Utilizing Custom Vietnamese WordPiece Vocabularies with TensorFlow Text Tokenizers

1. Introduction

This report details the process of employing a custom-generated Vietnamese WordPiece vocabulary, created using TensorFlow Text tools, for text tokenization and detokenization within TensorFlow pipelines. The focus is on leveraging the tensorflow_text.BertTokenizer and tensorflow_text.WordpieceTokenizer to process Vietnamese text, a language characterized by diacritics and case sensitivity, which necessitates careful configuration during both vocabulary generation and tokenizer initialization. Effective tokenization is a crucial preprocessing step for training and deploying sophisticated language models, particularly transformer-based architectures like BERT.[1] The methods described ensure that the nuances of the Vietnamese language captured during vocabulary creation are correctly applied during subsequent text processing tasks.

2. Understanding the Vocabulary Generation Context

The foundation of effective tokenization lies in the vocabulary file (vietnamese_wordpiece.vocab in this context) generated from a representative corpus (your_vietnamese_corpus.txt). The Python script provided utilizes tensorflow_text.tools.wordpiece_vocab.bert_vocab_from_dataset to learn this vocabulary. Several parameters set during this generation phase critically influence how the tokenizer must be configured later:

- **lower_case=False**: This setting preserves the original casing of the text in the corpus. Consequently, "Việt" and "việt" might be treated as distinct tokens or lead to different subword segmentations in the vocabulary. The tokenizer must also be configured to respect case sensitivity.
- **normalization_form='NFC'**: Unicode normalization is essential for languages with complex characters like Vietnamese. NFC (Normalization Form Canonical Composition) ensures that characters with diacritics (e.g., 'ệ', 'ô', 'â') are treated as single, precomposed units. This consistency must be maintained during tokenization to match vocabulary entries correctly.
- **reserved_tokens**: Tokens like [PAD], [UNK], ,, ``` are explicitly included at the beginning of the vocabulary. The tokenizer needs to be aware of these, particularly the [UNK] token for handling out-of-vocabulary words.
- **bert_compatible_tokenizer=True**: This ensures the standard WordPiece behavior expected by BERT, including the use of a specific suffix indicator

(defaulting to '##') for subword continuations.

Understanding these settings is paramount because the tokenizer's configuration must precisely mirror them to ensure accurate lookup and segmentation based on the generated vocabulary.

3. Choosing and Initializing the Right Tokenizer

TensorFlow Text offers several tokenizers suitable for subword tokenization. For vocabularies generated in the BERT style, text.BertTokenizer and text.WordpieceTokenizer are the primary choices.[2]

- **text.BertTokenizer**: This is often the preferred choice as it provides a higher-level interface that bundles the core WordPiece algorithm with essential pre-tokenization steps.[4] These steps typically include text normalization (potentially lowercasing and accent stripping, depending on configuration) and basic tokenization (splitting text into words based on whitespace and punctuation) before applying WordPiece segmentation.[3] This aligns well with the standard BERT preprocessing pipeline.[5]
- **text.WordpieceTokenizer**: This is a lower-level interface that implements only the WordPiece algorithm itself.[2] It expects the input text to be already pre-tokenized (split into words) and potentially normalized.[3] While offering more control, it requires manual implementation of the pre-tokenization steps, which must be consistent with the assumptions made during vocabulary generation.

3.1. Initializing text.BertTokenizer

Proper initialization is crucial for BertTokenizer to work correctly with the custom Vietnamese vocabulary.

Code Example:

Python

```
import tensorflow as tf
import tensorflow_text as tf_text # Assuming tf_text is imported
import pathlib
import os
```

```
# Assume vocab file is in the same directory or provide full path
vocab_path = 'vietnamese_wordpiece.vocab'

# Check if vocab exists (optional but good practice)
if not pathlib.Path(vocab_path).is_file():
    raise FileNotFoundError(f"Vocabulary file not found at: {vocab_path}")

# Initialize BertTokenizer
try:
    bert_tokenizer = tf_text.BertTokenizer(
        vocab_lookup_table=vocab_path,
        token_out_type=tf.int64, # Output integer IDs for model input
        lower_case=False, # MUST match vocab generation setting
        # normalization_form='NFC', # Check TF Text version docs if this param exists and applies here when lower_case=False [4, 6]
        unknown_token='[UNK]', # Standard UNK token, must be in vocab
        suffix_indicator='##' # Standard WordPiece suffix, must match vocab generation
    )
    print("BertTokenizer initialized successfully.")
except Exception as e:
    print(f"Error initializing BertTokenizer: {e}")
    # Add further error handling or investigation as needed
```

Parameter Deep Dive & Consistency Check:

The accuracy of the tokenizer hinges on the precise alignment of its parameters with those used during vocabulary generation. Any mismatch can lead to incorrect tokenization, often manifesting as excessive [UNK] tokens or complete failure.

