University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master's thesis

Source Code Generation from Descriptions in a Natural Language

Pilsen 2022 Bc. Jan Pašek

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Jan PAŠEK**
Osobní číslo:	**A20N0098P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Generování zdrojových kódů na základě popisu v přirozeném jazyce**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Prozkoumejte stávající metody generování zdrojového kódu algoritmy strojového učení.
2. Na základě studia existujících metod zvolte či navrhněte vhodnou techniku pro generování kódu na základě popisu v přirozeném jazyce. Při volbě techniky se zaměřte na využití moderních před-trénovaných generativních modelů neuronových sítí.
3. Vyberte a získejte datovou sadu zdrojových kódů, která svou velikostí postačuje k předtrénování hluboké neuronové sítě. Pomocí těchto dat předtrénujte vybraný generativní model zdrojového kódu na dostatečně výkonném uzlu gridové infrastruktury MetaCentrum.
4. Implementujte algoritmus pro generování zdrojového kódu z popisu problému v anglickém jazyce. Vyberte a získejte dodatečná data obsahující útržky kódu s odpovídajícím popisem. Získaná data použijte pro dotrénování navržené architektury neuronové sítě.
5. Porovnejte své řešení s již existujícími metodami a proveďte kritické zhodnocení dosažených výsledků.

Rozsah diplomové práce:	**doporuč. 50 s. původního textu**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování diplomové práce:	**tištěná/elektronická**
Jazyk zpracování:	**Angličtina**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce:	**Ing. Miloslav Konopík, Ph.D.** Katedra informatiky a výpočetní techniky
Datum zadání diplomové práce:	**10. září 2021**
Termín odevzdání diplomové práce:	**19. května 2022**

L.S.

Doc. Ing. Miloš Železný, Ph.D. děkan	**Doc. Ing. Přemysl Brada, MSc., Ph.D.** vedoucí katedry

V Plzni dne 11. října 2021

Declaration

I hereby declare that this master's thesis is completely my own work and that I used only the cited sources.

Pilsen, 15th May 2022

Bc. Jan Pašek

Acknowledgement

I want to thank Ing. Miloslav Konopík, Ph.D. for valuable advice, factual comments, and helpfulness in the consultation and elaboration of the master's thesis.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA CZ LM2018140) supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Abstract

This work introduces CodeFormer, a Python source code generator pre-trained on a massive GitHub crawl consisting of 230M Python functions. The released model, built on BART architecture, generates Python functions based on descriptions in English. On a CodeSearchNet dataset, the CodeFormer sets a new state of the art with 46.12 BLEU, representing an improvement of 13.86 BLEU. We also release a new parallel corpus for code generation called Stack Overflow Code Generation Dataset (SOCGD), on which our model sets a baseline of 47.68 BLEU. The resulting model is ready to be integrated into a source code suggestion system in an IDE, where it can improve software developers' productivity. During our research, we discovered a better way of training the BART for machine translation. However, the applicability of our approach to other domains must be verified in subsequent work.

Abstrakt

Tato diplomová práce představuje CodeFormer, nový model neuronové sítě, schopný na základě popisu úlohy v anglickém jazyce generovat funkce v programovacím jazyce Python. Tento model, založený na architektuře modelu BART, je předtrénovaný na 230 milionech funkcích získaných z veřejných GitHub repozitářů. Po dotrénování na CodeSearchNet datasetu náš model překonává konkurenční modely a nastavuje tak nové state of the art s 46,12 BLEU, což představuje zlepšení o 13,86 BLEU. Vedle CodeFormer modelu tato práce představuje nový Stack Overflow Code Generation Dataset (SOCGD), který je určený k trénování generativních modelů zdrojových kódů. Na tomto datasetu náš model dosahuje výsledku 47,68 BLEU. Výsledný model lze integrovat do vývojových prostředí a umožnit tak programátorům generovat části zdrojových kódů s cílem zvýšit efektivitu jejich práce. V rámci našeho výzkumu jsme také objevili lepší přístup k trénování modelu BART na úloze strojového překladu. Použitelnost tohoto přístupu na jiných doménách je třeba ověřit v navazující práci.

Contents

1 Introduction

In recent years, modern natural language processing methods have achieved excellent results in many different tasks and domains. One of the domains that have not escaped the attention of researchers is software engineering. Given that software development is a demanding discipline requiring a large amount of knowledge and is prone to human error, it is appropriate to develop automated tools to make software engineers' day-to-day work more manageable.

One way to make software developers more efficient is to create a system that generates a piece of code according to a prompt written in natural language. Such a system can be implemented directly in the integrated development environment (IDE), allowing the developers to effectively find solutions to their problems without visiting one of the well-known question and answer websites, such as Stack Overflow.

This work aims to create and train a modern generative model of a neural network, which can generate a suitable source code in Python according to a description in English. Although similar models already exist, they are often based on older text generation methods or are provided for a fee. The main benefits of this work are as follows: 1) We publish a fine-tuned CodeFormer model that generates source code from an English description and is ready for integration into IDEs. 2) We publish a new corpus of source codes in the Python language, containing about 230M training examples and thus allowing easy follow-up on our work. 3) We publish a new Stack Overflow Code Generation Dataset (SOCGD) for training Python source code generators.

This master's thesis is structured as follows. The first two chapters present a theoretical background of generative neural networks and their applications in source code processing. Subsequently, the analytical chapter discusses the various aspects of the problem and outlines our solution's direction. The last two chapters are then devoted to realizing the proposed solution. Namely, they focus on pre-training the chosen neural network model and its subsequent use for code generation.

2 Generative Models

The field of natural language processing (NLP) is extensively wide and encompasses dozens of different tasks with various real-world usages. A significant portion of the NLP tasks, to some extent, involve a kind of text generation. An example of such a task that naturally requires generating text is a machine translation, which aims to generate a text in a target language given a text in a source language. Since this work aims to build a source code generation system based on neural networks (NN) [1], the problem of *generative models* is very important for the rest of this thesis.

This chapter is structured as follows. Firstly, we present the *generative models* in the context of natural language processing, including their definition and possible applications. Subsequently, we provide architectural examples of neural network models classified as *generative*. Following that point, we also discuss technical topics related to training, evaluation, and usage of *generative models* such as text tokenization, decoding algorithms, and evaluation metrics.

2.1 Introduction to Generative Models

In statistics, we distinguish two elementary types of statistical models. The first class is represented by *discriminative models*, whereas the others are *generative models*. All of these models are extensively applied to different NLP tasks, but as mentioned previously, the *generative* class of models is the important one for our work. Therefore, in the rest of this section, we briefly introduce the generative models, including their definition, comparison to discriminative models, and possible applications.

2.1.1 Definition

A statistical model of a joint probability $P(X, Y)$, where X is an observation from a space of inputs and Y is a corresponding label, is called a **generative model** [2]. If working with unlabeled data, the *generative model* captures just $P(X)$. Intuitively, a *generative model* captures how likely an observation appears in the input space, which can be utilized to generate new examples by sampling from the modeled distribution [3].

Generative vs. Discriminative

When defining *generative models*, we should also compare them to *discriminative models* whose aim is entirely different. Instead of capturing the $P(X, Y)$, the *discriminative models* learn a conditional probability $P(Y|X = x)$. In other words, they model a probability of an observation x from an input space X having the label Y [3].

Put differently, the generative models can generate new data since they focus on data distribution. On the other hand, the discriminative models can classify data since they focus on a decision boundary. This difference is marked out in figure 2.1.

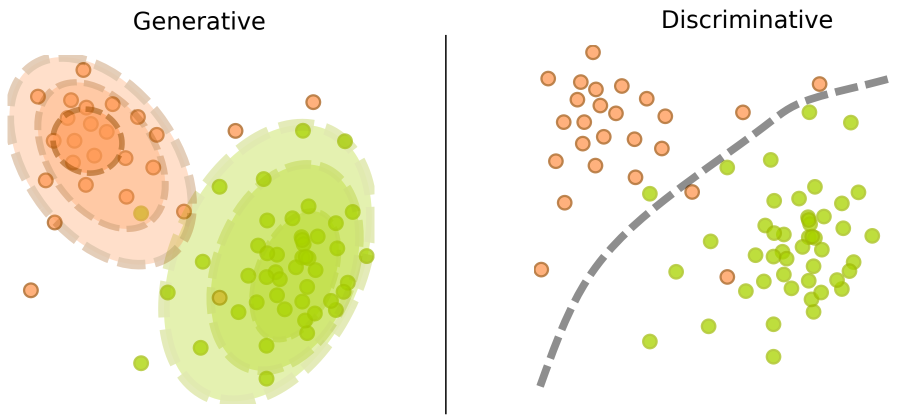


Figure 2.1: Demonstration of a difference between generative and discriminative models.

2.1.2 Language Modeling

A **language model** (LM) is a mathematical model of a conditional probability of a next word given all previous words (equation 2.1) [4]. Thanks to that, one can also compute a probability of a whole text $P(X)$ using the formula 2.2. Since equation 2.2 meets the definition of a generative model, we can say that the language models belong to the group of generative models.

$$P(x_i|x_1, x_2, ..., x_{i-1}) \tag{2.1}$$

$$P(X) = P(x_1, x_2, ..., x_n) = \prod_{i=1}^{N} P(x_i|x_1, x_2, ..., x_{i-1}) \tag{2.2}$$

Unlike the simple LM introduced previously, a *conditional LM* captures the probability of a sentence given some condition - $P(X, Y)$. For example, machine translation can employ a *conditional LM* to model a probability of a sentence in a target language given a sentence in a source language. Both of these LM variants represent a joint base for most of the tasks that involve some form of text generation. The applications of language modeling and text generation are further discussed in section 2.1.3.

Simple RNN Language Model

In the paragraphs above, we state the definition of an LM that represents a general framework rather than a particular realization. Therefore, we demonstrate how a language model can be built and trained with a **recurrent neural network** (RNN) [5]. RNNs are special neural networks designed to work with variable-length sequences [1]. A structure of a language model based on an RNN is depicted in figure 2.2.

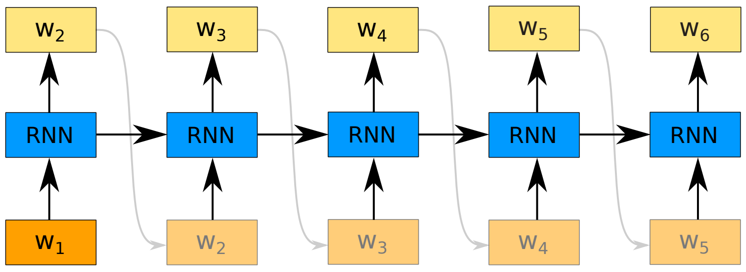


Figure 2.2: Visualization of a text generation using an RNN language model.

The figure shows how a language model accepts a token and produces a probability distribution of a subsequent token at each timestep. The information about all the preceding tokens is carried through the computation using a *hidden state* of the RNN. If we want to use an LM to generate text, we can greedily choose the most probable token at each timestep and use it as a subsequent input, as suggested in figure 2.2. More advanced decoding algorithms are then discussed in section 2.4.

However, following any decoding algorithm would not yield satisfying results during training. Especially in the early stages of the training, the output generated by the LM can be extremely noisy. Therefore, the inputs in later timesteps may not fit into the context. To mitigate this problem, we can employ a *teacher forcing* technique [6] that constantly feeds in target tokens instead of the generated output (figure 2.3).

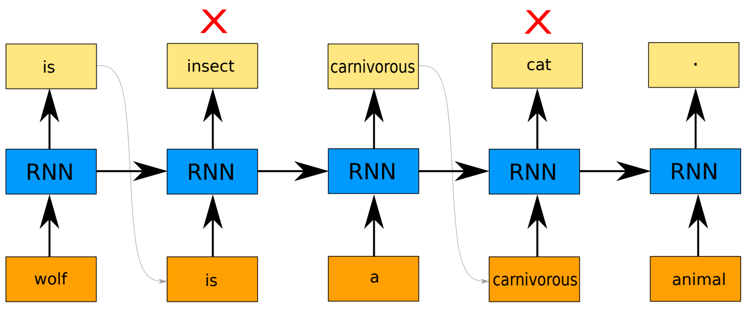


Figure 2.3: Demonstration of a teacher forcing method for training a language model.

2.1.3 Applications of Generative Models

Generally speaking, the generative models have a massive amount of possible usages, such as speech or image generation. However, in our work, we are primarily interested in generating text, and therefore, we discuss only the possible applications of *language models*, representatives of the group of generative models.

As stated in the previous section, language modeling forms a joint base for many tasks that involve some form of text generation. A simple example of LM usage is an auto-completion function that predicts the rest of the user's query (for example, at `https://www.google.com`). Implementing such a feature requires only a plain LM without any specific modification.

Besides this simple example, the generative models are essential for more complicated tasks such as *Question Answering*, *Summarization*, *Dialogue Systems*, and *Machine Translation* (MT). In all of the tasks above, we utilize a conditional LM. For example, for the MT, we model the probability $P(X|Y)$, where X is a sentence in a target language, and Y is a sentence in a source language.

To demonstrate how good the *generative models* are when generating text, we provide a short paragraph about car sharing, generated using GPT-2 model (section 2.2.3) [7]. Firstly, we provide the model with a short textual prompt (highlighted in bold) to direct it to the intended topic. We then let the model generate the rest of the text. The example[1] can be found below.

[1]The example was generated using `https://transformer.huggingface.co/doc/gpt2-large`.

An example of an AI-generated text

The future of car sharing is looking bright with an increase in car sharing startups and a growing number of new apps like Zipcar. This year, there were 14 new car sharing apps launched and the number is increasing rapidly, which is definitely helping the industry grow. As you can see, car sharing is definitely one of the hottest trends on the scene.

When reading the example, it can be noticed that the *GPT-2* mentions an application called *Zipcar*. Surprisingly, the *Zipcar* application exists (`https://www.zipcar.com`), indicating that the model either learned an extremely complex knowledge about the real world or memorized the training data. In either case, this example shows how complex knowledge can be stored in a generative NN.

2.2 Model Architectures

In the previous section, we have introduced the family of generative models. Furthermore, we discussed the *language models* that represent their subgroup. We also indicated that for some tasks, such as machine translation, we need to use *conditional language models*. Since generating source codes based on natural language descriptions can be perceived as a translation from English to Python, the conditional language models are essential for this work.

Therefore, in this section, we first present a *Seq2Seq* architecture, a general framework for implementing *conditional language models*. Afterward, we present some of the more advanced neural network models that follow the *Seq2Seq* architecture and are applicable for generative tasks.

2.2.1 Seq2Seq

A **Seq2Seq** architecture is utilized in many tasks where it is required to map one sequence to another [8]. As previously mentioned, an example of such a task is an MT or summarization. The core idea of this architecture is to use the first neural network to represent an input sequence using a tensor. The tensor is subsequently used by a second neural network that generates an output sequence. The first mentioned NN is called an *encoder*, and the second is a *decoder*. A general *Seq2Seq* architecture is depicted in figure 2.4.

The architecture of a *Seq2Seq* model is general and does not prescribe any specific realization. An implementation of a *Seq2Seq* model can build both the encoder and decoder, for example, using a vanilla RNN [5] or its improved variants such as *long short-term memory* (LSTM) [9] or *gated recurrent unit* (GRU) [10]. Furthermore, the encoder can utilize any neural network model dealing with variable-length sequences. This enables the usage of convolution neural networks (CNN) [11]. However, in the rest of this section, we focus on more advanced architectures discussed in the subsequent sections.

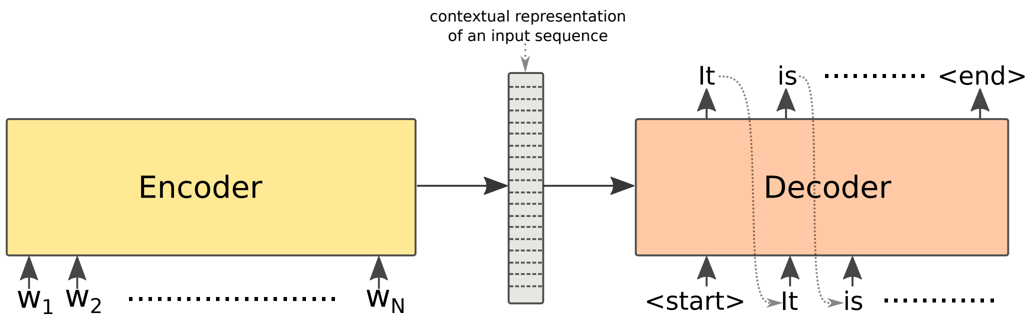


Figure 2.4: Architecture of a Seq2Seq model.

2.2.2 Transformer

A **Transformer** is an encoder-decoder model based exclusively on an *attention* mechanism [12]. After being published in 2017, it revolutionized the NLP field by serving as a joint base for numerous improved variants of the original architecture. A visualization of the whole architecture can be found in figure 2.5.

As indicated before, the model consists of two parts - an encoder and a decoder. The aim of the encoder (on the left-hand side of the figure) is to process an input sequence and produce its contextual representation [13]. Afterward, the contextual encoding is utilized by a decoder (on the right-hand side of the figure) that generates an output. The interaction between the encoder and decoder is realized using the *attention* mechanism, which is described in the following paragraphs alongside descriptions of the rest of the *Transformer*'s building blocks.

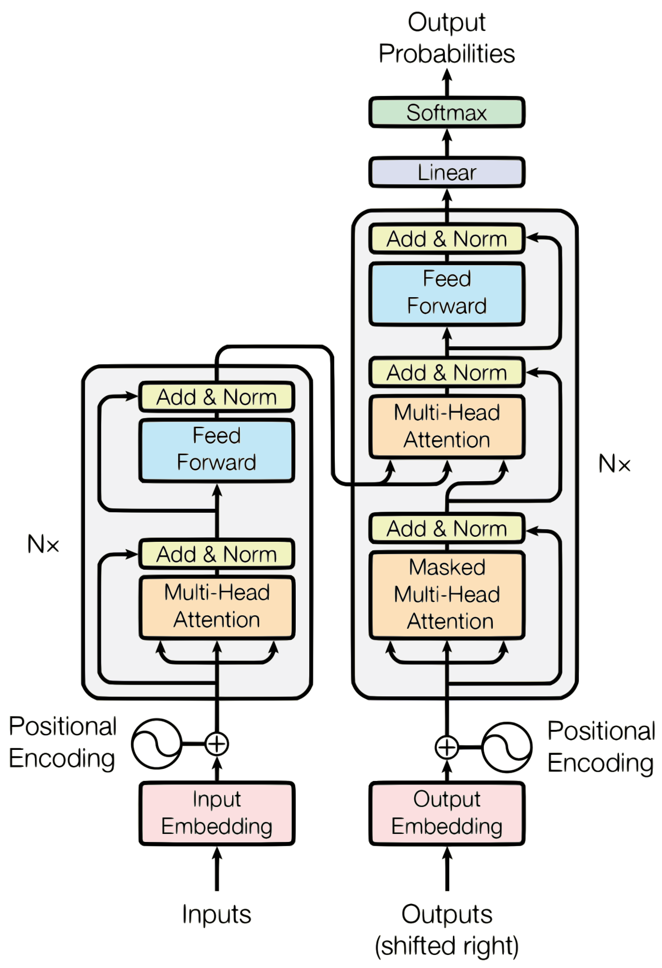


Figure 2.5: A visualization of a *Transformer*'s architecture.
Image source: [12].

