School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Python Generation from Natural Language with Decoder-Only Transformers and Realistic Resources

Tomás Pimentel Zilhão Pinto e Borges

Submitted in accordance with the requirements for the degree of Meng, BSc, Computer Science

2021/2022

COMP3931 Individual Project

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Final report	*PDF file*	*Minerva (03/05/2022)*
Source code and relevant files	*GitLab Repository URL:* *https://gitlab.com/tomaspzpeborges/tzb_fypa*	*Supervisor and Assessor (03/05/2022)*

Type of Project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)

Summary

The problem of code generation from natural language consists of synthesizing code fragments given natural language descriptions of intent.

In recent times, the transformer architecture has been the de facto approach for various natural language problems, including code generation from natural language .

This project intends to finetune a pre-trained decoder-only transformer model to the downstream task of python generation. We start by presenting a thorough historical study of the problem and solutions, followed by a walkthrough of all the steps involved in model training. We simultaneously explore techniques to improve the model performance despite the limitations of available computational resources. Namely, ways of making training more effective such as gradient accumulation, dynamic padding, standardizing the input format; and decoding techniques such as (external) sampling with temperature tweaking.

On evaluation, our primary objective is to study the effectiveness of finetuning a pre-trained model, and a secondary is to reflect on the choice of used methods.

We successfully finetune a decoder-only transformer - CodeParrot-CoNaLa-16-Django-11-100 - into generating one-line python snippets from pseudocode-like descriptions, with no guarantees of completeness. We report a BLEU score of 41.26 on the Django test dataset and it generates higher quality snippets than its baseline and other models of comparable size and pretraining.

Empirically, we find that standardizing the input format and that (external) sampling combined with temperature tweaking are effective ways of increasing performance. We find that a code pretraining phase is of higher importance than a natural language one for the problem of Code Generation from Natural Language.

We loosely follow the CRISP-DM methodology for reproducibility.

Acknowledgements

I would like to thank my supervisor Zheng Wang and assessor Natasha Shakhlevich for their guidance throughout the project. As well as Luke Conibear and John Hodrien for their support on high performance computing.

I would also like to thank my family and close friends for the unconditional support.

Table of Contents

Table of Figures

Chapter 1
Introduction and Background Research

1.1 Introduction

The latest technological advances in Artificial Intelligence show that it is possible to generate code from simple commands in natural language. This represents a huge paradigm shift in the current way programs are developed.

In fact, higher levels of expressiveness can be achieved with natural language than with programming languages [35], and such should increase developers' capabilities and lower entry barriers.

For Code Generation from Natural Language (CGNL), previous research has shown multiple ways of tackling this problem, with its earliest roots pointing to classic natural language processing and compiler approaches, in which the semantics of a language are studied and programmed into an explicit structure. More recently, state of the art (SOTA) performance for language representations has been delivered by models based on neural networks – a large corpora of a language is collected and fed into the network. And it's only very recently that this SOTA approach is targeting the particular problem of CGNL.

Inspired by state of the art models OpenAI Codex [43] and DeepMind Alphacode [71], we will explore ways of applying pre-trained language models to the downstream task of generating python code. Ultimately, we should finetune a model and tackle some of the challenges that SOTA are able to solve nowadays, adapted to the available time frame and resources.

1.2 Literature Review

We start the literature review by presenting the problem of Code Generation from Natural Language (CGNL). Next, we study various approaches for solving it, which are mainly based on general NLP techniques; for each approach, we then present implementations that look into solving our particular problem. Lastly, we justify our choices for problem scope and system of choice, in line with state of the art (SOTA) technology.

Literature concerned about program synthesis from other inputs and subfields that emulate code generation (e.g. code retrieval) are not considered.

1.2.1 Code Generation from Natural Language

Code Generation from Natural Language is originated from the symbiosis between the fields of Code Generation and Natural Language Processing and is concerned about exploring various techniques and systems that generate program code from natural language descriptions [65] [19]. In addition, the field itself can be segregated into different categories.

In general, we can categorise based on program form [35], *i.e.* based on how we specify the program in natural language (input) and how we expect the generated program to be (output). Inputs may range in a spectrum of 'naturalness', in which its values may go from pseudocode-like text and line by line descriptions all the way to abstract descriptions of functionality. For example, 'sort a list by ascending order and remove repeated elements' would be more abstract (and harder to solve) than its step-by-step correspondent. On the other hand, outputs can be defined by program domain, ranging from specific ones (e.g. excel functions) to general purpose programming language.

Throughout time, it is possible to observe a gradual shift to approaches that handle higher levels of abstraction/naturalness and deal with general purpose source code; mainly statistical and machine learning models [23][35]. However, such came with the price of losing completeness of generated code – no guarantees of being interpretable/runnable - and that's still an open problem [35].

We then position our ambitions for this project in providing simple descriptions (slightly more complex than a line of pseudocode) as input to a decoder-only transformer. In addition, the system should interpret it as functionality possible of being generated in a one-line python snippet, with no guarantees of completeness (of being readily interpretable).

1.2.2 Rule-based Approaches

The field of Natural Language Processing (NLP) enters the status quo in 1957 after Noam Chomsky [48] introduces Phase Structure Grammars, so natural language sentences can be represented by a set of rules, which makes them logically understandable to computers.

Continuous developments in the field eventually lead to applying some of the same principles to the emerging field of Code Generation and it originated a set of techniques; at the time meant to allow software developers to write code at a higher level of abstraction (e.g. compilers).

Some well-known techniques include context-free grammars (used to represent the production rules of a language), abstract syntax trees (AST's, used to represent syntactic structures), parsers (transform text utterances into logical forms), part of speech tagging (POS, encode additional information), among others.

Thus, early approaches to CGNL look into matching language utterances with predefined rules/logical representations of a language, which are then easily transformed into another language's set of rules.

As an implementation example, Little and Miller [21] propose a system based on keyword extraction. So if a user is familiar with the vocabulary of an application domain, he can describe a command in natural language accurate enough to be translated into an application domain's command. This is successfully implemented to the restricted domain of generating Microsoft Word commands.

A more ambitious system called NaturalJava, combines 3 subsystems to create a java program from line-by-line natural descriptions. First, it looks into extracting information from the sentences, then it generates case frames representing a program's construction through AST's and finally it translates them into actual java code [35][16].

1.2.3 Statistical Approaches

The switch to statistical and neural systems is motivated by the possibility of learning a language's rules through the analysis of large corpora - Language Modelling [19] [66]. In addition, these approaches show promising results in capturing more complex rules than its predecessors and allow higher levels of abstraction (as mentioned above) [35].

So, we introduce Statistical Machine Translation (SMT) models which gather the probability distribution of parallel corpora, therefore deducing relationships between the two modalities of the corpus. There are many approaches to modelling the distribution, but a common approach is to use Bayesian theory. In addition, SMT's allow some flexibility in defining the fundamental unit of translation: word-based, syntax-based, hierarchical phrase-based or

phrase-based [69]. The latter leads us to introduce the concept of N-grams, which define the size of the contiguous sequence.

Probabilistic Context-Free Grammars (PCFG) may be considered a hybrid approach between rule-based methods and statistical ones, since we assign a probability weight to each production of a grammar, which can be adapted based on the corpus distribution.

As with most technological transitions, they are done gradually and some of the earlier implementations of new technology are often combined with other established technologies as hybrid solutions.

For example, anyCode is a system that takes as input a mixture of a natural language description and java code and extracts relevant information from it. Then, it proceeds to use a unigram and probabilistic context-free grammar model to produce the expected code snippet [62].

T2API is a graph-based SMT tool that is trained on a StackOverflow corpus of description-code pairs and should generate an API code snippet from a natural description. It first combines information from its training phase with the input description to derive the correct API elements. Next it proceeds to ensemble the desired API call [61].

Another example from Tao Lei et al. [59] introduces a Bayesian generative model capable of capturing English characteristics and creating a specification tree from them. This tree can then be fed into a C++ input parsers that will generate code.

1.2.4 Neural Approaches

Machine Learning (ML) techniques found its way to NLP motivated by the need to capture more complex representations of meaning than what purely statistically based approaches could.

1.2.4.1 Vocabulary

The descriptions that we provide as input to an ML model are not readily interpretable by it. So, we split (tokenize) the text into words or subwords - tokens - and encode each of them into a number (input-id), representing a unique token.

This set of numbers is what we define as a Vocabulary.

1.2.4.2 Continuous Word Embeddings

The earliest language modelling works for machine learning systems were purely based on word embeddings.

Continuous Word Embeddings encode each token as a distinct vector in a high dimensional space, in which subparts of the vector may represent a certain characteristic about the token [60]. Furthermore, relationships between tokens can be interpreted as the relative distance between vectors [60].

In 2013, a set of ML models called Word2Vec [64] was presented as capable of "reconstructing linguistic contexts of words" [67], by training it on a large corpus of text and produce a vector space as described above. The 2 main models were capable of predicting a word based on context, and of predicting a predefined number of surrounding words based on a current one [64].

Even though there's no relevant Word2Vec models that tackle our problem, word embedding plays a crucial role in representing meaning in machine learning and is used on the following approaches. On the other hand, continuous word embeddings have the disadvantage of being context-independent, so words can't be disambiguated [60].

1.2.4.3 Recurrent Neural Networks

The original Recurrent Neural Network (RNN) architecture is developed in 1986 but it has assumed many forms throughout times [68], and was adopted for language modelling in 2014 [60].

In principle, RNN's are deep neural networks where its connections allow it to exhibit a temporal dynamic behaviour, usually through a loop. Thus, each input token is received at a different time instance 't' and uses the activation value from 't-1' to calculate the activation at 't'. In addition, they can have additional stored states – gated memory [68][14][53].

This natural ability to remember, means that RNN's "can understand sequences by the fact that the current state is affected by its previous states" [53], and therefore are suitable to overcome the problems faced with word embeddings.

On the other hand, the fact that we need to access tokens sequentially (left to right) make RNN's slow to compute (O(n), n token length of sequence) and it means that we can't access forward context, which is problematic as we often disambiguate a word after we actually read it. For example, in programming languages we know that a word is a function name after we find the opening parenthesis. In addition, RNN's may suffer from vanishing gradient: long sequences may cause the end state to completely forget some of its initial ones [60].

This last problem can be partially solved by adding Long Short Term Memory (LSTM) as gated memory (gated RNNs). This cell allows information to be retained for longer sequences by allowing it to flow in the network without further processing [53].

For our particular problem, the Tellina system uses semantic parsing and RNNs trained on pairs made of one-line descriptions and shell commands. An input (description) undergoes semantic parsing and is transformed into tokens with relevant details captured. Then, those tokens are read into the RNN, which should output a shell command [35] [70].