1. **vocab_lookup_table**: Must be the correct path to the vietnamese_wordpiece.vocab file.[4] This file contains the mapping from subwords to IDs.
2. **token_out_type**: Set to tf.int64 (or tf.int32) to output integer IDs suitable for model input.[4] Alternatively, tf.string can be used to output the subword strings themselves, which is useful for debugging.[5] The default is tf.int64.[6]
3. **lower_case**: This parameter **must** be set to False to match the vocabulary generation setting (lower_case=False in the user's script). The BertTokenizer applies basic tokenization *before* WordPiece, and this includes potential lowercasing if lower_case=True.[4] If the tokenizer lowercases input ("Việt" -> "việt") but the vocabulary was built without lowercasing (and potentially contains

"Việt" but not "việt"), the lookup will fail, resulting in [UNK] tokens or incorrect IDs.[4]

4. **normalization_form**: The vocabulary was generated using 'NFC'. While BertTokenizer's lower_case=True option applies NFD normalization [4], the behavior when lower_case=False regarding normalization needs careful verification based on the specific TensorFlow Text version. The underlying BasicTokenizer used by BertTokenizer *does* have a normalization_form parameter.[6] If BertTokenizer itself does not expose this parameter directly when lower_case=False, it might use a default normalization or none at all. If the internal normalization differs from the NFC used for vocabulary creation (e.g., if it defaults to NFD or no normalization), mismatches can occur for Vietnamese characters (e.g., NFC 'ệ' vs. NFD 'e' + combining marks). It is essential to consult the documentation for the specific TF Text version being used to confirm if and how NFC normalization can be enforced during BertTokenizer initialization when lower_case=False. If direct control isn't available, using WordpieceTokenizer with manual NFC normalization might be necessary.
5. **unknown_token**: Typically set to "[UNK]". This string must exist in the vocabulary file (which it should, as it was specified in reserved_tokens during generation).[4] When the tokenizer encounters a word or subword it cannot find in the vocabulary, it outputs this token (or its corresponding ID).[7]
6. **suffix_indicator**: Must match the indicator used during vocabulary generation (which was '##', the default for bert_compatible_tokenizer=True).[4] This marker signals that a subword is a continuation of the previous one. Mismatched indicators prevent correct subword identification and detokenization.
7. **split_unknown_characters**: Defaults to False. When False, any word containing even a single character not covered by the vocabulary's character set will be mapped entirely to [UNK].[4] Setting it to True would split such words into known subwords and individual unknown characters, which might be desirable in some niche cases but significantly alters tokenization behavior. For standard BERT processing, False is typical.

The strict requirement for parameter consistency arises because the vocabulary file is essentially a contract established under specific text processing conditions. The tokenizer must adhere to the exact same conditions when processing new text to correctly interpret and apply the rules encoded in the vocabulary. Deviating from these conditions (e.g., applying lowercasing when the vocabulary didn't, or using a different normalization form) breaks this contract, leading to failed lookups and incorrect tokenization.

3.2. Initializing text.WordpieceTokenizer (Alternative)

If finer control over pre-processing is needed, or if BertTokenizer's handling of normalization with lower_case=False is uncertain for a specific TF Text version, WordpieceTokenizer can be used.

Code Example:

Python

```
# Initialize WordpieceTokenizer
try:
    wordpiece_tokenizer = tf_text.WordpieceTokenizer(
        vocab_lookup_table=vocab_path,
        token_out_type=tf.int64,
        unknown_token='[UNK]',
        suffix_indicator='##',
        # Note: No lower_case or normalization_form parameters here [7]
        split_unknown_characters=False # Typically False
    )
    print("WordpieceTokenizer initialized successfully.")
except Exception as e:
    print(f"Error initializing WordpieceTokenizer: {e}")

# NOTE: Requires manual pre-tokenization (splitting & normalization) of input text before calling tokenize.
```

Explanation:

WordpieceTokenizer lacks parameters like lower_case and built-in normalization because it operates at a lower level.[2] It expects its input to be a tensor of strings that have *already* been appropriately normalized (to NFC in this case) and split into word-level tokens.[3] The user is responsible for performing these steps explicitly before calling wordpiece_tokenizer.tokenize(), ensuring the processing matches the vocabulary generation conditions.