Multi-Head Attention

Generally speaking, an **attention** mechanism is a way of expressing a measure of relevance. It can measure, for example, the relevance of different parts of an input to the task being solved. Alternatively, in the *Seq2Seq* architectures, it can express how relevant each part of an input is to each part of an output. There are numerous possibilities for implementing the *attention* [14]. However, we focus on *scaled dot-product attention* computed using equation 2.3 [12].

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.3}$$

As one can see, the *attention* function used in the *Transformer* has three inputs. These are keys, queries ($K, Q \in \mathbb{R}^{d_k \times N}$), and values ($V \in \mathbb{R}^{d_v \times M}$). The dot-product between keys (K) and queries (Q) is used to compute the relevance of keys (K) to the individual queries (Q). The result of the dot-product is then scaled by the square of the keys' dimension (d_k) and transformed using a softmax function. The resulting attention scores (organized into a matrix) are used to compute the weighted sum of values (V) for each element in the query.

Although this mechanism is very efficient and helpful in its simple form, the *Transformer* goes beyond and introduces *multi-head attention*. The idea behind the *multi-head attention* is to compute the previously introduced *attention* multiple times with keys, queries, and values transformed by different *linear* layers [12]. Each calculation of the *attention* is done by an *attention head*. Finally, the outputs of all the *attention heads* are concatenated together and transformed using another *linear* layer. The whole mechanism is depicted in figure 2.6.

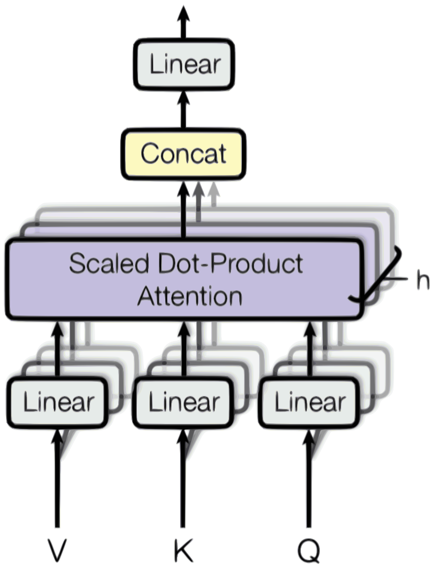


Figure 2.6: Visualization of a multi-head attention mechanism. Image source: [12].

It is also worth mentioning that the *Transformer* uses two different types of *attention*. The first type employed is called an *intra-attention* (or *self-attention*). It relates different parts of input to other tokens in the same sequence, producing context-aware representations of the tokens. The *intra-attention* is used in the encoder and the decoder, where it is applied immediately after the positional encoding. The second attention layer in the decoder utilizes an *inter-attention* (also called a *cross-attention*) that relates different parts of the sequence processed by the decoder to different tokens processed by the encoder. For example, the *cross-attention* in a machine translation context expresses which parts of the source sentence are relevant to different parts of the translation.

Other Building Blocks of the Transformer

Starting at the bottom of figure 2.5, the first employed layer in the *Transformer* is an *embedding layer*. The *embedding layer* assigns a trainable vector to every token in the vocabulary, transforming the 1-D input sequence $x_i \in \mathbb{R}^N$ into a matrix $x_e \in \mathbb{R}^{N \times E}$, where N represents the length of the input and E denotes the dimensionality of the embeddings. The *embedding layer* is followed by adding a *positional encoding* to preserve positional information otherwise brushed off by the attention mechanism, which is non-positional [12].

Besides the *multi-head attention*, the *Transformer* encoder and decoder blocks employ a *position-wise feed-forward network*, defined by equation 2.4 [12]. Last but not least, the *Transformer* employs an *add & norm* layer. It firstly applies a *residual connection* [15] around the *attention layer*, improving information and gradient flow through the model. Afterward, the layer applies *layer normalization* [16], an important regularization technique.

$$FFN(x) = max(0, xW_1, +b_1)W_2 + b_2 \tag{2.4}$$

2.2.3 GPT

A *Generative Pre-trained Transformer* (**GPT**) is a family of large neural network models whose architecture is based mainly on the *Transformer*'s decoder stack. The first published version of the model, called *GPT-1* [17], utilizes 12 layers of the decoder, resulting in 117M trainable parameters. Using a language modeling task, the authors first train the model on a large corpus of unlabeled text. Afterward, they fine-tune the pre-trained model on a final supervised task where the model can leverage the knowledge gained during the pre-training.

After establishing a few new state-of-the-art results on various tasks, the authors publish a new *GPT-2* [7] model with ten times more parameters than the first version. More specifically, the *GPT-2* model has 1.5B parameters and has a decoder of 48 layers. Their paper also shows that such a large model can solve various tasks without any specific fine-tuning, which is called *zero-shot learning*. In the *zero-shot* setup, it is sufficient only to specify the desired objective in an input. For example, for French to English translation, one can write the following prompt and let the *GPT-2* generate the rest [7]:

```
"As-tu aller au cinema?", translated to English:
```

The ability to solve new tasks with no examples provided (*zero-shot learning*) or with only few examples present in the input (*few-shot learning*) is further amplified by *GPT-3* model [18]. With 175B parameters (which is 100 times more than its predecessor), it can solve tasks such as summing up numbers or generating SQL queries out of a description in natural language (without any fine-tuning). A demonstration[2] of the *GPT-3* can be found below:

An example of GPT-3's ability to generate an SQL

SQL select all records from table users:
SELECT * FROM users

2.2.4 BART

Rather than a completely new architecture, the **BART** model brings up a new approach to pre-training *Seq2Seq* models using a denoising objective [19]. Except for slight modifications, the model follows the architecture of the *Transformer*. With the denoising objective, the encoder part reads a corrupted sequence and creates its contextualized representation. The decoder then tries to autoregressively reconstruct the original sequence. To construct a dataset of corrupted text, one can employ the following transformations that can be arbitrarily combined [19]:

- **token masking** - replace random tokens with `[MASK]` token
- **token deletion** - delete random tokens from the input

[2]Generated using `https://beta.openai.com/playground` on the first attempt. Our NL prompt is marked in bold font.

- **text infilling** - replace a span of tokens with `[MASK]` (span length is sampled from a *Poisson* distribution) or insert an extra `[MASK]` token to a random position
- **sentence permutation** - randomly shuffle the sentence order (applicable only if the input is a natural language text that consists of multiple sentences)
- **document rotation** - choose a random pivot position in the input and rotate the whole document around the chosen pivot

Using BART for Machine Translation

As explained previously, the *BART's encoder* consumes a corrupted text in the same language (same vocabulary) used by the decoder. Therefore, the pre-trained *BART* model cannot be used for MT as is and must be extended with an *additional encoder* consuming a sequence in a source language and producing its contextual representation. The acquired representation is afterward used as an input for the pre-trained *BART* model that generates a sequence in a target language [19]. This can be perceived as predicting a very noisy candidate translation in the *additional encoder* and polishing it in the pre-trained *BART*.

2.2.5 T5

In the paper "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer" (**T5**) [20], the authors extensively study pre-training objectives and architectures. Furthermore, they introduce a framework for converting every text-based task to a text-to-text problem, as shown in figure 2.7.

In the experiments, Raffel et al. [20] use a neural network architecture that follows the *Transformer* model except for slight modifications in *layer normalization* [16]. Furthermore, they also introduce a new dataset called **Colossal Clean Crawled Corpus** that comprises approximately 750 GB of text, which they use for pre-training experiments on the denoising learning objective. The denoising objective is slightly different from the one used in the *BART* model. Instead of reconstructing the whole sentence, the T5 is trained to predict only the tokens replaced by masking tokens.

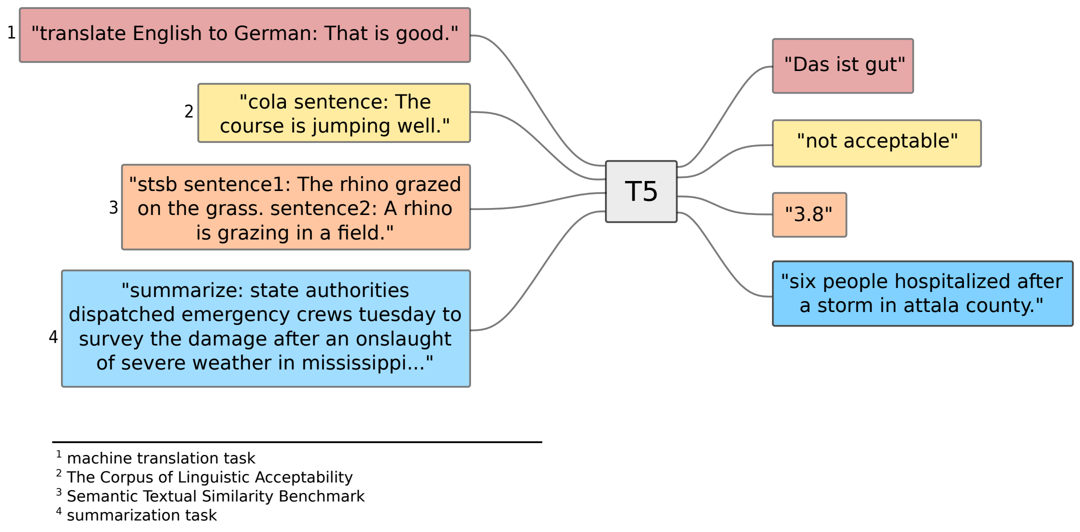

1 "translate English to German: That is good."
2 "cola sentence: The course is jumping well."
3 "stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
4 "summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi..."
T5
"Das ist gut"
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."
1 machine translation task
2 The Corpus of Linguistic Acceptability
3 Semantic Textual Similarity Benchmark
4 summarization task

Figure 2.7: Example of conversion of text-based tasks into text-to-text problems using the approach introduced by Raffel et al. [20].

2.3 Tokenization

Tokenization is an integral part of a data input pipeline for most NLP systems. It takes care of breaking down a whole bunch of text into smaller pieces, such as words that can be represented using numerical indices pointing into a vocabulary. The process of tokenization itself is pretty simple. The difficult part thereof is constructing a vocabulary suitable for a given problem.

One possible approach for building a vocabulary is to tokenize the text on white spaces and build a vocabulary containing all the words that occur in a corpus more than, for example, five times. The problem with this approach is that we might get vocabularies of millions of tokens which results in exhausting GPU memory consumption only for storing word embeddings. For example, Google's pre-trained Word2Vec [21] embeddings are trained with a vocabulary of 3M words[3].

In addition to the memory consumption problem, one must deal with tokens that are not part of the vocabulary. Such tokens are often referred to as *out of vocabulary* (OOV) tokens. We can either decide to entirely ignore the OOVs or create a unique `[OOV]` token representing all the OOV words

[3]`https://code.google.com/archive/p/word2vec/`

in an input. However, nowadays, other approaches can significantly reduce a vocabulary size and mitigate the OOVs completely. These techniques are called **subword tokenization** (example in figure 2.8) and are further described in the subsequent sections.

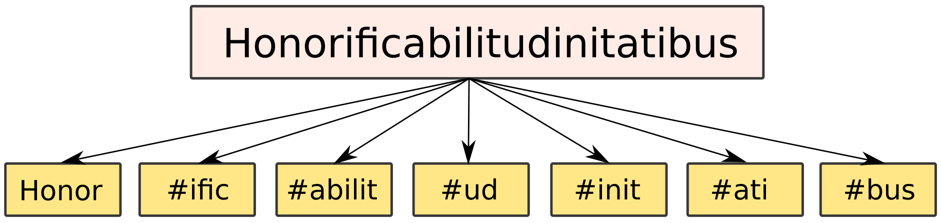


Figure 2.8: Example of subword tokenization of the longest word appearing in Shakespear's plays - *Honorificabilitudinitatibus*.

2.3.1 Byte Pair Encoding

Byte Pair Encoding (BPE) was initially a compression technique [22]. Nowadays, it is often applied for building vocabularies for subword tokenizers in the *NLP*. The *BPE* starts with a small vocabulary of all allowed characters and special tokens when building the vocabulary. In each step, it builds a co-occurrence statistic for all possible pairs of tokens from the current vocabulary and merges the two most common tokens into a new one. This simple process is applied in a loop until the desired vocabulary size is not reached. The *BPE* by design requires the input to be already pre-tokenized (for example, on whitespaces) to ensure that no token in the vocabulary spans over multiple words. It means that the input of the *BPE* algorithm is a list of all words from a corpus for which we build a vocabulary.

2.3.2 Word Piece

Another subword tokenization technique is the **Word Piece** encoding. It is, to some extent, identical to the *BPE*. The most significant difference between these two techniques lies in the merging step. Instead of combining the two most often co-occurring tokens, the *Word Piece* merges two tokens that would increase a likelihood of a training corpus the most if added to the vocabulary [23]. The likelihood of a training set ($\mathcal{X}$) can be computed using the formula 2.5, where $P(x_i|x_1, x_2, ..., x_{i-1})$ can be obtained using a trained language model.

$$P(\mathcal{X}) = \sum_{x \in \mathcal{X}} \prod_{i=1}^{N} P(x_i | x_1, x_2, ..., x_{i-1}) \tag{2.5}$$

The merging criterion may seem too complicated to be used in a real-world scenario. That is because a naive approach would require, for each merging step, to train a language model with many different vocabularies corresponding to every possible pair of tokens added to the vocabulary. However, we can employ several optimizations that significantly reduce the number of trained language models. For example, we can prune all pairs of tokens that do not occur in the corpus or apply a heuristic to test only the pairs with a high chance of being embraced in the vocabulary [24].

2.3.3 Sentence Piece

Both approaches mentioned above suffer from a common drawback - they do not, by design, treat a text that is not pre-tokenized. This represents a significant obstacle, for example, when working with languages that do not separate words using spaces (for example, Japanese or Chinese language). To mitigate this problem, Kudo and Richardson [25] proposed a novel approach called the **Sentence Piece**.

The *Sentence Piece* algorithm takes whole sentences as input and treats the space character as a special symbol in the vocabulary. Internally, it uses either a *BPE* or *Unigram model* [26] to build the vocabulary of a given size. Thanks to that modification of the original approaches, the resulting vocabulary can contain multi-word utterances such as `New_York`, represented with a single token.

2.4 Decoding Algorithms

As we know from the previous sections, a language model estimates $P(x_t | x_0, x_1, ..., x_{t-1})$, which is the probability of the next word in a sentence being x_t given the previous words. If we calculate the probability for each word in the vocabulary, we get a probability distribution that can be used to decide about the next generated word. The way of choosing the next word given the computed probability distribution is a crucial parameter affecting the quality of a generated text. This section describes the most common decoding algorithms for generating text. These involve a greedy approach, and a more sophisticated beam search. Lastly, we discuss how the decoding can be further adjusted using a straightforward yet powerful sampling.

2.4.1 Greedy Decoding

The most straightforward decoding algorithm is **greedy decoding**. It chooses the most probable word in each decoding step, as captured in equation 2.6. Although effortless and computationally efficient, it is not considered a good choice for generating high-quality text. That is because it often generates very noisy outcomes with many grammatical errors due to the lack of possibility to backtrack and re-evaluate previous decisions. In other words, it always chooses a local optimum without trying to converge to a global optimum [3].

$$x_t = \arg\max_x P(x|x_0, x_1, ..., x_{t-1}) \tag{2.6}$$

2.4.2 Beam Search

A more sophisticated approach that tries to find a sufficient approximation of a global optimum is called a **beam search**. Instead of choosing the most probable token at each time step, the *beam search* keeps track of the k most probable hypothesis and expands all of them [27]. The number k is a hyperparameter of this method and is often called *beam size* or *beam width.* The approach is visualized in figure 2.9.

The choice of *beam size* is crucial when choosing this decoding method. If we choose $k = 1$, the *beam search* degrades into the simple *greedy decoding.* On the other hand, if we set k to a high number, the computation is way too expensive.

Last but not least, we would like to elaborate on the statement that the beam search provides only an approximation of a global optimum. If one wants to find an exact global optimum, the computation complexity would be $\mathcal{O}(|V|^N)$, where $|V|$ is a size of a vocabulary and N is a maximum length of a generated sequence. In the real world, the vocabulary can have 50K tokens, and we could want to generate sequences of 64 tokens. Although this example is somewhat optimistic, we need to explore 5.42×10^{300} possible decoded sequences to find the global optimum. On the other hand, the *beam*

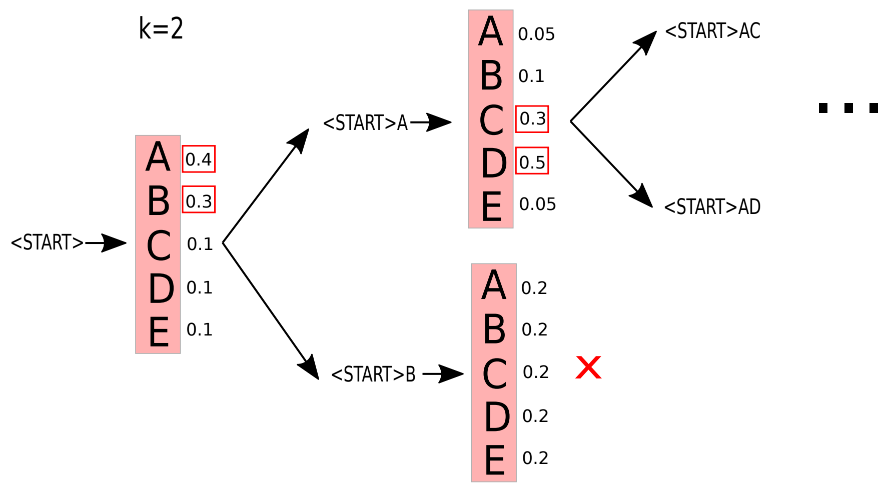


Figure 2.9: Visualization of the beam search algorithm with beam size $k = 2$.

search's complexity is $\mathcal{O}(k \times N)$, which significantly reduces the computation time if we preserve k in reasonable values $k \in 5, 6, ..., 10$.

