

Influence of Tokenization Strategies on the Prediction of CAD Model Descriptions

Sayeda Hadisa Habibi¹, Julia Bergelt¹, Michael Teichmann¹, Fred H. Hamker^{1*}

¹Chemnitz University of Technology, 09107 Chemnitz

*Corresponding author. E-Mail address: fred.hamker@informatik.tu-chemnitz.de; Tel.: 0371/53137875

Abstract

Computer-aided design (CAD) plays a critical role in the development of innovative 3D models and tackles many different industrial branches ranging from small enterprises to large corporations. In recent years, growing interest has emerged in automating CAD workflows, using deep learning, to reduce repetitive manual tasks and allow engineers to focus on higher-level design and innovation. However, despite major advances in deep learning, especially in Natural Language Processing (NLP), applying these methods to CAD data remains challenging. One major challenge is an efficient representation of CAD data. Traditional methods such as voxel-based and point-cloud representations suffer from limitations, such as high memory consumption and loss of geometric detail. On the other side, standard tokenization techniques for NLP such as WordPiece, originally developed for text, may not handle the discrete and structured nature of CAD data. We focus on advanced text-based representations for CAD, building upon the DeepCAD framework, which is well suited for transformer-based architectures. We compare three representation strategies: a common method from NLP, a method inspired by DeepCAD, and a hybrid approach that combines elements of both. Our study aims at evaluating their effectiveness and at identifying pathways toward more robust and scalable CAD representations for design automation.

Keywords:

Sequential CAD Representation, Tokenization, DeepCAD Tokenization, WordPiece Tokenization, Representation Efficiency

1 Introduction

Mechanical design must incorporate factors such as cost, maintenance and manufacturability. At its core, it involves shaping parts and assemblies, which directly influence product performance and compliance. While computer-aided design (CAD) software is widely used to create 3D models, it primarily digitizes the engineer's ideas, leaving the integration of requirements to human expertise. Since many designs involve repeated geometrical elements, automating CAD modeling tasks can streamline product creation, reduce redundancy, improve quality, and allow engineers to focus on innovation [1].

A promising approach is using generative AI trained on existing CAD models, which already contain best practice knowledge, for predicting the next necessary design steps to support an engineer during his design process. In this work, we focus on text-based CAD representations which allow to apply highly successful methods from natural language processing (NLP). The challenge is that conventional tokenization techniques for creating the required input sequences, such as WordPiece [2], may not optimally handle the discrete and structured nature of CAD data. Thus, we investigate the efficiency of CAD-specific tokenization methods and compare the widely used WordPiece tokenizer with the sketch-extrude-based technique introduced by DeepCAD [3]. As dataset we use the DeepCAD dataset [3],

which includes the design history and is derived from the large ABC dataset [4]. The design history is important because it provides sequential information about how a model was constructed, allowing AI models to learn patterns in the step-by-step design process rather than just static final geometries. We seek for an information efficient representation and base our comparison of the different tokenization methods on the metrics token fertility and prediction performance on masked sequence parts, using a BERT-style transformer model [5], to measure the suitability for generative AI.

2 State of the Art

Computer-aided design (CAD) models may be represented either as explicit geometric data, such as polygon meshes or surface and solid descriptions, or as text-based, parametric definitions that encode human-readable primitives and their associated parameters. Geometry-based representations are detailed descriptions of a 3D object that capture the physical form of an object in a spatial context [6]. Although geometry-based representations are used for different deep learning approaches, such as recognition, due to their complex structure, they are not suitable for our intended application of making design step suggestions. Text-based representations, however, have richer information related to metadata, part specification, and relationships between components. For example, FeatureScript, a domain-specific language developed by Onshape¹, enables the representation of CAD models as a sequence of parametric operations. This provides fine-grained control, interpretability, and a clear understanding of the design inputs. Moreover, the design steps performed by the designer are captured in the corresponding order. A collection of over 1 million designs from Onshape in the FeatureScript format is stored in the ABC dataset [4]. The DeepCAD dataset is a subset of the ABC dataset and also retains the full design history for each CAD model based on data from Onshape's public repository [7]. However, it limits the models to designs, that only includes *Sketches* and *Extrude* operations. Another dataset, SketchGraphs [8], consists of 15 million real-world parametric sketches, also from Onshape. As the name suggests, the dataset contains solely sketches, which are represented as geometric constraint graphs. All three datasets, ABC, DeepCAD, as well as SketchGraphs, are publicly available and free for academic usage.

