DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Data Engineering and Analytics

Exploring the Possibilities of Applying Transfer Learning Methods for Natural Language Processing in Software Development

Wei Ding

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master´s Thesis in Informatics: Data Engineering and Analytics

Exploring the Possibilities of Applying Transfer Learning Methods for Natural Language Processing in Software Development

Erkundung der Möglichkeiten der Anwendung von Transfer-Lernmethoden für die Verarbeitung natürlicher Sprachen in der Softwareentwicklung

Author:	Wei Ding
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Ahmed Elnaggar
Submission Date:	05 February 2021

I confirm that this master’s thesis in informatics: data engineering and analytics is my own work and I have documented all sources and material used.

Munich, 05 February 2021 Wei Ding

Acknowledgments

First of all, I would like to thank Prof. Dr. Florian Matthes for giving me this opportunity to work on this thesis in the SEBIS chair.

Secondly, I would like to offer my gratitude to my thesis advisor Ahmed Elnaggar for his guidance during this thesis. He offers his help generously, answers my every question patiently, and motivates me continuously throughout this thesis. I would not have the chance to use one of the most advanced hardware resources in the world and work so smoothly on such an up-to-date research topic if without his help.

This thesis is finished mainly from the year 2020 to 2021. It is an unusual year because of the Coronavirus SARS-CoV-2. I still remember the first meeting of this thesis in Ahmed's office. We chatted a bit about a virus wide-spreading in Asia from Wuhan. We hoped that it would disappear when the temperature went higher. At that moment, we did not know that this virus would come to Germany and Europe immediately and influence our study, work, and life so largely. And that meeting is the only face-to-face meeting for this thesis I had with Ahmed in his office.

Therefore, I would like to thank my husband, Qunfei, for his support when we have to take care of and teach our 6-year-old daughter Emilie at home. I want to thank all my friends who encourage me all the time so that I never feel isolated during the lockdown. I want to thank for the support from Technische Universität München to allow me to work from home safely. I also want to thank the medical and nursing staff fighting against this virus on the front line and the scientists working on the vaccine days and nights to bring our normal life back.

Abstract

Nowadays, we have a growing number of mature applications in the field of natural language processing (NLP), especially natural language understanding (NLU) and generation (NLG), like chatbots or auto-generated reports. Such applications relieve users from repeatable works and assist them in achieving high-demanding yields. We have different programming languages in the software development domain, which require deep understanding by human beings. Suppose we can apply methods for natural language processing in the software development world. In that case, we could help both programmers with good programming skills and project managers or data scientists, who need to understand code but do not have a strong programming background, to do their job more convenient, by generating documents to make the code easier to read and understand, generating code difference descriptions to compare and evaluate codes quickly, or generating code structure or dependencies to make programming more manageable.

In recent years, transfer learning is becoming quite successful. This machine learning method pre-trains a model first on a large amount of unlabeled data with an unsupervised task, then fine-tunes the same model on smaller labeled datasets. In this thesis, we examined the effect of transfer learning for tasks in the software development domain. We compared transfer learning with single-task learning and multi-task learning on thirteen tasks involving nine programming languages. We used the transformer encoder-decoder architecture to develop different sizes of models with these training strategies. We call these models CodeTrans. Our CodeTrans models outperform all the state-of-the-art models for all the tasks.

The pre-trained models generated by transfer learning could be applied to the tasks in the software development domain. Fine-tuning these models on new tasks would save a lot of training steps and time. Therefore, we published our CodeTrans pre-trained and fine-tuned models online so that everyone can use these models to generate text for relevant tasks or to fine-tune new tasks freely.

Kurzfassung

Heutzutage gibt es immer mehr Anwendungen im Bereich Natural Language Processing (NLP), insbesondere Natural Language Understanding (NLU) und Generierung (NLG), wie Chatbots oder automatisch generierte Reports. Solche Anwendungen entlasten den Benutzer von repetitiven Arbeiten und unterstützen ihn beim lösen anspruchsvoller Probleme. In der Softwareentwicklung benutzt man zur Problemlösung verschiedene Programmiersprachen, die ein tiefes Verständnis des Entwicklers erfordern. Hier können wir Methoden des Natural Language Processing in der Softwareentwicklung anwenden.

Wir können Dokumentationen für Programmier-Funktionen generieren, um den Code leichter lesbar und verständlich zu machen oder die Unterschiede von verschiedenen Code-Versionen zusammenfassen, um Code schnell zu vergleichen und zu bewerten. Außerdem kann auch der Programmcode selbst generiert werden. Auf diese Weise würden wir Programmierern, Projektmanagern oder Data Scientists bei der Entwicklung neuer Software helfen, egal ob sie einen ausgeprägten Programmier-Hintergrund haben oder nicht.

In den letzten Jahren hat sich das Transfer-Learning sehr erfolgreich entwickelt. Bei dieser Methode des maschinellen Lernens wird ein Modell zunächst auf einer großen Menge unannotierter Daten mit Hilfe eines unsupervised Tasks erst vor-trainiert, und dann das gleiche Modell auf kleineren annotierten Datensätzen weiter fine abstimmt. In dieser Masterarbeit untersuchten wir den Effekt des Transfer-Learnings auf Aufgaben im Bereich der Softwareentwicklung. Wir vergleichen Transfer-Learning mit Single-Task-Learning und Multi-Task-Learning bei dreizehn Aufgaben mit neun Programmiersprachen. Wir verwendeten die Transformer-Encoder-Decoder-Architektur, um verschiedene Größen von Modellen mit diesen Trainingsmethoden zu entwickeln. Wir nennen unsere Modelle CodeTrans. Unsere CodeTrans-Modelle haben für alle Aufgaben bessere Ergebnisse als State-of-the-Art-Modelle erzielt.

Die durch Transfer-Learning erstellten vor-trainierten Modelle können auf andere Aufgaben in der Software-Entwicklungs-Domäne angewendet werden. Die Weiter-Finetuning dieser Modelle auf neue Aufgaben würde eine Menge Trainingsschritte und Zeit sparen. Daher haben wir unsere vor-trainierten und fein-abgestimmten CodeTrans-Modelle online veröffentlicht, so dass jeder diese Modelle kostenlos und frei benutzen kann, um Texte für relevante Aufgaben zu generieren oder um neue Aufgaben fein-abzustimmen.

Contents

Acknowledgments **iii**

Abstract **iv**

Kurzfassung **v**

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Problem Statement . . . 2
- 1.3. Research Questions . . . 2
- 1.4. Thesis Contribution . . . 2
- 1.5. Research Approach . . . 3
- 1.6. Structure of the Thesis . . . 4

2. Background **5**
- 2.1. Software Engineering and Development . . . 5
- 2.2. Natural Language Processing . . . 6
  - 2.2.1. Deep Learning in Natural Language Processing . . . 6
  - 2.2.2. Transformer Model and Attention Mechanism . . . 10
  - 2.2.3. Transfer Learning . . . 11
  - 2.2.4. Multitask Learning . . . 14
- 2.3. Natural Language Processing in Software Development . . . 16

3. Related Work **21**
- 3.1. Tasks Related Work . . . 21
  - 3.1.1. Code Documentation Generation . . . 21
  - 3.1.2. Source Code Summarization . . . 23
  - 3.1.3. Code Comment Generation . . . 24
  - 3.1.4. Git Commit Message Generation . . . 25
  - 3.1.5. API Sequence Recommendation . . . 27
  - 3.1.6. Program Synthesis . . . 28
- 3.2. Model Related Work . . . 29
  - 3.2.1. Text-to-Text Transfer Transformer . . . 30

4. Approach 33
4.1. Datasets . . . 33
4.1.1. Unsupervised Dataset . . . 33
4.1.2. Supervised Dataset . . . 39
4.2. Vocabulary . . . 45
4.3. Model Architecture . . . 47
4.4. Evaluation Metrics . . . 49
4.4.1. BLEU score . . . 49
4.4.2. ROUGE score . . . 50
4.4.3. Accuracy . . . 50

5. Experiment 52
5.1. Experimental Setup . . . 52
5.1.1. Hardware . . . 52
5.1.2. Software Usage . . . 53
5.2. Single-task Learning . . . 54
5.3. Transfer Learning . . . 56
5.3.1. Pre-training . . . 56
5.3.2. Fine-tuning . . . 57
5.4. Multi-task Learning . . . 60
5.5. Multi-task Learning with Fine-tuning . . . 63

6. Evaluation Results and Discussion 65
6.1. Evaluation Results . . . 65
6.1.1. Code Documentation Generation . . . 65
6.1.2. Source Code Summarization . . . 69
6.1.3. Code Comment Generation . . . 71
6.1.4. Git Commit Message Generation . . . 71
6.1.5. API Sequence Recommendation . . . 72
6.1.6. Program Synthesis . . . 72
6.2. Discussion . . . 74
6.3. Models Publication . . . 77

7. Conclusions and Future Work 79
7.1. Conclusions . . . 79
7.2. Future Work . . . 80

A. Appendix - Hyperparameter Turning and Evaluation on Validation Set 81
A.1. Single-task Learning . . . 81
A.1.1. Code Documentation Generation . . . 81
A.1.2. Source Code Summarization . . . 86
A.1.3. Code Comment Generation . . . 88
A.1.4. Git Commit Message Generation . . . 89

A.1.5. API Sequence Generation 90
A.1.6. Program Synthesis 91
A.2. Transfer Learning 92
A.3. Multi-task Learning 97
A.4. Multi-task Learning with Fine-tuning 100

B. Appendix - Model Output Examples for each Task 105
B.1. Code Documentation Generation 105
B.1.1. Code Documentation Generation - Python 105
B.1.2. Code Documentation Generation - Java 105
B.1.3. Code Documentation Generation - Go 106
B.1.4. Code Documentation Generation - Php 106
B.1.5. Code Documentation Generation - Ruby 106
B.1.6. Code Documentation Generation - Javascript 107
B.2. Source Code Summarization 107
B.2.1. Source Code Summarization - Python 107
B.2.2. Source Code Summarization - CSharp 107
B.2.3. Source Code Summarization - SQL 108
B.3. Code Comment Generation 108
B.4. Git Commit Message Generation 108
B.5. API Sequence Generation 109
B.6. Program Synthesis 109

List of Figures 110

List of Tables 112

Bibliography 113

1. Introduction

1.1. Motivation

Software development can be considered as a process of designing, implementing, testing, and maintaining information systems such as applications, frameworks, or other software components[1]. It plays an inevitable role in today's society. No matter in which industry, every global company requires a solid software system to assist their business nowadays. However, software development is a very complicated and expensive process. These companies all need teams full of software experts to support and maintain their software systems. At the same time, experienced specialists in the software development domain try to invent and use different tools and methods (for example, design patterns, code documentation, unit tests, version control tools, etc.) to control and improve the software quality and make the software developing process more effective and convenient.

In software development, works are done by using different programming languages. Programming language can be considered a kind of language used to communicate with the computer systems for achieving the requirements in software development. While looking into the machine learning domain, we would notice the significant progresses achieved by natural language processing in recent years. A growing number of mature natural language processing applications, especially natural language understanding (NLU) and generation (NLG) applications, are becoming more widespread in the industry world. For example, chatbots for personalized customer communication, analytical intelligence dashboard, or auto-generated reports and summarizations for transforming data into insightful text or vice versa, such applications relieve users from repeatable works and help them concentrate on more high-demanding tasks.

There is a trend now to apply natural language processing techniques to programming languages to make the developing tools more helpful and the developing process smoother for developers. The improved developing tools and methods could also help the non-developing-experts like project managers or data scientists, who need to understand code deeply in work but not have a strong programming background, to do their jobs more efficiently.

Currently, in machine learning, especially the natural language processing world, transfer learning helps more and more models achieve the best results on benchmarks. Transfer learning allows the model to be fine-tuned for different kinds of downstream tasks in NLP with relatively small task-relevant datasets, and makes it much easier and faster to get good results on personalized tasks with a low computational cost.

Therefore, this master thesis focuses on applying natural language processing techniques in the software development domain. A large number of experiments are carried out. Different software development tasks with various datasets are explored using the transfer learning method in this thesis.

1.2. Problem Statement

We want to examine which deep learning models and training methods would reach the best results in the software development domain. Single-task training, transfer learning and multitask learning are involved and mixed during the experiment process. From small size to large size models are included in the training on a large number of datasets. Consequently, a lot of computational power will be consumed during the limited thesis working period. Therefore, to make our experiment more efficient, we use one Nvidia GPU[2] and multiple Google TPUs[3] for training.

Besides, the datasets involved in the experiment contain software code of different programming languages, which is different from standard natural language text. The different datasets also vary a lot in terms of size, source, and format. Such differences may also influence the experiments and the results a bit. Considering this aspect, we used different parsers and tokenizers for different programming languages to preprocess the data and saved them into the same format. We observe and compare different model performances based on different dataset in this thesis.

1.3. Research Questions

This thesis aims to investigate the following three main research questions:

- What kind of natural language processing models would work best for tasks in the software development domain?
- How would transfer learning improve the performance comparing with only training on the labeled data alone?
- Would transfer learning perform better than multi-task learning for the same tasks?

1.4. Thesis Contribution

In the course of this thesis, the following four main contributions are made:

Applying of different deep learning technologies on various tasks in the software development domain: In this thesis, we used different deep learning methods like single-task training, transfer learning, and multitask learning. The datasets we used cover nine programming languages, including Python, Java, Javascript, Php, Go, Ruby, SQL, Csharp, and Lisp. Our experiment contains in total of thirteen tasks in six categories.

Achievement of outstanding results on tasks in the software development domain: Our models outperform the state-of-the-art models for all the tasks by comparing the evaluation metrics.

Providing the transfer learning models which can be used for fine-tuning other tasks: Transfer learning is useful because the pre-trained models can be continued to fine-tune other similar tasks. However, pre-training would take most of the time and requires extensive data with high-demand hardware. Therefore, we provide our pre-trained transfer learning models for download. Other users can use it to fine-tune their tasks on datasets in the software development domain.

Online user interface for all the models: We also created the github repository[1] for this thesis. To allow users to use our models and generate results, we published our models to the Hugging Face Model Hub[2] with the built-in user interface and APIs. In addition, we chose the form of Google Colab[3] notebook to create another user interface, including preprocessing methods and best models from each training technology for each task in the github repository.

1.5. Research Approach

The following steps are taken to study the research questions, reach good performances, and contribute to the natural language processing in the software development domain.

1. Literature Review: In this stage, we reviewed literature about natural language processing and its different technologies, software development, and natural language processing tasks in the software development domain based on the research of Severini[4].

2. Defining Models: Text-to-Text Transfer Transformer (T5)[5] is the latest transformer model. This model is very suitable for transfer learning and multitasks learning. It also outperformed a lot of natural language processing tasks. Therefore, we chose to use T5 to carry out our experiment.

3. Defining Tasks: We adapted tasks from the experiment of Severini[4]. Furthermore, we added new tasks about natural language processing in software development from the latest paper we found during the literature review step.

[1]`https://github.com/agemagician/CodeTrans`
[2]`https://huggingface.co/SEBIS`
[3]`https://colab.research.google.com/notebooks`

4. Pre-processing Dataset: We used different parsers for different programming languages to parse and tokenize the code if the data is the programming language. For the natural language, we examined and removed the not English data. Then we pre-processed the data into the TSV format accepted by our models.
5. Training: We trained the tasks using single-task learning, transfer learning, as well as multi-task learning and fine-tuning. We used the small, base, and large models to train the tasks. We stopped training based on the performance of models on the validation set by early stopping, and selected the best checkpoints.
6. Evaluation: In this step, we evaluated the performance of the best checkpoints on the test set. We compared the results of different deep learning technologies taking the model size and dataset size into account.
7. Publishing the Models and Generalization for User Interface: After evaluating the models and getting the best checkpoints, we uploaded and published the models in GitHub and Hugging Face Model Hub. We also built the Colab notebooks, including input, pre-processing, model computation, and output as another user interface. In this way, everyone can access our models and use them to generate text outputs or fine-tune similar tasks.

1.6. Structure of the Thesis

This thesis is divided into seven chapters. In addition to the introduction in this chapter, background knowledge is introduced in Chapter 2, including the definition of software engineering, the development of natural language processing and deep learning in natural language processing, the features of different NLP models and technologies, as well as the current situation of natural language processing in software development. In Chapter 3, task-related and model-related works are introduced. Chapter 4 explains our experiment's approaches, including the dataset details, model architecture, and evaluation metrics. The experiment set-ups and processes are illustrated in Chapter 5. The evaluation results are compared and discussed in Section 6. The conclusion of this thesis and future improvements are drawn in Chapter 7.

2. Background

In the following sections, an overview of the background knowledge of this thesis is given.

2.1. Software Engineering and Development

The term *software* was being coined in 1958 by the famous statistician John Tukey in his paper "The Teaching of Concrete Mathematics[6]". Nowadays, there are millions of software professionals worldwide working in the software engineering and development field. However, generating high-quality software is not an easy thing. In the guide to software engineering[7], IEEE proposed ten knowledge areas for creating a software:

- Software requirements: How to correctly detect and discover users' requirements and record them clearly and preciously.
- Software design: How to design and organize the software architecture to meet the requirements and verify the defined models.
- Software construction: During the life circle of coding, verification, unit testing, integration testing, and debugging, how to minimize the complexity, anticipate changes, construct for verification, and apply standards.
- Software testing: How to use different techniques and measures to evaluate product quality and identifying defects or problems for improvement.
- Software maintenance: How to provide cost-effective post-implementation support to software.
- Software configuration management: How to identify and control the software configuration, account for the configuration status, audit the configuration, and manage the software release and delivery.
- Software engineering management: How to plan, coordinate, measure, monitor, control, and report - ensure that the development and maintenance of software are systematic, disciplined, and quantified.
- Software engineering process: How to manage the definition, implementation, assessment, measurement, management, change, and improvement of the software life cycle processes.

- Software engineering tools and methods: How to choose software development tools or software engineering methods to support and assist the software life circle processes.
- Software quality: How to achieve software quality using static and dynamic techniques.

All these knowledge areas are highly connected. For example, in every knowledge field (like design or construction), tools and methods could be very meaningful and improve software quality. Tasks in this thesis may be more directly relevant to the software construction but could implicitly influence software maintenance, software engineering tools and methods, and software quality.

2.2. Natural Language Processing

"Natural Language Processing is a theoretically motivated range of computational techniques for analyzing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like language processing for a range of tasks or applications."[8] It aims to let computer systems understand the natural languages and finally achieve processing language tasks like a human. Knowledge about computational linguistics, computer science, and cognitive psychology is required in the natural language processing process.

The first research project in natural language processing can be traced back to the late 1940s when the earliest machine translation project was launched to break the enemy codes using computer translation during World War II. Nowadays, NLP's main applications can be considered as Information Retrieval, Information Extraction, Question-Answering, Summarization, Machine Translation, and Dialogue Systems[8].

Moreover, the techniques applied in natural language processing have also frequently developed. Semantic approaches based on the relationship of concepts in the language, or rule-based systems assisted by regular expression has dominated this field for a long time in the beginning. In the 1980's, statistical approaches, especially TF-IDF[9], gained importance and was widely applied in NLP. Thanks to the development of the internet and computational powers, it is very convenient to collect a vast amount of data from the internet. Therefore, neural networks, especially deep learning, which requires a large amount of data for training and many computations, have become the center of attention in recent years. Different models based on deep learning break the records and achieve the highest score in various natural language processing tasks.

2.2.1. Deep Learning in Natural Language Processing

As stated in the book "Deep Learning in Natural Language Processing", deep neural networks are capable of learning representations from language data, using a cascade of multiple layers with nonlinear processing units to extract features[10]. These deep learning architectures

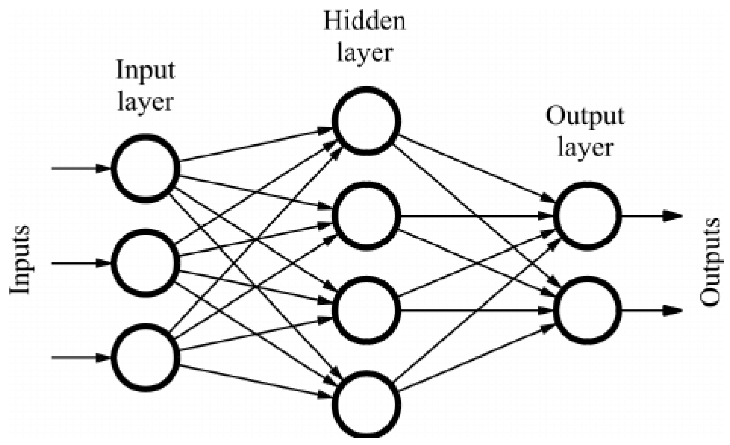


Figure 2.1.: The architecture of a single-layer feed-forward neural network[11].

can extract both lower-level features and higher-level features, and gain sufficient knowledge from these features.

