Distil-CodeGPT, Distilling Code-Generation Models for Local Use

Emil Malmsten

Supervisors: Prof. Dr. Arie van Deursen, Dr. Maliheh Izadi, ir. Ali Al-Kaswan

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Emil Malmsten
Final project course: CSE3000 Research Project
Thesis committee: Prof. Dr. Arie van Deursen, Dr. Maliheh Izadi, ir. Ali Al-Kaswan, Prof. Dr. Avishek Anand

Abstract

The application of large language models (LLMs) for programming tasks, such as automatic code completion, has seen a significant upswing in recent years. However, due to their computational demands, they have to operate on servers. This both requires users to have a steady internet connection and raises potential privacy concerns. Therefore, this study aims to explore the feasibility of compressing LLMs for code using knowledge distillation (KD), thereby facilitating local usage of these models. Existing research has primarily focused on the efficacy of using KD to compress BERT models for language tasks. Its application to GPT models for coding tasks and the impact of implementing KD in-training, as opposed to the pre-training, remain largely unexplored. To address these gaps we adapted DistilBERT, a pre-training KD algorithm for distilling BERT models for language tasks. Our adapted model, Distil-CodeGPT, utilizes in-training KD to compress LLMs for Python code. The findings of this study suggest that a substantial reduction in model size is achievable, albeit accompanied by a compromise in predictive accuracy. Specifically, using 8 layers, instead of the original 12, resulted in a 24% reduction in disk size and a 28% speed increase, with an accompanying accuracy decrease of 11%. These results show that this approach has potential and is a solid first step toward smaller code models.

1 Introduction

Over the previous years, the use of Large Language Models (LLMs) for coding tasks has become increasingly popular among developers. A notable example of this trend is the GitHub Copilot application, which grew a user base of 400,000 subscribers within a month of its launch in 2021 [18]. These tools have also, reportedly, contributed to more productivity, user satisfaction, and operational efficiency for developers [8].

However, a major problem with these applications is their reliance on server-based operations. Their considerable size and computational demands prevent them from running efficiently on local machines. This situation potentially excludes certain user demographics, such as those living in regions with suboptimal internet connectivity or individuals working with classified information. Additionally, the large electricity consumption of these large models raises concerns about their environmental impact.

One potential solution to this issue is compressing the models, making them more suitable for local usage. Some research has already shown promising results in utilizing knowledge distillation (KD) to do this compression. Two studies demonstrated significant speedups by decreasing the size of the model, all while maintaining comparable language-predicting abilities [13, 7]. However, these findings predominantly focus on pre-training KD of BERT-type [3] language models. The applicability of KD for code predictions, in-training KD, and KD on GPT models [12] remains largely unexplored.

To fill these gaps, the question this research will try to answer is: *what are the effects of compressing a CodeGPT [10] model, regarding size, accuracy, and speed, through the application of in-training KD?* The way of answering it is by adapting the DistilBERT algorithm, as proposed by Sanh et al. [13], to do in-training KD on CodeGPT models.

The main contributions of the paper are as follows:

- Using the algorithm, we show that it is feasible to use KD for compressing code models, albeit with a larger accuracy loss compared to its application in language models. We also investigate the reasons for this discrepancy and how to potentially resolve it.
- We provide some indication that in-training KD is less efficient and uses more resources than pre-training KD.
- We also examine what the optimal configurations for applying KD to coding tasks are, focusing on the parameter selection and the student-teacher model setup

Finally, our pre-trained models can be found on Hugging Face[1], under the name BRP-Malmsten-<MODEL_NAME>. Our code base, alongside instructions on how to train and test the different models, can be found on GitHub[2].

2 Preliminaries

In this section, the terms and concepts which the reader may not be familiar with will be explained.

2.1 Language Models

In this paper, two types of language models will be explored, namely BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pretrained Transformer). Since they are both are using a transformer architecture, that will first be explained.

Transformers

Transformer models, primarily used for natural language processing (NLP) tasks, are a type of model architecture developed to deal with sequential data. They utilize an attention mechanism that assigns varying degrees of importance to different parts of the input, enabling simultaneous processing of all sequence steps and the ability to effectively capture long-range dependencies. With their encoder-decoder architecture, they map input sequences to output sequences, making them suited for tasks like code prediction.

BERT

BERT is a transformer-based machine learning technique developed in 2018 by Devlin et al. [3]. BERT applies the bidirectional training of the transformer to language modeling. Unlike traditional language modeling, which predicts the next word in a sequence, BERT makes use of a mechanism called

[1]https://huggingface.co/AISE-TUDelft

[2]https://github.com/AISE-TUDelft/LLM4CodeCompression/tree/main/distill-CodeGPT

”Masked Language Model” (MLM) which randomly masks words in the sentence and then tries to predict them. This allows the model to understand the context in both directions (left and right of the word), from there the term ”bidirectional”.