To utilize deep learning methods, such as transformer architectures, the textual input that should be processed must undergo a tokenization. One common tokenization approach for textual data is WordPiece [2]. It first breaks the input text based on whitespaces. Then, each word is further broken down into so-called tokens. It uses a greedy longest-match-first strategy, where the algorithm iteratively selects the longest possible sub-word from a predefined vocabulary that matches the beginning of the remaining word. If no full match is found, the word is decomposed into smaller sub-word units. Sub-words that appear as continuations of previous tokens are typically prefixed with “##”, indicating that they are not standalone tokens but part of a larger word. Recent advanced text-based tokenization for 3D data can be divided into two categories: sketch-based and 3D-based. For instance, the sketch-based model Vitruvion [9] is a generative model designed for synthesizing structured 2D sketches. It decomposes sketch generation into two stages: First, it autoregressively generates geometric primitives (e.g., lines, arcs, and circles), and then it predicts geometric constraints (e.g., tangency, coincidence) between these primitives. In the tokenization process, each geometric shape is parametrized according to its geometric properties and is encoded into a structured format. The model uses a transformer-based architecture and is trained on the SketchGraphs dataset. Another sketch-based generative model designed for producing structured 2D sketches is TurtleGen [10]. It produces sketches that already meet the geometrical rules and follow the topological conditions for creating a sketch design, which makes

¹ <https://www.onshape.com>

the workflow easier for generating 3D models. Sketches are represented using Turtle Graphics, where each sketch is described as sequences of drawing commands. A sketch begins with a *LoopStart* token, followed by line, arc, or circle commands defined by relative displacement vectors (e.g., Δx , Δy). This discrete, interpretable tokenization captures both geometry and topology, enabling efficient training of autoregressive models such as transformers without requiring constraint solvers or explicit coordinate systems [10]. Contrarily, DeepCAD [3] is a transformer-based 3D model generator that comprises the two operations *Sketches* and *Extrusion*. A sketch is composed of one or more closed curves (e.g., lines, arcs, and circles) defined on a 2D plane. These serve as the foundational elements for solid modeling techniques, such as extrusion, which transforms a 2D sketch into a 3D object. The tokenization process involves encoding the parameters of each command into a vector, where each element is an integer value in the range of 0 to 255. SkexGen is another 3D-based generative model for computer aided design that builds upon TurtleGen and DeepCAD by explicitly modeling faces (2D regions defined by closed loops) as the basis for valid 3D geometry generation via extrusion. It represents geometry, topology, and extrusion parameters using separate codebooks and employs specialized sketch and extrusion decoders. While this structured design enables interpretable and reusable CAD generation, SkexGen is slower than DeepCAD due to autoregressive decoding [11].

3 Methods

We present the dataset used in our experiments, followed by the proposed tokenization methods. We then present the model architectures corresponding to each tokenization approach and conclude with details of the training procedures.

3.1 Data Collection

We used the DeepCAD dataset as data source for our research. While several other CAD datasets exist, such as the ABC dataset [4], which contains a larger number of samples and greater geometric variation, DeepCAD offers a unique advantage due to its structured sequential format. It captures the design process as a sequence of parametric modelling operations, specifically sketch and extrude commands. This format reduces geometric complexity while preserving essential information about the design process [3]. This makes DeepCAD particularly compatible with transformer-based architectures, which require structured, tokenized input sequences. Given the scope of this work, which focuses on investigating representation efficiency by using, among other measures, a lightweight transformer model, the DeepCAD dataset is sufficient.

The DeepCAD dataset includes a total of 179,133 CAD designs, following the same structural format as the Fusion360 Gallery dataset [12]. The designs are represented through a sequential representation stored in JSON files, organized under three main top-level keys *entities*, *properties*, and *sequence*, describing the structure, metadata, and timeline of operations for a given CAD model. *Entities* store all individual modeling operations such as sketches, extrusions, and other parametric features. A sketch entity is used to define 2D profiles. It contains a unique name, a transformation matrix, the reference plane as well as identifiable profiles, which define closed shapes through *Lines*, *Arcs*, and *Circles*. An extrusion entity is defined by several attributes, including the associated profiles (defined in *Sketches*), and the extent type (e.g. one-sided extrusion, equal extrusion on both sides of the sketch plane, or asymmetric extrusion with different lengths on each side). Finally, the *sequence* field defines the ordering of modeling operations within a design. While the design order may not be apparent from the file structure, the sequence list provides this information explicitly. Each entry contains an index, a type (e.g., *Sketch* or *Extrude*), and a reference to the corresponding entity ID. This allows for the reconstruction of the modeling timeline, identifying which operation occurs first and which follows in the design process.