A feed-forward neural network is the first and simplest type of artificial neural network[13]. It is composed of one input layer, one output layer, and the hidden layers. When there are many hidden layers in this neural network, this network is called the deep neural network. Each layer has a different number of nodes called neurons. Each neuron's value is computed by the neuron's value in the former layer with the weights connecting them and the activation function. The activation function of the neurons enables the network to learn the non-linear features. The loss function after the output layer calculated the differences between the model- and reference-output. The differences are backpropagated to the network to update the weights and improve the network performance. However, the feed-forward neural network does not have the ability to extract the features from sequence data like the language. Therefore, deep learning developed its specific architectures for processing language data in the natural language processing field.

One important architecture for acquiring the representation of words is the Word2Vec model[14]. As shown in Figure 2.2, it uses an unsupervised way to gain the meaning of words from sentences and uses embeddings to display these words with their relationships. The pre-trained model of Word2Vec shows that the embedding vectors gained from the text can form the mathematical equation as $Vector("King") - Vector("Man") + Vector("Woman") = Vector("Queen")$[15]. These embeddings are quite useful for information retrieval or information extraction tasks like sentiment classification.

When considering understanding the sentences or paragraphs, we need to take the context into account as humans. The previous words or the previous sentences could significantly influence the meaning of the current sentence or paragraph. So it is important that the neural network could also remember and consider the previous context when dealing with the natural language processing tasks. For this purpose, the Recurrent Neural Network (RNN)[16] and the Long Short Term Memory Neural Network (LSTM)[17] are invented. As shown in

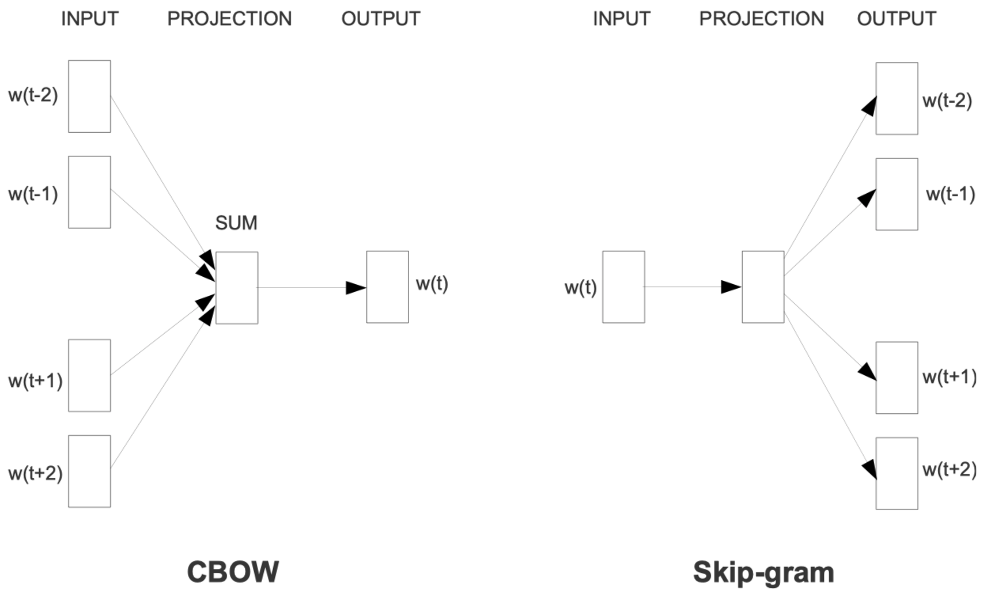
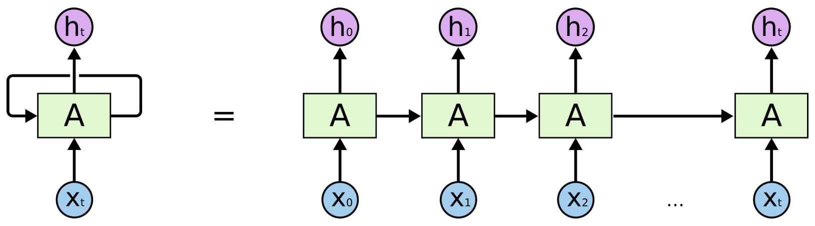
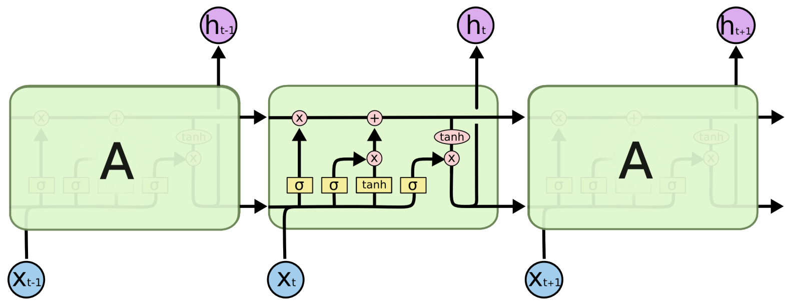


Figure 2.2.: The word2vec models architecture. "The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.[12]" The weight matrix gives the vector presentation of words.



(a) An unrolled recurrent neural network.



(b) Different functional ingredients in LSTM

Figure 2.3.: The architecture and details of model RNN and LSTM[1].

Figure 2.3a, a recurrent neural network loops over each input, extracts representations from the former input, keeps the information, and passes it to the next cell when processing the later input. LSTM is a special form of the recurrent neural network which can decide to *remember* or *forget* the long dependencies using different computational gates (Figure 2.3b) while RNN cannot drop the long redundant dependencies.

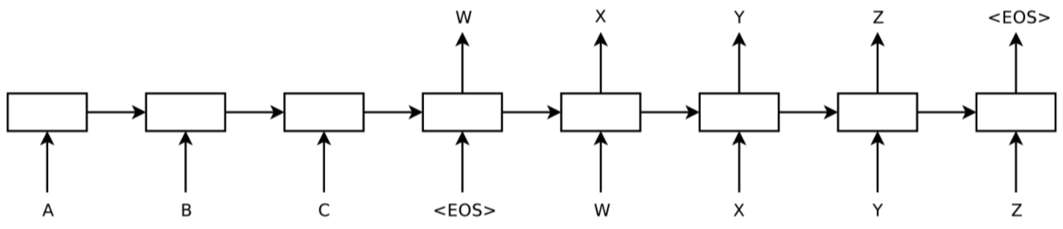


Figure 2.4.: The Sequence to Sequence model reads the input "ABC", and produces the output "WXYZ"[18].

Machine Translation is a kind of task that generates natural language as the final output. It requires not only the ability to extract the features from the input information but also the ability to process and interpret the features back to the natural language. Because of the high requirement of the input and output interpretation, machine translation tasks benefit most from deep learning.

[1] `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

The Encoder-Decoder Sequence to Sequence model[18] has the most impact on this kind of tasks. It uses the LSTM[19] to understand the input and obtain a vector representation and uses the RNN language model[20] to extract the output sequence from that vector (Figure 2.4). It is also worth mentioning that the neural network language models encode the input tokens by 1 out of N encoding, where N is the corpus vocabulary size. It estimates output probabilities based on the whole vocabulary history and produces the normalized output probability values using a softmax activation function[19].

2.2.2. Transformer Model and Attention Mechanism

The Sequence to Sequence model has some drawbacks. It relies on recurrent layers that have high computational complexity and hard to be parallelized for computation. The transformer model gets rid of the recurrent layers completely. It relies entirely on the attention mechanism to achieve global dependencies between input and output[21].

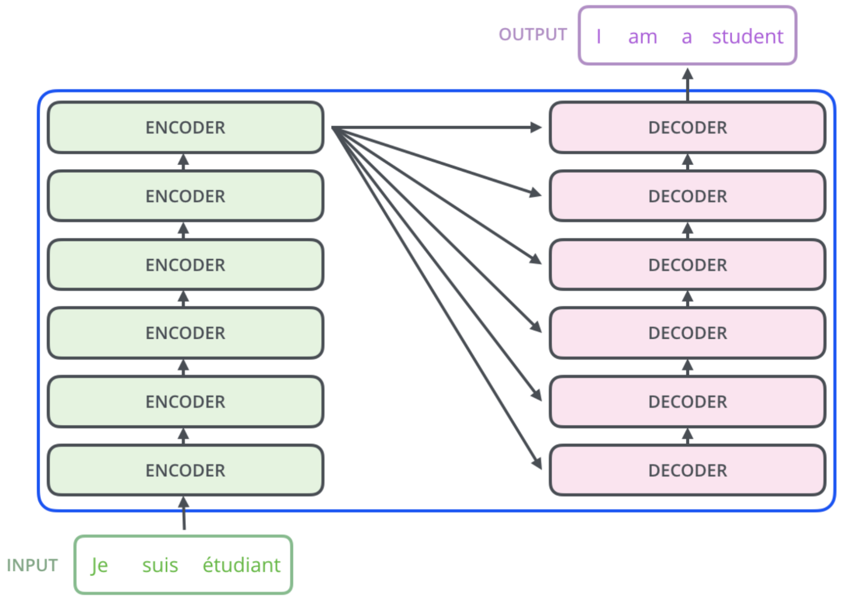


Figure 2.5.: The encoder-decoder transformer architecture[2].

As shown in Figure 2.5, the transformer also has an encoder-decoder structure. The encoder lies on the left part of the figure and is consist of N=6 identical layers. Each identical layer contains a multi-head self-attention layer followed by a simple feed-forward network. The decoder of the right part of the figure also contains six identical layers. However, every identical layer has two attention layers and one feed-forward layer. The additional attention layer performs multi-head attention over the output of the encoder part. Figure 2.6 illustrates the multi-head self-attention layer's computational steps with the head number of eight.

[2]`http://jalammar.github.io/illustrated-transformer/`

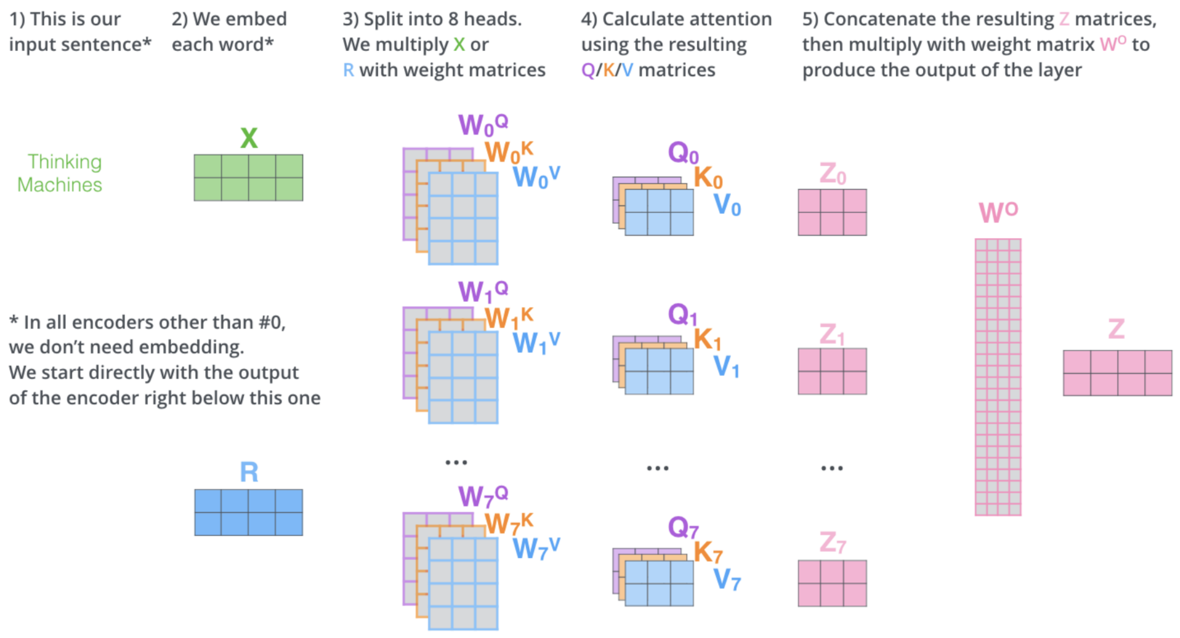

1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W0Q W0K W0V
W1Q W1K W1V
W7Q W7K W7V
Q0 K0 V0
Q1 K1 V1
Q7 K7 V7
Z0
Z1
Z7
WO
Z

Figure 2.6.: The computational steps of the multi-head self-attention layer[3].

Among these parameters, all the weight metrics presented by W are randomly initialized and updated during the training. Multi-head attention allows the model to focus on different sentence positions and gains multiple representations for each sentence's subspace.

Since the transformer is composed of feed-forwarding layers and self-attention layers, the computation can be parallelized easily. Moreover, computing self-attention layers is also faster than computing recurrent layers. It has shorter paths for forward and backward signals to traverse in the network, which improves the ability to learn long-range dependencies in the network.

2.2.3. Transfer Learning

Training a deep neural network requires a large amount of data. However, data could be outdated from time to time, and the data distribution may change along with society's development. In such a situation, the already-trained model may not perform well anymore. So it is necessary to recollect the data and retrain the model from scratch, which would be very costly to prepare the data and train the model with extra computational power, money and time.

Because it is common for humans to transfer the knowledge they learned from experience to solve new problems. It is also promising that neural networks can similarly apply the

[3]`http://jalammar.github.io/illustrated-transformer/`

knowledge they have gained from the previous training tasks to the new tasks in a new domain. Besides, when observing the first few layers of deep neural network gained when learning the images, the features are not specific to the dataset or task, but very general and basic like sharps or colors, which could also be used to continue to learn images in other tasks or domains[22]. Therefore, such a way of transferring knowledge becomes a beneficial neural network technology called transfer learning.

Transfer learning is divided into three types[23]:

- Inductive transfer learning: The target task is different from the source task. Some data in the target domain should be labeled. The data in the source domain could be labeled or not. So the target domain data are required to induce the knowledge learned in the source task for the target task.
- Transductive transfer learning: The source and target tasks are the same. However, the source and target domains are different. For example, feature spaces or the distribution of data in the source and target domains are different. So the knowledge about the skills dealing such kind of task should be learned.
- Unsupervised transfer learning: The target task is different from the source task. Nevertheless, no labeled data are available in either source or target domain. So this type of transfer learning focuses on unsupervised tasks like clustering or dimensional reduction.

Pan et al.[23] also summarized four ways to carry out the transfer learning:

1. Instance-based transfer learning approach: It assumes part of the source data suggests a similar or adjustable distribution as the target data. After training the source task, if we reweigh the source data, we can get the optimal result for the target data in the target task.
2. Feature-representation-transfer approach: The knowledge learned during pre-training in the source domain is encoded into the feature representation to the target task's input. Using the new feature representation generated by the pre-training model as input could improve the fine-tuning tasks' performance.
3. Parameter-transfer approach: Some parameters or features (weights) or prior distributions of the hyperparameters could be shared among the source and target tasks. Moreover, there are further options to use and freeze the parameters or update them during fine-tuning. Although the improvement of performance depends on the distance between the source and target tasks, transferring even the distant tasks can improve the performance better than the random parameters[22].
4. Relational knowledge-transfer problem: It assumes that the data are not independent and identically distributed random variables. However, the relationship between the source data and target data is similar, which can be transferred from source data to target data.

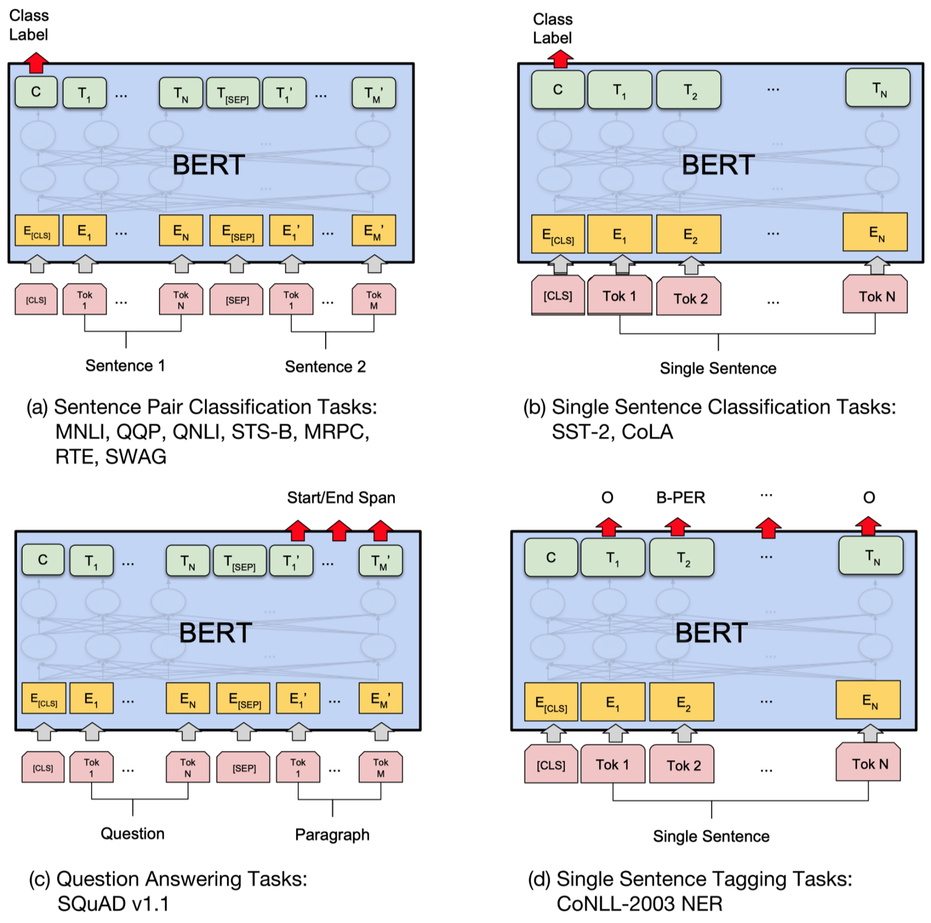

Class Label
BERT
Sentence 1
Sentence 2
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG
Class Label
Single Sentence
(b) Single Sentence Classification Tasks: SST-2, CoLA
Start/End Span
Question
Paragraph
(c) Question Answering Tasks: SQuAD v1.1
O
B-PER
O
Single Sentence
(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

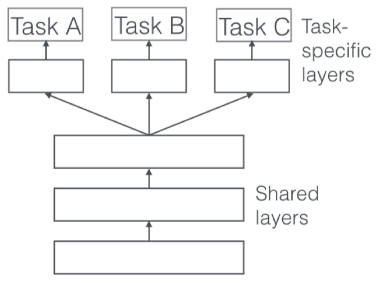
Figure 2.7.: Bert model with one additional output layer for fine-tuning tasks. (a) and (b) are sequence-level tasks, and (c) and (d) are token-level tasks. [24]

At the end of 2018, Jacob Devlin et al. from google AI Language published a bidirectional transformer with the attention mechanism for language modeling — Bert[24], which obtained new state-of-the-art results on eleven natural language processing tasks at that time. Bert's architecture has a multi-layer bidirectional Transformer encoder based on the Transformer model[21] we introduced in Section 2.2.2. It used two unsupervised tasks (Masked LM and Next Sentence Prediction) with BooksCorpus[25] and English Wikipedia dataset for pre-training the model. An additional output layer is added to the model for fine-tuning the downstream natural language processing supervised tasks, as shown in Figure 2.7. This single additional layer also ensures that a minimum number of parameters need to be learned from scratch again, which reduces the cost during fine-tuning.

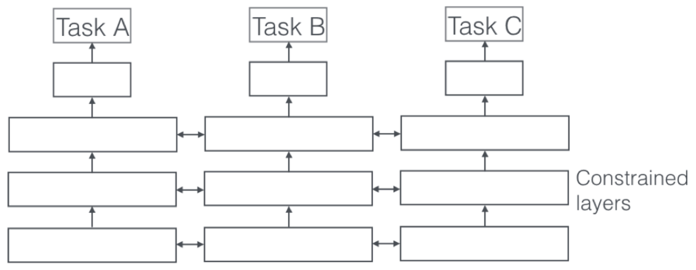
Since then, different pre-training language models like ALBERT[26], RoBERTa[27], Distil-Bert[28] and XLNet[29] have been published in the natural language processing field. The architectures of these models are based on the Transformer, and they were optimized and outperformed Bert. In addition to their good results on the benchmarks, these models can also be fine-tuned for different kinds of downstream NLP tasks with relatively small task-relevant datasets as transfer learning. Such models' good performance and convenience make the transfer learning a trend in the natural language processing world. With the help of this technology, it is much easier and faster to get good results on personalized tasks with a low computational cost.