Even if on the realm of Code Completion, Li et al. [34] analyse source code with an abstract syntax tree and make predictions based on RNNs with attention mechanism (see below).

1.2.4.4 Transformers and Self-Attention

In 2017, the Transformer architecture is introduced [3] and it has been the state of the art for language modelling. It is a deep learning architecture where each layer is mainly based on the self-attention mechanism.

Self-attention allows each token to look at all tokens of a sequence. This makes sequence processing non-sequential and context to have bidirectional access [60] [3].

Even though RNN's can be implemented with self-attention, it is proven in the transformer introductory paper [3] that "an architecture purely based on the attention mechanism and without any RNN's" [44] could get better results in language modelling tasks.

So compared with (gated) RNNs, Transformers solve gradient vanishing, unidirectional context and sequential access. In addition, they shorten path length between long term dependencies in the network and the amount of total computation per layer [3] .

On the downside, giving each token access to the whole sequence makes self-attention computationally expensive (O(n^2) per layer, n token length of sequence), but it can be parallelised [3]. In addition, Transformers must have a fixed number of input tokens (controlled by truncation and padding) [060] and they are especially deep structures.

As an example, GraphCodeGPT [32] is a decoder-only transformer based on GPT-2 [5] (12 layers, 1.5B parameters) pre-trained and finetuned on source code with additional data flow information as a graph. This additional structure improves performance when compared with a baseline without the graph.

The decoder-only transformer OpenAI Codex-S (96 layers, 175B parameters) [43] is pre-trained (unsupervised) on python code from GitHub and finetuned (supervised) on pairs made of a problem description and a standalone function. Alphacode from DeepMind (41B parameters) [71] is a full transformer that is equally pre-trained but undergoes a more ambitious finetuning phase with competitive problems – solution pairs. These 2 models

are the current SOTA in both performance and levels of abstraction (as described in Chapter 1.2.1).

1.2.4.5 Transformers and Self-Attention in detail

Starting from the point that it is now clear why the transformer is the preferred choice for language modelling, we proceed to analyse it in detail and why we will be using the decoder-only architecture.

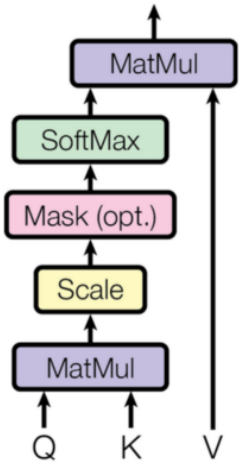

Figure 1: Scaled Dot Product Attention [3].

As we have just seen, self-attention works by allowing each token in a sequence to take into account all the others [60].

Let's think of a sequence of arbitrary length where we split it into vocabulary tokens and then convert each of them into a continuous word embedding as described above. Now the basic idea of attention is that this embedding originates 3 distinct vectors: Query, Key and Value (described in Fig. 1 by 'Q', 'V', 'K') [3].

The goal of self-attention is to update each token embedding based on all the others. To do so for a particular token, we determine its compatibility with another by taking the dot product between its query vector and the other's key vector (the higher the value the more compatible they are) [7]. And if we do this for all key vectors, we obtain a new vector of compatibility scores, each representing the usefulness of some token to the queried token [60].

Then, we scale these scores (avoids vanishing gradient for high values of the softmax) [7] and normalize them with softmax (increases the model's confidence in which tokens to attend) [46].

Finally, we multiply each score by its value (the score belongs to a certain key, and we find its value as in a dictionary) and add them together. A non-contextual embedding was transformed into a contextual one [60].

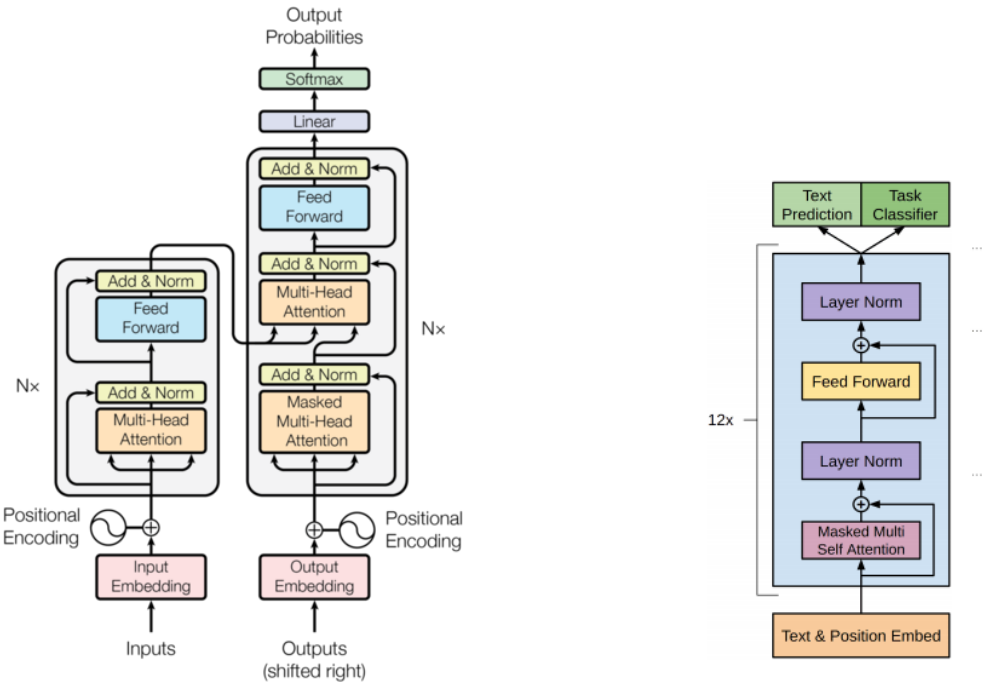

Figure 2: (right) Decoder-Only implementation used in OpenAI GPT and GPT-2 models [4] [5].

Figure 3: (left) Transformer Encoder-Decoder Architecture [3].

Transformers can be broken down into two main parts: an encoder layer, responsible for mapping all input sequences into an abstract continuous representation [46] (similar to the space introduced in word embeddings above); and a decoder layer, responsible for generating text sequences. In addition, a transformer can have an arbitrary number of encoder and/or decoder layers, where "each layer has the opportunity to learn different attention representations, therefore improving the predictive power of the transformer" [46].

Each layer can be subsequently broken down into a number of components which we will analyse in order to understand how transformers work (following Fig.3).

First on the Encoder layer, 'Input Embeddings' encode input text into token embeddings as described previously. Furthermore, they are added together with a "Positional Encoding" of

the same dimensionality in order to be able to express relative positions between tokens in a sequence [3].

Next, ‘Multi-head Attention’ is simply an implementation detail that parallelises a plurality of self-attention mechanisms together. Similar to adding more encoder or decoder layers, including multiple attention heads may allow the model to focus on distinct linguistic aspects [60] [3].

The following ‘Add & Norm” components are used to “allow gradients to flow through the networks directly” [46] and to stabilize training, respectively. The ‘Feed Forward’ neural network simply converts the output of the multi-head attention back into a single attention output.

Moving to the Decoder layer, ‘Output Embeddings” are subject to an analogue process to ‘Input Embeddings’, except that outputs are usually a target text sequence.

Its first multi-head attention uses a mask to predict the next token solely based on the context of the previous tokens (the ones to its left) [24][2]. This respects the temporal dependency of the output sentence and gives the decoder its auto regressive nature.

The next multi-head attention differs from all the others by implementing cross attention, i.e. it combines the knowledge representation built from the encoder with the decoder’s (masked) input [46].

Finally, ‘Linear’ is a linear classifier with one class per token in our vocabulary. ‘Softmax’ outputs the class with highest value, which corresponds to the next predicted token based on inputs and previously generated outputs.

To conclude, we will be using a decoder-only implementation (as described in Fig. 2) for the development of our project,. From a theoretical point of view, this simpler architecture has proven to be driving SOTA performance [43]. And from the examples given during this Chapter, transformers can (in theory) deal with higher levels of ‘naturalness’ in the input description and unrestricted domains as output.

Chapter 2
Methods

In this chapter, we discuss the design and implementation of the proposed solution.

We present some intermediate results and theory throughout the chapter where we believe they avoid breaking the reading flow. We leave to the next chapter the approach and results of the formal evaluation.

2.1 Project Management and Version Control System

The project plan is based on an iterative approach and on agile methodology.

During the first semester, our work is heavily based on acquiring a strong basis of knowledge both in general machine learning as well as in the CGNL.

On a second and final phase, we plan 5 sprints back to back, which will allow us to cover all aspects of the project multiple times, while gradually shifting our focus from analysis to development. At the end of each sprint, we reflect on the work done and re-assess the work that needs to be done. At the end of the 5 sprints, we leave 2 weeks for unexpected events and report writing.

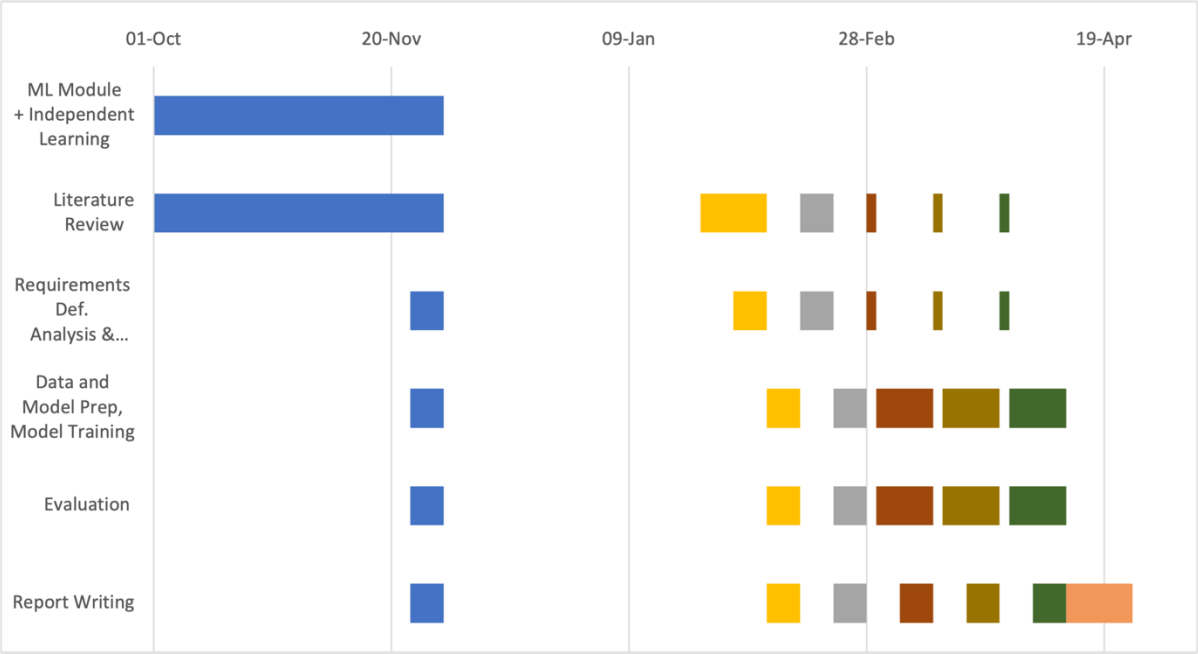

Figure 4: Gantt chart for the development timeline. Each colour represents a different sprint (except blue).

