**Development of a Predictive Model for Verilog Next-Token Prediction**

### **Von Davis**

https://github.com/Von-R/VerilogProtoToken

### For research inquiries, please reach out to: Von.Roth.1991@gmail.com

### Project Overview:

Verilog is a Hardware Description Language (HDL) used for designing electronic systems like Field Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). FPGAs can be reprogrammed after deployment, while ASICs are tailored to specific tasks using HDL, like Verilog.

Programming in Verilog, much like other coding languages, is often tedious and time-consuming. However, modern "copilot" tools can streamline the process, saving time and effort, ensuring best coding practices, and reducing errors.

My model is designed to be a foundation for a future Verilog coding copilot. It’s trained for next-token prediction using cleaned and open-source Verilog code from GitHub.

Transformers are particularly effective at understanding long-range dependencies in sequences, which is essential for grasping the context and structure of Verilog code. They also handle variable-length sequences well, making them ideal for the dynamic nature of Verilog.

By fine-tuning pre-trained transformer models, my model hones in on the unique aspects of Verilog code. This approach leverages pre-existing models to enhance token prediction, saving resources and adapting quickly to new Verilog environments. Using transfer learning to fine-tune preprocessed Verilog code, my model aims to predict the next token in a sequence, setting the stage for a future Verilog code copilot.

### Scraping: BigQuery and Github

To scrape Verilog code with permissive licenses from GitHub, I used Google BigQuery, which provides an efficient way to search and filter large datasets stored in Google's infrastructure. By querying the GitHub public dataset on BigQuery, I targeted repositories containing Verilog code, filtering results based on file extensions (.v and .sv) and checking for permissive licenses (such as MIT, BSD, and Apache 2.0).

This allowed me to compile a substantial corpus of Verilog code while ensuring compliance with licensing requirements. The data shards were compiled into a single dataset, which was uploaded to Huggingface. All datasets - initial, intermediary, and final - were hosted on Huggingface Datasets and interacted with through the HF Datasets API.

### Dataset Preparation and Preprocessing

The original dataset before any preprocessing was 2.78 GB and 171,127 code samples, contained in a single split. The steps I took to prepare the data for model training are as follows:

**1. Remove Duplicates**

A massive proportion of the corpus of code samples scraped from github were duplicates, mostly belonging to separate forks and branches of the same repo. Due to the sheer bulk of duplicates in the original dataset, failing to remove the duplicate files would drastically increase the risk of overfitting and decrease the model’s ability to generalize to unseen data. Removing duplicate code samples resulted in a reduction of 44.66% of the original codebase, reducing the size from 2.78 GB to 1.54 GB, or 55.40% of the original dataset remaining.

**2. Removal of non-synthesizable / non-verilog code**

Synthesizable Verilog is the type of code that synthesis tools can turn from human-readable code into actual hardware designs. This kind of Verilog is used to describe hardware circuits that can be built on FPGAs or ASICs.

On the other hand, non-synthesizable Verilog includes code used for simulation and verification, like COQ code, testbenches, and simulation scripts. COQ is a formal proof management system, and non-synthesizable Verilog often contains constructs not meant for hardware synthesis but for verifying hardware designs through simulation.

To create a clean dataset of synthesizable Verilog code, it's crucial to remove non-synth code. While useful for testing and verification, non-synthesizable code doesn’t help the model predict the next token in real hardware descriptions.

To ensure the dataset only had synthesizable Verilog, I filtered out code samples with file paths containing keywords indicating non-synth purposes: “COQ, testbench, tb, simulation, sim,” and so on. This helped exclude files that were non-synthesizable and irrelevant to training the model.

After filtering, I applied standard preprocessing techniques like removing whitespace, comments, and other non-essential elements. This made sure the dataset was clean and focused on the core syntactic and semantic elements of Verilog code.

**Specific Preprocessing Steps:**

**Removing Comments:** I used regular expressions to strip out single-line comments (//), multi-line comments (/\* ... \*/), and comments enclosed in ( ... ). The function remove\_verilog\_comments was implemented to handle this task, making sure as many forms of comments were effectively removed from the code samples as possible.

**Identifying Non-Synthesizable Code:** I compiled a list of regex patterns to detect non-synthesizable Verilog constructs. The isNonSynthesizable function applied these patterns to identify and mark non-synthesizable code sections, including those masked within module definitions to prevent false positives.

