Democratization of Large Language Models Trained on Code

Martin O.-Cañavate Ortiz

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2023

Abstract

In this report we introduce a new strategy for developing large language models aimed at code modelling tasks. We conduct a thorough experimentation of architecture choices, parameter fine-tuning, training data usage and training techniques.

For this we we have collected three clean datasets, the biggest one roughly 800GB. And we release a family of models developed using our experimental conclusions. We test our models using the HumanEval (Chen et al., 2021) golden standard, where our 2.5B-parameter model obtains a superior score in single-try code generation than OpenAI's equivalent 2.5B model (36.62 vs 35.42).

From the results of our work, we provide valuable insights into the capabilities of large language models trained on code, and how, despite the secretiveness of research in the big tech companies, with the publicly available research in the field, readily available free data and without the need of mind-shattering computational resources, a small research team can develop and contribute in the field of specific-task modelling — in our case code modelling.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Martin O.-Cañavate Ortiz*)

Acknowledgements

Thanks to my project supervisor Brian Mitchell, who provided help throughout the complicated journey of carrying out this project whenever I needed him most.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large sequence prediction models, widely known as large language models (LLMs or LMs) became a popular tool in the field of natural language processing (NLP) in the 2020s due to their ability to process and understand large amounts of text data. LMs are commonly used in NLP tasks, such as language translation, sentiment analysis, and topic classification. By training these models on large amounts of text data, they learn the patterns and structures of natural language. Recently, LMs, in particular autoregressive models, have also been applied to the field of programming language processing, where they can be used to model code and generate new code. This can be useful for tasks such as code completion and error correction, as well as to analyse code and extract information, such as code commenting and documentation.

One potential challenge in using LMs for code modelling is the highly structured and syntax-dependent nature of programming languages, which makes it difficult for the models to learn the complex rules and patterns of code. However, recent advances in machine learning training methods have made it possible to improve the performance of LMs on code modelling tasks. This use of machine learning for modelling code has the potential to greatly improve the efficiency and accuracy of programming tasks, and is an active area of research in the field of NLP.

Despite the success of large language models of code, the most powerful models are not open source. Training large language models on code requires a significant number of computational resources, which is often costly. on top of that, training these large models require huge sets of training data, which can also be hard to find. That is why, many of the large language models trained on code are not publicly available and its development and architecture is only known by the well-resourced companies that have the resources to develop and maintain them. For example, Codex, which is the best performing model in this field that has been recently deployed — and it is producing good results as GitHub's Copilot tool (an in-IDE developer assistant that automatically generates code based on the user's context) — has limited use access and the model's architecture specifics, weights, training strategy and training data have not been made public to the general scientific community.

This need of enormous resources to be able to produce LMs for code modelling prevents small research teams from investigating on their own, and the big companies that invest their resources on developing them are secretive about their advances. This prevents researchers from studying the model in detail and from building upon the work that went into training the model. It also prevents the community from contributing to the development of the model and from improving it, and as a result, the potential benefits of these models may not be fully realized. It is important for the development of these models to be transparent and accessible, so that the research community can continue to advance the state of the art in this field.

We hypothesised that using publicly researched model architectures, readily available data, and implementing the appropriate training methods, we could devise a model building strategy that could rival the current privately-owned ones. By making our methods public, we hope to help other small research teams in the field.

We believe that it is important to make large language models of code more accessible and available to a wide range of researchers and organizations. By democratizing these models, more people will be able to experiment with them and conduct research in this field. This can help promote collaboration and the sharing of ideas within the research community, which can lead to more rapid advances in the field, and could ultimately benefit the broader community of model developers and researchers. Additionally, making these models more widely available could also help to ensure that they are developed and used in an ethical and responsible manner. Overall, democratization can help to ensure that the benefits of large language models are more widely available and can be used to benefit society as a whole.

Chapter 2

Background

In this section we will cover the basic knowledge needed to appreciate this report — if you are familiar with transformer models you may feel free to skip this the first subsections. We cover how autoregressive transformer models work, which are the current state of art models used for code modelling, and we also cover the evaluation methods and working frameworks that are used in the field to develop said models.

2.1 Machine Learning

A machine learning (ML) model is a mathematical representation of a real-world process or system. It is trained on data and makes predictions or decisions based on that training. A **language model** (LM) is a type of ML model that is trained to process and understand natural language data such as text. It assigns probabilities (likelihood) to words and sentences, thus being able to recognize what combinations of words are likely and what combinations are unlikely. As we have already outlined on the introduction, LMs can be used to understand programming languages and thus produce code.

An artificial neural network is a computational model is inspired by the structure and functions of the biological nervous system. It consists of many interconnected nodes (can be interpreted as "neurons"), which are organized into layers. Each node receives input from other nodes and performs a simple mathematical operation on that input to produce an output. The output is then passed on to other neurons in the next layer, and so on. The **parameters** of an artificial neural network are the variables that the network uses to make these predictions. These parameters can include the weights of the connections between neurons, the thresholds that determine when a neuron is activated, and other factors that influence the network's behavior. Neural networks can be adapted based on some data and then they can be used to make predictions on similar data.

Deep learning is a type of machine learning that uses artificial neural networks to model high-level abstractions in data. It is called "deep" learning because the neural networks have multiple layers of interconnected nodes, which allow them to learn and represent data with multiple levels of abstraction. Deep learning algorithms use these artificial neural networks to automatically learn and extract high-level features from

large amounts of data, without the need for explicit programming. This allows them to solve complex problems that were previously difficult or impossible for traditional machine learning algorithms to tackle.

2.1.1 Language Models

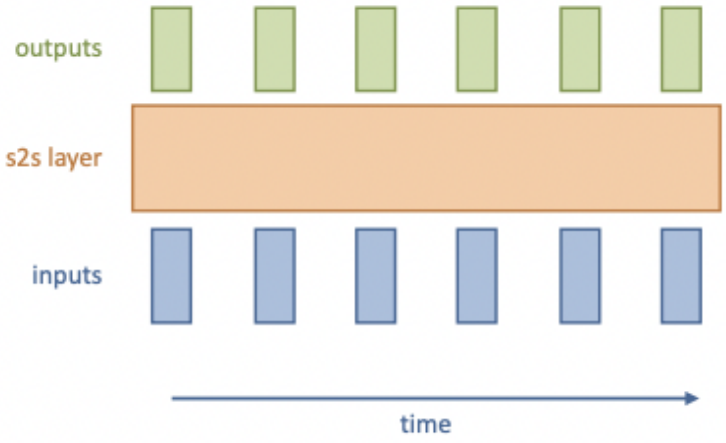
A language model is a type of sequence prediction machine learning neural network that is trained to predict the most likely sequence output for a given input, where both the input and output are sequences of words. The basic LMs are called sequence-to-sequence (S2S) language models, work as follows: given a input sequence of words $v_1, v_2, ..., v_n$, an LM estimates the most probable sequence of words $(w_1, w_2, ..., w_n)$ and returns it as an output Figure 2.1.

By augmentating the complexity and modifiying the structure of this S2S process, several large language models have been developed: n-gram models, recurrent neural networks (RNN), convolutional neural networks (CNN), **transformer-based models**... These language models are usually trained on a large corpus of text data to capture the statistical properties of the language, such as word frequencies and grammatical rules. And they are often used in natural language processing (NLP) tasks, such as machine translation, speech recognition, and text generation.

2.1.2 Basic Self-Attention

Transformers models are a type of deep learning models that were introduced in 2017. Transformer models are the models that produce best results for language modelling and

Figure 2.1: Basic Seq2Seq process. Input and output sequences consist of vector sequences, which are converted from words to vectors by a hidden layer called the embedding layer, and from vectors to words by another hidden layer called the softmax layer. Observe how the sequences are ordered — the direction in which they extend called the time direction — and also take notice how the input and output sequence have the same length. (Pictures in the background section have been obtained from Vrije Universiteit Amsterdam's Deep Learning course.



therefore code modelling (OpenAI's GPT and Codex model's are transformer models). As such, in this report we focus on this model family. Transformer models consist in a type of artificial neural network which uses **self-attention** mechanisms to process input data. But what is this self-attention mechanism and why is it so revolutionary?

At heart, self-attention is a simple operation used to calculate output vectors from an input vector sequence. It takes the form of a weighted sum

$$y_i = \sum_j w_{ij} x_j$$

where y_i is the ith output vector, x_j the jth input vector and w_{ij} is the pondering weight. By calculating this sum for every i, we get the output sequence. Many neural networks implement a similar mechanism, but what makes self-attention special is that w_{ij} is not a learned parameter of the model, but is computed from the inputs in the following way:

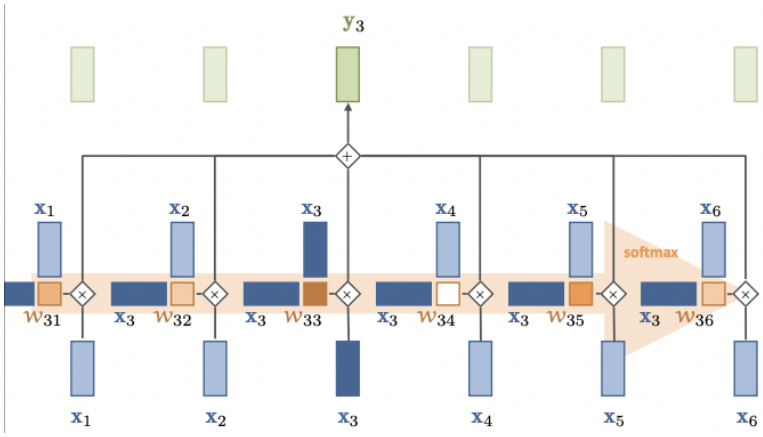
$w'_{ij} = x_i^T x_j$ and then $w_{ij} = softmax(w'_{ij})$ where $softmax(w'_{ij}) = \frac{exp(w'_{ij})}{\sum_j exp(w'_{ij})}$ is a function that ensures that all w_{ij} are positive and they sum to one. A visualization of how y_3 would be calculated given an input of length 6 can be seen in Figure 2.2.

Self-attention can be efficiently computed if we vectorise the operations. Therefore, for the matrix X where every ith column is the input vector x_i, we can calculate the matrix Y where every jth column is the output vector y_j in the following way:

$W' = X^T X$, then $W = softmax(W')$, and finally $Y^T = WX^T$

In self-attention, at its simplest, w_{ii} (x_i to y_i) usually has the most weight — due to its dot product calculation nature. This means that the i[th] output vector will be most similar to the ith input vector, and this is not always ideal. Also, the self attention mechanism itself has no parameters, the embedding layer that generates the X_is vectors form the word sequence is what drives the learning. On top of that, notice how the order of the input sequence only affects the outputs order, but not its content. This are all effects of self-attention not modelling the data as sequential, but as a set. This disadvantages — which we will show you now how to fix on the next part — are the price to pay for a

Figure 2.2: Self-attention example



virtually limitless attention window; every output vector is at the same mathematical distance form every input vector.

2.1.3 Improved Self-Attention

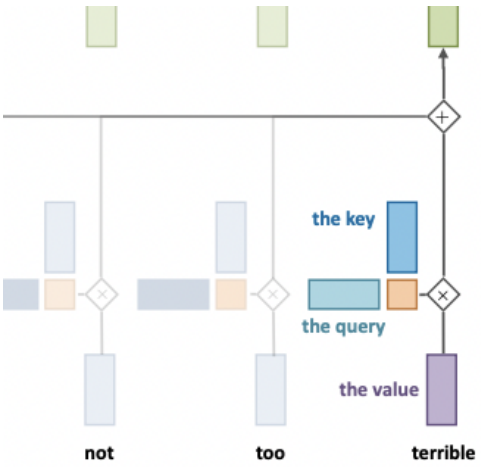
To improve the self-attention we add extra features that help us add complexity to the mechanism. This helps us increase its flexibility and make it more powerful. If we increase the complexity (size) of the word-representing vectors, the dot product between vectors will increase proportionally to the vector dimension (d) by the factor $\sqrt{d}$. Therefore, we first want to scale the self-attention by setting $w'_{ij} = \frac{x_i^T x_j}{\sqrt{d}}$ We can also recognise how every input vector takes three different roles in the self-attention mechanism, Figure 2.3.

First, as the jth vector used in the weighted sum that provides the output vector, indicated as 'value' in purple on Figure 2.3. Second, as the ith vector corresponding to the ith output vector, that it is matched against every jth input vector to calculate w_{ij}; indicated as 'query' in light blue on Figure 2.3. And third, the jth input vector that the query is matched against to calculate w_{ij}, indicated as 'key' in dark blue on Figure 2.3. These key, query and value names derive from thinking about self-attention as a soft-version of a dictionary; where every query matches every key to some degree — as determined by their dot product, and the value returned is a mixture of all the values — where the softmax-normalized dot products are the mixture weights.

Therefore, to make self-attention more powerful, we introduce some transformations for these three different roles. That way, even though the same vector is used in the roles, it can behave differently depending on the role. Therefore, the same input vector x_i will undergo the following linear transformations depending on if it is acting as a key, a query or a value:

$k_i = Kx_i + b_k$, $q_i = Qx_i + b_q$, and $v_i = Vx_i + b_v$. Where K, Q and V are weight matrices and b_k, b_q and b_v are the associated biases.

Figure 2.3: Different roles of the input vectors



This transformations make the self-attention more flexible. And by introducing the learnable parameters of the weight matrices and the associated biases we increase the capabilities of self-attention under proper training.