Development, planning and this report (loosely) follow CRISP-DM (Fig. 5), a cross-industry methodology that allows data mining projects to be reproducible and replicable [55].

It is not uncommon to adapt CRISP-DM to a project's needs [22][20], so we equate the first 3 steps on the Gantt chart (Fig. 4) to 'Business Understanding' and 'Data Understanding' . 'Data Preparation' and 'Modelling' are covered by "Data and Model Preparation, Model Training', 'Evaluation' remains unchanged and we omit 'Deployment' since it doesn't have significant stake on this project.

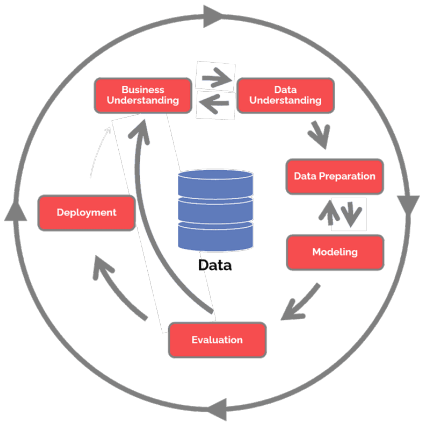

Figure 5: CRISP-DM [15].

During the full length of the project, we approach version control from a non-traditional standpoint and maintain copies of selected source code locally, on the cloud and on the ARC4 server (Chapter 2.2). This is due to the highly experimental nature of the project in which we play with model implementations and datasets of considerable size. At the end of development, we have over 72 000 files (almost 700GB) on ARC4 server, most of them generated as output from model training. File redundancy/safety is still made possible with cloud syncing.

At the end, we follow standard good practices, and create a Git repository with the final submission, and code correctly structured and commented.

2.2 Hardware Resources Limitations and Development Setup

For development, we have access to the University of Leeds' High Performance Computing (HPC) facility ARC4, where we mainly use an NVIDIA V100 graphics card with 32GB of RAM. ARC4 is "organised through a batch job scheduling system" [54] and the maximum duration for a job is 48 hours. In addition, we limit finetuning time on a smaller model (110M

parameters) to 3 hours for each dataset, and on a bigger one (1.5B parameters) to 15 hours for each dataset (Chapter 2.4). We limit formal evaluation to 48 hours (Chapter 3).

Initially, the plan included using as much resources as possible and even using distributed computing with multiple graphics cards. However, the unavailability of such capabilities originated a set of secondary objectives.

Nowadays, we recognize that there is an increased discrepancy between what can be achieved by resourceful companies and what can be achieved by the average individual or small business. This trend has the consequence of monopolizing the usage and development of large language models [1]. Hence, we deliberately choose to limit our resources to the ones above mentioned, in order to explore the limitations of what can be achieved with what we call 'Realistic Resources' – a possible scenario for these smaller entities that want to implement CGNL.

For the development setup, we create an anaconda environment with all the needed packages; we mostly use Pytorch and Huggingface frameworks, and a complete list of dependencies can be found in 'setup_dev.sh' in the submitted git repository.

2.3 Design

In this section we present the Model pipeline (Fig.6), by which we mean the ordered collection of operations that we apply to the initial model so that it achieves our goals.

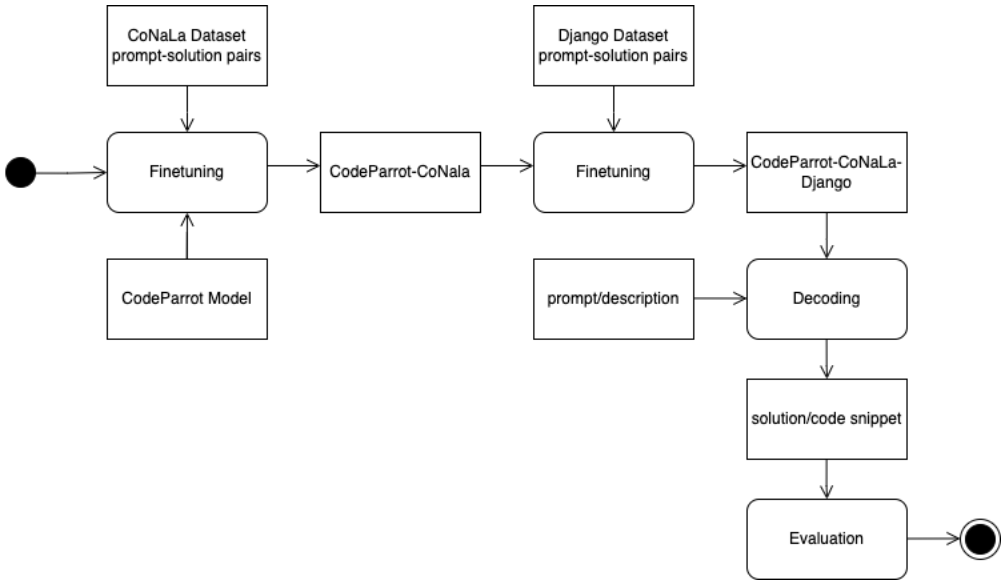

Figure 6: UML Activity Diagram representing the Model Pipeline Design

2.3.1 Finetuning

There exists multiple ways of training a model to perform a certain task, and that generally depends on the type of data it is being trained on. We can split training into 2 categories: pre-training and finetuning.

The transformer was originally thought of as a model capable of learning general purpose contextual representations of a certain domain or language [73] [4]. So, the way it acquires this type of knowledge is through pre-training. The main advantage of this process is that it uses unsupervised data as the input to the model, which is much more readily available than its counterpart. It works by maximising the probability of observing the (actual) token at a position, based on a predefined number of previous tokens (context). In addition, this process has proven to be quite effective and is considered a good starting point for solving more specific tasks [33] [4].

On the other hand, finetuning (or transfer learning) is the process of further training a pre-trained model so that it is capable of solving a downstream task (under the language model of the first pre-training) especially well, which has been proven to be effective [43]. To do so, we try to maximize the observation of a label, given context, and can do such by feeding a prompt to the model and trying to maximize the (token-level) average negative log loss likelihood via loss backpropagation (Chapter 3.1.1).

In addition, we can distinguish between 4 types of finetuning and they can be thought of as positions in a spectrum of “how much task-specific data they tend to rely on” [63]. These are Zero-shot, One-shot, Few-shot and Supervised Finetuning. We’ll only be concerned about supervised finetuning, since even though it requires more data to execute, it also increases the chances of good performance and therefore of achieving our objectives. From now on, we’ll be mentioning supervised finetuning as simply finetuning.

So, now we are in a position to explain which type of training is going to be executed in this project.

There are several factors that point to a preferred focus on finetuning over pretraining. First, the expected computational cost of generative pre-training is too high for the resources available (Chapter 2.2). For example, the CodeParrot models (1.5B and 110M parameters) that we end up choosing for finetuning (Chapter 2.4) were pre-trained for 1 week and 1 day (respectively) on 16 x A100 GPU machines with 40GB of RAM each [38], which is roughly 32 times more computational power than we have available. In addition, the SOTA model OpenAI Codex has 2 training phases (1 additional round of code unsupervised code finetuning on top of the natural language pre-training round executed for OpenAI GPT-3); where it is calculated that each of them would’ve taken 34 days on 1024 A100 GPUs [43] [17].

On the other hand, the environmental resources used for pre-training a transformer can be compared to the CO2 emissions of a car with fuel usage during its lifetime, especially if talking about the ever-increasing in size SOTA models [37] (Appendix A).

This should be enough to equate that pre-training is infeasible. However, we also recognize this phase as essential, since without it, we wouldn't be able to expose the model to enough data for effective language modelling [37].

So, we choose to finetune a pre-trained decoder-only transformer with 2 distinct phases (Fig. 6).

This last choice is justified with the approach explained by *Mark Chen et al* (2021) [43], in which they look for 2 finetuning phases of different complexities and that should complement each other. To match our objectives, we then finetune our model on prompt-solution pairs [35], where the former is a description and the latter is a corresponding line of phyton code, of arbitrary complexity. On the second round, we finetune on similar pairs, but the prompt is a pseudocode-like description of the next line of code.

2.3.2 Decoding Techniques

In the context of machine learning, the term decoding (or generating) is usually used to describe the process of translating model outputs back into formats that are human readable, in our case text. Furthermore, this process includes a set of techniques that do not modify the model's state, but allows us to tweak how it generates predictions and improve performance.

One common but effective [43] [71] method is temperature. For low values (close to zero) it makes a model increasingly confident in its top choices whilst temperatures close or greater than 1 decrease confidence [43]. In general, the higher the number of samples (see below) the higher the temperature should be [71].

Greedy search and beam search overlap in function. While greedy search simply chooses the highest probability token at every step, beam search allows us to keep a number of tokens at each step and only decide on the best sequence by calculating the highest overall probability at the end [27].

Finally, we can think about sampling in 2 ways. First, (internal) top-k sampling and top-p sampling limit the token choices at each prediction step to a defined number of tokens or to a number of tokens that satisfy a defined cumulative probability, respectively [27].

Second, (external) sampling can be concerned about decoding multiple sequences from the model (each sequence decoded with methods above mentioned) and that can increase the

probabilities of success on a certain task by finding patterns on (external) sampling itself or simply by increasing the chances.

We will be using the majority of the above mentioned methods to maximize performance during evaluation (Chapter 3).

2.3.3 Choice of Programming Language - Python

Lastly, we choose Python generation to the detriment of other programming languages, as the relatively simple syntax will yield a simpler language modelling for the model to learn. In addition, a number of datasets that match the requirements for the finetuning phases are found in python.

2.4 Dataset Selection – CoNaLa and Django

The main reason behind opting for choosing a dataset instead of curating one ourselves is the extreme expense of labelling data. With this in mind we set the task of finding 2 datasets that match the requirements that we expose in Chapter 2.3.

