Syntax-Aware Language Models for Programming Code

(Modele kodu świadome składni języka oprogramowania)

Mateusz Kitzol Patryk Rybak

Praca inżynierska

Promotor: dr Paweł Rychlikowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

17 czerwca 2025

Abstract

This thesis investigates a set of syntax-aware techniques for Python code generation on small transformer models, evaluating their effectiveness as a foundation for later large-scale deployment.

It combines a Pretokenizer, which replaces every Python keyword, operator and structural delimiter with a dedicated control token while enclosing user-defined identifiers and text literals in semantic brackets; a Segmentator that masks entire AST-consistent code spans during fine-tuning, mimicking T5's span-corruption objective but with structure-aware segments; and a Custom Loss with Logits Processor that jointly learn two disjoint sub-spaces (control vs. semantic tokens) and enforce mode-consistent decoding at inference.

Fine-tuning is performed on a cleansed subset of the CodeSearchNet-Python corpus after automatic syntax validation, PEP-8 formatting and deduplication. Compared with a vanilla T5-Large baseline, the best variant (Pretokenizer + standard fine-tuning) improves BLEU from $0.017 \to 0.210$ and ROUGE-L from $0.14 \to 0.38$. On the HumanEval benchmark the model solves one task, indicating emerging semantic competence despite its modest size. These results show that syntax-aware training can deliver large quality gains within small-scale transformer architectures, demonstrating the possibility of scaling some of the techniques to larger models and more complex coding tasks.

Contents

Chapter 1

Introduction

In November 2022, OpenAI released a revolutionary chatbot based on the GPT-3.5 transformer architecture. It quickly became the world's fastest adopted tool, gaining 1 million users in just five days. The product turned out to be a breakthrough in Artificial Intelligence, placing Large Language Models (LLMs) at the forefront of public discussion. Although many similar models had been developed before, this was the first time a language model became accessible at such a massive scale. OpenAI's success encouraged other companies to release competing products, resulting in a wave of new chatbots entering the market.

This phenomenon captured the imagination of programmers worldwide, many of whom wanted to join this technological race. However, ordinary enthusiasts were not able to contend with tech giants. The enormous computational power required for success in this field was simply out of reach for most individuals.

Nevertheless, a different approach enabled independent developers and startups to participate in AI development. While the biggest companies were occupied with developing more and more complicated models, the less popular firms started to create light, open-source ones. Although they were not comparable in terms of general performance, they offered the unique advantage of being adaptable to perform exceptionally well on specific tasks. The fewer trainable parameters made them available to developers with less powerful hardware. More and more people started to experiment on their own with these lighter models. The standard technique, known as fine-tuning, was based on delivering additional data related to the specific problem to the model.

It quickly became clear that careful data preprocessing can sometimes be even more important than the data itself. As humans, we often perceive a broader context that is not explicitly encoded in raw text. With careful examination of such non-obvious intuitions, we can come up with many creative ideas to help the model better understand the nuances of the task. Naturally, the results may vary, as incorporating additional information often increases computational cost. This makes it necessary to find efficiencies elsewhere. Ultimately, it's a process of trial and error - one that demands patience and resilience to uncover meaningful correlations between our adjustments and the outcomes.

In this work, we pursue this direction to address the challenging task of automated code generation. We decided to focus on the Python programming language as it presents several advantages aligned with our objectives. Firstly, its popularity makes it easy to find qualitative datasets for finetuning. Secondly, Python's relatively simple syntax, especially when compared to more complex languages, increases the likelihood that smaller models can learn recurring patterns effectively. Lastly, Python offers a rich ecosystem of libraries that can handle side tasks (such as data preparation or syntax examination) easily. Therefore, we can focus on the core challenges without being absorbed by secondary issues.

We aim to experiment with various ideas that leverage characteristics of the Python language and coding in general. Although our computational resources are limited, we would try to evaluate the potential of different methods that, when scaled, can make a significant difference.

Chapter 2

Background

2.1 Fundamentals of Deep Learning

In recent years, deep learning has become the dominant domain of machine learning, particularly for tasks involving unstructured data such as images, audio, or natural language. Deep learning refers to the use of multilayered artificial neural networks to approximate complex functions.

2.1.1 Neural Networks

The fundamental unit of a neural network is the perceptron[20]. It was first introduced in the year 1958 by Frank Rosenblatt in his work. It is a simple construct that performs a linear transformation on its input using assigned weights, adds a bias term, and passes the resulting value through a nonlinear activation function such as the sigmoid or ReLU[11] (Rectified Linear Unit).

Mathematically, for a perceptron with input $x_1, x_2, \ldots, x_n$, weights $w_1, w_2, \ldots, w_n$, and bias b, the output y is given by:

$$y = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where ϕ is the activation function.

A single perceptron, often referred to as a neuron, can be considered the simplest form of a neural network. In more complex architectures, perceptrons are organized into layers, which are then connected in various ways. In feedforward neural networks (FNNs), information flows strictly in one direction—from input to output—without cycles. Typically, neurons within a given layer are not connected to each other but are connected to neurons in adjacent layers.

The structure and types of layers used—such as fully connected, convolutional, or recurrent layers—define the so-called *network architecture*. This term refers to a general design pattern describing a family of models that share similar structural characteristics.

2.1.2 Learning and Loss Function

Most deep learning tasks are formulated as a supervised learning problem, where the model is trained on input-output pairs (x, y). The goal of training the network is to model a function $f_\theta(x) \approx y$, where θ represents the parameters of the network's perceptrons (i.e., weights and biases), such that f_θ approximates the true function $f(x) = y$ as closely as possible.

To evaluate how well the network approximates the desired function, a loss function $L(y, \hat{y})$ is used, where $\hat{y} = f_\theta(x)$. Typical loss functions include mean squared error for regression tasks and cross-entropy for classification. The training process aims to minimize the loss function value over the training data x via backpropagation.

Backpropagation is outlined very nicely on Wikipedia. The following quotation highlights the core idea:

> *Backpropagation is an efficient application of the chain rule to neural networks. It computes the gradient of a loss function with respect to the weights of the network for a single input–output example, and does so efficiently, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule* [19].

2.1.3 Neural Network Architectures

In deep learning, the architecture of a neural network largely determines how effectively it can model complex data. In the field of natural language processing, progress has been closely tied to the emergence of groundbreaking architectural innovations. In the following section, we will take a closer look at three particularly influential architectures that have played a key role not only in NLP, but also in many other domains.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are built for sequences — text, speech, or time series data, where what came before influences what comes next. They differ from feedforward networks because of their recurrent connections. The output of one processing step becomes an input for the subsequent one. This loop is key to how they capture patterns over time.

At their core, RNNs maintain a hidden state that evolves with each time step. This state is updated based on the current input and its own previous value, effectively giving the network a memory of past events. They can be found in problems like handwriting and speech recognition, various natural language processing tasks, and machine translation.

However, traditional RNNs struggle with learning long-term dependencies due to the vanishing gradient problem, which led to the development of long short-term memory (LSTM) networks.

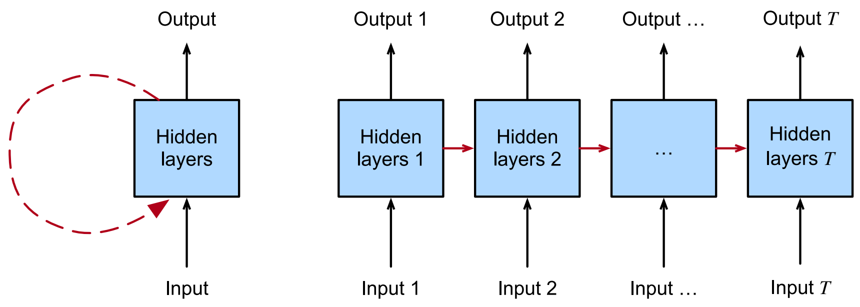

Figure 2.1: *On the left recurrent connections are depicted via cyclic edges. On the right, we unfold the RNN over time steps. Here, recurrent edges span adjacent time steps, while conventional connections are computed synchronously.* [2]

Image taken from `https://d2l.ai/chapter_recurrent-neural-networks/index.html`

Long Short-Term Memory

Long Short-Term Memory (LSTM) networks are a special kind of RNN designed to better capture long-range dependencies in sequences. Unlike standard RNNs, LSTMs use **memory cells** with internal states and a gating mechanism that controls the flow of information. The three main gates are:

- **Input gate** – decides how much new information from the current input should be added to the memory.
- **Forget gate** – determines what information from the previous memory should be discarded.
- **Output gate** – controls how much of the memory is used to compute the current output.

