggest the *best replacement token* for that specific position, based on the surrounding context window.

Crucial Constraints:

- Single Pass:** This repair must happen in a *single forward pass* of the chosen repair model per window. We are *not* performing iterative denoising or sampling.
- Suggestion, Not Generation:** The goal is to get the single most likely token suggestion for the culled position, not to generate new text.
- Brightness Guidance:** The process should leverage the provided brightness map for the context window, potentially influencing the repair process (e.g., biasing attention, modulating noise/masking).
- Diff Output:** The final output must be a list of *changes* (diffs) in the format ``(global_position, old_token_id, new_token_id)``, including only positions where the suggested token differs from the original culled token.

Input Data (Provided by Step 3c):

The input to Step 3d is a *list* of dictionaries, where each dictionary represents one culled token and its surrounding context window.

```
`python
diffuser_input_list: List[Dict]`
```

Each dictionary in the list has the following structure:

```
`python
window_input_data = {
    'global_cull_position': int, # The original index in the full sequence that was culled.
    'window_token_ids': torch.Tensor, # Shape: [actual_window_len], dtype: torch.long. Token IDs for the window.
    'window_attention_mask': torch.Tensor, # Shape: [actual_window_len], dtype: torch.long or torch.float32. Attention mask for the window.
    'window_brightness_map': torch.Tensor, # Shape: [actual_window_len], dtype: torch.float32. Brightness map for the window.
    'masked_position_local': int, # The index *within the window* that corresponds to the culled token.
    'original_token_id_at_mask': int # The token ID that was originally at the masked_position_local.
}
```

* `actual_window_len` is the size of the extracted window (typically ≤ 511 , depending on proximity to sequence boundaries).

****Core Problem for Research:****

For each `window_input_data` dictionary in the `diffuser_input_list`:

1. How do we prepare the `window_token_ids` and `window_brightness_map` as input for a suitable repair model (e.g., DistilBERT, MDLM)? Should we use embeddings directly? Should we apply masking or noise based on brightness?
2. How exactly should the `window_brightness_map` be used to guide the single-pass prediction for the `masked_position_local`? (e.g., modify input embeddings, bias attention, apply selective masking).
3. What is the precise procedure for performing the single forward pass of the chosen repair model?
4. How do we efficiently extract the highest probability token prediction *only* for the `masked_position_local` from the model's output (likely logits)?
5. How do we compare this predicted token ID against the `original_token_id_at_mask` to determine if a change occurred?
6. How do we construct the final output `List[Tuple[int, int, int]]`, ensuring the `global_cull_position` is used for the position index in the tuple?

****Specific Questions:****

1. Given the single-pass, suggestion-only requirement, what specific model architecture (e.g., DistilBERT, a specific MDLM variant, DiffusionBERT) is most suitable and efficient?
2. What is the most effective way to incorporate the `window_brightness_map` into the single forward pass to guide the repair towards coherent, low-perplexity suggestions, drawing from the concepts in `Generative Refinement of Text via Diffusion Models.pdf` (like modulating noise or biasing based on information content/brightness)?
3. Are there existing Hugging Face `transformers` pipelines or model methods that can be adapted for this single-token prediction task with brightness guidance, or does this require a custom implementation loop around a base MLM?
4. What are the exact inputs the chosen model expects (raw IDs, embeddings, specific mask format)?
5. What is the recommended implementation strategy (pseudo-code or high-level steps) for processing each input window and generating the final diff list?

We need a concrete, implementable procedure for Step 3d based on the provided input data structure and the outlined constraints.

Thanks for the detailed setup. Before I begin the research, could you please confirm:

1. Do you already have a preferred model shortlist (e.g., DistilBERT, BERT-base, DiffusionBERT, or MDLM), or should I explore a range including cutting-edge diffusion-style masked language models?
2. Is the brightness guidance expected to be implemented in a soft way (e.g., modifying attention weights or input embeddings) or as hard masking or dropout?
3. Do you require the proposed method to be easily runnable in Hugging Face Transformers without needing model retraining, or is custom training/fine-tuning acceptable?
4. What is your desired runtime performance goal for this step (e.g., ms/token, batch size)?

Once I have this, I'll proceed with a research pass tailored to your setup.



Bryn: No training. That's beyond our scope. Working badly is better than well trained right now. We can customize batch size to the model. Soft brightness would be ideal but we'll accept a hard cutoff at this point in development. Our preferred model would have a tokenizer compatible with tinyllama, our ideal would be able to support the raw embeddings but that's probably beyond scope as well unless we just get lucky.