**Handling COQ and Simulation Code:** I implemented a comprehensive set of patterns (COQ\_keywords, non\_synth\_path\_keywords) to detect and exclude COQ-related code and simulation scripts. This involved both URL-based and content-based filtering to thoroughly remove non-synthesizable code. The COQ\_module\_pattern was specifically used to identify COQ module definitions, while other regex patterns targeted testbenches and simulation-specific constructs.

**Processing Preprocessor Directives and Macros:** Directives and macros, particularly those starting with backticks, were selectively removed unless they were relevant to conditional compilations, which I determined contained meaningful semantic and contextual information.

**Failsafes and Error Handling:** Various checks are in place to handle edge cases and ensure the integrity of the preprocessing. This includes dumping lines with escaped COQ keywords and ensuring no valid parameters are removed from module definitions, by masking them during the code cleaning process.

To ensure the robustness of the preprocessing pipeline, several fail safes and error-handling mechanisms were incorporated:

* **Escaped COQ Keywords:** Despite filtering, some COQ keywords could potentially escape detection. The script included a failsafe that dumps lines with missed COQ keywords to a log for manual inspection. This step ensured that no COQ content remained in the final dataset.
* **Unbalanced Begin/End Blocks:** The function tracked the balance of begin and end blocks using the begin\_counter. If the counter indicated unbalanced blocks, an unbalanced\_counter was incremented. This information was logged, allowing for further inspection and artificial correction of structural imbalances in the Verilog code.
* **Empty Content Handling:** Files with empty content were flagged and removed early in the preprocessing pipeline. This check ensured that only files with substantive Verilog code were processed further.
* **Module Declaration Check:** Files lacking at least one module declaration were flagged and removed. This step ensured that only valid Verilog files with module definitions were included in the dataset.
* **Logging Invalid Paths:** Files with non-synthesizable keywords in their paths were flagged and logged. This step helped in identifying and excluding irrelevant files efficiently.

Due to these failsafes and error-handling mechanisms, the preprocessing pipeline maintained a high level of integrity and robustness. These measures ensured that the final dataset was clean, synthesizable, and suitable for training the next-token prediction model.

After the initial deduplication process, which reduced the dataset size from 2.78GB to 1.54GB, further preprocessing steps were applied to clean and refine the dataset. These steps, including the removal of non-synthesizable code, comments, and invalid constructs as described, further reduced the dataset size to 1.21GB. This means that 78.57% of the deduplicated dataset remained after preprocessing, resulting in 43.553% of the original dataset size being retained. These reductions highlight the effectiveness of the preprocessing techniques in distilling a large corpus of Verilog code down to its most relevant and synthesizable components, resulting in a high-quality dataset for training the next-token prediction model.

**3. Dropping Lines Based on Thresholds:**

After the initial deduplication and preprocessing steps, it was observed that many files contained very few lines of useful Verilog code. These files were often remnants of files that had been mostly non-synthesizable code and had been drastically reduced in content through preprocessing, and whose remaining code is both structurally and contextually mostly useless. To ensure the dataset maintained high-quality training data, a decision was made to remove files with fewer than a certain number of lines of code.

Various thresholds were tested to determine the optimal number of lines to retain. Files with fewer than 20 lines of code resulted in a 27% reduction in a test dataset of 500 samples, suggesting an extrapolated reduction to 0.76GB of the preprocessed dataset. Conversely, using a 50-line threshold led to a 52% reduction in the same test dataset, which would extrapolate 0.58GB remaining.

Ultimately, a threshold of 30 lines of code was selected as the optimal balance. Applying this threshold to the test dataset resulted in a 38.4% reduction, decreasing from 500 samples to 308. Extrapolated to the full dataset, this threshold led to a 38.0% reduction, reducing the dataset from 1.21GB to 0.75GB remaining of the cleaned dataset. This approach ensured that files with insufficient training information were excluded, improving the overall quality and relevance of the dataset for training the next-token prediction model.

**Identifier Anonymization**

The preprocessed dataset contained a vast number of unique tokens (11,571,698) - most being identifiers - significantly inflating the vocabulary size beyond what tokenizers like Byte Pair Encoding (BPE) or WordPiece, and transformer models such as GPT-2 and Mistral, could handle. To mitigate the effect of this massive vocabulary on the tokenizer and model and thus improve the model's performance, I anonymized the identifiers in each code example.