3.2 Tokenization methods

We investigate three different tokenization methods: WordPiece-naïve, WordPiece-DC, and DeepCAD. All tokenization pipelines (Figure 1) begin with the raw JSON files from the DeepCAD dataset. From this common starting point, each method applies its own sequence of processing steps to transform CAD construction information into a text-based representation, suitable for a BERT-style transformer. These steps include extracting or constructing the CAD sequence, normalizing numerical values, and converting sketches and extrudes into structured text rows when applicable. The differences between the pipelines reflect the distinct philosophies behind each representation: the WordPiece-naïve method focuses on flattening entities and sequence, the WordPiece-DC approach relies on structured sketch-extrude descriptions while preserving command information as text, and the DeepCAD method incorporates quantization and vectorization.

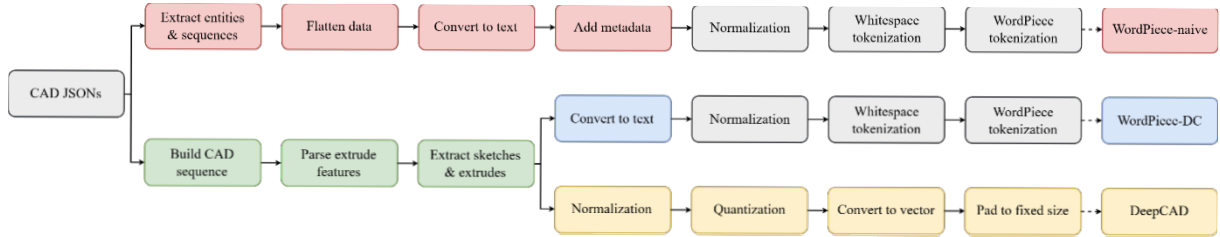


Figure 1: Pipeline of preprocessing for the three tokenization methods WordPiece-naïve (red), WordPiece-DC (blue), and DeepCAD (yellow). Common steps are depicted in gray.

We first determined suitable vocabulary sizes for the two WordPiece methods. Our analysis is based on token fertility within our CAD dataset and aims at identifying an optimal vocabulary size for training each tokenizer. Fertility is defined as the ratio between the number of output tokens (sub-words generated by the tokenizer) and the number of input words [13]. In general, lower fertility values indicate better efficiency, since information is distributed among less tokens. Our results are as follows:

3.2.1 WordPiece-naïve

Before training the WordPiece-naïve tokenizer, we converted the DeepCAD dataset into a flat, linear format suitable for token-based modeling. There, each CAD design is represented as a linear sequence of commands, analogous to a sentence in natural language. The flattening process iterates through each JSON file and recursively flattens each nested dictionary and list into key-value pairs, using dot notation and index markers to preserve hierarchy. Then we put every word in a list of sequence, e.g., ID, entity, and numerical values (integers and decimals). The tokenized sequence for an example is given in Figure 3a, step 2.

We determined the vocabulary for our tokenizer by constructing a corpus from the entire (flattened) dataset. We first applied whitespace tokenization to remove unnecessary spacing, then determined and built the vocabulary of the tokenizer on this cleaned corpus. We chose the vocabulary size through statistical analysis, based on token frequency distributions and the resulting fertility rates. Token frequency distribution was extracted from the full dataset, and all tokens were ranked in descending order based on their occurrence frequency. We estimated the minimum vocabulary size by computing the cumulative sum of token frequencies and identifying the smallest number of tokens needed to cover a predefined percentage of all token instances. In this work, we selected a 95% coverage threshold, meaning that the vocabulary should include the most frequent tokens that collectively account for 95% of the corpus. This approach reduces model complexity by reducing rare words in our vocabulary that contribute little to the overall data structure. Using this criterion, the resulting minimum vocabulary size was found to be 162. We computed the corresponding fertility rate of 5.88 (see Table 1), which however indicates poor tokenizer performance. Consequently, we further analyzed the efficiency of different

vocabulary sizes aiming at a fertility rate close to 1. We determined that a vocabulary size of 10,000 provides a good balance between vocabulary size and fertility. Although increasing the vocabulary size up to 15,000 slightly reduced the fertility rate, the improvement was negligible and did not yield any measurable benefits in our final analysis of model performance. Thus, to avoid unnecessary model complexity, we selected a vocabulary size of 10,000.