Each gate is implemented as a fully connected layer with a sigmoid activation, producing values between 0 and 1 that act as "soft" switches. The memory update is performed using elementwise (Hadamard) multiplication, combining the influence of the input and forget gates. This design enables LSTMs to retain relevant information over long sequences and effectively addresses the vanishing gradient problem.

To compute the output at each time step, the internal memory is first passed through a `tanh` activation, then modulated by the output gate. This ensures that the hidden state remains bounded and selectively exposes relevant information. In essence, LSTMs learn *what to remember, what to forget, and what to output*[1], making them powerful tools for sequence modeling tasks.

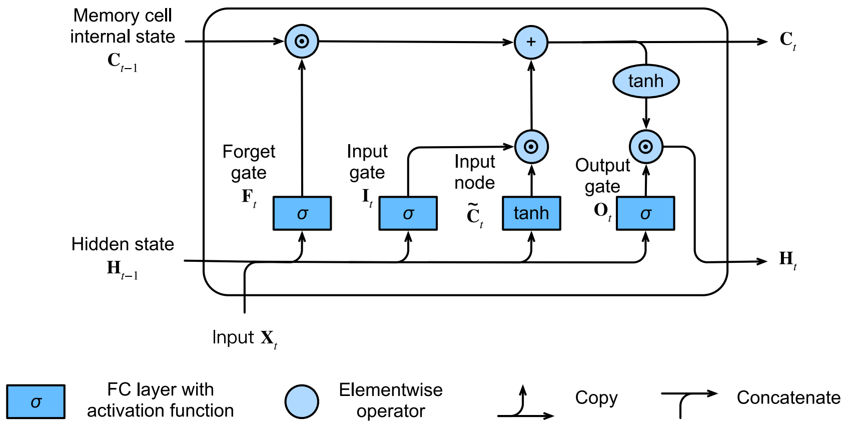

Figure 2.2: *Computing the hidden state in an LSTM model.* [1]

Image taken from `https://d2l.ai/chapter_recurrent-modern/lstm.html#fig-lstm-0`

Convolutional Neural Networks

A typical CNN architecture for NLP consists of the following layers:

- **Input Layer**: Represents the text input, often as a matrix where each row corresponds to a word embedding vector of a word in the sentence.
- **Convolutional Layer**: Applies filters (kernels) that slide over the input matrix to capture local features. Each filter is designed to detect specific patterns, such as n-grams.
- **Activation Function (e.g., ReLU)**: Introduces non-linearity into the model, allowing it to learn complex patterns.
- **Pooling Layer**: Reduces the dimensionality of the feature maps obtained from the convolutional layer, typically using max-pooling to retain the most significant features.

- **Fully Connected Layer**: Flattens the pooled features and passes them through dense layers for final classification or regression.

CNNs can be also applied at the character level and treat text as sequences of characters, enabling the model to capture subword information, handle misspellings and manage out-of-vocabulary words. This approach is especially useful for morphologically rich languages and noisy text.

To capture complex patterns and hierarchical features, deeper CNN architectures have been developed:

- **Very Deep CNNs**[9]: Stack multiple convolutional and pooling layers to learn intricate text representations.
- **Deep Pyramid CNNs**[6]: Use a pyramid structure that progressively reduces feature map dimensions, capturing global context effectively.

These architectures enhance the model's capacity to understand nuanced linguistic structures and long-range dependencies, but despite their strengths, they are not the only successful architecture in NLP. In recent years, a new paradigm based on self-attention mechanisms namely, the Transformer architecture has emerged and significantly advanced the state of the art in many NLP tasks.

2.2 Transformers

The Transformer is a neural architecture built around the multi-head attention mechanism. Text is first tokenized into numerical representations, and each token is mapped to a vector using a word embedding table. At every layer, tokens are contextualized using parallel multi-head attention, which highlights important tokens and downplays less relevant ones. Unlike earlier RNN models such as LSTMs, Transformers have no recurrent components, allowing for faster training. They became the foundation for large language models (LLMs) trained on massive text corpora.

In the following sections, we will look at the three main Transformer architectures: encoder-only, decoder-only, and encoder-decoder models.

2.2.1 Encoders

An encoder architecture starts with an embedding layer, followed by a series of encoder layers. Within each encoder layer, two primary components are found which are: a self-attention mechanism and a feed-forward network. The input sequence of vectors first is processed via the self-attention mechanism, which produces an intermediate sequence representation which is next passed through the feed-forward network, within which each vector is handled independently. The output generated from this stage serves as the input for the subsequent encoder layer, leading to a progressive enrichment of the intermediate representation.

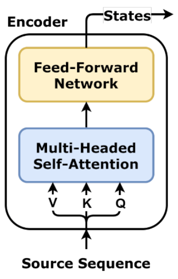

Figure 2.3: *One encoder layer.* [21]

Image taken from `https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)`

Training of such an architecture is performed using a ”masking and guessing” manner. The model receives input with some tokens replaced by a special mask token, and its task is to maximize probability for the original tokens in the masked positions.

The state-of-the-art models of this kind are BERT and its numerous extensions e.g. RoBERT. Their objective is to process input into representations that encapsulate semantic information from the entire input. This is beneficial for tasks like sentiment analysis, feature extraction, text comparison, and many others where text generation is not the primary requirement, although it is also achievable with this architecture.

2.2.2 Decoders

A decoder architecture starts with an embedding layer, followed by a series of decoder layers, and ends in an un-embedding layer. Within the decoder layer, three primary components are found: a causally masked self-attention mechanism, a cross-attention mechanism, and a feed-forward network. The input is processed by the masked self-attention mechanism, where masking prevents information flow from future tokens to enable autoregressive generation. Subsequently, in architectures that include a distinct encoder (i.e., not in ”decoder-only” models), the cross-attention mechanism is employed. This allows the decoder to draw relevant information from the output representations previously generated by said encoder. Following this, the feed-forward network processes each vector independently. The output generated from this stage serves as input to the subsequent decoder layer, leading to progressive enrichment of the intermediate representation.

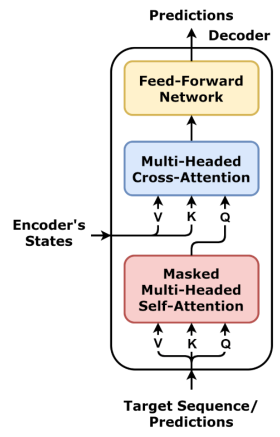
Predictions
Decoder
Feed-Forward Network
Multi-Headed Cross-Attention
V K Q
Encoder's States
Masked Multi-Headed Self-Attention
V K Q
Target Sequence/ Predictions

Figure 2.4: *One decoder layer.* [21]

Image taken from `https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)`

In contrast to the encoder architecture, training consists of predicting the next token based on all previous ones. The model receives a sequential input and learns to generate its shifted version, so that each token maximizes the probability of its successor.

Decoders are designed for autoregressive sequence generation, which makes them effective in tasks such as text writing and continuation. As a result, work well in applications like chatbots, creative writing, prompt-based reasoning, and similar generative tasks. GPT family models are a good example

2.2.3 Encoder-Decoders

The foundation of the Transformer model, in its original concept as presented in the paper 'Attention is All You Need,' is an encoder-decoder architecture which is connecting two previously described ones. Like previously, the encoder component, comprising a sequence of encoding layers, handles the simultaneous processing of the complete input sequence. The decoder, also built from dedicated layers, operates in an iterative mode. It utilizes data from the encoder and its own previously generated elements to progressively build the output sequence in an autoregressive manner.

Each encoder layer aims to generate contextual representations of tokens; each such representation applies to a token that aggregates information from the other input tokens via the self-attention mechanism. Decoder layers, possess two distinct attention sublayers — the first is cross-attention, enabling the incorporation of the encoder's output (i.e., the contextualized representations of input tokens), and the second is self-attention, which serves to consolidate information among the tokens already input to the decoder (generated at the current inference stage).

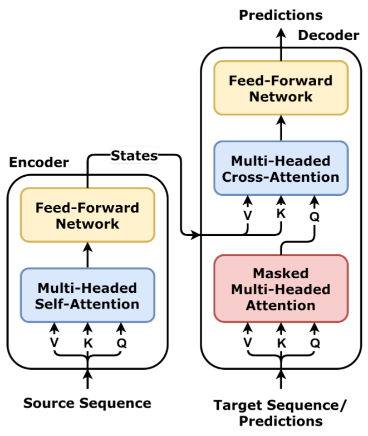
Predictions
Decoder
Feed-Forward Network
Multi-Headed Cross-Attention
V K Q
Masked Multi-Headed Attention
V K Q
Target Sequence/ Predictions
Encoder
States
Feed-Forward Network
Multi-Headed Self-Attention
V K Q
Source Sequence

Figure 2.5: *Encoder-decoder architecture.* [21]

Image taken from `https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)`

Classic examples of models with this architecture are T5[17] and BART[12]. Due to their shared architecture, these models typically perform well on similar tasks; however, their training methods differ, which creates room for potential advantages. The general task of such models is to transform one input sequence into another output sequence (e.g., processing natural language input text into output text tailored to a specific task), which allows them to be effectively used for problems such as machine translation or text summarization.