Gemini, do you have anything to add to my response?

Gemini: Okay, Bryn, your constraints are clear and very helpful for directing the research focus.

My additions/clarifications to your response for O3 would be:

1. **Highlight the Tokenizer Incompatibility:** This is the most immediate technical hurdle based on your requirements.
 - * "While TinyLlama uses a SentencePiece-based tokenizer (like Llama), the readily available pre-trained Masked Language Models (MLMs) like DistilBERT, BERT, RoBERTa, etc., typically use WordPiece or BPE tokenizers with different vocabularies."
 - * "This means the token IDs from TinyLlama (our main model) won't directly correspond to the token IDs understood and predicted by a standard pre-trained MLM like DistilBERT. The `old_token_id` we identify from our context won't match the MLM's vocabulary, and the `new_token_id` predicted by the MLM won't directly map back to a TinyLlama token ID without potentially lossy conversion or using the MLM's own tokenizer for the replacement."
2. **Confirm the Implication of "No Training":**
 - * "Given the strict 'no training' constraint, we cannot fine-tune an MLM on TinyLlama's vocabulary or fine-tune TinyLlama itself for MLM tasks. We *must* use an existing, pre-trained MLM off-the-shelf."
3. **Propose Handling the Mismatch (Leading Question):**
 - * "Therefore, the core question for research becomes: How can we best use a pre-trained MLM (like DistilBERT, chosen for speed) for single-pass token *suggestion* at specific low-brightness positions, *accepting* that the suggested token ID will come from the MLM's vocabulary, not TinyLlama's? Is the goal primarily semantic coherence (filling the gap plausibly), even if the token ID itself isn't a perfect match for the original vocabulary?"
 - * "Alternatively, is there any known pre-trained MLM that *does* share TinyLlama's tokenizer/vocabulary, or is the vocabulary mismatch an acceptable trade-off for getting *any*

functional repair mechanism working quickly?"

4. ****Reiterate Brightness Approach:****

* "Confirming the brightness guidance: an initial implementation can use a hard cutoff. We will use the brightness map simply to *select which positions* to mask and query the MLM for, rather than implementing soft guidance initially."

5. ****Confirm Token IDs (Not Embeddings):****

* "Confirming we should proceed using token IDs as the primary interface for the MLM, not raw embeddings, due to the complexity and model differences."

Essentially, we need O3 to focus the research on using a standard, pre-trained MLM like DistilBERT for ****single-token prediction**** at specified indices within a window, using brightness primarily for ****selection/masking****, and acknowledging the unavoidable ****tokenizer/vocabulary mismatch**** with TinyLlama. The research should detail the best way to perform this single-pass prediction and extract the suggested token ID from the MLM's vocabulary.

Great. I'll begin a research pass to identify the best method for performing single-token repair using an off-the-shelf, pre-trained Masked Language Model (e.g., DistilBERT) in a single forward pass. This approach will account for a hard brightness threshold (no soft guidance yet), operate without any model training or fine-tuning, and work under the constraint that token IDs are from TinyLlama's vocabulary, which is incompatible with most MLMs.

The goal is to identify a practical and implementable method to repair low-brightness tokens by predicting suitable replacements from a different tokenizer's vocabulary and constructing diffs accordingly.

I'll report back with models, proposed procedures, and workarounds for the tokenizer mismatch.



Single-Pass Diffusion Repair via Off-the-Shelf MLM

Single-Pass Token Repair with a Pre-trained Masked LM

Overview

In a long-sequence inference system (the *Positronic Brain*), certain tokens are "culled" based on low attention-derived brightness scores. The goal is to repair each removed token in **one pass** using a pre-trained Masked Language Model (MLM) – **without** any fine-tuning. We will insert a mask at each culled position and let the MLM predict a single replacement token, then compare it to the original. This procedure must handle mismatched tokenizers (TinyLlama vs. the MLM's) and use minimal custom code by leveraging Hugging Face's standard APIs.

Below, we address the core questions step by step: choosing an efficient MLM, constructing masked inputs, obtaining predictions, interpreting differences, handling vocabulary mismatches, useful Hugging Face utilities, and finally an implementation outline.