2.2.4. Multitask Learning

"Multitask learning is an approach to inductive transfer. Inductive transfer can help improve generalization by using the domain information contained in the training signals of related tasks as an inductive bias."[30] Similar to inductive transfer learning, they both use inductive transfer mechanisms to improve generalization performance. However, multitask learning tries to learn both the source task and the target task simultaneously, while inductive transfer learning aims to transfer the knowledge learned from the source task to the target task[23]. During multitask learning, the model could gain the essential information shared by several tasks efficiently. Moreover, multiple tasks' task-specific knowledge being gathered at the same time would lead to the inductive bias, which causes the model to try to explain each task more general and avoid over-fitting.



(a) Hard parameter sharing



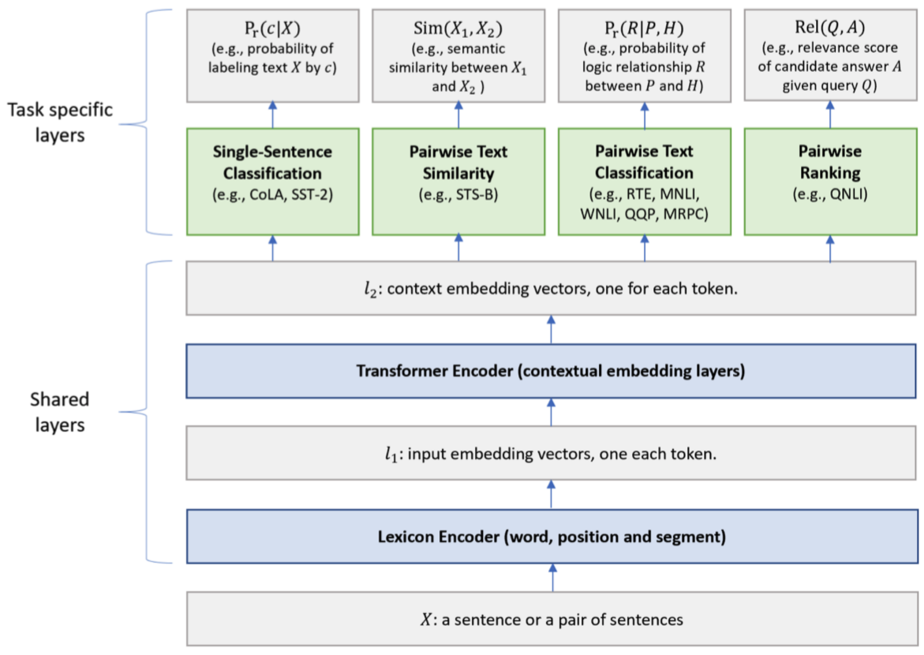
(b) Soft parameter sharing

Figure 2.8.: Two different multitask learning methods for deep learning[31].

In general, there are two types of multitask learning in deep neural networks: hard and soft parameter sharing of hidden layers[31]. Figure 2.8a illustrates the hard parameter sharing. All the tasks share the first several hidden layers and their parameters, while the output layers are designed separated for different tasks. During the sharing, the model tries to find the representation capturing all the tasks, avoid being too specific for any single task or data, and reduce the chance of over-fitting. In the soft parameter sharing, each task has its own hidden layers and output layers. However, the hidden layers are being connected to each other so that distance among the hidden layers' parameters is being regularized. In this way, soft parameter sharing has a regularization effect on the model's weights and avoids over-fitting on each single task.

There are five reasons for the good performance of multitask learning[31]:

1. Implicit data augmentation: Multitask learning collects several tasks, and implicitly increases the amount of training data with different noise patterns. This helps the model learn a general feature representation for all the data and average the noise patterns.
2. Attention focusing: Different tasks will help the model focus on the features that are important for all the tasks. So when a task dataset is very noisy or limited, or high-dimensional, other datasets will guide the model to find the meaningful features of that nonoptimal dataset.
3. Eavesdropping: Some features in one task could be hard to learn, while its representation in another task is more obvious. When a model learns these two tasks simultaneously, they could share the information and help each other to discover the very unclear hidden features.
4. Representation bias: The model will be biased to learn the representations which are preferred by most of the tasks. This bias also helps the model to learn other tasks more efficiently if these tasks are similar.
5. Regularization: Different tasks force the model to find the best weights for all the tasks, avoid the model to learn the feature representation of a single training dataset, and reduce the risk of over-fitting a single task.



$P_r(c|X)$ (e.g., probability of labeling text X by c)
$\mathrm{Sim}(X_1, X_2)$ (e.g., semantic similarity between X_1 and X_2)
$P_r(R|P,H)$ (e.g., probability of logic relationship R between P and H)
$\mathrm{Rel}(Q,A)$ (e.g., relevance score of candidate answer A given query Q)
Task specific layers
Single-Sentence Classification (e.g., CoLA, SST-2)
Pairwise Text Similarity (e.g., STS-B)
Pairwise Text Classification (e.g., RTE, MNLI, WNLI, QQP, MRPC)
Pairwise Ranking (e.g., QNLI)
l_2: context embedding vectors, one for each token.
Transformer Encoder (contextual embedding layers)
Shared layers
l_1: input embedding vectors, one each token.
Lexicon Encoder (word, position and segment)
X: a sentence or a pair of sentences

Figure 2.9.: The architecture of Multi-Task Deep Neural Network (MT-DNN) model[32]

In 2019, Liu et al. from Microsoft Research proposed a model called Multi-Task Deep Neural Network (MT-DNN)[32]. The model is based on the Bert large model[24]. Its architecture consists of the transformer encoder with task-specific output layers as shown in Figure 2.9. The training procedure contains the pre-training stage of Bert and multitask learning. After using multitask learning and fine-tuning, MT-DNN obtains new state-of-the-art results on ten Natural Language Understanding tasks. MT-DNN model could be considered as a combination of transfer learning and multitask learning. Moreover, its better performance than Bert also proves the effectiveness of multitask learning in natural language processing.

2.3. Natural Language Processing in Software Development

Software engineering, especially software development, is a way to generate applications or automatic tools with a high complexity. According to the waterfall model, software engineering has a life circle of 1) system and software requirements, 2) analysis, 3) software design, 4) coding, 5) testing, and 6) operation. Various processes with tools for assistance are invented, like version control or type check, to make software engineering phases easier and keep the code quality high. On the one hand, these processes and tools help make the software development process much easier to manage and control. On the other hand, they increase the complexity of software engineering and the workload when requiring filling the contents manually.

With the development of machine learning techniques, more and more mature machine learning, especially deep learning applications, are being applied in daily life, like different recommendation systems or object recognition systems. These applications usually are trained on large amounts of data. Simultaneously, more and more open- or closed-source code repositories are available online, which provides the condition to use machine learning to automate the software engineering tasks.

We use natural language to communicate with each other. We use the programming language to communicate with computer systems during software development. Natural language and programming language both share the word *language*. So natural language processing techniques are very promising to solve the tasks in the software development domain.

When looking into the history of natural language processing in software development, a syntactic parser is commonly used to understand the SVO structure of natural language, like a who-does-what structure or a subject-verb-object structure. After that, semantic patterns of syntactic correspondences are applied to generate the object-oriented code skeletons. Furthermore, the discovered statements are converted to the programming language statements under the rule-based settings. Mihalcea et al. used this approach to generate programming language code based on the programming assignment[33]. Stenmark et al. also produced code instructions for industrial robots using the semantic and syntactic parser[34].

Deep learning becomes the trend in machine learning since 2010. Both technology and hardware levels ensure that the training of neural networks with many layers of non-linear hidden units could be competent and proficient[36]. However, the massive applications or researches using deep learning in software engineering happen after 2015. Li et al. did

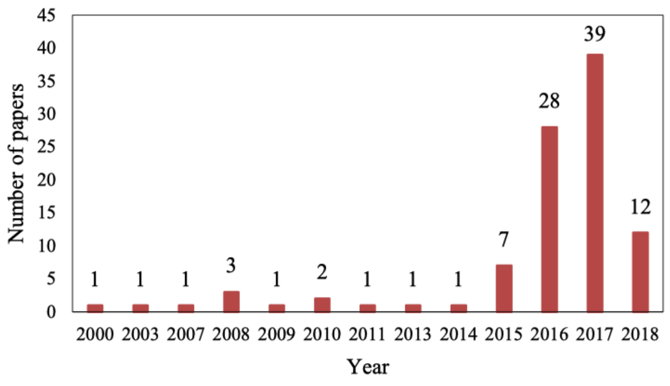
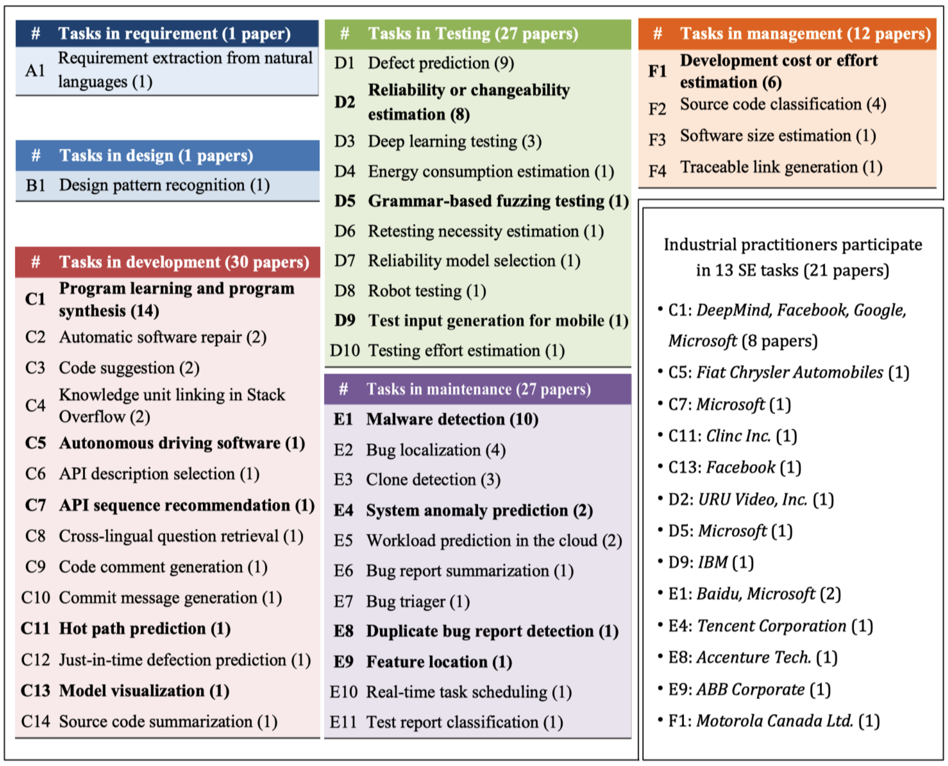


Figure 2.10.: The numbers of publications about deep learning in software engineering and development domain per year[35].



#	Tasks in requirement (1 paper)
A1	Requirement extraction from natural languages (1)

#	Tasks in design (1 papers)
B1	Design pattern recognition (1)

#	Tasks in development (30 papers)
C1	**Program learning and program synthesis (14)**
C2	Automatic software repair (2)
C3	Code suggestion (2)
C4	Knowledge unit linking in Stack Overflow (2)
C5	**Autonomous driving software (1)**
C6	API description selection (1)
C7	**API sequence recommendation (1)**
C8	Cross-lingual question retrieval (1)
C9	Code comment generation (1)
C10	Commit message generation (1)
C11	**Hot path prediction (1)**
C12	Just-in-time defection prediction (1)
C13	**Model visualization (1)**
C14	Source code summarization (1)

#	Tasks in Testing (27 papers)
D1	Defect prediction (9)
D2	**Reliability or changeability estimation (8)**
D3	Deep learning testing (3)
D4	Energy consumption estimation (1)
D5	**Grammar-based fuzzing testing (1)**
D6	Retesting necessity estimation (1)
D7	Reliability model selection (1)
D8	Robot testing (1)
D9	**Test input generation for mobile (1)**
D10	Testing effort estimation (1)

#	Tasks in maintenance (27 papers)
E1	**Malware detection (10)**
E2	Bug localization (4)
E3	Clone detection (3)
E4	**System anomaly prediction (2)**
E5	Workload prediction in the cloud (2)
E6	Bug report summarization (1)
E7	Bug triager (1)
E8	**Duplicate bug report detection (1)**
E9	**Feature location (1)**
E10	Real-time task scheduling (1)
E11	Test report classification (1)

#	Tasks in management (12 papers)
F1	**Development cost or effort estimation (6)**
F2	Source code classification (4)
F3	Software size estimation (1)
F4	Traceable link generation (1)

Industrial practitioners participate in 13 SE tasks (21 papers)

- C1: *DeepMind, Facebook, Google, Microsoft* (8 papers)
- C5: *Fiat Chrysler Automobiles* (1)
- C7: *Microsoft* (1)
- C11: *Clinc Inc.* (1)
- C13: *Facebook* (1)
- D2: *URU Video, Inc.* (1)
- D5: *Microsoft* (1)
- D9: *IBM* (1)
- E1: *Baidu, Microsoft* (2)
- E4: *Tencent Corporation* (1)
- E8: *Accenture Tech.* (1)
- E9: *ABB Corporate* (1)
- F1: *Motorola Canada Ltd.* (1)

Figure 2.11.: Software engineering tasks are categorized into six software engineering steps. The right lower part with white background listed the tasks participated by industry practitioners.[35].

a research on publications about deep learning in software engineering from 2000 to the first quarter of 2018[35]. Figure 2.10 categories these 98 related research papers into the publication year. This figure shows that a maximum of three papers about deep learning in software engineering was published yearly before 2015. Nevertheless, there were already twelve relevant papers in the first three months of 2018.

The authors also grouped these 98 papers into six software engineering steps as Figure 2.11. We may notice, most of the software engineering tasks lie in development (with 30 papers), testing (with 27 papers), and maintenance steps (with 27 papers). They also listed 21 papers in 13 software engineering tasks involving industrial practitioners, including Google, Facebook, Microsoft, and DeepMind. This indicates that the industry's interest for deep learning in software engineering will push the relevant task-study from research to the production level.

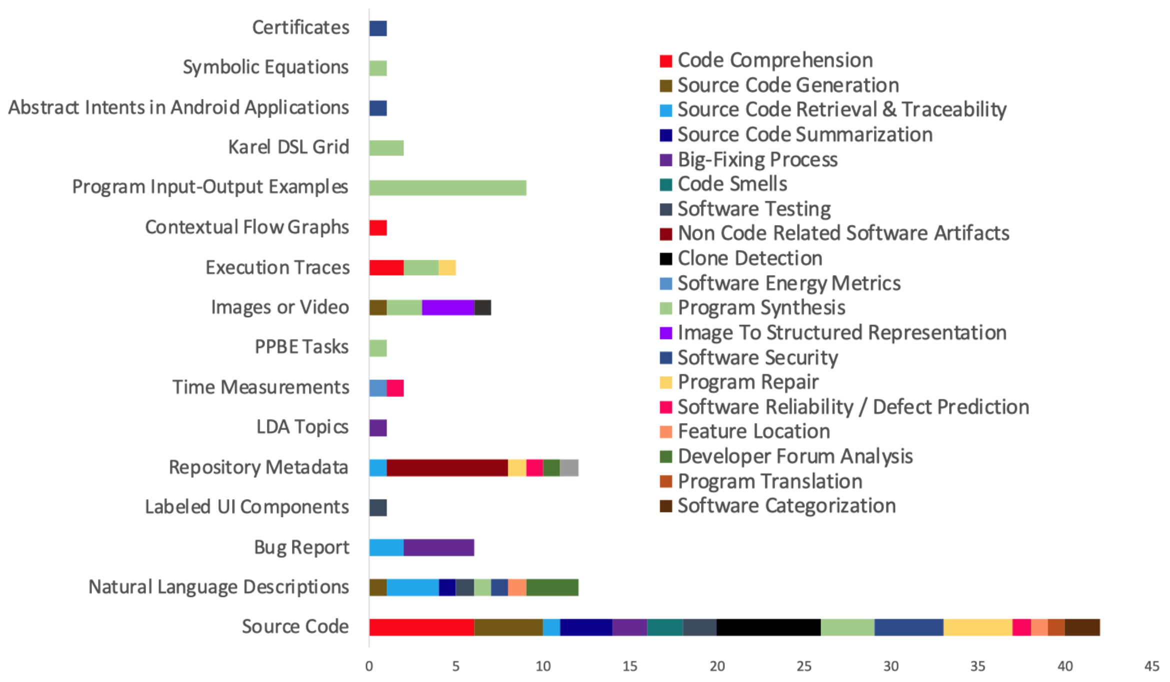


Figure 2.12.: Data type used in software engineering tasks[37]. The y-axis lists different type of data. The x-axis indicates the number of tasks using each data type. The color bars illustrates different software engineering tasks.

When analyzing the tasks and the type of data for deep learning in software engineering, Cody Allen Watson[37] researched 84 related studies with 111 SE tasks from 2009 to 2019, while it is confirmed that there was little work between 2009 to 2014. From Figure 2.12 we can observe that the most common type of data being used is source code. Source code is used in 49 out of 111 tasks from the relevant studies. One reason for the popularity of source code is the plenty of code repositories in Github, Gitlab, or code snippets in StackOverflow that can be downloaded freely. Common tasks like program comprehension, code generation, description or summarization, and source code testing all need to be carried out based on the source code. Other primary data types for software engineering tasks are natural language descriptions, repository metadata, input-output examples, and visual data. All these types comprised 78.57% of the distribution in the data type.

The most popular task of deep learning in software engineering is Program Synthesis, which is being researched in 20 papers out of the 84 related studies. It follows by Code Comprehension, and Source Code Retrieval and Traceability. The author also listed some topics which could be beneficial but still unexplored or underrepresented. Such topics include refactoring and program analysis, software systems and mobile testing, software documentation, feature location, and defect prediction. Because the problem itself is not well defined, or no current architecture is suitable for such tasks or to process the available data[37].

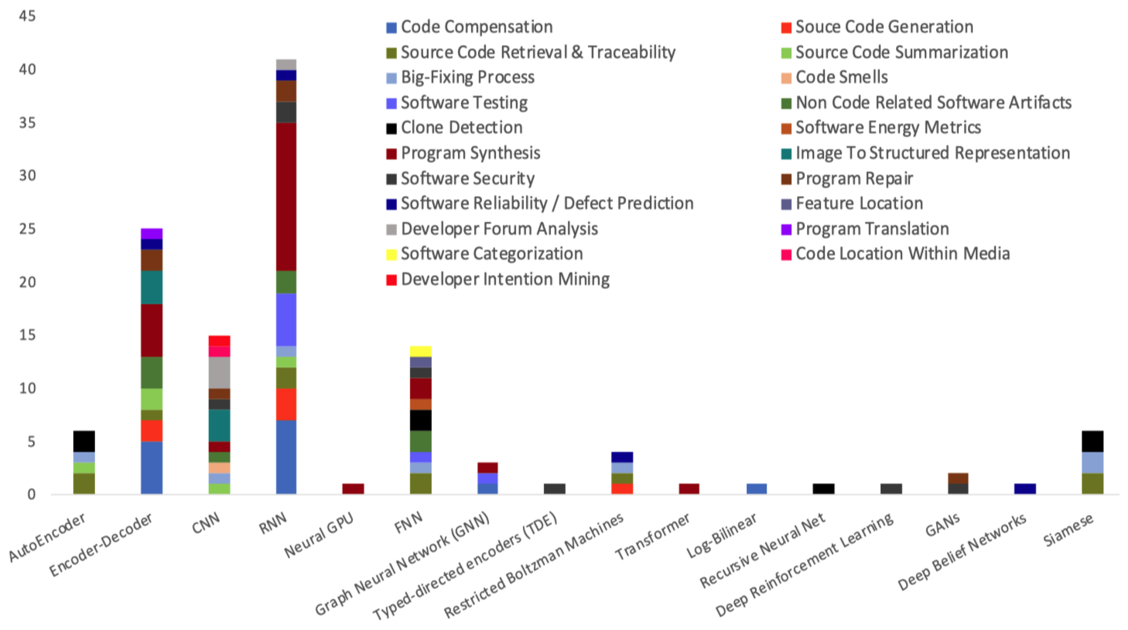


Figure 2.13.: Deep learning architectures used by software engineering tasks[37]. The y-axis lists the number of tasks. The x-axis represents the different deep learning architectures. The color bars illustrates different SE tasks.