T5 (Text-to-Text Transfer Transformer)

The T5 model's key innovation is the unification of all NLP tasks under the text-to-text paradigm. This means that every task, from translation to classification, is formulated as a transformation of one text sequence into another, often using textual prefixes to define the specific task (e.g., `translate English to German:`).

Its pre-training on the vast C4[3] (Colossal Clean Crawled Corpus) dataset primarily utilized a learning objective based on the **"span corruption"** technique. In this method, random spans of text are replaced by a single masking token, and the model learns to reconstruct the original, removed content of these spans. While its architecture is a standard Transformer-based encoder-decoder, T5's main contribution was not revolutionary architectural changes, but rather the thorough, systematic study and optimization of various aspects of training, tasks, and datasets for Transformers conducted by its creators.

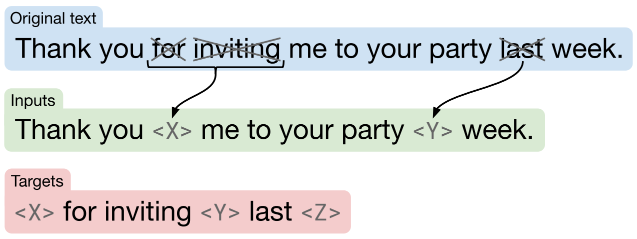
Original text
Thank you ~~for inviting~~ me to your party ~~last~~ week.
Inputs
Thank you <X> me to your party <Y> week.
Targets
<X> for inviting <Y> last <Z>

Figure 2.6: *Schematic of the objective we use in our baseline model. In this example, we process the sentence "Thank you for inviting me to your party last week." The words "for", "inviting" and "last" (marked with an* $\times$*) are randomly chosen for corruption. Each consecutive span of corrupted tokens is replaced by a sentinel token (shown as <X> and <Y>) that is unique over the example. Since "for" and "inviting" occur consecutively, they are replaced by a single sentinel <X>. The output sequence then consists of the dropped-out spans, delimited by the sentinel tokens used to replace them in the input plus a final sentinel token <Z>.* [21]

Image taken from `https://arxiv.org/pdf/2401.03003`

The T5 model is highly scalable and has been released in several versions with different parameter counts:

- **Small** (approx. 60 million parameters)
- **Base** (approx. 220 million parameters)
- **Large** (approx. 770 million parameters)
- Larger variants: **3B** (approx. 2.8 billion parameters) and **11B** (approx. 11 billion parameters).

T5's focus on the universal text-to-text paradigm, in contrast to models primarily focusing on various denoising objectives, has contributed to its wider adoption as a foundation in subsequent research on code generation.

2.3 Language Models for Code

Transformer models have revolutionized Natural Language Processing and opened new possibilities in the analysis and generation of source code. Although source code possesses a formal and rigid structure, it shares many characteristics with natural language, such as local and global dependencies and the semantic meaning of tokens.

Models for code can be divided into several categories based on their architecture and primary application:

- **Code Understanding (Encoder-based Models)**: These models analyze code to create rich representations for tasks like classification (e.g., programming language detection), code search based on natural language descriptions, and bug detection. A state-of-the-art example is CodeBERT [7], a BERT-based model pre-trained on a large corpus of code and associated comments. It was trained using a Masked Language Modeling objective.
- **Code Generation (Decoder-based Models)**: These models specialize in creating new code for tasks such as code completion, code translation, and text-to-code synthesis. An early yet influential adaptation is CodeGPT, based on the GPT-2 [16] architecture, which demonstrated the potential for generating syntactically correct code blocks.
- **Universal Models (Encoder-Decoder)**: This architecture enables models to perform both understanding and generation tasks. A classic example is CodeT5 [18], a model based on the T5 architecture. It is trained on a diverse set of tasks, from code generation to summarization, making it a powerful and flexible tool.

A New Generation: Large-Scale Models and Coding Assistants

The aforementioned models laid the foundation for a new generation of tools: large-scale language models that act as programmer's assistants. Their development has followed several key trends:

- **General-Purpose Assistants**: The breakthrough in this category was OpenAI Codex [16], a model from the GPT-3 [4] family that forms the core of GitHub Copilot. Trained on billions of lines of code and natural language text, it can generate complex functions and algorithms from natural language descriptions, significantly boosting developer productivity and changing their interaction with code editors.
- **Models for Competitive Problem-Solving**: A different direction is represented by DeepMind's AlphaCode [14], which focuses on deep algorithmic reasoning. This model demonstrated the ability to solve problems from programming competitions at a level comparable to human experts. This requires synthesizing complete and correct solutions from scratch, rather than just completing existing code.
- **Open-Source Alternatives**: In response to proprietary models, the research community has developed powerful, open-source alternatives. A key example is StarCoder [13], trained as part of the BigCode project on "The Stack," a carefully curated and permissively licensed dataset of code. The availability of such models democratizes research and enables the creation of specialized tools without relying on closed, commercial infrastructure.

2.4 Metrics

Metrics play a crucial role in evaluating the performance of language models. Their deterministic and well-defined nature enables objective comparison of outputs across different models. In this work, we employ three types of evaluation metrics: BLEU, ROUGE, and HumanEval. The first two are based on surface-level text similarity. They are easy to compute and widely adopted, but were originally designed for general natural language tasks rather than code generation. As a result, they may penalize semantically correct code that uses different variable names or structures. Nevertheless, due to their simplicity and accessibility, we use them to facilitate comparisons across models in our experiments. In contrast, HumanEval is specifically designed for evaluating code generation models. While it is more complex, it serves as the final benchmark for our best-performing model, offering a more realistic and task-specific assessment of its code generation capabilities.

2.4.1 BLEU

BLEU [10] is primarily focused on whether the tokens in the prediction also appear in the reference. It is not highly sensitive to extra (unnecessary) tokens in the prediction as long as they don't dominate the output (due to the use of modified n-gram precision). Only short predictions are penalized by the brevity penalty. To understand the BLEU metric, we must first introduce two key components that contribute to the final score: the **modified n-gram precision** and the **brevity penalty**.

Modified n-gram Precision

$$p_n = \frac{\sum_{i=1}^{M} \sum_{s \in G_n(\hat{y}^{(i)})} \min\left(C(s, \hat{y}^{(i)}), C(s, y^{(i)})\right)}{\sum_{i=1}^{M} \sum_{s \in G_n(\hat{y}^{(i)})} C(s, \hat{y}^{(i)})}$$

where:

- M is the number of evaluated examples (prediction–reference pairs);
- $\hat{y}^{(i)}$ denotes the i-th candidate (predicted) sequence;
- $y^{(i)}$ denotes the i-th reference sequence;
- $G_n(x)$ denotes the set of all n-grams in sequence x;
- $C(s, x)$ denotes the number of occurrences of substring s in x.

Brevity Penalty (BP)

$$c = \sum_{i=1}^{M} \operatorname{len}(\hat{y}^{(i)}), \quad r = \sum_{i=1}^{M} \operatorname{len}(y^{(i)})$$

$$\mathrm{BP} = \begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r \end{cases}$$

where:

- c is the total length of all predicted sequences;
- r is the total length of all reference sequences.

Final BLEU Score

The final BLEU score is computed as the geometric average of n-gram precisions, scaled by a brevity penalty:

$$\mathrm{BLEU} = \mathrm{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

where $w_n = \frac{1}{N}$ for $n = 1, 2, \ldots, N$ (typically $N = 4$)

2.4.2 ROUGE

ROUGE [15] evaluates the similarity between predicted and reference sequences by computing the overlap of n-grams or subsequences. Unlike BLEU, which is primarily precision-oriented, ROUGE incorporates both **recall** and **precision**. Recall measures how much of the reference content is successfully captured by the prediction, while precision assesses how much of the predicted content overlaps with the reference. These two components are typically combined into an F1 score, which balances completeness and accuracy and serves as the final ROUGE metric.

ROUGE-1 computes this overlap based on unigrams, while ROUGE-2 considers bigram matches. ROUGE-L further generalizes the metric by using the length of the Longest Common Subsequence (LCS) between the prediction and the reference to compute both precision and recall.