Table 1: Total number of tokens and fertility rate across different vocabulary size for WordPiece-naive

Vocabulary size	162	2000	4000	5000	10000	15000
Total tokens (after tokenization)	1,613,936,272	306,279,707	300,254,836	298,662,500	294,669,840	292,459,964
Fertility rate	5.878	1.115	1.093	1.088	1.073	1.065

After determining the vocabulary size, we tokenized the dataset using the WordPiece tokenizer. We refer to this approach as WordPiece-naive. In this naive method, all command tokens, such as entity and entity's IDs in the sequence, are preserved to examine their effect on tokenization and token prediction performance. One motivation for retaining all terms is that maintaining the full structure may help the model capturing meaningful relationships between elements, thereby improving its understanding of contextual patterns in the data.

3.2.2 DeepCAD

This tokenization method was first introduced in [3]. The process begins by parsing the JSON file into a hierarchical object representation by iterating over all operations within a design. For each operation of type *ExtrudeFeature*, the associated sketch and 2D profile are processed. A local coordinate system is then computed using a transformation matrix. The sketch geometry is converted into a profile object, and key extrusion parameters, such as the direction, are calculated. All *ExtrudeFeature* operations are collected into a list for further processing. Each CAD command is tokenized with a fixed-length vector consisting of a command type and its associated arguments. The supported commands include *Line*, *Arc*, *Circle*, and *Ext* (for extrusion), as well as two structural tokens: *SOL* (Start of Loop) and *EOS* (End of Sequence).

In contrast to standard tokenization methods for natural language, the DeepCAD tokenizer reserves for each tokenized command exactly 16 arguments, representing all possible numerical parameters. To handle the variable presence of parameters across command types, a tokenization mask is used. This mask is defined as a binary array and indicates which parameters are active for each command. For example, sketch-based commands such as *Line* activate only the sketch-related fields (e.g., the coordinates x and y), while all other fields are padded with a masking value (e.g., -1).

After tokenization, each CAD model undergoes a normalization process to ensure geometric consistency across the dataset. First, each sketch profile is independently normalized within its own bounding box. The corresponding scaling factor is stored as part of the associated extrusion command, enabling accurate recovery of the original sketch dimensions during downstream processing or visualization. In addition to sketch-level normalization, a global CAD normalization is applied to the entire model. Specifically, all geometric entities are uniformly scaled such that the maximum absolute coordinate value lies within the range $[-1, 1]$. As a result, all designs are consistently normalized to fit within a canonical $2 \times 2 \times 2$ cube. Following normalization, all continuous values, integers, and floating-point parameters are discretized into the integer range $[0, 255]$. The masked tokens and padding values are assigned to -2 and -1, respectively. When converting these integers into token indices for embedding,

the indices are shifted such that masked tokens map to index 0, padding tokens map to index 1, and valid parameter values 0-255 map to indices 2-257 (see Figure 3c, steps 2 and 3).

This discretization step transforms all geometric and operational parameters into a fixed set of discrete tokens, enabling efficient encoding and compatibility with the transformer-based architecture. The result is a sequence of tokenized commands, each consisting of a command type and its corresponding discrete arguments. These sequences are serialized and stored, serving as the model-ready input for the subsequent BERT training.

3.2.3 WordPiece-DC

We developed a new tokenization method inspired by WordPiece and DeepCAD. We refer to this method as WordPiece-DC. Instead of using a fixed vector length as for DeepCAD, which reserves space for all possible parameters leading to inefficiency and inflexibility, we keep parameters as separate elements. Our approach allows the inclusion of additional data elements without adaptation of the tokenization pipeline. Rather than converting commands into purely numerical representations, we also preserve them in text form. For each JSON file, the full CAD sequence is reconstructed by converting every *ExtrudeFeature* entry into a list of *Extrude* objects. Bounding box information is also extracted to maintain geometric context. Each *Extrude* operation contains one or more sketch profiles composed of curves (*Lines*, *Arcs*, *Circles*). WordPiece-DC converts these curves into a text-preserved, structured sequence. All numerical values are normalized to the range $[-1, 1]$ (min-max normalization over the dataset) to stabilize training. An example is given in Figure 3b, step 2.

Similar to the WordPiece-naive tokenization, we have to determine a suitable vocabulary size. Therefore, we trained the tokenizer with different vocabulary sizes. Although a vocabulary size of 15,000 achieves an ideal fertility rate of 1.0 (see Table 2), it did not improve the final model performance in the subsequent analyses. Therefore, we avoided introducing unnecessary complexity to the model and use a vocabulary size of 5,000 which provided the best performance.