2.4.3 Adjusting the Generated Output

The previous decoding methods considering the k most probable tokens for expanding candidate hypotheses can be modified with a non-deterministic random **sampling**. An advantage is that such a modification is computationally cheap, and it produces more variable output, which may be desired in some applications. However, its significant disadvantage is that it might produce many words that do not make any sense in a given context due to the randomness of the choice. However, there are two additional options for better control over the quality of the output. The **Top-K Sampling** and **Softmax Temperature** are discussed in the subsequent paragraphs.

Top-K Sampling

A *Top-K* is a simple extension of the basic *sampling* method that allows the decoding algorithm to sample from k most probable words only. This significantly mitigates the occurrence of all the words with a very low probability that would make no sense if they appear in a given context. This minor enhancement can have a non-negligible impact on the achieved results.

Softmax Temperature

Unlike the *Top-K Sampling*, the *Softmax Temperature* technique [28] does not reduce a decision space. Instead, it enables us to regulate the variability of the output by either smoothing or roughening the distribution.

$$p_i = \frac{e^{z_i}}{\sum_j (e^{z_j})} \tag{2.7}$$

The probability distribution of the next token in a sequence is usually computed using a softmax function that can be found in the equation 2.7. To use the *softmax temperature* method, we slightly adjust the original equation by adding a temperature term $\mathcal{T}$ as shown in equation 2.8. The setting of the $\mathcal{T}$ parameter affects the distribution in the following way:

- $\mathcal{T} = 1 \Rightarrow$ normal softmax
- $\mathcal{T} > 1 \Rightarrow$ softens the distribution - more diverse output that tends to be less correct
- $\mathcal{T} < 1 \Rightarrow$ prefers more probable words - the output tends to make more sense and be grammatically correct

$$p_i = \frac{e^{z_i/\mathcal{T}}}{\sum_j (e^{z_j/\mathcal{T}})} \tag{2.8}$$

2.5 Evaluation Metrics

In the previous sections, we presented a text generation framework based on neural networks. So far, we have discussed model architectures, tokenization techniques, and generating algorithms. This section focuses on evaluating the quality of a text generated using the previously mentioned methods.

2.5.1 Cross-entropy and Perplexity

Suppose we want to train a language model and evaluate it without applying it to an end task. In that case, we can use an *intrinsic* evaluation metric such as **cross-entropy** or **perplexity**. The *cross-entropy* can be perceived as an average number of bits needed to encode data originating from probability distribution p while encoding them using an approximate probability q [29]. The *cross-entropy* of a language model on a test set Y of length N can be computed using equation 2.9.

$$H(Y) = -\frac{1}{N} log_2 P(Y) \tag{2.9}$$

The *perplexity* can be directly derived from the *cross-entropy* and is defined as an average number of words encoded by $H(Y)$ bits (equation 2.10) [30]. Another way of thinking about the *perplexity* is to perceive it as a weighted branching factor or an average number of possibilities, between which the language model decides when predicting the next word.

$$PP(Y) = 2^{H(Y)} = 2^{-\frac{1}{N} log_2 P(Y)} \tag{2.10}$$

Furthermore, the *perplexity* has an alternative definition using which we can compute the *perplexity* as an inverse probability of a test corpus, normalized by its length (equation 2.11) [3]. When measuring both the *cross-entropy* and *perplexity*, we aim to minimize it, which corresponds to maximizing the predicted probability of the corpus.

$$PP(Y) = (\prod_{i=0}^{N} P(y_i | y_0, y_1, ..., y_{i-1}))^{-\frac{1}{N}} \tag{2.11}$$

2.5.2 BLEU

BLEU, which stands for bilingual evaluation understudy [31], is generally a metric that evaluates the quality of a generated text given a known target. It is widely used for machine translation tasks (MT) since it has been shown that it correlates well with human judgments.

The core idea behind the metric is to compute a ratio of *n-grams* from a generated text that also appears in the target sequence (in other words, precision). The *n-gram* overlap (equation 2.14) is computed for *n-grams* of different lengths ($n \in 1, 2, ...k$) and is combined via product (equation 2.12). Additionally, the metric employs a *brevity penalty* β (equation 2.13) that penalizes a model for generating too short translations [31]. Without the penalty, it would be possible to generate only a single word matching at least one word in the target while receiving the highest possible score. This is caused by the fact that the *BLEU* is a precision-based metric.

$$BLEU = \beta \prod_{i=1}^{k} p_n \tag{2.12}$$

$$\beta = e^{min(0,1-\frac{len_{ref}}{len_{trans}})} \tag{2.13}$$

$$p_n = \frac{|\text{translation n-grams} \cap \text{target n-grams}|}{|\text{translation n-grams}|} \tag{2.14}$$

2.5.3 ROUGE

ROUGE, which stands for Recall-Oriented Understudy for Gisting Evaluation [32], does not provide a single number that measures how good a model is. Instead, it forms a set of metrics suitable for evaluating sequence generation tasks. Unlike the *BLEU* score, *ROUGE* is recall-oriented and, therefore, more suitable for tasks such as summarization. Below we state some of the metrics that are available in the *ROUGE* evaluation set:

- ROUGE-N - n-gram recall computed between a generated and target sequence (equation 2.15)
- ROUGE-L - longest common subsequence (LCS) [33] between a candidate and target sequence
- ROGUE-W - weighted version of LCS that scores consecutive matches higher than more distant ones
- ROGUE-S - skip-gram overlap between generated and target sequence [32]
- ...

$$ROUGE_n = \frac{|\text{translation n-grams} \cap \text{target n-grams}|}{|\text{target n-grams}|} \tag{2.15}$$

2.5.4 BERT Score

A **BERT Score** is a novel technique for evaluating text generation tasks, based on an *F1 score* (equation 2.18). However, the *BERT Score* employs adjusted version of a recall (equation 2.16) and precision (equation 2.17), calculated from contextual embeddings of candidate (x) and reference sequence ($\hat{x}$) [34]. Although the name indicates the usage of the *BERT* model [35], the metric can be used in combination with any other model capable of producing contextual embeddings.

$$R_{BERT} = \frac{1}{|x|} \sum_{x_i \in x} \max_{\hat{x}_j \in \hat{x}} x_i^T \hat{x}_j \tag{2.16}$$

$$P_{BERT} = \frac{1}{|\hat{x}|} \sum_{\hat{x}_j \in \hat{x}} \max_{x_i \in x} x_i^T \hat{x}_j \tag{2.17}$$

$$F_{BERT} = 2 \frac{P_{BERT} R_{BERT}}{P_{BERT} + R_{BERT}} \tag{2.18}$$

The equations above show the adjusted calculation of recall and precision. The recall (equation 2.16) is a sum of dot-products between each token from the candidate sequence with the most similar token from the target sequence. On the other hand, the precision (equation 2.17) is a sum of dot-products between each token from the target sequence with its most similar token from the generated sequence.

One can see that the significant benefit of this metric is that it can consider synonyms. For example, imagine a candidate sequence "*He's got black hair*" and a target sequence "*He's got dark hair*". Unlike *BERT Score*, all the metrics mentioned earlier would penalize the model for generating the word "black" instead of "dark" as if they are entirely dissimilar. However, it shall be noted that utilizing the *BERT Score* requires having a well-trained model for contextual representations in the target language. If no such model is available, it might be time-consuming to train a new one just for a metric calculation. In such a case, it might be better to utilize some of the previous metrics.

3 Source Code Processing using Machine Learning

3.1 Motivation and Relation to NLP

Machine learning (ML) techniques can be applied to a wide range of sophisticated problems. The tasks that work with some kind of text are focused by natural language processing (NLP), which is a hot research field with innovative approaches being introduced every month. The NLP methods benefit from the repetitiveness and predictability of utterances produced by humans. This raises the question of whether the source code has similar properties, allowing natural language processing techniques to be applied to source codes.

Suppose the presumption of source code having similar properties as natural language is valid. In that case, it will enable us to build or improve software development tools helping software developers with their job. The impact of such tooling is not-negligible since software is becoming an essential part of almost everything in the modern world. Moreover, developing reliable software is highly time-consuming and very expensive. It means that every improvement speeding up the whole development process is expected to be appreciated by the software community.

The research question about source code having similar properties as natural language is examined in the paper by Hindle et al. [36]. The authors build multiple *n-gram language models* [3] on different natural text and source code corpora and compare the naturalness of the individual corpora using the *perplexity* (section 2.5.1). The conducted experiments conclusively show that *language models* can benefit from the same properties as in the case of the natural language. Therefore, applying NLP techniques to various tasks related to source code processing might work very well. The possible applications of the NLP techniques alongside existing datasets and approaches are discussed in the rest of this chapter.

3.2 ML Applications in Software

In this section, we briefly present some of the tasks where artificial intelligence can be helpful in terms of improving the software development toolkit. For each of the mentioned tasks, we state an example of how it can be integrated into real life and reference papers aiming to solve the task. For a more detailed survey of ML applications in the source code processing domain, see the work by Allamanis et al. [37].

3.2.1 Documentation

The first task we present is automated *code documentation*. An objective of this task is to generate documentation in the natural language given a source code. A system that can provide us with such functionality can serve to better understand legacy parts of old software systems that often lack documentation. The lack of documentation makes it difficult for a developer to orient himself in the code base and make changes without introducing new errors. Several research papers [38, 39] already target the task of code documentation generation. Furthermore, there are also available datasets [40] for training such documentation generators.

3.2.2 Code Migration

A *code migration* aims to convert a source code written in one programming language into another. The two languages can be, for example, only different versions of the same programming language, such as *Python 2* and *Python 3* [41]. Or the migration might be done between entirely different languages [42]. Either way, the ability to convert the source code between two languages is beneficial, for example, if a software company wants to keep up with software development trends. The tricky part of this task is collecting training data since one needs a parallel corpus of source codes doing the same thing in two different languages. To avoid the complex dataset collection process, one can use an existing dataset such as *CodeXGLUE* [43].

3.2.3 Code Completion

Another valuable application of ML techniques in the software development domain is *code completion*, which is already included in every modern integrated development environment (IDE). The code completion system aims to finish a piece of code that a developer started to type. Traditionally, these systems try to predict just the unfinished token, which can often be done only based on the information obtained using syntactical and semantic analysis of a surrounding context. The newer approaches employ modern neural network models [44] and go further by predicting longer pieces of code that may span over multiple lines. A real-world example of using a neural code completion system in an IDE is, for example, the *Codota*[1] plugin for the *IntelliJ IDEA*[2].

3.2.4 Code Repair

To improve the reliability of a source code, *automatic repair* [45, 46] systems are attempting to autonomously detect and fix buggy code. Such a task is very complex because fixing a detected syntactical or semantic error may induce a logical error in the system or break another function working correctly for a long time. This is probably why such tools are not yet present in our software development tool-chains.

3.2.5 Code Generation / Program Synthesis

Last but not least, we present the *code generation* (sometimes also called *program synthesis*) task that is the main subject of this work. This task aims to generate source code in a target programming language based on a specification. For example, the specification can be an English text describing the desired functionality or several input-output examples. In the rest of this work, we will focus only on program generation based on a natural language description since it reflects the way how a developer can use it when working in an IDE.

Generating source code using a neural network might first seem a little bit unreal. However, it shall be noted that at the moment, we do not expect the neural network to generate a whole codebase of a large-scale project from the customer's description only. Instead, a generative model can generate shorter pieces of code, solving a problem that the developer would otherwise have to search for on the internet [47].

To give an example of such usage, imagine displaying current weather

[1]`https://www.codota.com`
[2]`https://www.jetbrains.com/idea/`

on a webpage. Since probably no one remembers the URL to an API that provides such information, we use Google to search for something like "stack-overflow get current weather using API". The first link we find is probably a StackOverflow (section 6.2.1) thread[3] with the desired solution (listing 3.1). With the program generation plugin in our IDE, we could write a similar query directly in the source code and use a command to trigger the generation process. The plugin can then allow the developer to select between multiple feasible solutions, which reduces all the time required for searching the topic on the internet, improving the developer's workflow.

```
var jsonData;

$(document).ready(function ()
{
  $.getJSON('http://api.openweathermap.org/data/2.5/weather?q
   ↪ =London,uk', function(data) {
      jsonData = data;
      $('.city').text(jsonData.name);
      // etc
  });
});
```

Listing 3.1: JavaScript solution for querying current weather at the given location.

3.3 Existing Code Generation Datasets

In this section, we describe existing datasets related to source code processing using machine learning methods. Most of the presented datasets are directly applicable for source code generation tasks. The description starts with more extensive collections such as *CodeNet*, *CodeSearchNet*, or *CoNaLa*. The smaller-sized datasets are then described in the latter part of this section.

3.3.1 CodeNet

The **CodeNet** dataset represents a collection of 14M source files acquired from programming challenge platforms with automatic judgment systems

[3]https://stackoverflow.com/questions/27639668/open-weather-api

like AIZU[4] or AtCoder[5] [48]. Each data example from the dataset represents a single submission for one of the 4053 problems. For each submission, the dataset also provides additional metadata such as programming language, judgment result, and computation resources used during execution.

This dataset can be utilized for various tasks such as classifying a programming language, computing code similarity, or predicting resource consumption. Furthermore, one can also acquire a definition of the individual problems using a RestAPI of the judgment platforms. It makes it possible to use the dataset for a generation. However, the definitions of programming challenge problems are usually overly complex. Therefore, the dataset does not seem to be a good fit for generating a source code.

3.3.2 CodeSearchNet

Another dataset providing a large set of source code-related examples is a **CodeSearchNet** [49]. The dataset is designated for training information retrieval systems and contains a test set with human-annotated relevance judgments. However, since the dataset contains both a source code and its description, it can be utilized for other tasks such as code generation. Together, the dataset comprises 2.3M examples from 6 different programming languages. Detailed information about the dataset's size can be found in table 3.1.

CodeSearchNet Dataset Size	
Language	**# Examples**
Go	347 789
Java	542 991
JavaScript	157 988
PHP	717 313
Python	503 502
Ruby	57 393

Table 3.1: Number of examples in the CodeSearchNet[49] dataset depending on a programming language

Unlike the previous dataset, the *CodeSearchNet* is automatically collec-

[4]`https://judge.u-aizu.ac.jp/onlinejudge/`
[5]`https://atcoder.jp`

ted from public *GitHub*[6] repositories. Each example from the dataset represents a single function whose documentation comment is used to extract the corresponding description. However, a naive approach to the extraction can produce a noisy outcome, and therefore, the authors employ several filtering criteria to clean up the dataset. Those criteria involve, for example, filtering out the functions whose implementation is shorter than three lines, filtering out functions whose name contains "*test*", et. cetera [49].

3.3.3 CoNaLa

The next dataset designated directly for generating source code snippets from a natural language query is a **CoNaLa** dataset [50]. It represents a collection of approximately 600K examples acquired from the *StackOverflow* (section 6.2.1) platform. To produce a clean dataset, the authors introduce a filtering method that selects only high-quality pairs of source code and its natural language (NL) description.

As a first step in collecting the dataset, the authors have collected 527 human annotations of Python-related *StackOverflow* question-answer pairs. For each presented question-answer pair, the annotators were asked to write a curated description of the source code's intent. Afterward, the human annotations were combined with a set of hand-crafted features to train a classifier predicting the quality of the NL-PL (natural language-programming language) pair. The following listing states a few examples of the utilized hand-crafted features [50]:

- ***contains import*** - a flag indicating whether a code snippet contains an `import` statement
- ***starts with assignment*** - a flag indicating whether a source code starts with an assignment statement (for example, `a = [1, 2]`)
- ***accepted answer*** - a flag indicating whether a source code originates from an accepted answer
- ***number of lines*** - number of lines of a source code
- **...**

Afterward, the trained classified is used to predict the quality of each NL-PL pair when crawling the *StackOverflow* content. During the dataset

[6]`https://github.com`

assembling procedure, the script crawls all question-answer pairs from which a set of NL-PL pairs is generated. The NL description in each pair is made up of a question's title. The corresponding programming language snippets are then extracted from related answers considering possible code snippet lengths are considered. For example, if a code contains three lines, six candidate snippets will be generated - lines 1, 2, 3, 1-2, 2-3, 1-3 [50]. Finally, the one with the highest relevance score determined by a neural network classifier is selected from all the possible candidate pairs. As a result, most of the examples present in the dataset contain a target code consisting of a single line only.

3.3.4 Django

A **Django** dataset [51] is a collection of 18 805 lines from a web application written in a Python framework called *Django*[7]. For each line in the source code, a human-written pseudo code is available. Initially, the dataset was created for Python to pseudo-code translation to provide an understandable description of what a Python source code does. However, the dataset can be used the other way to train a generative model capable of generating source code out of a pseudo code.

3.3.5 NAPS

The next dataset we discuss in our work is a **NAPS** [52] dataset that collects problem statements with correct solutions from the programming contest website `http://codeforces.com`. The dataset provides 19 126 examples of problem statements with the corresponding *Java* solution. Most of the examples are automatically acquired from the aforementioned website, whereas nearly 500 contain a high-level description of the problem collected using crowd-sourcing.

3.3.6 Hearthstone

A **Hearthstone** dataset introduced in a paper by Ling et al. [53] is extracted from a Python simulation of a well-known game called *Hearthstone*[8]. Each data example represents a game card that matches a single Python

[7]`https://www.djangoproject.com`
[8]`https://github.com/danielyule/hearthbreaker`

class. The dataset provides 665 examples split into train, evaluation, and test parts. The dataset is designated for the source code generation task, so each example contains an English description of a card. Furthermore, each example contains additional information such as card type, power, or health.

3.4 Existing Models and Approaches

Previously, we have presented several source code processing tasks and datasets. Following this, we present existing neural network models and approaches dealing with source codes. Firstly, we present two source code encoders, and afterward, we describe *Transformer*-based (section 2.2.2) approaches that generate source codes from natural language descriptions. In the end, we also discuss other relevant research papers with a completely different approach, which should give the reader a broader view of the problem domain.

3.4.1 CodeBERT

A **CodeBERT** is a pre-trained bi-modal neural network capable of processing both source code and natural language simultaneously [39]. The architecture of the *CodeBERT* employs a stack of twelve *Transformer's encoder* (section 2.2.2) layers whose weights are initialized using the weights from the *RoBERTa* model [54].