ROUGE-1 F1 and ROUGE-2 F1

$$\mathrm{Recall}_n = \frac{\sum\limits_{s \in G_n(\hat{y})} \min\left(C(s, \hat{y}), C(s, y)\right)}{\sum\limits_{s \in G_n(y)} C(s, y)}$$

$$\text{Precision}_n = \frac{\sum_{s \in G_n(\hat{y})} \min\left(C(s,\hat{y}), C(s,y)\right)}{\sum_{s \in G_n(\hat{y})} C(s,\hat{y})}$$

$$\text{F1}_n = \frac{2 \cdot \text{Precision}_n \cdot \text{Recall}_n}{\text{Precision}_n + \text{Recall}_n}$$

where:

- $n = 1$ for ROUGE-1 and $n = 2$ for ROUGE-2;
- $G_n(x)$ denotes the set of all n-grams in sequence x;
- $C(s, x)$ denotes the count of n-gram s in sequence x;
- $\hat{y}$ is the model prediction and y is the reference sequence.

ROUGE-L

Let $n = \text{len}(\hat{y})$ and $m = \text{len}(y)$. Then:

$$\text{Precision}_{\text{LCS}} = \frac{\text{LCS}(\hat{y}, y)}{n}, \quad \text{Recall}_{\text{LCS}} = \frac{\text{LCS}(\hat{y}, y)}{m}$$

$$\text{F1}_{\text{ROUGE-L}} = \frac{2 \cdot \text{Precision}_{\text{LCS}} \cdot \text{Recall}_{\text{LCS}}}{\text{Precision}_{\text{LCS}} + \text{Recall}_{\text{LCS}}}$$

2.4.3 HumanEval

The final evaluation metric used in this work is fundamentally different from those previously discussed, as it does not rely on textual similarity. HumanEval [5] is a benchmark specifically designed to assess the functional correctness of code generated by language models. It comprises a curated set of 164 programming problems, each framed as a function body completion task.

Each problem in the HumanEval suite consists of three components:

- a **docstring**, which provides a natural language specification of the desired functionality;
- a **function signature**, which defines the function's name, parameters, and return type;
- a **set of unit tests**, which serve as the ground truth for evaluating the correctness of the generated implementation.

In contrast to n-gram-based metrics such as BLEU or ROUGE, HumanEval directly measures a model's ability to produce syntactically valid and semantically correct code.

2.5 Hyperparameters

Even though our model is already trained and ready for use, the generation process can still be customized. While we cannot directly change the probabilities assigned to tokens, we can indirectly influence how the model selects the next token by adjusting certain hyperparameters. These settings do not alter the model itself but affect its behavior during inference. In the following section, we describe some of the hyperparameters that proved particularly useful in our experiments and helped enhance the final results.

- **Sampling**: A boolean flag that controls the randomness of token selection.
  - When set to `True`, the next token is chosen randomly according to the probability distribution predicted by the model.
  - When set to `False`, the most probable token is always selected deterministically.
- **Temperature**: a parameter that rescales the probability distribution over tokens (only used when Sampling set to `True`).
  - When set below 1, it makes the distribution sharper, causing the model to behave more deterministically by favoring high-probability tokens.
  - When set above 1, it makes the distribution flatter (softer), encouraging the model to be more creative by sampling from a wider range of tokens.
- **Number of beams**: An integer (by default 1) specifying how many parallel sequences the model generates during decoding to select the most promising one. This hyperparameter helps prevent the model from committing too early to sequences that appear optimal initially but later diverge into less probable or incoherent completions.
- **Length penalty**: A hyperparameter that, when the number of beams is greater than 1, adjusts the model's preference for sequence length during generation.
  - When set below 1, it penalizes longer sequences, favoring more concise outputs.
  - When set above 1, it penalizes shorter sequences, encouraging longer completions.

Chapter 3

Our Approach

3.1 Data preparation

3.1.1 Dataset

Selecting an appropriate dataset is a critical first step in any model fine-tuning process. This is especially true in a sensitive domain like code generation, where the data must meet several specific criteria to be effective.

First and foremost, the dataset should be structured in a way that allows for clear separation between the input prompt and the expected output. In this context, leveraging the natural structure of code - functions - is particularly useful. Well-written functions typically solve a single, well-defined problem, which aligns closely with the requirements of language models. This makes it feasible to use the function's docstring as the input (prompt) and the function body as the desired output.

Another important consideration is the size of the dataset. Language models require large amounts of data to learn meaningful patterns and generalize effectively. With this in mind, we selected the CodeSearchNet Python dataset, which contains over 450,000 Python functions paired with corresponding docstrings. These examples are sourced from open-source repositories, making the dataset both extensive and suitable for our task.

3.1.2 Data preprocessing

Even if the dataset appears to be of high quality, it is good practice to apply custom preprocessing to ensure it meets our specific requirements. To this end, we developed a preprocessing pipeline that enhances the original dataset through several key steps.

First, we verify the syntactic correctness of each function by attempting to parse it into an abstract syntax tree (AST). If parsing fails, it indicates that the interpreter would likely raise a syntax error. To avoid reinforcing incorrect patterns during training, we discard such functions.

Next, we ensure that our function adheres to Python Enhancement Proposal 8 (PEP8) — the official style guide for Python. PEP8 promotes best practices for writing clean and readable code, such as consistent indentation and limiting line length. To enforce these standards automatically, we use the autopep8 library. This helps eliminate stylistic inconsistencies that might otherwise introduce noise and hinder the learning process.

We then check for duplicate functions by applying a simple hashing technique, ensuring that repeated entries are removed.

Finally, we remove comments from within function bodies. Although comments can occasionally offer useful context, they are often written inconsistently and lack a standardized format. Moreover, they may interrupt the logical flow of consecutive instructions, making it more difficult to model the underlying reasoning. To reduce this potential source of noise, we choose to eliminate them. We also discard docstrings, as previously noted, since they serve as the input in our setup, while the function body represents the output.

Listing 3.1: Example of Function Before and After Preprocessing

```python
# Before Preprocessing:
def bbox(self):
        """BBox"""
        return self.left, self.top, self.right, self.bottom

# After Preprocessing:
def bbox(self):
    return self.left, self.top, self.right, self.bottom
```

At this stage, the preprocessed data is ready for use. However, in the following chapters, we introduce an additional augmentation step that will be applied just before fine-tuning.

3.2 Fine-Tuning features

The elements of our methodology described below are motivated by the following intuition.

When training a relatively small model, it is crucial not to overload its weights with too many correlations to learn. If the model's "capacity" is insufficient, these dependencies might compete, potentially resulting in the model failing to learn any of the intended relations.

Keywords in Python, such as `for` or `if`, overlap with words found in natural language, yet carry completely different meanings and usage contexts. If such words are left indistinguishable in the embedding space, we implicitly expect the model weights to represent radically different semantics using the same token. To address this issue, we explicitly differentiate between the syntax of two domains — natural language and programming language — by introducing additional tokens into the model. Each Python keyword is assigned a unique token unrelated to the natural language. This process is implemented through the Pretokenizer, described in the later section.

However, this approach comes with a significant risk. If training is performed without selective updating of embeddings and weights, we risk degrading the model's knowledge of natural language. To mitigate this, we modified the loss computation and carefully considered which model parameters should be updated during training, as detailed in the Custom Loss section.

Furthermore, inspired by the work AST-T5[8] of Linyuan Gong, Mostafa Elhoushi and Alvin Cheung, we implemented segmentator, which is discussed later.

3.2.1 Pretokenizer

This tool modifies the source code in a programming language in a way that explicitly distinguishes language keywords and enables a tokenizer to tokenize them differently from their natural language counterparts.

Before describing the functionality of the pretokenizer, we distinguish between the `control` and `semantic` parts of the code:

- The control part includes all syntactic elements of the programming language, such as:
  - keywords (e.g., `def`, `if`),
  - operators (e.g., `=`, `<=`, \\),
  - brackets and the dot symbol,
  - indentation, newline characters, and block delimiters such as the colon (`:`).
  - EOS token

  A complete list of control elements and their mappings to token representations can be found in appendix A.

- The semantic part consists of all fragments of code that are not part of the control structure. In other words, these are elements that carry semantic meaning, and their content is defined by the programmer. This includes variable names, function and class names, module identifiers, as well as string contents and numerical values.

After this distinction, we can easily define two groups of tokens that we use in the training and subsequent sections.

- **Control tokens** – Tokens forming the representation of the control part (all tokens in this group are newly added to the tokenizer) – used solely in generation (output).
- **Semantic tokens** – Tokens used for the representation of the semantic part. These are used in the tokenization of docstrings (input) and the generation of the semantic part of the code (output).