GPT

GPT, developed by OpenAI in 2018 by Radford et al. [12], is another transformer-based model that utilizes unsupervised learning for understanding natural language. Unlike BERT, GPT is a unidirectional model. When GPT reads a sentence, it reads from left to right and uses the previous words to predict the next word in the sequence. After GPTs initial training, to predict the next word, it can be fine-tuned on more specific tasks such as question answering and summarization.

2.2 Knowledge Distillation

Knowledge distillation (KD) is a compression technique that involves training a smaller model, or the ’student’, to emulate the performance of a larger model referred to as the ’teacher’. The primary focus is not necessarily on replicating the exact classification predictions of the teacher model but on the probability distribution over the classes that the teacher model outputs. These probabilities carry valuable information about the relationships between different classes, and this is part of the ’knowledge’ that the teacher model can pass on to the student model. To achieve this, we typically use two key components: a ”hard” loss and a ”soft” loss, these are explained below. The total loss for KD is usually a weighted sum of these two losses, where the weights can be adjusted to balance the emphasis between matching the teacher’s predictions and correctly classifying the training instances.

Hard Loss

The hard loss ensures that the student model correctly classifies instances in the training set. It is calculated between the student model’s predictions and the actual ground-truth labels in the training set, just like any standard training process. Commonly used functions for calculating the hard loss include Cross Entropy (CE) loss.

Soft Loss

The soft loss, on the other hand, encourages the student model to mimic the output probability distribution of the teacher model. The teacher model’s predictions are typically transformed using a technique known as temperature scaling. The aim is to make the model’s output probabilities more ’soft’ or ’smooth’, emphasizing the relative differences between class probabilities. This smoothness provides additional information about the class relationships, beyond what is in the hard labels alone.

The method for calculating the loss is often CE loss, between the teacher and student models’ outputs. Another method is the cosine similarity loss which is used to ensure that the student model’s intermediate representations align closely with those of the teacher model. The cosine similarity measures the cosine of the angle between the student’s and teachers’ representation vectors and the aim is to maximize it.

Pre-training vs In-training Knowledge Distillation

KD can be performed both pre-training and in-training. Pre-training KD means that the teacher is already fine-tuned before the KD and will not learn during the process. In-training KD, or online KD, on the other hand, means that the teacher does also learn during the process. Commonly the teacher is not fine-tuned when doing in-training KD but this does not necessarily have to be the case.

2.3 Other Compression Techniques

This paper will focus on the compression of language models using KD but other compression techniques have also had promising results. The most prominent ones are pruning and quantization.

Pruning

Pruning involves removing unnecessary weights or connections within the neural network. This is often done iteratively with the least important being removed first. The pruning usually continues until a desired sparsity level is reached, but other termination conditions can also be applied.

Quantization

Quantization is a method used to compress neural networks by representing weights and activation tensors with low-bit representations like 8-bit. This has two benefits, firstly it shrinks the model’s disk size and secondly, GPUs are faster when doing fixed point arithmetic.

3 Related Work

Several studies have been conducted to understand the effects of compressing language models using various techniques. In this section, we will elaborate on their approaches and results.

3.1 Studies on Knowledge Distillation

As highlighted in section 1, the use of KD for compressing language models has shown promising results. Two studies that used only KD for compression, without pruning or quantization are discussed below.

DistilBERT

The study by Sanh et al. [13] used a tripartite loss function, integrating a hard CE loss, a soft CE loss, and a cosine-distance loss during the pre-training phase. This approach yielded a 60% performance improvement and a 40% resource reduction while preserving approximately 97% of the original language understanding capabilities.

With the same model, they also conducted an experiment on distilling a GPT2 model. This is not reported in the paper but can be seen on their GitHub[3]. The results showed an increase in accuracy by 30% despite decreasing the number of layers from 12 to 6. However, one potential issue with these results is that the student model was fine-tuned on the dataset it was benchmarked on, but the teacher model (the baseline) was not.

[3]https://github.com/huggingface/transformers/tree/main/examples/research_projects/distillation

TinyBERT
Another study achieved even better compression by also including losses from the hidden attention and embedding layers [7]. This work introduced a two-stage learning framework that performed transformer distillation during the pre-training and task-specific learning stages. The model was reduced to 13.3% of its original size and had a speed increase of 9.4x while retaining 96.8% of its performance.

3.2 Studies on Pruning and Quantization

Pruning
There are several notable studies that have explored the effectiveness of pruning. One example examines the use of Optimal BERT Surgeon (oBERT) [9]. The method leverages approximate second-order information and enables the pruning of blocks of weights. This achieved a 10-fold reduction in size while preserving nearly identical performance compared to the dense BERT-base. Pruning has also been successfully employed on GPT models, as demonstrated by Frantar and Alistarh [4]. They showed that it was possible to discard over 100 billion weights during inference, culminating in a 50% decrease in model size.