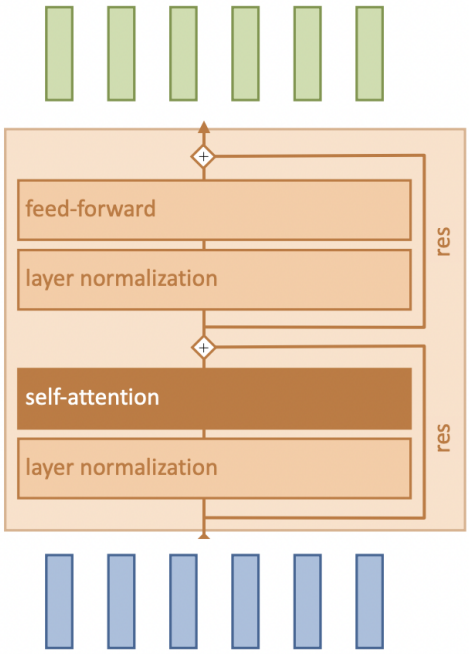
The last improvement is introducing **multi-head attention**. Which consists on applying self-attention layers in parallel. This is implemented by undergoing the input vector through a transformation that divides it in n lower-dimension vectors, then making the n vectors go independently through a different attention head (layer), and then concatenating the n vectors into a final output vector.

The improved self-attention mechanism creates a sequence to sequence layer which can be easily computed in parallel and has a virtually endless long-term memory. We should pay attention to the fact that this self-attention mechanism does not have access to the sequential structure of the input. Any output vector is equally influenced by every input vector, regardless of it is position in the input. Therefore, although we are using self-attention as a seq2seq layer, it works as a set-to-set layer.

2.1.4 Transformer Block

A transformer model is a model that uniquely uses self-attention to propagate the information between the basic input units. A transformer model is built by defining a transformer block and then stacking consecutively the transformer blocks a number of times. The structure of a transformer block can have minimal variations, but it generally has the structure in Figure 2.4.

Figure 2.4: Transformer Block. The input sequence is first normalised through a layer normalization, then the self attention is computed. A residual connection then adds the original input to the self-attention output. After the sequence goes through layer normalisation again, it is fed to the feed-forward layer. Finally, a residual connection adds the feed-forward output to the self-attention output and that is returned as the transformer's block output



The feed-forward layer is a simple neural network, usually consisting of two linear transformations with an **activation function** — non-linear transformation — in between. We can think of the linear transformations as doing a multi-variable linear regression and the activation function as a way to add complexity to the model's learning capabilities.

The feed forward layer operates with the same learned parameters — the two matrices for the linear operations and the activation function parameters — on every token of the sequence in isolation. This makes it highly parallelizable and, as we said, makes self-attention the unique propagator of information along the inputs.

Layer normalisation consists just of an individual vector normalisation, where every isolated vector is treated so that its values have mean zero and standard deviation of one. For every vector x of dimension d where the x_i token is the ith position on the vector the following operations are conducted:

Mean over token: $\mu = \frac{1}{d}\sum_i x_i$; variance over token: $\sigma = \frac{1}{d}\sum_i (x_i - \mu)^2$; standardization: $\hat{x} = \frac{x-\mu}{\sqrt{\sigma}}$; and re-scaling: $y = \gamma^T \hat{x} + \beta$.

Where y is the output, and γ and β are learnable parameter vectors.

2.1.5 Autoregressive self-attention

To model language, or code in our case, an autoregressive model — whose task is to predict the next token of an input sequence — is needed. But these transformer blocks have access to future tokens in the sequence, and hence cannot be trained in this predictive way. This is due to the fact that self-attention is a set-to-set operation, it is not a causal mechanism. In order to make self-attention causal, we use masking:

Where the attention matrix is computed the same as in Section 2.1.2, but before applying the softmax function, we set $W'_{ij} := -\infty$ if $j > i$. This means that, when applying the softmax function, W will be a lower triangular matrix. And thus, when computing our output matrix $Y^T = WX^T$ no forward connections will be used. A block that uses this causal self-attention, where the forward connections in the weigh matrix are masked out, is called a causal transformer block.

The causal transformer block allows us to build an autoregressive transformer model, as the next token predicted only depends on the previous tokens. The problem now is that, due to the set-to-set nature of self-attention, the order of the previous tokens does not matter. That is, the model will erroneously interpret the sentence `b = 2*(x + 3)` as identical to `x = 3*(b) +2`.

To make self-attention sensible to input position, we define a positional encoder layer, which takes place before the transformer blocs. The positional encoder layer adds to every input vector a positional vector that represents the input's position in the input sequence, Figure 2.5. Positional vectors usually consists of several sinusoidal functions that generate a unique vector for every integer in the input length Figure 2.6.

Figure 2.5: Positional Encoding. Example of how positional encoding works. In this case we show the usefulness in differentiating between the two 'the' present.

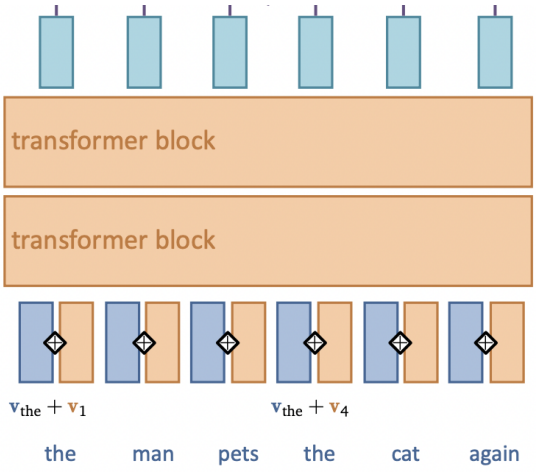
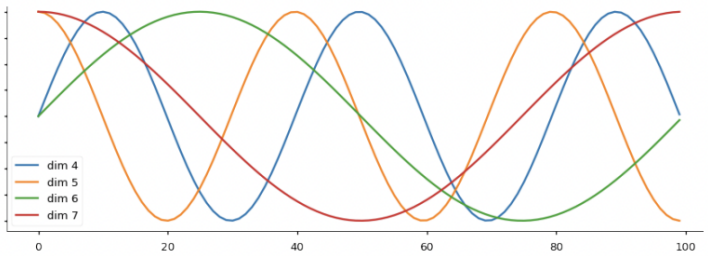


Figure 2.6: Positional Vector. Example of a positional vector of 4 extra dimensions. We can observe how a vector in the form of $< dim4, dim5, dim6, dim7 >$ will be different for at least the first 100 input positions



2.1.6 From Self-Attention to Transformer Model

Going from a simple-self attention mechanism to a large auto-regressive transformer model starts with an embedding layer to convert input sequences to vector sequences, a positional encoding layer to add the sequence position information to the vectors, and then a variable number of stacked causal transformer blocks. Thus, depending on the dimension choices of our self-attention mechanism and the number of transformer blocks, a transformer based model's architecture can be defined in several **hyperparameters**:

- The **n_layers** hyperparameter specifies the number of transformer blocks in the model.
- The **d_model** hyperparameter specifies the size of the hidden states of the self-attention layers in the model. That is, it represents the size of the K, Q and V learned weight matrices.
- The **d_ff** hyperparameter specifies the size of the feedforward sublayer in the transformer block.
- The **n_heads** hyperparameter specifies the number of attention heads that are used in the self-attention mechanism.

These hyperparameter values are what directly impact the transformer's general architecture and the number of trainable parameters that will be in the resulting model. Hence, there are virtually infinite architectures for a transformer model — depending on the hyperparameter values set.

2.2 Model Training

Once we have built the transformer model's architecture — by selecting the appropriate architecture hyperparameters — , we have to think about what values to assign to the model's parameters; those being the indices of the K, Q, V matrices and the b_k, b_q, b_V biases in the self-attention mechanism and the indices and weights of the feedforward layer for every transformer block. Machine learning model training is the process by which the learnable parameters of a machine learning model are update, where the goal of the this updating is to find the set of model parameters that result in the best possible performance on the task at hand. Performance is rated via the **loss function**, which is a measure of how well the model understands a given data. Loss functions are calculated by comparing the model's predicted output with the true output for a given input, we will explore the different mathematical ways to express this comparison in the future.

2.2.1 Tokenization

Natural language and coding language is complex and unstructured, and it is difficult for computers to process and understand it in its raw form. **Tokenization** is a preprocessing step that breaks down a piece of text into smaller, more manageable pieces called tokens. These tokens can be individual words, numbers, punctuation marks, or other elements of the text. That way, in the embedding layer of the model, a unique vector representation can be assigned to every token. By breaking the text down into these tokens, LM algorithms can more easily analyze the structure and meaning of the language and perform tasks. The tokenizer layer is trained — insdependently of the rest of the model — to choose the best tokens based on the vocabulary size we want.

2.2.2 Optimization Algorithms

Gradient descent is the **optimizer** algorithm used to find the values of the model's parameters that minimize the given loss function. Gradient descent works by iteratively updating the parameters in the direction of the negative gradient of the loss function with respect to the parameters. For every training input, the model computes its output and loss, called forward pass. Afterwards it calculates the loss gradient based on the partial derivative of the loss function with respect to each learnable parameter, called backward pass. This process of calculating the predicted output, loss, and the parameters' updates to make the output better match the training data is called backpropagation. Using this process, at each iteration the learnable parameters are moved a small step in the direction that reduces the loss function the most. The **learning rate** is the training hyperparameter that determines this step size at each. If it is too small, the algorithm takes a long time to converge, but it will eventually find a locally good solution. If it is too large, the algorithm may overshoot the minimum and fail to converge. Therefore, success of the optimization is sensitive to the choice of learning rate.

Gradient descent can become slow and inefficient when dealing with large datasets, consequence of the algorithm calculating the backpropagation algorythm for every sample in the training dataset. Gradient descent may also get stuck in local minima, which are points where the cost function has a lower value than the surrounding points but is not the global minimum. **Minibatch stochastic gradient descent** (SGD) is a variation of gradient descent, where the learnable parameters are updated using the mean of all the backpropagation gradients of a subset of the training data (known as a **batch**). Thus the parameters are updated less times — once for every batch instead of once for every sample — , which is less computationally expensive and leads to faster convergence, making this method better suited for large training datasets. SGD also takes a more erratic path towards the global minimum, which has been found to minimize local minimum convergence problems (Hinton, Srivastava, and Swersky, 2012; Li et al., 2014; Konečnỳ et al., 2015).

2.2.3 Epochs, iterations and steps

An epoch in machine learning refers to a single pass through the entire training dataset. When implementing SGD training, during each **epoch**, the model will update the parameters with every batch in the training dataset, every update in an epoch is called a **training iteration**, and the total number of updates undergone during training are called **training steps**. For example, if a model is trained on a dataset with 1000 samples, and the batch size is 100, there would be 10 training iterations in each epoch; and if the model is trained for five epochs, there will be 50 training steps. After every training step, the model can be evaluated on a **validation dataset** to assess its performance. As Gradient Descent is an iterative process, the increase of epochs will increase the fitting of the model to the training data.

2.2.4 Pretraining and Fine-Tuning

Pretraining is the process of training a machine learning model on a large, general-purpose dataset. Pretraining allows the model to learn general patterns and features that can be applied to a wide range of tasks and datasets. This initial training phase is called "pretraining" because the model is not yet ready to be used for making predictions on the specific intended task.

Fine-tuning is then process of adapting a pretrained model to a specific task. It is carried out by further training the pretrained model on a smaller, more specific dataset; adjusting the model's parameters in order to better fit the data and improve its performance on the specific task. The process where a model's previous training on a dataset benefits the model's training and predictions on another dataset is called **transfer Learning**

2.3 Model Evaluation

Model evaluation is the process of evaluating the performance of a machine learning model on a given separate test dataset in order to test how accurately it can make predictions on new data, which is the general aim of all language models. On top of

being evaluated on an evaluation dataset during training via a loss function, as we have explained inn Section 2.2, models are evaluated when fully fine-tuned, in order to evaluate the model's performance in the intended task.

The most popular loss function in the field of large LMs is cross-entropy:

$$CE = \sum y_i * log(p_i)$$

where y_i is the ground-truth label, p_i is the predicted probability of the corresponding class, and the sum is over all classes. Due to its logarithmic formula, it is easy to compute the gradient of the cross-entropy loss with respect to the model parameters, making it the most computationally efficient loss function for training deep neural networks. This and its better training behaviour compared to other loss function (Zhang and Sabuncu, 2018; Gordon-Rodriguez et al., 2020) is what makes cross-entropy the best loss function for our models.

Historically, to evaluate the the effectiveness of trained models in generating accurate and readable code that can be easily understood and maintained by developers, extrinsic evaluation on code generating models has been conducted by evaluating the generated code against reference solutions, mainly using the BLEU score. However, some recent publications have raised issues with this practice, as a reference solution does not take into account all the different ways a code function may be written as. Another problem with BLEU is that it has problems capturing semantic features specific to code (Ren et al., 2020). More recent works (Lachaux et al., 2020) have shown that optimising for semantic correctness in code does not translate to code functionality. Thus, recent research is directly evaluating models using functional correctness.

Popularised by (Chen et al., 2021), the modified **pass@k** metric is the golden standard to assess functional correctness:

$$pass@k := E_{HumanEval}[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}]$$

where $n = 200$, c is the number of correct samples generated — samples that pass the tests for the given prompt, and the expectation is taken over all the HumanEval dataset problems. The HumanEval dataset (also presented in the Codex paper) is preferred for the **pass@k** computation because, unlike other proposed evaluation datasets that contain public repositories (APPS by Hendrycks et al. (2021)), it was written after the creation of The Pile (a widely-used training dataset explained later). Therefore, the usage of HumanEval as testing dataset ensures prevention training-to-test data leakage in all our models.