4. Tokenizing Vietnamese Text

Once the tokenizer is initialized correctly, it can be used to convert Vietnamese text

into sequences of subword tokens.

4.1. The tokenize Method

The tokenize method is the core function for this conversion. It accepts a tf.Tensor or tf.RaggedTensor containing UTF-8 encoded strings and outputs a tf.RaggedTensor containing the corresponding subword tokens.[4] The output type (integer IDs or strings) depends on the token_out_type specified during initialization.

4.2. Code Example (BertTokenizer)

```
Python


# Sample Vietnamese sentences (ensure UTF-8)
vietnamese_sentences = tf.constant([
    "tôi là sinh viên đại học bách khoa.", # Lowercase start
    "Học tiếng Việt rất thú vị!", # Uppercase start, punctuation
    "Phở là món ăn ngon của Việt Nam." # Proper nouns, diacritics
])

# Ensure the bert_tokenizer was initialized successfully before proceeding
if 'bert_tokenizer' in locals():
    # Tokenize using the initialized BertTokenizer
    # Outputting integer IDs (token_out_type=tf.int64 was set)
    token_ids = bert_tokenizer.tokenize(vietnamese_sentences)
    print("Token IDs (RaggedTensor):\n", token_ids)

    # Example: Tokenize to see string subwords for inspection
    try:
        bert_tokenizer_string = tf_text.BertTokenizer(
            vocab_lookup_table=vocab_path,
            token_out_type=tf.string, # Output string subwords
            lower_case=False,
            # normalization_form='NFC', # If applicable and verified for the TF Text version
            unknown_token='[UNK]',
            suffix_indicator='##'
        )
        subword_strings = bert_tokenizer_string.tokenize(vietnamese_sentences)
```

```
        print("\nSubword Strings (RaggedTensor):\n", subword_strings)
    except Exception as e:
        print(f"\nError initializing string-output BertTokenizer: {e}")
else:
    print("BertTokenizer was not initialized successfully. Skipping tokenization.")
```

4.3. Code Example (WordpieceTokenizer - Requires Pre-processing)

Using WordpieceTokenizer requires explicit pre-processing steps that mirror the vocabulary generation conditions.

Python

```
# Ensure the wordpiece_tokenizer was initialized successfully
if 'wordpiece_tokenizer' in locals():
    # --- Manual Pre-processing Step ---
    # 1. Apply NFC Normalization (MUST match vocab generation)
    normalized_sentences = tf.strings.unicode_normalize(vietnamese_sentences, 'NFC')

    # 2. Split into words (e.g., using whitespace or a more sophisticated method)
    # Using simple whitespace splitting here for demonstration.
    # NOTE: This splitting MUST be consistent with how the vocab expects words.
    # BertTokenizer's internal BasicTokenizer performs more sophisticated splitting
    # (e.g., on punctuation).[6] Using tf_text.WhitespaceTokenizer is basic.
    # For closer parity with BertTokenizer's pre-processing, one might need
    # tf_text.UnicodeScriptTokenizer or custom regex splitting.[3, 9]
    whitespace_tokenizer = tf_text.WhitespaceTokenizer()
    word_tokens = whitespace_tokenizer.tokenize(normalized_sentences)
    # WordpieceTokenizer expects rank >= 2 input [batch, words]
    print("\nManually Split Words (RaggedTensor):\n", word_tokens)

    # --- Tokenize with WordpieceTokenizer ---
    wordpiece_token_ids = wordpiece_tokenizer.tokenize(word_tokens)
    print("\nToken IDs from WordpieceTokenizer (RaggedTensor):\n", wordpiece_token_ids)
else:
    print("WordpieceTokenizer was not initialized successfully. Skipping tokenization.")
```

4.4. Output Explanation and Vocabulary Role

The output of tokenize is typically a tf.RaggedTensor. For BertTokenizer applied to sentences, the shape is often [batch_size, num_wordpieces] after internal flattening.[5] For WordpieceTokenizer applied to pre-split words ([batch_size, num_words]), the output shape is [batch_size, num_words, (num_wordpieces)].[7]