Quantization
Quantization has also shown promising outcomes. One study demonstrated a group-wise quantization scheme together with a Hessian-based mixed-precision method to compress a BERT model [15]. The approach resulted in a mere 2.3% performance loss while reducing precision quantization to 2-bits.

Combining the Techniques
A different approach combined KD with quantization and pruning, seeking to optimize model efficiency on CPUs, as opposed to the conventional reliance on GPUs [14]. The strategy involved pruning for speed augmentation, KD for accuracy enhancement during fine-tuning, and post-training quantization for further model optimization without additional training steps. The result was near-perfect accuracy retention and a 50% performance improvement compared to the state-of-the-art (of when the paper was written).

3.3 Studies on Compressing Code Models

Concurrent with this study, three other works have explored code compression, all employing slightly different techniques.

CodeGPT on XTC
This study, by de Moor [2], compressed a CodeGPT model utilizing the XTC (eXTreme Compression) pipeline. This entailed using KD to reduce the layers and also employing quantization. The compressed model, a 6-layer one with 1-bit weight and 8-bit activation quantization, signifies a 15-fold reduction in size while preserving a fair amount of the original accuracy.

MP and PEG PTQ on CodeGPT
In their work, Storti [17] applied various Post Training Quantization (PTQ) techniques on a CodeGPT model. One such technique is PEG, which involves splitting activation tensors into equal-sized sets, each encompassing similar activation ranges. This is done in order to minimize the decrease in accuracy attributed to quantization. Using this technique, the authors managed to compress a CodeGPT model to 25% of the original size while maintaining almost all their accuracy.

CodeGPT on Intel
The work by Sochirca [16] employs the Intel-extension-for-transformers toolkit [6] to prune and quantize the CodeGPT model post-training. They do this to enable running the model efficiently on a CPU. Their model achieves a 60% size reduction while maintaining an acceptable accuracy level, scoring 30.5% on ES and 9.0% on EM. However, the study did not record improvements in memory usage or inference time.

4 Method

Our research method involved adapting DistilBERT, a pre-training KD algorithm for distilling language models. The study on DistilBERT and the results of it can be found in section 3. Our changes turned DistilBERT into an in-training KD algorithm for compressing code models. The decision to leverage this approach stemmed from DistilBERT's demonstrated success with BERT language models. Moreover, altering an existing model was deemed more efficient than devising entirely new KD methods.

4.1 Training and Distilling

The data preparation and training processes are both fairly standard ones for neural networks. There are, however, a few differences since KD is performed during the training. Firstly, both a soft and a hard loss are used. The hard loss contributes to 33% of the total student loss. The remaining 67% is a soft loss, equally divided between the CE loss and cosine similarity loss. This process is exactly the same as DistilBERT used.

The major difference between Distil-CodeGPT and DistilBERT is that the teacher model undergoes training alongside the student model since it uses in-training KD. The teacher's loss comprises solely of the hard CE loss. Another difference is that our model works for code while DistilBERT on language. This adaptation was simply a matter of selecting a different teacher and student model configuration and tokenizer.

We also took a slightly different approach in the research by benchmarking different models, all with slight modifications regarding parameters or student-teacher setup. A detailed description of the different models can be found below. The intention of doing this is to aid future research by providing insights into which setup performs best.

4.2 Parameters

The teacher model used was the CodeGPT-py-adapted model from Microsoft, as first introduced in [10]. Its training architecture closely mirrored that of GPT2. This teacher model is already fine-tuned on Python code. The student model had the same configuration but without the pre-trained weights. The reason it was used is that it was state-of-the-art at the time.

During the experiment, there were several options for the parameters but we stuck with the defaults as seen in the GitHub [3] associated with the paper by Sanh et al. [13]. The reason for this was that DistilBERT did have good results with those parameters and we saw no apparent reason to change it. This basically meant that we used a temperature of 2, the soft CE-loss, cosine loss, and hard CE-loss all took the value 0.33. The only exception we made was to change the number of epochs to 1, down from 3, to decrease, the already long, training time.

4.3 Experimental Configuration

In order to identify potential causal factors of the observed outcomes and to propose future improvements for distillation, several variant models were systematically benchmarked. The Standard Model was subjected to trials with layer counts of 4, 6, 8, 10, and 12, whereas the alternative models were benchmarked specifically with a layer count of 8. Instructions for running each model are provided on our GitHub [2].

Standard Model

The Standard Model has the teacher model, parameters, etc. as presented earlier in this section.

Not-Adapted Model

With this experiment, we explore the difference the teacher models make by using the CodeGPT-py model instead of the modified CodeGPT-py-adapted model as the teacher. The key distinguishing factor between these models is that CodeGPT-py is trained from scratch, whereas CodeGPT-py-adapted begins the training with identical weights and biases as GPT2.