2.4 Building Frameworks

In terms of programming frameworks to build transformer models, there is great consensus on what tools are to be used. `Python` is the lingua franca to develop models in machine learning, libraries are written in `C++` for better computing efficiency, but

they are developed to be used in Python. The two existing libraries for high-speed large-scale machine learning are `PyTorch` and `TensorFlow`, `PyTorch` being the better optimised for NLP models. All relevant libraries for NLP in specific are developed by `Hugging Face`, which is the biggest open source community in Transformers for NLP. A sort of GitHub for machine learning widely used by top technology companies such as Microsoft, Facebook, and Salesforce. The main libraries maintained by them are the `Transformers` library, specifically designed for developing transformer models; the `Datasets` library, designed to manipulate datasets used for model training; and `Accelerate`, developed for implementing pararell computation in model training.

2.5 Synthesis and Directions

We have now seen the usefulness of previous research conducted in the machine learning field for code modelling. We know that for autoregressive tasks such as ours, Transformer architectures work best and that the best way to evaluate autogenerated code is by its functionality. Moreover, in terms of what frameworks and libraries to use for NLP model developing, there is a lot of consensus in the field. The research we had to pursue then, was oriented towards finding the best way to use these tools to develop a transformer model for code modelling. It had not previously been researched what structure hyperparameters resulted in the most optimal transformer architecture for code modelling, what were the best techniques to train the model or what was the best data to train for code modelling.

In chapters 3 and 4 we conduct our experiments using Google's Cloud computing platform (Google Colab) — the use of a cloud computing platform allows us to access the necessary computational resources to train large language models efficiently. We walk you through the investigation process we underwent to find the most optimal combination of model architecture choices and training methods to devise the best model building strategy for code modelling LMs.

Chapter 3

Transformer Implementation

In this chapter we describe how we have investigated the best way to implement the transformer architecture for code modelling tasks. We used the `PyTorch` library and HuggingFace's `Transformers` library to build the models' architecture, and HuggingFace's own `datasets` and `accelerate` libraries for training them. We evaluated model training using cross-entropy loss and model performance using `pass@k` Section 2.3.

Due to our limited computing resources — using big models (1B+) for architecture testing was out of the question. Therefore, we used smaller models of gradually increasing sizes (Raffel et al., 2020). Once the optimal parameters had been determined for the smaller models, we extrapolated them to train larger models (Kaplan et al., 2020).

3.1 Structure Dimension Hyperparameters

Tuning structure hyperparameters is a time-consuming and iterative process, as different combinations of hyperparameters produce significantly different results and the training of each model is costly. We use grid search: a brute force method that involves training the model with a range of different hyperparameter values and evaluating the performance of each combination. A common approach in the field to hyperparameter tuning for LMs (Vaswani et al., 2017) used to explore a range of hyperparameter values and to select the combination that produces the best results on the validation data. To evaluate and fine-tune the different hyperparameter values, we used a plain autoregressive model, which consisted of an unmodified transformers decoder trained on 20GB of our python code dataset (we will explain the origins of this dataset later) for 25k steps. We evaluate the models on the HumanEval `pass@k` benchmark and present the results in Table 3.1.

In models A, B and C, we can observe that increasing the number of layers, and the size of the attention and feedforward layers improves model's performance. This, as expected, confirms that in general bigger models are better. Next, in the D rows, we can observe that while multi head attention is good, quality drops after certain number. This might be due to the fact that an excess of attention heads might cause some over-fitting with some specific code patterns. The last row shows the specs of our base model.

Table 3.1: Comparison of models depending on their architecture dimensions. Unlisted values are the same as those in the Initial Model

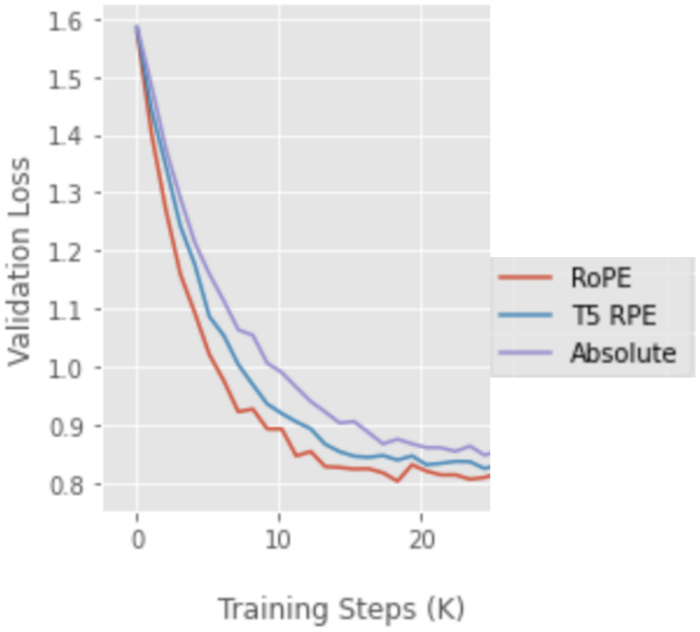
Models	**N_layers**	**D_model**	**D_ff**	**N_heads**	**pass@1**	**pass@10**
Initial Model	4	768	1024	16	1.23%	2.30%
A2	2	-	-	-	0.7%	1.46%
A1	4	-	-	-	1.23%	2.30%
A3	6	-	-	-	2.79%	5.45%
B1	-	512	-	-	1.17%	2.25%
B2	-	768	-	-	1.23%	2.30%
B3	-	1024	-	-	2.12%	4.08%
C1	-	-	512	-	1.19%	2.24%
C2	-	-	1024	-	1.23%	2.33%
C3	-	-	2048	-	1.42%	4.15%
D1	-	-	-	2	0.31%	0.44%
D2	-	-	-	4	0.78%	1.96%
D3	-	-	-	16	1.23%	2.30%
D4	-	-	-	32	1.21%	2.08%
Base Model (110M param)	12	768	1024	16	3.10%	6.20%

3.2 Positional Encoding

We have seen in the background the importance of positional encoding in transformer models. It is what allows the model to capture the structure and order of the input data, which is essential for many language processing tasks, such as code modelling in our case. Without positional encoding, the model would not have any explicit information about the position of the input tokens in the sequence, which would make it difficult to capture the relationships and dependencies between the input tokens.

There are two main types of positional encoding: absolute positional encoding and relative positional encoding. **Absolute positional encoding** encodes the absolute position of each token in the input sequence. This can be done by using sinusoidal functions or learned embeddings that are added to the input embeddings for each position in the sequence. The main idea behind this approach is to have a fixed and unique position embedding for each token that is based on its absolute position in the input. This is useful when the model needs to be aware of the exact position of a token. **Relative positional encoding**, on the other hand, encodes the relative position of each token in the input sequence. This can be done by creating a vector for each position relative to the current position in the sequence, and these embeddings are then concatenated to the input embeddings before they are passed through the transformer blocks. This approach focuses on the relations between tokens rather than the exact position, making it suitable to work with sequences of different lengths.

Figure 3.1: Positional Embedding comparison under 25k training steps. We can observe how RoPE obtains the lowest cross-entropy loss in the validation set throughout the 25k training steps.



Relative PE has serious advantages over absolute PE, especially for our modelling task:

Independence on sequence length: because relative PE encodes the position of tokens relative to each other, it can be used with input sequences of different lengths, whereas absolute PE would require a separate set of embeddings for each different sequence length.

Better generalization: By encoding the position of a token based on its relations to other tokens, it is less likely for the model to overfit to the specific sequence length, making the model more generalizable (Brown et al., 2020; Gao et al., 2020)

Greater robustness: since the relative position of a token is more invariant to changes in the input sequence, models that use relative PE are more robust to small changes in the input data, making them more suitable for tasks that involve sequences of different lengths (Tan and Bansal, 2020; Radford et al., 2021)

Consequently, we thought that Relative PE would be our best choice of positional encoding. Therefore, we've tested the two existing RPE methods — Rotary Positional Embedding (RoPE), developed by Su et al. (2021); and T5's Relative position bias, developed by2020 (2020) — that have been derived from the original relative PE presented by Shaw, Uszkoreit, and Vaswani (2018).

To make sure we had not discarded absolute PEs prematurely, we also included the learned absolute positional encodings used in GPT-3 (Codex's model). We trained our model on the same subset of our python code dataset using the different encoding techniques and we evaluated the against the same validation set. We can see the results of our experiments comparing these three positional embeddings in Figure 3.1.

3.3 Activation Function

As we've explained in the background section, activation functions are a crucial component in the feedforward layers of the transformer blocks. An activation function must be differentiable, which allows the model to be trained using gradient-based optimization algorithms and it must be non-linear, which allows the model to learn complex relationships in the data.

There are many different types of differentiable non-linear activation functions that can be used in transformer models, including the GELU function, the ReLu function, the sigmoid function, and the tanh function. The choice of activation function can affect the performance of the model, and different activation functions may be more suitable for different types of problems. Most of the activation functions have a saturating form (such as the sigmoid and the tanh functions) and thus suffer from the vanishing gradient problem: a problem that arises when the gradients (derivatives) of the weights in the network become very small, making training painstakingly slow, it occurs when the activation function of the neurons has a saturating point for large values (the partial derivative tends to zero). This problem is particularly problematic for networks with many layers, as the gradients must flow through all the layers of the network. If the gradients become very small at any point in the network, it becomes difficult for the optimization algorithm to make updates to the weights.

As we have just proved that bigger models work better for code modelling, we have designed many-layered models; and as such, we need to use activation functions with a non-saturating form. There are three such activation functions in the field: the ReLU function (Rectified Linear Units), proposed by (Nair and Hinton, 2010); the ELU function (Exponential Linear Units), proposed by (Clevert, Unterthiner, and Hochreiter, 2015); and the GELU function (Gaussian Error Linear Units), proposed by (Hendrycks and Gimpel, 2016)). The GELU functionn is defined as:

$$GELU(x) = x * \phi(x),$$

where x is the input and $\phi(x)$ is the standard Gaussian cumulative distribution function, which gives the probability that a random variable with a normal distribution is less than or equal to x.

We have compared our model after 25k training steps using the different activation functions, and we've shown the results in Table 3.2.

Table 3.2: Comparison of activation functions

Activation Function	**Validation Loss after 20k training steps**
ReLU	8.23
GELU	8.02
ELU	8.76

We can see how the GELU activation function is superior

3.4 Training Hyper-Parameters

Training hyperparameter is an important aspect of training machine learning models and is especially critical when training a language models. The choice of hyperparameters can have a significant impact on the performance of the model, and regardless of how well tuned the architecture parameters are, careful training-hyperparameter tuning is necessary to achieve the best possible results.

3.4.1 Optimisation Algorithm

One of the most important aspects of the approach in training a model is the optimisation algorithm. We will be using the Adam (Adaptive Moment Estimation) optimiser family (Kingma and Ba, 2014), as it has been established as the most efficient type of optimiser in the field (Nado et al., 2021; Norouzi and Ebrahimi, n.d.). It has seen many improvements since (Loshchilov and Hutter, 2017), and we will be using the newest version of AdamW which has been proved to provide the best results in convergence (Zhuang et al., 2022).

One big advantage of the Adam optimisers is its bias correction (Goodfellow, Bengio, and Courville, 2016) at the beginning of the training. All papers on the topic agree on the advantages of using it (Kingma and Ba, 2014; Bock and Weiß, 2019; St John, 2021), so we've trained all our models using Adam's initial bias correction.

The Adam optimiser family has several hyperparameters, β_1 and β_2, are used to control the exponential decay rates of the moving averages of the gradients; and Epsilon, that it is used to control the numerical stability of the algorithm. Epsilon shouldn't be changed from $1e-8$, beta1 should have the value of either 0 or 0.9, and $beta2$ should be between 0.9 and 1, as is well explained in the original Adam paper (Kingma and Ba, 2014). We approached the tunning of this hyperparameters using grid search as well. We evaluated our base model with different hyperparameter values under 25k training steps using Adam's cross-entropy loss function, we present the results in Figure 3.2.

3.4.2 Learning Rate and Warmup Steps

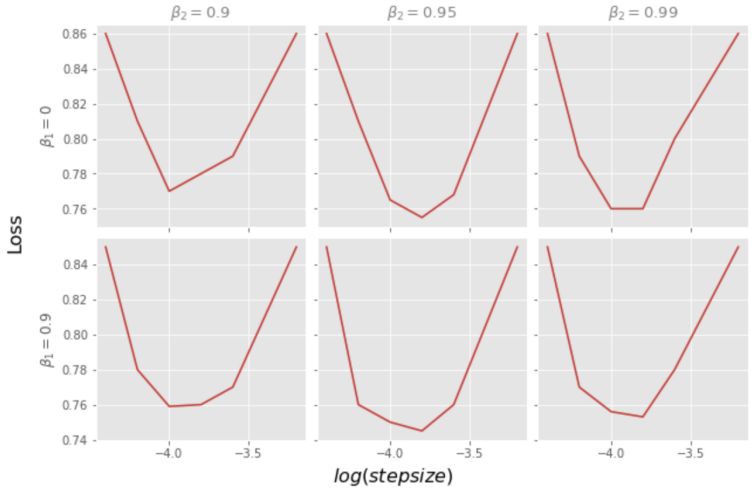
Transformers based models benefit from an initial learning rate linear increase and a posterior learning rate decay (Vaswani et al., 2017) to avoid over-fitting; and as seen before, the set value of the learning rate step-size highly impacts the model's training. That's why initial stepsize and warmup steps are important hyperparameters that can have a great impact on the performance of a machine learning model.