**Process:**

**1. Counting Identifiers:** Using regular expressions, I counted the unique identifiers within each code example.

**2. Creating Replacement Sets:** For each code example, I generated a set of generic variables and module names using the count I got. Variables were labeled as VAR1, VAR2, etc., up to the number of unique variables present in the code. Similarly, module and submodule names were labeled as MODULE1, SUBMODULE1, etc.

**3. Randomized Replacement:** To avoid introducing spurious patterns, identifiers were replaced randomly. This meant that instead of sequentially replacing identifiers in a fixed order (e.g., the first variable always becoming VAR1, the second VAR2), replacements were drawn randomly from the generated set. This randomization helped maintain the integrity and variability of the data.

The necessity for anonymizing identifiers stemmed from several key factors:

**- Vocabulary Size Reduction:** By anonymizing identifiers, I significantly reduced the vocabulary size, making it manageable for tokenizers and transformer models. I ultimately reduced the number of distinct tokens from 11,571,698 to 84005 anonymization, then to 42163 by manually trimming the tokenizer, or 0.36% of the initial set of distinct tokens. This reduction is crucial for efficient training and inference.

**- Mitigating Overfitting:** Unique identifiers can cause models to overfit to specific instances in the training data, limiting their ability to generalize to new, unseen data. Anonymization helps mitigate this risk by removing the dependence on specific variables and module names.

**- Preserving Data Integrity:** Randomizing the replacement of identifiers ensured that no artificial patterns were introduced into the dataset. Sequential replacement could have created predictable patterns, leading the model to learn these patterns rather than the underlying syntax and structure of Verilog code.

**- Consistency Across Examples:** Using standardized placeholder names (VAR, MODULE) ensured consistency across different code examples, further aiding the model in learning the general structure and syntax of Verilog code without being distracted by irrelevant variations in identifier names.

**Additional Benefits:**

The anonymization of identifiers in the Verilog dataset brings several additional benefits beyond reducing vocabulary size and mitigating overfitting:

**1. Improved Generalization:**

- By replacing specific identifiers with generic placeholders, the model focuses on learning the structural and syntactic patterns of Verilog code rather than memorizing specific instances. This improves the model's ability to generalize from the training data to new, unseen code examples.

**2. Enhanced Model Efficiency:**

- A smaller, more manageable vocabulary size can lead to faster training times and reduced computational expense. The model can process smaller vocabulary sets more efficiently, potentially leading to quicker convergence and lower costs.

**3. Simplified Tokenization:**

- Tokenizers, such as BPE or WordPiece, perform better with a smaller and more consistent set of tokens. Anonymizing identifiers standardizes the token set, allowing the tokenizer to operate more effectively and produce more meaningful subword units.

**4. Reduced Noise:**

- Unique and varied identifiers can introduce noise into the dataset, making it harder for the model to discern useful patterns. Anonymization reduces this noise, providing cleaner data from which the model can learn the patterns more easily.

**5. Consistency Across Data:**

- Standardizing identifiers helps maintain consistency across different code samples. This uniformity can lead to better learning as the model encounters fewer outliers and odd variations.

**6. Simplified Debugging and Analysis:**

- With standardized identifiers, debugging the model and analyzing its predictions become easier. Identifiers like VAR1, VAR2, etc., make it straightforward to trace variables and understand their roles in the model's output.

**7. Enhanced Interpretability:**

- Anonymized identifiers improve the interpretability of the model's predictions. By focusing on generic placeholders, it is easier to analyze and explain the model's behavior and decision-making processes.

**Evaluation Technique: Masking Variables and Module Names**

Continuing from anonymization, part of the model evaluation technique involves masking anonymized variables and module names and not factoring their prediction accuracy into the final evaluation. This is then compared to the performance of the baseline, unmasked evaluation. The masking approach has several merits:

**1. Focus on Syntax and Structure:**

By masking variables and module names, the evaluation can focus on the model's understanding of the syntax and structure of Verilog code, rather than its ability to predict specific identifiers. This ensures that the model is being evaluated on its core task—understanding and predicting the flow of Verilog code—without being distracted by the inconsistent nature of variable and module names.

**2. Enhancing Generalization:**

Masking encourages the model to generalize its learning to new and unseen code samples. By not including variable and module name predictions in the evaluation, I attempt to ensure the model’s performance metrics reflect its ability to generalize beyond the training data, leading to a more robust model.

**3. Consistency with Anonymization:**

Specific to the baseline evaluation approach, given that the training data involves anonymized identifiers, evaluating the model on the prediction of these anonymized forms aligns the training and evaluation phases. This consistency ensures that the model's evaluation is fair and accurately reflects its performance on the task it was trained for.