During a pre-training phase, the *CodeBERT* learns two objectives on a *CodeSearchNet* dataset (section 6.2.4). The first learning objective is a *masked language modeling* (MLM), in which the model predicts original tokens that were replaced by `[MASK]` token in the input. As a second objective, the *CodeBERT* chooses a replaced token detection (RTD). The RTD objective replaces some of the tokens in input with another token from the model's dictionary. The model then attempts to predict whether each token is replaced or not. However, this would be very simple for the model to decide if the replacement is chosen randomly. Therefore, the *CodeBERT* employs an additional generator model, whose aim is to produce as trustworthy replacement tokens as possible to fool the CodeBERT model (discriminator). Such a setup is depicted in figure 3.1.

The resulting model can produce contextual representations [13] of the

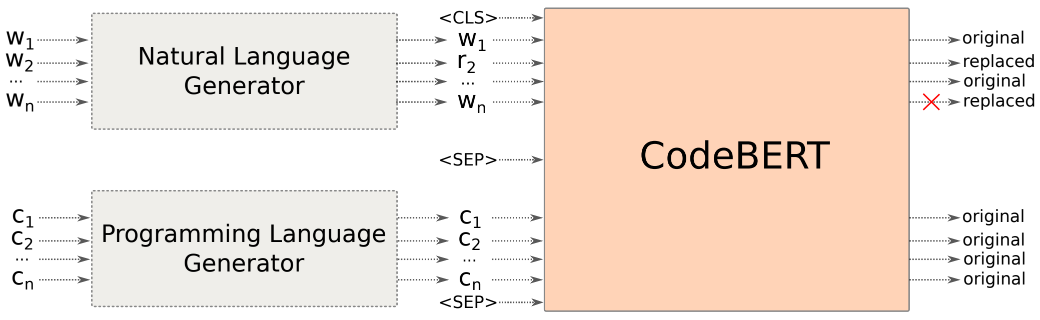


Figure 3.1: Generator-discriminator architecture used when pre-training the CodeBERT model.

input, encompassing both a natural language and source code. The representations can be directly applied to various tasks such as programming language classification or source code retrieval from a large base of queries and code snippets (for example, StackOverflow[9]).

3.4.2 MQDD

Like the *CodeBERT*, **MQDD** [55] represents a bi-modal pre-trained model for both natural and programming languages. The model is built on a *Longformer* architecture [56], which modifies the utilized attention scheme to scale linearly with the growing input sequence length. Therefore, *MQDD* is more suitable for processing longer pieces of source code.

Another significant difference from the *CodeBERT* is the pre-training objective. In addition to the MLM, the *MQDD* employs two other tasks specific to the *StackOverflow Dataset* introduced in the work [55]. The first dataset-specific task is called *Question-Answer*, and its target is to predict whether an input pair represents a question-answer relationship. The second learning objective, *Same Post*, aims to predict whether the input pair originates from the same post (either from the same question or the same answer).

The aforementioned pre-training objectives are designed to force the model to build a deep understanding of the source code's meaning to resolve the relationship between the input pair. Therefore, the proposed learning objectives are especially beneficial for tasks such as automatic detection of

[9]`https://stackoverflow.com`

duplicate questions from the *StackOverflow*. However, the paper [55] shows that the resulting model has limited generalization abilities, and if applied to different tasks, a *negative transfer* [57] effect might be observed.

3.4.3 Codex & GitHub Copilot

Unlike the two models above, a **Codex** [58] is a generative model for source code generation. Its architecture follows the *GPT* discussed in section 2.2.3. The model has 12B parameters trained on 159GB of Python source code acquired from *GitHub*. The resulting model can solve 37.7% of 164 programming problems available in the *HumanEval* dataset introduced in the same paper [58]. Additionally, by sampling 100 candidate solutions for each problem, the model can solve 77.5% of the problems. Although this is an imposing result, we must keep in mind that no developer is willing to examine 100 candidate solutions before choosing the right one.

A bit different version of the *Codex* model is already integrated into *GitHub Copilot*[10], providing code snippet suggestions as you write source code. However, neither the *Codex* model nor the *GitHub Copilot* is publicly available. The *Codex* is even provided through a paid API, which raises an ethical question in the development community of whether it is fair to use public source codes produced by thousands of developers to train a neural network model and sell it afterward to the same developers [59]. Furthermore, there are concerns that the *Codex* may reproduce memorized source code licensed under GPL license[11], which may potentially induce many legal obstacles when using it in a commercial environment.

3.4.4 CodeT5

Another generative model for source code and natural language is called **CodeT5** [60]. The model uses the same architecture as the original *T5* (section 2.2.5) but employs a re-designed pre-training strategy. The new pre-training objectives leverage code-specific aspects. More specifically, the pre-training objectives used in the work are the following [60]:

- **Masked Span Prediction** - same denoising objective as in the case of the original *T5*, where the model predicts the original content of a span of text replaced by `[MASK]`

[10] `https://copilot.github.com`
[11] `https://www.gnu.org/licenses/gpl-3.0.html`

- **Masked Identifier Prediction** - all input tokens that represent an identifier are replaced with `[MASK]`, and the neural network tries to recover the original identifiers
- **Identifier Tagging** - the neural network predicts whether each token represents an identifier or not
- **Bimodal Dual Generation** - for each of the NL-PL pairs in the dataset, the model tries to generate PL out of NL and vice versa

Since the model copies the T5's framework of converting every task to a sequence to sequence problem, the model can solve a wide range of tasks, including code summarization, code generation, or code migration. Furthermore, the model was pre-trained on the *CodeSearchNet* dataset (section 6.2.4) extended with `C` and `C#` code. Therefore, it can handle eight different programming languages. This makes the model a universal choice for various real-world applications.

3.4.5 Other Approaches

Unlike the previously presented approaches, some of the prior works are trying to leverage available definitions of a programming language grammar [61–63]. It means that instead of treating the source code as a sequence, these approaches build a parse tree of the code by expanding non-terminal symbols until only terminals remain. Such approaches leverage neural networks to predict which expansion rule from the grammar shall be used.

An attempt to improve the aforementioned parse-tree-based approaches by extending training data can be seen in the paper by Xu et al. [64]. In their work, they propose to augment the NL description of code snippets using official API documentation of the related method calls. Other research papers also expand an original PL grammar with additional rules representing code idioms [65]. A *code idiom* in the context of the work is an often-occurring subtree of a valid parse-tree. Adding such higher-level derivation rules into grammar enables the model to compose the desired source code using high-level building blocks.

Other PL generation papers also focus on specific languages such as SQL. For example, Xu et al. [66] introduce a method for generating SQL queries leveraging structural information about the database. This work is followed by Zhong et al. [66], who extended the method by employing reinforcement learning to improve the quality of the generated queries.

4 Analysis of the Problem

The problem of source code generation from a description in natural language is a complex task with many possible approaches. Many of these methods were presented in the preceding chapters, including neural network architectures, available datasets, and many more. This chapter aims to outline the key points of our approach to generating Python source codes from queries in English. Firstly, we describe our approach from a high-level point of view. Afterward, we discuss our selection of employed neural network model, and lastly, we analyze whether to use some of the existing datasets or create a completely new one.

4.1 General Approach

In section 3.4, we have presented several approaches to source code generation task, including those that leverage programming-language-specific aspects such as grammar. Despite the attractiveness of these methods, the approaches based on the *Transformer* architecture (section 2.2.2) seem to surpass the source-code-specific approaches. The *Transformer*-based models such as CodeT5 (section 3.4.4) or Codex (section 3.4.3) treat the programming language as a sequence of tokens similarly to natural language. Based on the information stated in section 3.1, the programming language shows an even lower degree of entropy compared to the natural language. Therefore, we presume that applying modern machine translation and text generation methods to our problem without employing programming language grammar is a reasonable choice.

Since the previous considerations lead us to use a *Transformer*-based model, we also need to consider the training strategy. That is because the *Transformer*-based models usually contain a massive amount of trainable parameters, and therefore, they require a large-scale dataset for training. To the best of our knowledge, there are no available parallel corpora of NL-PL pairs (further discussed in section 4.3) that would have sufficient size for training such large models end-to-end. Therefore, we will use the strategy followed by many *Transformer*-based models. We will pre-train a model on a large unsupervised dataset using a learning objective different from our target task. Afterward, we will use the pre-trained model to fine-tune it on the source code generation task. Such an approach is often called *transfer learning.*

The *transfer learning* strategy has more benefits than training large models without sufficient data for the end task. Another significant advantage is that the pre-trained model can serve as a base for a much more comprehensive range of tasks than a specialized model built and trained just for a single task.

4.2 Model

The source code generation from natural language can be perceived as a machine translation task. More specifically, in our work, we focus on translating English to Python. This fact leads us to employ a *Transformer*-based architecture that shows state-of-the-art results on the MT task, restricting our selection to the original *Transformer* (section 2.2.2), *T5* (section 2.2.5), *GPT* (section 2.2.3) or *BART* (section 2.2.4).

If we choose to use the original *Transformer*, we would need first to train an autoregressive decoder using a language modeling task on code. Afterward, either a randomly initialized or already pre-trained encoder must be added. Unfortunately, the pre-training phase cannot utilize the *encoder-decoder attention* due to the lack of paired NL descriptions to be processed by an encoder. Therefore, the model can not learn to consider the information from the encoder during decoding.

Another option is to use a decoder-only architecture, following the *GPT* model. However, using such a model will not allow us to use a monolingual dataset of source codes during the pre-training. That is because the decoder needs to process both the natural and programming language to understand and complete prompts such as *"translate 'get the last element from a list' from English to Python"*. Furthermore, another research paper has already explored this approach, so we try to find a different method.

Like the *GPT* model, prior publications have already explored the *T5* architecture and its usage for code generation. Furthermore, the strength of this model is the capability of multitask learning using the unified text-to-text framework (section 2.2.5). We presume that we can achieve better results by focusing on code generation tasks only, and therefore, we choose a different model.

Finally, after considering all the aforementioned possibilities, we choose to utilize the *BART* architecture. We will first pre-train a whole encoder-decoder model on source code denoising. Afterward, such a model can be directly used for tasks such as automatic code repair, which is, however, out of the scope of this work. After the pre-training, an additional encoder, which replaces the input embeddings of the base model, will be added. The additional encoder can be either initialized randomly, or we can employ an encoder pre-trained on a different task. Intuitively, the additional encoder performs a very noisy translation, afterward cleaned up using the *BART* model. This setup shows superior results on multiple MT datasets, supporting our choice.

With selecting an additional encoder for the fine-tuning phase, we will aim to choose an existing pre-trained model that can leverage a lot of previously gained knowledge. Besides that, we will compare the results achieved using randomly initialized and pre-trained additional encoders.

4.3 Datasets

So far, our approach involves pre-training a denoising *BART* model using a monolingual corpus of source codes and fine-tuning it together with an additional encoder to generate source code. However, we have not yet discussed which dataset we use in our experiments. The data selection is, therefore, analyzed in this section. Firstly, we discuss what data we can use for pre-training, and afterward, we focus on the fine-tuning datasets.

Dataset	**# Examples**
CodeNet [48]	3 340 048
CodeSearchNet [49]	1 156 085
CoNaLa [50]	600 000
Django [51]	18 805
NAPS [52]	16 000
Hearthstone [53]	665

Table 4.1: Size comparison of datasets for code generation (section 3.3).

4.3.1 Pre-training Dataset

Pre-training a model that consists of millions of parameters requires to have an extensively large dataset. Otherwise, the model can suffer from underfitting. In related works, we see that their training corpora come up to multiple hundreds of millions of training examples. However, table 4.1 shows that none of the available datasets provide such a huge amount of data. It means that we will need to prepare a new dataset.

To create a large base of training examples in feasible time, we will need to focus on data sources that can be automatically crawled without any human interaction. In other words, we cannot rely on having annotators that would help us to assemble and curate our dataset. Luckily, several possible data sources fulfill our criteria. One of these sources is StackOverflow (section 6.2.1), which offers a large amount of source code snippets included in questions and answers. Based on tags associated with the StackOverflow threads, we know that there are approximately 1.8M threads related to the Python programming language. From the thread count, we deduce that the StackOverflow will not yield a dataset of the desired size. Nonetheless, it may serve as a good source for building a fine-tuning dataset, as discussed later.

Another promising source of the large unlabeled corpus is the GitHub platform, which hosts more than 279M repositories[1]. Since Python is very popular these days, we expect that the Python repositories make up a large percentage of the total repository count. Therefore we choose GitHub as the primary data source for our pre-training phase.

4.3.2 Fine-tuning Datasets

For model fine-tuning, the amount of required training data is significantly smaller than in the pre-training, which enables us to use multiple datasets enlisted in table 4.1. However, we can not use the *CodeNet* dataset since it lacks paired natural language descriptions. Despite the possibility of downloading the corresponding NL descriptions, we conjecture that the descriptions from programming competitions are too high-level. Therefore, they may be overly challenging for a neural network model. Another dataset that we do not employ during fine-tuning is the *NAPS*, which focuses on *Java* rather than *Python*.

Furthermore, the *Django* and *Hearthstone* dataset do not match our intended goal of generating meaningful Python source codes out of English descriptions. In the case of the *Hearthstone*, the target source codes represent game cards; therefore, a resulting model would not be helpful if

[1]The repository count is declared by GitHub at `https://github.com/search`

integrated into an IDE. Furthermore, the *Django* dataset contains NL-PL pairs consisting of a single line of code and corresponding pseudo code. This is, however, not suitable for our work since we want to teach our model to generate meaningful pieces of source code that are syntactically valid. For example, teaching the model to do a translation like demonstrated in the example[2] below would produce a source code that is neither syntactically valid nor meaningful.

```
PSEUDO CODE                        -> PYTHON CODE
if KeyError exception is caught -> except KeyError:
```

In our experiments, we will employ only the *CodeSearchNet* and *CoNaLa* datasets. Besides, we will build our training data based on StackOverflow (section 6.2.1) platform. We will do so despite having one existing StackOverflow-based dataset enlisted in this work. Our motivation behind creating our own dataset lies in having more control over selecting the examples. The *CoNaLa* dataset assigns text from questions to source code snippets based on relevance score acquired using a neural network predictor. In our work, we will pair only questions with the answers marked as accepted. The accepted answer relation can be perceived as a label assigned by humans, and it expresses that the code snippet solved the author's problem. Therefore, we believe that the accepted answer is a significantly better relevance measure than a neural network prediction used to construct the CoNaLa dataset.

[2]The example represents a real training example obtained from `https://github.com/odashi/ase15-django-dataset`.

5 Pre-training a Generative Model for Source Code

As outlined in the preceding analysis, our approach to generating source code from a natural language description involves two main stages - pre-training and fine-tuning. This chapter focuses on the pre-training part aiming to obtain a pre-trained model based on *BART* architecture capable of denoising corrupted source code in Python language. We call the resulting model a **CodeFormer**.

The whole chapter is structured as follows. In the beginning, we focus on acquiring a large unlabeled corpus of Python source code for training the *CodeFormer* model. Following the dataset construction, we discuss tokenization used in the pre-training, which involves several source code-specific steps such as source code linearization. Subsequently, we present further details about the exact parametrization and implementation of the utilized *BART* model, followed by a description of our pre-processing pipeline. Lastly, we discuss the setup of our pre-training experiment and present the achieved results.

5.1 Dataset

Section 4.3 states that training a large pre-trained model requires at least a few million training examples. However, none of the existing datasets presented in section 3.3 provide such an extensive training corpus, and therefore, we choose to create a new dataset. We appoint *GitHub*[1] to be a data source for the new dataset, thanks to the fact that it embraces more than 279M repositories from different languages (section 4.3.1). The rest of this section discusses how the *GitHub* data can be obtained using a *Rest API* and how the downloaded repositories can be processed to create a training corpus.

[1]`https://github.com/`

5.1.1 GitHub API

GitHub exposes a comprehensive *API*[2], providing endpoints for repository manipulation, issue management, et cetera. Most importantly, it enables a client to search repositories based on criteria provided using a request parameters. The most important filter options are the *number of stars*, *fork count*, and *source code language*.

Since the work focuses on generating *Python* source code, the *language* filter can be used to omit repositories containing no *Python* code. Furthermore, the *forks* and *stars* can measure the repository's quality, hence the underlying source code. However, we conjecture that favoring the quality of training examples over the dataset size would not improve the resulting neural network model. In order to be able to select suitable filter parameters, thanks to which we get a sufficiently large and high-quality dataset, we performed a detailed analysis of the individual filter settings presented in the next paragraph.

Table 5.1 shows a few examples of *GitHub API* queries with the resulting number of matching repositories. One can see that the number of found repositories decreases rapidly with every slight tightening of the selection criteria. This work chooses to process all repositories comprising *Python* source code marked with at least 1 star based on the analysis. We believe that even such a soft criterion filters out most not-maintained repositories containing noisy source code while preserving a sufficient number of repositories for acquiring a massive dataset.

Query	**# Repositories**
language:Python	2 708 833
language:Python stars:>0	1 451 639
language:Python forks:>0	1 018 460
language:Python stars:>1	738 090
language:Python stars:>0 forks:>0	651 445
language:Python forks:>1	480 654
language:Python stars:>1 forks:>1	350 666

Table 5.1: Analysis of the number of repositories found depending on filtering conditions.

[2]`https://docs.github.com/en/rest/reference`

5.1.2 Crawling Repositories

In order to download all the repositories acquired using the previously mentioned filtering, the query must be split into a series of queries yielding a lower number of results. That is due to the limitations of the *GitHub API*, which allows us to get only the first 1000 results for each posted query. Besides this limitation, the API returns results in pages with a maximum of 100 items. Moreover, there is also a request rate constraint that allows the client to execute 5000 requests per hour (in the case of a user authenticated by an access token). Fortunately, these obstacles can be mitigated by fetching the results in buckets determined by the *creation date* filter.

When crawling the repositories, it is also essential to filter out source files that would impair the ability of the neural network to generate syntactically correct code. Therefore, source codes with a syntax error shall be excluded from the training set. To filter out such files, one can use the *ast*[3] library, which is a part of the *Python* installation. More specifically, we can use `ast.parse()` function. If the function execution ends up with no *exception*, the source code can be considered compliant with the language grammar. Furthermore, we exclude all files with the name *__init__.py* since such files are used to structure the source code modules into packages and contain no functional code. After considering all the constraints and filters discussed previously, we end up with the crawling Algorithm 1.