Additionally, the control tokens set includes the `[SEMANTIC_START]` token, and similarly, the semantic tokens set includes `[SEMANTIC_STOP]`. This is necessary for the model to learn to distinguish between generation modes, which will be further elaborated in the "Custom Loss" and "Logits Processor" sections.

The `Pretokenizer` uses Python's out-of-the-box `ast` module, which allows the processing of the language's abstract syntax in the form of an Abstract Syntax Tree (AST). Below is an example showing the parsed code in two forms:

¡ dwa zdjecia obok siebie, kodu i reprezentacji po parsowaniu ¿

The `Pretokenizer` is implemented in the style of the *visitor design pattern*, with awareness of the composite AST structure. It is a modification of an existing internal visitor class from the `ast` module, namely the private class `_Unparser`.

By distinguishing node types based on their syntactic role, the tool can effectively separate the control and semantic parts of code previously described at a fine-grained level. During string construction, control elements such as keywords are replaced with corresponding tags — typically, these are the keywords written in uppercase and wrapped in square brackets (e.g., `[IF]`, `[DEF]`).

Semantic parts of the code remain unchanged in content but are enclosed with `[SEMANTIC_START]` and `[SEMANTIC_STOP]` markers.

The use of semantic markers and the method for representing indentation are configurable in the `Pretokenizer` class. For our experiments, we opted for a representation that maximizes information retention — an example of which is shown below:

Listing 3.2: The python code and corresponding pretokenized version

```
def add(a, b):
    return a + b

[DEF] [SEMANTIC_START] add [SEMANTIC_END] [DELIMIT_1_L]
   ↪ [SEMANTIC_START] a [SEMANTIC_END] [COMMA]
   ↪ [SEMANTIC_START] b [SEMANTIC_END] [DELIMIT_1_R] [BLOCK]
[INDENT]
```

```
    [RETURN]
    [SEMANTIC_START] a [SEMANTIC_END] [ADD] [SEMANTIC_START] b
    ↪ [SEMANTIC_END]
[DEDENT]
```

The decision to use semantic markers is essential for the design of the `Custom Loss` component, as well as for the inference phase of the trained model.

3.2.2 Custom Loss

Custom Loss was developed as a response to the potential problem of knowledge degradation when adding new tokens and attempting to continue training. It consists of two parts - selective token learning and a classifier.

Selective Learning

When running training for a new task (i.e., translating natural language to Python code), it is natural for the model not to be accustomed to using new tokens, and consequently, many predictions will miss. Among these, we can distinguish misses that we consider useful and utilize for backpropagation. However, there are also those that, in our opinion, are detrimental from the perspective of preserving natural language knowledge and the accompanying ”understanding” of docstrings (inputs). Let’s differentiate these types of misses:

- **The last token in the sequence is a control token, and the expected prediction is also a control token:**
  1. The predicted token is a mismatching control token – OK
  2. The predicted token is a semantic token – NOT OK
- **The last token in the sequence is a semantic token, and the expected prediction is also a semantic token:**
  1. The predicted token is a control token – NOT OK
  2. The predicted token is a mismatching semantic token – OK
- **The last token in the sequence is `[SEMANTIC_START]` (a control token), and the expected prediction is a semantic token:**
  1. The predicted token is a control token – NOT OK
  2. The predicted token is a mismatching semantic token – OK
- **The last token in the sequence is `[SEMANTIC_STOP]` (a semantic token), and the expected prediction is a control token:**

1. The predicted token is a mismatching control token – OK
2. The predicted token is a semantic token – NOT OK

Where,
OK – stands for "considered in the learning process"
NOT OK – stands for "not considered in the learning process"

The decision to disregard predictions whose tokens are from an "incorrect context" (i.e., different from the expected one) is crucial, in our opinion. If we didn't do this, token embeddings could shift their positions within the embedding space relative to tokens from their incorrect context, and crucially, also from their own. This could potentially lead to the loss of existing relationships with tokens from their original context. This issue primarily impacts semantic tokens, as control tokens are still in the process of being learned.

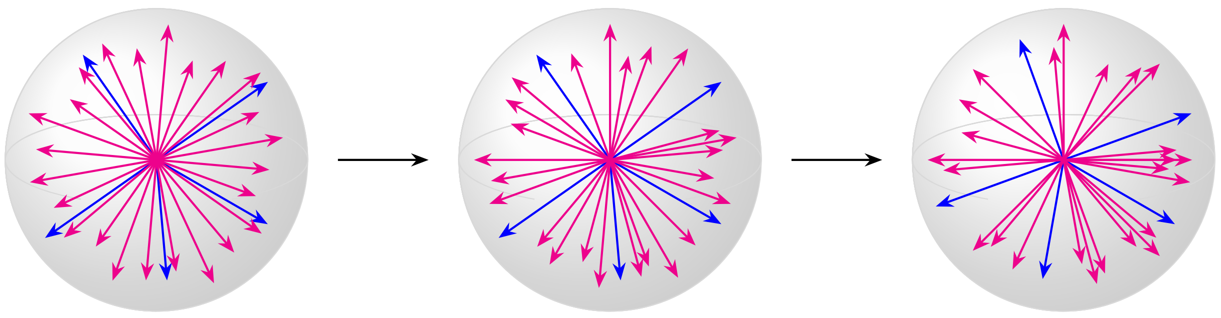
Figure 3.1: Expected progressive knowledge degradation in the semantic token embedding space when a custom loss function is not applied. Magenta arrows represent semantic token vectors, blue arrows represent control token vectors. Straight arrows between spheres represent successive training steps. Notice how the blue vectors tend to push away in the space, while the magenta ones drift away from them over time.

One might assume this problem could be mitigated if we included examples of natural language generation (using semantic tokens) in our training. However, our task focuses on code generation, where original tokens are used much less frequently. Hence, the idea to apply the principles mentioned above and thereby treat and construct these two distinct groups of tokens as maximally independent subspaces within the same embedding space.

The loss value calculated in this manner is as follows:

$$\mathcal{L}_{\text{selective}} = \sum_{i=1}^{N} I(\text{prediction}_i \text{ is OK}) \cdot \text{CrossEntropyLoss}(\text{prediction}_i, \text{target}_i) \quad (3.1)$$

This approach addresses one problem, but it simultaneously creates another - a significant slowdown in the learning process. Specifically, applying the aforementioned method substantially limits the number of examples on which the model can calculate the loss function and perform backpropagation. Most predictions at the beginning of training will be "illegal" misses.

Classifier

To minimize the number of "illegal misses" and thereby accelerate the learning process, we decided to add an additional fully-connected layer to the model's architecture, serving as a binary classifier. This layer accepts values corresponding to tokens from the sequence, which are obtained from the last decoder layer's hidden state and batched. For each token representation, the classifier predicts either 0 or 1 (i.e., to which group the token belongs). To measure the classification accuracy, we use binary cross-entropy. The loss value calculated in this manner is as follows:

$$\mathcal{L}_{\text{classifier}} = \text{BinaryCrossEntropy}(\text{classifier\_prediction}, \text{target\_group}) \tag{3.2}$$

It's important to note that if the model accurately predicts the "generation mode" (i.e., control or semantic) during training, the classifier's loss value will be zero. In such scenarios, we'd expect the primary prediction to be accurate or result in a "legal" miss. This implies that the classifier's influential role is primarily limited to cases of "illegal misses", where it compensates for the reduction in learning speed caused by ignoring these instances in the loss function from the "Selective Learning" section.

Custom Loss in its Entirety

The fundamental loss function is expressed as the sum of the two aforementioned components, with the value derived from the classifier weighted by a constant α (in our case, equal to 0.2).

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{selective}} + \alpha \cdot \mathcal{L}_{\text{classifier}} \tag{3.3}$$

3.2.3 Segmentator

As previously discussed in the T5 (Text-to-Text Transfer Transformer) section, T5-like models are pretrained using the span corruption objective. This suggests that a similar strategy could be beneficial during fine-tuning. Building on this idea, we propose going a step further by leveraging the structural properties of code:

instead of masking arbitrary spans, we apply masks to syntactically coherent code segments. The implementation of this concept is the primary motivation behind the development of the segmentator tool.

To begin, we need to clarify what we mean by syntactically coherent code segments. These segments can be formally defined using specific tokens introduced by the pretokenizer. Since there are multiple ways to determine which tokens should define a segment, the segmentator is designed to be generic and flexible. The span extraction method is fully customizable by enabling or disabling a set of Boolean flags.