The segmentation observed (e.g., "sinh viên" becoming [b'sinh', b'viên'] or perhaps [b'sinh', b'##viên'] if "viên" only appeared as a suffix in the training corpus) is entirely dictated by the contents of vietnamese_wordpiece.vocab.[7] Words or subwords encountered in the input text that are *not* present in this vocabulary file will be mapped to the [UNK] token (or its ID).[4]

This highlights the critical dependence of tokenization quality on the vocabulary itself. The vocabulary is generated by analyzing the your_vietnamese_corpus.txt file to identify common character sequences and build subword units that allow for efficient representation of that specific corpus.[2] If the corpus used for vocabulary generation was small, domain-specific, or otherwise unrepresentative of the Vietnamese text intended for tokenization later, the resulting vocabulary will lack coverage. Poor coverage inevitably leads to a high frequency of [UNK] tokens when processing new, unseen text.[4] Since [UNK] tokens represent lost information, their prevalence significantly degrades the performance of downstream natural language processing models. Therefore, ensuring the vocabulary is trained on a large, diverse corpus relevant to the target application is essential for effective tokenization.

4.5. Tokenization Examples Table

The following table illustrates expected tokenization outputs for sample Vietnamese sentences using a correctly configured BertTokenizer (assuming an appropriately trained vocabulary).

Original Sentence (Vietnamese)	Tokenizer Used	Output Subword Strings (token_out_type=tf.string)	Output Token IDs (token_out_type=tf.int64)	Notes
"tôi là sinh viên đại học bách khoa."	BertTokenizer	[[b'tôi', b'là', b'sinh', b'viên', b'đại', b'học', b'bách', b'khoa',	[] (Example IDs)	Demonstrates basic sentence tokenization.

		b'.']] (Example)		
"Học tiếng Việt rất thú vị!"	BertTokenizer	[[b'Học', b'tiếng', b'Việt', b'rất', b'thú', b'vị', b'!']] (Example)	[] (Example IDs)	Shows handling of case (preserved) and punctuation.
"Phở là món ăn ngon của Việt Nam."	BertTokenizer	[[b'Phở', b'là', b'món', b'ăn', b'ngon', b'của', b'Việt', b'Nam', b'.']] (Example)	[] (Example IDs)	Includes proper nouns and diacritics.
"Đây là từkhôngcótrong từđiển." (OOV Word)	BertTokenizer	[[b'Đây', b'là', b'[UNK]', b'.']] (Example)	[] (Example IDs, assuming 1 is ID for [UNK])	Demonstrates [UNK] token for Out-Of-Vocabulary words.
"Học tiếng Việt rất thú vị!"	WordpieceTokenizer*	[[[b'Học'], [b'tiếng'], [b'Việt'], [b'rất'], [b'thú'], [b'vị'], [b'!']]] (Example)	[[, , , , , ,]] (Example IDs)	Shows output structure when input is manually pre-split words (note extra dimension).

() Assumes input was manually normalized and split into words before passing to WordpieceTokenizer.*

5. Detokenizing Vietnamese Subword IDs

Detokenization reverses the process, converting sequences of token IDs back into readable text strings.

5.1. The detokenize Method

Both BertTokenizer and WordpieceTokenizer provide a detokenize method.[4] It takes a Tensor or RaggedTensor of integer token IDs (like those produced by tokenize with token_out_type=tf.int64) and returns a Tensor or RaggedTensor of reconstructed strings.[4]