Research (Ma and Yarats, 2021) further supports the traditional use of Adam optimizers with warmup steps instead of newer methods that don't use warmup steps (RAdam), and the strong theoretical and empirical evidence proposed by the paper suggest that the number of warmup steps should be:

$$warmupSteps = 2 * (1 - \beta_2)^{-1}$$

for a linear stepsize increase warmup.

Figure 3.2: Fine-tuning of AdamW. Here there are several plots of loss against stepsize using different β values. We can observe that using β_1 with value of 0 yields worse results than using $\beta_1 = 0.9$ (lower row). We can also observe that for our type of models the optimal β_2 lies around 0.95 (central column)



Studies in large transformer LMs show that larger models benefit from lower initial stepsize values in preventing divergence and smaller models can sustain bigger initial learning rates. Extensive studies in the field (Kaplan et al., 2020) propose the using a rule of thumb of the format of:

$$stepSize = 0.003239 - 0.0001395 * ln(N)$$

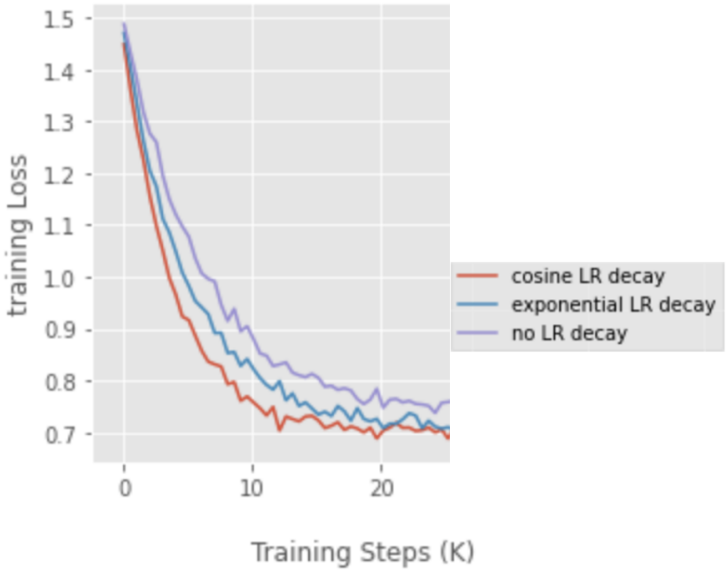
where N is the number of non-embedding parameters and where the constant values have been adjusted in accordance with the observations made in previous experiments.

As they also show that when using optimizers (such as Adam) where the learning rate is scheduled, the variance due to the initial learning rate is minimal, we haven't spent much resources fine tuning the learning rate stepsize.

3.4.3 Learning Rate Decay

Most LMs developed in the field that use optimisers from the Adam family don't use learning rate decay because Adam already computes individual adaptive learning rates for each parameter, with the initial learning rate as an upper limit. Nevertheless, many previous theoretical studies (Loshchilov and Hutter, 2017; Wilson et al., 2017) and empirical evidence in recent works (Brown et al., 2020) suggests a benefit in decreasing the upper limit of the learning rate by using an external learning rate decay a.k.a. "adaptive" step sizes, especially with big models. Exponential decay — known with many names (Hazan and Kale, 2011; Aybat et al., 2019; Ge et al., 2019; Kulunchakov and Mairal, 2019; Yuan et al., 2019; Davis et al., 2021) — and cosine decay (Liu, Simonyan, and Yang, 2018; Cubuk et al., 2019; Ginsburg et al., 2019; Lawen et al., 2019; Zhang et al., 2019a; Zhang et al., 2019b; Grill et al., 2020; You et al., 2020; Zhao, Jia, and Koltun, 2020; Chen et al., 2021) are the most used step size schedules,

Figure 3.3: Comparison between types of learning rate decay. We can observe how having some sort of LR decay clearly works better than not having any. Between exponential and cosine decay, we can see how cosine LR decay has a decent edge over exponential.



whose effectiveness has been repeatedly confirmed empirically and also theoretically (Li, Zhuang, and Orabona, 2021). Cosine decay consists of:

$$lr(t) = lr_{min} + 0.5 * (lr_{max} - lr_{min}) * (1 + cos(pi * t/T)),$$

The formula can be interpreted as follows: the learning rate starts at lr_{min} (minimum LR) lr_{max} (maximum LR) and decreases smoothly until it reaches lr_{min} (minimum LR) at time step T. The cosine function makes the learning rate oscillate between lr_{max} and lr_{min}, with a period of $2 * T$. The amplitude of the oscillation is $(lr_{max} - lr_{min})/2$, and the average value is $(lr_{max} + lr_{min})/2$.

We evaluate our base model when training with no external LR decay, with exponential LR decay and with cosine LR decay. We show the results in Figure 3.3.

3.5 Regularization Techniques

In this section we discuss the regularization techniques and parameters that — although not inherently tied to the model's architecture (unlike the GELU activation function) — are nevertheless used to prevent over-fitting in our model.

3.5.1 Small Weight Encouragement

One typical way of model regularization is to encourage the model to use smaller weights, that way the model is encouraged to generalize instead of memorizing the

Figure 3.4: Comparison of different combinations of dropout parameters

Dropout Parameters	pass@10
FD	6.73%
SD	6.59%
DD	6.84%
FD+SD	6.79%
FD+DD	6.97%
SD+DD	6.63%
FD+SD+DD	7.03%
No dropout	6.56%

We can observe how using all feature dropout (FD), structure dropout (SD) and data dropout (DD) provides the best results.

training data. This is done by penalizing the weight values when computing the loss function. The most used methods for this are L2 regularization (Moore and DeNero, 2011; Cortes, Mohri, and Rostamizadeh, 2012; Bilgic et al., 2014) and Weight Decay regularisation (Larsen et al., 1996; Gnecco, Sanguineti, et al., 2009). Recent studies have shown the superiority of weight decay (Van Laarhoven, 2017; Zhang et al., 2018; Xie, Sato, and Sugiyama, 2020a), and how for Adam optimizers using Weight Decay is clearly superior when decoupling the weight decay from the optimizer itself (Xie, Sato, and Sugiyama, 2020b; Bjorck, Weinberger, and Gomes, 2021). Furthermore, the optimizer we are using, AdamW, has been designed to specifically exploit this advantage (Loshchilov and Hutter, 2017), hence we will be using weight decay.

We need to take into account that the effectiveness of weight decay is dependent on the dimensions and format of the training data (Li et al., 2018). Therefore, the weight decay parameter in AdamW (lambda) should be normalized depending on the size and format of the training dataset. Hence, informed by some experiments (Figure 3.2) — and inspired by the weight decay normalization in the AdamW paper (Loshchilov and Hutter, 2017), we've arrived at the following weight decay normalization formula suitable or the range of training dimensions we're currently carrying out:

$$\lambda = K_{norm} * \sqrt{\frac{batchSize}{N_{tokens} * N_{batches}}}$$

Where k_{norm} is the normalization constant which roughly equals 5000 ($k_{norm} \approx 5000$).

3.5.2 Dropout

Dropout is another efficient method to avoid over-fitting (Srivastava et al., 2014; Gal and Ghahramani, 2016; Hernández-García and König, 2018; Wang et al., 2019; Wei, Kakade, and Ma, 2020) and it still benefits adaptive gradient optimization methods such as Adam optimisers (Vaswani et al., 2017). In particular, there is ample empirical evidence to the benefit of pairing AdamW with dropout (Ginsburg et al., 2019; Guo, Liu, and Wang, 2021; Stoffl, Vidal, and Mathis, 2021; Setiawan, Yudistira, and Wihandika, 2022); hence, we will use the dropout method. Dropout methods used when training Transformer LMs can be classified in three groups.

Feature Dropout (Srivastava et al., 2014) which consists of randomly setting neuron weights to 0 during training, transformers has two specific dropouts tailored to its structure: attention dropout (attn_pdrop in the `huggingface` library), applied to the attention weights; and activation dropout, applied to the activation layer of the FFN. **Structure Dropout**, which consists in dropping (setting to default) entire structure parts of the transformer architecture. There are three types of Structure Dropout: DropHead, presented by (Zhou et al., 2020) and HeadMask, presented by (Sun et al., 2020); both proposed to avoid the problem of a small portion of heads dominating the multi-headed attention mechanism (Michel, Levy, and Neubig, 2019; Voita et al., 2019); and then we have LayerDrop (resid_pdrop in the `huggingface` library) — proposed by Fan, Grave, and Joulin (2019), which is a higher-level and more delicate structure dropout which drops some training layers during training decreasing the model size. Finally we have **Data Dropout** (embd_pdrop in the huggingface library), which randomly removes some of the training tokens.

Alerted by the work of Wu et al. (2021) and after several experiments where we evaluated our base model under 50k training steps (Figure 3.4), we acknowledge the importance of implementing all types of dropouts (feature dropout, structure dropout and data dropout) when training our models, following a strategy of using attention dropout, LayerDrop and data dropout. Due to the size of our training data, we use $attn\_pdrop = 0.1$, $resid\_pdrop = 0.1$ and $embd\_pdrop = 0.1$ in the training data. We can observe on Figure 3.4 the advantages of using all three dropout strategies.

3.6 Encoding and Decoding Methods

Tokeniser use for NLP has been considered for a long time (Webster and Kit, 1992), and thus they also have a crucial role in code modeling . As we have already seen in the background section, a tokeniser breaks down the input text into individual tokens (words or sub-words) that can be fed into the model. These tokens are then converted into numerical values, called token IDs, which are used as input for the model. The tokeniser is also responsible for creating a vocabulary of all the unique tokens in the training data, which is used to map the text to token IDs.

On the other hand, the decoder is the one responsible for generating the output text. First the model takes the encoded representation of the input text, along with the hidden state of the encoder, as input and generates a sequence of token IDs. These token IDs are then passed through a final linear layer to produce a probability distribution over the vocabulary for each token. The decoder then uses this probability distribution to select the most likely token to be generated next, and this process is repeated until the desired length of the output text is reached.

3.6.1 Encoding of Training Data (tokenisation)

There are multiple ways one can go about tokenising input data. In this section we will discuss the possible methods there are.

3.6.1.1 Types of Tokenisers

There are several types of tokenisers, each with their own strengths and weaknesses:

- **Word tokenisers:** These tokenisers use whitespace and punctuation to segment text into individual words. They are the simplest and most used tokenisers.
- **Sub-word tokenisers:** These tokenisers split words into smaller sub-word units, such as character n-grams or byte pair encoding. Sub-word tokenisers are useful for handling out-of-vocabulary words and for handling inflectional languages like German, Japanese, or Arabic.
- **Character tokenisers:** A character tokeniser is a type of tokeniser that segments text into individual characters, instead of words or sub-words. This means that each character in the text becomes a separate token, regardless of its nature.

We have selected sub-word tokenisers as the best option for code modelling for the following reasons:

- **Handling OOV words:** Sub-word tokenisation methods handle out-of-vocabulary words more effectively than word tokenisation, by breaking these words down into sub-word units that are present in the model's vocabulary. This allows the model to generalize better and to handle words that were not present in the training data.
- **Context:** Sub-word tokenisation captures character-level information by segmenting words into sub-word units, allowing the model to learn about character-level information, as well as word-level context.
- **Vocabulary size:** Sub-word tokenisation leads to a reduction in the vocabulary size compared to character tokenisation, which helps in keeping the computational complexity and memory requirements of the model at manageable levels, something very important when building large scale models such as we are.

3.6.1.2 Sub-word tokenisers

There are several types of sub-word tokenisers that are currently used in the field:

- Byte-Pair Encoding (BPE) tokeniser (Sennrich, Haddow, and Birch, 2015) is a data-driven approach that learns to segment words into sub-word units by picking the most frequent patterns of word segments, that way the same information can be represented with a reduced vocabulary size. A minimal python implementation is shown in Figure 3.5.
- Unigram Language Model (ULM) tokeniser, presented by (Kudo, 2018) , is a sub-word tokenisation algorithm that segments words by training a language model on the input text and then iteratively merges the most probable character sequences based on the model's likelihood of the sub-words rather than picking the most frequent.
- WordPiece tokeniser (Wu et al., 2016; Devlin et al., 2018; Song et al., 2020) is a type of sub-word tokenisation, it is similar to BPE but it uses a different algorithm to merge sub-words. The algorithm is based on the maximum likelihood principle and is designed to be more computationally efficient than BPE.
- SentencePiece tokeniser, presented by Kudo and Richardson (2018) is a text tokeniser mainly used for Neural Machine Translation (NMT). It is praised

Figure 3.5: Minimal BPE tokeniser implementation

```
import re, collections
def get_stats(vocab):
  pairs = collections.defaultdict(int)
  for word, freq in vocab.items():
    symbols = word.split()
    for i in range(len(symbols)-1):
      pairs[symbols[i],symbols[i+1]] += freq
  return pairs

def merge_vocab(pair, v_in):
  v_out = {}
  bigram = re.escape(' '.join(pair))
  p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
  for word in v_in:
    w_out = p.sub(''.join(pair), word)
    v_out[w_out] = v_in[word]
  return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10

for i in range(num_merges):
  pairs = get_stats(vocab)
  best = max(pairs, key=pairs.get)
  vocab = merge_vocab(best, vocab)
  print(best)
```

for its ability to handle rare words, out-of-vocab words, and languages with morphological inflections and complex character sets.