Figure 2.13 shows the distribution of deep learning architectures applied in the software engineering tasks. It is obvious that the recurrent neural network (RNN) is the most widely used deep learning model architecture in SE. The Encoder-Decoder Sequence to Sequence model, which generates a latent representation by the encoder from the input, and for the decoder to understand and decode the representation to target, ranked directly after RNN. As we have introduced in Section 2.2.1, recurrent neural network and Encoder-Decoder Sequence to Sequence model have the ability to extract the sequential features of the data, and the most common data type in SE tasks is source code. Source code has many features that are embedded in its sequential nature. So RNN and Encoder-Decoder would perform well on such tasks, which explains this architecture distribution. In addition, when the software engineering tasks include image or media data type, like the task image to structured representation, convolutional neural network (CNN) is being applied. Because CNN has good performance for extracting features in the images.

The transformer architecture is being applied to only one Program Synthesis task among the studies researched by Cody Allen Watson[37]. The occasional use of transformer is because he collected the papers till the year 2019, when transformer architecture is relatively new at that time. Later, transformer and other transformer based models outperform RNN and Sequence to Sequence architectures in a variety of natural language processing tasks. It is worth applying the transformer architecture to more and more different natural language processing tasks in software engineering.

In 2020, Facebook AI Research published a model called Transcoder for the Program Translation task[38]. This model translates functions between C++, Java, and Python based on the transformer architecture. They downloaded the GitHub public repositories having C++, Java, and Python files and broke down the files into function level code. Then they used three unsupervised methods of machine translation to train the model: 1) cross-lingual masked language model pre-training to build the language model using the masked pre-training, 2) denoising auto-encoding to train the decoder always to generate valid sequences regardless of the noisy input data, and 3) the back translation to let the model generate target programming language from the source programming language, and to translate the target language back to the source language. They evaluated a dataset composed of parallel functions in C++, Java, and Python from the online platform GeeksforGeeks[4]. Their model outperformed the rule-based and commercial baselines[5,6] significantly using computational accuracy by evaluating the code functions' output when given the same input to the reference and generated code.

Meanwhile, Microsoft Research also proposed their pre-trained models for programming and natural languages called CodeBert. We go into the details of this publication in Section 3.1.1.

[4]`https://practice.geeksforgeeks.org`

[5]j2py: `https://github.com/natural/java2python`

[6]Tangible Software Solutions: `https://www.tangiblesoftwaresolutions.com/`

3. Related Work

In this chapter, we introduce the tasks related work and model related work about this thesis.

3.1. Tasks Related Work

This thesis focuses on natural language processing tasks in software development. We include six main tasks as follows, containing a total of thirteen subtasks:

- Code Documentation Generation
- Source Code Summarization
- Code Comment Generation
- Git Commit Message Generation
- API Sequence Recommendation
- Program Synthesis

These six main tasks together with the original models to solve these tasks are explained in this section.

3.1.1. Code Documentation Generation

Making documentation for code is very important in software development. On the one hand, writing documentation costs time and energy of programmers. It also occurs quite often that documentation is no more up-to-date after changing the code, which may cause misunderstandings. On the other hand, well-written documentation would help the reader understand the code's details quickly. Good documentation could also reduce the cost of project maintenance and updating processes. So it is necessary if an AI system could automatically generate the documentation for different programming language codes, save time, and provide insight into the code.

Feng et al. from Microsoft Research Center Asia published CodeBERT[39] - a pre-trained model for programming and natural languages, which presents the state-of-the-art performance for the Code Documentation Generation task. Their multi-layer bidirectional Transformer model followed Bert[24] and used the same model architecture as RoBERTa-base[27] with 125M total number of model parameters.

They used CodeSearchNet Corpus Collection[40] as their pre-training data. This corpus is collected from the publicly available open-source non-fork GitHub repositories. All the selected repositories are used by at least one other project and have the license that permits the re-distribution of parts of the project. Six programming languages, including Go, Java, JavaScript, Python, PHP, and Ruby, are involved in the corpus. Part of the corpus has functions or methods with their documentation, which contains more than three tokens. The rest data are functions/methods without documentations. These functions are longer than three lines. These function whose name contain the substring "test" is removed. Furthermore, the documentations are truncated to the first full paragraph.

Pre-training

They have two unsupervised pre-training tasks, masked language modeling (MLM) and replaced token detection (RTD). For the masked language modeling, 15% of the tokens from the natural language and programming language pairs are randomly selected and masked out. During the pre-training, the model needs to predict the original tokens, which are masked out. For the replaced token detection task, tokens in some random position of natural language documentations or programming language codes are replaced by plausible alternatives. In this case, the model needs to determine whether the token in each position is original or replaced. So the model learns to solve a binary classification problem.

Fine-tuning for Code Documentation Generation

After pre-training the model, the authors added a transformer[21] with 6 layers, 768-dimensional hidden states, and 12 attention heads as the decoder for this model to generate the text. Then they fine-tuned this model on six programming languages of the CodeSearch-Net Corpus separately to generate the documentation for the programming language code. The max input code length is set as 256, and the max output text length is 64. They used Adam optimizer with a learning rate of 5e-5 and the batch size 64. Early stopping is applied when tuning hyperparameters on the development set.

Evaluation

The authors used a smoothed BLEU score[41] to evaluate the CodeBERT with its decoder for the Code Documentation Generation task. They compared their encoder model with three other models, including the RNN-based Sequence to Sequence model[18], the Transformer, and the RoBERTa. They also compared the CodeBERT model using different pre-training ways like pre-training on code only, with RTD task only or MLM task only or with both RTD task and MLM task. As shown in Figure 3.1, CodeBERT pre-trained with RTD and MLM

MODEL	RUBY	JAVASCRIPT	GO	PYTHON	JAVA	PHP	OVERALL
SEQ2SEQ	9.64	10.21	13.98	15.93	15.09	21.08	14.32
TRANSFORMER	11.18	11.59	16.38	15.81	16.26	22.12	15.56
ROBERTA	11.17	11.90	17.72	18.14	16.47	24.02	16.57
PRE-TRAIN W/ CODE ONLY	11.91	13.99	17.78	18.58	17.50	24.34	17.35
CODEBERT (RTD)	11.42	13.27	17.53	18.29	17.35	24.10	17.00
CODEBERT (MLM)	11.57	14.41	17.78	18.77	17.38	24.85	17.46
CODEBERT (RTD+MLM)	**12.16**	**14.90**	**18.07**	**19.06**	**17.65**	**25.16**	**17.83**

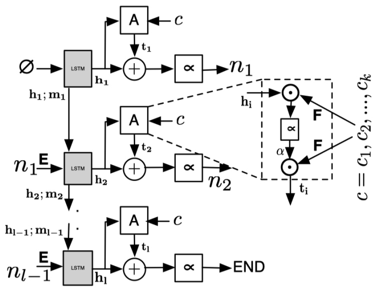
Figure 3.1.: The evaluation result of codeBERT for Code Documentation Generation tasks[39]

achieved the state-of-the-art performance for the Code Documentation Generation tasks on CodeSearchNet Corpus.

3.1.2. Source Code Summarization

Unlike Code Documentation Generation, the Source Code Summarization task tries to summarize the code not only at the function or method level but also at the code snippet level. Code appearing on the internet like online forums is mostly in the form of code snippet. Such code snippet is usually the critical part of a function, helping users exchange knowledge with each other. So it is necessary to gain information from the code at the code snippet level and summarize the code so that users can search and understand code more efficiently.

Based on the questions and code answers from the popular programming help website StackOverflow[1], Iyre et al. proposed the model CODE-NN[42] to summarize SQL and CSharp code snippets. The CODE-NN architecture consisted of LSTM guided by a global attention model[43] to compute a weighted sum of the embeddings of the code snippet tokens based on the current LSTM state as shown in Figure 3.2a.



(a) The CODE-NN architecture illustrates the relationship of LSTM and the Attention model

	Model	METEOR	BLEU-4
C#	IR	7.9 (6.1)	13.7 (12.6)
	MOSES	9.1 (9.7)	11.6 (11.5)
	SUM-NN	10.6 (10.3)	19.3 (18.2)
	CODE-NN	**12.3 (13.4)**	**20.5 (20.4)**
SQL	IR	6.3 (8.0)	13.5 (13.0)
	MOSES	8.3 (9.7)	15.4 (15.9)
	SUM-NN	6.4 (8.7)	13.3 (14.2)
	CODE-NN	**10.9 (14.0)**	**18.4 (17.0)**

(b) Evaluation result on CSharp and SQL human-annotated development and test dataset. Performance on the development set is indicated in parentheses.

Figure 3.2.: CODE-NN Architecture and the evaluation result[42].

[1]`http://stackoverflow.com`

For creating the dataset, they downloaded anonymized versions of posts having tags of SQL and CSharp, containing a short title, a detailed question, and one or more responses, of which one can be marked as accepted. Furthermore, they selected only the title and the code snippet from accepted answers that contain exactly one code snippet. They also removed the data whose title has no relation to the code snippet, parsed the code, and replaced the context-specific literals with tokens denoting their types. After splitting the data into training, validation, and test sets, they asked human annotators to provide two additional titles for 200 randomly chosen code snippets from the validation and test set. As a result, each code snippet from the human-annotated dataset has three titles as the golden references.

The authors used supervised training with mini-batch stochastic gradient descent and back-propagation. They also applied dropout and learning rate decay for training. For decoding, the beam search with the beam size of 10 was chosen. The maximal summary length was set as 20 words.

METEOR[44] and the smoothed BLEU-4[41] score were used for evaluation. The authors only evaluated on the human-annotated data set. They also compared their model with three other models, including an information retrieval baseline, the phrase-based machine translation system MOSES[45], and the neural attention-based abstractive summarization model SUM-NN[46]. Figure 3.2b shows the evaluation results on these two metrics. Moreover, five English native speakers rated the output summarizations in terms of the naturalness. Additional five human evaluators familiar with SQL and CSharp evaluated the generated titles for informativeness on a scale between 1 and 5. As a result, CODE-NN outperformed all the other methods across all the metrics and achieved state-of-art performance.

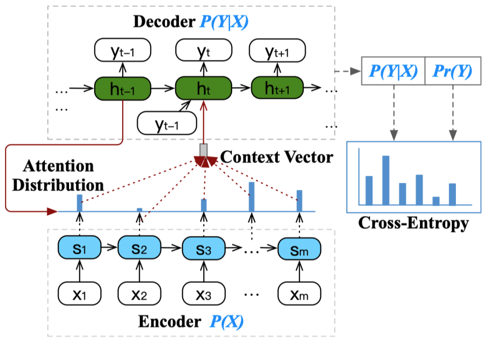
3.1.3. Code Comment Generation

Similar to Code Documentation Generation, Code Comment Generation also focuses on automatically generating code comments to help developers save time on understanding the functionality of programming methods.

Hu et al. published the DeepCom[47] model for this task. They focused on Javadoc comments extracted by Eclipse's JDT compiler[2] from 9,714 Java open source projects from Github. They considered the Javadoc description's first sentence as the comment. Because following the Javadoc guidance[3], the first sentence describes the functionality of Java methods most. They used a Sequence to Sequence[18] model consisting of an Encoder, an Attention[48], and a Decoder Component to learn the Java code and generate comments. The encoder and decoder are both LSTMs[19]. Figure 3.3a illustrates model architecture.

One highlight of their work is how they took advantage of the *structured* code feature. Unlike the natural language text, programming languages are formal languages that are unambiguous, structured, and contains strong logic. They did not input the source code as

[2]`http://www.eclipse.org/jdt/`
[3]`http://www.oracle.com/technetwork/articles/java/index-137868.html`



(a) The DeepCom Sequence-to-Sequence model.

Approaches	BLEU-4 score (%)
CODE-NN	25.30
Seq2Seq	34.87
Attention-based Seq2Seq	35.50
DeepCom (Pre-order)	36.01
DeepCom (SBT)	**38.17**

(b) Evaluation results on Java methods.

Figure 3.3.: DeepCom Architecture and the evaluation result[47].

plain text directly into the model. Instead, they first used Eclipse's JDT compiler to convert the Java methods into Abstract Syntax Tree sequences. They then proposed a Structure-based Traversal (SBT) method to traverse the AST to generate the final sequence as the model input to use structure information of source code. Figure 3.4 shows how the Java code AST is converted to the DeepCom input sequence by SBT. Figure 3.4a lists the original Java code. The left part of Figure 3.4b is the AST of the code and the right part of that figure is the converted sequence by SBT.

They used the smoothed BLEU-4[41] score as the evaluation metrics. CODE-NN[42], which we introduced in Section 3.1.2, was used as the baseline model for this task. Moreover, they compared DeepCom with a basic Sequence to Sequence model having unprocessed source code as input, an attention-based Sequence to Sequence model using also the unprocessed source code, and a DeepCom with a classical pre-order traversal method to process the code for input. Figure 3.3b shows that DeepCom with the SBT traversal method outperformed the other four models and methods. This result proved the effectiveness of both the DeepCom architecture and the SBT preprocessing method.

3.1.4. Git Commit Message Generation

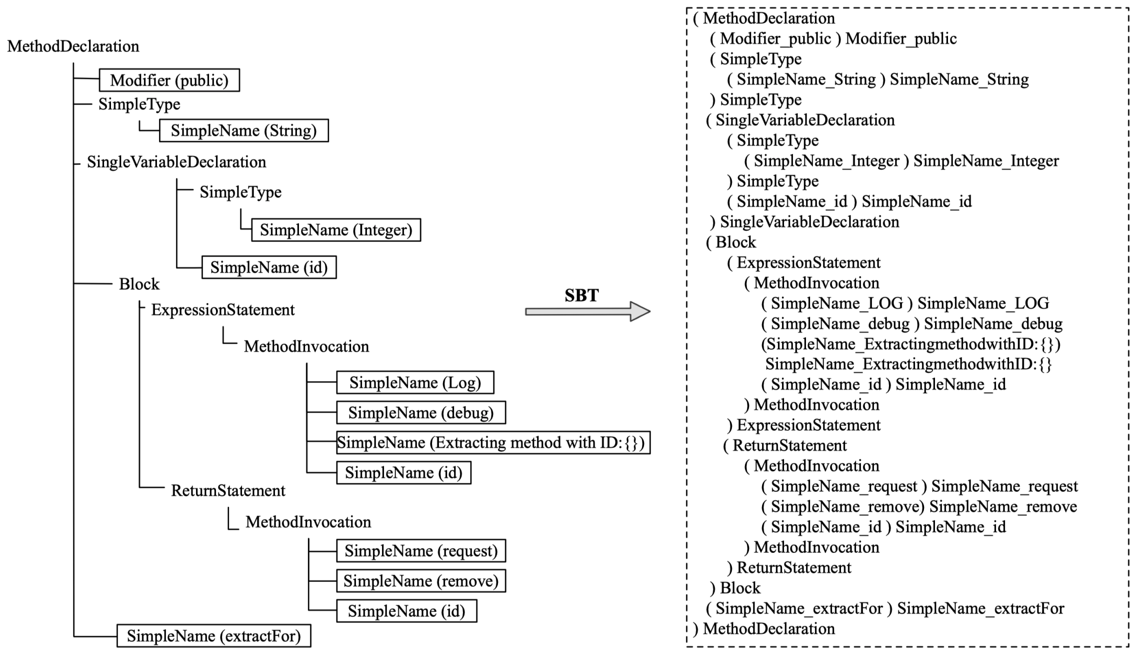
The development-assisted tool Git[4] is a version-control system for tracking changes in files and codes during software development. Each git change contains the differences between the current and previous versions of attached files and a commit message that summarizes the change content and describes this change's purpose. A well-structured code commit helps to overview the project development and control the code changes and the development quality.

Jiang et al. developed the Neural Machine Translation (NMT)[49] model to generate commit-messages from git change diffs automatically. Their model architecture is composed of an

[4]`https://git-scm.com/`

```java
public String extractFor(Integer id){
  LOG.debug("Extracting method with ID:{}", id);
  return requests.remove(id);
}
```

(a) The Java code example.



```
( MethodDeclaration
  ( Modifier_public ) Modifier_public
  ( SimpleType
    ( SimpleName_String ) SimpleName_String
  ) SimpleType
 ( SingleVariableDeclaration
    ( SimpleType
      ( SimpleName_Integer ) SimpleName_Integer
    ) SimpleType
    ( SimpleName_id ) SimpleName_id
  ) SingleVariableDeclaration
 ( Block
    ( ExpressionStatement
      ( MethodInvocation
        ( SimpleName_LOG ) SimpleName_LOG
        ( SimpleName_debug ) SimpleName_debug
        (SimpleName_ExtractingmethodwithID:{})
         SimpleName_ExtractingmethodwithID:{}
        ( SimpleName_id ) SimpleName_id
      ) MethodInvocation
    ) ExpressionStatement
   ( ReturnStatement
      ( MethodInvocation
        ( SimpleName_request ) SimpleName_request
        ( SimpleName_remove) SimpleName_remove
        ( SimpleName_id ) SimpleName_id
      ) MethodInvocation
    ) ReturnStatement
  ) Block
 ( SimpleName_extractFor ) SimpleName_extractFor
) MethodDeclaration
```

(b) Converting the Java code AST to input sequence by SBT.

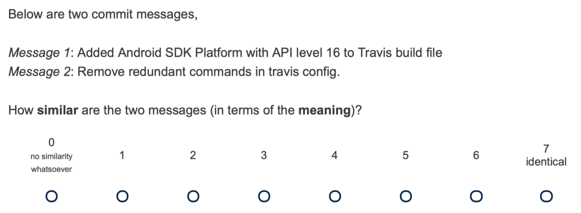
Figure 3.4.: DeepCom SBT method example.

Encoder, an Attention, and a Decoder component[50]. The Encoder had two RNNs[20]: a forward and a backward RNN. They read the source sequence with less than 100 tokens in the forward and the reversed order and generated two hidden state sequences. The Attention component took the concatenation of these two hidden state sequences. Furthermore, the RNN-Decoder computed the hidden state to text sequence with a maximum of 30 tokens.

Based on a dataset with 1000 Java repositories having most Github stars, they extracted more than 2 million commits from GitHub. They kept only the first sentences from the commit message since the first sentence typically summarizes the entire commit message. Then they removed issue ids from the sentences and the commit ids from the diffs. They dropped merge, and rollback commits since these messages do not contain too much summary information. They also removed diffs that are larger than 1 MB. Then they implemented the Verb-Direct Object filter to select sentences having a verb/direct-object pattern. Finally, they trained on these selected commits with the mini batch-size of 80.

Model	BLEU	Len_{Gen}	Len_{Ref}	p_1	p_2	p_3	p_4
MOSES	3.63	129889	22872	8.3	3.6	2.7	2.1
NMT1	31.92	24344	22872	38.1	31.1	29.5	29.7
NMT2	32.81	21287	22872	40.1	34.0	33.4	34.3
	23.10*	20303	18658	30.2	23.3	20.7	19.6

(a) The model evaluation score results on the test set. NMT1 is the NMT model with V-DO filter. NMT2 is a model trained without V-DO filter. Len_{Gen} is the total length of the generated messages. Len_{Ref} is the total length of the reference messages.


Below are two commit messages,

Message 1: Added Android SDK Platform with API level 16 to Travis build file
Message 2: Remove redundant commands in travis config.

How **similar** are the two messages (in terms of the **meaning**)?

(b) One of the evaluation page for participants to score the similarity. The scala is from 0 to 7.

Figure 3.5.: NMT models evaluation results by metrics and one human evaluation example.[49]

The authors evaluated the model using the BLEU[51] score that having the modified n-gram precision. They used MOSES[45] model as the baseline. To test the Verb-Direct Object filter's effectiveness, they also trained NMT on unfiltered data and evaluated results on test data with and without Verb-Direct Object filter. From the Figure 3.5a, we can observe that NMT models performed much better than the MOSES baseline. NMT without Verb-Direct Object filter outperformed the model with Verb-Direct Object filter. This is because the model had 2.5 times more training data when the filter was not being applied.