5.1.3 Dataset Statistics

The resulting *GitHub* dataset obtained using the steps described in the previous section comprises 230M training examples. Compared with existing datasets (table 4.1), our novel dataset is an order of magnitude larger and, therefore, more suitable for training a large pre-trained model. For detailed information about the dataset, see table 5.2.

In addition to comparing the size of our dataset with those mentioned earlier, we also compare our dataset with the corpus used to train the *Codex* model (section 3.4.3), which used a training set of *159GB* of source code. Our dataset uses *184GB* of disk space, which indicates that our dataset is slightly larger. However, it is not clear how many training examples the *Codex* model used. Therefore, the comparison is very rough.

[3]`https://docs.python.org/3/library/ast.html`

Algorithm 1 Algorithm for crawling Python repositories using stars count

```
Require: $n_{stars} \geq 0$
Ensure: Filtered python source code stored on the FS (file system)
  $date \leftarrow$ 2018-01-01
  $processed \leftarrow \emptyset$
  $page \leftarrow 1$
  while $True$ do
      wait(2)                        ▷ wait for 2 seconds due to the request rate limit
      $repos \leftarrow$ get $page$-th page of repos with stars $\geq n_{stars}$ created on $date$
      if $\|repos\| = 0$ then
          $date \leftarrow date + 1$                   ▷ add one calendar day
          $page \leftarrow 1$
          continue
      end if
      for $repo$ in $repos$ do
          if $repo$ not in $processed$ then
              $dir \leftarrow$ clone $repo$ to a directory on the FS
              $dir \leftarrow$ filter out all files from the $dir$ with name ___init___.py
              $dir \leftarrow$ filter out all files from the $dir$ that cannot be parsed
              $processed \leftarrow processed \bigcup repo$
          end if
      end for
      if $\|repos\| \geq 100$ then              ▷ all repos did not fit to one page
          $page \leftarrow page + 1$
          continue
      else
          $date \leftarrow date + 1$                   ▷ add one calendar day
          $page \leftarrow 1$
          continue
      end if
  end while
```

Dataset statistics	
number of source files	23.9M
number of training examples	230M
number of tokens	35.8B
average examples per source file	10.4
average tokens per example	155.9

Table 5.2: Statistic of the *GitHub* pre-training dataset.

5.2 Tokenizer

For tokenizing and converting the input text into a sequence of numerical indices, this work uses a *WordPiece* tokenizer (section 2.3.2) since it is a usual choice of *Transformer*-based models (section 2.2.2). The following subsections describe pre-tokenizing the input, determining the vocabulary size, and training the tokenizer.

5.2.1 Pre-tokenization

In order to train the tokenizer, we need to feed in the input sequences as a list of words. In the terminology of a source code, we do not feed in words but instead tokens produced by a *lexical analyzer* of the given programming language. These can be obtained using the *tokenize*[4] library provided by the standard *Python* installation. However, the token stream produced by the *tokenize* library must be further transformed using the following rules:

- remove all characters that are not printable (not in `string.printable`)
- replace newline tokens with special `[NL]` token
- replace indentation tokens with special `[IND]` token
- replace dedentation (inversion of indentation) tokens with special `[DED]` token
- remove the encoding token that is located at the beginning of each token stream produced by the *tokenize* library
- remove all docstrings (single-line comments are removed by default)
- return the filtered tokens as a single string, where each token is separated by a white space (pre-tokenization)

The whole process described above is entirely reversible so that it is possible to convert a tokenized sequence into a syntactically correct source code. It can also be noticed that the pre-tokenization pipeline deals with indentation levels. Such information needs to be preserved while working with *Python* code since, unlike other programming languages, *Python* uses indentation as a part of the grammar definition. The importance of indentation is also reflected in the *tokenize* library that produces the `[IND]` and `[DED]`

[4] `https://docs.python.org/3/library/tokenize.html`

tokens only when the indentation level changes. Otherwise, the tokenized sequence would be over-saturated with *tabulator* characters. All the presented pre-tokenization steps are implemented by our `PythonTokenizer` class.

5.2.2 Training the Tokenizer

As soon as the tokenization algorithm and pre-tokenization steps are defined, it is essential to define a vocabulary. In other words, we need to train the tokenizer. To do so, we use an existing implementation of the *WordPiece* tokenizer (section 2.3.2) from the *tokenizers*[5] library. Thanks to utilizing the existing implementation, we need to provide only a few configuration options. These are:

- ***pre-tokenizer*** - an algorithm to split the input string into single-word tokens
- ***normalizer*** - normalization procedures to apply to the input text, including accents handling, text polishing, et cetera.
- ***decoder*** - an algorithm to be used for decoding a sequence of tokens into a string

Since we split the original tokens by a space character during the pre-tokenization (section 5.2.1), we can use the most simple *Whitespace*[6] pre-tokenizer. For normalization, we utilize the *NFD*[7] normalizer in conjunction with the *BertNormalizer*[8]. Thanks to that, we acquire tokens with no accents, all control characters removed, and all *Unicode* characters decomposed to a canonical form. It shall be noted that we do not perform lowercasing since *Python* is case sensitive language. Last but not least, we use a *WordPieceDecoder*[9] to transform the tokenized text into its original form.

[5]`https://huggingface.co/docs/tokenizers/python/latest/api/reference.html#tokenizers.models.WordPiece`

[6]`https://huggingface.co/docs/tokenizers/python/latest/api/reference.html#tokenizers.pre_tokenizers.Whitespace`

[7]`https://huggingface.co/docs/tokenizers/python/latest/api/reference.html#tokenizers.normalizers.NFD`

[8]`https://huggingface.co/docs/tokenizers/python/latest/api/reference.html#tokenizers.normalizers.BertNormalizer`

[9]`https://huggingface.co/docs/tokenizers/python/latest/components.html#decoders`

5.2.3 Vocabulary Size Selection

Vocabulary size is a crucial hyperparameter that affects the speed of training and the results achieved using a model. Smaller-sized vocabulary occupies less space in the *GPU*'s memory so that the *batch size* can be significantly larger. On the other hand, tiny vocabulary may reduce the *neural network*'s ability to understand the language. Thus, the vocabulary size must be carefully chosen.

Firstly, we analyze how many unique tokens are present in our dataset and how often they occur. If we work with a natural language text, we can tokenize the corpus on whitespaces and count the occurrences of all the individual tokens. However, such an approach to tokenization tends to yield too long and rare tokens when applied to a source code. Therefore, we tokenize the corpus using a *Python* lexical analyzer. The significant difference between these two approaches can be observed below:

```
whitespace tokenization:
    def test_func(par1): -> def, test_func(par1):

Python's lexical analyzer:
    def test_func(par1): -> def, test_func, (, par1, ), :
```

After running the token occurrence analysis, we need to filter out rare tokens. We choose a threshold of five occurrences that yields a vocabulary of approximately 8.8M tokens. Having such a colossal vocabulary is infeasible for our experiments since only the embedding matrix would consume an excessive portion of the GPU's memory. To get a better baseline, we try to filter out all the tokens with less than a hundred occurrences, resulting in a vocabulary of 393K tokens, which seems to be a very high number too. Therefore, we need to choose a different approach to choosing the vocabulary size.

To determine a suitable vocabulary size, we calculate several statistics for different tokenizers with vocabularies of 2K, 6K, 11K, and 25K tokens (table 5.3). Based on the statistics, we can understand how many whole-word tokens occur in the vocabulary and how many must be split into multiple subword tokens. Based on the presented experiment, we choose to work with a vocabulary of 25K tokens, which splits only less than 40% of the original tokens into multiple subword tokens.

vocab. size	# tokens	word starts [%]	word continuations [%]
25K	2.1B	62.5	37.5
11K	2.4B	55.0	45.0
6K	2.7B	48.9	51.1
2K	3.5B	37.1	62.9

Table 5.3: Statistics of tokenizers trained with different vocabulary sizes. The statistics were computed on a reduced dataset of 34K repositories. The third and fourth columns show how many tokens occurring in the dataset represent word starts and continuations. Word continuation is a token that starts with '##'.

5.3 Model

As stated in section 4.2, we use the *BART* model (section 2.2.4) as a base for our *CodeFormer*. In our version of the model, we use 12 *Transformer* layers with 16 *attention heads* in both an encoder and decoder. We also keep the original *hidden size* of 1024, and the same value is used for the number of *positional embeddings*. Such setup leads to a neural network model with 380M parameters.

To implement the model, we utilize *HuggingFace's* library called *transformers*[10]. The library provides an implementation of many models derived from the *Transformer* (section 2.2.2) architecture. The provided models are available for both *Tensorflow*[11] and *Pytorch*[12], from which we decided to use the latter one. Out of all the *BART* model implementations from the *transformers* library, we use *BartForConditionalGeneration*[13], which comes with an in-built language modeling head suitable for our pre-training task.

5.4 Pre-processing

In the previous sections, we described the choice of the data, tokenizer, and model. Based on those design choices, we can define how to pre-process

[10]`https://huggingface.co/transformers/`

[11]`https://pytorch.org/`

[12]`https://www.tensorflow.org/`

[13]`https://huggingface.co/transformers/model_doc/bart.html#transformers.BartForConditionalGeneration`

the data before being processed by the *CodeFormer*. Our pre-processing involves extracting code units representing training examples, tokenization, and efficient data storage. Furthermore, the input pipeline needs to add noise and pad or trim the sequences to the same length. The lastly mentioned steps are done on the fly during training, enabling us to change hyperparameters (for example, maximum sequence length) after generating a pre-processed training set. The whole pre-processing pipeline is depicted in figure 5.1. The most important steps - code unit selection and noise generation, are further described in the subsequent sections.

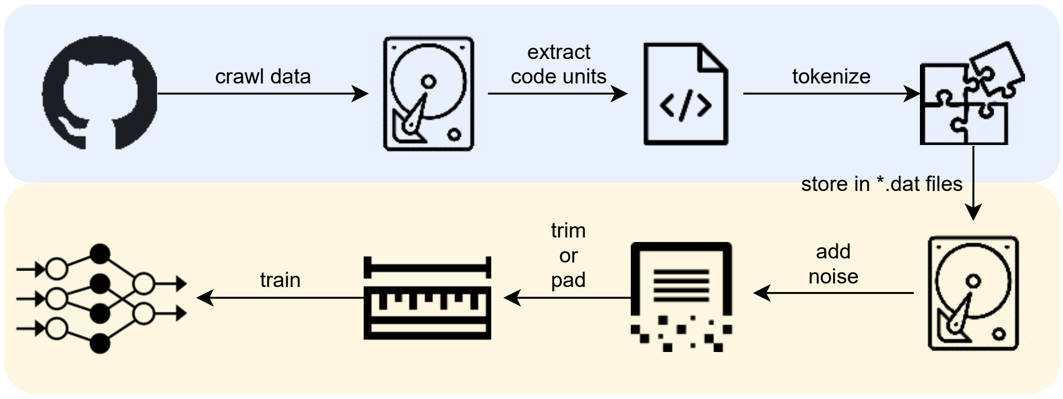


Figure 5.1: Visualization of data pre-processing steps. The area marked with blue color represents a pre-processing that is done before pre-training. The yellow part of the figure represents transformations done on the fly during pre-training.

5.4.1 Code Unit Selection

Since some source files can be very long, we need to split them into multiple shorter training examples - *code units*. In the context of our work, we consider each function to be a *code unit* and hence a self-standing training example. Furthermore, we also extract and treat methods as *code units*. However, unlike functions, methods need to be extracted with a class declaration. Otherwise, it will not be possible to distinguish between them, and the neutral network will not learn the syntactical difference (*self* as a first positional parameter).

For *code unit* extraction (depicted in figure 5.2), we employ Python's *ast* library, which can parse a source code and build an *abstract syntax* *tree* (AST). Afterward, we iterate over all nodes in the AST and search for `ast.ClassDef` and `ast.FunctionDef` nodes. In the case of the latter one, the whole subtree of the selected node is extracted, converted back into a source code, and added to the training dataset. In the case of the class definition node, the algorithm searches for underlying `ast.FunctionDef` nodes. Each such node is then extended with the class header, unparsed, and appended to the training set.

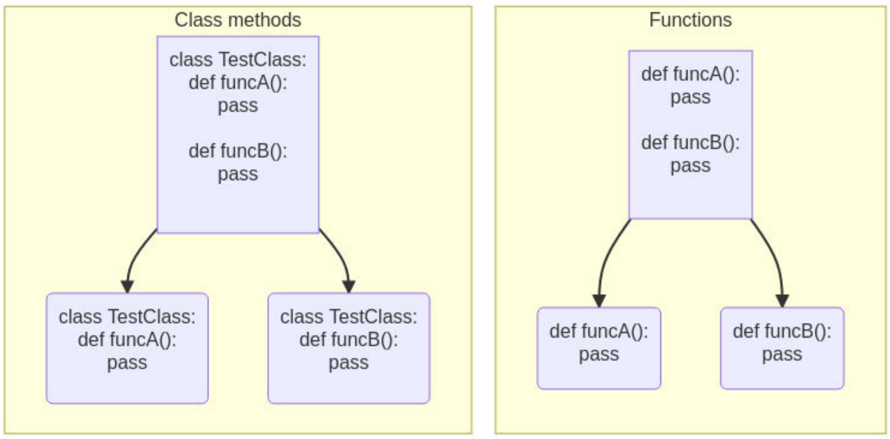

Class methods
class TestClass: def funcA(): pass def funcB(): pass
class TestClass: def funcA(): pass
class TestClass: def funcB(): pass
Functions
def funcA(): pass def funcB(): pass
def funcA(): pass
def funcB(): pass

Figure 5.2: Extraction of code units from a class (left-hand side) or a source file with plain functions (right-hand side).

5.4.2 Noise Generation

In order to generate a noisy input for the encoder part of the model, we follow the approach laid out in the original BART paper (see section 2.2.4 for more details). Namely, we choose *Text Infilling* and *Document Rotation* transformations. For the *Text Infilling*, we use *Poisson distribution* with $\lambda = 3$ to sample the length of a masked span. During this step, we replace 24% of tokens with `[MASK]` token and 6% of the tokens with a replacement, randomly chosen from the vocabulary. The *Document Rotation* (section 2.2.4) is applied to 50% of input sequences.

In addition to the transformations described above, we extend the noising scheme with a *Token Swap*, injecting additional syntactic or semantic errors into the input. We apply the *Token Swap* transformation to 5% of the tokens by randomly selecting two tokens and swapping their positions.

5.5 Pre-training Procedure

This section discusses the pre-training-related implementation and procedure. Firstly, the experimental setup section includes information about the data input pipeline, progress logging, checkpointing, and hyperparameter setup. In the next part, we discuss the obtained results.

5.5.1 Experimental Setup

Our pre-training procedure is implemented in *Python* using the *PyTorch* library. Furthermore, we employ the *HuggingFace Transformers* library that provides a verified implementation of the BART (section 2.2.4) model that we train using a training script described in the following paragraphs.

We conduct the whole experiment on a node of *MetaCentrum* (MC)[14] grid computing infrastructure. More specifically, we choose a computation node from a cluster **zia**, where we use two cores of *AMD EPYC 7662 (64 Core) 2.00 GHz* CPU and two *Nvidia A100* GPUs. The MC uses a PBS[15] task scheduler that allows us to acquire the aforementioned resources for a maximum of 24 hours. Afterward, a new job needs to be scheduled, and the training procedure has to be resumed.

Data Input Pipeline

In order to retrieve the pre-processed training data from a file system, we implement the `GithubDataset` class. The dataset implementation can obtain an example of the data based on a provided index via the `__getitem__()` method. Since the training data are chunked into multiple files, the dataset provider must determine where the requested example is located. Furthermore, the `GithubDataset` buffers the last accessed data file in memory so that the subsequent reads that fall into the same data file are served from the buffer directly. The buffering mechanism is a crucial feature for the speed of the whole data input pipeline. However, it heavily depends on a sequential sampling presumption. Therefore, the data must be pre-shuffled during the pre-processing.

The sequential training data sampling is realized by our implementation of `RestartableSequentialSampler` class inspired by *PyTorch's* implement-

[14]`https://metavo.metacentrum.cz/en/`

[15]`https://www.altair.com/pbs-professional/`

ation of `SequentialSampler`. Unlike the implementation provided by the *PyTorch*, our custom sampler can continue sampling from a point where it left off. Resuming the sampling seamlessly is a crucial requirement since each pre-training job can last up to 24 hours. Afterward, a new training job needs to be scheduled in the *MetaCentrum*, and the training shall continue from the latest checkpoint (for more details, see section 5.5.1).

Except for retrieving the training examples from a file system, the implementation of `GithubDataset` is also responsible for the pre-processing steps highlighted in yellow color in figure 5.1. More specifically, the dataset implementation alters the data with noise (by using `BARTNoiseGenerator` class). Furthermore, it pads or trims the sequences to match the defined sequence length and generates attention masks for the model. After all these steps, the `GithubDataset` yields a data point that can be directly passed to the model.

Progress Logging & Checkpointing

Besides the data input pipeline, the pre-training procedure requires logging and checkpointing. The logging is essential to monitor whether the loss decreases continuously, and the neural network learns as expected. On the other hand, checkpointing allows us to resume the training after the 24 hours long MetaCentrum training session or recover in case of any failure.

In our work, we choose to realize the logging using the *Weights & Biases*[16] service (`wandb` library for Python) that provides online visualization of the logged data. Furthermore, it offers automatic monitoring of parameter gradients, hyperparameter searches, artifact (models or datasets) versioning, et cetera. During the pre-training, we log the current learning rate and loss each 1024 training examples. Additionally, we store a neural network's prediction every 1000 batches to intuitively see how the neural network is improving in the denoising task.

Alongside the logging, we create a new checkpoint each 200K training examples. A checkpoint always contains all model parameters, the current state of an optimizer, and the number of processed training examples. When we need to restart the training, we find the newest checkpoint stored on the filesystem, load all the weights into the model, load the optimizer's state and continue sampling the training dataset from the step recorded in the checkpoint.

[16]`https://wandb.ai`

Hyperparameter Setup

To train our model, we use the Adam optimizer with an initial learning rate $\lambda = 10^{-5}$. Additionally, we employ a learning rate warmup for the first 500K steps, and then we apply a linear decay to zero. Such a setup is used to perform a single iteration ($\approx$ 220M examples) over the training set with a batch size of 64 and a maximum sequence length of 256 tokens.