For the first dataset containing pairs made of a prompt and 1 line of python (of arbitrary complexity), we propose using the CoNaLa dataset, developed by Carnegie Mellon University NeuLab and STRUDEL Lab. This dataset is “designed to test systems for generating program snippets from natural language” [11] (which matches our requirements perfectly) and is made of 2,379 training and 500 test examples that are manually curated, and 598,237 examples of automatically mined pairs with an additional field for probability of correctness. This last set of examples is obtained through a promising classifier-based method that should move the field forward as “quality parallel NL-code datasets are currently a significant bottleneck in the development of new data-driven software engineering tool” [51]. Despite this, a quick analysis leads us to dismiss most of the examples with lower probabilities, as they clearly don’t seem to have good enough quality to be used for supervised training.

Fortunately, we find a curated set from Ahmed S. Soliman (2022) [6] which combines the manually curated examples with high quality mined ones. So, we decide to use this particular version of CoNaLa, totaling in 11,125 training, 1,237 validation and 500 testing samples.

For the second dataset composed of pairs prompt (pseudocode-like description)-solution (1 line of python), we propose using the Django dataset. Originally developed for automatic generation of pseudocode from source code, it contains datapoints consisting of “a line of

python together with a manually created natural language description" [72]. We find 16 000 training, 1000 validation and 1805 testing samples [72].

So even if its initial purpose is different from our goal, the dataset holds the right information and can be pre-processed to the correct format (check Chapter 2.6).

2.5 Model Selection – CodeParrot 1.5B and 110M

We've already reached the conclusion that we are selecting a pre-trained decoder-only transformer for finetuning and we found a number of publicly available candidate models for the task.

Model	Pretraining Dataset	Size (in parameters)
CodeParrot-small	Curated dataset from GitHub (python)	110M
CodeGPT	CodeSearchNet (python)	110M
CodeGPT-adapted	CodesearchNet (python) + GPT-2's natural language pretraining	110M
GPT-Neo	The Pile (various programming languages and natural language)	125M
GPT-2	Curated dataset from Web (natural language)	110M

Figure 7: Model selection for the smaller size

Out of these candidates (Fig. 7), we'll progressively reach the conclusion that the models CodeParrot with 1.5B and 110M parameters are the best choice.

First, CodeGPT (110M parameters) [58] is pre-trained on CodeSearchNet which (only) has 3.2GB of corpora in multiple programming languages [12].

CodeGPT-adapted (110M parameters) has the advantage of executing the same training process as CodeGPT above, but as unsupervised finetuning on top of a pre-trained GPT2 model (of the same size) [058]. It is argued that this makes the model have the same vocabulary and natural language ability as GPT-2 [58]. However, results from OpenAI Codex

training show that (surprisingly) there are no increases in performance between this 2 scenarios and that a natural language vocabulary (obtained during pretraining) needs some adaptation to represent code correctly [43]. The only advantage is that training converges more quickly [43], but that doesn't outweigh the small size of CodeSearchNet in pretraining context for us.

GPT-2-medium (345M parameters) and GPT-2 (1.5B) [5] would be equally good candidates if doing a round of unsupervised code finetuning was feasible (Chapter 2.3).

GPT-Neo models [18] (1.3B parameters and 125M parameters) are pretrained on The Pile [42] dataset (825 GB, but unclear if used in its totality), which is composed of a mixture of source code and natural language. However, it's pretraining process is not as clear as with CodeParrot (see below) and is based on the GPT-3's architecture, an improved version of GPT-2, but with less support currently available.

Finally, both CodeParrot models share GPT2's architecture and are pre-trained on 50GB of code (which also yields its vocabulary) [38]. The pre-training dataset is optimized to best train the model, by removing duplicate samples (which may cause leakage into the validation set) and applying the "same cleaning heuristics found in the Codex paper" [41][38].

Extensive work proves that bigger models perform better in general [63] so we primarily choose CodeParrot with 1.5B parameters. CodeParrot 110M will prove to be useful during training (Chapter 2.7) [39] [40]. In addition, the other models considered in this section are used for comparison during formal evaluation.

2.6 Dataset Pre-Processing

As we've seen in Chapter 1, transformers can only accept tensors as inputs so in this section our concerns are threefold: altering our datasets so that they: match our model's input format, are suitable for supervised learning and are resource-efficient.

2.6.1 General Pre-Processing

There's a number of techniques that are common for pre-processing both datasets.

The first tool that we use is a tokenizer, and it splits text (our pairs docstring-code) into tokens which are then encoded into input-ids that are part of the vocabulary (in theory looking for the most meaningful representation).

The tokenizer needs to be the same that was used for pretraining, since it was responsible for creating the base vocabulary of the model. So, we use Byte-Pair Encoding (BPE) which

is a subword tokenizer that works based on the rule that “frequently used words should not be split into smaller subwords, but rare words should be decomposed into meaningful subwords” [25].

In addition, the tokenizer creates attention masks which will be useful for padding (see below), and we extend the tokenizer’s functionality by creating ‘labels’, which will be used for masking out loss on prompts (Chapter 2.7.4).

For safety, we also truncate each example on prompt side in order to comply with the model’s sample size of 512 tokens, even though probably all of our samples should be of a smaller size.

Next, we utilize a Data Collator to implement dynamic padding (a tensor needs to have a uniform size). Dynamic padding has the advantage of only doing padding at batch creation time, which will avoid that padding is set to the longest sequence in the dataset, but simply to the longest sequence in a batch, therefore saving memory [26].

2.6.2 CoNaLa Pre-Processing

The CoNaLa dataset already holds prompt-solution pairs by default, so we could simply pass them to the tokenizer above mentioned.

However, we standardize the prompt format by wrapping it with triple quotation marks to represent python comments. This can help integrating the 2 distinct finetuning phases and aligns better with the evaluation scenario (check Chapter 3). Furthermore, we find that this small change improves training (Fig. 8).

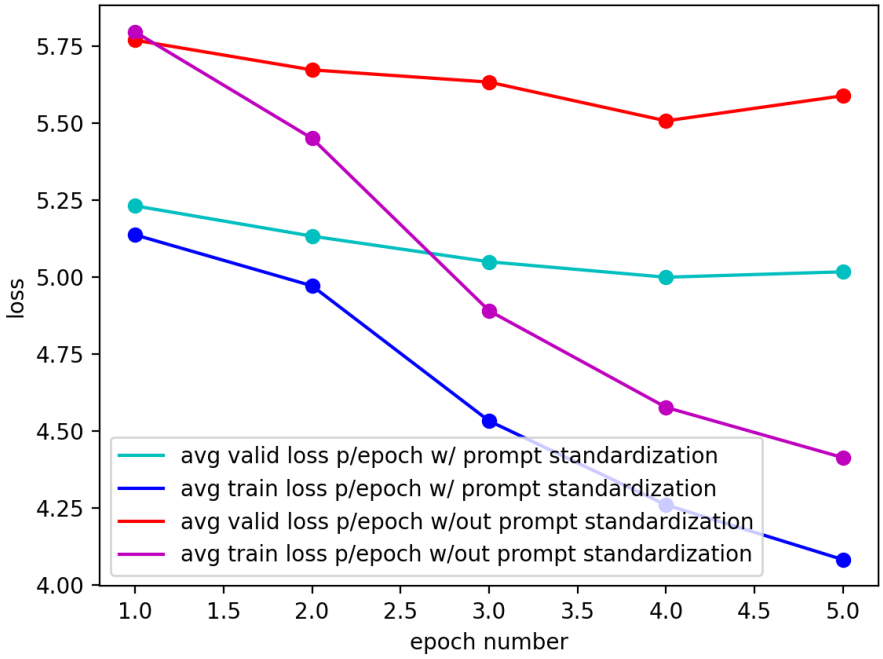

Figure 8: Preliminary Finetuning CodeParrot 1.5B on CoNaLa train dataset

2.6.3 Django Pre-Processing

The Django dataset requires a bit more work, as even though it holds the information we need, the prompts and the solutions exist in different files and can't be readily turned into a dictionary object. Thus, we create a json lines file which compiles line by line both of the above mentioned files into prompt-solution pairs.

Then, we standardize the prompt format in the same way we did for the CoNaLa dataset.

```
'''params is an empty list,'''
params = [ ]
```

Figure 9: Final format for a training sample; in blue the intent, in white the code snippet.

2.7 Model Training

Let's start by discussing the common ground between training both of our models.

2.7.1 Optimizer

Optimizers are algorithms responsible for minimizing the error function/loss that we define for finetuning (Chapter 2.3.1 and Chapter 3.1.1); they do so by changing weights and learning rates, and therefore updating the model [45]. We choose AdamW (learning rate=1e-3, betas = [0.9,0.95], weight decay= 0.1,eps = 1e-8), since it is a general consensus that it is among the best options [45] [57] and the one used with relevant literature for the project [43] [71].

2.7.2 Learning Rate Scheduler

Learning rate schedulers update the learning rate between training steps according to a given function. In general, it is useful to reduce the learning rate as training progresses, in order to travel smaller distances away from the minimum. So, we use Cossine Annealing, which implements the above mentioned with the add-on of briefly oscillating back to higher values in order to avoid being stuck in a local minimum [31][8].

2.7.3 Gradient Accumulation

So after we pre-process the training data as described in the last chapter and set up both the scheduler and optimizer, we're in a position to feed our training data into the model. The bigger the batch size we manage to feed, the smaller the number of training steps and therefore training will be performed quicker. However, we quickly encounter "Pytorch Out of memory errors" due to resource limitations, and so we settle for a batch size of 1.

To overcome such limitation, we proceed to implement Gradient Accumulation. This technique emulates bigger batch sizes by accumulating gradients and only updating network weights once every predefined number of training steps. So, we emulate a batch size of 8 by updating every 8 steps [47].

2.7.4 The Training Loop

So, the training loop: we feed each batch into the model, compute the loss, calculate the gradients, and update the model by stepping on the optimizer and learning rate scheduler.

To comply with our finetuning training objective (Chapter 2.3.1 and 3.1.1), we input our pairs intent-code snippet to the model and mask out loss (relative to labels) for all the tokens belonging to the prompt. We do so, by using the 'labels' field created during pre-processing, which replaces all the prompt tokens by the default value '-100', which is ignored by Pytorch.

Hence, each token after the prompt is predicted based on the prompt and the already decoded tokens (auto regressive nature of the decoder).

For each model and each dataset training combination, we save checkpoints after epochs where validation loss increases (while training loss keeps on decreases), since it is proven to be a good indicator to prevent against overfitting [56].