Choosing an Efficient Pre-trained MLM

For fast single-token predictions, small **BERT-based** MLMs are ideal. A few strong candidates available on Hugging Face include:

- **DistilBERT** – A distilled BERT model with ~40% fewer parameters than BERT-base (66M vs. 110M) and ~60% faster inference, while retaining over 95% of BERT’s language understanding performance huggingface.co. DistilBERT is well-suited for quick masked token predictions.
- **MiniLM** – A very compact model (e.g. 12 layers, 384 hidden size, ~33M params) distilled from BERT. MiniLM runs ~2.7× faster than BERT-base, making it efficient for fill-mask tasks. Versions like `microsoft/MiniLM-L12-H384-uncased` or even a 6-layer variant (`MiniLM-L6`) trade minimal accuracy for speed.
- **ALBERT** – An “ALite BERT” with heavy parameter sharing (only ~11M parameters for ALBERT-base). It’s memory-efficient but uses 12 repeating layers, so **inference speed** is not much improved (since it still processes 12 layers). ALBERT can be used if memory is critical, but its speed gain is less than DistilBERT or MiniLM.
- **TinyBERT or MobileBERT** – Other distilled models (TinyBERT, MobileBERT) exist, but DistilBERT and MiniLM are more commonly supported via Hugging Face. For simplicity and broad support, DistilBERT (English uncased) is a top choice.

Recommendation: Use **DistilBERT-base-uncased** (or a similar small MLM). It provides fast, single-mask predictions with minimal accuracy loss huggingface.co. If even smaller, MiniLM can be tried for slightly faster inference. These models come with convenient Hugging Face pipelines for masked token filling.

Preparing Input Windows with Mask Tokens

Each culled token is given in a context *window* (a dictionary with token IDs, mask position, etc.). To use an off-the-shelf MLM on this data, we need to create a **masked input sequence** that the MLM’s tokenizer and model can accept:

- **Token ID Incompatibility:** TinyLlama uses Llama2’s SentencePiece tokenizer, whereas BERT-based models use WordPiece or BPE. This means the `window_token_ids` (TinyLlama vocab IDs) **cannot be fed directly** to DistilBERT or similar – the IDs and vocabularies are different. For now, we *ignore* this mismatch by reconstructing the sequence in a form the MLM understands (explained next).
- **Reconstructing the Text:** Ideally, convert the `window_token_ids` back to text using TinyLlama’s tokenizer. This yields the actual sequence of words/subwords in that window. For example, if the window corresponds to text “The cat sat on the mat” and “sat” was removed, we’d reconstruct `"The cat [MASK] on the mat"`. Using the original text ensures semantic consistency.
- **Inserting the `[MASK]`:** At the `masked_position_local` within the window, insert the special mask token for the MLM. In a text representation, one can literally insert the string `"[MASK]"` at that position. If using Hugging Face’s tokenizer directly, one can also insert the mask token ID. For instance, with a BERT tokenizer:
 - Find the token index corresponding to `masked_position_local` in the TinyLlama sequence.

- Remove or mark that token, and insert the MLM's mask token (DistilBERT uses `[MASK]` which the tokenizer will convert to its `mask_token_id`).
- **Special Tokens and Attention Mask:** Ensure the sequence is properly bounded. BERT-style models expect `[CLS]` at start and `[SEP]` at end; Hugging Face tokenizers add these by default when encoding text. The provided `window_attention_mask` indicates which parts of `window_token_ids` are real tokens vs. padding. When reconstructing text, ignore pads (they won't appear in text). The MLM's tokenizer will generate its own attention mask. Essentially, we feed the MLM only the actual context with one `[MASK]` in place of the culled token.

Example:

Suppose `window_token_ids` decoded to: "A quick brown fox jumps over the lazy dog", and the token "fox" was culled. We create the masked input "A quick brown [MASK] jumps over the lazy dog". The MLM tokenizer will produce IDs for `[CLS] A quick brown [MASK] jumps over the lazy dog [SEP]` (if using BERT/DistilBERT). The mask token sits exactly where fox was.

This way, we leverage the MLM's own vocabulary and tokenizer for the input. We temporarily *sidestep vocabulary mismatch* by operating in the space of actual text.

Single-Pass Inference to Predict the Masked Token

With the input prepared, we can perform a single forward pass through the MLM to get the masked token prediction:

- **Encode Input:** Use the MLM's `AutoTokenizer` to encode the text with the mask. For example:

```
python

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
inputs = tokenizer(masked_text, return_tensors="pt")
```

This returns `input_ids` and `attention_mask` suitable for the model.