Non-Fine-Tuned Teacher Model

As previously highlighted, the teacher model in our experiments is typically fine-tuned on code. However, in one experimental variation, we utilized a non-fine-tuned version of the teacher model. This variant is denoted as the NFTT Model (Non-Fine-Tuned Teacher Model). This experiment was performed to see the impact of not having to fine-tune the teacher before distillation.

Altered-Parameter Model

On the GitHub repository[3] for the adapted model, default values for parameters were provided which we used in our experiments. However, the repository also detailed an alternative set of parameters. These encompassed other values for the soft CE-loss (5.0), cosine loss (1.0), and hard CE-loss (2.0). This test thus aimed to determine the influence of the parameters.

8-Epoch Model

The Standard Model is typically trained over a single epoch over the entire dataset. Conversely, the Extended-Epoch Model is trained over 8 epochs, but on a smaller subset of the data. This subset comprises the 50% shortest sequences, which is done to decrease the training time. The experiment was performed to see how additional training data would impact the results.

Pretrained-Weights Model

This model had the student using the pre-trained weights from the CodeGPT-py-adapted model. The student thus has some capabilities in predicting code before the distillation. The experiment aimed to explore the worth of pre-training the student.

5 Setup

The details of the setup illustrate the research questions we aim to answer and provide guidance on how the research can be replicated.

5.1 Research Questions

In this research, we answer the question: *what are the effects of compressing a CodeGPT model, regarding size, accuracy, and speed, through the application of in-training KD?* To aid in answering this question we also answer these subquestions:

1. *What factors contribute to better/worse accuracy of the distilled model?* We try different models, as outlined in subsection 4.3, in order to see which has the highest accuracy. Thereafter we draw conclusions based on these findings.
2. *What is the effect on the accuracy of the model as the layer count decreases?* We try a standard model with several different layer counts to discover how decreasing it changes the accuracy.
3. *How does in-training KD compare to pre-training KD?* We compare our results, which use in-training KD, to those of DistilBERT which uses pre-training KD. We also benchmark a model which uses in-training, but without a fine-tuned teacher, to see the differences.

5.2 Implementation Details

Environment

The benchmarks were run on an AMD EPYC 7402 24C 2.80 GHz 4x NVIDIA Tesla V100S 32GB GPU. The training was done on an RTX A6000 48GB GPU. The operating system is Red Hat Enterprise Linux 8 and the project was run with Python 7.12. The packages and libraries used can be seen in the environment.yml on our GitHub[2]. These can be installed using Conda for running on a Linux machine.

Packages

There were several libraries used but the one having the most impact was Pytorch[4] 1.13 since it provided the necessary deep learning functionalities and GPU support to efficiently train this model on large-scale language datasets.

Also, the transformers library from Hugging Face[5] played an important role in downloading the datasets, models, and configurations. This was useful for both benchmarking and training. The version used was 4.27.4.

A full list of packages and versions can be found in the environment.yml file on our GitHub[2].

[4]https://pytorch.org/

[5]https://huggingface.co/transformers

Dataset

The dataset used for this study was CodeXGlue [10]. The dataset consisted of 150k lines of Python code, with 95k designated for training, 50k for testing, and 5k for validation. To protect privacy, uncommon string literals and numbers are masked with the tags <STR LIT> and <NUM LIT> respectively.

5.3 Evaluation

The Benchmarks

The evaluation of the results involved comparing the output of the teacher model with that of the student model. The primary metrics used for testing the accuracy are Edit Similarity (ES) and Exact Match (EM). ES calculates the likeness between two strings or data sequences, determined by the minimum number of edits (insertions, deletions, or substitutions) needed to transform one string into the other. EM is used to evaluate if the predicted output exactly matched the reference output, with no discrepancies.

When discussing the results mostly the ES metric is used since it is more representative of what we want to achieve, good predictions. Since EM only gives a score when the test set is perfectly matched, it is less indicative of a good prediction. For example, say the test set contains this snippet:

```
for i in range(10):
    x += i
```

If the model were to predict the same code but with another number instead of 10 the EM score would be 0 although it is still a useful prediction. Es, however, would give almost 100%.

The disparity in size between the models was assessed by comparing their parameter counts, their sizes on disk, and the amount of memory they used on the processing unit in question. The difference in efficiency was determined by measuring the relative time taken by each model to generate predictions. We did this by measuring the number of predictions that could be made in one second.

All evaluations were first done on GPU for nine evenly distributed checkpoints. These checkpoints represented how many percent of the dataset the model was trained on (10%, 20%,..,90%). The final model was benchmarked on both CPU and GPU. The ES and EM were measured from the final model, and the size and inference were taken as the mean over all the checkpoints + the final model.

Exceptions

There were a few exceptions made to the benchmarks that, according to us, have no impact on the results. For reproducibility and integrity reasons, we will naturally still share them.