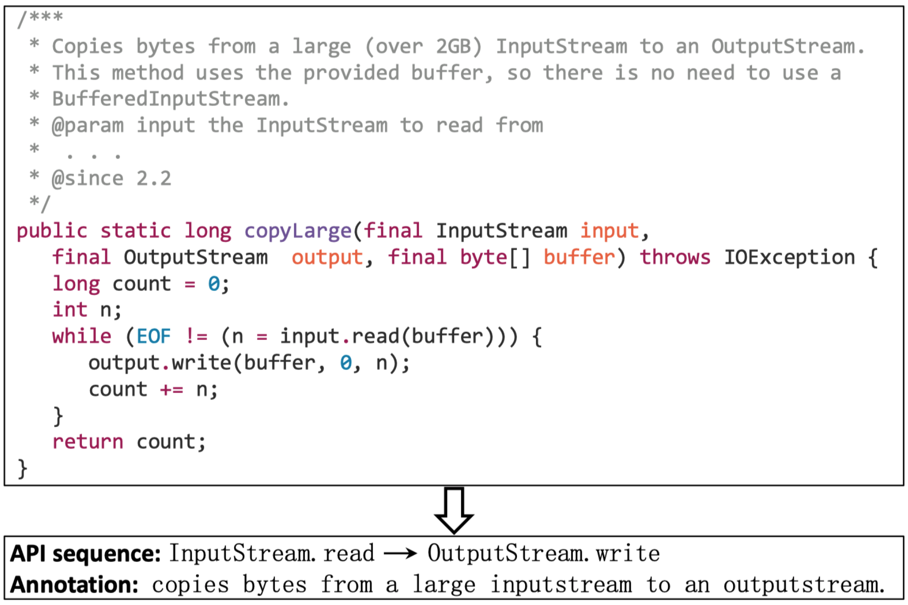
Moreover, the authors also carried out human evaluation. Unlike the way CODE-NN[42] used to generate more reference data, they hired 20 participants with programming experience for 30 minutes to evaluate the similarity of the generated message with the original message in a survey study. Figure 3.5b showed one of the evaluation questions participants needed to answer.

3.1.5. API Sequence Recommendation

Developers usually learn new libraries or software frameworks through understanding how to use their APIs. However, it is challenging to obtain the API usage sequence if the usage patterns are not well documented. So it would be beneficial to suggest developers the API sequence when searching and asking about the corresponding usages.

Gu et al. proposed the model DeepAPI[52] that generated API usage sequences for a given natural language query. Their model architecture was an Attention-based Encoder-Decoder model[53]. The Encoder and Decoder were GRUs[53] with 1000 hidden units. The dimension of word embedding was 120. They applied their own defined negative log-likelihood as the cost function. They used minibatch Adadelta[54] with a batch size of 200 to train the model and applied the Beam Search[55] to generate output.

By creating the dataset, they downloaded Java projects with at least one star from Github.



```java
/***
 * Copies bytes from a large (over 2GB) InputStream to an OutputStream.
 * This method uses the provided buffer, so there is no need to use a
 * BufferedInputStream.
 * @param input the InputStream to read from
 *  . . .
 * @since 2.2
 */
public static long copyLarge(final InputStream input,
   final OutputStream  output, final byte[] buffer) throws IOException {
   long count = 0;
   int n;
   while (EOF != (n = input.read(buffer))) {
      output.write(buffer, 0, n);
      count += n;
   }
   return count;
}
```

API sequence: InputStream.read ⟶ OutputStream.write
Annotation: copies bytes from a large inputstream to an outputstream.

Figure 3.6.: An example for extracting an API sequence and its query from a Java method.

Then they used the Eclipse JDT compiler to parse the source code files into Abstract Syntax Trees and split the JavaDoc comments and the code. They extracted the first sentence of a documentation comment for a method since the first sentence can summarize a method. After excluding the irregular comments like those starting with "TODO," they considered these comments as the code query. For getting the API sequences, they traversed the AST of code and applied several replacements like replacing *new C()* as the API *C.new* to the API sequence. Figure 3.6 showed an example for extracting an API sequence and its query from a Java method.

They evaluated DeepAPI using the BLEU[51] score and compared it with the other models with totally different Architecture - Lucene+UP-Miner[56] and SWIM[57]. Besides, they also compared DeepAPI with a pure RNN architecture and another attention-based encode decoder without the specifically designed cost function. DeepAPI outperformed all other models and reached the BLEU score of 54.52%.

3.1.6. Program Synthesis

Program synthesis is the task of synthesizing or generate programming codes based on the users' commands. It would be beneficial to reduce the workload of programmers. However, this task is very challenging because the natural language commands could be very ambiguous, and the generated programs should meet high requirements and have satisfying functionality.

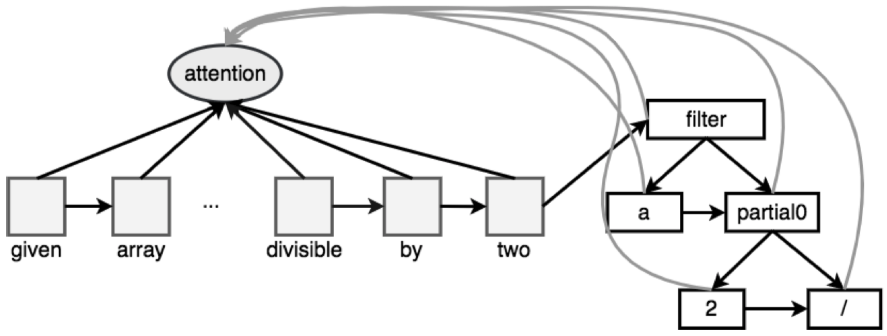


Figure 3.7.: The architecture of Seq2Tree encoder-decorder model.

Polosukhin and Skidanov proposed the Seq2Tree[58] model for this task. They used a sequence encoder and a tree decoder to synthesize LISP-inspired domain-specific language (DSL). The encoder used GRU[53] cell. The decoder used a doubly-recurrent neural network for generating AST tree-structured output. Attention component was also applied to augment the current step with information from the encoder. Figure 3.7 illustrates the model architecture. Moreover, they used Tree-Beam search to control the output length and select the best-generated program.

The authors also built a dataset AlgoLisp[59] focusing on LISP-inspired DSL. They chose tasks from homework assignments for basic computer science and algorithms courses. Since the number of tasks is limited, they then modified and combined assignments and the corresponding code to generate more similar tasks. For example, they generated a new task, "find all *odd* elements in an array," based on "find all *even* elements in an array."

For evaluation, they also implemented ten tests for each task. Since the same problem can be solved by the programs written differently, they judged a solution as correct if the solution passed all the tests for the given assignment. In this way, they used accuracy to evaluate the model output and defined accuracy as:

$$Acc = \frac{N_c}{N}$$

where N_c is the number of tasks passing tests, and N is the number of total tasks. They compared Seq2Tree with an Attentional Sequence to Sequence model[43] and an IO2Seq[60] model. They also made a difference among the models with and without Beam Search. Seq2Tree outperformed all other models and achieved the state-of-the-art for this LISP-inspired DSL program synthesis task.

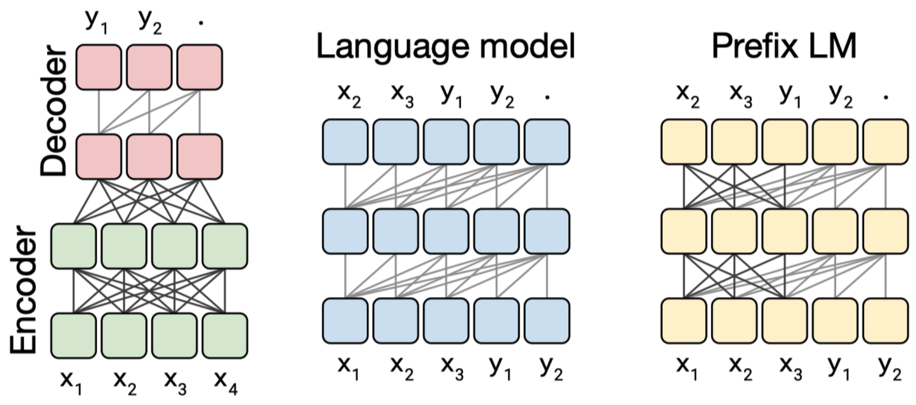
3.2. Model Related Work

In this section, we introduce the model we used for this thesis.

3.2.1. Text-to-Text Transfer Transformer

Pre-training a language model and then fine-tuning on the downstream tasks has proven its effectiveness on natural language processing tasks in recent years. Nowadays, there are many unlabeled text data on the Internet, which can be used for the unsupervised pre-training. So, this method is particularly friendly to fine-tune downstream tasks with datasets having only a small amount of labeled data.

Raffel et al. from Google proposed a model called Text-to-Text Transfer Transformer (T5)[5] for this scenario. The architecture of T5 is an encoder-decoder Transformer closely following its originally-proposed form as we introduced in Section 2.2.2[21]. They built an unsupervised pre-training clean and natural English large dataset "Colossal Clean Crawled Corpus[5]" based on the Common Crawl's web extracted text. Then they applied their model to the various set of downstream tasks, including machine translation, question answering, abstractive summarization, and text classification. They also explored a variety of pre-training and fine-tuning technologies to gain insights and achieve state-of-the-art in many of the tasks.



y_1 y_2 .
Decoder
Encoder
x_1 x_2 x_3 x_4
Language model
x_2 x_3 y_1 y_2 .
x_1 x_2 x_3 y_1 y_2
Prefix LM
x_2 x_3 y_1 y_2 .
x_1 x_2 x_3 y_1 y_2

Figure 3.8.: Different transformer architectures. Dark grey lines means the fully-visible masking and light grey lines means the causal masking. The left one is used in the base model.

Their baseline transformer model consists of 12 blocks. Each block comprises self-attention, optional encoder-decoder attention, and a feed-forward network with a dropout probability of 0.1. This model has about 220 million parameters. They compared their baseline model with Language model architecture[61], prefix LM architecture[62], and the encoder-decoder with reduced layers, parameters, and denoising objective. Figure 3.8 illustrated these Transformer architecture variants. The encoder-decoder model with the denoising objective outperformed other architectures. The Encoder-decoder with sharing parameters performed similar well as the former one. This result confirmed that "sharing parameters across Transformer blocks can be an effective means of lowering the total parameter count without sacrificing much performance."[26]

[5]`https://www.tensorflow.org/datasets/catalog/c4`

Objective	Inputs	Targets
Prefix language modeling	Thank you for inviting	me to your party last week .
BERT-style	Thank you <M> <M> me to your party apple week .	*(original text)*
Deshuffling	party me for your to . last fun you inviting week Thank	*(original text)*
I.i.d. noise, mask tokens	Thank you <M> <M> me to your party <M> week .	*(original text)*
I.i.d. noise, replace spans	Thank you <X> me to your party <Y> week .	<X> for inviting <Y> last <Z>
I.i.d. noise, drop tokens	Thank you me to your party week .	for inviting last
Random spans	Thank you <X> to <Y> week .	<X> for inviting me <Y> your party last <Z>

Figure 3.9.: The different denoising objective examples.

They tried different pre-training denoising methods regarding the unsupervised objectives, including Prefix language modeling, BERT-style[24], and Deshuffling. They concluded that BERT-style denoising objectives perform best. Furthermore, they explored different masking strategies to the BERT-style objective, including masking token-spans, replacing corrupted token-spans, and dropping corrupted token-spans. Figure 3.9 showed the different denoising details. It turned out that all the variants perform similarly, and replacing corrupted token-spans made the target sequence shorter and the training faster. So they further tried different corruption rates and different span lengths by replacing corrupted token-spans. They found that a larger corruption rate slowed down the training, so a corruption rate of 15% would be optimal. Moreover, an average span length of 3 slightly but significantly outperformed other length options. The flow chart in Figure 3.10 summarized their experiment process on unsupervised objectives. Based on this evaluation result, we applied the replacing corrupted token-spans with a corruption rate of 15% and a length of 3 to our pre-training unsupervised tasks.

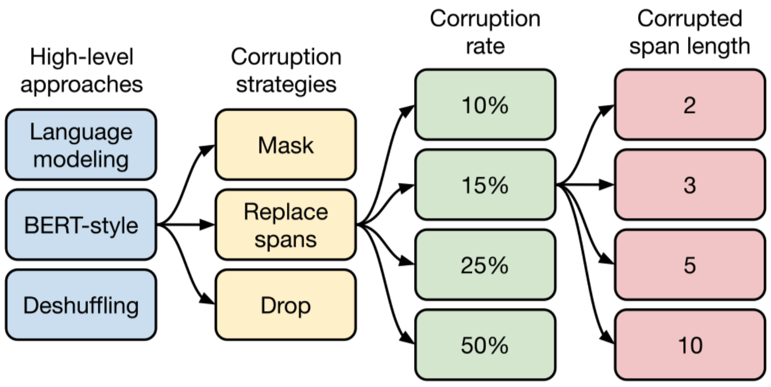


Figure 3.10.: Experiment steps on unsupervised objectives and the selected decisions.

Besides, they also tried different per-tained datasets, different fine-tuning parameters, different multitask training ways, and different model sizes. They concluded that additional pre-training, increasing the batch size, and the number of training steps could help to get good results. They also showed that pre-training on a multi-task mixture of unsupervised and supervised tasks performed similarly to pre-training on the unsupervised task alone. Their research gave us many inspirations for training our tasks in the software development domain. Their proposed **tensorflow T5**[6] library is very suitable for applying transfer learning and multitask learning. So we used their Small (around 60 million parameters), Base (about 220 million parameters), and Large (roughly 770 million parameters) model configurations implemented by the T5 library in our experiments.

[6]https://github.com/google-research/text-to-text-transfer-transformer

4. Approach

In this chapter, we explain our approach used in this thesis. We introduce the datasets we used, including the unsupervised and supervised datasets, the model architecture, and its parameters. Also, we describe the vocabulary model and the evaluation metrics for the experiments.

4.1. Datasets

In this section, we introduce our unsupervised and supervised datasets, their statistics, and the pre-processing methods on each dataset.

4.1.1. Unsupervised Dataset

We first introduce the unsupervised Datasets we used in this thesis. These datasets involve different programming languages and the English natural language. These datasets are used in the transfer learning pre-training steps and the multi-task learning. They are helpful to build a language model for tasks in the software development domain and make the final model more generalized against overfitting.

CodeSearchNet Corpus Collection

As we have mentioned in Section 3.1.1, CodeSearchNet Corpus Collection[40] is extracted from the open-source GitHub repositories. It contains six programming languages' functions/methods, including Python, Java, Go, Php, Ruby, and Javascript. This dataset can be divided into two parts - functions *with* the function documentation and functions *without* documentation. The dataset is stored in JSON format and contains already tokenized code and docstrings. The code is parsed and tokenized by the modified **tree-sitter**[1] library for each programming language.

Both parts of the dataset are being used for the pre-training. We directly used the parsed and tokenized functions without the documentation as unsupervised input. In this way, the model could understand the relationship between code structures, parameters, and statements for each programming language.

[1]`https://github.com/tree-sitter/tree-sitter-python`

When applying the dataset with functions containing documentation for pre-training, we first used the language detection library **langdetect**[2] to extract documentation having English text. Then we concatenated each pair of tokenized-function and its tokenized-documentation as one input sentence sequence:

$$(function, documentation) \rightarrow function \cdot documentation$$

In this way, the model could learn the programming languages and their relation to the English documentation. Since we only used the training data from this dataset, our model would not see the documentations in the validation and test set and would not corrupt the evaluation stage.

The Public Git Archive - Java

Four out of six supervised tasks involve the programming language Java. So an unsupervised dataset for Java would be essential to help the model understand this programming language. We used the Java code from the Public Git Archive dataset[63] as our unsupervised dataset for Java. This dataset contains top-rated repositories on GitHub. It has an index file with files in the Siva format, which is a novel archive format tailored for efficiently storing Git repositories.

Different from the fact that code from CodeSearchNet Corpus is function-level, the Public Git Archive has code in the Java file-level containing the import statements, multiple functions, and comments. This file-level data could help the model understand more information like API usage and benefit our downstream tasks.

After getting the Java code, we applied the **javalang**[3] Python library to parse and tokenize the code and consider the file-level tokenized Java code as the input Java data for pre-training. We replaced the code type of string and number as *CODESTRING* and *CODEINTEGER*.

The Public Git Archive - CSharp

Similar to Java unsupervised dataset, we also used the CSharp code extracted from the Public Git Archive dataset[63]. The CSharp data is the file-level code as well. Each input sequence contains importing libraries, multiple CSharp functions, and comments to part of the code. We used the **ANTLR** (ANother Tool for Language Recognition)[4] library to parse and tokenize the CSharp code before putting it into our pre-training model. We replaced the code type of string and number as *CODESTRING* and *CODEINTEGER*.

[2]`https://pypi.org/project/langdetect/`
[3]`https://github.com/c2nes/javalang`
[4]`https://github.com/antlr/antlr4`

150k Python Dataset

We also have two supervised tasks about Python code. One task involves the function-level code understanding while the other tries to summarize Python in the code-snippet-level. So when choosing the unsupervised tasks, in addition to the Python dataset from CodeSearchNet Corpus Collection, we also included the 150k Python Dataset[5][64] from the SRILAB[6] (the Secure, Reliable, and Intelligent Systems Lab) at ETH Zurich.

The Python programs in this dataset are collected from GitHub repositories with permissive and non-viral licenses by removing duplicate files, forked projects, and obfuscated files. Raychev et al.[64] also parsed the code using the **Python AST parser**[7] included in Python 2.7 and kept only programs having at most 30,000 nodes in the AST.

We used the file-level Python code as the pre-training input for our model. We applied the Python **tokenize**[8] library to parse the code and replace the token type of string and number as *CODESTRING* and *CODEINTEGER*. Then we tokenized code and fed the pre-processed code into our neural network.

StaQC - SQL

Our source code summarization task for SQL uses the code snippets from StackOverflow. So we chose another SQL Dataset, StaQC[9][65], which is also extracted from StackOverflow. StaQC contained SQL question-code pairs of questions tagged by "sql," "database," or "oracle" from StackOverflow. The SQL code in StaQC is code-snippet level as well.

We only used the SQL code part of StaQC. We further applied the Python library **sqlparse**[10] to parse the dataset. We cleaned the code and replaced the column and tab names as *col* or *tab* followed by an integer to differentiate different columns and tabs in one code snippet. We also replaced numbers in the code as *CODEINTEGER*, *CODEFLOAT*, or *CODEHEX*. Then we tokenized the cleaned SQL code as the input to our model.

LISP Dataset

The supervised program synthesis task used LISP inspired DSL programming language. However, we did not find any LISP or LISP related dataset on the internet. So we built the LISP dataset by ourselves.

We selected 20 GitHub repositories having the most stars from the Lisp Topic[11] in Github.

[5]sri.inf.ethz.ch/py150
[6]https://www.sri.inf.ethz.ch/
[7]https://docs.python.org/2.7/library/ast.html
[8]https://docs.python.org/2/library/tokenize.html
[9]https://github.com/LittleYUYU/StackOverflow-Question-Code-Dataset
[10]https://pypi.org/project/sqlparse/
[11]https://github.com/topics/lisp?o=desc&s=stars

Then we used the GitHub Rest API[12] to download these repositories. Moreover, we wrote a LISP parser to go through each file, selected the function-level LISP code, removed the comments, and tokenized the code as the input to our pre-training model.

One Billion Word Language Model Benchmark - English

Despite the programming languages, our model would also understand and generate the natural English language. So we need an unsupervised English dataset to help the model understand English words, word phrases, and sentences. We chose one Billion Word Language Model Benchmark corpus[13][66] as our unsupervised English dataset.

Text data in one Billion Word Language Model Benchmark corpus is obtained from the WMT11 website[14]. Normalization and tokenization are applied to the data. Duplicated sentences are removed. The vocabulary is constructed by discarding all words with a count below three. Words outside of the vocabulary are mapped to <*UNK*> token. Sentence order is randomized. Finally, the corpus contains almost one billion words of training data.

Since the text data from this corpus is already tokenized, we directly used the sentence from this corpus as our input to the pre-training models.

Statistics

Before we put the data into the model, we removed the unnecessary empty spaces in the data. In addition we replaced the token $\backslash t$ as <*tab*> and the token $\backslash n$ as <*newline*>. We used the Python **pandas**[15] library to load the data, carry out the preprocessing, and save the data in the TSV file format.

Table 4.1 shows the number of samples each dataset used in unsupervised learning. In total, we have around 40 million samples for this unsupervised pre-training. One Billion Word Language Model Benchmark corpus has more than 30 million data samples and is the corpus with the most number of samples. Among the programming language datasets, CodeSearchNet Corpus is the most extensive corpus. When comparing only the programming languages, the Java language has the most unsupervised samples with more than two million inputs. Javascript and Python follow it. Both of them have more than one million samples. Ruby, SQL, and LISP have the least number of unsupervised inputs. They have only around 150,000 samples or even fewer samples each.