2.7.5 Training CodeParrot 1.5B

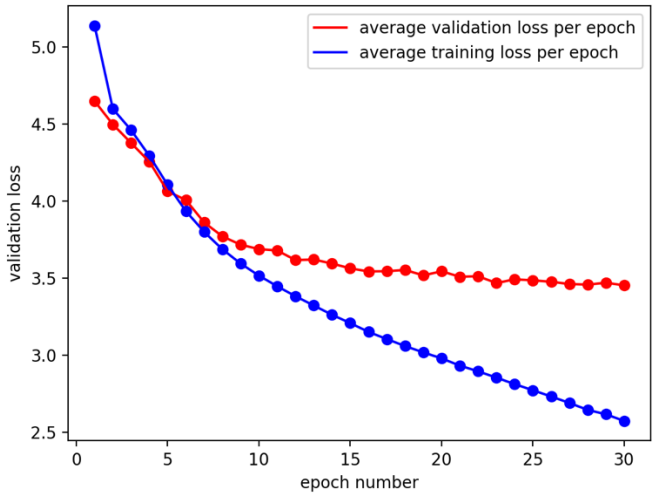

Figure 10: Finetuning CodeParrot 1.5B on CoNaLa dataset for 30 epochs

We train CodeParrot 1.5B on the CoNaLa dataset for 30 epochs, but we consider that validation loss starts increasing / stops making significant gains around the 16th epoch (Fig. 10). So, we choose the model checkpoint from that same epoch for further training on the Django dataset.

For the second training phase, we encounter “Pytorch: Out of memory” errors, which after debugging, point to an incapacity to fit in 32 GB of RAM the longer sequences from Django alongside the model and, surprisingly, the optimizer’s state.

We find that the AdamW optimizer is a stateful optimizer which holds large amounts of information about training at all times in order to make better decisions on reducing the loss.

As a possible alternative, we introduce the Adafactor optimizer, created to tackle some of the Adam’s memory issues and use it for the second finetuning phase. We initially train for 30 epochs. However, we find that the validation loss per epoch increases steadily after epoch 2, so the model is losing generalization capabilities. (Fig. 11)

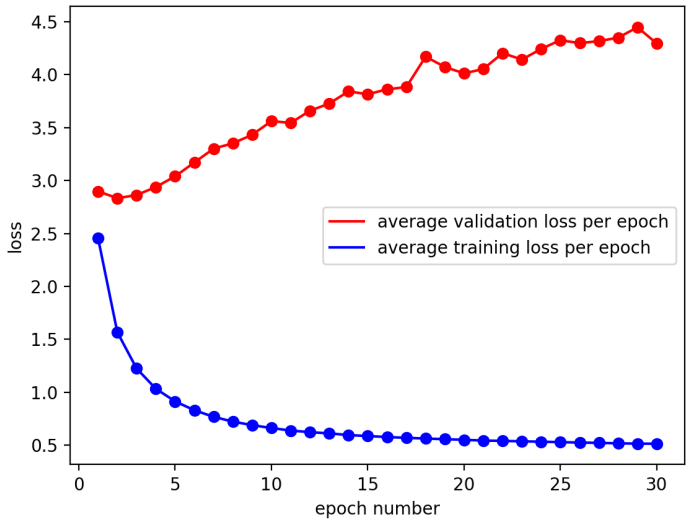

Figure 11: Finetuning CodeParrot-CoNaLa-1.5B on Django dataset for 30 epochs

We find evidence pointing Adafactor's inefficiencies for big models, specifically beyond 1B parameters [10], and attribute the results to such choice.

After this, we turn ourselves to the smaller CodeParrot 110M.

2.7.6 Training CodeParrot 110M

We train CodeParrot 110M on the CoNaLa dataset for 30 epochs again and we consider that validation loss starts increasing / stops making significant gains during the 16th epoch (Fig.12).

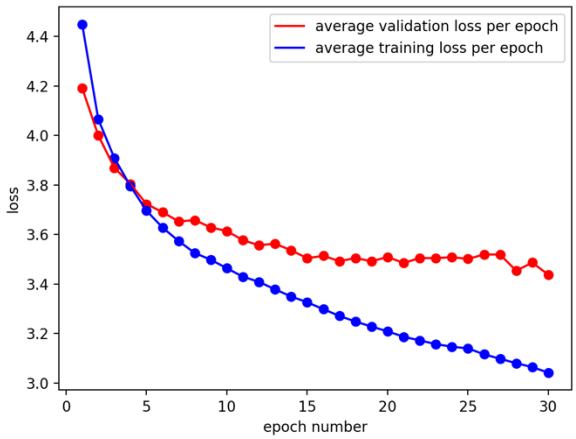

Figure 12: Finetuning CodeParrot 110M on CoNaLa dataset for 30 epochs

For the second training phase, we go back to using AdamW optimizer with the hyperparameters initially specified. We then duplicate training by finetuning on Django one model from the 16th epoch and another one from the 30th epoch.

As we can see from Figures 13 and 14 training looks less effective than in the first round and we don't observe significant differences between finetuning from the 16th and the 30th epochs. We consider that validation loss starts increasing / stops making significant gains after the 11th epoch.

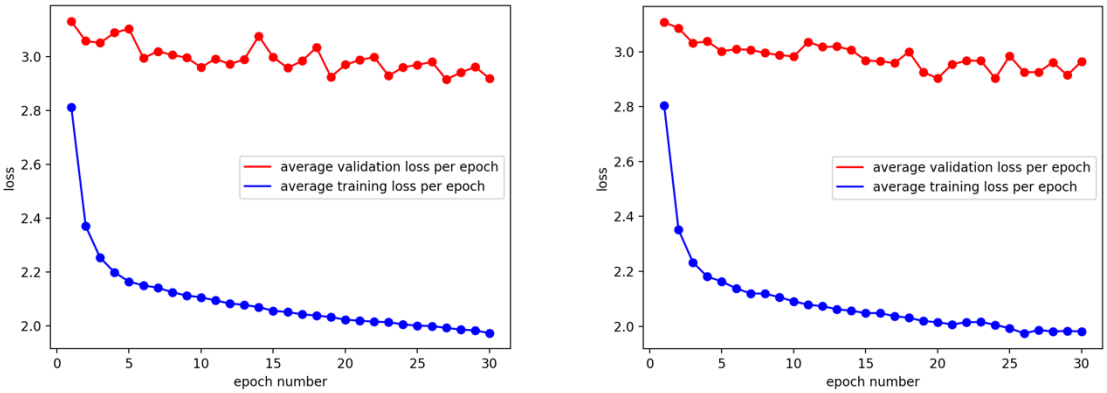

Figure 13: (on the right) Finetuning CodeParrot-CoNaLa-epoch-30-110M for 30 epochs

Figure 14: (on the left) Finetuning CodeParrot-CoNaLa-epoch-16-110M for 30 epochs.

Thus, we successfully reach to the end of training and will be using CodeParrot-CoNaLa-epoch-16-Django-epoch-11-110M as the main target for evaluation.

Chapter 3
Results and Discussion

In this final chapter, we present a quantitative and systematic approach to evaluating the CodeParrot-Conala-16-Django-11-110. We then present results and discuss them so that we gain valuable insights about our model and compare its performance with baseline and other similar models. Lastly, we present a conclusion and suggest future work.

3.1 Evaluation Framework

Recently, SOTA models in CGNL have moved evaluation from text distribution to functional correctness (e.g. pass@k and unit tests) [43] [71]. However, such requires the generation of a standalone functions which is not aligned with our training objective (to generate 1 line of python) and requires completeness (python ready to be interpreted). Thus, we opt for a set of more traditional metrics, which should still indicate how close a code snippet is to completeness relative to a solution and how helpful it can be.

We calculate the Average Negative Log Loss, BLEU, CRHF++ and Accuracy by taking each highest value out of n (external) samples for each example in the test dataset, and then taking the average between them.

We choose the models presented in Chapter 2.5 for eval as they have similar sizes to our model and are trained in different splits of natural language-code, which can be useful for comparison. SOTA models Codex [43] and Alphacode [71] are currently close-source and reported evaluations are mainly concerned with functional correctness, so won't be included.

3.1.1 Average Negative Log Likelihood and Perplexity

During the course of this report, we have mentioned minimizing negative log loss likelihood (NLL) as the chosen guideline for the finetuning process. So in the context of training, minimizing NLL represents maximizing the probability of estimating a given label correctly.

$$\mathrm{PPL}(X) = \exp\left\{-\frac{1}{t}\sum_{i}^{t}\log p_\theta(x_i|x_{<i})\right\} \qquad L_2(C) = \sum_{x,y} log\ P(y|x_1,\ldots,x_n)$$

Figure 15: (right) Log Loss Likelihood, label y, n previous tokens for context [52].

Figure 16: (left) Perplexity as exponentiation of the Average Negative Log Likelihood [28].

In practice, we calculate the average NLL for a generated code snippet by taking the negative average of the log probabilities calculated for each predicted token. In addition, if we want to calculate the average negative log loss likelihood relative to a set of generated code snippets (or a dataset), we take the average of the average NNL of each code snippet.

Perplexity is a concept closely related with the average NLL, and can be calculated as the exponentiation of the latter (Fig. 15 and 16).

In evaluation, Perplexity can be used to measure “how well a selection of text matches the distribution of text that the input model was trained on” [13]. So, we calculate the average NLL for each model on Django test dataset and use it as a proxy to talk about Perplexity. The lower the perplexity the better.

3.1.2 BLEU / sacreBLEU

BLEU is a well-known metric to assess the quality of a machine translation relative to a reference/human one [9]. It works by computing precision – the fraction of tokens and n-grams from the candidate that appear on the solution [9]– but it penalizes if a certain token frequency goes beyond its frequency on the solution. Even though there’s no direct relation between high scores and functional correctness [43], BLEU claims a “high correlation with human judgements of quality” [36], which leads us to deduce that a high score will indicate a code snippet that will be deemed helpful by a developer.

We will be using SacreBLEU [29], a BLUE implementation which provides flexibility for input format and don’t calculate scores code snippets that are smaller than 4 tokens, as the score is automatically zero.

3.1.3 CHRF++

CHRF++ is a machine translation metric that uses F-score statistic (harmonic mean of precision and recall) for character and word n-grams [30].

We use CHRF++ (with bi-grams) for evaluation and look to mitigate BLEU’s limitations for small code snippets.

3.1.4 Accuracy

Standard Accuracy is thought of as the proportion of correct predictions among the cases processed. We calculate it at generated snippet level.

3.2 Results and Discussion

An early manual analysis (example outputs on Appendix C) points to some possible limitations of the model. Firstly, it seems that the model is often capable of predicting snippets (close to) structural correctness for the Django test dataset. However, it doesn't show good capabilities in learning variable names from the prompt, usually replacing them with an apparently random words.

Second, it looks like the model does not deviate greatly from the expected solution in the cases when it comes close to the correct generation. We argue that this can be considered a model limitation in the generalization capacity, but can also be due to the existence of a reduced number of possible solutions in simple code snippets.

Thirdly, the model is incapable of predicting the correct length for decoding, and needs to be "hard limited" by the length of the reference solution.