Table 2: Total number of tokens and fertility rate across different vocabulary size for WordPiece-DC

Vocabulary size	965	2000	4000	5000	10000	15000
Total tokens (after tokenization)	10,368,158	9,813,855	9,622,580	9,585,626	9,508,852	9,482,328
Fertility rate	5.878	1.115	1.093	1.088	1.073	1.065

3.3 Model Architecture of Transformer

The model architecture used in this research is a lightweight variant of the original BERT model [5], but consisting of only two encoder blocks. The model was intentionally designed for its reduced parameter count and computational efficiency, making it ideal for rapid experimentation and consistent comparison across different tokenization strategies. While more complex models could potentially yield higher performance, the aim of this research is not to have an optimized model architecture, but to isolate the effect of tokenization methods under a fixed setup. Using the same model structure and trainable parameters across experiments ensures a fair comparison of each tokenization approach.

WordPiece-naive and WordPiece-DC share the same model architecture, as they differ only in their preprocessing steps but ultimately use WordPiece tokenization. Although DeepCAD adopts an embedding strategy similar to that used in natural language processing, where each token is projected into a continuous embedding space, the construction of CAD command embeddings differs in its

composition. Following the approach introduced in DeepCAD [3] (see Figure 2 for an illustration and Table 3 for the configuration), the embedding of each command is computed as the sum of three components. The first component is the command embedding, which represents the semantic category of the operation. Six command types are considered: *Line*, *Arc*, *Circle*, *SOL*, *EOS*, and *Ext*. Each command type is mapped to a learnable embedding vector. The second component is the parameter embedding. Each command is associated with a fixed set of 16 parameters, where each parameter is quantized into an 8-bit integer. These integers are converted into one-hot vectors of dimension 256. Finally, a (learned) positional embedding is added to encode the order of commands within the sequence. The final command representation is obtained by summing the command embedding, the argument embedding, and the positional embedding before being passed to the transformer encoder. The encoder blocks follow the standard transformer architecture, comprising a multi-head self-attention mechanism, identical to that used in the original transformer [14] followed by a feed-forward network, with residual connections, layer normalization, and dropout.

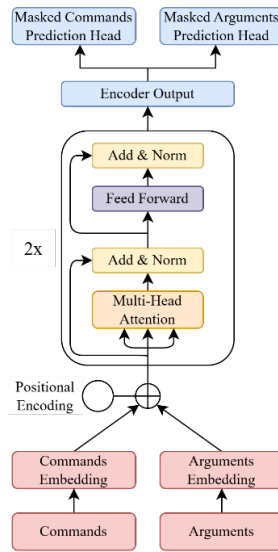


Figure 2: Network architecture of the used transformer for the DeepCAD method. From bottom to top: embedding, added positional encoding, two transformer blocks, two output heads for command prediction and argument prediction.

For the WordPiece models, a masked language modeling (MLM) head is added. It consists of a linear layer that projects the encoder outputs back to the vocabulary size. During training, the predictions are passed through a softmax, and the model is optimized using cross-entropy loss to predict randomly masked tokens. For the DeepCAD representation, a structured prediction head is used to jointly model command types and their associated arguments (Figure 2). This head includes two distinct linear projections: one produces logits over CAD commands, the other predicts arguments, structured according to the expected number and dimensionality of arguments per command. During training, both outputs are passed through a softmax, and the model is optimized using cross-entropy loss. The cross-entropy loss quantifies the difference between the predicted probability distribution and the true distribution of token classes. Cross-entropy loss is especially effective in classification tasks, as it penalizes confident yet incorrect predictions more heavily.

Table 3: Hyperparameters used for building and training the transformer models shared across all three tokenization methods

Hidden size	Feed-forward hidden size	#Blocks	#Attention heads	Dropout	Max learning rate	Min learning rate	Batch size	Warmup epochs
128	512	2	2	0.1	1e-4	1e-5	100	10

3.4 Training

Our lightweight BERT model was trained on an NVIDIA RTX 4090 GPU and optimized using AdamW with a weight decay of 0.01 [15]. To improve training stability, we used a warmup period to ramp up the learning rate to its maximum of $1e-4$ over the first 10 epochs and then reduced the rate following a cosine schedule to a minimum learning rate of $1e-5$ for the remainder of training. Additionally, early stopping was used during training in the sense that if the validation loss did not improve by at least 0.001 after 10 epochs, the training process is terminated. We applied early stopping to allow each model to train as long as required for convergence. For WordPiece-naive, the best model was obtained at epoch 288. In contrast, for WordPiece-DC and DeepCAD, the best models had converged at epochs 281 and 181, respectively. Consequently, the training times for WordPiece-DC (5h) and DeepCAD (4h) were shorter than for WordPiece-naive (29h), also due to the differences in sequence lengths. WordPiece-DC and DeepCAD generate more compact token representations, leading to shorter input sequences and lower attention overhead during training. This factor resulted in faster convergence and more efficient training.