Table 4.2 lists the data attributes of each programming language unsupervised dataset, like the data source, the code level, and the average length per sample. Most of the unsupervised programming language data is extracted from GitHub, except the StaQC - SQL dataset, which

[12] `https://docs.github.com/en/free-pro-team@latest/rest/reference/repos#contents`
[13] `http://www.statmt.org/lm-benchmark/`
[14] `http://statmt.org/wmt11/training-monolingual.tgz`
[15] `https://pandas.pydata.org/`

Language	CodeSearchNet Without Doc	CodeSearchNet With Doc	150k Python Dataset	The Public Git Archive	StaQC	LISP	One Billion Word Corpus	Total
Python	657,030	375,210	149,114					1,181,354
Java	1,070,271	373,412		720,124				2,163,807
Go	379,103	300,882						679,985
Php	398,058	369,923						767,981
Ruby	110,551	43,803						154,354
Javascript	1,717,933	99,646						1,817,579
CSharp				469,038				469,038
SQL					133,191			133,191
LISP						122,602		122,602
English							30,913,716	30,913,716
Total		5,895,822	149,114	1,189,162	133,191	122,602	30,913,716	38,403,607

Table 4.1.: The number of samples of each unsupervised dataset for different programming languages and the English natural language. The first column listed the languages. For programming languages, each sample can be considered as one function or a programming file, or part of the code, depending on the code level of that dataset. For the English language, one sample means one sentence.

Dataset	Data Source	Code Level	Average Length per Sample
CodeSearchNet - Without Documentation	GitHub	Function	178
CodeSearchNet - With Documentation	GitHub	Function	191
150K Python Dataset	GitHub	File	1055
The Public Git Archieve - Java	GitHub	File	770
The Public Git Archieve - CSharp	GitHub	File	239
StaQC - SQL	StackOverflow	Code Snippet	55
LISP	GitHub	Function	101

Table 4.2.: The programming language datasets have different data attributes, including the data source, code level, and average length per sample in each dataset.

contains the SQL queries from StackOverflow. The code level of the StaQC is also code snippet level. Each sample of the CodeSearchNet and LISP dataset is a code function or code method. The rest of the corpus contains each sample as a whole code file, including multiple functions.

We also list here the average length per sample in each dataset. Since we used the **SentencePiece**[16] library to generate the model vocabulary, few single tokens may be split into multiple components in our vocabulary. Here we calculated the number of vocabulary SentencePiece components in each sample. We explain the SentencePiece vocabulary in Section 4.2. We can observe that each code function and each code snippet has less than 200 vocabulary components averagely. The average of each CSharp file has a length of 239. The average length of each Java file is 770. Each Python sample has the most vocabulary components averagely with a length of 1055.

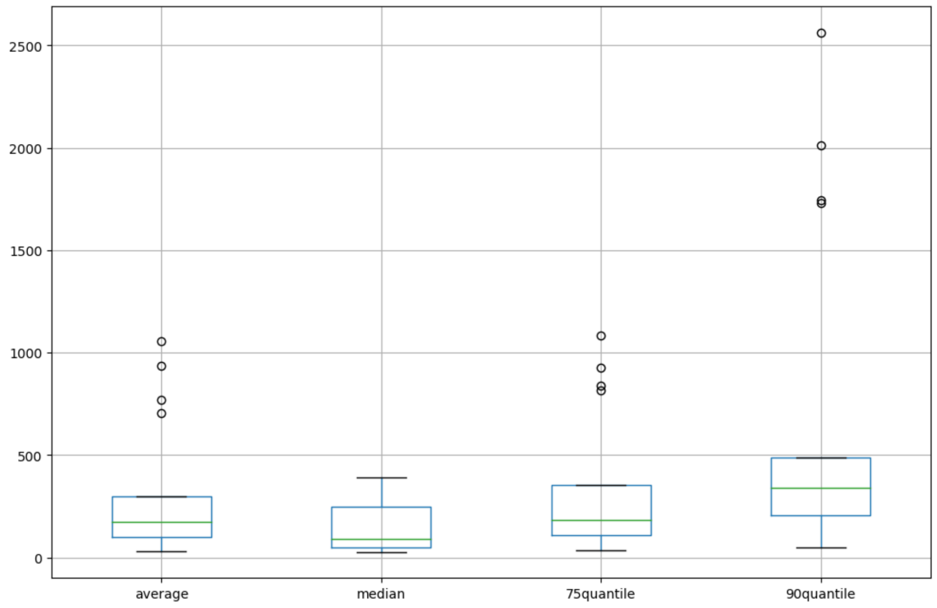


Figure 4.1.: The Boxplot of the unsupervised datasets' sample-length in average, median, 75-percent-quantile and 90-percent-quantile. The vertical axis shows the number of SentencePiece tokens in one sample. We call it the length of the sample. We calculated the average, median, 75-percent-quantile, and 90-percent-quantile of the sample length in each dataset. We collected statistics for the whole unsupervised datasets and plotted each value using the boxplot. We can infer from this plot that 90% of the samples in most datasets have less than 500 tokens.

Furthermore, we calculated the average, median, 75-percent-quantile, and 90-percent-quantile of each unsupervised data corpus component length and plotted them using the boxplot in Figure 4.1. We can see that 75 percent of the 90-percent-quantile sample length in each corpus

[16]https://github.com/google/sentencepiece

is less than 500. So we would cover most data information for unsupervised learning if we choose the input length as 512.

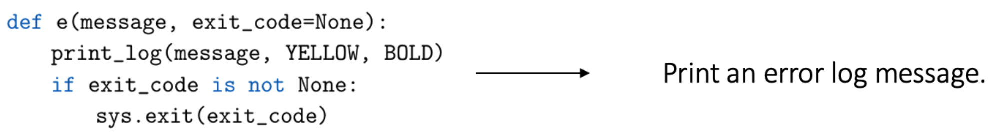
4.1.2. Supervised Dataset

In this section, we introduce the supervised Datasets we used in this thesis. There are six datasets, with those, we did experiments on six tasks. We have already introduced the original source of these datasets in Section 3.1. We give more details and examples about the datasets here.

Code Documentation Generation

We selected CodeSearchNet Corpus Collection[40] for the Code Documentation Generation supervised task. We used the dataset preprocessed by CodeBERT[39]. This part of the dataset contains functions with their documentations for six programming languages. Based on the data downloaded from the CodeSearchNet GitHub repository[17], CodeBERT removed comments in the code, and programming codes that cannot be parsed into an abstract syntax tree. In addition, they removed documents contain special tokens like "<img ...>" or "https" and only kept English documentations with the token size between 3 and 256. The code and documentations we used are already tokenized.

For the Code Documentation Generation task, we inputted the code function into our model and trained the model to generate the corresponding documentations. The standard reference for the model is the documentation from the dataset. Figure 4.2 shows an Python program example. The left side of the arrow is the Python function and the right side of the arrow is the desired documentation to generate.



```python
def e(message, exit_code=None):
    print_log(message, YELLOW, BOLD)
    if exit_code is not None:
        sys.exit(exit_code)
```

Print an error log message.

Figure 4.2.: A Python example from CodeSearchNet Corpus Collection. The left side of the arrow is an example of the Python method as the input to our model. The right side of the arrow is the expected output from the model.

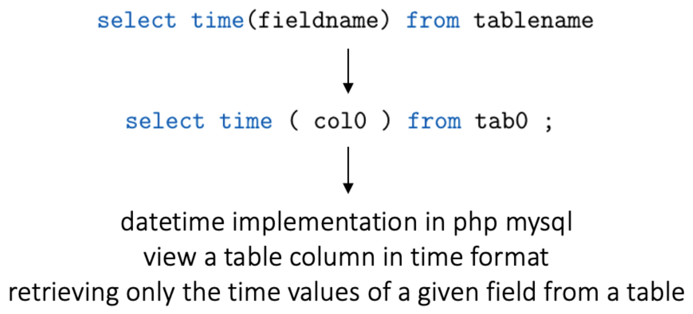
Source Code Summarization

We used the same datasets as CODE-NN[42] for the Source Code Summarization task. In their experiments, they only trained and tested CODE-NN on CSharp and SQL code. Nevertheless,

[17]https://github.com/github/CodeSearchNet

they also provided Python training and testing datasets in their repository. We downloaded the CODE-NN GitHub repository[18], then we followed their instructions, preprocessed the dataset, parsed the code, and replaced some tokens with their code types. We tokenized the natural language summarization using the **tokenize** package from the Natural Language Toolkit (NLTK). We inputted the Python, SQL, and CSharp code snippet into our model and expected the model to generate a summarization for this code snippet. We also chose only the examples annotated by human-annotators, to evaluate our model output performance.

Figure 4.3 illustrates one SQL example from the dataset. The most above SQL is the original code snippet from StackOverflow. We preprocessed the original SQL and generated the SQL in the middle. We also list three golden references for the model output at the bottom of this Figure. The first of three is the summarization extracted from StackOverflow. The rest two are human-annotated summarization for this SQL snippet.



```
select time(fieldname) from tablename
```

```
select time ( col0 ) from tab0 ;
```

datetime implementation in php mysql
view a table column in time format
retrieving only the time values of a given field from a table

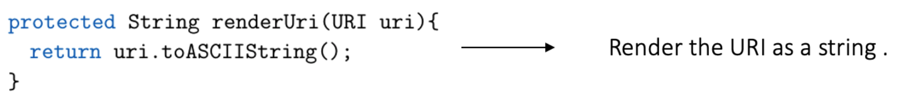
Figure 4.3.: A SQL example for the Souce Code Summarization task. The upper row is the original code snippet from StackOverflow. The middle row is the code snippet after preprocessing. We list three ground truth summarizations for this code snippet at the end of the arrow.

Code Comment Generation

We used the same corpus[19] as DeepCom[47] for the Code Comment Generation task. Different from their pre-processing steps, we did not convert the code into AST or SBT. We used **javalang** library to parse and tokenized the Java method. We then used the tokenized code to let the model generate a comment for the code, which described what the code is used. We tokenized the reference comment using the **tokenize** package from the Natural Language Toolkit (NLTK).

Figure 4.4 shows an example for this task. The left side of the arrow is the Java code. The right side of the arrow is the comment our model need to generate.

[18]`https://github.com/sriniiyer/codenn`
[19]`https://github.com/xing-hu/DeepCom`



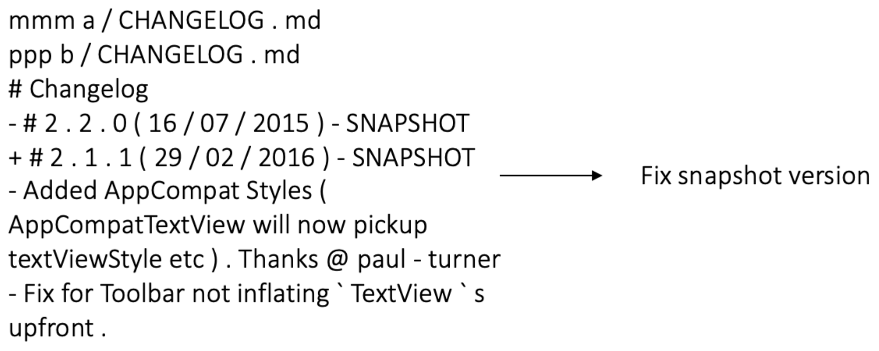
```
protected String renderUri(URI uri){
  return uri.toASCIIString();
}
```

Render the URI as a string .

Figure 4.4.: A Java example for Code Comment Generation task. The left part of the arrow is an example of the Java method as input to the model. The right part of the arrow is the desired model output.

Git Commit Message Generation

The task Git Commit Message Generation aims to generate a commit message describing the git commit changes. We used the dataset[20] provided by Jiang et al[49]. We inputted the Java comment changes into the model and got a commit message as the output. The input comment changes and output commit messages are all preprocessed and tokenized. Since the reference commit message only contained one sentence. So the output also should have one sentence. Figure 4.5 is an example of this task.



```
mmm a / CHANGELOG . md
ppp b / CHANGELOG . md
# Changelog
- # 2 . 2 . 0 ( 16 / 07 / 2015 ) - SNAPSHOT
+ # 2 . 1 . 1 ( 29 / 02 / 2016 ) - SNAPSHOT
- Added AppCompat Styles (
AppCompatTextView will now pickup
textViewStyle etc ) . Thanks @ paul - turner
- Fix for Toolbar not inflating ` TextView ` s
upfront .
```

Fix snapshot version

Figure 4.5.: An example for the Git Commit Message Generation task. The left side of the arrow is the git commit diff. "+" means the adding content for this change while "-" means the removing part during this commit. We put this whole diff in the model and expected the output as the arrow's left side to describt this commit change.

The left side of the arrow is the git commit diffs. The right side of the arrow is the commit message our model should generate.

[20]https://sjiang1.github.io/commitgen/

API Sequence Recommendation

We aimed to generate the API usage sequence from a short natural language description in this task. We adopted the dataset[21] extracted by Gu et al[52]. for training and evaluating the DeepAPI model. Figure 4.6 gives an example from the dataset for this task. The sentence above tells the model to give API suggestions for converting RGB to HSB. We put this sentence into our model. The expected model output below suggests that we need to use the library *Color*, use the *RGBtoHSB* method from this library to finish the converting process, and use the *getHSBColor* method to return the HSB result. The original authors already tokenized the dataset they have published.

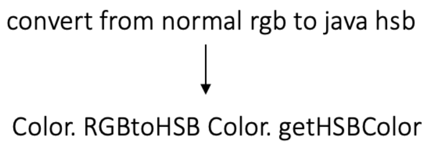

convert from normal rgb to java hsb
Color. RGBtoHSB Color. getHSBColor

Figure 4.6.: An example for the API Sequence Recommendation task. The upper part of the arrow is the description for a programming task request, which is also the input to our model. It expects the recommendation that the Color library needs to be used and call methods "RGBtoHSB" and "getHSBColor." The model should suggest this API Sequence Recommendation as the output.

Program Synthesis

We used AlgoLisp[22] dataset[59] for the Program Synthesis task. This dataset is extracted from homework assignments for introductory computer science courses, so each example in this dataset consists of a question and an answer. We inputted the question into our model and expected the model to output the correct LISP-inspired DSL answer. The dataset is already parsed and tokenized as a list format and stored in a JSON file. We concatenated each element from the list using the space and converted the list to the String. So we have String as the model input and output as shown in Figure 4.7.

Statistics

Table 4.3 compares the number of samples in training, validation, and testing datasets per supervised learning tasks. We could observe that the API Sequence Recommendation has the largest number of samples. It has 600 times more samples than the smallest dataset for the Source Code Summarization Python task. It is also larger than all the unsupervised programming language datasets. The second-largest dataset is the Code Comment Generation dataset. The sample number of the rest datasets is around or less than 250,000. Four out of

[21]`https://github.com/guxd/deepAPI`
[22]`https://github.com/nearai/program_synthesis/tree/master/program_synthesis/algolisp`

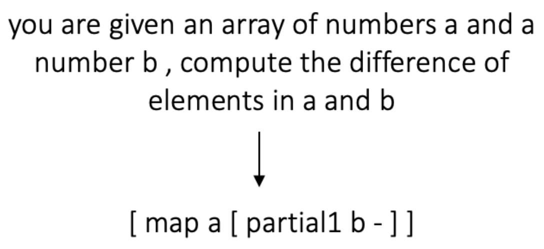

you are given an array of numbers a and a number b , compute the difference of elements in a and b
[map a [partial1 b -]]

Figure 4.7.: An example for the Program Synthesis task. Above the arrow is the input for the model. It describes a question the code should solve. Below the arrow is the expected output. It is a LISP inspired DSL code.

six tasks have the datasets extracted from GitHub. Furthermore, three out of six tasks used the function-level datasets.

Moreover, we calculated the SentencePiece token length per input and output sample. It is evident that programming languages have more tokens than natural languages. Besides, we calculated the average, median, 75-quantile, and 90-quantile of the input and output sample SPM lengths. We plot the result using the Boxplots in Figure 4.8. We can see that most of the samples have a token length of less than 500. So it is also acceptable for the supervised learning model to have an input size of 512.

Task	Language	Train	Valid	Test	Data Source	Data Level	Average Token Length per Input	Average Token Length per Output
Code Documentation Generation	Python	251,820	13,914	14,918	GitHub	Function	169.98	21.05
	Java	164,923	5,183	10,955				
	Go	167,288	7,325	8,122				
	Php	241,241	12,982	14,014				
	Ruby	24,927	1,400	1,261				
	Javascript	58,023	3,885	3,291				
Source Code Summarization	Python	12,004	2,792	2,783	StackOverflow	Code Snippet	58.84	10.8
	Csharp	52,943	6,601	108				
	SQL	25,671	3,326	100				
Code Comment Generation	Java	470,451	58,811	58,638	GitHub	Function	123.21	21.68
Git Commit Message Generation	Java	26,208	3,000	3,000	GitHub	Commit	116.24	9.15
API Sequence Recommendation	Java	7,475,850	-	10,000	GitHub	API	10.95	16.57
Program Synthesis	DSL	79,214	10,819	9,967	Homework	Function	37.97	51.94

Table 4.3.: The summarization of supervised datasets. It includes the number of samples in training, validation and testing datasets, data source, programming data level for each sample, and average token length per input and per output sample for each supervised dataset. Each sample could be a code function, a code snippet, a commit diff or a natural language sentence, depending on the data level.

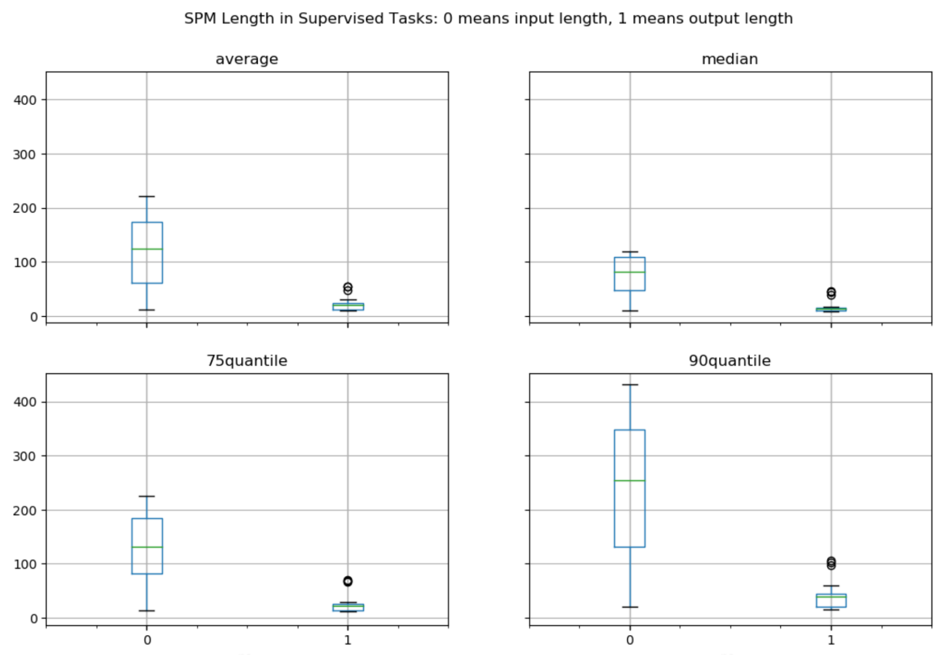


Figure 4.8.: Input and output sample length in boxplot for supervised datasets. The vertical axis means the number of SentencePiece tokens in each sample as the sample length. 0 means the input sample, 1 means the output sample. We calculated the average, median, 75-quantile-percentage, and 90-quantile-percentage of the sample length for each dataset. Then we collected this statistic for all the supervised datasets and plotted them using the boxplot. From this Figure, we can infer that 90% of the samples in most datasets have samples with less than 500 tokens.

4.2. Vocabulary

Vocabulary is an essential aspect in natural language processing. Vocabulary itself contains much information about the corpus, like the corpus domain, formality, tone, and target audience. Vocabulary is helpful when preprocessing the text corpus. The preprocessing methods like one-hot-encoding or bag-of-words are based on a fixed size of the vocabulary. Vocabulary is also the storage to construct the output of the model. The choice of vocabulary would have a critical impact on model performance and output quality. The token frequency in the vocabulary also indicates the different importance of the text information.

However, there is one problem when choosing the vocabulary - It is impossible to choose an unlimited size of the vocabulary. So the chosen vocabulary should contain most of the tokens from the datasets and could reconstruct the most information from the corpus.

We used the **SentencePiece**[23] library to construct the vocabulary for this thesis. It extracts

[23]https://github.com/google/sentencepiece

the sub-word units from the complete corpus to create the open-vocabulary. It is designed initially to solve the fixed word vocabulary problem in machine translation. Because limiting vocabulary size would increase the number of unknown tokens and makes the translation output inaccurate. Breaking up rare words into subword units is a common way to deal with this problem.

We used the unigram language model algorithm[67] provided by this library. This algorithm first uses all characters' union and the most frequent substrings in the corpus to obtain an initial vocabulary. In the second step, it optimizes the subword occurrence probabilities based on the EM algorithm. Then it computes the loss about how likely the input sentences having lower likelihood if using the current subwords to construct the sentences. Top subwords are then kept by sorting based on the loss. This second step is performed repeatedly to get the desired vocabulary size. The final vocabulary can be considered as a probabilistic mixture of characters, subwords, and word segmentations.