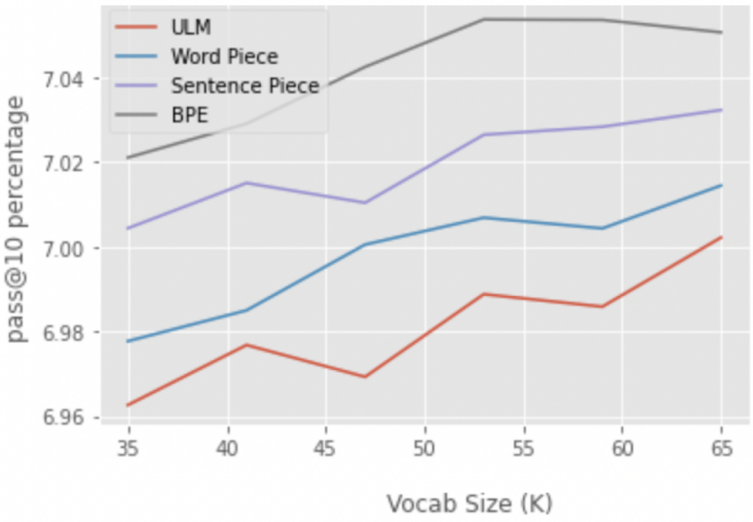
To evaluate the best tokeniser for our model, we have trained the four discussed tokenisers on our 180GB python dataset varying the vocabulary sizes. We then test our base model using the different tokenisers, we present the results in Figure 3.6.

3.6.2 Decoding Methods

There are two main types of decoding methods, deterministic and stochastic methods:

- Deterministic decoding methods are methods that always generate the same output for a given input. These methods select the most likely token at each step, based on the probability distribution generated by the model. A greedy decoding algorithm would be an example of a deterministic decoding method. Deterministic decoding is fast and easy to implement, but it can lead to repetitive or nonsensical outputs, especially when the model is not confident about the next token.
- Stochastic decoding methods, on the other hand, involve randomly sampling tokens from the probability distribution generated by the model. These methods allow the decoder to explore multiple options and generate more diverse and creative outputs, although they can also result in more errors or nonsensical text.

Figure 3.6: Comparison between sub-word tokenisers. We can observe how the BPE tokeniser outperforms the rest of tokenisers. We can also observe how the best vocabulary size for BPE stabilizes somewhere between 50 an 55 thousand tokens.



Note that the choice of the decoding method depends on the use case, the resources available, and the desired trade-off between speed, quality, and diversity of the generated text. In the case of text generation, several studies in the field have provided empirical evidence that stochastic decoding methods are superior for a task such as ours (Holtzman et al., 2019; Ippolito et al., 2019; Wiher, Meister, and Cotterell, 2022). These papers show that stochastic decoding methods have several advantages over deterministic decoding methods, such as the ability to generate more diverse and creative text, handle uncertainty and out-of-vocab words, and handle rare events; properties that are sought out in a model that can model code for a variety of problems.

3.6.2.1 Stochastic Decoders

The most basic stochastic decoder, sampling decoding, involves randomly sampling tokens from the probability distribution generated by the model, rather than selecting the most likely token. This method generates more diverse and creative text, but it can also result in more errors or nonsensical text due to its high uncertainty (Zhang et al., 2020). Therefore, more complex stochastic decoding methods have been developed in order to implement the diversity of generation that stochastic sampling provides but without risking text incoherence:

Top-k sampling decoding (Fan, Lewis, and Dauphin, 2018; Holtzman et al., 2018) is similar to sampling decoding, but it only samples from the top k most likely tokens. This method is a good trade-off between diversity and quality of the generated output.

Top-p (nucleus) sampling , presented by Holtzman et al. (2018), also samples from the model's probability distribution, but it only samples from a subset of tokens with a

cumulative probability of at least p, called the "nucleus." This method allows to generate coherent outputs while still allowing for some diversity.

Contrastive Search Decoding (Su and Collier, 2022) is the most recently developed:

$$x_t = \text{argmax}_{v \in V(k)} \left\{ (1-\alpha) \cdot \underbrace{p_\theta(v|x<t)}_{model-confidence} - \alpha \cdot \underbrace{\max_{1 \le j \le t-1} s(h_v, h_{x_j})}_{degeneration-penalty} \right\}$$

where $V(k)$, is the set of top-k predictions from the language model's probability distribution $p_\theta(\cdot|x<t)$; model confidence, is the probability of candidate v predicted by the LMs; $s(\cdot,\cdot)$ computes the cosine similarity between token representations; and degeneration penalty, measures the maximum cosine similarity between the representation of the candidate v and that of all tokens in $x<t$, where the candidate representation h_v is computed by the LMs given the concatenation of $x<t$ and v. Intuitively, a larger degeneration penalty of v means it is more similar to the context. Intuitively, contrastive search generates k candidates and then ranks them according to the previous formula, where the α parameter determines the weight of the contrastive objective function penalty in the final score assigned to each candidate. A high α value will give more importance to how different a candidate is from the other candidates. This generates a more diverse and creative output, but it can also lead to more errors or nonsensical answers. A low α value, on the other hand, will give more importance to the likelihood of the candidate assigned by the model. This can result in more coherent and semantically meaningful text, but it can also lead to less diversity.

3.6.3 Temperature

Stochastic Decoding methods sample by using a probability density function that represents the likeliness of the different token predictions proposed by the model. The probability distribution is calculated via

Table 3.3: Best decoding tunings. We can observe how contrastive search provides the best results for our code modelling tasks

	pass@1		**pass@10**		**pass@100**	
	Best parameters	Score	Best parameters	Score	Best parameters	Score
Top-k	T=0.3 K=25	3.62%	T=0.6 K=35	8.03%	T=0.7 K=50	12.77%
Top-p	T=0.2 P=0.96	3.85%	T=0.7 P=0.95	8.42%	T=0.8 P=0.93	13.06%
Contrastive Search	T=0.2 α=0.1	4.32%	T=0.6 α=0.2	9.56%	T=0.8 α=0.4	15.24%

$$\sigma(z_i) = \frac{e^{z_i/\theta}}{\sum_{j=0}^{N} e^{z_j/\theta}},$$

take notice that it is a modified `softmax` function, where the previous probability z_i assigned to the ith token is dampened by the temperature value $theta$.

The temperature value affects the model's decoding by controlling the sharpness or flatness of the probability distribution generated by the model, thus it is used to control the randomness of the sampling process. A high temperature value will assign higher probabilities to a wider range of tokens. This, will result in a trade of output coherence for diversity. A low temperature value will assign higher probabilities to a smaller range of tokens. This often results in more coherent and semantically meaningful text, but it can also lead to less diversity.

The optimal temperature value will depend on the specific task and data. For example, as `pass@k` can be interpreted as determining the top-performing sample out of k options, higher temperatures produce better results when working with larger values of k as it produces a more varied collection of samples and the evaluation criteria only considers if the model can produce a correct solution, regardless of how wrong some other solutions are. Testing different temperature values we have determined the best decoder — with its best hyperparameter tuning — and the best temperature for each `pass@k` task. We present the results in Table 3.3.

3.7 Hyperparameters and Model Choices Conclusions

In this research, we aimed to investigate the impact of different hyperparameter configurations on the performance of our code modelling. After all the previous experiments exploring all the different hyperparameter options, we have arrived to the following conclusions about what the best hyperparameters are and why:

- With respect to the **structure dimensions** we have unsurprisingly gathered that, while there is enough variety of training data, the bigger the model is the better it will do in terms of performance. The only structure aspect where we have found that moderation in size is preferable, is in the number of attention heads, were we have found that with the specifics of the modelling we are conducting, over 16 heads is excessive. We attribute this to overfitting caused by the naturally repetitive structures present in python code
- With respect to the **positional encoding** we have seen that relative positional encoding works best for code modelling. This was expected as positional encoding has shown it's superiority most modelling tasks of high complexity. We have experimentally deemed Rotary Positional Embedding (`RoPE`) the best encoder for code modelling.
- Due to the big size of the model, it was pretty clear that a non-saturating **activation function** was necessary. Between all the activation functions with a non-saturating form we found that `GELU` is the best performing one for code modelling purposes.

- For the **optimiser algorithm**, we decided on using optimisers of the Adam family as they have been proven best in this field. We found out that the best optimiser is `AdamW` with $\beta_1 = 0.9$ and $\beta_2 = 0.95$.
- For the **warmup steps** and the **learning rate stepsize**, we have used the experimentally proven formulas of $n = 2 * (1 - \beta_2)^{(} - 1)$ and $stepsize = 0.003239 - 0.0001395 * ln(N)$ respectively. And we have found that a `cosine learning rate decay` is the best one for our type of models.
- With regard to the small weight encouragement regularisation technique, we have found that **Weight Decay** is superior when using the AdamW optimiser. Moreover we have developed the rough formula of $\lambda = K_{norm} * \sqrt{\frac{batchSize}{N_{tokens} * N_{batches}}}$ to determine the weight decay constant.
- With regard to the **dropout** regularisation technique, we have found out that it's beneficial to use all three feature dropouts, structure dropout and data dropout. We have opted to use small values in all dropouts (0.1) due to the big size of our training data.
- We have decided on using **sub-word tokenisers** as the encoding method due to all its advantages already explained. We have found that the Byte-Pair Encoding (`BPE`) tokeniser works best for us
- We have decided on using **stochastic decoders** as the decoding method due its several advantages. We have discovered that the new `contrastive search` decoding technique produces the best results, and that its best α value is 0.6. Moreover, for higher k values in `pass@k` we have found that a higher temperature is beneficial. We have found as optimum the temperature values of 0.2, 0.6 and 0.8 for the respective k values of 1, 10 and 100.

Chapter 4

Training Methods

In this chapter we investigate the training data and the training process to be used for our model building strategy.

4.1 Training Data

Data are crucial when training a language model. The quality and quantity of the data used to train a model will have a significant impact on its performance. A model that is trained on a large, diverse dataset will be able to generalize to new examples and to perform well on a wide range of tasks. In contrast, a model that is trained on a small, homogeneous dataset may probably struggle to generalize to new examples and may perform poorly on tasks that are not well-represented in the training data.

Therefore, we have strived for training data that includes a diverse set of code examples from different tasks and domains.

4.1.1 Data Collection

We collected three datasets:

- We wanted to experiment if being trained on natural language would be beneficial to further understand the code's comments. Hence, we also used the publicly available The Pile dataset. The Pile dataset is composed of 825.18 GB of English text used for language modeling purposes. Created by Gao et al. (2020), it is made up of 22 different high-quality subsets, including programming language data obtained from popular GitHub repositories with over 100 stars, which accounts for 7.6 % of the entire dataset (which roughly accounts for 95 GB of code).
- Our python dataset was obtained from Google's BigQuery GitHub dump (biggest open-source code database) Python files. The dataset was 180 GB in size and contained around 20 million files. Which is comparable in size to Codex's training data.
- We also wanted to experiment with training the model using different coding languages, to see if performance on the python language could be further increased

by this practice. To obtain this dataset we used Google's BigQuery GitHub selecting several popular programming languages (not including python). The size of this dataset was 241 GB in size, containing around 18 million files.

A problem with the initial datasets was their enormous size and the low quality of their data. In order to fix this we conducted some data processing.

4.1.2 Data Processing

We didn't do any data processing on The Pile dataset, as this dataset has already been cleaned by (Gao et al., 2020). But, to decrease the size of the datasets we collected and increase their content quality, we conducted some methods to remove excess files that weren't beneficial in the model's training.

4.1.2.1 Code Filtering

We filtered out files >1 MB to further decrease the size of the training dataset), which they accounted to less than 1 % of the total files but for a very big amount of the dataset size (almost 40 GB). And, inspired by the Codex paper (Chen et al., 2021), we filtered out files with a max line length superior to 1000 characters, files with a mean line length superior to 100 characters, files with a fraction of alphanumeric characters less than 0.25 % or more than 90 % of the characters being decimal or hexadecimal digits are removed and files containing the word 'autogenerated' or similar in the first 10 lines. All these measures reduced the size of both combined datasets by roughly 30 %.

4.1.2.2 Code Deduplication

(Allamanis, 2019) have done a wonderful research on how training data affects model's performance, and it provides extensive evidence showing how code duplication present in most code datasets is detrimental for the model's training. On that account, we decided on deduplicating the code files present on the datasets by comparing their SHA-256 hash values and eliminating exact matches. Furthermore, as we noticed that there are many non exact-duplicates files that are essentially the same, we decided on

Table 4.1: Nature and sizes of datasets. We can observe how our processing has reduced the size considerably while improving the dataset quality

Dataset	Languages	Size before processing (GB)	Size after processing (GB)
The Pile	Natural Language	730	-
	Code	95	-
Varied Code	C	270	126
	C++	130	33
	Java	230	72
	JavaScript	260	84
	PHP	210	66
Python	Python	241	83

conducting near deduplication. We used the `MinHash` algorithm to compute the Jaccard similarity, $J(A,B) = \frac{|A \cap B|}{|A|+|B|-|A \cap B|}$, where A and B represent the sets of tokens of the two code files being considered for near-similarity. We've considered the Jaccard thershold of 0.9 for near-deduplication. Unsurprisingly, the amount of exact duplicates and near duplicates is high due to the high number of forks and copies present in github. We found that, on average, for each programming language, 0.1 % of the unique files make up 10 % of all files, 1 % of the unique files make up 30 % of all files, 10 % of the unique files make up 60 % of all files.

We roughly reduced the file number and size of both datasets by another 30 % just by deduplication. Showing that deduplication is a very effective method to decrease the size and augment the quality of a code dataset. We can visualise the size and nature of the datasets in Table 4.1

4.2 Training Process

The examples the model uses to train can have different lengths, each batch that the model uses to update its weights can contain a different number of examples, and the more training steps that our model undertakes the more batches our model will process when training. All these different possibilities of training parameters can highly affect our end model after training, we will discuss them in this section.