Model	Average Cross Entropy Loss (CoNaLa test dataset)	Average Cross Entropy Loss (Django test dataset)
CodeParrot-CoNaLa-epoch-16-Django-epoch-11-110M	8.2710	2.8646
CodeParrot-110M (baseline)	3.8364	4.91825
CodeGPT-adapted-110M	11.2283	12.2277
CodeGPT-110M	12.51778	12.1402
GPT-Neo-125M	9.8780	10.7058

Figure 17: Average Negative Log-Likelihood for each model (greedy decoding)

Moving to formal analysis, Perplexity should be considered with caution when comparing it between distinct models, as preprocessing, vocabulary and context length greatly affect its value and are different between models. Nevertheless, our model and its baseline can be directly compared and in Fig. 17 we observe a strong improvement on Django test dataset.

On the other hand, we observe a performance decrease on CoNaLa test dataset and we hypothesize that our developed model lost language modelling capacities relative to the CoNaLa dataset during its second finetuning phase or that it didn't learn it enough well during its first phase (underfitting). To investigate further, we calculate NLL for another

checkpoint from our finetuning process with less epochs done on Django training dataset, and then another checkpoint only finetuned on CoNaLa training dataset.

Model	Average Cross Entropy Loss (CoNaLa test dataset)
CodeParrot-CoNaLa-epoch-16-110M	3.6641
CodeParrot-CoNaLa-epoch-16-Django-epoch-6-110M	8.0658

Figure 18: Average Negative Log-Likelihood for each model (greedy decoding)

The results (Fig. 18) confirm worsening of performance during the second finetuning phase but also not a significant increase in performance during the first finetuning phase. In Chapter 2.7, we observe a plateau on validation loss, so we reach a scenario where our model underfits the distribution of the CoNaLa dataset, but further training doesn't improve its performance. In line with Chip Huyen (2019) [13], this indicates that with "the given optimization algorithm, the model does not have enough capacity to fully leverage the data scale". In addition, for CodeParrot-CoNaLa-16-Django-11-100, we calculate BLEU (19.3834), CHRF++ (26.4728) and Accuracy (20.99%) on the CoNaLa test dataset and the results represent a significant deficit when compared with the results on Django test dataset (see below).

This leads us to conclude that our model is not effective in generating snippets with the complexity of the CoNaLa dataset (higher than pseudocode-like descriptions).

Next, we look into evaluating the other 3 metrics (BLEU, CHRF++., Accuracy) on Django test dataset while studying how to best use the chosen decoding techniques (Chapter 2.3.2).

Even though results from SOTA Alphacode [71] indicate that top-p and top-k sampling don't significantly increase performance, we start by executing a preliminary BLEU evaluation to compare greedy decoding with top-k=50 and top-p=0.95 – this latter combination is believed to "avoid very low ranking tokens while preserving some dynamic selection" [50]. More aggressive combinations with higher values for both top-k and top-p or a preference for (internal) random sampling could in theory increase the model's creativity, but that could come with a decrease on correctness of prediction. The results confirm Alphacode's, since top-p + top-k is only marginally better than greedy search (35.6889 vs 35.4580). We proceed to use the best performing method for the rest of evaluation.

We now evaluate the 3 metrics above mentioned for 1 and 10 (external) samples, with temperature= 0.2 and temperature=0.6, respectively (higher sampling number infeasible due to resource limitations). We look into studying how (external) sampling affects performance and to directly compare all models.

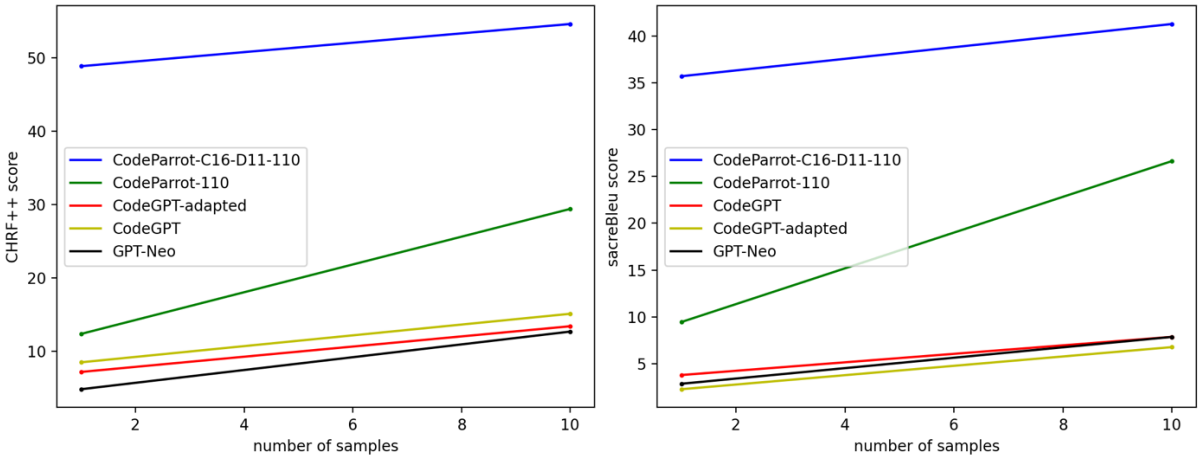

Figure 19: (left) CHRF++ score for 1 and 10 samples for each model, on Django test dataset

Figure 20: (right) SacreBleu score for 1 and 10 samples for each model, on Django test dataset

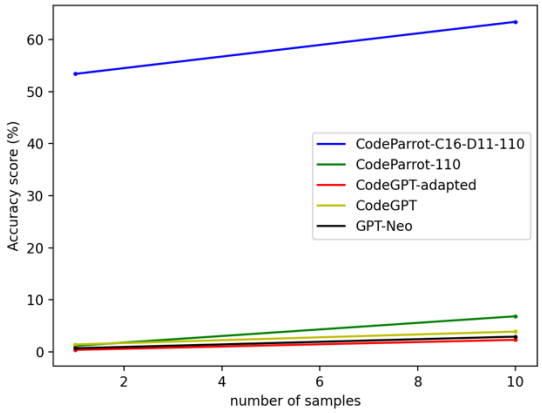

Figure 21: Accuracy score for 1 and 10 samples for each model, on Django test dataset

From Figures 19 – 21, we observe that CodeParrot-CoNaLa-16-Django-11-100 significantly outperforms the other models on all metrics and that (external) sampling consistently improves performance. In addition, similar gains between BLEU and CHRF++ scores for our model indicate that decodings smaller than 4 tokens are similarly well generated.

As a secondary point, we can also observe the importance of a code pretraining phase over a natural language one (CodeParrot and CodeGPT are consistently second and third ranked), and that a mixture in pretraining doesn't (surprisingly) improve results.

With completeness as an infeasible aim, the development and evaluation goal becomes (in practice) to find the highest number of tokens in the correct position so that the overall structure may help developers. So, the set of metrics calculated show us that the finetuning process causes the generation of code snippets with longer correct character and word sequences. Hence, we argue that generating code snippets from pseudocode-like descriptions (Django dataset) is improved .

3.3 Conclusion

We present a literature review for the problem of Code Generation from Natural Language, which allows us to execute an educated development plan, based on industry-wide methodologies and state of the art technologies.

We successfully finetune a decoder-only transformer into generating one-line python snippets from natural language descriptions. We find that CodeParrot-CoNaLa-16-Django-11-100 is capable of generating higher quality code than its baseline and other models of comparable size and pretraining for pseudocode-like descriptions. We report a BLEU score of 41.26 on the Django test dataset.

On the other hand, the model's limitations are clear. First, it struggles to generate code of similar quality from slightly more complex descriptions. Furthermore, it doesn't capture variable names from prompt and doesn't provide guarantees of being readily runnable (completeness).

As secondary conclusions, we empirically find that standardizing the input format and that (external) sampling combined with temperature tweaking are effective ways of increasing performance. We find that a code pretraining phase is of higher importance than a natural language one for the problem of Code Generation from Natural Language.

Regarding resource limitations, one quite obvious conclusion is that we can't reach SOTA performance and that both size and computational power are the main obstacles. Techniques like gradient accumulation and dynamic padding help making training more effective. Finally, for the democratization of language models to happen, smaller and more effective models are necessary, perhaps with an evolution analogue to the personal computer since the mainframe.

3.4 Suggestions for Future Work

Regarding future work for this particular project, we suggest looking into merging the 2 finetuning phases into a single one so that the model learns a balanced distribution of both corpus, which should make it generalize better. In addition, improving training quality through variable masking and through an improved method for in-context learning may help overcome some of the model's limitations.

Regarding future work for the problem of Code Generation from Natural Language, it seems clear that improving performance through an increase in model size should not be the main objective of research. As an alternative, we suggest looking into techniques that make training more effective, into creating more efficient learning objectives, and into defining a metric that defines a model's performance per unit of size (for example: BLEU score per 100M parameters). Furthermore, techniques to guarantee completeness both in length and interpretability can open the door for industry-wide usage of better evaluation methods for this problem – for example pass@k and unit tests.

List of References

[1] Alberto Romero. 2021. Yet Another Largest Neural Network — But why?.[Online]. [Accessed 20 March 2022]. Available: https://towardsdatascience.com/yet-another-largest-neural-network-but-why-f48d231972a9

[2] Aleksey Bilogur. 2020. Notes on GPT-2 and BERT models. Online]. [Accessed 20 March 2022]. Available:https://www.kaggle.com/code/residentmario/notes-on-gpt-2-and-bert-models/notebook

[3] Ashish Vaswani et al. 2017. Attention is All You Need. [Online]. Available:

https://arxiv.org/abs/1706.03762

[4] Alec Radford et al. 2018. Improving Language Understanding by Generative Pre-Training. [Online]. Available: https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf

[5] Alec Radford et al. 2019. Language Models are Unsupervised Multitask Learners. [Online]. Available:https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[6] Ahmed S. Soliman. 2022. CoNaLa Dataset for Code Generation. [Online]. [Accessed 20 March 2022]. https://huggingface.co/datasets/AhmedSSoliman/CoNaLa

[7] Ari Seff. 2021. What are Transformer Neural Networks? [Online]. [Accessed 22 March 2022]. Available: https://www.youtube.com/watch?v=XSSTuhyAmnI

[8] Aleksey Bilogur. 2021. A brief history of learning rate schedulers and adaptive optimizers. [Online]. [Accessed 20 March 2022]. Available: https://spell.ml/blog/lr-schedulers-and-adaptive-optimizers-YHmwMhAAACYADm6F

[9] Boaz Shmueli. 2019. NLP metrics made simple. [Online]. [Accessed 22 March 2022]. Available: https://towardsdatascience.com/nlp-metrics-made-simple-the-bleu-score-b06b14fbdbc1