- For the 4-layer model, the tests were made on 6 checkpoints instead of 9, due to a calculation error. We decided not to run the model again since this problem only introduces a bit more variance in the average speed and size of the 4-layer model.
- For the 4-layer model, the inference of the first checkpoint was tossed because it was over 50% slower than the average of the rest. Since it was the first benchmark performed, we believe that it had to do with the dataset not being cached at the time of running it.
- In the dataset, there was one sequence, number 655 in seed 42, which caused an error. Therefore, this single sample was swapped with another one. Since it was one, out of a thousand, we believe that it does not affect the results. It might, however, be an indication of a larger underlying problem but we also deem this unlikely.
- On the 8-Epoch model, we only benchmarked on the final model, not every single epoch. This is because the only relevant results are the ES and EM.
- On all the alternative models, we did not run tests on the CPU since they are expected to be very similar to the ones of the standard 8-layer model. Furthermore, they are not relevant to answering the research question.

Model	Params	Size	Inf.	ES	EM
Baseline	124	510	26	39.1	14.5
12 Layers	124	510	24	30.3	6.4
10 Layers	110	450	27	29.5	6.1
8 Layers	96	390	31	29.6	6.3
6 Layers	82	340	36	28.7	6.4
4 Layers	68	280	43	27.6	5.9

Table 1: The results on the standard models with different amounts of layers. Inf. stands for inference and is measured in samples per second

6 Results

Several benchmarks were performed over the different models to test their accuracy, speed, and size. The most notable observations and results will follow below.

6.1 Performance of the Standard Models

As shown in Table 1, both the number of parameters (shown in the 'params' column) and the model's size on the GPU (shown in the 'size' column) decrease linearly. Specifically, we see a decrease of around 14M parameters or 60 MB of disk size for every 2 layers we remove. The inference speed (shown in the 'inf.' column) also decreases as the model gets smaller, but the rate of decrease slows down over time. These findings meet our initial expectations. However, the accuracy of the model is not as high as hoped for. Even the 12-layer model performed about 9 points worse in terms of ES and 8 points worse in terms of EM compared to the baseline, which has the same amount of layers.

In Figure 1 it seems that the performance of the models, unrelated to their amount of layers, follows the same curve. As the layer count increases, the models have slightly better performance throughout the training. It is however only slightly better as the difference between the smallest model (4 Layers) and the largest model (12 Layers) is a mere 2.7 points.

6.2 Performance of the Alternative Models

The performance of the alternative models can be seen in the legend of Figure 1. The data suggest that the Not-

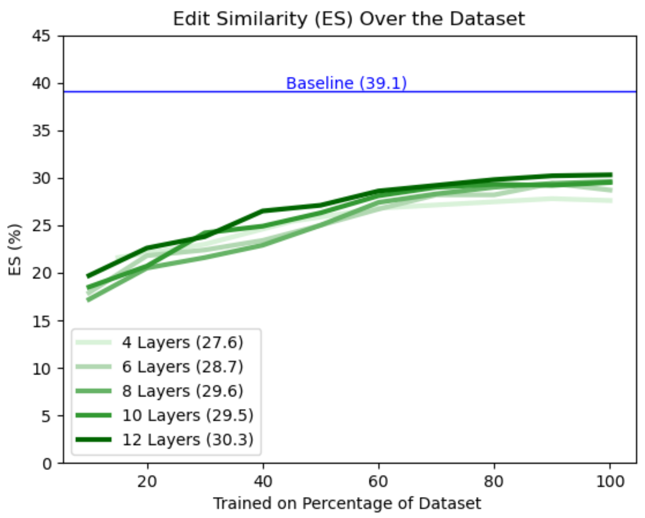

Figure 1: The increase in ES as the models with different layers are trained. The darker the shade of green, the more layers the model has. The parenthesis after the model name is the final ES score

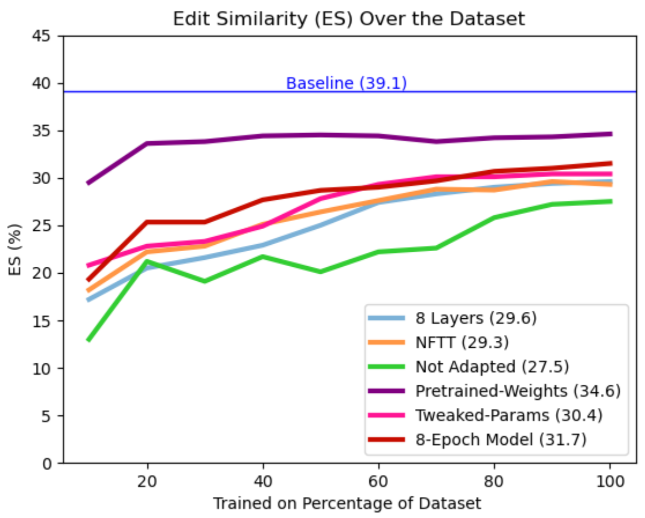

Figure 2: The increase in ES as the different model are trained. The parenthesis after the model name is the final ES score

Adapted model performs the worst among the models and the Pretrained-Weights model by far the best. Also, both the 8-Epoch model and Tweaked-Params model perform better than the standard 8-layer one.