4.2.1 Sequence Length

The sequence length of the training examples is an important hyperparameter that affects the training of a model. In language models, the sequence length determines the number of tokens that the model considers at a time when making predictions. If the sequence length is too short, the model may miss important context and fail to capture the relationships between tokens. On the other hand, if the sequence length is too long, the model may have difficulty processing the input efficiently and may suffer from overfitting to the training dataset. As we have a very big and varied corpus of training data, we have found out that over-fitting is not a problem. This being the case, we set the sequence length to the context length (the maximum amount of tokens that our model architecture can process at one time).

4.2.2 Batch Size

As we are training using AdamW, which is a type of minibatch stochastic gradient descent optimiser, the model's parameters are updated after each batch using the average gradient computed over that batch. Therefore, batch size in stochastic gradient descent can affect bias and overfitting.

If we choose a small batch size, we might be unfortunate enough that all the examples present in that batch have the same type of particularity (for example, they might all be pieces of code with variables defined with capital letters). Hence, such a batch may lead to a biased model, as the model would try to fit the particularity in the selected samples

(in this example, it would assign to much importance to capital letters when defining variables). This being so, using a larger batch size in SGD will reduce the probability of the batch having a particularity, hence it will avoid biasing the model. In this case, the model will be trained on more examples in each update and will capture more general patterns in the data. Therefore, a large batch size is preferable when using SGD.

To make training faster and better make use of our parallel computing capabilities, we scaled up the batch size as our models grew; and to make sure the samples in the batches were varied enough, we used bigger batches with bigger datasets. Therefore, our batch sizes range from 250k samples (when training the smallest model in the only Python dataset), to 2M samples per batch.

4.2.3 Training set steps

The number of training steps has a significant impact on the performance of a model. It determines the number of batches of data the model will learn from in training. The number of training steps influences the model's to converge to an optimal solution. If the number of training steps is too small, the model may not have seen enough examples to learn the patterns in the data, and its performance may be poor. On the other hand, if the number of training steps is too large, the model may start to over-fit the training data, memorizing the training examples instead of learning the underlying patterns.

We did not detect any over-fitting in our large and varied corpus of training data in any of the models except with the biggest models (>2B), which occurred around the 110kth step when training under the Python dataset (we never reached overfitting with The Pile and the varied code datasets). Ideally, we would keep training and monitoring the model's performance on a validation set, training multiple epochs if necessary, and only stop training when the eval loss plateaued or started to degrade. But due to our limited computing capacity, we couldn't just do that, especially when using bigger batches. Hence, we settled ourselves at training for 150k steps when using The Pile, 100k steps when using the varied code dataset and 50k steps when using the Python dataset.

4.2.4 Training procedures

As we have already mentioned before, we used Google's Colab cloud services to train our models. Colab will allocate resources depending on the demand, but under a paid subscription it can provide over 50GB of rum and up to almost 5000 GPU Cuda cores. As the models we have trained are very big, we had to find an efficient way to use our available computing resources to tokenise our data and to update the model's weights. Therefore, we used PyTorch's `DDP` library (Kim et al., 2019; Stooke and Abbeel, 2019; Li et al., 2020) and HuggingFace's `Accelerate` to implement parallel computing.

Some samples in the dataset are very short compared to the sequence limit selected, so tokenizing and passing those exmples individually would be a waste of data and computing. Therefore, we concatenated the samples — using an EOS token — when possible in order to approximate all individul samples to the maximum passable sequence length. In order to do this efficiently, the samples in the buffer were tokenised in parallel and then concatenated. Once the the batch of chunked samples is passed on to train the model and the buffer is cleared, the process starts all over again.

The training iterations were also conducted in parallel. For each batch this was achieved in the following fashion: The batch of chunked and tokenised samples is divided into mini-batches, each GPU worker receives a mini batch and computes the gradients, finally the model is updated using the average of all the gradients calculated in all the different GPU workers. In order to carry out this such parallel computing of the gradient during training, each GPU worker must have in memory a copy of the model. Therefore, we used gradient checkpointing (Chen et al., 2016; Hung et al., 2019; Rojas et al., 2020) as a measure to decrease the GPU memory footprint.

4.2.5 Training Strategy

As we have explained in the training data section, we collected several databases which we think are most appropriate to train a model for code modelling: a general natural language database (with some code as well), another database that contains several coding languages, and a more specific database containing filtered python code.

After training our models using a different combination of the datasets, we found out that all three of them where beneficial. Doing some experimentation by changing the order of data training, we found that — concurring all the studies in transfer learning (Mesnil et al., 2012; Patricia and Caputo, 2014; Wolf et al., 2019; Zhuang et al., 2020) — the best and fastest way to train the model was to start with the more general dataset and progress to the most specific one. This makes sense from a conceptual sense: By starting with a more general database, when further training the model, these general learned features are leveraged and only a few layers of the model end up being updated with the new data, resulting in a faster and more effective training process. On the other hand, starting training with a very specific dataset will render training with the general one pointless, as the model would have learned features that are too specific and not general enough to be further tuned by the new data. Additionally, starting with a more general dataset will help to avoid the problem of limited data, which is common when working with specific datasets (such as the python dataset), by allowing the model to learn more generalizable features that can be useful even when working with more limited data. This is one of the reasons why we think training on general coding languages is so beneficial for python language modelling.

Therefore, we developed a new method of training. We first train models on The Pile, then we fine-tune the model on the Varied Code dataset. Finally we fine-tune the model in the Python dataset

Chapter 5

Results and Analysis

5.1 Results

We trained three differently-sized models, we initialised their parameters according to the findings on the fine-tuning experimentation discussed in Chapter 4. We have shown the architecture parameters in Table 5.1.

Following the training strategy we devised, we trained for each model size three different models, one trained in NL (The Pile dataset), another one trained on code (Varied Code dataset) on top of NL, and a final one further trained on python code (specific Python dataset). We have shown the three evolutions on the model in Table 5.2.

5.1.1 Model Performance

We used the HumanEval dataset gold-standard to compare the performance of our models to the state-of-art models on the field. Following (Chen et al., 2021), we created a prompt for each problem in the HumanEval dataset by combining the function header and the docstring comment. We then generate tokens from the model and continue doing so until we reach one of the following tokens: `'/nclass'`, `'/ndef'`, `'/nif'`, or `'/nprint'`, which are tokens that indicate the end of the predicted function. Then we allowed our models to generate 200 responses for each HumanEval prompt, from which then the probability of a response being correct out of a k sample from the 200 generated responses was calculated using `pass@k` as explained in Section 2.3. We show in Figure 5.1 an example of a HumanEval prompt, and we show in Table 5.3 our models' `pass@k` scores.

The improvement of performance against model size apparently follows a logarithmic growth in both OpenAI's Codex models (Chen et al., 2021) and in our models. Although we have not managed to train a large enough number of models to do significant statistical testing in confirming this behaviour, most large LMs show logarithmic diminishing returns in performance when scaled up. Therefore, we have opted to analise the growth behavior with the model's we have:

Table 5.1: Architecture parameters of the three final models

Model Size (trainable parameters)	**Number of Layers**	**Model Dimension**	**FeedForward Dimension**	**Number of Heads**	**Head Dimension**
300M	24	1024	16384	16	256
2.5B	32	2560	16384	32	256
6B	32	4096	16384	32	256

Table 5.2: Training hyperparameters at each training phase. In this chart we can see the different training processes we have used to gradually specialise our models to the code modelling task

Model	**Dataset**	**Parameters**	**300M**	**2.5B**	**6B**
NL	The Pile	Learning Rate Steps Batch Size	3.0e-4 150k 1M	1.6e-4 150k 2M	1.2e-4 150k 2M
NL and Varied Code	Varied Code	Learning Rate Steps Batch Size	1.8e-4 100k 500k	0.8e-4 100k 1M	0.4e-4 100k 1.5M
NL and Varied Code and Python	Python	Learning Rate Steps Batch Size	1.8e-4 50k 250k	0.8e-4 50k 500k	0.4e-4 50k 750k

Table 5.3: Training hyperparameters at each training phase. In this chart we can see the `pass@k` results on the HumanEval dataset. We have used three different sampling temperatures t — 0.2, 0.6 and 0.8 — for the different k tests — 1, 10, and 100 respectively. We have selected in bold our best performing model.

Model Size	**Training Type**	**pass@1**	**pass@10**	**pass@100**
300M	NL	1.92	4.03	7.08
300M	NL and Varied Code	6.61	10.60	6.79
300M	NL and Varied Code and Python	12.72	21.08	35.17
2.5B	NL	6.40	14.11	22.64
2.5B	NL and Varied Code	14.47	24.63	38.54
2.5B	NL and Varied Code and Python	21.85	36.62	56.99
6B	NL	10.45	18.39	29.91
6B	NL and Varied Code	8.17	28.74	44.89
6B	NL and Varied Code and Python	**26.16**	**42.33**	**66.23**

Figure 5.1: HumanEval problem example. Where on top we have the prompt, and highlighted in blue we have our model's generated answer.

```
def count_words(filename):
    """Count the number of occurrences of each word in the file"""
    word_counts = {}
    with open(filename) as f:
        for line in f:
            line = line.strip()
            words = line.split()
            for word in words:
                if word not in word_counts:
                    word_counts[word] = 1
                else:
                    word_counts[word] += 1
    return word_counts
```

On the `pass@1` our model's achieve a performance growth of $3.94 * \log modelSize + 1.24$ vs OpenAI's $3.19 * \log modelSize + 1.28$

On the `pass@10` our model's achieve a performance growth of $5.68 * \log modelSize + 1.26$ vs OpenAI's $5.49 * \log modelSize + 1.27$

On the `pass@1` our model's achieve a performance growth of $12.64 * \log modelSize + 1.21$ vs OpenAI's $10.42 * \log modelSize + 1.24$

We have plotted in Figure 5.2 a visualization of the performance of our models and OpenAI's. We can observe how OpenAI's family of models perform slightly better for lower sizes in the `pass@1` and `pass@100` categories. And that the best performance in each `pass@k` category is achieved by OpenAI's 12B model — , which achieves `pass@1`, `pass@10` and `pass@100` scores of 28.81, 46.81, and 72.31 respectively. Nevertheless, our family of models show a better logarithmic growth when increasing the size.

5.1.2 Code Commenting

We also tested our models' capacity of to generate docstring comments, we believe that comment generation is an important evaluation, as the comments can provide information and context about how the model interprets and understands code. We graded the comments by hand, only considering the docstring correct if it specifically describes in a correct and meaningful way the code function.

Inducing our model to generated comments from code was easier than apparent, although code usually follows from docstring comments. To prompt our model to generate an explanation for a given code we just added at the end of the function the line: `"""Code explanation: the previous function --` This urged our models to complete the comment with an explanation of the function. We show in Figure 5.3 an example of a comment generating prompt.

Due to the long task of manually evaluating the explanations, we only generated five samples per problem in the HumanEval dataset, grading a total of 820 problems. We report our calculated performance of the different models in Table 5.4.

Figure 5.2: Model Performance Comparison. This graph shows the models' performance growth against their log scaled size.

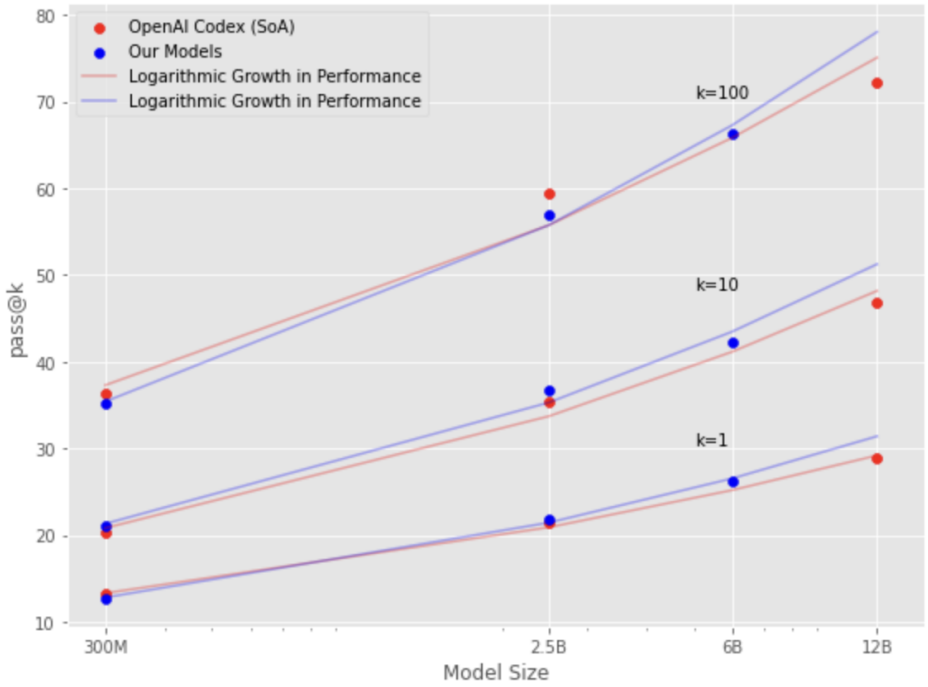


Table 5.4: Manually Evaluated Code Commenting Scores. We can observe how the performance of code commenting clearly improves with our training system and when scaling the model.