For example, setting the control tags flag to True selects spans corresponding to multi-line code blocks introduced by control structures such as `[IF]`, `[FOR]`, `[DEF]`, `[CLASS]`, and others. Similarly, enabling the `inline_tags` flag generates spans based on one-line constructs marked by tags like `[RAISE]`, `[RETURN]`, or `[DEL]`. Additional options allow segmentation based on delimiters, line breaks, indentation levels, or semantically cohesive units. It's also possible to provide an arbitrary set of tags to define custom segmentation behavior.

This setup defines the purpose of the initial segmentation step. As a result, we obtain a set of spans represented as tuples in the form `(start_index, end_index)`. At this stage, it's important to note that some spans may overlap or be nested within larger ones. These characteristics interfere with proper span masking, so the output must be further processed.

At this step, we randomly sample a subset of non-overlapping spans. The sampling process is controlled by parameters such as the minimum and maximum proportion of tokens to be masked, as well as the maximum number of spans to select.

Finally, we format the input by substituting each selected span with a placeholder token of the form `<extra_id_i>`, which the model is trained to reconstruct. The corresponding labels are constructed as a sequence of these tokens followed by their associated masked content.

Input:
Subtracts two vectors.
```
[DEF] [SEMANTIC_START] sub [SEMANTIC_END] <extra_id_0>
[BLOCK] [INDENT] [RETURN] [SEMANTIC_START] tuple [SEMANTIC_END]
[DELIMIT_1_L] [DELIMIT_1_L] [SEMANTIC_START] a [SEMANTIC_END] [SUB]
[SEMANTIC_START] b [SEMANTIC_END] [FOR_COMP]
[SEMANTIC_START] a [SEMANTIC_END] [COMMA] [SEMANTIC_START] b [SEMANTIC_END]
[IN_COMP] [SEMANTIC_START] zip [SEMANTIC_END]
<extra_id_1> [DELIMIT_1_R] [DELIMIT_1_R] [DEDENT]
```
Labels:
```
<extra_id_0> [DELIMIT_1_L] [SEMANTIC_START] v1 [SEMANTIC_END] [COMMA]
```

```
[SEMANTIC_START]v2[SEMANTIC_END][DELIMIT_1_R]<extra_id_1>[DELIMIT_1_L]
[SEMANTIC_START]v1[SEMANTIC_END][COMMA][SEMANTIC_START]v2[SEMANTIC_END]
[DELIMIT_1_R]<extra_id_2>
```

Figure 3.2: Example of the input and labels after segmentation

3.3 Inference

3.3.1 Logits Processor

In the "Pretokenizer" section, we introduced the concepts of control and semantic tokens. During inference, these can be readily translated into generation modes: control mode and semantic mode. These are the only modes the model can operate in while generating code, and the logits processor enables controlled switching between them.

From a technical perspective, the Logits Processor is the final unit in our framework that modifies the scores calculated by the model for each token. This allows it to manipulate these values, thereby influencing the ultimate prediction outcome. We leveraged this capability to create an abstraction of generation modes.

The Logits Processor examines the last token in the sequence to determine the current generation mode. If the last token is a control token, the generation mode is in control mode. In this mode, the Logits Processor masks all semantic tokens. It operates analogously when the last token is a semantic token. Masking is achieved by changing the score value to $-\infty$, ensuring that after the softmax operation, the values corresponding to the masked tokens become 0, which means zero probability of selecting that token for generation.

The generation mode switches when the last token is `[SEMANTIC_START]` or `[SEMANTIC_STOP]`. Although `[SEMANTIC_START]` is a control token, the Logits Processor recognizes it and reverses the masking; that is, it masks as if the last token were a semantic token. The situation is analogous when the last token is `[SEMANTIC_STOP]`. This mechanism ensures that the next token will be from the opposite generation mode.

The application of such a Logits Processor is extremely important. Given that the scores are the result of a dot product between the last hidden representation of a token from the decoder and all possible tokens, it's possible for the highest scores to be for tokens from the opposite generation mode, even without a mode-switching token. This is entirely possible because during training, we do not concern ourselves with the relative arrangement of these subspaces. Hence, to ensure that the generated tokens are indeed closest to the last hidden representation in terms of dot product and learned relationships during training, the masking performed by the Logits Processor is necessary.

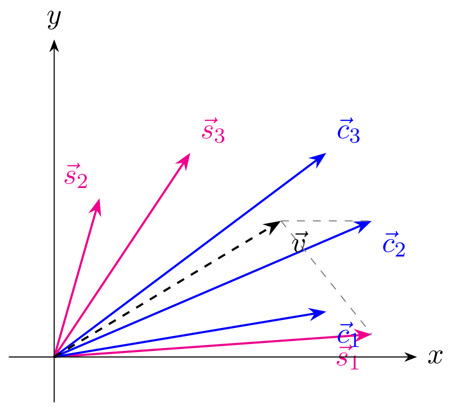

Figure 3.3: The illustration shows a situation where the dot product only makes sense within the current generation mode. In the figure, the generation mode is *semantic mode*, and the closest token in terms of dot product is a technical token that does not make sense during the generation of the semantic part of the code. Magenta vectors represent semantic tokens, while blue vectors represent control tokens. Vector $\vec{v}$ represents the prediction vector.

Chapter 4

Experiments

In this section, we test the methods introduced in our approach. In the following experiments, we begin with the original T5-Large model. Then we test out the finetuned one and later on add our improvements one by one, starting from pre-tokenizer, segmentator, and custom loss. That helps to divide the methods into those that, in fact, enhance the model performance and those that do not present a significant improvement. It is worth noting that in each experiment, we explore different combinations of hyperparameters; however, we report only the results obtained using the best-performing configuration.

4.1 T5

In the first experiment, we evaluate the performance of the T5-Large model. As previously mentioned in the T5 (Text-to-Text Transfer Transformer) section, there are several different versions of the T5 architecture. We chose the T5-Large model primarily due to hardware limitations, as it is the largest variant we can train on our current setup. Since T5 was pretrained on diverse natural language texts and not specifically designed for Python code generation, we anticipated suboptimal results—and this expectation was confirmed.

Most of the model's predictions are merely mixtures of words found in the input. In some cases, the output includes Python-like terms, but they do not form any coherent code structure and largely reflect the input content

As expected, the evaluation metrics are weak. Given their low values, we did not attempt to optimize the model using hyperparameter tuning. It is evident that the model requires significantly more robust modifications to generate meaningful Python code.

4.2 Finetuned T5

In this experiment, we follow a standard finetuning approach by directly providing input–output pairs—docstrings as inputs and corresponding reference code as outputs. At this stage, we do not incorporate any task-specific characteristics or domain knowledge. These will be introduced in subsequent experiments. Our goal here is to establish a baseline performance that can serve as a reference point for evaluating future improvements.

Due to hardware limitations, we impose certain constraints. Specifically, we use 20% of the dataset and limit the maximum number of tokens in the reference code to 512. Each example is seen by the model three times.

It is important to note that these settings may vary across experiments. While one might argue that this approach does not offer identical conditions for every model, there is a strong rationale behind it: different experiments operate on differently modified data. Enforcing exactly the same settings for all models would be inefficient and could suppress the unique strengths of each one. Therefore, we tailor the settings for each model individually to maximize its potential.

The results we obtain significantly outperform those of the standard T5-large model. For example, the BLEU score is more than ten times higher, while the ROUGE metrics show improvements ranging from two- to fivefold. Interestingly, the best performance is achieved without sampling, although the differences are quite subtle. BLEU scores typically range between 0.15 and 0.18, with ROUGE-L values hovering around 0.33 to 0.35.

Listing 4.1: Example of Finetuned T5 Input, Reference and Prediction

```python
# Input:
# Using the blob object, write the file to the destination path

# Reference:
def write_file(self, blob, dest):
    with salt.utils.files.fopen(dest, 'wb+') as fp_:
        blob.stream_data(fp_)

# Prediction:
def write_blob(blob, destination):
    with open(destination, 'wb') as f:
        f.write(blob.encode('utf-8'))
```

4.3 Finetuned T5 + Pretokenizer

In this experiment, we introduce the first custom method—a pretokenizer. The motivation behind this approach is to provide the model with additional information derived from the abstract syntax tree (AST), which we expect to improve its ability to generate syntactically accurate code.

However, this enhancement comes with a trade-off. Due to the constraint imposed by the maximum output length (512 tokens), we are unable to include some of the longer code functions. The inclusion of descriptive tokens from the AST increases sequence length, causing certain examples to exceed the allowed limit. As a result, the dataset used for training becomes both smaller and composed of shorter examples.