**4. Simplified Evaluation Metrics:**

Predicting variable and module names can add noise to the evaluation metrics, as these names can be arbitrary and not indicative of the model's true performance. By excluding these predictions, the evaluation metrics become clearer and more meaningful, focusing on the model’s ability to predict relevant tokens that dictate the code structure.

**5. Reduced Vocabulary Complexity:**

Variables and module names add complexity to the vocabulary, and their presence in evaluation could unduly affect performance metrics. Masking these names reduces vocabulary complexity, allowing for a more streamlined and accurate assessment of the model's capabilities.

By implementing identifier anonymization, the intent is to optimize the training process but also unlock several strategic advantages that enhance the overall effectiveness and applicability of the Verilog next-token prediction model. This approach ensures the model is evaluated fairly and accurately, reflecting its true capabilities in understanding and predicting Verilog code structure and syntax.

**Preserving Port Identifiers**

In the process of anonymizing identifiers within the Verilog dataset, a deliberate decision was made to preserve port identifiers such as addr, lut, and reg from among a relatively small, semantically rich set of 62 keywords that follow meaningful and consistent patterns in Verilog code. The rationale behind this decision is:

**Semantic Importance:** Port identifiers carry significant meaning related to the design and functionality of hardware components. Terms like addr (address), lut (lookup table), and reg (register) are standard in Verilog and indicate specific roles or types of connections within the code. Preserving these identifiers helps maintain the semantic integrity of the code, allowing the model to learn and understand the specific functions and interactions these ports represent.

**Pattern Consistency:** Unlike general variable names, port identifiers follow consistent and predictable patterns that are valuable for understanding the structure and behavior of the Verilog code. By retaining these identifiers, the model can better capture the logical flow and connectivity of the hardware design, which is essential for accurate next-token prediction.

**Preservation of Context:** Keeping port identifiers intact ensures that the model retains crucial contextual information about the hardware design. This context is important for training and inference, as it enables the model to make more informed predictions based on the functional roles of these ports.

While preserving port identifiers enhances the model's understanding of Verilog code, there is room for further refinement. One potential area of improvement is to anonymize other identifiers and module names that contain meaningful subwords in a similar manner. For instance, identifiers that include terms like addr, reg, or lut as subwords could be anonymized to ADDR1, REG2, LUT3, etc. This approach would preserve the semantic value of these identifiers while still reducing vocabulary size and preventing overfitting.

By implementing this anonymization strategy, the model continues to learn from meaningful patterns in the data while maintaining a manageable vocabulary size. This balance between anonymization and preservation of key terms will contribute to the overall effectiveness and robustness of the Verilog next-token prediction model.

The size of the dataset before anonymization was 0.579GB. After implementing the identifier anonymization process, the dataset size was reduced to 0.165GB. This represents a reduction of 71.50% of the size of the remaining dataset, and *a total reduction of 94.10% of the size of the original dataset*. This significant decrease demonstrates the substantial impact of lengthy and varied identifiers on the dataset's size, highlighting the effectiveness of anonymization in streamlining the dataset and making it more manageable for training purposes.

During the preprocessing of the Verilog dataset, a significant issue was identified: files containing extremely long lists of identifiers, often representing netlists or very long module definitions with no other structure or logical code. These files were characterized by having a high proportion of variable names (VAR) relative to the rest of the code, with some files containing up to 40,000+ identifiers. This inflated the vocabulary size for the tokenizer and added semantically useless data, which would negatively impact the model's performance.

To address this issue, I wrote a script to calculate the ratio of variable names to the total content in each file. The content was scanned to identify all variable names (VAR). The total length of these variable names was then divided by the total length of the content to compute the ratio.

After experimenting with different thresholds, I determined that files with a variable-to-content ratio greater than 50% should be removed. This threshold effectively filtered out files that were predominantly composed of variable names, leaving those with a more balanced and meaningful code structure. This resulted in the removal of 737 files, significantly cleaning the dataset.

By implementing this method, the dataset was further refined to focus on high-quality, meaningful Verilog code, reducing unnecessary complexity and improving the overall training process for tokenizers and the next-token prediction model.

**Tokenizer Training Process**

For the Verilog next-token prediction model, training effective tokenizers is a crucial step. The main candidates for the model are GPT-2 and Mistral, among others. Here's a high-level overview of my training process for the tokenizers:

The dataset was loaded and combined across different splits (train, test, validation) to create a single list of all text contents. This provided a comprehensive corpus for training the tokenizers.