Model Size	**Training Type**	**pass@1**	**pass@5**
300M	NL	0	0
300M	NL and Varied Code	3.2	6.5
300M	NL and Varied Code and Python	8.5	16.3
6B	NL	1.2	5.3
6B	NL and Varied Code	16.5	31.2
6B	NL and Varied Code and Python	**34.2**	**57.3**

Figure 5.3: Docstring generation example. Where on top we have the prompt, and highlighted in blue we have our model's generated answer.

```
def num_in_str(s):
    return any(i.isdigit() for i in s)

"""Code explanation: the previous function
takes in a string and returns a boolean
indicating whether that string contains any
numbers. We are using the any() function to test if
any number in the string is a digit.
"""
```

5.2 Analysis

Based on the results, unfortunately we have not surpassed (Chen et al., 2021)'s current state-of-art performance with their biggest Codex model. But we have managed to achieve an almost comparable performance with a model that is 50 % smaller — 6B vs 12B non embedded parameters — with very little resources and using data and model architectures that can be freely implemented by anyone with the required knowledge. On top of that, based on its apparent superior logarithmic growth, our model building method obtains better results for higher sized models: from around 2.5 B parameters onward, our models are predicted to achieve better performance than OpenAI's counterparts of an equivalent size. As a matter of fact, if we had had the computational resources to train an equivalent sized model to Codex 12B, it is predicted that our model would achieve a slightly better performance — 31.7 52.31 and 78.04 compared to 28.81, 46.81, and 72.31.

Regarding the code commenting results, we have found that our models have a surprisingly great capacity for explaining code, even though docstrings in our dataset are bound to be of lower quality compared to the code as programmers usually do not put that much effort in commenting their code. Although, taking into account that natural language is more forgiving than code — the syntax is more flexible — this great performance in writing NL comments makes more sense. The great correlation we have found between a model's capability to generate code and its capability to generate code explanations has allowed us to infer that fine-tuning a model for code modelling improves the model's performance in code generation and completing tasks **and** in code explanation and commenting tasks. But bear in mind that this model building method is not optimized for the latter tasks, there's almost surely a better way to build models designed for the specific task of generating code comments and explanations.

Chapter 6

Conclusions

6.1 Limitations and Biases

Our models and research have been conducted to improve the code completion on specific prompts — the Human Eval dataset — that although fairly complete and significant of the various uses of python, it is still a limited sample. Therefore, the reassuring performance we have achieved in this specific task might not be reflected in other code tasks such as code correcting or code optimization. Although from our observed improvement in code commenting when improving code completion, we can guess that when fine-tuning a model for any code modelling task, the improvement will likely transfer to the rest of code tasks. Therefore, although this approach of improving code modelling is specific for code completion tasks, we might infer that a similar model construction technique will render good results for other code modelling tasks.

To continue, the fine-tuning of our model's architecture and hyperparameters has been limited. As we did not have great computational resources, we could not explore every possible combination of architecture and parameter choices. So, although we have computed a thorough and exhausting grid search through all the possible parameters, a better fine-tuning could be achieved by conducting a proper optimizing technique.

One last limitation is the dataset collection and training strategy we have implemented. Although we have collected the best data we found available to the public and cleaned it as well as we could, a better training datset could be very well obtained by employing more thorough data scrapping and tuning a better data processing strategy — we think our code deduplication was well achieved. Apart from that, although we are pretty confident that our training strategy is optimal, we did not have the resources to carry out the training — on each dataset — until its optimal point. Therefore, under more training steps our model strategy would most probably see an improvement.

Now, the main bias in this project is the model-comparison we conduct in the results section. Although our model building strategy is apparently superior to OpenAI's SoA, please bear in mind that we are comparing our models with their last published information, which was in 2021. Since 2021 many advances in the transformers ML field have happened. Therefore, plenty of our model strategy's improvement compared to Ope-

nAI's is due to our application of these advancements in encoding, activation functions, optimizers, decoding strategies. Therefore, OpenAI might very well have implemented these techniques in the models they are currently using and achieved an even better model developing strategy. But, as they have not released any publications since 2021 we cannot compare our models to their current models, if they have developed any that is.

6.2 Closing words

On a personal note, this project has been a great insight into the field machine learning and modeling tasks. We have learnt how,from the general concepts of neural network and attention mechanism, we can build a full-fledged language model. Where, introducing some clever modifications to the attention mechanism and stacking them, we have created autoregressive transformer models which obtain state-of-art results in modelling sequential data. Then, we have applied this sequential models to code modelling. By modifying the number of transformer blocks we stack, or the amount of learnable parameters in the transformation matrices, we have explored the versatility that transformer models can show depending on their architecture. Furthermore, we have learnt the mechanics with which neural networks "learn," and how this tuning of parameters can be made more efficient depending on the positional encoding, optimiser, and activation function one uses during backpropagation.

We have learnt about the fine line between underfitting and overfitting, and the regularization techniques one can use to avoid them. And we have ascertained the techniques to avoid training convergence to local minima. We have explored different ways to construct model training datasets, and ways to reduce their size with minimal information power loss. Moreover, we have employed transfer learning during training: how training a model on general data first and then fine-tuning it for a task, the model is able to better generalize for the specific task than if it had only be trained with data from said task. On top of that have learnt about efficient use of computing resources, in particular parallel computing, which has allowed us to reduce training time, and thus costs, considerably.

On a more general note, we have contributed to the best of our possibilities in the code modelling field. We have shown how pairing proper architecture choices and hyperparameter fine-tuning, a better performance to model-size ratio can be achieved. How, by collecting varied data of quality, devising a concretizing training strategy and using transfer learning, one can achieve a more efficient training in programming languages. Furthermore, we have explored the different encoding and decoding options, finding out that which tokenisation and decoding techniques work best for coding tasks. Implementing this techniques we have devised an apparently superior model building method, that requires less computational resources and data to develop state-of-art code modelling transformers. We hope this project motivates more people to investigate model optimization for specific tasks, and thus promote achievable model development.

References

Allamanis, Miltiadis, 2019. The adverse effects of code duplication in machine learning models of code. *Proceedings of the 2019 acm sigplan international symposium on new ideas, new paradigms, and reflections on programming and software*, pp.143–153.

Aybat, Necdet Serhat, Fallah, Alireza, Gurbuzbalaban, Mert, and Ozdaglar, Asuman, 2019. A universally optimal multistage accelerated stochastic gradient method. *Advances in neural information processing systems*, 32.

Bilgic, Berkin, Chatnuntawech, Itthi, Fan, Audrey P, Setsompop, Kawin, Cauley, Stephen F, Wald, Lawrence L, and Adalsteinsson, Elfar, 2014. Fast image reconstruction with l2-regularization. *Journal of magnetic resonance imaging*, 40(1), pp.181–191.

Bjorck, Johan, Weinberger, Kilian Q, and Gomes, Carla, 2021. Understanding decoupled and early weight decay. *Proceedings of the aaai conference on artificial intelligence*. Vol. 35, 8, pp.6777–6785.

Bock, Sebastian and Weiß, Martin, 2019. A proof of local convergence for the adam optimizer. *2019 international joint conference on neural networks (ijcnn)*. IEEE, pp.1–8.

Brown, Tom, Mann, Benjamin, Ryder, Nick, Subbiah, Melanie, Kaplan, Jared D, Dhariwal, Prafulla, Neelakantan, Arvind, Shyam, Pranav, Sastry, Girish, Askell, Amanda, et al., 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33, pp.1877–1901.

Chen, Mark, Tworek, Jerry, Jun, Heewoo, Yuan, Qiming, Pinto, Henrique Ponde de Oliveira, Kaplan, Jared, Edwards, Harri, Burda, Yuri, Joseph, Nicholas, Brockman, Greg, et al., 2021. Evaluating large language models trained on code. *Arxiv preprint arxiv:2107.03374*.

Chen, Tianqi, Xu, Bing, Zhang, Chiyuan, and Guestrin, Carlos, 2016. Training deep nets with sublinear memory cost. *Arxiv preprint arxiv:1604.06174*.

Clevert, Djork-Arné, Unterthiner, Thomas, and Hochreiter, Sepp, 2015. Fast and accurate deep network learning by exponential linear units (elus). *Arxiv preprint arxiv:1511.07289*.

Cortes, Corinna, Mohri, Mehryar, and Rostamizadeh, Afshin, 2012. L2 regularization for learning kernels. *Arxiv preprint arxiv:1205.2653*.

Cubuk, Ekin D, Zoph, Barret, Mane, Dandelion, Vasudevan, Vijay, and Le, Quoc V, 2019. Autoaugment: learning augmentation strategies from data. *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*, pp.113–123.

Davis, Damek, Drusvyatskiy, Dmitriy, Xiao, Lin, and Zhang, Junyu, 2021. From low probability to high confidence in stochastic convex optimization. *The journal of machine learning research*, 22(1), pp.2237–2274.

Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, and Toutanova, Kristina, 2018. Bert: pre-training of deep bidirectional transformers for language understanding. *Arxiv preprint arxiv:1810.04805*.

Fan, Angela, Grave, Edouard, and Joulin, Armand, 2019. Reducing transformer depth on demand with structured dropout. *Arxiv preprint arxiv:1909.11556*.

Fan, Angela, Lewis, Mike, and Dauphin, Yann, 2018. Hierarchical neural story generation. *Arxiv preprint arxiv:1805.04833*.

Gal, Yarin and Ghahramani, Zoubin, 2016. Dropout as a bayesian approximation: representing model uncertainty in deep learning. *International conference on machine learning*. PMLR, pp.1050–1059.

Gao, Leo, Biderman, Stella, Black, Sid, Golding, Laurence, Hoppe, Travis, Foster, Charles, Phang, Jason, He, Horace, Thite, Anish, Nabeshima, Noa, et al., 2020. The pile: an 800gb dataset of diverse text for language modeling. *Arxiv preprint arxiv:2101.00027*.

Ge, Rong, Kakade, Sham M, Kidambi, Rahul, and Netrapalli, Praneeth, 2019. The step decay schedule: a near optimal, geometrically decaying learning rate procedure for least squares. *Advances in neural information processing systems*, 32.

Ginsburg, Boris, Castonguay, Patrice, Hrinchuk, Oleksii, Kuchaiev, Oleksii, Lavrukhin, Vitaly, Leary, Ryan, Li, Jason, Nguyen, Huyen, Zhang, Yang, and Cohen, Jonathan M, 2019. Stochastic gradient methods with layer-wise adaptive moments for training of deep networks. *Arxiv preprint arxiv:1905.11286*.

Gnecco, Giorgio, Sanguineti, Marcello, et al., 2009. The weight-decay technique in learning from data: an optimization point of view. *Computational management science*, 6(1), pp.53–79.

Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron, 2016. *Deep learning*. http://www.deeplearningbook.org. MIT Press.

Gordon-Rodriguez, Elliott, Loaiza-Ganem, Gabriel, Pleiss, Geoff, and Cunningham, John Patrick, 2020. Uses and abuses of the cross-entropy loss: case studies in modern deep learning.

Grill, Jean-Bastien, Strub, Florian, Altché, Florent, Tallec, Corentin, Richemond, Pierre, Buchatskaya, Elena, Doersch, Carl, Avila Pires, Bernardo, Guo, Zhaohan, Gheshlaghi Azar, Mohammad, et al., 2020. Bootstrap your own latent-a new approach to self-supervised learning. *Advances in neural information processing systems*, 33, pp.21271–21284.

Guo, Dugang, Liu, Jun, and Wang, Xuewei, 2021. On development of multi-resolution detector for tomato disease diagnosis. *Journal of intelligent & fuzzy systems*, 41(6), pp.6461–6471.

Hazan, Elad and Kale, Satyen, 2011. Beyond the regret minimization barrier: an optimal algorithm for stochastic strongly-convex optimization. *Proceedings of the 24th annual conference on learning theory*. JMLR Workshop and Conference Proceedings, pp.421–436.

Hendrycks, Dan, Basart, Steven, Kadavath, Saurav, Mazeika, Mantas, Arora, Akul, Guo, Ethan, Burns, Collin, Puranik, Samir, He, Horace, Song, Dawn, et al., 2021. Measuring coding challenge competence with apps. *Arxiv preprint arxiv:2105.09938*.

Hendrycks, Dan and Gimpel, Kevin, 2016. Gaussian error linear units (gelus). *Arxiv preprint arxiv:1606.08415*.

Hernández-García, Alex and König, Peter, 2018. Do deep nets really need weight decay and dropout? *Arxiv preprint arxiv:1802.07042*.

Hinton, Geoffrey, Srivastava, Nitish, and Swersky, Kevin, 2012. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8), p.2.

Holtzman, Ari, Buys, Jan, Du, Li, Forbes, Maxwell, and Choi, Yejin, 2019. The curious case of neural text degeneration. *Arxiv preprint arxiv:1904.09751*.

Holtzman, Ari, Buys, Jan, Forbes, Maxwell, Bosselut, Antoine, Golub, David, and Choi, Yejin, 2018. Learning to write with cooperative discriminators. *Arxiv preprint arxiv:1805.06087*.

Hung, Che-Lun, Hsin, Chine-fu, Wang, Hsiao-Hsi, and Tang, Chuan Yi, 2019. Optimization of gpu memory usage for training deep neural networks. *Pervasive systems, algorithms and networks: 16th international symposium, i-span 2019, naples, italy, september 16-20, 2019, proceedings 16*. Springer, pp.289–293.

Ippolito, Daphne, Kriz, Reno, Kustikova, Maria, Sedoc, João, and Callison-Burch, Chris, 2019. Comparison of diverse decoding methods from conditional language models. *Arxiv preprint arxiv:1906.06362*.