Nevertheless, the trade-off proves worthwhile. The best results are obtained using sampling with a low temperature of 0.1, a beam size of 4, and a length penalty of 1.3. BLEU scores show a clear improvement over the previous experiment, ranging from 0.19 to 0.21. Comparable gains are observed across the ROUGE-L metrics, with scores increasing to the 0.35–0.39 range. While most examples contain some flaws, we identified one case of exact correspondence between the prediction and the reference, presented below.

Listing 4.2: Example of Finetuned with Pretokenizer T5 Input, Reference and Prediction

```
# Input:
# Pass through to provider AssessmentSearchSession.
    get_assessments_by_search

# Reference:
def get_assessments_by_search(self, assessment_query,
    assessment_search):
     if not self._can("search"):
         raise PermissionDenied()
     return self._provider_session.get_assessments_by_search(
         assessment_query, assessment_search)

# Prediction:
def get_assessments_by_search(self, assessment_query,
    assessment_search):
     if not self._can("search"):
         raise PermissionDenied()
     return self._provider_session.get_assessments_by_search(
         assessment_query, assessment_search)
```

4.4 T5 + Pretokenizer + Segmentator

In this experiment, we introduce the final feature: the segmentator. This training method serves as an auxiliary objective, since the model still needs to learn to generate entire function bodies rather than just predicting masked spans. To balance these goals, we adopt the following strategy: each function undergoes one pass of training with the segmentator and then two additional passes of fine-tuning with the pretokenizer. This approach allows us to integrate an alternative learning signal without compromising the model's primary objective.

However, the results obtained with this method are slightly worse than those from the previous approach, although the difference is subtle. It is likely that a single fine-tuning run using the new method is insufficient for the model to effectively learn code in a different way. BLEU scores fall within the range of 0.18–0.20, while ROUGE-L scores range from 0.34 to 0.38. The best performance is achieved when using the same set of hyperparameters as in the previous experiment. Nonetheless, this experiment also includes a few examples that are exceptionally well predicted.

Listing 4.3: Example of Finetuned with Pretokenizer and Segmentator T5 Input, Reference and Prediction

```
# Input:
# Get list of telegram recipients of a hook
# ---
# serializer: TelegramRecipientSerializer
# responseMessages:
#     - code: 401
#       message: Not authenticated

# Reference:
def get(self, request, bot_id, id, format = None):
    return super(TelegramRecipientList, self).get(request,
        bot_id, id, format)

# Prediction:
def telegram_recipients(self, request, bot_id, id, format =
   None):
    return super(TelegramRecipientList, self).get(request,
        bot_id, id, format)
```

Metric	T5 Baseline	Finetuned T5	Pretokenizer T5	Segmentator T5
BLEU	0.0173	0.1782	0.2097	0.2017
Precision@1	0.1505	0.4992	0.5321	0.5171
Precision@2	0.0207	0.2390	0.2706	0.2552
Precision@3	0.0085	0.1413	0.1647	0.1517
Precision@4	0.0034	0.0894	0.1092	0.0978
Brevity Penalty	1.0000	0.9045	0.9298	0.9586
Translation Length	10288	43550	40703	41895
Reference Length	9001	47919	43666	43666
Length Ratio	1.1430	0.9088	0.9321	0.9594

Table 4.1: BLEU Metrics Across Experiments

Metric	T5 Baseline	Finetuned T5	Pretokenizer T5	Segmentator T5
ROUGE-1	0.1536	0.3913	0.4077	0.4000
ROUGE-2	0.0330	0.1499	0.1600	0.1494
ROUGE-L	0.1402	0.3570	0.3774	0.3684

Table 4.2: ROUGE Metrics Across Experiments

4.5 T5 + Pretokenizer + LogitsProcessor with Custom Loss

Unfortunately, the experiment involving the LogitsProcessor and Custom Loss produced very poor results that offer no practical value. The generated outputs consisted primarily of repeated tokens, lacking both semantic meaning and any clear connection to the input. While the exact cause of this behavior remains uncertain, we propose two main hypotheses that may explain the failure.

4.5.1 Mismatch Between Training and Inference Behavior

During inference, a custom LogitsProcessor strictly controls which group of tokens the model is allowed to generate either control tokens or semantic tokens, depending on the current generation mode. Switching between these modes requires the model to output special tokens like `[SEMANTIC_START]` and `[SEMANTIC_STOP]`. However, if the model never properly learns to generate these switching tokens during training perhaps because the loss function ignores too many of its mistakes or the training signal is too weak it may get "stuck" in one mode. This means that one group of tokens becomes completely blocked during inference, making it impossible to produce valid code. As a result, the model might generate broken or meaningless sequences.

4.5.2 Conflict with T5 Pretraining

The T5 model was pretrained on natural language tasks, such as translation or summarization, which differ significantly from code generation. In our setup, the model was fine-tuned directly on a structurally different task, using code as output and docstrings as input. This sudden shift in objective—combined with the introduction of a new token structure may have overwhelmed the model's pretrained representations. Without a gradual transition phase (e.g., intermediate tasks or curriculum learning), the model may have struggled to adapt, leading to unstable training and poor generalization. As a result, it likely failed to effectively leverage its pretrained language understanding for the code generation task.

4.6 Performance of Our Best Model on the HumanEval

As the final experiment, we evaluate how our best-performing model — the T5 fine-tuned with Pretokenizer, using sampling with a temperature of 0.1, a beam size of 4, and a length penalty of 1.3 — performs on the most challenging benchmark: HumanEval. It is worth noting that even the most advanced code generation models on the market have historically struggled to pass a majority of these tests. This is primarily because the problems span a wide range of programming challenges and are not drawn from the datasets used during training. Given the difficulty of the task, we initially suspected that our model might not pass any tests at all. Surprisingly, however, it successfully solved one task.

This specific task required implementing a function that concatenates a list of strings into a single string. While the expected solution involved a straightforward use of Python's `join` function, our model produced a more convoluted version. Instead of directly passing the list of strings to `join`, it constructed a list comprehension with a condition that always evaluates to true, effectively returning the original list. Although unnecessarily complex, the solution was nonetheless correct.

This outcome is encouraging and suggests that our model demonstrates a fundamental understanding of code semantics. It offers hope that, when scaled appropriately, the model has the potential to generate functionally correct Python code more reliably.

Listing 4.4: Example of a correct solution generated by the model for HumanEval/28

```python
# Task:
# def concatenate(strings):
#     """
#     Concatenate list of strings into a single string
#     >>> concatenate([])
#     ''
#     >>> concatenate(['a', 'b', 'c'])
#     'abc'
#     """
def concatenate(strings):
    return "".join([string for string in strings if string.
        endswith("")])
```

Chapter 5

Code Organization

5.1 Repository

The structure of the project was designed to facilitate the modification of its individual components in order to conduct alternative experiments. The project is divided into three main folders:

- **core**: contains abstract classes and interfaces,
- **data_processing**: contains implementations responsible for all kinds of data preprocessing,
- **model_operations**: contains implementations of methods described in *Our Approach* section as well as scripts for training and evaluation.

An important file in the root directory of the project is **config.py**, where parameters related to data processing, training, and evaluation are defined.

core

This folder contains abstract classes for the pretokenizer, data preprocessor, and segmentator, as well as an interface that — when implemented by the pretokenizer class — ensures compatibility with the segmentator. It guarantees that the pretokenizer provides sufficient information for the segmentator to correctly divide the data.

data_preprocessing

This folder contains the implementations of the pretokenizer, segmentator, and data preprocessor classes, each of which extends the appropriate abstract class from the `core` folder.

The pretokenizer is actually a modification and extension of the `_Unparser` class from Python's standard `ast` module.

The segmentator serves as a wrapper around methods responsible for segmenting the data.

The data preprocessor is designed as a skeleton that accepts individual instances of data operations, which are then composed into a pipeline executed sequentially. This allows for flexible configuration of what modifications are used, or for implementing custom ones.

These classes are built to allow seamless data manipulation — they are designed to accept data as input and return it after transformations.

model_operations

This folder contains scripts intended to be run directly, i.e., training and evaluation scripts. The `training_additions.py` file contains implementations of components such as a custom loss function and a logits processor.

5.2 Dependencies

The implementation relies on `Python 3.13.3` and several external Python libraries, including: `numpy`, `pandas`, `autopep8`, `tqdm`, `wandb`, `evaluate`, `torch`, `datasets`, `transformers`, `pynvml`, `psutil` and `human_eval`. They can be found with specific versions in `requirements.txt` file.

Chapter 6

Further Investigation

In many generated outputs, we observed repetitive patterns where the model cycles through a small set of tokens. This suggests that adding a repetition penalty during decoding might improve output quality. However, in source code, some tokens naturally appear much more frequently than others (e.g., colons, parentheses, indentation), so applying such a penalty blindly could hurt valid generations. This would require code-aware heuristics.