- **Forward Pass:** Pass the inputs to the `AutoModelForMaskedLM`. For example:

```
python

model = AutoModelForMaskedLM.from_pretrained("distilbert-base-uncased")
outputs = model(**inputs)
logits = outputs.logits
```

This obtains the **logit scores** for each vocabulary token at each position in the sequence.

- **Locate Mask Position:** Identify the index of the `[MASK]` token in the input. We can find it by:

```
python

mask_index = (inputs["input_ids"] == tokenizer.mask_token_id).nonzero(as_tuple=True)
```

In a batch of size 1, this yields the index (e.g., `(0, position)` for batch 0, some position). Extract that position (let's call it `m_idx`).

- **Top-1 Prediction:** Examine the logits at `m_idx`. The logits tensor has shape `[batch_size, seq_length, vocab_size]`. We take `logits[0, m_idx, :]` which is the score distribution over

the MLM's vocabulary for the masked position. We then select the highest-scoring token:

```
python

mask_token_logits = logits[0, m_idx, :]
predicted_token_id = int(mask_token_logits.argmax(dim=-1))
```

This gives the **top-1 predicted token ID** from the MLM's vocabulary. (If multiple masks were present, we'd do this for each, but here it's one masked token per window.)

- **Single-Pass Only:** We do **not** iterate or refine the prediction – one forward pass per window yields one prediction. This keeps the repair “single-pass” and minimal in overhead.

Using Hugging Face's high-level pipeline API, this process can be even simpler. The `fill-mask` pipeline directly returns the top predictions:

```
python

from transformers import pipeline
mask_filler = pipeline("fill-mask", model="distilbert-base-uncased")
result = mask_filler("A quick brown [MASK] jumps over the lazy dog", top_k=1)
pred_token_str = result[0]['token_str'] # e.g., 'fox'
pred_token_id = result[0]['token_id'] # e.g., 18622 (MLM vocab ID)
```

However, for batch processing and to get the raw token IDs, using the model and tokenizer directly (as above) is straightforward and efficient.

Comparing Predicted vs. Original Tokens

After obtaining the MLM's top prediction for the masked token, we must interpret it relative to the original token that was removed:

- **Decode Tokens to Strings:** Because the original and new token IDs come from different vocabularies, direct numeric comparison is meaningless. Instead, convert each to a textual form:
 - Use TinyLlama's tokenizer to decode the `original_token_id_at_mask` to the actual token string (e.g., the word or subword that was removed).
 - Use the MLM's tokenizer to decode the `predicted_token_id` to its token string (e.g., `tokenizer.decode([predicted_token_id])` for BERT).
- **Check if Token Matches:** If the decoded strings are *identical*, it means the MLM predicted essentially the same token (e.g., original was `"fox"` and MLM also predicted `"fox"`). In that case, there is no real change to log – the model successfully recovered the same token. If the strings differ, the repair changed the token.
 - *Note:* Because of tokenizer differences, even if the strings match, the underlying numeric IDs will differ (since one is TinyLlama's ID and one is BERT's ID for the same word). We consider that *not a change* in content.
- **Logging the Diff:** For each difference, record a tuple `(global_position, old_token_id, new_token_id)`:

- ``global_position`` is the position of this token in the overall sequence (provided as ``global_cull_position`` in the input).
- ``old_token_id`` is the original TinyLlama token ID that was removed.
- ``new_token_id`` is the MLM's predicted token ID (from the MLM's vocab) that we are using as the repair. We include this tuple only if the predicted token *differs* from the original token. Essentially, this becomes a "diff" list of changes.
- **Example:** If at ``global_position` = 42`, the removed token was TinyLlama ID 7351 (which might decode to ``"fox"``), and DistilBERT predicted token ID 198 at the mask (which decodes to ``"dog"``), we would log ``(42, 7351, 198)`` indicating the token at position 42 changed from "fox" to "dog". If it predicted something that also means "fox" (even if a different subword piece), we might interpret it as no change if the word is the same.

In summary, we interpret "mismatch" at the **semantic/token level**, not by raw ID. This acknowledges the tokenizer mismatch – we care about the content of the token, not the numeric ID itself in different vocabularies.