As shown in Figure 2, there are minor differences in the performance of the alternative models over time. The Not-Adapted Model varies more, while the others show major improvement at first to then stabilize. Another interesting observation is that the Tweaked-Params model stabilizes after being trained on about 70% of the training set and the Pretrained-Weights one just after 20%. The Not Adapted model and the 8-Epoch model seem to not even have stabilized after training on all the data. However, none of the models seem to be able to reach the baseline even if they were to get more training.

7 Discussion

The results of the study are a bit worse than expected and in this section, we will try to discern the reasons for this. We will also talk about the threats to the validity of the study and explore the potential studies that could further our effort of compressing code models in the future.

7.1 Comparison with DistilBERT

DistilBERT demonstrated moderately better accuracy retention in comparison to our results. It achieved a compression rate of 40% and an inference speedup of 60% while preserving 97% of the original accuracy. In contrast, our 8-layer model demonstrated an approximate speedup of 20%, on both CPU and GPU, a size reduction of 24%, while preserving only 89% of its language understanding capacity.

The size discrepancy can be attributed to the number of layers used. Our 4-layer model was closer in size and speed to what they used, with a size reduction of 65% and speed enhancement of 40%. However, the underlying reasons behind the accuracy difference between the models are more difficult to discern. Below are some of the explanations that we find most plausible.

The Student Model

The aim of DistilBERT is to distill a natural language model, and subsequently, they also used a student model pre-trained on language. Our aim is to predict Python code but we started with a model that was only pre-trained on language, not code. This decision may have inadvertently compromised our results since a base model with some level of code understanding could have offered a more suitable starting point. This thesis is supported by the superior results of the Pretrained-Weights model, which was the model pre-trained on code.

Number of Epochs

The underperformance of our Distil-CodeGPT compared to DistilBERT could also be attributed to the fewer epochs used in training. This claim is backed by the fact that the 8-Epoch model had superior performance compared to the other 8-layer models, it even outperformed the standard 12-layer model. If the model were to be trained over the entire dataset for more epochs, we could come to see an even larger increase in performance, although this could also lead to overfitting. Hence, additional experiments are required to ascertain the optimal training duration.

This explanation does, however, have some counterarguments. As earlier mentioned, only the 8-Epoch model and the Not-Adapted model seem to have the potential to increase their accuracy with more training. All the other models, including all the standard ones, seem to plateau after being trained on 70-80% of the dataset. The Pretrain-Weights model even plateaus after 20%. At the least, this means that some other measures, except merely training the models for more epochs, must be taken to achieve more accuracy. Especially if the goal is to reach the accuracy baseline, which neither of the models seem close to achieving.

Parameter Selection

Given that the authors of DistilBERT did not elaborate on their parameter selection, there is uncertainty about whether the default parameters from their GitHub repository match those in their study. We used the default ones provided on their GitHub, so if they used different ones for their study, this could partly explain the performance difference between our model and theirs. A marginal accuracy improvement observed in our Tweaked-Params model suggests that varying parameter values may influence model efficacy. For that model, only the weights of the loss functions were altered but modifications to parameters such as temperature and learning rate might potentially enhance accuracy even further.

In-training vs Pre-training

One major difference between Distil-CodeGPT and DistilBERT is that ours utilizes in-training KD, as opposed to pre-training KD. This factor could contribute to our model being less accurate. This is further supported by the fact that the NFTT model, which used a more classical in-training algorithm since the teacher was not fine-tuned, had a slightly worse performance than the standard 8-layer model. Moreover, using in-training KD also slowed the training process. Training two models simultaneously on the GPU means that smaller batch sizes have to be chosen (due to VRAM memory limits) and backtracking had to be done twice as often.

7.2 Potential of Compressing Code Models

Results of studies on compressing code models, which ran in parallel to this one, indicate that code model compression is not inherently more challenging than language model compression. They adapted algorithms for compressing language models to instead compress code models and managed to get equivalent results. Both de Moor [2] and Storti [17] were able to significantly compress their models while retaining almost all the original accuracy.

We also believe that our results do not necessarily indicate that compressing code models is more difficult than language models. This is supported by the inferior performance of our 12-layer model relative to the baseline. It suggests that the reduced accuracy is not due to the smaller model size, but other factors. Further evidence of this is that both the 8-Epoch and the Tweaked-Params models have slightly better accuracy than the 12-layer standard one, and the Pretrained-Weights model is far better.