A generic function train\_tokenizer was created to handle the training of tokenizers for different models. This function takes the model name, texts, and the desired output file name as inputs and trains a tokenizer accordingly. The process varies slightly depending on the model:

BERT: Utilizes the WordPiece algorithm with special tokens like [CLS], [SEP], [PAD], [MASK], and [UNK]. The vocabulary size is set to 30,522.

GPT-2: Employs the BPE algorithm with special tokens such as [UNK], [PAD], [MASK], [BOS], and [EOS]. The vocabulary size is set to 50,257, and specific subword strategies are applied.

Mistral: Also uses the BPE algorithm with a similar setup to GPT-2, including the same special tokens and vocabulary size.

Gemma: Uses BPE with a slightly different configuration, tailored to the requirements of the Gemma model.

LLaMA: Configured with BPE and special tokens [BOS], [EOS], [UNK], and [PAD], also with a vocabulary size of 50,257.

The tokenizer function begins by initializing the tokenizer model along with its pre-tokenizer component. The script specified different parameters and special tokens for each tokenizer to be trained for each model. After initialization, the tokenizer script uses a text iterator to step through the data from which each model tokenizer is trained. Once the training phase is complete, the function saves the trained tokenizer to a file, making it available for future use.

To verify the tokenizers were trained correctly, sample Verilog code strings were tokenized using each model's tokenizer, and then decoded back to their original form.

Lastly, the total vocabulary of the GPT-2 tokenizer was manually trimmed to 42,163 tokens, or a reduction of 99.65% of the 11,571,698 distinct tokens in the raw dataset.

**Evaluating the Tokenizers**

To evaluate the effectiveness of the tokenizers trained for different models evaluation aimed at measuring key metrics such as vocabulary coverage, mean and median token counts, out-of-vocabulary (OOV) rate, and subword fragmentation was used. The primary focus was on the Mistral and GPT-2 tokenizers.

The evaluation script was designed to assess the performance of the tokenizer by performing several key calculations:

**Word and Token Counting:** The total number of words and tokens in each code sample were counted. Words were split based on whitespace, and tokenization was performed using the tokenizer.

**Out-of-Vocabulary (OOV) Detection:** The number of words not covered by the tokenizer's vocabulary (resulting in [UNK] tokens) was tracked.

**Subword Splits:** The number of words split into multiple subwords during tokenization was recorded.

Vocabulary Coverage: The proportion of unique words covered by the tokenizer's vocabulary was calculated.

**Token Count Distribution:** The mean and median token counts for the code samples were computed, and the distribution of token lengths was analyzed.

**Subword Fragmentation:** The ratio of subword splits to total words was calculated to understand how frequently words were broken into multiple tokens.

**Results:**

**Mistral Tokenizer:**

Vocabulary Coverage: 1.0 (100% coverage)

Mean Token Count: 736.72

Median Token Count: 244.0

OOV Rate: 0.0 (no out-of-vocabulary words)

Subword Fragmentation: 0.4514 (45.14% of words were split into subwords)

**GPT-2 Tokenizer:**

Vocabulary Coverage: 1.0 (100% coverage)

Mean Token Count: 737.28

Median Token Count: 244.0

OOV Rate: 0.0 (no out-of-vocabulary words)

Subword Fragmentation: 0.4514 (45.14% of words were split into subwords)

Analysis:

Both the Mistral and GPT-2 tokenizer evaluations yielded impressive results, demonstrating their effectiveness in handling Verilog code.:

Vocabulary Coverage: Both tokenizers achieved 100% vocabulary coverage, indicating that all unique words in the dataset were represented in their vocabularies.

Mean and Median Token Counts: The mean and median token counts were similar for both tokenizers, reflecting a consistent tokenization process.

OOV Rate: Both tokenizers had an OOV rate of 0.0, meaning they could tokenize every word in the dataset without resorting to [UNK] tokens. This is a significant achievement, showcasing their comprehensive vocabulary.

Subword Fragmentation: The subword fragmentation rate for both tokenizers was around 45.14%. While a substantial portion of words were split into subwords, this is expected in technical and domain-specific datasets like Verilog code.

Overall, the performance of both the Mistral and GPT-2 tokenizers is impressive. Their ability to cover the entire vocabulary and effectively tokenize Verilog code ensures that the model will have high-quality inputs for training and inference.