[10] Boris Dayma, Rohan Anil. 2022. Evaluation of Distributed Shampoo. [Online]. [Accessed 22 March 2022]. Available: https://wandb.ai/dalle-mini/dalle-mini/reports/Evaluation-of-Distributed-Shampoo--VmlldzoxNDIyNTUy

[11] Carnegie Mellon University NeuLab and STRUDEL Lab. 2018. CoNaLa: The Code/Natural Language Challenge. [Online]. [Accessed 20 March 2022].https://conala-corpus.github.io/

[12] CodeSearchNet. 2019. Data Exploration. [Online]. [Accessed 20 March 2022]. Available:https://github.com/github/CodeSearchNet/blob/master/notebooks/ExploreData.ipynb

[13] Chip Huyen. 2019. Evaluation Metrics for Language Modeling. [Online]. [Accessed 22 March 2022]. Available: https://thegradient.pub/understanding-evaluation-metrics-for-language-models/

[14] Christopher Thomas.2019. Recurrent Neural Networks and Natural Language Processing. [Online]. [Accessed 22 March 2022]. Available: https://towardsdatascience.com/recurrent-neural-networks-and-natural-language-processing-73af640c2aa1

[15] Data Science Process Alliance. 2022. What CRISP-DM? [Online]. [Accessed 20 March 2022]. Available from: https://www.datascience-pm.com/crisp-dm-2/

[16] David Price et al. 2000. NaturalJava: A Natural Language Interface for Programming in Java. [Online]. Available: https://www.cs.utah.edu/~riloff/pdfs/iui2000.pdf

[17] Deepak Narayanan. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. [Online]. Available: https://arxiv.org/pdf/2104.04473.pdf

[18] EleutherAI. 2021. GPT-Neo. [Online]. [Accessed 20 March 2022]. Available: https://github.com/EleutherAI/gpt-neo

[19] Frank F. Xu et al. 2021. In-IDE Code Generation from Natural Language: Promise and Challenges. [Online]. Available: https://arxiv.org/abs/2101.11149

[20] Fernando Martínez-Plumed. 2021.CRISP-DM Twenty Years Later: From Data Mining Processes to Data Science Trajectories. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8943998

[21] Greg Little & Robert C. Miller. 2006. Translating Keyword Commands into Executable Code [Online]. Available: http://people.csail.mit.edu/glittle/Papers/Keyword%20Commands%20UIST%202006.pdf

[22] Hajo Weimer et al. 2019. Data Mining Methodology for Engineering Applications (DMME)—A Holistic Extension to the CRISP-DM Model. [Online]. Available: https://www.mdpi.com/2076-3417/9/12/2407

[23] Hendrig Sellik. 2019. Natural Language Processing Techniques for Code Generation Available: https://repository.tudelft.nl/islandora/object/uuid:3a6625b0-21da-4746-bab3-f409eed93f30/datastream/OBJ/download

[24] Huggingface. 2021.Transformer models: Decoders. [Online]. [Accessed 20 March 2022]. Available: https://www.youtube.com/watch?v=d_ixlCubqQw

[25] Huggingface. 2021. Summary of the tokenizers. [Online]. [Accessed 20 March 2022]. Available from: https://huggingface.co/docs/transformers/tokenizer_summary#bytepair-encoding-bpe

[26] Huggingface. 2021. What is dynamic padding? [Online]. [Accessed 20 March 2022]. Available from: https://www.youtube.com/watch?v=7q5NyFT8REg

[27] Huggingface. 2020. How to generate text: using different decoding methods for language generation with Transformers. [Online] [Accessed 20 March 2022]. Available: https://huggingface.co/blog/how-to-generate

[28] Huggingface. 2021. Perplexity of fixed-length models. [Online]. [Accessed 22 March 2022]. Available: https://huggingface.co/docs/transformers/perplexity

[29] Huggingface. 2022. Metric card for SacreBLEU. [Online]. [Accessed 22 March 2022]. Available: https://github.com/huggingface/datasets/tree/master/metrics/sacrebleu

[30] Huggingface. 2022. Metric card for chrF++. [Online]. [Accessed 22 March 2022]. Available: https://github.com/huggingface/datasets/tree/master/metrics/chrf

[31] Ilya Loshchilov, Frank Hutter. 2016. SGDR: Stochastic Gradient Descent with Warm Restarts. [Online]. Available: https://arxiv.org/abs/1608.03983v5

[32] Incheon Paik and Jun-Wei Wang. 2021. Improving Text-to-Code Generation with Features of Code Graph on GPT-2. [Online]. Available: https://www.mdpi.com/2079-9292/10/21/2706/htm

[33] Jacob Devlin et al. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139.pdf

[34] Jian Li et al. 2017. Code Completion with Neural Attention and Pointer Networks. [Online]. Available: https://arxiv.org/abs/1711.09573

[35] Jiho Shin, Jaechang Nam, 2021. A Survey of Automatic Code Generation from Natural Language. [Online]. Available: http://jips-k.org/digital-library/2021/17/3/537

[36] Kishore Papineni. 2002. Corpus-based Comprehensive and Diagnostic MT Evaluation: Initial Arabic, Chinese, French, and Spanish Results. [Online].Available: https://web.archive.org/web/20160304062044/http://mt-archive.info/HLT-2002-Papineni.pdf

[37] Lewis Tunstall et al. 2021. How do Transformers work? [Online]. [Accessed 20 March 2022]. Available: https://huggingface.co/course/chapter1/4?fw=pt

[38] Leandro von Werra. 2018.Training CodeParrot from Scratch. [Online]. [Accessed 20 March 2022]. Available: https://huggingface.co/blog/codeparrot

[39] Leandro von Werra. [2018]. CodeParrot. [Online]. [Accessed 20 March 2022]. Available: https://huggingface.co/lvwerra/codeparrot?text=def+hello_world%28%29%3A

[40] Leandro von Werra. [2018]. CodeParrot Small. [Online]. [Accessed 20 March 2022]. Available: https://huggingface.co/lvwerra/codeparrot-small?text=def+hello_world%28%29%3A

[41] Leandro von Werra. [2018]. CodeParrot Dataset Cleaned. [Online]. [Accessed 20 March 2022]. Available: https://huggingface.co/datasets/lvwerra/codeparrot-clean

[42] Leo Gao et al. The Pile. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modelling. [Online]. Available: https://arxiv.org/abs/2101.00027

[43] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. [Online]. Available: https://arxiv.org/abs/2107.03374

[44] Maxime. 2019. What is a Transformer? [Online]. [Accessed 22 March 2022]. Available: https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04

[45] Musstafa. 2021. Optimizers in Deep Learning. [Online]. [Accessed 20 March 2022]. Available: https://medium.com/mlearning-ai/optimizers-in-deep-learning-7bf81fed78a0

[46] Michael Phi. 2020. Illustrated Guide to Transformers- Step by Step Explanation. [Online]. [Accessed 22 March 2022]. Available: https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

[47] Nikita Kozodoi. 2021.Gradient Accumulation in PyTorch. [Online]. [Accessed 22 March 2022]. Available: https://kozodoi.me/python/deep%20learning/pytorch/tutorial/2021/02/19/gradient-accumulation.html

[48] Noam Chomsky. 1957. Syntactic Structures. [Online]. Available: https://doubleoperative.files.wordpress.com/2009/12/chomsky-syntactic-structures-2ed.pdf

[49] O'Keefe et al. 2019. Comment regarding request for comments on intellectual property protection for artificial intelligence innovation. [Online]. Available: https://perma.cc/ZS7G-2QWF

[50] Patrick Von Platen. 2020. How to generate text: using different decoding methods for language generation with Transformers. [Online]. [Accessed 22 March 2022]. Available: https://huggingface.co/blog/how-to-generate

[51] Pengcheng Yin et al. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. [Online]. Available: https://arxiv.org/pdf/1805.08949v1.pdf

[52] Priya Shree. 2020. The Journey of Open AI GPT models. [Online] [Accessed 20 March 2022]. Available: https://medium.com/walmartglobaltech/the-journey-of-open-ai-gpt-models-32d95b7b7fb2

[53] Renita Priya. 2017. A DEEP DIVE INTO AUTOMATIC CODE GENERATION USING CHARACTER BASED RECURRENT NEURAL NETWORKS. [Online] [Accessed 22 March 2022]. Available: https://scholarworks.calstate.edu/downloads/h415pc67p

[54] Research Computing Leeds. 2022. Batch jobs. [Online]. [Accessed 20 March 2022]. Available: https://arcdocs.leeds.ac.uk/usage/batchjob.html

[55] Rüdiger Wirth and Jochen Hipp. 2000. CRISP-DM: Towards a standard process model for data mining. [Online]. Available: http://www.cs.unibo.it/~danilo.montesi/CBD/Beatriz/10.1.1.198.5133.pdf

[56] Shaeke Salman, Xiuwen Liu. 2019. Overfitting Mechanism and Avoidance in Deep Neural Networks. [Online]. Available: https://arxiv.org/abs/1901.06566

[57] Sanket Doshi. 2019. Various Optimization Algorithms For Training Neural Network. [Online]. [Accessed 20 March 2022]. Available:https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6

[58] Shuai Lu et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. [Online]. Available: https://arxiv.org/pdf/2102.04664.pdf

[59] Tao Lei et al. 2013. From Natural Language Specifications to Program Input Parsers. [Online]. Available: https://people.csail.mit.edu/taolei/papers/acl2013.pdf

[60] Tensorflow. 2021. Transfer learning and Transformer models (ML Tech Talks). [Online]. [Accessed 22 March 2022]. Available: https://www.youtube.com/watch?v=LE3NfEULV6k

[61] Thanh Nguyen. 2016. T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2950290.2983931

[62] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-Form Queries. [Online]. Available: http://laraserver.epfl.ch/~gvero/anycode.pdf

[63] Tom B. Brown et al. 2021. Language Models are Few Shot Learners. [Online]. Available:

https://arxiv.org/pdf/2005.14165.pdf

[64] Tomas Mikolov et al. 2013. Efficient Estimation of Word Representations in Vector Space. [Online]. Available: https://arxiv.org/pdf/1301.3781.pdf

[65] Wikipedia. 2022. Code Generation. [Online]. [Accessed 22 March 2022]. Available: https://en.wikipedia.org/wiki/Code_generation

[66] Wikipedia. 2022. Language Model. [Online]. [Accessed 22 March 2022]. Available: https://en.wikipedia.org/wiki/Language_model

[67] Wikipedia. 2022. Word2Vec. [Online]. Available: https://en.wikipedia.org/wiki/Word2vec

[68] Wikipedia. 2022. Recurrent Neural Network. [Online]. [Accessed 22 March 2022]. Available: https://en.wikipedia.org/wiki/Recurrent_neural_network

[69] Wikipedia. 2022. Statistical Machine Translation. [Online]. [Accessed 22 March 2022]. Available: https://en.wikipedia.org/wiki/Statistical_machine_translation

[70] X. V. Lin et al. 2017. Program synthesis from natural language using recurrent neural networks. [Online]. Available: https://dericpang.com/static/e511b5f8f6060087dc29dae9fed86200/tellina-tr170510.pdf

[71] Yujia Li et al. 2022. Competition-Level Code Generation with AlphaCode. [Online]. Available: https://storage.googleapis.com/deepmind-media/AlphaCode/competition_level_code_generation_with_alphacode.pdf

[72] Yusuke Oda et al. 2015. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation. [Online]. Available: https://ieeexplore.ieee.org/document/7372045

[73] Zhangyin Feng et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139.pdf

Appendix A
Self-appraisal

A.1 Critical self-evaluation

During the development of this project, the correct methodologies were used to guarantee that work was evenly distributed throughout the available time. In reality, development wasn't much different from what can be seen on the Gantt chart in Chapter 2.1. This can be seen as a good point and a representation of good planning. In addition, both 'Literature Review' and effective study of 'Machine Learning' proved successful in providing the basis and understanding needed for development.