As common, we split the used dataset (that is the DeepCAD dataset) into three parts: 80% of the models for training, 10% for validation, and 10% for testing [16]. We ensured that the CAD designs in each split (training, validation, and test) remained consistent across all models. During training of the WordPiece-naive model, we found that the input sequence length after tokenization posed challenges. With a vocabulary size of 10,000, our statistical analysis showed a maximum sequence length of 573,367 tokens. Standard transformer models exhibit quadratic time and memory complexity with respect to sequence length due to the self-attention mechanism. This leads to GPU memory constraints and extended training time. To address this challenge, we apply a sequence length cutoff strategy. Our statistical analysis show, that the data samples have a wide range of sequence length and contain many outliers. Thus, for practical reasons, we limited the maximum sequence length to 1024 tokens. In contrast, the WordPiece-DC and DeepCAD tokenization methods represent more information by a single token. Thus, these methods were not affected by the sequence length limit resulting in a maximum sequence length of 60 tokens for DeepCAD and 308 tokens for WordPiece-DC, respectively. For WordPiece-DC, each CAD design was represented as a single sequence containing all commands and their corresponding arguments, without structured or fixed-sized arrays. This approach, unlike DeepCAD, avoids padding and allows us to evaluate whether omitting pad values affects model accuracy.

4 Evaluation and Results

We found for WordPiece-naive an optimal vocabulary size of 10,000 and 5,000 for WordPiece-DC. This difference in size is due to the different preprocessing, where the input sequence for WordPiece-DC is reduced to a clean stream of a command with its arguments. Similarly, the DeepCAD tokenization represents commands along with their arguments, however embeds them in a single token. Thus, WordPiece-DC and DeepCAD tokenization produce less tokens than WordPiece-naive (see Table 4), which indicates better efficiency. Notably, the DeepCAD method achieves a significant reduction in fertility, an important advance for processing large CAD models.

We further investigated the information prediction performance for the resulting different data representations. The prediction task for commands and arguments differs in its difficulty. There is a limited number of commands, while the arguments can be various numerical values. Thus, we report the prediction accuracy (perfect match) for commands and arguments separately. For the arguments, we additionally report the normalized mean squared error (NMSE) to indicate how far the predicted values deviate from the data (Table 4).

Table 4: Comparison of the tokenization efficiency and information representation quality

Tokenization Method	Fertility	Total words (input)	Total tokens (after tokenization)	Accuracy (commands)	Accuracy (arguments)	NMSE (arguments)
WordPiece (naive)	1.0731	274,585,076	294,669,840	95.84 %	94.45 %	0.02
WordPiece-DC	1.0164	9,475,439	9,630,567	93.47 %	74.95 %	0.18
DeepCAD	0.2683	9,475,439	2,542,247	100 %	80.89 %	0.10

Although WordPiece-naive showed very good performance for argument prediction, both in terms of accuracy and NMSE, it is not practical for industrial use. The sequence length cutoff led to the loss of a significant amount of information. Furthermore, WordPiece-naive cannot preserve the hierarchical structure of the data, making reverse engineering after prediction both challenging and time consuming (Figure 3a). Thus, reconstructing the CAD model after prediction is challenging.

In contrast, DeepCAD retains all design steps, including numerical values. While the predicted numbers are not always exact, they remain close to the ground truth (see Figure 3c, compare steps 5 and 6), and its structured representation preserves the hierarchy of the design. WordPiece-DC addresses the issue of long sequences of WordPiece-naive by producing a more compact representation. However, our findings show that it underperforms in both classification and regression tasks. For instance, when it is supposed to predict a numerical value, it sometimes predicts a command instead (see Figure 3b, compare steps 5 and 6).