Vocabulary Mismatch and Possible Solutions

One challenge is that TinyLlama's vocabulary (Llama2's SentencePiece) and the MLM's vocabulary (e.g., BERT's WordPiece) are different. There are a few considerations and workarounds:

- **No Off-the-shelf MLM with Llama Vocabulary:** Currently, there is no standard pre-trained masked LM that uses Llama2's exact tokenizer. LLaMA itself is an autoregressive model (no ``[MASK]`` concept). Models like **XLM-Roberta** use SentencePiece and have large vocabularies (~250k tokens) which might overlap many tokens with Llama's vocab, but they are not identical and XLM-R's vocab IDs won't directly map to Llama's.
- **Decode and Re-Encode Strategy:** The simplest solution (as described) is to go through actual text. Decode TinyLlama tokens to string, let the MLM predict a replacement word, then if needed you could encode that predicted word back into TinyLlama's vocabulary (to obtain a TinyLlama token id for reinsertion). This leverages semantic consistency via text, albeit with a small overhead of decoding and encoding.
- **Semantic Mapping:** If decoding to text is undesirable, one could attempt a *semantic embedding* approach. For example, use TinyLlama's embedding matrix and the MLM's embedding matrix: find the TinyLlama token's embedding and identify the closest token in the MLM's embedding space (or vice versa). However, this ignores context – it's like a blind dictionary lookup, which may not be as accurate as using the context-aware MLM prediction.
- **Vocabulary Alignment Research:** There is ongoing research on aligning or bridging vocabularies between models. For instance, techniques exist to align embedding spaces of two models with different vocabularies. Some methods (e.g., *UnifyVocab*) try to statistically map tokens from one vocab to another to handle vocabulary mismatches. These approaches, however, typically require additional computation or training and may be overkill for single-token repair.
- **Practical Workaround:** In practice, using text as the interchange format is reliable. After the MLM suggests a token (from its vocab), we can find the closest representation in TinyLlama's vocab by simply encoding the predicted word with TinyLlama's tokenizer. If the word is OOV for

TinyLlama (unlikely, given Llama2's large vocab), it would be split into sub-pieces – in such cases, a direct 1-to-1 token replacement isn't possible, but since we only aim for single-token repair, one might choose the first sub-piece or reconsider such replacements. Generally, common words will exist in both vocabularies (though with different IDs).

Key point: Because we are not fine-tuning or altering the MLM or TinyLlama, we accept that the **MLM's suggestion may not perfectly map** to TinyLlama's tokens in all cases. The simplest path is treating the MLM's output as authoritative for the word choice and using text conversion to bridge any gaps.

Hugging Face Utilities for Masked Token Prediction

Hugging Face's `transformers` library provides convenient tools to implement this repair procedure with minimal custom code:

- **AutoTokenizer & AutoModel:** These classes allow loading a tokenizer and model for masked LM easily by model name. For example:

```
python

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModelForMaskedLM.from_pretrained("distilbert-base-uncased")
```

This gives a tokenizer that knows `[MASK]` and a model that can predict masked tokens. No special configuration needed – the mask token ID (`tokenizer.mask_token_id`) is pre-defined in the tokenizer.

- **Fill-Mask Pipeline:** `pipeline("fill-mask", model=..., tokenizer=...)` sets up a high-level pipeline. You provide a string with `[MASK]` and it returns the top predictions with scores. This is great for quick checks. For example, `unmasker("Hello [MASK] world")` might return `[{'token_str': 'beautiful', 'score': 0.9, ...}, ...]`. Under the hood, it's doing exactly the steps of tokenization, model forward pass, and decoding. We can use `top_k=1` to get just the best prediction.
- **Batch Processing:** Hugging Face models can handle batches of inputs. If performance is critical, one could batch multiple masked sequences together (concatenate the windows in a batch) up to the model's max sequence length (typically 512 for BERT). The `AutoTokenizer` with `padding=True` and `return_tensors="pt"` can pad multiple sequences to the same length in a batch. Then a single model call yields predictions for all masks at once.
- **Utilities for Mask Index:** As shown in the documentation, we can find the mask token index programmatically (useful in case the position is needed for extracting logits manually). The pipeline handles this internally, but if writing our own loop, using `torch.argmax` on the logits for the mask position is straightforward.
- **Logging and Diff Creation:** Python's standard data structures suffice for collecting diffs. One can append tuples to a list as differences are found. Optionally, Python's `logging` module could be used to record details (like original token vs. predicted token string) for debugging or analysis. This is outside transformers' scope but useful for verifying the procedure works on example inputs.

In summary, the **transformers** library provides nearly all the needed pieces out-of-the-box. We mostly need to connect them: prepare the masked input, call the model, and interpret the output.