7.3 Other Observations

It is interesting to note, in Figure 1, that all the standard models seem to have a similar training curve, disregarding some random noise. This seems to imply that the amount of training needed might remain constant regardless of model size.

The configuration and pre-trained weights of the teacher model also have significance it seems, even though it trains during the distillation. This can be concluded by observing the poor performance of the Not-Adapted model which receives worse results than the 4-layer model.

7.4 Threats to Validity

Internal

One internal threat to the validity is the evaluation metrics employed. The problem is that it may not accurately reflect the true performance of the model, especially if the goal is to give useful predictions. For instance, a model which receives a high ES score could still generate unreliable code. Consider an example where the model predicts a function but misrepresents one character, for example altering a less-than operator to a greater-than operator. Despite the majority of the prediction being correct, yielding a high ES, such a minor bug could be challenging for the user to find.

On the other hand, it could also give a score that is low compared to the usability of the generated code. If the ground truth is an expression like x = x + y and the model gives the prediction x += y, it will not provide 100% ES while the prediction is equivalent.

Another internal threat to the validity is the inherent quality of the code from the dataset. If the dataset has buggy code, deprecated code, or code employing bad practices, it is likely that the model will generate predictions having the same issues. The reason for this is that the model does not know what is considered good code, it is simply trained to predict code based on what it has learned from the training data. This concern also links back to the discussion on the evaluation metrics used, a model that generates suboptimal code is undesirable even though it receives a high ES score.

External

An external threat to the validity is that our results may not generalize to other GPT models. Our testing was confined to CodeGPT-py and CodeGPT-adapted-py. It is plausible that other models might experience a larger/smaller performance decline during distillation. For example, some research suggests that higher-accuracy teachers perform less effectively during distillation [1]. It is also conceivable that some models are already optimized to their smallest feasible size or they may possess inherent structural characteristics that impede effective KD. Naturally, the opposite may also be the case, that some models are significantly larger than necessary and therefore can be distilled more effectively.

7.5 Future Work

There are several directions that future research on KD may take. One approach is to pre-train the student model on code-related tasks before learning from the teacher, as we did for the Pretrained-Weights model. This could provide a good starting point and therefore potentially better performance. Fine-tuning the student on the dataset after the teacher-led training might also provide improvements.

Our findings also underscore the potential benefits of identifying the optimal parameters for distilling code models. This could be done by performing more experiments with parameter changes, and identifying which have the largest impact. Similarly, longer training with more epochs covering the entire dataset, instead of just the 50% shortest sequences like the 8-Epoch model, could result in better performance. Both these approaches do, however, call for substantial GPU resources. In our study, training a single epoch on a 48GB RTX A6000 required approximately 17 hours.

Finally, the adaptation of other KD techniques for code models, such as those proposed by Jiao et al. [7], could yield intriguing results. It would also be interesting to explore the potential of further refining methods like that of de Moor [2], which blends KD with quantization. Using the methods that performed best in compressing language models seems like a good starting point for compressing code models.

8 Conclusions

In our study, we adapted DistilBERT, a pre-training KD algorithm developed by Sanh et al. [13] for compressing LLMs for natural language tasks. The adapted model, which we call Distil-CodeGPT, makes use of in-training KD to compress LLMs made for predicting Python code.

The findings demonstrated that while Distil-CodeGPT enabled significant model compression, it did also reduce the language-comprehension capabilities moderately. One of our compressed models, with just one-third of the original layers, retained about 70% of its language understanding ability. Another, with two-thirds of the layers, saw a 90% retention. We noted that in-training did not offer significant benefits compared to pre-training KD, but this conclusion requires further confirmation through additional research. We also saw that pre-training the student model on code before the distillation had significant benefits.

The outcomes suggest that our method needs further refinement. Nevertheless, this initial effort to condense language models and the explicit acknowledgment of its potential constraints can provide value for future studies. Such future studies could include the modification of model parameters, like epochs and loss functions, and the potential benefits of using a student model pre-trained on code. We also believe that using other algorithms than DistilBERT as the basis for the compression has potential.

9 Responsible Research

Scientific research demands a high degree of responsibility and ethical consideration. It is also necessary to ensure that future researchers can follow what has been, both to build upon and to verify it. Therefore, this section will be aimed at discussing the ethical considerations and reproducibility of the study.

9.1 Ethical Considerations

Copyright and Privacy

The datasets usage, for training these models, is a point that may raise ethical and legal issues if not appropriately addressed. For example, a class action lawsuit has already been directed towards GitHub Copilot for copyright infringement, though no decision has yet been rendered [5]. This shows the importance of taking the right precautions when working with data.