**High-Level Overview of the Model Training Script**

Training a transformer model in Python involves several steps. First, the environment is set up by installing necessary libraries such as PyTorch or TensorFlow, along with libraries for data handling. Following this, the dataset is gathered and preprocessed, which includes tokenizing the text data into a format the model can process.

With the data prepared, the next step is to define the transformer model. This typically involves specifying the number of layers, attention heads, and other relevant parameters. After the model is defined, it is initialized and the loss function and optimizer are set up. These components are important as they guide the model during the training process.

The core of the training involves a loop where batches of tokenized data are fed through the model. During this phase, the loss is calculated and the model's weights are updated to minimize this loss. This process is repeated over many epochs until the model's performance reaches a satisfactory level. My model training script implemented early stopping to save computation time and avoid overfitting.

Once training is complete, the model is evaluated on a validation dataset to assess its performance and identify any necessary adjustments. Upon achieving satisfactory results, the trained model is saved for future use, enabling it to be used for predictions or further fine-tuning as needed.

In order to attempt to determine the best set of hyperparameters for ideal model performance, the script implemented a grid search technique in order to test different hyperparameter combinations.

By following this process, the script efficiently trains a GPT-2 model for Verilog code, leveraging advanced ML techniques and distributed computing capabilities to ensure robust and scalable model training.

**Overview of Model Testing**

**Task Overview**

The primary task my project is focusing on is the next token prediction using a language model. This task involves predicting the next token in a sequence based on the preceding tokens. This is a fundamental problem in natural language processing (NLP) with applications in text generation, auto-completion, and more.

The testing framework I developed is designed to evaluate the performance of language models across multiple metrics.

**Metrics Explored**

**Next Token Prediction Loss:**

* Measures how well the model predicts the next token in a sequence.
* Lower values indicate better performance.

**Perplexity:**

* Exponential of the average negative log-likelihood per token.
* Lower values indicate that the model is more confident in its predictions.

**Accuracy:**

* Measures the proportion of correct predictions out of all predictions.
* Higher values indicate better performance.

**Precision, Recall, F1 Score:**

* Precision: Proportion of true positive predictions out of all positive predictions.
* Recall: Proportion of true positive predictions out of all actual positives.
* F1 Score: Harmonic mean of precision and recall.

**Top-5 Accuracy:**

* Measures the proportion of times the correct token is within the top 5 predictions.
* Higher values indicate better performance.

**Entropy:**

* Measures the randomness in the model's predictions.
* Lower entropy indicates more confident and decisive predictions.

**Prediction Confidence:**

* Measures the model’s confidence in its predictions.
* Higher values indicate greater confidence in the predictions.

**Masking Identifiers**

My evaluation script includes a mechanism to mask identifier tokens in the test dataset: variables, modules and submodules. These tokens match specific regex patterns (e.g., VAR1, MODULE1, SUBMODULE1) and are placeholders with no semantic meaning but carry contextual placement information.

**Pros of Masking Identifiers:**

- Improved Model Generalization: By ignoring these tokens, the model's performance metrics reflect its ability to predict more meaningful tokens.

- Focused Evaluation: Provides a clearer picture of how well the model understands and predicts the substantive content of the text.

**Cons of Masking Identifiers:**

- Contextual Loss: The placement and context of these tokens are still meaningful in the sequence, and ignoring them might lead to loss of contextual information.

- Potential Overlooking of Anomalies: Masking could lead to ignoring potential weaknesses in the model's ability to handle specific types of tokens.

**Evaluation Results**

The testing framework I have developed not only evaluates the model's performance on the next token prediction but also provides deeper insights through additional metrics. I aim to refine my understanding of the model's capabilities and limitations by exploring metrics beyond loss.

**Preliminary Results: Masked vs. Unmasked Tokens**

In my initial evaluations, I compared the performance of the model on unmasked and masked datasets. Here are the results I obtained:

Unmasked Dataset

{

"Next Token Prediction Loss": 0.8410023064430141,

"Perplexity": 2.3186898502473587,

"Accuracy": 0.4576058528680817

}

Masked Dataset

{

"Next Token Prediction Loss": 0.9509887154996395,

"Perplexity": 2.5882673263549805,

"Accuracy": 0.5194997294130735

}

**Analysis of Preliminary Results**

**1. Next Token Prediction Loss:**

- Unmasked: 0.8410

- Masked: 0.9510

- Interpretation: The loss increased when masked tokens were excluded from the evaluation. This suggests that the model finds it easier to predict the placeholders, likely because they have less variability and follow more predictable patterns.