We applied the Python **glob**[24] library to collect all the supervised and unsupervised datasets used in this thesis by recursively searching TSV files in our corpus folders. Then we connected all the file names to a string using the comma symbol as the file input for our SentencePiece training method. We set the id for padding token (<pad>) as 0, EOS token (</s>) as 1, Unknown token (<unk>) as 2, and BOS token (<s>) as 3. We set the size of the vocabulary as 32,000. The whole datasets have in total more than 46 million lines (Each line could be considered as one model input example and one SentencePiece input sentence). It is tremendous when using the unigram language model algorithm and would cause the training crash for training on the whole sentences. So we limited the "input sentence size" to 40 million, shuffled the input sentences to get random sentence inputs, and enabled the setting for training an extremely large corpus. We set the character coverage as 0.9999 because the corpus may contain not English characters or meaningless symbols. In this way, we could exclude these noises from the vocabulary. The SentencePiece training code is listed as follows:

```python
import sentencepiece as spm

spm.SentencePieceTrainer.train(input=spm_input,
                               pad_id=0, eos_id=1, unk_id=2, bos_id=3,
                               model_prefix='code_spm_unigram_40M',
                               vocab_size=32000,
                               input_sentence_size=40000000,
                               shuffle_input_sentence=True,
                               character_coverage=0.9999,
                               model_type='unigram',
                               train_extremely_large_corpus=True)
```

Listing 4.1: Code example for exacting the vocabulary from the total corpus using SentencePiece library

[24]https://docs.python.org/3/library/glob.html

From the generated vocabulary, we may notice there are quite a lot tokens indicating the programming languages and processes, including "function," "String," "var," "import," etc. So this vocabulary is suitable for the natural language processing tasks in the software development domain.

4.3. Model Architecture

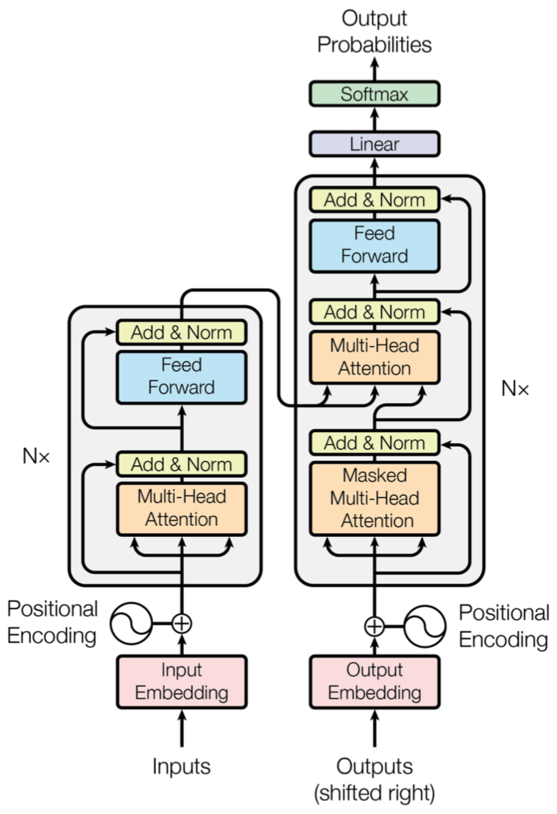


Figure 4.9.: The transformer architecture[21].

We used the Text-to-Text Transfer Transformer (T5) framework to build our model. As introduced in Section 3.2, T5 is an encoder-decoder Transformer while both the encoder and decoder part contains attention-layers followed by the feed-forward neural network. Figure 4.9 illustrate these components in the architecture. Five different sizes of T5 are provided - Small, Base, Large, 3B, and 11B. We used the Small, Base, and Large model in this thesis.

Table 4.4 lists the Small, Base, and Large model's size and hyperparameters. Each block consists of self-attention, optional encoder-decoder attention, and a feed-forward network. The Small, Base and Large model has 6, 12, and 24 blocks in both the encoder and the decoder. The feed-forward networks in each block contain a dense layer with an output dimensionality

	Small	Base	Large
Number of Blocks Each	6	12	24
Dense Layer Output Dimension	2048	3072	4096
Attention Layer Key Value Dimension	64	64	64
Number of Attention Heads	8	12	16
Sub-layers and Embeddings Dimension	512	768	1024
Total Parameters (in Million)	60	220	770

Table 4.4.: Important hyperparameters for the architecture of text-to-text transfer transformer model in the size of small, base, and large model.

of 2048, 3072, and 4096, followed by a ReLU nonlinearity and another dense layer in the Small, Base Large models. These three models have the inner dimensionality of 64 for the "key" and "value" matrices in each attention layer. Nevertheless, the number of attention heads and the dimensionality of all other sub-layers and embeddings are different. In conclusion, the Base T5 model has 3.6 times more parameters than the Small model, and the Large model has 3.5 times more parameters than the Base model.

We call our models **CodeTrans** because these models are based on the Transformer architecture. We set the input and output length of the model as 512. We disabled the method of *reduce_concat_tokens*. This method is designed originally to concatenate multiple unrelated documents to create the exact right length and avoid wasting space on padding. However, this would also cause the training example to be split into multiple parts and break the programming codes' sequences. For the unsupervised objective, we applied the replacing corrected spans corruption strategies with the corruption rate of 15% and the corrupted span length of 3. We considered a span of average 3 corrupted tokens as an entirety and used an unique mask token to replace it. The target sequence consisted of the corrupted spans with the mask tokens in front of these spans, which were used to replace them in the input.

The T5 framework is very suitable for transfer learning, multi-task learning, and fine-tune the models. It has the Python Class *TaskRegistry* and *MixtureRegistry*. Each task can be built as one TaskRegistry. One or more TaskRegistries can build one MixtureRegistry. We built 13 TaskRegistries, one MixtureRegistry for unsupervised learning, and another MixtureRegistry for multi-task learning. We used the mesh_transfomer method to train the model with a null init checkpoint. We just specified the pre-trained checkpoint as the init checkpoint for fine-tuning the model and continued to train diverse tasks based on that checkpoint. We chose the norm decay learning rate for training and fine-tuning. All these can be explicitly configured using the gin-config[25] settings.

[25]`https://github.com/google/gin-config`

4.4. Evaluation Metrics

The evaluation of natural language generation tasks measures the quality of the generated texts. It is not easy to compare the meaning of texts, because different combinations of tokens would have the same meaning. We used in this thesis mainly the objective human likeness measures. Such methods compare the model's output with the golden standard reference and calculate the overlap between generated outputs and the standard reference. We explain these metrics in this section.

4.4.1. BLEU score

BLEU score is proposed by Papineni et al. in 2002[51]. It is used initially to compare the machine translation outputs with several human translation references. It calculates the word overlap between the model output and the reference to get the model's precision. The basic formula of BLEU score is:

$$BLEU = BP \cdot exp(\sum_{i=1}^{N} W_n \log p_n) \tag{4.1}$$

$$BP = \begin{cases} 1 & c > r \\ e^{1-\frac{r}{c}} & c \leq r \end{cases} \tag{4.2}$$

$$p_n = \frac{\text{Word maximal occurrence of model output in reference}}{\text{Word occurrence in model output}} \tag{4.3}$$

N means the word n-gram, which considers the continuous sequence of n tokens as a whole for the calculation. It is normally set as 4. W_n is the weight to n-grams. p_n is the modified precision. c is the length of the model output. r is the length of the golden standard reference.

The smoothed BLEU score is proposed by Lin and Och[41]. This smoothing technique adds one count to the n-gram hit and the total n-gram count if n is larger than 1. In this way, the candidate output with less than n words can still get a positive smoothed BLEU score from shorter n-gram matches. Moreover, it will not influence the zero result score if nothing matches.

T5 framework applies the tool **ScareBLEU**[26] to compute the model text output. ScareBLEU is proposed by Matt Post[68] and aims to solve different reference preprocessing and parameter settings when computing the BLEU score. It tries to get a unified BLEU score for different models. ScareBLEU expects detokenized outputs and applies its own metric-internal preprocessing before calculating the score.

[26]`https://github.com/mjpost/sacrebleu`

We used the score from the original task papers, which we described in Section 3.1 as the baseline. We compared our model performance with theirs. So in addition to the T5 ScareBLEU script, it is essential that we apply the same BLEU score scripts with theirs when evaluating the results to keep consistency.

4.4.2. ROUGE score

ROUGE is called Recall-Oriented Understudy for Gisting Evaluation[69]. The BLEU score is a precision-related measurement, while the ROUGE score is a recall-related measurement. BLUE score calculates how much n-grams in the model outputs appear in the reference, while ROUGE score computes how much n-grams in the references appear in the machine outputs.

The ROUGE score formular is as follows:

$$ROUGE_n = \frac{\sum\limits_{S \in \{ReferenceSummaries\}} \sum\limits_{gram_n \in S} Count_{match}(gram_n)}{\sum\limits_{S \in \{ReferenceSummaries\}} \sum\limits_{gram_n \in S} Count(gram_n)} \tag{4.4}$$

where n means the length of n-gram. $Count_{match}(gram_n)$ is the maximum number of the n-gram co-occurring in the model output and the standard references.

We selected n equals 1 and 2, which refer to the overlap of unigram and bigrams between the system and reference summaries. Besides, we also applied ROUGE-L standing for the Longest Common Subsequence. It identifies the longest co-occurring in sequence n-grams automatically.

The T5 ROUGE score metrics are based on the Python library **rouge-score**[27].

4.4.3. Accuracy

Accuracy is a very common metrics to evaluate machine learning models. It is defined as:

$$Accuracy = \frac{\text{number of correct predictions}}{\text{total number of predictions}} \tag{4.5}$$

This metrics is rigorous when computing for a sequence data like a sentence or a programming function. Only if the complete sequence data is exactly the same as the reference data, it could be counted as one correct prediction. In the software development domain, if a programming code function could return the expected result, this code function could also be considered as correct. Therefore, functional testing can be designed to calculate the code accuracy defined as follows:

$$Code\_Accuracy = \frac{\text{number of predictions that passed the functional tests}}{\text{total number of predictions}} \tag{4.6}$$

[27]`https://pypi.org/project/rouge-score/`

The Program Synthesis task applied the Code_Accuracy by Polosukhin and Skidanov[59] to evaluate their model on the AlgoLisp dataset. Due to the complexity of converting text strings to a programming code, we applied the absolute accuracy to the same task. Since this accuracy is more strict than the Code_Accuracy, if a programming function is the same as the reference, it could definitely also pass the functional tests.

5. Experiment

This chapter explains our experiment details, including the hardware information, training and evaluation settings, and training processes using different training strategies.

5.1. Experimental Setup

In this section, we list the hardware information and software usage for our experiment.

5.1.1. Hardware

Calculation inside one neural network model requires a large number of matrix computations. Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) are very suitable to carry out such large computational tasks. These two kinds of hardware can focus on multiplication computations, which are exceptionally costly during matrix computation. According to the research by Wang et al.[70], TPU is well suited for large batch training, while GPU is a good choice for large datasets.

We used two types of Google TPUs, v2-8, and v3-8. We got access to two TPUs v2-8 through the Google Colab notebooks[1] and multiple TPUs v3-8 using Google Cloud Console. TPUs v3-8 are mainly used for multi-task learning, transfer learning pre-training, and fine-tuning models for large datasets, while TPUs v2-8 are applied for single-task training for the base model and fine-tuning the pre-trained models on relatively small datasets. Table 5.1 lists the specifications of these two types of TPU.

TPU type	TPU cores	Total TPU memory
v2-8	8	64 GiB
v3-8	8	128 GiB

Table 5.1.: Specifications of two kinds of Google Cloud TPUs we used in this thesis.

We also used one NVIDIA GPU Quadro RTX 8000[2]. It has 576 NVIDIA Tensor Cores, 72 NVIDIA RT Cores, and 48 GB GDDR6 with ECC GPU memory. This GPU is mainly utilized for single-task training the T5 small models.

[1]`https://colab.research.google.com/notebooks/intro.ipynb#recent=true`
[2]`https://www.nvidia.com/en-us/design-visualization/quadro/rtx-8000/`

5.1.2. Software Usage

We split the experiment stage into training, fine-tuning, and evaluation. The training stage for single-task learning uses similar scripts as the pre-training stage for multi-task learning and transfer learning. We evaluated the models on the validation sets to apply early stopping and find the best checkpoints. We got the models' final performance by evaluating them on the test sets.

Training

As explained in Section 4.3, T5 uses TaskRegistry and MixtureRegistry to assign the experiment tasks. Therefore, we first configured the 13 single-task learning tasks in TaskRegistry. Then we set two MixtureRegistries: one for multi-task learning tasks and one for transfer learning tasks. These registries were written in a Python file.

The training script is as follows. We imported the Python file containing the above configured tasks using the *module_import*. T5 uses gin-config to set the function parameters. So we set the *MIXTURE_NAME* as the name of the task to train. We chose the input token length as 512, hence the *tokens_per_batch* is 512 multiple batch size. The model size was set as "t5.1.0.small/base/large.gin". We set the training steps in *train_steps* and the *init_checkpoint* as none. We also configured the hardware settings like TPU type, TPU zone, model parallelism, etc. We ran the scripts to train the models.

```
python -m t5.models.mesh_transformer_main \
    --module_import="${MODULE_PYTHON_FILE}" \
    --tpu="${TPU_NAME}" \
    --gcp_project="${PROJECT}" \
    --tpu_zone="${ZONE}" \
    --model_dir="${MODEL_DIR}" \
    --gin_file="models/t5.1.0.large.gin" \
    --gin_file="dataset.gin" \
    --gin_file="learning_rate_schedules/rsqrt_no_ramp_down.gin" \
    --gin_param="MIXTURE_NAME='${NAME}'" \
    --gin_param="utils.tpu_mesh_shape.model_parallelism=1" \
    --gin_param="utils.tpu_mesh_shape.tpu_topology='v3-8'" \
    --gin_param="utils.run.save_checkpoints_steps=15000" \
    --gin_param="utils.run.batch_size=('tokens_per_batch',2097152)" \
    --gin_param="utils.run.train_steps=500000" \
    --gin_param="init_checkpoint=None" \
    --gin_param="utils.run.iterations_per_loop=5000" \
    --gin_param="SentencePieceVocabulary.extra_ids=100" \
    --gin_param="run.perplexity_eval_steps=100"
```

Listing 5.1: Code example for training tasks using T5 library

Fine-tuning

The fine-tuning scripts are almost identical to the training scripts. But fine-tuning is a kind of training based on the pre-trained checkpoints. So we set the pre-trained checkpoint address for *init_checkpoint* in the gin parameter. It worthes to notice that the *train_steps* should also take the pre-trained steps into account. So if the pre-training takes 100 steps, and we want to fint-tune 50 steps, we need to set the *train_steps* as 150. We changed different *MIXTURE_NAME* to fine-tune different tasks.

Evaluation

The evaluation script is listed below. We need to specify the address of *operative_config.gin* file of the trained model. We used the beam search to evaluate the model. The beam size is set as four. This script generates a file containing all the output for the evaluation examples, and returns the scores for evaluation metrics. These evaluation metrics are also defined in the Python file imported using the *module_import*.

```
python -m t5.models.mesh_transformer_main \
    --module_import="${MODULE_PYTHON_FILE}" \
    --tpu="${TPU_NAME}" \
    --gcp_project="${PROJECT}" \
    --tpu_zone="${ZONE}" \
    --model_dir="${MODEL_DIR}" \
    --gin_file="dataset.gin" \
    --gin_file="${MODEL_DIR}/operative_config.gin" \
    --gin_file="eval.gin" \
    --gin_file="beam_search.gin" \
    --gin_param="run.dataset_split='validation'" \
    --gin_param="utils.tpu_mesh_shape.tpu_topology='v3-8'" \
    --gin_param="MIXTURE_NAME='${NAME}'" \
    --gin_param="eval_checkpoint_step=${STEPS}"
```

Listing 5.2: Code example for evaluating models with tasks using T5 library

5.2. Single-task Learning

For Single-task Learning, we trained the 13 tasks separately using the T5 framework. We applied the small and base models. So we generate two models for each task and, in total, 26 models. We tuned the batch size using the grid search inside the range of 2^5 and 2^{10}. We determined the training steps using early stopping concerning the models' performance on the validation sets based on the T5 built-in BLEU and ROUGE scores.

Table 5.2 shows the batch-size, hardware, and the training step for the small and base model with the best performance on each task. We also list here the number of samples in each

Task	Language	Sample Size	Model Size	Batch	Hardware	Steps
Code Documentation Generation	Python	251,820	Small	256	TPU v2-8	20,000
			Base	384	TPU v2-8	90,000
	Java	164,923	Small	256	TPU v2-8	60,000
			Base	256	TPU v3-8	80,000
	Go	167,288	Small	256	TPU v2-8	5,000
			Base	256	TPU v2-8	80,000
	Php	241,241	Small	256	TPU v2-8	200,000
			Base	1024	TPU v3-8	30,000
	Ruby	24,927	Small	128	GPU	10,000
			Base	128	TPU v2-8	8,000
	Javascript	58,023	Small	256	TPU v3-8	16,000
			Base	256	TPU v3-8	18,000
Source Code Summarization	Python	12,004	Small	233	GPU	5,000
			Base	32	GPU	1,000
	Csharp	52,943	Small	128	GPU	2,000
			Base	32	GPU	500
	SQL	25,671	Small	128	GPU	500
			Base	32	GPU	500
Code Comment Generation	Java	470,451	Small	256	TPU v2-8	520,000
			Base	256	TPU v2-8	80,000
Git Commit Message Generation	Java	26,208	Small	128	GPU	15,000
			Base	512	TPU v3-8	4,000
API Sequence Recommendation	Java	7,475,850	Small	256	TPU v2-8	840,000
			Base	256	TPU v2-8	145,000
Program Synthesis	DSL	79,214	Small	512	TPU v2-8	6,000
			Base	256	TPU v2-8	10,000

Table 5.2.: The single-task learning experiment setups for each task. We list here the batch-size, hardware, and the training step for the small and base model with the best performance on each task using single-task learning.

dataset for comparison. We used GPUs to train part of the small models and base models with a small batch size for tasks having a small sample size in the dataset. We could notice the following points during single task training:

- **The number of samples** in a dataset has an essential impact on the **model size** and the **training steps**. Task API Sequence Recommendation and Code Comment Generation have the two most enormous datasets. The small models for these two tasks require almost seven times more training steps than the base models till the models could converge.
- Corpus for Source Code Summarization converges extremely fast. For SQL and CSharp datasets in this corpus, the base model converged already in 500 training steps even the batch size is only 32, and the model had not seen the complete dataset yet. The scores on this task's validation set became worse if we trained the model with more steps. So it is very **easy to overfit** the models for the Source Code Summarization task.

- Half of the models achieves the best performance with **a batch size of 256**. However, it varies slightly among different tasks, like the Source Code Summarization task requiring small batch sizes. Nevertheless, large batch sizes do not result in better performance, no matter the number of samples in the dataset.

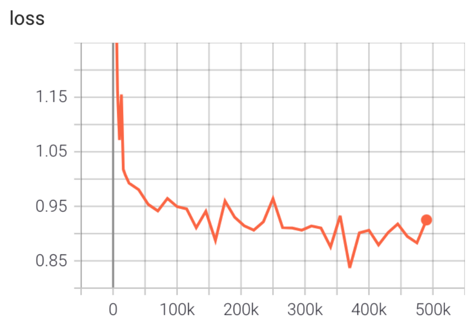
We evaluated the models using the BLEU and ROUGE scores (and Accuracy for the Program Synthesis task additionally) on the test dataset to obtain its final performance and compare the result with the baseline. We explain these comparisons in Section 6.

5.3. Transfer Learning

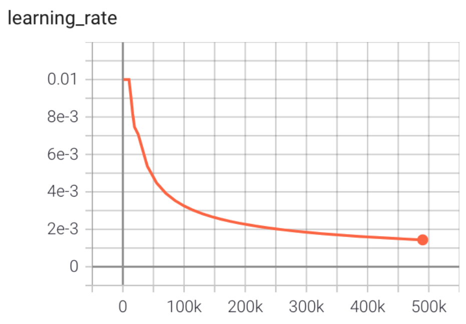
Transfer Learning has two steps, pre-training and fine-tuning. We applied the small, base, and large T5 models for transfer learning.

5.3.1. Pre-training

All the unsupervised tasks are used in the pre-training step. We set the T5 model to mask the spans of input data by enabling unsupervised parameter gin file. The model needs to predict what is the masked content and builds an initial language model in this way. Since our pre-trained models used the datasets containing nine programming languages, these models are suitable be fine-tuned on other downstream tasks in the software development domain.