1.	entity Ff101xV43HBZz0_1 entity transform origin y -0.4408 entity transform origin x -0.8157																							
2.	entity	Ff	##11	##01	##xV	##43	##HB	##Zz	##0_1	entity	transform	origin	y	-0.44	##08	entity	transform	origin	x	-8157				
3.	155	885	Mask	2121	2753	683	1440	3315	1430	155	211	314	70	Mask	264	155	211	314	69	Mask				
4.	155	885	3951	2121	2753	683	1440	3315	1430	155	211	314	70	155	264	155	211	314	69	314				
5.	entity	Ff	##11	##01	##xV	##43	##HB	##Zz	##0_1	entity	transform	origin	y	entity	##08	entity	transform	origin	x	origin				
6.	entity	Ff	##11	##01	##xV	##43	##HB	##Zz	##0_1	entity	transform	origin	y	-0.44	##08	entity	transform	origin	x	-8157				

a) WordPiece-naive

1.	Extrude 0.3333 -0.3333 0.3333 -0.3333 0.4537 -0.0915 -0.2503 -0.3227 -0.3333 NewBodyFeatureOperation SymmetricFeatureExtentType															
2.	Extrude	0.3333	-0.3333	0.3333	-0.3333	0.45	##37	-0.091	##5	-0.2503	-0.3227	-0.3333	NewBodyFeatureOperation	SymmetricFeatureExtentType		
3.	117	102	85	102	Mask	1324	235	4936	52	2669	Mask	85	139	Mask		
4.	117	102	85	102	85	1324	235	4936	52	2669	129	85	139	129		
5.	Extrude	0.3333	-0.3333	0.3333	-0.3333	0.45	##37	-0.091	##5	-0.2503	OneSideFeatureExtentType	-0.3333	NewBodyFeatureOperation	OneSideFeatureExtentType		
6.	Extrude	0.3333	-0.3333	0.3333	-0.3333	0.45	##37	-0.091	##5	-0.2503	-0.3227	-0.3333	NewBodyFeatureOperation	SymmetricFeatureExtentType		

b) WordPiece-DC

1.	Sketch-Extrude pair: origin: [192 128 192], normal: [0.2459 -0.2559 -0.9349], x_axis: [-0.3497 0.8761 -0.3318], y_axis: [0. 0. 1.], sketch position: [128 216 155], sketch size: 8, operation: 1, type: 1, extent_one: 129, extent_two: 128															
2.	5	-1	-1	-1	-1	192	128	192	128	206	152	8	129	128	0	1
3.	7	1	1	1	1	194	130	194	Mask	208	154	10	Mask	130	2	3
4.	7	1	1	1	1	194	130	194	130	208	154	10	108	130	2	3
5.	Ext	-1	-1	-1	-1	0.5	0.0	0.5	0.0	0.6094	0.1875	-0.9375	-0.1719	0.0	-1	-0.9922
6.	Ext	-1	-1	-1	-1	0.5	0.0	0.5	0.0	0.6094	0.1875	-0.9375	-0.0078	0.0	-1	-0.9922

c) DeepCAD (Note: Padding values are indicated with -1)

Figure 3: Example of masked token prediction results produced by a) WordPiece-naive, b) WordPiece-DC, and c) DeepCAD model. 1. processed input sequence; 2. tokenized text; 3. token IDs, including mask for MLM; 4. predicted token IDs; 5. final detokenized predictions; 6. ground-truth text. Bold entries refer to the masked tokens, also recognizable by the mask token.

5 Conclusion

This study compared three tokenization methods: “naive” WordPiece, WordPiece with DeepCAD inspired CAD-specific preprocessing, and the CAD-specific approach from DeepCAD. We found that WordPiece-based tokenization can achieve few sub-tokens per word, which is seen as an efficient

representation. When being advanced with domain specific preprocessing, as done in WordPiece-DC, the total token amount is largely lowered, however, this approach yields the lowest performance in terms of both command accuracy and argument accuracy. In contrast, WordPiece-naive tokenization achieves the highest accuracy. Despite this advantage, it is impractical for industrial use due to its complex, inefficient, and excessively long token sequences. Although the DeepCAD approach exhibits lower accuracy than WordPiece-naive, we attribute this difference to the increased complexity of the tokens it processes, which presumably demands a more scaled network architecture. Advantageous is that, unlike the WordPiece-naive method, DeepCAD does not require data truncation, allowing it to retain the full sequence information. However, DeepCAD has several limitations: It relies on fixed vector positions for arguments, with unused positions padded, which reduces efficiency. Additionally, it lacks the ability to generalize to extensions of the CAD representation language, such as primitives not included during the development phase. In summary, our results demonstrate that standard tokenization methods already achieve good performance. Domain specific tokenization can additionally reduce sequence length substantially, which enables the use of larger neural architectures and facilitates the processing of long-range dependencies in more complex, and more realistic, CAD constructions. These results underscore the need for further research into CAD-specific tokenization methods that effectively balance representational compactness and the preservation of predictive performance.