Later on, we increase the maximum sequence length to 1024 tokens and start iterating over the dataset from the beginning to train higher positional embeddings. Using the increased sequence length, we train the network for additional 20M examples using a batch size of 8. Furthermore, we need to increase the learning rate so that the training can impact the embedding weights. Therefore, we set the learning rate to $\lambda = 9 \times 10^{-7}$ and apply the linear decay. The resulting learning rate function is depicted in figure 5.3.

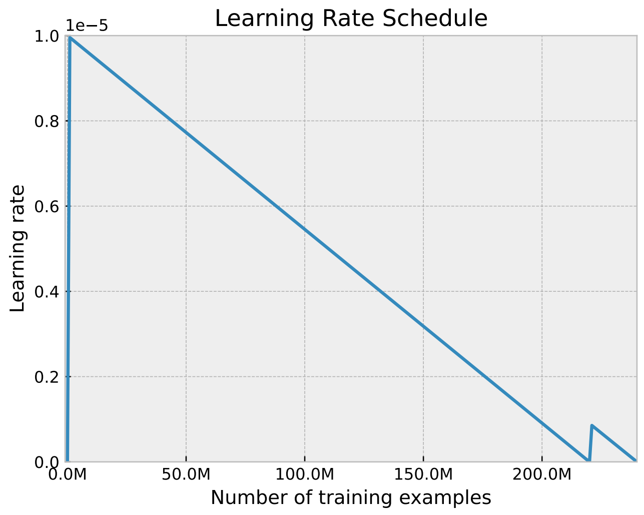


Figure 5.3: Learning rate schedule used for the pre-training. The learning rate schedule uses a warmup and linear decay.

5.5.2 Pre-training Results

The pre-training procedure in the previously described setup took approximately 1058 hours (1.5 months) on two *NVidia A100* GPUs. To track the pre-training progress, we monitor the cross-entropy loss during the whole training (figure 5.4). As one can see, the slope of the loss is steep during the first 50M steps, whereas it decreases very slowly during the later stages.

For a more human-readable evaluation of the model's output, refer to the appendix A, which lists a few examples of source code denoising using our resulting **CodeFormer** model.

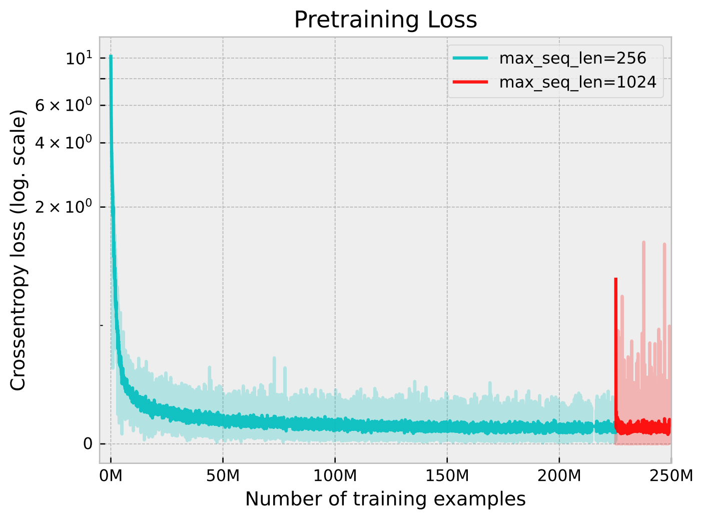


Figure 5.4: Smoothed progress of the cross-entropy loss during pre-training. The first 230M steps represent the pre-training with a maximum sequence length of 256, whereas the last 20M steps represent the training with a maximum sequence length of 1024 tokens.

In the end, we export the model using `model.save_pretrained()`, which results in storing the model's configuration together with its weights on the file system. In addition, we upload our resulting model to the Hugging Face's model repository[17] so that the model is easily accessible to everyone. The *CodeFormer* model can be loaded using the Python source code presented in Snippet 5.1. Following that point, we can load the pre-trained model and fine-tune it for various tasks such as code correction or generation. The latter usage is thoroughly discussed in the subsequent chapter.

```python
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

MODEL_NAME = "janpase97/codeformer-pretrained"
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForSeq2SeqLM.from_pretrained(MODEL_NAME)
```

Listing 5.1: Python source code used to load our pretrained model using transformers library.

[17]`https://huggingface.co/janpase97/codeformer-pretrained`

6 Generating Source Code Based on Natural Language Descriptions

This chapter focuses on applying the pre-trained *CodeFormer* model to the code generation task. Firstly, the chapter discusses the required changes in the model's architecture. Subsequently, we describe the datasets used for fine-tuning, including their pre-processing. At the end of this chapter, we describe the concluded experiments and discuss the achieved results.

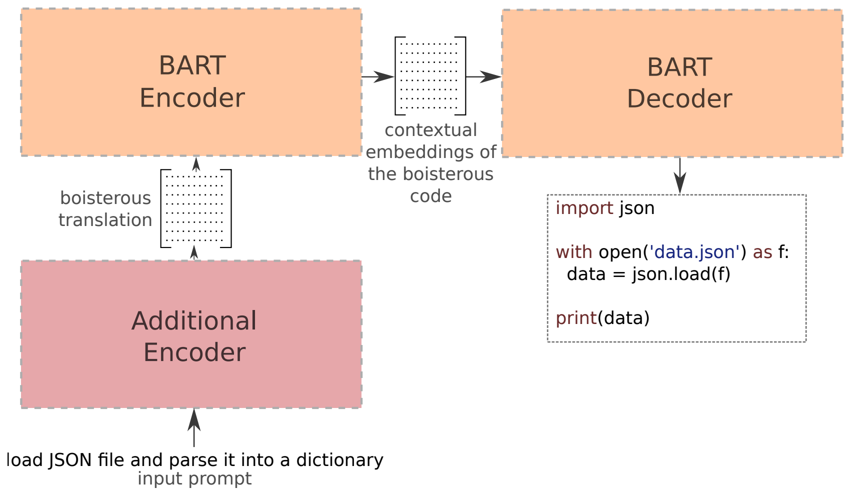
6.1 Model

As discussed in section 4.2, we follow the BART's machine translation architecture (section 2.2.4) to adapt the *CodeFormer* model to the English to Python translation task. This approach extends the model with an additional encoder responsible for translating a text in a source language into a noisy text in a target language. In our case, we expect the additional encoder to consume NL texts and produce boisterous source codes, which are then denoised by the *CodeFormer*. Such architecture is visualized in figure 6.1.

Despite not being visualized in the figure, there is an extra linear layer (`torch.nn.Linear`) in between the *CodeFormer* and the additional encoder. Its purpose is to perform a conversion between dimensions of the two separate models. This is crucial, especially when using existing pre-trained models whose dimensionality cannot be changed. In our experiments, we employ both randomly initialized and pre-trained encoders, whose choice is further discussed in the next section.

6.1.1 Additional Encoder Selection

We consider two different pre-trained encoders to be employed as additional encoders for our code generation model. The first one is the *CodeBERT* model (section 3.4.1), and the second is the *MQDD* model (section


BART Encoder
contextual embeddings of the boisterous code
BART Decoder
boisterous translation
Additional Encoder
load JSON file and parse it into a dictionary
input prompt

```
import json

with open('data.json') as f:
  data = json.load(f)

print(data)
```

Figure 6.1: Architecture of the CodeFormer model adjusted for conditional generation task.

3.4.2). Both of these models are pre-trained on a large corpus of bilingual NL-PL pairs, and therefore, they might significantly reduce the time necessary for our model to converge. It is difficult to predict which of these models will give better results. Nevertheless, based on the pre-training datasets used by the *CodeBERT* and *MQDD*, we presume that the *CodeBERT* can improve results when applied to datasets based on GitHub, whereas the *MQDD* can show better results on the StackOverflow-based datasets. Besides the pre-trained models, we also experiment with utilizing a randomly initialized encoder of a smaller size.

6.2 Dataset

Unlike in the case of the pre-training, a monolingual dataset is not suitable for generating source code based on an NL description. To train such a generator, we need a parallel corpus that encompasses pairs of code and corresponding descriptions. As discussed earlier, there are several suitable datasets available. In section 4.3, we chose to employ the CodeSearchNet (section 3.3.2) and CoNaLa (section 3.3.3) datasets accompanied by our own StackOverflow-based dataset.

The rest of this section is structured as follows. Firstly, we briefly introduce the StackExchange and Stackoverflow platforms, a joint base for two of the datasets we utilize. Afterward, we describe the whole process of acquiring and preparing our custom *StackOverflow*-based dataset. In the end, we briefly set out the pre-processing strategy for the CoNaLa and CodeSearch-Net datasets.

6.2.1 StackExchange & StackOverflow

With its 431.8M monthly visits, the StackExchange is probably the largest community question-answering (Q&A) platform in the world. It consists of 173 subpages, each designated for a different topic. The subpages are heavily dominated by the StackOverflow that programmers use to discuss their problems.

Each thread on the StackOverflow consists of a question made up of a title and body describing the problem to be discussed. Additionally, the author labels a question with a set of tags, defining the question's topic. Other users can then post answers to the stated question and receive upvotes/downvotes that reflect the quality of the answer. If one of the answers is good enough to solve the author's problem, the author marks the question as accepted.

All this information with a lot of additional metadata can be acquired from `www.archive.org/details/stackexchange`. Especially for the StackOverflow, one can select from the following data:

• **badges**	(304.6 MB)	- user's honors data
• **comments**	(4.9 GB)	- comments to different answers
• **post history**	(30.3 GB)	- history of all the posts
• **post links**	(105.6 MB)	- links between posts (e.g., duplicates)
• **posts**	(17.2 GB)	- set of all questions and answers
• **tags**	(873.1 KB)	- set of all available tags
• **users**	(821.9 MB)	- registered users and their public data
• **votes**	(1.3 GB)	- upvotes and downvotes related to posts

Within each of the files mentioned above, there is an XML structure described by a `readme.txt`[1] file stored alongside the datafiles.

[1] `https://ia800107.us.archive.org/27/items/stackexchange/readme.txt`

6.2.2 Custom StackOverflow Dataset

As stated previously, we decided to create a novel **Stack Overflow Code Generation Dataset** (SOCGD) to complement the existing datasets used in our experiments. The rest of this section describes how we can obtain high-quality NL-PL pairs from StackOverflow questions and accepted answers obtained from an XML dump downloaded from `www.archive.org`.

Assembling the Dataset

Since the downloaded data are stored in an XML format, we use the `BeautifulSoup` library that parses the XML. Firstly, we need to filter only the Python-related questions. To do so, we use the `PostTypeId` and `Tags` attributes associated with every post in the *StackOverflow* data export. The first mentioned attribute can distinguish between a question and an answer, whereas the latter is utilized to filter out the questions containing `python` or `python-3.x` tag. The filtered questions are then stored in a MongoDB (a NoSQL database used to store structured data) in the following format:

```
__id:       <id of the question>
title:      <question's title>
text:       <text extracted from the question's body>
answer_id: <id of the related accepted answer>
tags:       <list of all the tags assigned to the question>
```

Afterward, we apply a similar approach to extract all answers marked as accepted. To extract the answers, we only consider the posts with the attribute `PostTypeId = 2`. Furthermore, we query the MongoDB for the answer's parent question based on the identifier stored in the answer's `ParentId` field. If no record is found, the answer can be skipped immediately. Otherwise, an additional check verifies whether the identifier (attribute `Id` in the XML representation of the answer) matches the `parent_question.answer_id` from the database. All answers that pass through such a filter represent accepted answers and are stored in the MongoDB following the structure described below:

```
__id:       <id of the answer>
parent_id: <id of the parent question>
code:       <source code extracted from the answer's body>
```

In the end, we export each of the question-answer pairs into a simple text file. We export a title and text of a question, source code extracted from an answer, and an identifier of the StackOverflow thread, required by the license conditions. All those fields are separated by a `<SPLIT>` delimiter.

Pre-processing

In the previous section, we intentionally omitted information related to data pre-processing. The pre-processing pipeline is split into three distinct parts. The first part involves a simple pre-processing while assembling the dataset. In the second phase, we prepare tensors that can be processed by the neural network and are model-specific (due to the tokenization step). The last stage is then executed directly during the training and involves generating attention masks, token type ids, et. cetera. In the following paragraphs, all those stages (figure 6.2) are described in detail.

As stated above, the first stage of pre-processing is done during the dataset assembly process, right before creating MongoDB records. Data transformations vary depending on whether we process a question or an answer. For each question (both a title and body), we need to strip all new line characters, remove the HTML markup, and possible code snippets. On the contrary, we need to extract the code snippets and leave out any natural language text for the answers. For most of these transformations, we utilize the `BeautifulSoup` library.

In the second pre-processing stage, we work with the textual export of the dataset to tokenize the inputs and store them as tensors in multiple data files. Firstly, we pre-tokenize the source code using our custom `PythonTokenizer` (section 5.2.1) and tokenize the code using a WordPiece tokenizer (section 5.2.2). Furthermore, we tokenize the textual part (body and title) using two different tokenizers corresponding to the *MQDD* (section 3.4.2) or *CodeBERT* model (section 3.4.1). Before the tokenization, we concatenate both the question's title and body separated using `[SEP]` token. Finally, we wrap the textual part into `[CLS]`...`[SEP]` tokens and the code part into `[START]`...`[END]` tokens. Then, we trim or pad the examples to a length of 256 tokens and store them into multiple `*.dat` files, 1000 examples each. Finally, we split the `*.dat` files into train, dev, and test splits in 90:5:5 ratio.

The last part of the pre-processing happens directly during the training. From the last pre-processing stage, we acquire a tensor with tokenized NL

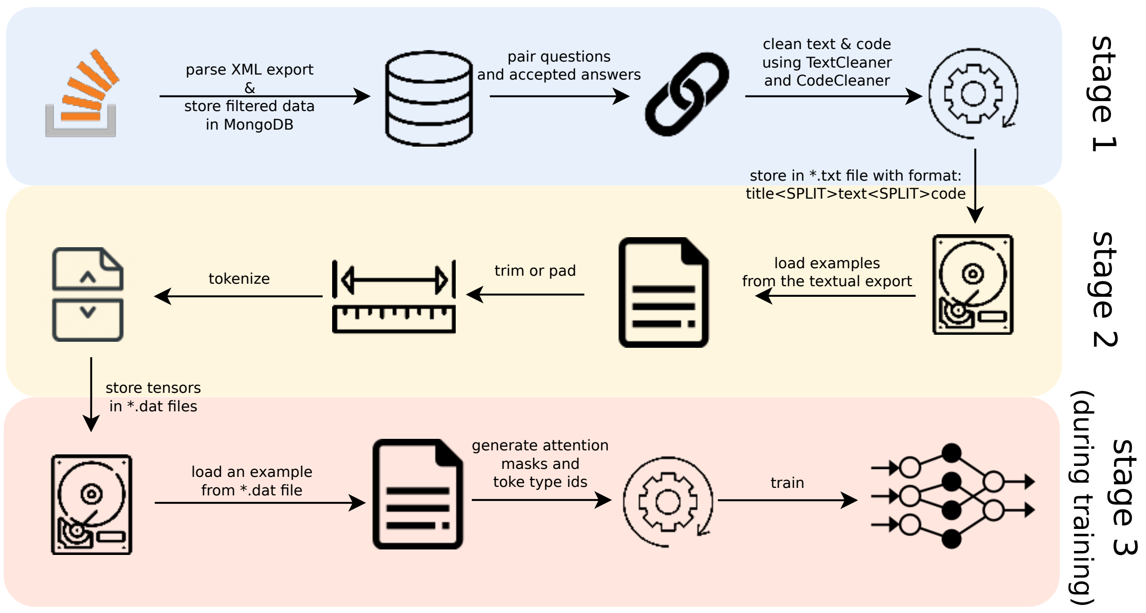

parse XML export & store filtered data in MongoDB
pair questions and accepted answers
clean text & code using TextCleaner and CodeCleaner
stage 1
store in *.txt file with format: title<SPLIT>text<SPLIT>code
load examples from the textual export
trim or pad
tokenize
stage 2
store tensors in *.dat files
load an example from *.dat file
generate attention masks and toke type ids
train
stage 3 (during training)

Figure 6.2: Visualization of a pre-processing pipeline used for fine-tuning datasets.

query (x_i) and a tensor with tokenized target source code (y_i). From those two tensors, we need to generate the rest of the necessary inputs using the following rules:

- **encoder input** $= x_i$ (format: `[CLS] <title> [SEP] <body> [SEP]`)
- **encoder attention mask** - $xm_{ij} = \begin{cases} 1 & \text{if } x_{ij} = \text{[PAD]} \\ 0 & \text{if } x_{ij} \neq \text{[PAD]} \end{cases}$
  (it is allowed to attend to all positions except the padding)
- **encoder token type ids** - $xt_{ii} = 0$ (additional encoders require the NL text to have token type id equal to 0, whereas the source code needs to have 1 $\Rightarrow$ all zeros, since only NL is present in the query)
- **decoder input** - $yi_i = y_i[: -1]$ (whole desired source code except the last `[END]` token)
- **decoder attention mask** - $ym_{ij} = \begin{cases} 1 & \text{if } yi_{ij} = \text{[PAD]} \\ 0 & \text{if } yi_{ij} \neq \text{[PAD]} \end{cases}$
  (it is allowed to attend to all positions except the padding)
- **labels** - $l_i = y_i[1 :]$ (whole desired source code except the `[START]` token at the beginning)

Dataset Size

The resulting *SOCGD* dataset created from StackOverflow questions and accepted answers contains about 400K examples. Detailed information about the dataset's size can be found in table 6.1.

SOCGD Dataset	
Split	*Size*
train	364K
dev	20K
test	20K

Table 6.1: Size of different splits of our custom SOCGD dataset.

6.2.3 CoNaLa Dataset

The second dataset we utilize in our work is the CoNaLa dataset (section 3.3.3). The data can be downloaded from the author's website `https://conala-corpus.github.io` in *JSON lines* format containing the following information:

- **question_id** - identifier of the original StackOverflow question
- **intent** - NL description of the code snippet (extracted from StackOverflow)
- **rewritten_intent** - the intent rewritten by a human annotator (available only for the small train split)
- **snippet** - code snippet corresponding to the intent

Since the dataset is already assembled and shipped in a very user-friendly way, the preprocessing of the dataset employs only the stages two and three described in section 6.2.2. It means that we only tokenize the textual export and convert it to tensors. Afterward, we generate the masks and token type ids directly during the training. Since the data input pipeline is the same as in the case of our *SOCGD* dataset, we can seamlessly choose which of the dataset is used for the training without the need to write dataset-specific code. Moreover, we can combine both datasets to create a more extensive training set. For detailed information about the dataset size see table 6.2.