**2. Perplexity:**

- Unmasked: 2.3187

- Masked: 2.5883

- Interpretation: Perplexity also increased in the masked dataset. Perplexity measures the model's confidence in its predictions. Higher perplexity indicates that the model is less certain about its predictions when masked tokens are excluded, which aligns with the increased loss.

**3. Accuracy:**

- Unmasked: 0.4576 (45.76%)

- Masked: 0.5195 (51.95%)

- Interpretation: Interestingly, accuracy improved when masked tokens were excluded. This might indicate that the model is more accurate in predicting the remaining meaningful tokens, which are likely to be more consistent and have clearer patterns.

The preliminary results reveal a nuanced picture of the model's performance:

- Increased Loss and Perplexity in Masked Evaluation: These metrics suggest that masked tokens (placeholders) are easier for the model to predict, likely due to their predictable nature. When these tokens are removed overall unpredictability is increased, and the model faces a more challenging prediction task, leading to higher loss and perplexity.

- Improved Accuracy in Masked Evaluation: The improvement in accuracy when masked tokens are excluded indicates that the model performs better on the remaining tokens. This suggests that the placeholders might possibly be introducing noise or bias in the accuracy measurement.

**Speculative Pros and Cons of Masked Evaluation**

**Pros:**

By excluding semantically meaningless tokens, the evaluation becomes more focused on the model's ability to understand and predict meaningful content. The discrepancy between different metrics under masked and unmasked conditions provides deeper insight into the model's strengths and weaknesses.

**Cons:**

Even though the masked tokens are placeholders, their placement, and contextual information could still be important for the model's overall understanding of the sequence. Masking could introduce a bias by excluding certain types of tokens, possibly leading to an overestimation of the model's performance on meaningful tokens.

In summary, the preliminary results highlight the trade-offs involved in masked evaluation. While it provides a clearer picture of the model's performance on meaningful tokens, it also presents challenges related to the loss of contextual information. Moving forward, it will be important to balance these aspects to develop a robust evaluation strategy, as well as to explore the effects of masking in relation to performance as a meta-evaluation of the masking strategy.

### **Comparison of LSTM and GPT-2 Models**

Based on the previous results I chose to exclude the masked token from the GPT and LSTM model evaluation scripts. I trained each model on 1, 10 and 50 training epochs over the dataset.

#### **Performance Metrics**

**LSTM Model:**

* **1 Epoch:**
  + Next Token Prediction Loss: 8.31
  + Perplexity: 4056.09
  + Accuracy: 0.0
  + Precision, Recall, F1 Score: 0.0
* **10 Epochs:**
  + Next Token Prediction Loss: 6.13
  + Perplexity: 461.57
  + Accuracy: 0.45
  + Precision: 0.016
  + Recall: 0.021
  + F1 Score: 0.018
* **50 Epochs:**
  + Next Token Prediction Loss: 6.29
  + Perplexity: 539.42
  + Accuracy: 0.44
  + Precision: 0.016
  + Recall: 0.020
  + F1 Score: 0.017

**GPT-2 Model:**

* **1 Epoch:**
  + Next Token Prediction Loss: 0.82
  + Perplexity: 2.27
  + Accuracy: 0.52
  + Precision: 0.023
  + Recall: 0.024
  + F1 Score: 0.023
* **10 Epochs:**
  + Next Token Prediction Loss: 0.78
  + Perplexity: 2.17
  + Accuracy: 0.33
  + Precision: 0.024
  + Recall: 0.016
  + F1 Score: 0.018
* **50 Epochs:**
  + Next Token Prediction Loss: 1.18
  + Perplexity: 3.26
  + Accuracy: 0.18
  + Precision: 0.024
  + Recall: 0.009
  + F1 Score: 0.011