On the other side, it became evident after the first semester that the initially defined milestones were too broadly defined, which made work effectiveness decrease significantly. Thus, the outputs from the first semester were a compilation of unstructured research notes and a couple of try runs with transformer models, but in hindsight, more emphasis should have put on creating structures that could have been iteratively improved, instead of completely transformed in semester 2. For example, a full draft of the literature review should have been done (which would have avoided a "reminder stage" during the second semester), the code structure should have been built end-to-end (according with CRISP-DM methods) and with model outputs in mind. This last point means that we should have predicted better the size of such output and separate them from the developed source code; this way would have permitted a traditional git approach to version control from the beginning.

During the second semester, we started by correcting some of the issues regarding development objectives and clearly defined them. On the other hand, these objectives proved to be too close to state of the art performance, and didn't provide us with a good starting point. This caused development setbacks like trying to go over the limit of the available computational resources and exploring ambitious techniques to overcome such problems (for example the use distributed computing, and evaluation techniques that required code completeness). Eventually, the 'real' objectives were found empirically, but this brute-force approach affected the balance between tasks, especially report writing. This could have been avoided with a more bottom to top approach to the whole development, and this is a clear point for improvement.

On the other hand, semester 2 proved to be a challenging but exciting period where development obstacles were faced with perseverance. 'Data Pre-processing and Model Training' was done extensively and there existed many try runs of smaller success with other

dataset-model combinations. We argue that not having gone end-to-end early in the development may have left some early evaluation unreported, but we also strived for the best model and that's the one we presented.

A.2 Personal reflection and lessons learned

Personally, the main lessons that I have learned with this project are to avoid blind optimism and to start simple.

Firstly, there were times during development when I knew what I had to do, but not in great detail, and that affected how focused I was during the time allocated to this project. Secondly, my initial ambitions were to fully mimic state of the art performance without comprehending the resources needed to such endeavour.

Thus, I believe that even though I tend to define bigger pictures and holistic views of systems, I learned to be much more focused on the details and that they are the ones that make a difference. In addition, these focus on details should have a structured approach so that we can 'build' a pyramid of knowledge with the correct foundation. I found myself trying to execute leaps of knowledge early in development and that was definitely a lesson learned.

Another point that I found hard was to find the correct level of detail to this report. Even though I just mentioned the importance of details, I recognize that the field of Machine Learning and the field of Code Generation from English didn't have a historically linear development and that there's lots of details that could itself be a full research paper, so aiming the report for general computer science was definitely a challenge.

I correctly predicted development roadblocks when I did the planning, but I believe that I could have handled such circumstances better. For example, I wouldn't have taken so much time off reporting writing if I tackled some of the problems faced with model training with more theory-based methods, i.e. looking for scientifical explanations to why things were not going as intended to. In addition, it was my first time dealing extensively with GPU batch jobs and it was hard for me to predict their running time and parallelise them with other tasks.

As a sidenote, the small extension needed at the end was due to circumstances beyond my reach.

A.3 Legal, social, ethical and professional issues

Machine Learning applications have a multitude of possible issues that affect Code Generation from Natural Language.

A.3.1 Legal issues

Legally, most models are trained on data from publicly available repositories like GitHub, which even though is considered “fair use” [49][43], doesn’t overwrite the fact that each user holds intellectual property rights over their code and repositories.

This may cause issues if models gain an overreliance on particular source code files, despite their generative nature. In addition, we recognize that even if an end-user holds the last saying over using code generated from a model, he should not be responsible for recognizing such patterns.

A.3.2 Social issues

As ML models grow in size and capacity of capturing meaning, the issue of bias amplification becomes especially harmful. In fact, data that is fed into model training is written by humans and therefore may represent their flaws and biased opinions on important matters like gender, race, class, etc [43].

So, these flaws may be amplified during training (a process that still holds a high degree of uncertainty) and models may even re-ignite Humanity’s past issues by looking into old data.

A.3.3 Ethical issues

First, over reliance on high preforming models can cause a user’s lack of understanding of the system, which can lead to the generation of source code with arbitrary capabilities [043]. On the other hand, a user’s intentions may not be understood correctly if there exists a misalignment on the model’s comprehension (a common case nowadays) which may equally lead to unpredictable code [43]. Thus, generated code may cause harm to individuals and entities and should require human supervision before execution.

Second, the same exciting ML capabilities that may improve the world, can also unfortunately be used to cause harm. For example, the recent proliferation of fake news has gotten worse due to transformer capabilities. Hence, we fear that similar episodes may happen with code generation from natural language and suggest further research on ways of distinguish between generated outputs and human ones.

Thirdly (and as briefly seen in Chapter 2.2), an increase in model size has been one of the main factors driving performance increases and pre training large models consumes enormous amounts of energy (Fig. 22). So, deciding whether to pretrain a new model or not may be one of most important environmental choices a human can do today. Furthermore, we suggest further research that seeks improving performance by improving learning effectiveness

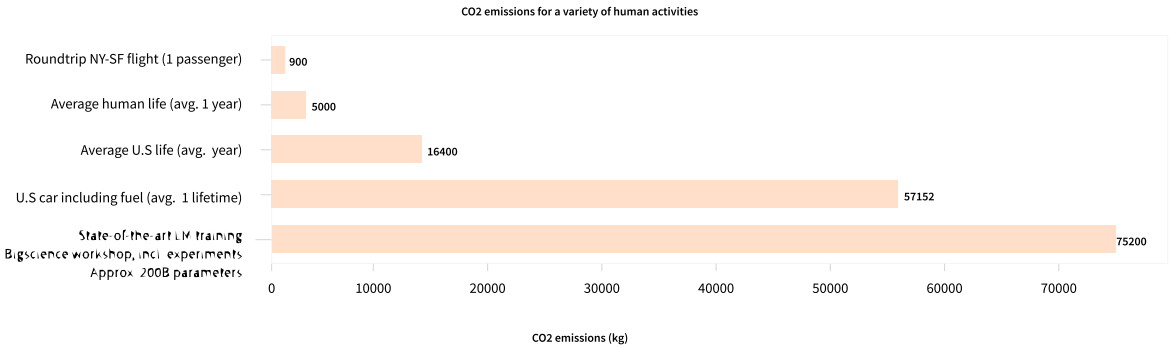

Figure 22: CO2 emissions of pretraining a large transformer model compared with other human activities [37].

A.3.4 Professional issues

It is still unpredictable how Code Generation from Natural Language may affect the job market but its impact can be weighed.

Programming technologies have historically been built to increasingly increase the levels of abstraction and allow end users to build more with less code. This trend should continue whilst “lowering the barrier to building software and enabling entirely new kinds of software to be built” [43].

On the other hand, such developments may reduce the need for programmers, but it is unlikely that the need for software developers decreases. In addition, we argue that future roles in the field may reduce the need for programming but that new responsibilities and dynamics will arise.

Appendix B
External Materials

For this project's development, the CodeParrot baseline models for finetuning were provided by Leandro von Werra and Huggingface [038]. In addition, the datasets CoNaLa and Django are provided by Carnegie Mellon University NeuLab and STRUDEL Lab [11], and Yusuke Oda et al [72], respectively.

During evaluation, we use the following readily available models for comparisons: CodeGPT [58], CodeGPT-adapted [58], GPT-Neo [18], and CodeParrot (mentioned above).

Appendix C
Sample Outputs

This appendix includes some example outputs from the developed model CodeParrot-CoNaLa-16-Django-11-110M, decoded with internal sampling (temperature = 0.2, num_return_sequences = 3, top-k = 50, top-p=0.95).

```
-prompt: params is an empty list

bits = [ ] ] = [ ] ] ] if not, else_
bits = [ ] ] = [ ] ] if not,'''
bits = [ ] ] = [ ] ] ] if not, else_

reference solution: params = [ ]

-prompt: if not

else : not : ] : ] : ] : value is_si :
else : : ] : ] : ] : value is None,'''
else : ] : not in_bytes : is_bytes is_bytes

reference solution: else :

-prompt: check if all elements in list `myList` are identical

check = [ : - 1 ] for check in list ( ) ] if
check = [ : - 1 ] for check in all if sql ] ]
check = [ : - 1 ] for check in all if all_ ]

reference solution: all(x == myList[0] for x in myList)


-prompt: define the function remove_move with an argument name.

def remove_module_name_name ( name ) : : ] :
def remove_decorator_compatible ( name ) : : ] : : ]
def remove_view ( name ) : : ] : self. remove_

reference solution: def remove_move ( name ) :
```

```
-prompt: append val to self.vals

self. append ( val ) ). append ( val ) ) : val
self. append ( val ) ). append ( val ) ) : val
self. append ( val ) ). append ( val ) ) : val

reference solution: self.vals.append(val)


-prompt: append string "airplane" to foo
ex. append ( 'airpp' ) ) : '
ex. append ( codecs ) ) +'' ''
ex. append (''''' '

reference solution: foo.append(“airplane”)


-prompt: call the function force_text with 2 arguments: value under
the is_dst key of the time.tzname and DEFAULT_LOCALE_ENCODING

locale_text (  DEFAULT_text [ is_safe ], DEFAULT_LOCALE_ENCODING )
locale_text (  DEFAULT_text (  DEFAULT_DB_ALIAS
locale_text (  DEFAULT_text (  DEFAULT_DB_ALIAS

reference solution: force_text( time.tzname[“is_dst”],
DEFAULT_LOCALE_ENCODING )


-prompt: sum all elements of list lst

join ( list ( list ( list ( ) ) ) : : - 1
join ( list ( list ( list ( list_ ) ) ) : :
where [ : ]. extend ( ) ] ) ] = self. Extend

reference solution: sum(lst)
```