CoNaLa Dataset	
Split	*Size*
mined	593K
train	2 379
test	500

Table 6.2: Size of different splits of the CoNaLa [50] dataset.

6.2.4 CodeSearchNet Dataset

The last dataset we use in the fine-tuning is the CodeSearchNet (section 3.3.2), which originates from GitHub. Due to its origin, it might seem that our model should deal with that dataset significantly better than with StackOverflow-based datasets (the *CodeFormer* was pre-trained on a similar GitHub-based dataset). However, we must consider that the NL queries are extracted from high-level source code docstring, which might not reflect how users phrase their queries. On the other hand, the source codes from GitHub usually represent longer pieces of code closer to the model's desired outcome.

The CodeSearchNet dataset can be obtained via a simple script from the authors' GitHub[2] repository. The resulting dataset stored on a local FS consists of multiple `*.tar.gz` files, each containing the examples in *JSON lines* format. All the records consist of the following information:

- *__code__ - source code of a single function,
- **code_tokens** - tokenized source code,
- *__docstring__ - description of the function extracted from an original documentation comment,
- **docstring_tokens** - tokenized docstring,
- **func_name** - name of the function from which the example is extracted,
- **language** - language of the source code,
- **original_string** - source code including comment from which the sample was extracted,
- **partition** - categorization of the example into a train/dev/test split,

[2]`https://github.com/github/CodeSearchNet`

- **path** - path to a source code file from which the example was extracted (from the repository's root),
- **repo** - repository from which the example was obtained,
- **sha** - checksum of the example,
- **url** - URL address that points to the source file at GitHub.

From all the available information, we use only those marked with *. Namely, we employ the *code* and *docstring*, which we pre-process the same way as the two datasets mentioned previously. Information about the size of the final CodeSearchNet dataset is present in table 6.3.

CodeSearchNet Dataset	
Split	*Size*
train	412K
dev	23K
test	22K

Table 6.3: Size of different splits of the CodeSearchNet [49] dataset.

6.3 Training

In the preceding sections, we have introduced the model and datasets that we use to generate source codes. In this section, we put this information together to train the extended *CodeFormer* model that can produce meaningful source codes based on English prompts. Firstly, we give a throughout description of the executed experiments, and afterward, we present the achieved results.

6.3.1 Experimental Setup

The general framework of fine-tuning experiments is quite similar to the pre-training setup in section 5.5.1. We use the same libraries - *HuggingFace's Transformers, PyTorch*, and even the same hardware - a single MetaCentrum node from cluster *zia* with two *NVidia A100 GPUs*. Due to the significant similarity with the pre-training, we focus mainly on the differences. Unless otherwise stated, the setup is identical.

Data Input Pipeline

Similar to the pre-training, we implement a custom dataset wrapper that provides access to an example identified by an index (the wrapper implements `__getitem__()`). The implementation of the dataset wrapper is realized using `CodeGenerationDataset` class that takes a list of `*.tar` files as inputs. In the constructor, it unpacks the `*.tar` files into temporary storage and constructs a list of all available `*.dat` files that contain tensors with tokenized inputs (see section 6.2.2). Afterward, the data files are shuffled, and the dataset is ready to be sampled. A significant difference between the implementation of the pre-training and fine-tuning dataset wrappers is that the one designated for fine-tuning does not employ any noising scheme. Otherwise, the basic mechanism of example serving is identical.

Progress Logging & Checkpointing

Similar to the pretraining, we employ the *Weights & Biases* service to log training progress every 1024 steps. In addition, we save a checkpoint of the whole model every 4096 steps to allow the training to be resumed. Furthermore, we print out a sample prediction produced by the model every 500 batches to intuitively observe the training progress.

Metrics

To assess the results achieved by our model, we utilize two different metrics - a standard BLEU score (section 2.5.2) and our custom metric that focuses on producing valid Python source code.

Our custom metric is called *Python Validity* (PV). It represents a percentage of syntactically valid source codes produced by a model. The validity of a given source code can be effortlessly verified using Python's built-in library function `ast.tokenize()`. The function either returns a parse tree of the given source code or throws an exception, indicating an invalid code. The PV metric can be calculated using equation 6.1, where c_v is the number of valid Python source codes produced by a model and c_t is a size of a test set.

$$PV = \frac{c_v}{c_t} \tag{6.1}$$

Hyperparameters

In the fine-tuning experiments, we employ the *Adam* optimizer with an initial *learning rate* $\lambda = 10^{-5}$. Similar to the pre-training, we employ a *learning rate warmup* for the first 10K examples, and then we apply a *linear decay* to zero. The resulting learning rate schedule is depicted in figure 6.3.

Furthermore, in all of our experiments, we use the *batch size* of 32 examples and *L2 normalization* with regularization factor set to 0.03. For the linear layer placed in between the *additional encoder* and the *CodeFormer*, we use a *dropout* probability of 0.2, and for the rest of the *CodeFormer* model, we set the *dropouts* to 0.12. Additionally, all experiments limit the maximum sequence length of the inputs and outputs to 256 subword tokens. Last but not least, for generating outputs, we use an implementation of the *beam search* decoding algorithm (section 2.4.2) with the *beam width* set to three.

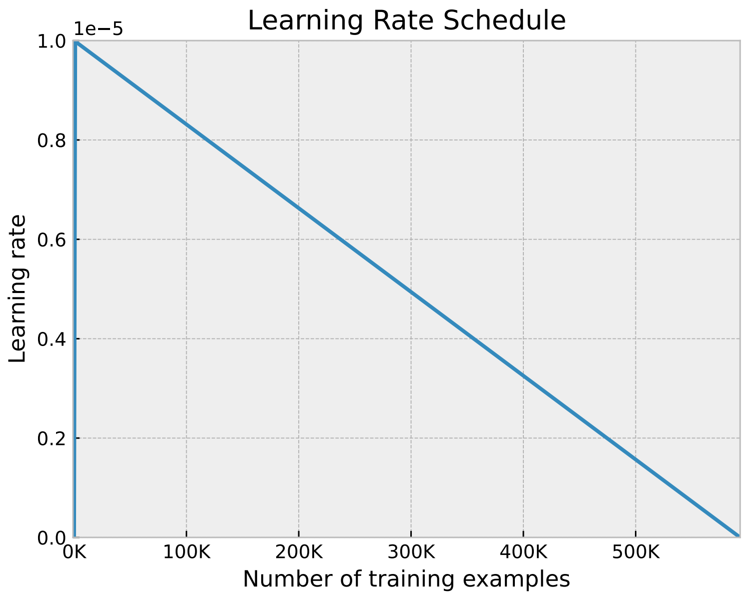


Figure 6.3: Learning rate schedule used for the fine-tuning. The learning rate schedule uses a warmup and linear decay.

CodeFormer Experiments

We conduct numerous experiments to determine the best possible approach to source code generation using our pre-trained *CodeFormer* model.

Our initial experiments confirmed the findings of the original study (section 2.2.4), which say that the BART architecture for machine translation is prone to overfitting. More specifically, our experiments have shown that if we use the pre-trained additional encoder as is, the encoder degrades during the first several thousand steps into a state where it produces the same outputs, not considering the NL prompt.

We presume that the authors of the BART model (section 2.2.4) employed a randomly initialized additional encoder to mitigate the behavior we described above. We further investigate this topic by introducing experiments with a pre-trained additional encoder, whose last four layers are initialized randomly. Such a setup shall enable the pre-trained additional encoder to preserve a lot of previously gained knowledge and mitigate the overfitting. Generally, we conduct the following types of experiments:

- using a randomly initialized additional encoder made up of six Transformer encoder layers (section 2.2.2)
- using a pre-trained additional encoder with the last four layers initialized randomly

For each of the experiments above, we train a separate model for each dataset and perform ten training epochs. In all of the experiments, we freeze all the weights of the base CodeFormer model except for positional embeddings and projection weights of the first attention layer during the first half of the training. Freezing the previously mentioned weights helps to prevent the encoder from degrading. The results of our experiments and a summary of our findings are presented in the following section.

6.4 Results and Discussion

This section presents the results of the experiments introduced in the preceding section. In addition to experiments with our *CodeFormer* model, we train the *CodeT5* (section 3.4.4) model for each dataset to compare our work with another source code-oriented representative of modern generative models. Besides, we compare our results with the achievements of other related papers, whose results we only overtake without replicating them. The results on the *CoNaLa*, *CodeSearchNet*, and *SOCGD* datasets are summarized in tables 6.4, 6.5, and 6.6, respectively. The results are further discussed in the subsequent paragraphs.

6.4.1 CoNaLa Results

The results in table 6.4 suggest that our *CodeFormer* model does not surpass the results of other generative models for source code. We presume that this might be caused by the fact that the *CodeFormer* model is pre-trained on GitHub data, where every training example represents a larger piece of code, including a function or class header. On the contrary, the *CoNaLa* examples extracted from Stack Overflow are often as short as one line of code. This significant discrepancy between the pre-training and fine-tuning data could lead to an observed impairment in results compared to competing models.

Model	BLEU	PV
CodeFormer + random encoder	9.25	96.37
CodeFormer + MQDD	21.51	97.58
CodeFormer + CodeBERT	6.41	**99.19**
CodeT5	7.89	3.83
Baseline Seq2Seq [50] †	14.26	-
TranX [62] †	24.30	-
TranX + BERT [67] †	**34.20**	-

Table 6.4: Results of Python source code generation experiments conducted on the CoNaLa dataset (section 3.3.3). The results are stated in the BLEU score (section 2.5.2) and PV (section 6.3.1). The PV is not available for the experiments whose results are overtaken from the referenced literature and are marked with the † sign.

Nevertheless, an interesting observation in table 6.4 is the significant gap in results achieved using *CodeFormer* with *MQDD* as the additional encoder over the *CodeBERT* or random additional encoder variant. We believe that this is implied by the *MQDD*'s pre-training on a StackOverflow-based dataset. Thanks to that, the *MQDD* can provide the *CodeFormer* model with a better understanding of the NL prompts extracted from StackOverflow questions.

6.4.2 CodeSearchNet Results

Despite a partial failure on the *CoNaLa* dataset, our *CodeFormer* model sets a new state-of-the-art in code generation on the CodeSearchNet dataset (see table 6.5). Our best model (*CodeFormer with CodeBERT*) achieves

46.12 BLEU score, representing an improvement of ≈13.8 BLEU over the former state-of-the-art. Furthermore, the source codes produced by the model are syntactically valid in almost 80%, which is also an excellent result.

Such success is probably a result of a perfect alignment between the pre-training domain of both the *CodeFormer* and *CodeBERT* models with the resulting task. On the other hand, we see the *MQDD* variant reporting significantly worse results. We attribute this phenomenon to the worse generalization capabilities of the *MQDD* model, which are also reported in the original study (section 3.4.2). Despite this, even the use of the *MQDD* model brings a significant improvement over the variant that uses a randomly initialized encoder. It shows that using a pre-trained additional encoder makes sense, although the model is more prone to encoder erosion (section 6.4.4).

Model	**BLEU**	**PV**
CodeFormer + random encoder	36.35	**88.15**
CodeFormer + MQDD	39.67	71.47
CodeFormer + CodeBERT	**46.12**	79.97
CodeBERT [68] †	4.06	-
PLBART [68] †	4.89	-
BM25 + PLBART [68] †	6.99	-
CodeT5	16.74	9.57
GPT-2 (fine-tuned for code) [69] †	22.00	-
REDCODER-EXT [68] †	24.43	-
TranX + API knowledge [64] †	32.26	-

Table 6.5: Results of Python source code generation experiments conducted on the CodeSearchNet dataset (section 3.3.2). The results are stated in the BLEU score (section 2.5.2) and PV (section 6.3.1). The PV is not available for the experiments whose results are overtaken from the referenced literature and are marked with the † sign.

6.4.3 SOCGD Results Discussion

The results in Python source code generation on our *StackOverflow Code Generation Dataset* (SOCGD) are summarized in table 6.6. With our best model, we achieved a BLEU score of 47.68, which roughly matches our state-of-the-art result on the CodeSearchNet dataset. Since our *SOCGD* dataset is new, we can compare our results only with the *CodeT5* that we significantly outperformed.

However, like in the case of the *CodeSearchNet* dataset, we see a significant drop in the BLEU score achieved when using the *MQDD* as an additional encoder. On the *SOCGD* dataset, the results of the *MQDD* variant are even worse than in the case of the variant with random initialization. This is surprising, given that *MQDD* is pre-trained on the StackOverflow. However, this can also be explained by poor generalization capabilities and differences in the pre-training dataset. While *CodeBERT* is pre-trained on code documentation comments in six programming languages, including Python, the *MQDD* model was pre-trained on general questions found on the StackOverflow. Therefore, *CodeBERT* might be able to better understand questions that are closely related to Python. In addition, the authors of the *MQDD* model (section 3.4.2) describe that if the model is applied to tasks other than duplicate question detection, it may suffer from an effect called *negative transfer*.

Model	**BLEU**	**PV**
CodeFormer + random encoder	47.16	70.73
CodeFormer + MQDD	21.18	19.35
CodeFormer + CodeBERT	**47.68**	**70.74**
CodeT5	8.67	14.21

Table 6.6: Results of Python source code generation experiments conducted on the Stack Overflow Code Generation dataset (section 6.2.2). The results are stated in the BLEU score (section 2.5.2) and PV (section 6.3.1).

6.4.4 Encoder Erosion Problem

As stated in section 6.3.1, our experiments have shown that using a pre-trained additional encoder can lead to an encoder erosion, which results in producing the same outputs no matter the NL prompt. We speculate that the authors of the *BART* (section 2.2.4) faced the same problems when adapting *BART* for the MT task, and therefore, they chose to train a randomly initialized encoder instead.

We went a little further in our research and tried to find other ways to mitigate the problem of encoder erosion. Our experiments suggest that traditional regularization techniques such as *L2 regularization* and *dropout* do not prevent the encoder from degrading. The strategy of using a pre-trained additional encoder and freezing most of the BART model parameters in the first half of the training, as outlined in chapter 6.3.1, was also unsuccessful when applied alone.

However, we found out that freezing *BART*'s weights for the first half of the training combined with a pre-trained encoder (such as *CodeBERT* or *MQDD*), whose last *N* layers are initialized randomly, prevents the additional encoder from degrading. Moreover, such a configuration significantly increases the achieved BLEU score compared to using a randomly initialized encoder.

In future work, our findings need to be verified outside the domain of this thesis. The verification can be done by applying our approach to machine translation from Romanian into English, as in the case of the original paper introducing the *BART* model. Furthermore, future work can follow our approach and try to discover other techniques, further improving the usage of pre-trained additional encoders in the *BART*'s MT setup.

6.4.5 Results of the CodeT5 Model

The results presented above show that our *CodeFormer* model significantly outperforms the *CodeT5* (section 3.4.4) on all evaluation datasets. This outcome is expected since our model, unlike the *CodeT5*, specializes in Python programming language and is designed specifically for source code generation tasks. However, since the original study reported a 41.48 BLEU score in Java source code generation, we expected the *CodeT5* model to have significantly better results than measured.

We conjecture that the lower BLEU score we observe can be caused by the *CodeT5* struggling to learn the correct indentation rules required by Python's grammar. To the best of our knowledge, the original study does not consider the indentation in the *CodeT5*'s training, which we perceive as an essential aspect of Python source code generation. For example, if we replace *tab* and *newline* characters from the following source code snippets, we acquire the same sequence of tokens despite both source codes are different.

```python
a = 1
for i in range(10):
  a *= i
  return a
```

```python
a = 1
for i in range(10):
  a *= i
return a
```

7 Conclusion

This work aims to train a modern generative neural network model that can generate source codes based on descriptions in natural language. Therefore, the first part of the thesis introduces the reader to the field of generative models and subsequently to the existing methods for applying artificial intelligence in the software engineering domain.

Subsequently, the work focuses on training a source code generator based on the BART architecture. The work first pre-trains the model on a source code denoising objective, which requires a massive training corpus of source codes. Therefore, we automatically crawl GitHub repositories, obtaining 230M training examples in Python. After more than 1000 hours of training in the MetaCentrum cluster, we obtained a CodeFormer model that can repair artificially noised source codes very reliably. In addition to its intended use for code generation, such a model can be applied to various other tasks in the source code processing domain. An example is the use of our model to fix buggy source codes automatically.

We then further extend the pre-trained CodeFormer model with an additional natural language encoder, adapting the BART architecture to the task of machine translation from English to Python. We then fine-tune the modified CodeFormer on several datasets, including our novel SOCGD parallel corpus. The obtained results of 46.1 and 47.7 BLEU on SOCGD and CodeSearchNet datasets, respectively, are encouraging and show that we could integrate the resulting model into IDEs, helping the developer find the right solutions for their problems.

This work can be followed up by further research of other applications of our pre-trained CodeFormer model, such as code migration or code fixing. Moreover, it is possible to extend our approach by training a joint source code generator for multiple programming languages. Last but not least, we see great potential in building on our findings regarding the prevention of additional encoder erosion that occurs when using the BART architecture for machine translation.

All outputs of this work, including trained models and newly created datasets, can be obtained from our GitHub repository: `https://github.com/janpasek97/CodeFormer`.

List of Abbreviations

AI - artificial intelligence

API - application programming interface

AST - abstract syntax tree

BPE - byte pair encoding

CNN - convolution neural network

FS - file system

GRU - gated recurrent unit

IDE - integrated development environment

LCS - longest common subsequence

LM - language model

LSTM - long short-term memory

MC - MetaCentrum

MLM - masked language modeling

MT - machine translation

NL - natural language

NLP - natural language processing

NN - neural network

OOV - out of vocabulary

PL - programming language

PV - Python Validity

Q&A - question answering

RNN - recurrent neural network

RTD - replaced token detection

SOCGD - Stack Overflow Code Generation Dataset

URL - unified resource locator

Bibliography

[1] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning.* Massachusetts, MA, USA: The MIT Press, 2012.

[2] A. Y. Ng and M. I. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes," in *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, ser. NIPS'01. Cambridge, MA, USA: MIT Press, 2001, p. 841–848.

[3] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. USA: Prentice Hall PTR, 2000.

[4] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, no. null, p. 1137–1155, mar 2003.

[5] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, Mar 2020. [Online]. Available: http://dx.doi.org/10.1016/j.physd.2019.132306

[6] P. Goyal, S. Pandey, and K. Jain, "Deep learning for natural language processing," in *Apress*, 2018.