5.2. Code Example

Python

```python
# Ensure token_ids from BertTokenizer exist and are valid
if 'bert_tokenizer' in locals() and 'token_ids' in locals() and isinstance(token_ids, tf.RaggedTensor):
    # Detokenize the integer IDs generated by BertTokenizer
    detokenized_text = bert_tokenizer.detokenize(token_ids)
    print("Detokenized Text (Tensor):\n", detokenized_text)
else:
    print("Skipping BertTokenizer detokenization due to missing tokenizer or token_ids.")

# Example using IDs from WordpieceTokenizer (if generated)
# Ensure the correct tokenizer instance and IDs are used
if 'wordpiece_tokenizer' in locals() and 'wordpiece_token_ids' in locals() and isinstance(wordpiece_token_ids, tf.RaggedTensor):
    # Detokenize requires rank >= 2 input
    detokenized_from_wordpiece = wordpiece_tokenizer.detokenize(wordpiece_token_ids)
    print("\nDetokenized Text from WordpieceTokenizer (RaggedTensor, pre-merge):\n", detokenized_from_wordpiece)

    # The output shape might be [..., words, 1]; merge dims for sentence reconstruction [7]
    # Note: This simple merge joins words with spaces, which might differ from original spacing.
    try:
        # Attempt to merge the last two dimensions
        merged_detokenized = detokenized_from_wordpiece.merge_dims(-2, -1)
        print("\nMerged Detokenized Text from WordpieceTokenizer (RaggedTensor):\n", merged_detokenized)
    except tf.errors.InvalidArgumentError as e:
        print(f"\nCould not merge dimensions for WordpieceTokenizer output: {e}")
        print("This might happen if the input to detokenize wasn't rank 3 or higher.")
    except Exception as e:
         print(f"\nAn error occurred during merging: {e}")

else:
    print("Skipping WordpieceTokenizer detokenization due to missing tokenizer or token_ids.")
```

5.3. Output Explanation and Lossiness

The detokenize method reconstructs words by looking up IDs in the vocabulary and joining subwords. The subword prefix (e.g., '##') is typically removed during this process.[11]

However, it's crucial to understand that for BertTokenizer, the tokenize/detokenize process is generally *not* perfectly reversible.[4] The initial basic tokenization performed by BertTokenizer.tokenize (like splitting on punctuation and whitespace, potential lowercasing) involves irreversible steps.[6] Consequently, BertTokenizer.detokenize might reconstruct the words accurately, but the spacing or punctuation might differ from the original input string.[11] For example, an input "word." might tokenize to ['word', '.'] and detokenize to b'word.' (note the space).

WordpieceTokenizer.detokenize, on the other hand, only reverses the WordPiece splitting itself.[4] If applied to the output of WordpieceTokenizer.tokenize, it should reconstruct the *words* that were fed into tokenize more faithfully. However, reconstructing the original sentence structure from these detokenized words still requires careful handling of spacing and punctuation, often involving joining the detokenized words, potentially with custom logic.[10]

Both detokenization methods assume that the input token IDs form a dense range corresponding to the vocabulary indices, typically [0, vocab_size).[4]

6. Best Practices and Advanced Considerations

- **Parameter Consistency Checklist**: Before deployment, verify alignment between vocabulary generation and tokenizer initialization:
  - vocab_lookup_table: Correct path?
  - lower_case: Matches generation (False in this case)? [4]
  - normalization_form: Consistent with generation ('NFC')? Verified for TF Text version if using BertTokenizer with lower_case=False? [4]
  - suffix_indicator: Matches generation ('##')? [6]
  - unknown_token: Exists in vocab ('[UNK]')? [4]
  - reserved_tokens: Consistent handling expected?
- **Using tokenize_with_offsets**: For tasks requiring mapping tokens back to the original text span, methods like BertTokenizer.tokenize_with_offsets are invaluable. They return token IDs along with start and end byte offsets for each token in the original string.[4]

  Python
  ```
  # Example with BertTokenizer (assuming it's initialized)
  ```

```
# try:
#     ids, start_offsets, end_offsets = bert_tokenizer.tokenize_with_offsets(vietnamese_sentences)
#     print("\nToken IDs:\n", ids)
#     print("\nStart Offsets:\n", start_offsets)
#     print("\nEnd Offsets:\n", end_offsets)
# except NameError:
#     print("BertTokenizer not defined, skipping tokenize_with_offsets example.")
# except Exception as e:
#     print(f"Error during tokenize_with_offsets: {e}")
```

- **Integration with tf.data**: For efficient model training, integrate the tokenizer directly into a tf.data pipeline using the .map() transformation. This applies tokenization on the fly during data loading.[5]

```
Python
# Conceptual tf.data pipeline (assuming bert_tokenizer is initialized)
# try:
#     raw_dataset = tf.data.Dataset.from_tensor_slices(vietnamese_sentences)
#     # Wrap the tokenizer call in a function for mapping
#     def tokenize_sentence(sentence):
#         # Ensure sentence is a scalar tensor if needed by tokenizer map function
#         return bert_tokenizer.tokenize(tf.expand_dims(sentence, axis=0)) # Tokenize single
sentence, remove batch dim
#
#     tokenized_dataset = raw_dataset.map(tokenize_sentence)
#     # Further processing (e.g., padding, batching) would follow
#     print("\nConceptual Tokenized Dataset Element Spec:\n", tokenized_dataset.element_spec)
# except NameError:
#     print("BertTokenizer not defined, skipping tf.data example.")
# except Exception as e:
#     print(f"Error creating tf.data pipeline: {e}")
```