References

- [1] L. Mandelli and S. Berretti, "CAD 3D Model classification by Graph Neural Networks: A new approach based on STEP format," Oct. 30, 2022, *arXiv*: arXiv:2210.16815. doi: 10.48550/arXiv.2210.16815.
- [2] X. Song, A. Salcianu, Y. Song, D. Dopson, and D. Zhou, "Fast WordPiece Tokenization," Oct. 05, 2021, *arXiv*: arXiv:2012.15524. doi: 10.48550/arXiv.2012.15524.
- [3] R. Wu, C. Xiao, and C. Zheng, "DeepCAD: A Deep Generative Network for Computer-Aided Design Models," Aug. 16, 2021, *arXiv*: arXiv:2105.09492. Accessed: Oct. 21, 2024. [Online]. Available: <http://arxiv.org/abs/2105.09492>
- [4] S. Koch et al., "ABC: A Big CAD Model Dataset For Geometric Deep Learning," Apr. 30, 2019, *arXiv*: arXiv:1812.06216. Accessed: Oct. 21, 2024. [Online]. Available: <http://arxiv.org/abs/1812.06216>
- [5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," May 24, 2019, *arXiv*: arXiv:1810.04805. doi: 10.48550/arXiv.1810.04805.
- [6] S. Feist, L. Jacques De Sousa, L. Sanhudo, and J. Poças Martins, "Automatic Reconstruction of 3D Models from 2D Drawings: A State-of-the-Art Review," *Eng*, vol. 5, no. 2, pp. 784–800, May 2024, doi: 10.3390/eng5020042.
- [7] M. F. Alam and F. Ahmed, "GenCAD: Image-Conditioned Computer-Aided Design Generation with Transformer-Based Contrastive Representation and Diffusion Priors," Apr. 08, 2025, *arXiv*: arXiv:2409.16294. doi: 10.48550/arXiv.2409.16294.
- [8] A. Seff, Y. Ovadia, W. Zhou, and R. P. Adams, "SketchGraphs: A Large-Scale Dataset for Modeling Relational Geometry in Computer-Aided Design," Jul. 16, 2020, *arXiv*: arXiv:2007.08506. doi: 10.48550/arXiv.2007.08506.
- [9] A. Seff, W. Zhou, N. Richardson, and R. P. Adams, "Vitrivion: A Generative Model of Parametric CAD Sketches," Apr. 28, 2022, *arXiv*: arXiv:2109.14124. doi: 10.48550/arXiv.2109.14124.
- [10] K. D. D. Willis, P. K. Jayaraman, J. G. Lambourne, H. Chu, and Y. Pu, "Engineering Sketch Generation for Computer-Aided Design," Apr. 19, 2021, *arXiv*: arXiv:2104.09621. doi: 10.48550/arXiv.2104.09621.
- [11] X. Xu, K. D. D. Willis, J. G. Lambourne, C.-Y. Cheng, P. K. Jayaraman, and Y. Furukawa, "SkexGen: Autoregressive Generation of CAD Construction Sequences with Disentangled Codebooks," Jul. 11, 2022, *arXiv*: arXiv:2207.04632. Accessed: Oct. 21, 2024. [Online]. Available: <http://arxiv.org/abs/2207.04632>
- [12] K. D. D. Willis et al., "Fusion 360 Gallery: A Dataset and Environment for Programmatic CAD Construction from Human Design Sequences," May 17, 2021, *arXiv*: arXiv:2010.02392. doi: 10.48550/arXiv.2010.02392.
- [13] M. Ali et al., "Tokenizer Choice For LLM Training: Negligible or Crucial?," in *Findings of the Association for Computational Linguistics: NAACL 2024*, Mexico City, Mexico: Association for Computational Linguistics, 2024, pp. 3907–3924. doi: 10.18653/v1/2024.findings-naacl.247.
- [14] A. Vaswani et al., "Attention Is All You Need," Aug. 02, 2023, *arXiv*: arXiv:1706.03762. Accessed: Oct. 21, 2024. [Online]. Available: <http://arxiv.org/abs/1706.03762>
- [15] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," Jan. 04, 2019, *arXiv*: arXiv:1711.05101. doi: 10.48550/arXiv.1711.05101.
- [16] H. Babaei, M. Zamani, and S. Mohammadi, "The impact of data splitting methods on machine learning models: A case study for predicting concrete workability," *Mach. Learn. Comput. Sci. Eng.*, vol. 1, no. 1, p. 21, May 2025, doi: 10.1007/s44379-025-00021-3.