To ensure that we adhered to copyright laws and principles, our datasets were sourced from open-access platforms. We believe that the use of open-source information not only ensures ethical conduct but also promotes transparency and collaborative research in AI. Furthermore, to mitigate privacy concerns, our dataset contains only the most common numbers and strings. The remaining ones are in the format <STR LIT> and <NUM LIT> depending on whether it is a string or number, respectively. This approach ensures that our datasets are sufficiently generic while significantly reducing the risk of any inadvertent release of personal information.

Environmental Impact

LLMs require substantial amounts of energy. For instance, training GPT-3, which has 175B parameters, is estimated to require 1,287MWh of energy, leading to an estimated carbon emission of 500 tonnes [11]. Running it daily for millions of users is not included in this calculation.

From this, it could be argued that it is unethical to research these models. Our experimental approach used multiple rounds of training, and when people use these models even more emissions would follow. However, we believe this study to be ethical as the aim is to compress these models, making them use less energy. Since the results of these studies could lead to vast amounts of users running queries on small efficient models instead of unnecessarily large ones, emissions could be heavily reduced. Furthermore, our experiments were done on relatively small models and we took care not to run experiments unnecessarily.

AI Tools

ChatGPT served as a tool in various stages of this study. Primarily, it was used to refine grammar and fix spelling mistakes. It also provided clarifications on some of the principles of KD. However, it should be noted that no text within this paper was composed originally by ChatGPT, nor was it used as a source for any information.

We believe that using it in this manner poses no ethical concern. Employing it for writing merely ensures better readability, without infringing upon any copyright or privacy regulations. Furthermore, using it for learning about specific subjects is simply a form of research, especially since the information was not used directly in the paper.

The prompt we used for writing was: *Highlight the parts of this text that do not use academic language and give suggestions on how it can be improved. Be concise and precise. Keep the citations and other latex-related text as is.* The prompt for learning about some concepts within KD was: *explain in simple language how X works*, where X was words such as attention or temperature.

We also used Grammarly, another tool powered by AI technology. This tool was also used in order to better readability and avoid spelling and grammar mistakes.

9.2 Reproducibility of the Study

The reproducibility of scientific research is paramount to maintaining integrity and fostering progress in the field. Our study was conducted with these values at its core. For instance, the code developed for this research is made publicly available on GitHub footnote 2 and the pre-trained models on Hugging Face footnote 1. This approach not only ensures that our research can be independently verified but also promotes collaborative development and refinement of the techniques we employ.

References

[1] Jang Hyun Cho and Bharath Hariharan. On the efficacy of knowledge distillation, 2019.

[2] Aral de Moor. Codegpt on xtc. 2023.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[4] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

[5] Brendan Hodgens and Shae Sawyer. Lawsuit on ai for copyright infringement. URL https://theeditionga.com/index.php/2023/04/15/lawsuit-on-ai-for-copyright-infringement/.

[6] Intel. Intel extension for transformers. https://github.com/intel/intel-extension-for-transformers, 2023. Accessed: 13-06-2023.

[7] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. TinyBERT: Distilling BERT for Natural Language Understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4163–4174, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.372. URL https://aclanthology.org/2020.findings-emnlp.372.

[8] Eirini Kalliamvakou. Research: quantifying GitHub Copilot's impact on developer productivity and happiness — The GitHub Blog — github.blog. https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/, 2022. [Accessed 11-May-2023].

[9] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The Optimal BERT Surgeon: Scalable and Accurate Second-Order Pruning for Large Language Models, October 2022. URL http://arxiv.org/abs/2203.07259. arXiv:2203.07259 [cs].

[10] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021. URL https://arxiv.org/abs/2102.04664.

[11] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model, 2022.

[12] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[13] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. October 2019. doi: 10.48550/arXiv.1910.01108. URL https://arxiv.org/abs/1910.01108v4.

[14] Haihao Shen, Ofir Zafrir, Bo Dong, Hengyu Meng, Xinyu Ye, Zhe Wang, Yi Ding, Hanwen Chang, Guy Boudoukh, and Moshe Wasserblat. Fast DistilBERT on CPUs, December 2022. URL http://arxiv.org/abs/2211.07715. arXiv:2211.07715 [cs].

[15] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Q-BERT: Hessian Based Ultra Low Precision Quantization of BERT. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):8815–8821, April 2020. ISSN 2374-3468. doi: 10.1609/aaai.v34i05.6409. URL https://ojs.aaai.org/index.php/AAAI/article/view/6409. Number: 05.

[16] Dan Sochirca. Compressing code generation language models on cpus. 2023.

[17] Mauro Storti. Leveraging efficient transformer quantization for codegpt: A post-training analysis. 2023.

[18] Business Telegraph. Github copilot adds 400k subscribers in first month - cio dive — businesstelegraph.co.uk. https://www.businesstelegraph.co.uk/github-copilot-adds-400k-subscribers-in-first-month-cio-dive/, 2022. [Accessed 11-May-2023].