Listing 6.1: Example of Finetuned with Pretokenizer and Segmentator T5 Input, Reference and Prediction Showing Severe Repetition in Output

```
# Input:
# Wraps ‘os.environ‘ to filter out non-encodable values.

# Reference:
def _environment_variables():
    return {key: value for key, value in os.environ.items() if
        _is_encodable(value)}

# Prediction:
def filter_environ(self, os):
    os.environ = os.environ
    os.environ = os.environ
    os.environ = os.environ
    os.environ = os.environ
    return os.environ
```

Another possible improvement involves better training for the special mode-switching tokens (`[SEMANTIC_START]` and `[SEMANTIC_STOP]`). In the current setup, these tokens are treated like any other, with no extra emphasis during training. Only we, as designers, knew their importance—while the model did not receive any additional signal to learn their role. A dedicated pretraining phase could help the model focus on learning when and how to switch between generation modes.

Additionally, a curriculum learning approach might help introduce and learn new toknes. For example, we could start with simpler tasks—such as generating only control tokens or short code snippets—and gradually move to full function generation. This would allow the model to build useful patterns incrementally, instead of learning everything at once.

Appendix A

New tokens

A.1 Control tokens

```
{   '[ADD]': ' + ',
    '[ADD_ASSIGN]': ' += ',
    '[AND]': ' and ',
    '[ASSERT]': 'assert ',
    '[ASSIGN]': ' = ',
    '[ASYNC_DEF]': 'async def ',
    '[ASYNC_FOR]': 'async for ',
    '[ASYNC_FOR_COMP]': ' async for ',
    '[ASYNC_WITH]': 'async with ',
    '[AS_ITEM]': ' as ',
    '[AWAIT]': 'await ',
    '[BITAND]': ' & ',
    '[BITAND_ASSIGN]': ' &= ',
    '[BITOR]': ' | ',
    '[BITOR_ASSIGN]': ' |= ',
    '[BITXOR]': ' ^ ',
    '[BITXOR_ASSIGN]': ' ^= ',
    '[BLOCK]': ':',
    '[BREAK]': 'break',
    '[CASE]': 'case ',
    '[CLASS]': 'class ',
    '[COMMA]': ', ',
    '[CONTINUE]': 'continue',
    '[DEDENT]': '',
    '[DEF]': 'def ',
    '[DELIMIT_1_L]': '(',
    '[DELIMIT_1_R]': ')',
    '[DELIMIT_2_L]': '[',
    '[DELIMIT_2_R]': ']',
    '[DELIMIT_3_L]': '{',
    '[DELIMIT_3_R]': '}',
    '[DEL]': 'del ',
    '[DICT_COLON]': ': ',
    '[DIV]': ' / ',
    '[DIV_ASSIGN]': ' /= ',
    '[DOT]': '.',
    '[ELIF]': 'elif ',
    '[ELLIPSIS]': '...',
    '[ELSE]': 'else',
    '[ELSE_COMP]': ' else ',
    '[EQ]': ' == ',
    '[EXCEPT*]': 'except* ',
    '[EXCEPT]': 'except ',
    '[FINALLY]': 'finally',
    '[FLOORDIV]': ' // ',
    '[FLOORDIV_ASSIGN]': ' //= ',
    '[FOR]': 'for ',
    '[FOR_COMP]': ' for ',
    '[FROM]': ' from ',
    '[FROM_IMPORT]': 'from ',
    '[F]': 'f',
    '[GT]': ' > ',
    '[GT_E]': ' >= ',
    '[GUARD]': ' if ',
'[IF]': 'if ',
'[IF_COMP]': ' if ',
'[IMPORT]': 'import ',
'[IMPORT_FROM]': ' import ',
'[INDENT]': '    ',
'[INF]': 'inf',
'[INVERT]': '~',
'[IN]': ' in ',
'[IN_COMP]': ' in ',
'[IS]': ' is ',
'[IS_NOT]': ' is not ',
'[KWARGS]': '**',
'[LAMBDA]': 'lambda ',
'[LAMBDA_BODDY]': ': ',
'[LSHIFT]': ' << ',
'[LSHIFT_ASSIGN]': ' <<= ',
'[LT]': ' < ',
'[LT_E]': ' <= ',
'[MATCH]': 'match ',
'[MATCH_AS]': ' as ',
'[MATCH_DEFAULT]': '_',
'[MATCH_GUARD]': ' if ',
'[MATCH_OR]': ' | ',
'[MATCH_STAR]': '*',
'[MATMULT]': ' @ ',
'[MATMULT_ASSIGN]': ' @= ',
'[MOD]': ' % ',
'[MOD_ASSIGN]': ' %= ',
'[MULT]': ' * ',
'[MULT_ASSIGN]': ' *= ',
'[NAMED_EXPR]': ' := ',
'[NAN]': 'nan',
'[NEW_LINE]': '\n',
'[NOT]': 'not ',
'[NOT_EQ]': ' != ',
'[NOT_IN]': ' not in ',
'[OR]': ' or ',
'[PASS]': 'pass',
'[POSONLYARGS]': '/',
'[POW]': '**',
'[POW_ASSIGN]': ' **= ',
'[QUOT_1]': "'",
'[QUOT_2]': '"',
'[RAISE]': 'raise ',
'[REFERENCE]': '*',
'[RETURN]': 'return ',
'[RSHIFT]': ' >> ',
'[RSHIFT_ASSIGN]': ' >>= ',
'[SEMANTIC_END]': '',
'[SEMANTIC_START]': '',
'[SLICE]': ':',
'[SUB]': ' - ',
'[SUB_ASSIGN]': ' -= ',
'[TRY]': 'try',
'[TUPLE_L]': '(',
'[TUPLE_R]': ')',
'[UADD]': '+',
'[UNPACK]': '**',
'[USUB]': '-',
'[U]': 'u',
'[WHILE]': 'while ',
'[WITH]': 'with ',
'[YIELD]': 'yield ',
'[YIELD_FROM]': 'yield from '}
```

Additionally `<custom_eos>` (EOS token) and `[SEMANTIC_START]`.

A.2 Additional semantic tokens

`[SEMANTIC_STOP]`

Appendix B

Work Contribution Summary

Section	**Mateusz Kitzol**	**Patryk Rybak**
Technical Research	★☆☆	★★☆
Code Organization	★☆☆	★★☆
Dataset Preparation	★★☆	★☆☆
Pretokenizer Strategy	☆☆☆	★★★
Segmentator Strategy	★★☆	★☆☆
Custom Loss Strategy	☆☆☆	★★★
Logits Processor Strategy	☆☆☆	★★★
Training & Generation	★★☆	★☆☆
Evaluation & Metrics	★★★	☆☆☆
Hyperparameter Tuning	★★★	☆☆☆

Source code can be found in our repository.

Bibliography

[1] 10.1. Long Short-Term Memory (LSTM) &x2014; Dive into Deep Learning 1.0.3 documentation — d2l.ai. `https://d2l.ai/chapter_recurrent-modern/lstm.html#fig-lstm-0`. [Accessed 16-06-2025].

[2] 9. Recurrent Neural Networks &x2014; Dive into Deep Learning 1.0.3 documentation — d2l.ai. `https://d2l.ai/chapter_recurrent-neural-networks/index.html`. [Accessed 08-06-2025].

[3] c4 — TensorFlow Datasets — tensorflow.org. `https://www.tensorflow.org/datasets/catalog/c4`. [Accessed 04-06-2025].

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[6] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann LeCun. Very deep convolutional networks for natural language processing. *CoRR*, abs/1606.01781, 2016.

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[8] Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. Ast-t5: Structure-aware pretraining for code generation and understanding, 2024.

[9] Rie Johnson and Tong Zhang. Deep pyramid convolutional neural networks for text categorization. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 562–570, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[10] Todd Ward Kishore Papineni, Salim Roukos and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 311–318. Association for Computational Linguistics, 2002.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[12] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461, 2019.

[13] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish

Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[14] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.

[15] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[16] openai. gpt-2. `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`. [Accessed 07-06-2025].

[17] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[18] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *CoRR*, abs/2109.00859, 2021.

[19] Wikipedia. Backpropagation — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Backpropagation&oldid=1292918637`, 2025. [Online; accessed 03-June-2025].

[20] Wikipedia. Perceptron — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Perceptron&oldid=1291484724`, 2025. [Online; accessed 04-June-2025].

[21] Wikipedia. Transformer (deep learning architecture) — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Transformer%20(deep%20learning%20architecture)&oldid=1292887874`, 2025. [Online; accessed 04-June-2025].