- **Handling Reserved Tokens**: The vocabulary includes reserved tokens like ,, [PAD], [User Code]. Downstream processing often involves adding and tokens to structure input sequences for models like BERT, padding sequences to a fixed length using `[PAD]`, or using for tasks like Masked Language Modeling. Operations like text.combine_segments() can assist in adding special tokens correctly.[5]
- **Vocabulary Size (vocab_size)**: The target vocab_size set during generation (e.g., 10000 in the script) is a hyperparameter influencing tokenization granularity and model size.[1] A larger vocabulary might represent more words directly (reducing subword splits) but increases the embedding layer's size in the model. An optimal size often depends on the corpus size and downstream task requirements.[12]

7. Conclusion

Successfully utilizing a custom-trained Vietnamese WordPiece vocabulary with TensorFlow Text involves careful selection and configuration of the tokenizer (BertTokenizer or WordpieceTokenizer). The paramount consideration is ensuring strict consistency between the parameters used during vocabulary generation (lower_case=False, normalization_form='NFC', suffix_indicator='##') and those used during tokenizer initialization. BertTokenizer offers convenience by handling pre-tokenization, but verification of its normalization behavior with lower_case=False for the specific TF Text version is recommended. WordpieceTokenizer provides explicit control but requires manual pre-processing.

The tokenize method converts text into subword IDs based on the vocabulary, while detokenize attempts to reconstruct text, though BertTokenizer's detokenization may not be perfectly reversible due to its pre-processing steps. The quality and coverage of the vocabulary, determined by the representativeness of the corpus used for its generation, directly impact the effectiveness of tokenization and the frequency of information-losing [UNK] tokens.

Users should meticulously test their chosen tokenization pipeline with diverse Vietnamese text samples relevant to their application, verifying the handling of case, diacritics, punctuation, and out-of-vocabulary words. Consulting the specific TensorFlow Text documentation for the version in use is advised to confirm parameter availability and behavior, particularly regarding normalization options within BertTokenizer.

Works cited

1. BERT - Hugging Face, accessed May 4, 2025, https://huggingface.co/docs/transformers/model_doc/bert
2. Subword tokenizers | Text - TensorFlow, accessed May 4, 2025, https://www.tensorflow.org/text/guide/subwords_tokenizer
3. Tokenizing with TF Text - TensorFlow, accessed May 4, 2025, https://www.tensorflow.org/text/guide/tokenizers
4. text.BertTokenizer | Text | TensorFlow, accessed May 4, 2025, https://www.tensorflow.org/text/api_docs/python/text/BertTokenizer
5. BERT Preprocessing with TF Text - TensorFlow, accessed May 4, 2025, https://www.tensorflow.org/text/guide/bert_preprocessing_guide
6. text/tensorflow_text/python/ops/bert_tokenizer.py at master · tensorflow/text - GitHub, accessed May 4, 2025, https://github.com/tensorflow/text/blob/master/tensorflow_text/python/ops/bert_tokenizer.py

7. text.WordpieceTokenizer - TensorFlow, accessed May 4, 2025, https://www.tensorflow.org/text/api_docs/python/text/WordpieceTokenizer
8. text.FastBertTokenizer - TensorFlow, accessed May 4, 2025, https://www.tensorflow.org/text/api_docs/python/text/FastBertTokenizer
9. Issue in detokenization of token_ids in BertTokenizer · Issue #949 · tensorflow/text - GitHub, accessed May 4, 2025, https://github.com/tensorflow/text/issues/949
10. text.Detokenizer - TensorFlow, accessed May 4, 2025, https://www.tensorflow.org/text/api_docs/python/text/Detokenizer
11. How to untokenize BERT tokens? - python - Stack Overflow, accessed May 4, 2025, https://stackoverflow.com/questions/66232938/how-to-untokenize-bert-tokens
12. GPT text generation from scratch with KerasHub, accessed May 4, 2025, https://keras.io/examples/generative/text_generation_gpt/