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[8] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[10] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," 2014.

[11] L. Atlas, T. Homma, and R. Marks, "An artificial neural network for spatio-temporal bipolar patterns: Application to phoneme classification," *Neural Information Processing Systems*, pp. 31–40, 01 1988.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[13] Q. Liu, M. J. Kusner, and P. Blunsom, “A survey on contextual embeddings,” *CoRR*, vol. abs/2003.07278, 2020. [Online]. Available: https://arxiv.org/abs/2003.07278

[14] A. Galassi, M. Lippi, and P. Torroni, “Attention, please! A critical review of neural attention models in natural language processing,” *CoRR*, vol. abs/1902.02181, 2019. [Online]. Available: http://arxiv.org/abs/1902.02181

[15] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[16] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” 2016.

[17] A. Radford and K. Narasimhan, “Improving language understanding by generative pre-training,” 2018.

[18] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, and D. Amodei, “Language models are few-shot learners,” 05 2020.

[19] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 7871–7880. [Online]. Available: https://aclanthology.org/2020.acl-main.703

[20] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” 2020.

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[22] P. Gage, “A new algorithm for data compression,” *C Users J.*, vol. 12, no. 2, p. 23–38, feb 1994.

[23] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. R. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. S. Corrado, M. Hughes, and J. Dean, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *ArXiv*, vol. abs/1609.08144, 2016.

[24] M. Schuster and K. Nakajima, “Japanese and korean voice search,” in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2012, pp. 5149–5152.

[25] T. Kudo and J. Richardson, “SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations.* Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 66–71. [Online]. Available: https://aclanthology.org/D18-2012

[26] T. Kudo, “Subword regularization: Improving neural network translation models with multiple subword candidates,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 66–75. [Online]. Available: https://aclanthology.org/P18-1007

[27] Z.-w. Xu, F. Liu, and Y.-x. Li, “The research on accuracy optimization of beam search algorithm,” in *2006 7th International Conference on Computer-Aided Industrial Design and Conceptual Design*, 2006, pp. 1–4.

[28] A. Holtzman, J. Buys, M. Forbes, and Y. Choi, “The curious case of neural text degeneration,” *CoRR*, vol. abs/1904.09751, 2019. [Online]. Available: http://arxiv.org/abs/1904.09751

[29] K. P. Murphy, *Machine learning: A probabilistic perspective.* The MIT Press, 2012.

[30] C. Campagnola, “Perplexity in language models,” Dec 2021. [Online]. Available: https://towardsdatascience.com/perplexity-in-language-models-87a196019a94

[31] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th*

Annual Meeting of the Association for Computational Linguistics. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[32] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out.* Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013

[33] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, 2000, pp. 39–48.

[34] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with BERT," *CoRR*, vol. abs/1904.09675, 2019. [Online]. Available: http://arxiv.org/abs/1904.09675

[35] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[36] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 837–847.

[37] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, jul 2018. [Online]. Available: https://doi.org/10.1145/3212695

[38] M. Arthur, "Automatic source code documentation using code summarization technique of nlp," *Procedia Computer Science*, vol. 171, pp. 2522–2531, 01 2020.

[39] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *CoRR*, vol. abs/2002.08155, 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[40] A. V. Miceli Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," in *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers).* Taipei, Taiwan: Asian Federation of Natural Language Processing, Nov. 2017, pp. 314–319. [Online]. Available: https://aclanthology.org/I17-2053

[41] K. Aggarwal, M. Salameh, and A. Hindle, "Using machine translation for converting python 2 to python 3 code," 10 2015.

[42] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming Software*, ser. Onward! 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 173–184. [Online]. Available: https://doi.org/10.1145/2661136.2661148

[43] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[44] J. Li, Y. Wang, I. King, and M. R. Lyu, "Code completion with neural attention and pointer networks," *CoRR*, vol. abs/1711.09573, 2017. [Online]. Available: http://arxiv.org/abs/1711.09573

[45] N. Jiang, T. Lutellier, and L. Tan, "CURE: code-aware neural machine translation for automatic program repair," *CoRR*, vol. abs/2103.00073, 2021. [Online]. Available: https://arxiv.org/abs/2103.00073

[46] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *CoRR*, vol. abs/1603.06129, 2016. [Online]. Available: http://arxiv.org/abs/1603.06129

[47] W. Crichton, "Human-centric program synthesis," *CoRR*, vol. abs/1909.12281, 2019. [Online]. Available: http://arxiv.org/abs/1909.12281

[48] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. R. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, and U. Finkler, "Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks," *CoRR*, vol. abs/2105.12655, 2021. [Online]. Available: https://arxiv.org/abs/2105.12655

[49] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436

[50] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *International Conference on Mining Software Repositories*, ser. MSR. ACM, 2018, pp. 476–486.

[51] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Lincoln, Nebraska, USA: IEEE Computer Society, November 2015, pp. 574–584. [Online]. Available: https://doi.org/10.1109/ASE.2015.36

[52] M. Zavershynskyi, A. Skidanov, and I. Polosukhin, "NAPS: natural program synthesis dataset," *CoRR*, vol. abs/1807.03168, 2018. [Online]. Available: http://arxiv.org/abs/1807.03168

[53] W. Ling, E. Grefenstette, K. M. Hermann, T. Kociský, A. W. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *CoRR*, vol. abs/1603.06744, 2016. [Online]. Available: http://arxiv.org/abs/1603.06744

[54] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[55] J. Pašek, J. Sido, M. Konopík, and O. Pražák, "Mqdd: Pre-training of multimodal question duplicity detection for software engineering domain," 2022. [Online]. Available: https://arxiv.org/abs/2203.14093

[56] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *CoRR*, vol. abs/2004.05150, 2020. [Online]. Available: https://arxiv.org/abs/2004.05150

[57] M. Rosenstein, Z. Marx, L. Kaelbling, and T. Dietterich, "To transfer or not to transfer," 01 2005.

[58] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[59] A. Romero, “Openai sold its soul for 1 billion,” Aug 2021. [Online]. Available: https://onezero.medium.com/openai-sold-its-soul-for-1-billion-cf35ff9e8cd4

[60] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” *CoRR*, vol. abs/2109.00859, 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[61] P. Yin and G. Neubig, “A syntactic neural model for general-purpose code generation,” *CoRR*, vol. abs/1704.01696, 2017. [Online]. Available: http://arxiv.org/abs/1704.01696

[62] ——, “TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation,” pp. 7–12, 2018. [Online]. Available: https://doi.org/10.18653/v1/d18-2002

[63] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, “A grammar-based structural CNN decoder for code generation,” in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.* AAAI Press, 2019, pp. 7055–7062. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33017055

[64] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, “Incorporating external knowledge through pre-training for natural language to code generation,” *CoRR*, vol. abs/2004.09015, 2020. [Online]. Available: https://arxiv.org/abs/2004.09015

[65] E. C. R. Shin, M. Allamanis, M. Brockschmidt, and A. Polozov, “Program synthesis and semantic parsing with learned code idioms,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10 824–10 834. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/cff34ad343b069ea6920464ad17d4bcf-Abstract.html

[66] X. Xu, C. Liu, and D. Song, “Sqlnet: Generating structured queries from natural language without reinforcement learning,” *CoRR*, vol. abs/1711.04436, 2017. [Online]. Available: http://arxiv.org/abs/1711.04436

[67] N. Beau and B. Crabb'e, "The impact of lexical and grammatical processing on generating code from natural language," *ArXiv*, vol. abs/2202.13972, 2022.

[68] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Retrieval augmented code generation and summarization," *CoRR*, vol. abs/2108.11601, 2021. [Online]. Available: https://arxiv.org/abs/2108.11601

[69] L. Perez, L. Ottens, and S. Viswanathan, "Automatic code generation using pre-trained language models," *CoRR*, vol. abs/2102.10535, 2021. [Online]. Available: https://arxiv.org/abs/2102.10535

A CodeFormer Denoising Output Examples

This section presents several examples of denoising intentionally corrupted source code using our CodeFormer model. The presented examples suggest that the model has acquired the necessary knowledge to produce a denoised source code of high quality. We even speculate that humans would not be able to produce better outputs than our neural network.

If we assess the mistakes observed in the examples, we can see that our model produces a redundant `class` or `def` keyword at the beginning of the predicted code snippets. Such a mistake can be corrected using a simple deterministic rule, and therefore, it does not represent any obstacle in utilizing our model. Furthermore, in example 3, we can see that the model confused functions `loss()` and `linear()`. However, there are no clues in the noisy input that would suggest favoring the `linear()` function. A more major mistake can be observed in example 4, where our model completely mismatched both the function name and its body. Nevertheless, if we look at the context acquired from the noisy input, we can see that it does not provide enough information to make a correct prediction.

A common feature of all these examples is that they are syntactically valid and formatted according to Python indentation rules. This is a promising result, which shows that our model could be easily used, for example, to correct errors in source code.

1. noisy

```
class [MASK] not :
[MASK]  def [MASK] [MASK]
  dt = as [MASK] dt ). if kwds ) self. onOffset ( = ) ) [
   ↪ MASK] dt - self. __class__ ( 1, normalize = self
  normalize, ** self. ( ) )
return dt
```

denoised

```
class class Date DateOffset(tzinfo):
  def rollback(self,dt):
    dt=as_timestamp(dt)
    if (not self.onOffset(dt)):
      dt=(dt-self.__class__(1,normalize=self.normalize,**
   ↪ self.kwds))
    return dt
```

label

```
class DateOffset(BaseOffset):
  def rollback(self,dt):
    dt=as_timestamp(dt)
    if (not self.onOffset(dt)):
      dt=(dt-self.__class__(1,normalize=self.normalize,**
   ↪ self.kwds))
    return dt
```

2. noisy

```
def peak_prominences [MASK] [MASK] [MASK]
   ↪ _arg_x_as_expected ( [MASK]rg_peak [MASK]s )
wlen = _arg_wlen_as_expected ( wlen )
return _peak_prominences ( x, peaks, wlen )
```

denoised

```
def peak_prominences(x,peaks,wlen=None):
  x=_arg_x_as_expected(x)
  peaks=_arg_peaks_as_expected(peaks)
  wlen=_arg_wlen_as_expected(wlen)
  return _peak_prominences(x,peaks,wlen)
```

label

```
def peak_prominences (x,peaks,wlen=None):
  x=_arg_x_as_expected(x)
  peaks=_arg_peaks_as_expected(peaks)
  wlen=_arg_wlen_as_expected(wlen)
  return _peak_prominences(x,peaks,wlen)
```

3.

noisy

```
) )
:class LossTest ( unittest. TestCase ) [MASK] [MASK], x [
    ↪ MASK] m, c [MASK]
return ( ( m * x c +
```

denoised

```
class class LossTest(unittest.TestCase):
  def loss(self,x,m,c):
    return ((m*x)+c)
```

label

```
class LossTest(unittest.TestCase):
  def linear(self,x,m,c):
    return ((m*x)+c)
```

4.

noisy

```
ModuleRegister. [MASK]Map ( module_map [MASK]Map [MASK] )
    ↪  :
```

denoised

```
def def def_init(module_map:ModuleMap):
  ModuleRegister.init()
```

label

```
def registerModuleMap(module_map):
  ModuleRegister.registerModuleMap(module_map)
```

5.

noisy

```
def ion ( ) :
  [MASK] ( True [MASK]_repl ding_displayhook ( )
```

denoised

```
def def ion():
  matplotlib.interactive(True)
  install_repl_displayhook()
```

label

```
def ion ():
  matplotlib.interactive(True)
  install_repl_displayhook()
```

6.

noisy

```
def [MASK] entry ) : [MASK] path. normcase ( entry ) [NL
    ↪ ]. ) WINDOWS and ( not os if path. [MASK] ] ) ) :
[MASK] = entry. replace ( [MASK] ( self [MASK] ( entry )
    ↪ [MASK[MASK]installPthEntries ( object ) :
```

denoised

```
class class UninstallPthEntries(object):
  def add(self,entry):
    entry=os.path.normcase(entry)
    if (WINDOWS and (not os.path.splitdrive(entry)[0])):
      entry=entry.replace('\\','/')
    self.entries.add(entry)
```

label

```
class UninstallPthEntries(object):
  def add(self,entry):
    entry=os.path.normcase(entry)
    if (WINDOWS and (not os.path.splitdrive(entry)[0])):
      entry=entry.replace('\\','/')
    self.entries.add(entry)
```

B Examples of Generated Code

This appendix lists several examples of source codes generated using our CodeFormer model with the CodeBERT used as an additional encoder. In the first part, we present source code generated based on our own prompts. Examples 1-6 are generated using a model trained on SOCGD, whereas examples 7 and 8 were obtained using a model trained on CodeSearchNet. The rest of the examples show source codes generated from NL prompts obtained from the test split of CodeSearchNet (examples 9-13) and SOCGD (examples 14-16).

1.

load JSON file and parse it into a dictionary

```
import json

with open('data.json') as f:
  data = json.load(f)

print(data)
```

2.

send an HTTP post request with a JSON body to a REST API

```
requests.post(url, data = json.dumps(data), headers=
    ↪ headers)
```

3.

execute an OS command from a Python script

```
import subprocess
subprocess.call(['ls', '-l'])
```

4.

find all files matching given GLOB in a given directory

```
glob.glob(os.path.join(root,'*.txt'))
```

5.

save dictionary into a file using pickle

```
with open ('myfile.txt', 'wb') as f:
  pickle.dump(dict1, f)
```

6.

use argparse to parse commandline arguments

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-a','--a',type=int,help='a number')
parser.add_argument('-b','--b',type=int,help='b number')
args = parser.parse_args()
```

7.

find substring matching a provided regex

```
def search(pattern, string, flags=0):
  return re.search(pattern, string, flags=flags )
```

8.

read CSV file into a list

```
def csv(self):
  with open(self.path,'r') as csvfile:
    reader = csv.reader(csvfile, delimiter=',', quotechar
    ↪ ='"')
    for row in reader:
      yield row
```

9.

prompt

Returns True if this node has no children, or if all of its children are ParseNode instances and are empty.

prediction

```
def has_children(self) :
  if not self.children:
    return False
  return all(isinstance(child, Node) for child in self.
    ↪ children)
```

label

```
def is_empty(self):
  return all(isinstance(c, ParseNode) and c.is_empty for
    ↪ c in self.children)
```

10.

prompt

Queries mysql and returns a cursor to the results.

prediction

```
def cursor(self):
  if self._cursor is None:
    self._cursor = self.conn.cursor()
  return self._cursor
```

label

```
def _query_mysql(self):
  mysql = MySqlHook(mysql_conn_id=self.mysql_conn_id)
  conn = mysql.get_conn()
  cursor = conn.cursor()
  cursor.execute(self.sql)
  return cursor
```

11.

prompt

Add a Set-Cookie header to response object. For a description about cookie attribute values. Arguments: name (str): Name of the cookie value (str): Value of the cookie attrs (dict): Dicitionary with cookie attribute keys and values.

prediction

```python
def add_cookie(self, name, value, domain=None, path='/',
    ↪ secure=False, httponly=False):
  cookie = SimpleCookie()
  cookie.name = name
  cookie.value = value
  cookie.domain = domain
  cookie.path = path
  cookie.secure = secure
  cookie.httponly = httponly
  cookie.path = path
  cookie.secure = secure
  cookie.httponly = httponly
  return cookie
```

label

```python
def set_cookie(self, name, value, attrs={}):
  cookie = http.cookies.SimpleCookie()
  cookie[name] = value
  for key, value in attrs.items():
    cookie[name][key] = value
  self.add_header('Set - Cookie ', cookie[name].
    ↪ OutputString())
```

12. prompt

Scans through a string for substrings matched some patterns. Args: text: A string to be scanned. patterns: a list of regex pattern. Returns: a list if matched. empty if not.

prediction

```
def match(text, patterns):
  if not patterns:
    return []
  for pattern in patterns:
    if re.search(pattern, text):
      return [pattern]
  return []
```

label

```
def matchall(text, patterns):
  ret = []
  for pattern in patterns:
    match = re.findall(pattern, text)
    ret += match
  return ret
```

13. prompt

Write the content into the given file. :param content: The dict to write. :type content: dict :param file: The file to write. :type file: str

prediction

```
def write_file(content, filename):
  with open(filename, 'w') as f:
    f.write(content)
```

label

```
def _write_content(cls, content, file):
  if not PyFunceble.CONFIGURATION[" no_files "]:
    if not isinstance(content, dict):
      content = {}
    Dict(content).to_json(file)
```

14. prompt

How to replace extracted HTML tags in a text with a blank? I am extracting text from an html page without all tags (using Python and BeautifulSoup). However, tags are not replaced with a blank. So, for example, for "blah blahDIVTAGblah" I get the following text "blah blahblah". How can I insert a blank between the second and third blah? I am using the following code. The code is from BeautifulSoup Grab Visible Webpage Text

prediction

```
import re

text = re.sub(r'<.*?>', '', text)
print(text)
```

label

```
for script in soup(["script", "style"]):
  script.replace_with(" ")
```

15. prompt

how to replace a NaN value in a column in data frame in pandas? i'm a newbie to pandas library and trying to do some analysis on the Titanic survival data using pandas. in the data set there are some cells in the Age column that doesn't have value so they appear in the data frame as 'NaN'. so i'm trying to replace these cells with the average of the male-ages for males and the average of the female ages for females. first i tried this well it works but doesn't change the values of the cells in the data frame when i print it later so i tried this and got this error message so anyone knows how to do it efficiently?

prediction

```
df['Age'] = df['Age'].fillna(df['Age'])
```

label

```
df['Age'] = df['Age'].replace(df[(df['Sex'] == 'male') &
    ↪ (df['Age'].isnull())]['Age'], df['Age'].mean)
```

16.

prompt

How to write a pandas dataframe to_json() to s3 in json formatI have an AWS lambda function which creates a data frame, I need to write this file to a S3 bucket.

prediction

```
import boto3

s3 = boto3.resource('s3')
bucket = s3.Bucket('mybucket')
key = bucket.Object('mykey')
key.put(Body=json.dumps(data))
s3.put_object(Bucket='mybucket', Key=key)
```

label

```
import boto3
import io

destination = " output_ " + str(datetime.datetime.now().
    ↪ strftime('%Y_%m_%d_%H_%M_%S'))+'.json'
json_buffer = io.StringIO()
df.to_json(json_buffer)
s3 = boto3.resource('s3')
my_bucket = s3.Bucket('my-bucket-name')
my_bucket.put_object(Key=destination, Body=json_buffer.
    ↪ getvalue())
```