Implementation Outline (Pseudocode)

Finally, here's a high-level plan for implementing the single-pass token repair in code. This assumes we have the list of input dictionaries (``culled_windows``) as described and that we have access to TinyLlama's tokenizer for decoding:

1. **Initialize MLM Model:** Load a pre-trained masked language model and tokenizer (e.g., DistilBERT):

```
python

tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModelForMaskedLM.from_pretrained("distilbert-base-uncased")
model.to(device) # e.g., move to GPU if available for speed
```

2. **Prepare Results List:**

```
python

diffs = []
```

3. **Loop Over Culled Token Windows:** For each ``entry`` in ``culled_windows``:

1. **Reconstruct Context Text:** Use TinyLlama's tokenizer to decode ``entry["window_token_ids"]`` into a text string. Then insert the special ``[MASK]`` token at ``entry["masked_position_local"]``. For example:

```
python

text_tokens = tinyllama_tokenizer.convert_ids_to_tokens(entry["window_token_ids"])
text_tokens[ entry["masked_position_local"] ] = tokenizer.mask_token # "[MASK]"
masked_text = tinyllama_tokenizer.convert_tokens_to_string(text_tokens)
```

Alternative: If direct TinyLlama decoding is available: decode the whole sequence and replace the substring corresponding to the masked token with ``[MASK]``. Ensure spaces/punctuation are handled correctly.

2. **Tokenize for MLM:**

```
python

inputs = tokenizer(masked_text, return_tensors="pt").to(device)
```

This yields ``inputs["input_ids"]`` (with the MLM's special tokens and mask inserted) and ``inputs["attention_mask"]``. The sequence length should be within the MLM's limit (if not, truncate or skip this window).

3. **Forward Pass:**

```
python

outputs = model(**inputs)
logits = outputs.logits # shape: [1, seq_len, vocab_size]
```

4. Get Mask Index & Prediction:

```
python

mask_idx = (inputs["input_ids"] == tokenizer.mask_token_id).nonzero(as_tuple=True)[1].i
mask_logits = logits[0, mask_idx, :]
new_token_id = int(mask_logits.argmax(dim=-1))
```

Now `new_token_id` is the predicted token's ID in the MLM's vocabulary.

5. Compare with Original:

- Obtain `old_token_id = entry["original_token_id_at_mask"]` (the TinyLlama token that was removed).
- Decode both IDs to strings:

```
python

old_token_str = tinyllama_tokenizer.decode([old_token_id])
new_token_str = tokenizer.decode([new_token_id])
```

- If `new_token_str` (strip any special chars) is different from `old_token_str` (consider case or trailing spaces appropriately), then a change is detected.

6. Record Diff if Changed: If changed, append the tuple:

```
python

diffs.append(( entry["global_cull_position"], old_token_id, new_token_id ))
```

This uses the global position so the diff is anchored to the original sequence.

4. **Output:** After processing all windows, `diffs` will contain the list of all changed tokens with their positions and new IDs. This can be returned or logged as needed.

Notes on the Implementation:

- We assume each window has exactly one masked token. If for some reason multiple tokens were removed contiguously and need repair, this approach would need one mask per missing token. (Most standard MLMs can fill multiple masks, but often one at a time gives the best accuracy. The question specifies single-pass, so presumably one mask at a time.)
- Special care with punctuation or spaces: when reconstructing `masked_text`, ensure that we don't inadvertently join words. Using the TinyLlama tokenizer's own conversion to string should handle spacing (SentencePiece often uses a leading space marker on tokens – the tokenizer's convert_to_string should respect that).
- If TinyLlama's decode isn't available in the environment, an alternative is to map `window_token_ids` to actual text via a vocabulary lookup or by running them through the TinyLlama model's tokenizer (since TinyLlama uses Llama2's tokenizer, one could load Llama2's tokenizer from Hugging Face if available).

- We ignore cases where the MLM's predicted token might not cleanly correspond to one TinyLlama token. If such a case arises (e.g., MLM predicts a word that TinyLlama would encode as two subword tokens), one might decide to still treat it as a suggestion for that single position (perhaps replacing the one removed token with the first subword of the predicted word, though that may be sub-optimal). This is a corner case; usually common single tokens will map to common single tokens.

By following this plan, we use an off-the-shelf MLM in a **single forward pass per token** to repair missing tokens. We rely on the MLM's powerful contextual prediction to fill in a plausible token, and we carefully compare it with the original, logging any differences. This approach uses pre-trained models and standard tokenizer behaviors, avoiding any retraining and keeping the integration overhead low. Each repaired token is handled independently in one shot, making the process efficient for long sequences where only a few tokens were removed.



Sources