### **Analysis:**

1. **Learning Speed:**
   * **GPT-2:** The GPT-2 model shows a significantly lower loss and perplexity after just one epoch compared to the LSTM model. This indicates that GPT-2 learns much more quickly. However, its performance deteriorates with more epochs, suggesting overfitting.
   * **LSTM:** The LSTM model shows improvement over multiple epochs. It does not exhibit the same rapid overfitting as GPT-2 but learns more slowly overall.
2. **Accuracy and F1 Scores:**
   * **GPT-2:** The initial accuracy and F1 scores for GPT-2 are higher after the first epoch compared to LSTM. This supports the conclusion that GPT-2 captures the underlying patterns in the data more quickly.
   * **LSTM:** The LSTM model shows better accuracy at 10 epochs compared to GPT-2 at the same number of epochs, but this performance does not surpass GPT-2's one-epoch performance.
3. **Perplexity:**
   * **GPT-2:** The low perplexity in the early epochs for GPT-2 suggests it is highly confident in its predictions early on.
   * **LSTM:** Higher perplexity in LSTM indicates less confidence in predictions initially, but it decreases significantly with more training epochs.
4. **Overfitting:**
   * **GPT-2:** Rapid overfitting is evident in the GPT-2 model as the performance metrics worsen with additional epochs.
   * **LSTM:** The LSTM model doesn't overfit as quickly, suggesting it might be more robust for longer training periods or smaller datasets.

### **Conclusion:**

1. **GPT-2 Model:**
   * **Strengths:** Quick learning and high initial performance metrics. Suitable for tasks requiring fast model adaptation and fewer epochs.
   * **Weaknesses:** Prone to overfitting quickly, requiring regularization or other techniques to mitigate overfitting effects.
2. **LSTM Model:**
   * **Strengths:** More stable performance over multiple epochs, less prone to rapid overfitting.
   * **Weaknesses:** Slower learning, requiring more epochs to reach optimal performance.

**Future Improvements**

**Identifier and Tokenization Enhancements**

- Deep Metric Learning and K-Means Clustering: Implement deep metric learning combined with k-means clustering for more meaningful identifier separation. Attempt to preserve more highly semantically rich identifiers.

- Refine Tokenization Strategy: Enhance tokenization to better handle underscores, common prefixes, and suffixes. Optimize subword fragmentation and average token length.

**Regularization and Generalization**

- Introduce Noise for Regularization: Introduce noise during training to improve model generalizability and prevent overfitting.

- Implement Regularization Techniques: Explore and apply various regularization methods to further enhance model generalization capabilities.

**Model Evaluation and Validation**

- Evaluate Performance Through Granularity and Manual Inspection: Assess model performance at a granular level and through manual inspection to ensure accuracy and reliability.

- K-Fold Validation: Use k-fold validation to thoroughly evaluate and compare model performances, ensuring consistency and robustness across different data splits.

**Dataset and Preprocessing**

- Train on Various Stages of Preprocessing: Conduct training on datasets at different preprocessing stages to identify the most effective preprocessing techniques.

- Refine Preprocessing and Tokenization: Continue improving preprocessing steps to maximize dataset quality and relevance.

- Create AST Trees for Identifier Replacement: Develop Abstract Syntax Trees (AST) to aid in variable name replacement and other preprocessing tasks.

- Revisit High Attrition Steps: Reevaluate and adjust preprocessing steps that result in high data attrition to preserve more of the dataset.

**Model Optimization Techniques**

- Model Pruning: Reduce model size by eliminating unnecessary weights, enhancing inference speed while maintaining accuracy.

- Quantization: Convert model weights from floating-point to lower precision formats, such as INT8, to reduce memory usage and boost inference speed.

- Distillation: Train smaller models using the outputs of larger, high-performing models to retain accuracy while significantly reducing computational demands.

- Hyerparameter tuning: Implement a grid search technique to identify the optimal set of hyperparameters for performance.

By implementing these improvements, I aim to enhance the Verilog next-token prediction model's efficiency, accuracy, and robustness.

**Conclusion: Paving the Way for a Verilog Code Copilot**

This project is a step toward creating a Verilog code copilot, a tool designed to make writing Verilog code easier and more efficient. By developing a complete process from data collection and preprocessing to model training and evaluation, a framework for next-token prediction in Verilog code has been set up. This lays the groundwork for future improvements that could change how developers work with Verilog.

The model predicts the next token in a sequence of Verilog code, which helps reduce the time and effort needed for coding. This feature can be integrated into development environments to provide real-time suggestions, assisting developers in writing code faster and more accurately. This allows developers to focus on higher-level design and logic, knowing that the copilot will handle the routine syntax and structure.

In summary, this project demonstrates the potential of a Verilog next-token prediction model and outlines a path toward developing a Verilog code copilot. By using advanced machine learning techniques and refining the approach, this project aims to improve Verilog coding, making it more efficient, accurate, and accessible.