Kaplan, Jared, McCandlish, Sam, Henighan, Tom, Brown, Tom B, Chess, Benjamin, Child, Rewon, Gray, Scott, Radford, Alec, Wu, Jeffrey, and Amodei, Dario, 2020. Scaling laws for neural language models. *Arxiv preprint arxiv:2001.08361*.

Kim, Soojeong, Yu, Gyeong-In, Park, Hojin, Cho, Sungwoo, Jeong, Eunji, Ha, Hyeonmin, Lee, Sanha, Jeong, Joo Seong, and Chun, Byung-Gon, 2019. Parallax: sparsity-aware data parallel training of deep neural networks. *Proceedings of the fourteenth eurosys conference 2019*, pp.1–15.

Kingma, Diederik P and Ba, Jimmy, 2014. Adam: a method for stochastic optimization. *Arxiv preprint arxiv:1412.6980*.

Konečnỳ, Jakub, Liu, Jie, Richtárik, Peter, and Takáč, Martin, 2015. Mini-batch semi-stochastic gradient descent in the proximal setting. *Ieee journal of selected topics in signal processing*, 10(2), pp.242–255.

Kudo, Taku, 2018. Subword regularization: improving neural network translation models with multiple subword candidates. *Arxiv preprint arxiv:1804.10959*.

Kudo, Taku and Richardson, John, 2018. Sentencepiece: a simple and language independent subword tokenizer and detokenizer for neural text processing. *Arxiv preprint arxiv:1808.06226*.

Kulunchakov, Andrei and Mairal, Julien, 2019. A generic acceleration framework for stochastic composite optimization. *Advances in neural information processing systems*, 32.

Lachaux, Marie-Anne, Roziere, Baptiste, Chanussot, Lowik, and Lample, Guillaume, 2020. Unsupervised translation of programming languages. *Arxiv preprint arxiv:2006.03511*.

Larsen, Jan, Hansen, Lars Kai, Svarer, Claus, and Ohlsson, M, 1996. Design and regularization of neural networks: the optimal use of a validation set. *Neural networks for signal processing vi. proceedings of the 1996 ieee signal processing society workshop*. IEEE, pp.62–71.

Lawen, Hussam, Ben-Cohen, Avi, Protter, Matan, Friedman, Itamar, and Zelnik-Manor, Lihi, 2019. Attention network robustification for person reid. *Arxiv preprint arxiv:1910.07038*.

Li, Hao, Xu, Zheng, Taylor, Gavin, Studer, Christoph, and Goldstein, Tom, 2018. Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31.

Li, Mu, Zhang, Tong, Chen, Yuqiang, and Smola, Alexander J, 2014. Efficient mini-batch training for stochastic optimization. *Proceedings of the 20th acm sigkdd international conference on knowledge discovery and data mining*, pp.661–670.

Li, Shen, Zhao, Yanli, Varma, Rohan, Salpekar, Omkar, Noordhuis, Pieter, Li, Teng, Paszke, Adam, Smith, Jeff, Vaughan, Brian, Damania, Pritam, et al., 2020. Pytorch distributed: experiences on accelerating data parallel training. *Arxiv preprint arxiv:2006.15704*.

Li, Xiaoyu, Zhuang, Zhenxun, and Orabona, Francesco, 2021. A second look at exponential and cosine step sizes: simplicity, adaptivity, and performance. *International conference on machine learning*. PMLR, pp.6553–6564.

Liu, Hanxiao, Simonyan, Karen, and Yang, Yiming, 2018. Darts: differentiable architecture search. *Arxiv preprint arxiv:1806.09055*.

Loshchilov, Ilya and Hutter, Frank, 2017. Decoupled weight decay regularization. *Arxiv preprint arxiv:1711.05101*.

Ma, Jerry and Yarats, Denis, 2021. On the adequacy of untuned warmup for adaptive optimization. *Proceedings of the aaai conference on artificial intelligence*. Vol. 35, 10, pp.8828–8836.

Mesnil, Grégoire, Dauphin, Yann, Glorot, Xavier, Rifai, Salah, Bengio, Yoshua, Goodfellow, Ian, Lavoie, Erick, Muller, Xavier, Desjardins, Guillaume, Warde-Farley, David, et al., 2012. Unsupervised and transfer learning challenge: a deep learning approach. *Proceedings of icml workshop on unsupervised and transfer learning*. JMLR Workshop and Conference Proceedings, pp.97–110.

Michel, Paul, Levy, Omer, and Neubig, Graham, 2019. Are sixteen heads really better than one? *Advances in neural information processing systems*, 32.

Moore, Robert C and DeNero, John, 2011. L1 and l2 regularization for multiclass hinge loss models.

Nado, Zachary, Gilmer, Justin M, Shallue, Christopher J, Anil, Rohan, and Dahl, George E, 2021. A large batch optimizer reality check: traditional, generic optimizers suffice across batch sizes. *Arxiv preprint arxiv:2102.06356*.

Nair, Vinod and Hinton, Geoffrey E, 2010. Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27th international conference on machine learning (icml-10)*, pp.807–814.

Norouzi, Sajad and Ebrahimi, M, n.d. *A survey on proposed methods to address adam optimizer deficiencies*.

Patricia, Novi and Caputo, Barbara, 2014. Learning to learn, from transfer learning to domain adaptation: a unifying perspective. *Proceedings of the ieee conference on computer vision and pattern recognition*, pp.1442–1449.

Radford, Alec, Kim, Jong Wook, Hallacy, Chris, Ramesh, Aditya, Goh, Gabriel, Agarwal, Sandhini, Sastry, Girish, Askell, Amanda, Mishkin, Pamela, Clark, Jack, et al., 2021. Learning transferable visual models from natural language supervision. *International conference on machine learning*. PMLR, pp.8748–8763.

Raffel, Colin, Shazeer, Noam, Roberts, Adam, Lee, Katherine, Narang, Sharan, Matena, Michael, Zhou, Yanqi, Li, Wei, and Liu, Peter J, 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The journal of machine learning research*, 21(1), pp.5485–5551.

Ren, Shuo, Guo, Daya, Lu, Shuai, Zhou, Long, Liu, Shujie, Tang, Duyu, Sundaresan, Neel, Zhou, Ming, Blanco, Ambrosio, and Ma, Shuai, 2020. Codebleu: a method for automatic evaluation of code synthesis. *Arxiv preprint arxiv:2009.10297*.

Rojas, Elvis, Kahira, Albert Njoroge, Meneses, Esteban, Gomez, Leonardo Bautista, and Badia, Rosa M, 2020. A study of checkpointing in large scale training of deep neural networks. *Arxiv preprint arxiv:2012.00825*.

Sennrich, Rico, Haddow, Barry, and Birch, Alexandra, 2015. Neural machine translation of rare words with subword units. *Arxiv preprint arxiv:1508.07909*.

Setiawan, Adhi, Yudistira, Novanto, and Wihandika, Randy Cahya, 2022. Large scale pest classification using efficient convolutional neural network with augmentation and regularizers. *Computers and electronics in agriculture*, 200, p.107204.

Shaw, Peter, Uszkoreit, Jakob, and Vaswani, Ashish, 2018. Self-attention with relative position representations. *Arxiv preprint arxiv:1803.02155*.

Song, Xinying, Salcianu, Alex, Song, Yang, Dopson, Dave, and Zhou, Denny, 2020. Fast wordpiece tokenization. *Arxiv preprint arxiv:2012.15524*.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan, 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1), pp.1929–1958.

St John, John, 2021. Adamd: improved bias-correction in adam. *Arxiv e-prints*, arXiv–2110.

Stoffl, Lucas, Vidal, Maxime, and Mathis, Alexander, 2021. End-to-end trainable multi-instance pose estimation with transformers. *Arxiv preprint arxiv:2103.12115*.

Stooke, Adam and Abbeel, Pieter, 2019. Rlpyt: a research code base for deep reinforcement learning in pytorch. *Arxiv preprint arxiv:1909.01500*.

Su, Jianlin, Lu, Yu, Pan, Shengfeng, Murtadha, Ahmed, Wen, Bo, and Liu, Yunfeng, 2021. Roformer: enhanced transformer with rotary position embedding. *Arxiv preprint arxiv:2104.09864*.

Su, Yixuan and Collier, Nigel, 2022. Contrastive search is what you need for neural text generation. *Arxiv preprint arxiv:2210.14140*.

Sun, Zewei, Huang, Shujian, Dai, Xin-Yu, and Chen, Jiajun, 2020. Alleviating the inequality of attention heads for neural machine translation. *Arxiv preprint arxiv:2009.09672*.

Tan, Hao and Bansal, Mohit, 2020. Vokenization: improving language understanding with contextualized, visual-grounded supervision. *Arxiv preprint arxiv:2010.06775*.

Van Laarhoven, Twan, 2017. L2 regularization versus batch and weight normalization. *Arxiv preprint arxiv:1706.05350*.

Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Łukasz, and Polosukhin, Illia, 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Voita, Elena, Talbot, David, Moiseev, Fedor, Sennrich, Rico, and Titov, Ivan, 2019. Analyzing multi-head self-attention: specialized heads do the heavy lifting, the rest can be pruned. *Arxiv preprint arxiv:1905.09418*.

Wang, Hongjun, Wang, Guangrun, Li, Guanbin, and Lin, Liang, 2019. Camdrop: a new explanation of dropout and a guided regularization method for deep neural networks. *Proceedings of the 28th acm international conference on information and knowledge management*, pp.1141–1149.

Webster, Jonathan J and Kit, Chunyu, 1992. Tokenization as the initial phase in nlp. *Coling 1992 volume 4: the 14th international conference on computational linguistics*.

Wei, Colin, Kakade, Sham, and Ma, Tengyu, 2020. The implicit and explicit regularization effects of dropout. *International conference on machine learning*. PMLR, pp.10181–10192.

Wiher, Gian, Meister, Clara, and Cotterell, Ryan, 2022. On decoding strategies for neural text generators. *Transactions of the association for computational linguistics*, 10, pp.997–1012.

Wilson, Ashia C, Roelofs, Rebecca, Stern, Mitchell, Srebro, Nati, and Recht, Benjamin, 2017. The marginal value of adaptive gradient methods in machine learning. *Advances in neural information processing systems*, 30.

Wolf, Thomas, Sanh, Victor, Chaumond, Julien, and Delangue, Clement, 2019. Transfertransfo: a transfer learning approach for neural network based conversational agents. *Arxiv preprint arxiv:1901.08149*.

Wu, Yonghui, Schuster, Mike, Chen, Zhifeng, Le, Quoc V, Norouzi, Mohammad, Macherey, Wolfgang, Krikun, Maxim, Cao, Yuan, Gao, Qin, Macherey, Klaus, et al., 2016. Google's neural machine translation system: bridging the gap between human and machine translation. *Arxiv preprint arxiv:1609.08144*.

Wu, Zhen, Wu, Lijun, Meng, Qi, Xia, Yingce, Xie, Shufang, Qin, Tao, Dai, Xinyu, and Liu, Tie-Yan, 2021. Unidrop: a simple yet effective technique to improve transformer without extra cost. *Arxiv preprint arxiv:2104.04946*.

Xie, Zeke, Sato, Issei, and Sugiyama, Masashi, 2020a. Stable weight decay regularization.

Xie, Zeke, Sato, Issei, and Sugiyama, Masashi, 2020b. Understanding and scheduling weight decay. *Arxiv preprint arxiv:2011.11152*.

You, Jiaxuan, Leskovec, Jure, He, Kaiming, and Xie, Saining, 2020. Graph structure of neural networks. *International conference on machine learning*. PMLR, pp.10881–10891.

Yuan, Zhuoning, Yan, Yan, Jin, Rong, and Yang, Tianbao, 2019. Stagewise training accelerates convergence of testing error over sgd. *Advances in neural information processing systems*, 32.

Zhang, Guodong, Wang, Chaoqi, Xu, Bowen, and Grosse, Roger, 2018. Three mechanisms of weight decay regularization. *Arxiv preprint arxiv:1810.12281*.

Zhang, Hugh, Duckworth, Daniel, Ippolito, Daphne, and Neelakantan, Arvind, 2020. Trading off diversity and quality in natural language generation. *Arxiv preprint arxiv:2004.10450*.

Zhang, Xinyu, Wang, Qiang, Zhang, Jian, and Zhong, Zhao, 2019a. Adversarial autoaugment. *Arxiv preprint arxiv:1912.11188*.

Zhang, Zhi, He, Tong, Zhang, Hang, Zhang, Zhongyue, Xie, Junyuan, and Li, Mu, 2019b. Bag of freebies for training object detection neural networks. *Arxiv preprint arxiv:1902.04103*.

Zhang, Zhilu and Sabuncu, Mert, 2018. Generalized cross entropy loss for training deep neural networks with noisy labels. *Advances in neural information processing systems*, 31.

Zhao, Hengshuang, Jia, Jiaya, and Koltun, Vladlen, 2020. Exploring self-attention for image recognition. *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*, pp.10076–10085.

Zhou, Wangchunshu, Ge, Tao, Xu, Ke, Wei, Furu, and Zhou, Ming, 2020. Scheduled drophead: a regularization method for transformer models. *Arxiv preprint arxiv:2004.13342*.

Zhuang, Fuzhen, Qi, Zhiyuan, Duan, Keyu, Xi, Dongbo, Zhu, Yongchun, Zhu, Hengshu, Xiong, Hui, and He, Qing, 2020. A comprehensive survey on transfer learning. *Proceedings of the ieee*, 109(1), pp.43–76.

Zhuang, Zhenxun, Liu, Mingrui, Cutkosky, Ashok, and Orabona, Francesco, 2022. Understanding adamw through proximal methods and scale-freeness. *Arxiv preprint arxiv:2202.00089*.