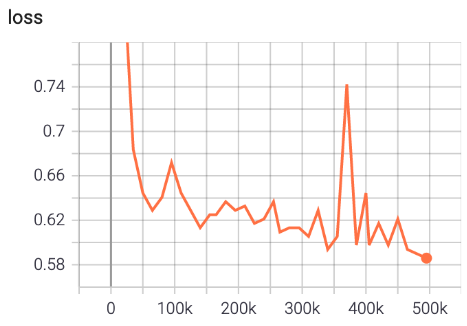
(a) The loss change of T5 small model in transfer learning pre-training steps. The y-axis shows the loss and the x-axis lists the training steps.



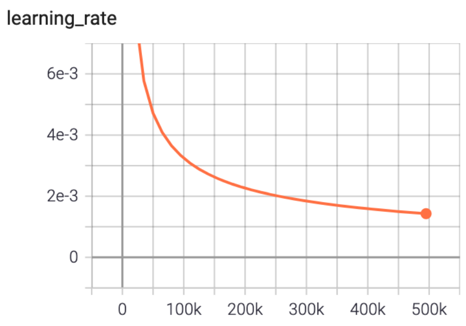
(b) The learning rate change of T5 small model in transfer learning pre-training steps. The y-axis shows the learning rate and the x-axis lists the training steps.

Figure 5.1.: The development of Loss and learning rate of the T5 small model during pre-training

We chose the batch size as 4096 and training steps as 500,000 for pre-training the small model. We utilized the TPU v3.8 in the pre-training. It took around 17 days to pre-train the small T5 model for half a million steps. Figure 5.1 illustrates the loss and learning rate changes during the pre-training. We obtained these charts using the Tensorboard[3]. After training more than 50,000 steps, the pre-training loss stayed under 1.0 stably. There existed variations, but the primary trend of the loss was going down slightly. The learning rate decayed, along with the increase of training steps. The pre-trained small model's final loss is 0.926.



(a) The loss change of T5 base model in transfer learning pre-training steps. The y-axis shows the loss and the x-axis lists the training steps.



(b) The learning rate change of T5 base model in transfer learning pre-training steps. The y-axis shows the learning rate and the x-axis lists the training steps.

Figure 5.2.: The development of Loss and learning rate of the T5 base model during pre-training

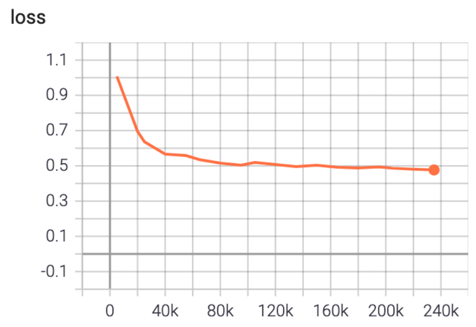
We chose the batch size as 4096 and training steps as 500,000 for pre-training the base model utilizing the TPU v3.8. Pre-training the base T5 model for 500,000 steps cost around 53 days. The changes of the base model's loss and learning rate during the pre-training are shown in Figure 5.2. We could see that the training loss went down rapidly during the first 50,000 steps. It increased largely again in the 370,000 steps then continued to decrease. This may mean that our base model jumped out of its local minimum in that step. The overall trend of the loss was decreasing. The pre-trained base model's final loss is 0.586.

We chose the batch size as 4096 and stopped the transfer learning pre-training at 240,000 steps for the large model because of the time constraint. We used TPU v3-8 during the pre-training. Pre-training the large T5 model for 240,000 steps cost more than 83 days. The changes of loss and learning rate during the pre-training are shown in Figure 5.3. We could observe that the training loss went down rapidly during the first 20,000 steps. The loss achieved 0.5 after 80,000 steps. After that, it continued to decrease slowly. The complete loss change was very smooth. The pre-trained large model's final loss is 0.476.

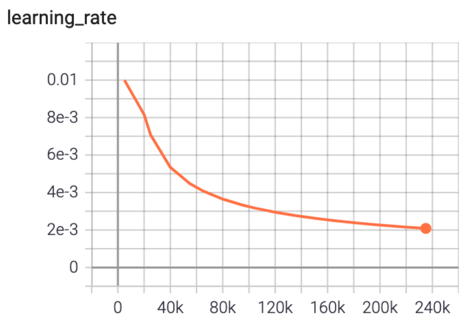
5.3.2. Fine-tuning

After obtaining the pre-training model on the 500,000 training steps for the small and base models and 240,000 steps for the large model, we fine-tuned the models for the 13 supervised

[3]https://www.tensorflow.org/tensorboard



(a) The loss change of T5 large model in transfer learning pre-training steps. The y-axis shows the loss and the x-axis lists the training steps.



(b) The learning rate change of T5 large model in transfer learning pre-training steps. The y-axis shows the learning rate and the x-axis lists the training steps.

Figure 5.3.: The development of Loss and learning rate of the T5 large model during pre-training

tasks. We have noticed that half of the single-task learning models reached their best performance with a batch size of 256. So we chose 256 as the batch size for fine-tuning the downstream tasks. We applied the early stopping to determine the fine-tuning steps based on the models' performance on the validation sets using BLEU and ROUGE scores.

Based on the experience gained from the single-task learning, we started to fine-tune the small models 5,000 steps for most of the tasks, recorded the validation scores, and continued to fine-tune 5,000 steps repeatedly until the models' performance converged on the validation sets. We also adjusted this **step interval** concerning the dataset attributes and the model size. We set this interval as 2,000 for the base models and 500 for the large models. We reduced this fine-tuning step interval for the Source Code Summarization task to 1,000 for the small models, 500 for the base models, and 100 for the large models because this corpus is very easy to overfit the models. For Code Comment Generation and API Sequence Recommendation tasks, we increased this interval to 50,000 for the small models and 10,000 for the base and the large models.

Table 5.3 lists the fine-tuning steps for the small, base, and large models to reach the best performance on each task. Quite a lot tasks reach the best performance already after fine-tuning the first 500/2,000/5,000 steps iteration. This can prove that fine-tuning downstream tasks using transfer learning can save the downstream tasks' training steps. The larger the model is, the fewer fine-tuning steps the model requires. Nevertheless, tasks with large datasets like Code Comment Generation and API Sequence Recommendation still require many fine-tuning steps, especially for the small models.

To make sure that our fine-tuning step interval is small enough to cover the best performance checkpoint, especially for the Source Code Summarization task, we fine-tuned the SQL task for 1,000 steps using the interval of 100 steps. Table 5.4 presents the evaluation results on the

Task	Language	Sample Size	Model Size	Hardware	TF-FT-Steps
Code Documentation Generation	Python	251,820	Small	TPU v3-8	5,000
			Base	TPU v3-8	2,000
			Large	TPU v2-8	500
	Java	164,923	Small	TPU v2-8	10,000
			Base	TPU v3-8	5,000
			Large	TPU v2-8	500
	Go	167,288	Small	TPU v2-8	10,000
			Base	TPU v2-8	5,000
			Large	TPU v2-8	1,000
	Php	241,241	Small	TPU v2-8	10,000
			Base	TPU v2-8	65,000
			Large	TPU v2-8	18,000
	Ruby	24,927	Small	TPU v3-8	5,000
			Base	TPU v3-8	5,000
			Large	TPU v2-8	1,000
	Javascript	58,023	Small	TPU v2-8	40,000
			Base	TPU v3-8	35,000
			Large	TPU v2-8	4,000
Source Code Summarization	Python	12,004	Small	TPU v3-8	5,000
			Base	TPU v2-8	1,000
			Large	TPU v2-8	100
	Csharp	52,943	Small	TPU v2-8	2,000
			Base	TPU v2-8	500
			Large	TPU v2-8	200
	SQL	25,671	Small	TPU v3-8	1,000
			Base	TPU v2-8	500
			Large	TPU v2-8	200
Code Comment Generation	Java	470,451	Small	TPU v3-8	750,000
			Base	TPU v3-8	80,000
			Large	TPU v3-8	60,000
Git Commit Message Generation	Java	26,208	Small	TPU v2-8	5,000
			Base	TPU v2-8	2,000
			Large	TPU v2-8	4,500
API Sequence Recommendation	Java	7,475,850	Small	TPU v2-8	1,400,000
			Base	TPU v3-8	340,000
			Large	TPU v3-8	180,000
Program Synthesis	DSL	79,214	Small	TPU v3-8	5,000
			Base	TPU v2-8	45,000
			Large	TPU v2-8	3,500

Table 5.3.: The transfer learning fine-tuning experiment setups for each task. We list here the hardware, and the fine-tuning step for the small and base model with the best performance on each task.

Score\Step	100	200	300	400	**500**	600	700	800	900	1000
Bleu	1.625	1.981	1.973	2.015	**2.095**	2.068	1.761	1.482	1.225	1.215
Rouge1	17.33	19.04	20	18.15	**19.29**	18.82	18.73	18.39	16.76	16.7
Rouge2	3.85	4.38	4.41	3.73	**4.22**	4.07	3.93	3.4	3.19	2.98
RougeLsum	16.11	17.38	18.18	16.45	**17.54**	17.13	16.83	16.6	15.38	15.22

Table 5.4.: Evalutation on the validation set of the Source Code Summarization SQL task for fine-tuning 1,000 steps using the interval of 100. We evaluated using the T5 in-built Bleu and Rouge scores here.

validation set. It shows that our model achieved the best performance exactly after fine-tuning 500 steps. The model has been overfitting afterward, and the scores on the validation set decreased.

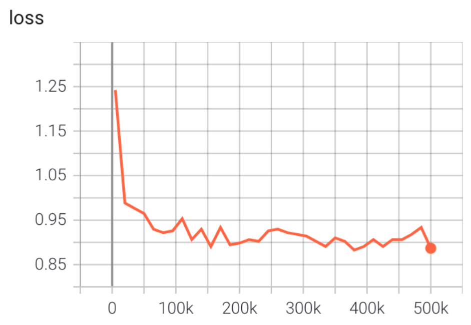
5.4. Multi-task Learning

Multi-task learning trains a single model on a mixture of tasks. We trained 13 supervised tasks together with all the unsupervised tasks using the T5 framework. The unsupervised tasks are desired to help the model gain information about the language attributes and build a language model in the software development domain. Simultaneously, the supervised tasks assist each other in making the model more generalized for all the tasks and avoid overfitting on each specific task.

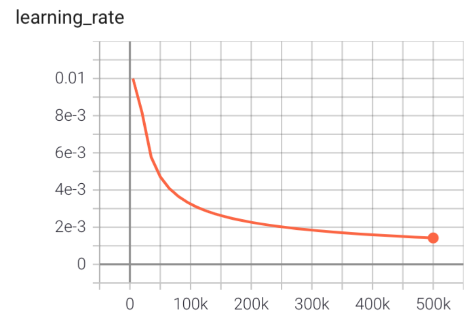
We used examples-proportional mixing to select samples in proportion to the size of each task's dataset and concatenated them. We recorded the model checkpoint every 20,000 training steps. The batch size is 4096. Usually, all the tasks should share one same best performance checkpoint. T5 paper proposed a way to relax this goal and select a different checkpoint for each task. We also evaluated the model on the validation set and selected the best checkpoint for each task. So each task could have a different checkpoint from the same model. We investigated the T5 small, base, and large models. We utilized only TPU v3-8 to train these models for multi-task learning.

We trained the T5 small model for 500,000 steps and collected 25 checkpoints. The whole training took more than 17 days. Figure 5.4 illustrates the change of loss in the training set and the change of the model learning rate. The training loss went down very fast at first and reached 0.9 at around 150,000 steps. The loss changed slightly around 0.9 during further training. The final loss is 0.887 after training half a million steps. The multi-task learning small model's loss change curve is smoother than the transfer learning small model.

We trained the T5 base model for 500,000 steps and collected 25 checkpoints. The whole training took more than 52 days. Figure 5.5 illustrates the change of loss in the training set and the change of the model learning rate. The training loss reached 0.6 at around 100,000 steps. Since then, the loss changed slightly around 0.6 but still had a tiny trend to decay.

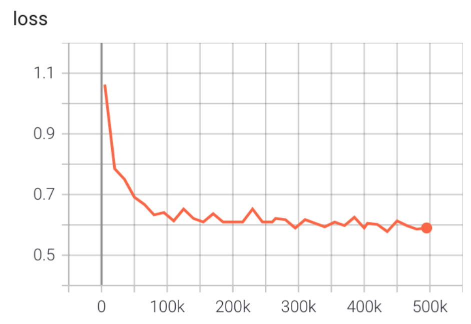


(a) The loss change of T5 small model in multi-task learning pre-training steps. The y-axis shows the loss and the x-axis lists the training steps.

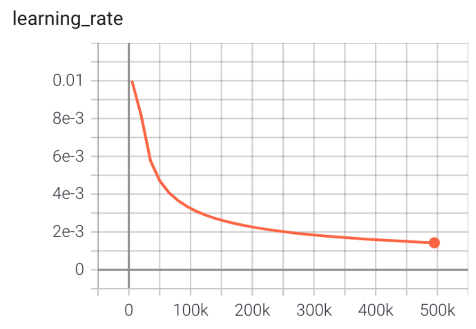


(b) The learning rate change of T5 small model in multi-task learning pre-training steps. The y-axis shows the learning rate and the x-axis lists the training steps.

Figure 5.4.: The development of Loss and learning rate of the T5 small model during the multi-task learning.



(a) The loss change of T5 base model in multi-task learning pre-training steps. The y-axis shows the loss and the x-axis lists the training steps.

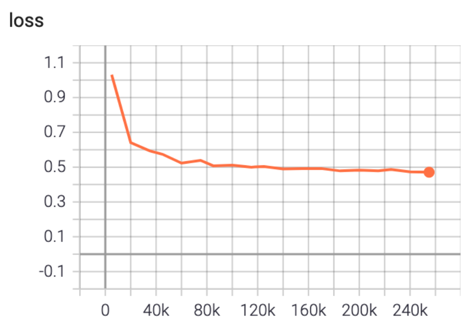


(b) The learning rate change of T5 base model in multi-task learning pre-training steps. The y-axis shows the learning rate and the x-axis lists the training steps.

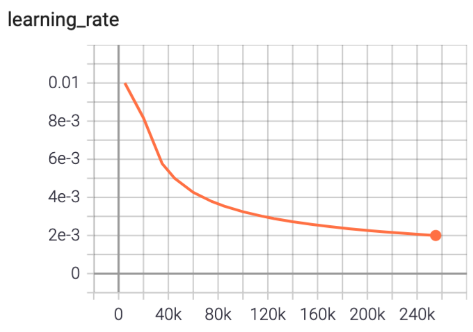
Figure 5.5.: The development of Loss and learning rate of the T5 base model during the multi-task learning.

The final loss is 0.590 after training 500,000 steps. The multi-task learning base model's loss change curve is also smoother than the transfer learning base model.

We trained the T5 large model for 260,000 steps and collected 13 checkpoints because of this thesis's time constraint. The whole training took more than 86 days. Figure 5.6 illustrates the change of loss in the training set and the change of the model learning rate. The training loss went down very fast at the first 20,000 steps. It reached 0.5 at around 80,000 steps. Since then, the loss decayed slightly and slowly. It did not have much fluctuation like the small and base multi-task learning models. The final loss is 0.4707 after training 260,000 steps.



(a) The loss change of T5 large model in multi-task learning pre-training steps. The y-axis shows the loss and the x-axis lists the training steps.



(b) The learning rate change of T5 large model in multi-task learning pre-training steps. The y-axis shows the learning rate and the x-axis lists the training steps.

Figure 5.6.: The development of Loss and learning rate of the T5 large model during the multi-task learning.

Task	Language	Sample Size	Steps in Small Model	Steps in Base Model	Steps in Large Model
Code Documentation Generation	Python	251,820	420,000	420,000	80,000
	Java	164,923	400,000	480,000	180,000
	Go	167,288	340,000	340,000	60,000
	Php	241,241	420,000	360,000	240,000
	Ruby	24,927	420,000	160,000	80,000
	Javascript	58,023	500,000	440,000	120,000
Source Code Summarization	Python	12,004	300,000	260,000	80,000
	Csharp	52,943	300,000	160,000	120,000
	SQL	25,671	460,000	500,000	120,000
Code Comment Generation	Java	470,451	360,000	460,000	260,000
Git Commit Message Generation	Java	26,208	280,000	480,000	220,000
API Sequence Recommendation	Java	7,475,850	500,000	480,000	240,000
Program Synthesis	DSL	79,214	300,000	360,000	220,000

Table 5.5.: The best multi-task learning checkpoints (training steps) for different tasks in the small and base model.

Table 5.5 lists the best training steps we chose for each task regarding the checkpoint performance on the validation sets. For the T5 small model, these checkpoints are collected in the latter half of the multi-task learning and distributed from 280,000 to 500,000 training steps. Like the small model, the base model also has its best checkpoints in the last 100,000 training steps for more than half of the tasks. This could indicate that these two models still improved in the latter part of the training. However, two of the tasks achieve their best performance for the base model in 160,000 training steps. So the best checkpoint for each task could vary a bit. For the large model, most tasks from Code Documentation Generation and Source Code Summarization achieve the best performance during the first 120,000 steps. Tasks with a large dataset like Code Comment Generation and API Sequence Recommendation need more steps to have the best performance. So the best checkpoints for these tasks lie at the end of pre-training. If we train the large model using multi-task learning with more steps, the large model could better perform the tasks.

5.5. Multi-task Learning with Fine-tuning

We fine-tuned each supervised task separately based on the multi-task learning checkpoints of 500,000 steps for the small and base model, and the checkpoint of 260,000 steps for the large model. Like the transfer learning fine-tuning, we chose the batch size of 256 and applied the early stopping to determine the fine-tuning steps based on the models' performance on the validation sets.

We used the TPU v3-8s mainly for the Code Comment Generation and the API Sequence Recommendation tasks. These two tasks have enormous datasets and consumed the longest training time when carrying out the transfer learning fine-tuning. The rest of the tasks were trained mostly using the TPU v2-8s. For small and base models, We set the fine-tuning step interval to 2,000 for most of the tasks, while we adjusted this value to 500 for the Source Code Summarization, and 20,000 for the Code Comment Generation and API Sequence Recommendation tasks. We reduced this value to 500 for the majority of tasks for the large models, 100 for the Source Code Summarization, 5,000 for the Code Comment Generation task, and 10,000 for the API Sequence Recommendation task.

Table 5.6 lists the small, base, and large models' fine-tuning steps to reach their best performance for each task. Most of the tasks reach the best performance directly after fine-tuning the step interval (e.g., 500 steps or 2,000 steps) once. However, the number of fine-tuning steps varies significantly among the different sizes of models for tasks with large datasets. For such tasks with large datasets, the smaller the model is, the more steps fine-tuning requires. The Code Comment Generation task large model needs 25,000 steps to have the best performance. The base model requires double the steps (60,000), and the small model requires 30 times more fine-tuning steps (750,000). This situation also happens in the API Sequence Recommendation task. In this way, although running one iteration would take more time for larger models, the less fine-tuning steps would still help a large model reach the best performance with less time in total.

Task	Language	Sample Size	Model Size	Hardware	MT-FT-Steps
Code Documentation Generation	Python	251,820	Small	TPU v2-8	4,000
			Base	TPU v2-8	4,000
			Large	TPU v2-8	500
	Java	164,923	Small	TPU v2-8	2,000
			Base	TPU v2-8	2,000
			Large	TPU v2-8	500
	Go	167,288	Small	TPU v2-8	2,000
			Base	TPU v2-8	2,000
			Large	TPU v2-8	4,500
	Php	241,241	Small	TPU v2-8	2,000
			Base	TPU v3-8	5,000
			Large	TPU v2-8	8,000
	Ruby	24,927	Small	TPU v2-8	2,000
			Base	TPU v2-8	12,000
			Large	TPU v2-8	2,000
	Javascript	58,023	Small	TPU v2-8	32,000
			Base	TPU v3-8	10,000
			Large	TPU v2-8	2,500
Source Code Summarization	Python	12,004	Small	TPU v2-8	600
			Base	TPU v2-8	1,000
			Large	TPU v3-8	100
	Csharp	52,943	Small	TPU v2-8	1,200
			Base	TPU v2-8	500
			Large	TPU v3-8	100
	SQL	25,671	Small	TPU v2-8	1,200
			Base	TPU v2-8	500
			Large	TPU v3-8	100
Code Comment Generation	Java	470,451	Small	TPU v3-8	750,000
			Base	TPU v3-8	60,000
			Large	TPU v3-8	25,000
Git Commit Message Generation	Java	26,208	Small	TPU v2-8	8,000
			Base	TPU v2-8	16,000
			Large	TPU v2-8	3,000
API Sequence Recommendation	Java	7,475,850	Small	TPU v3-8	1,150,000
			Base	TPU v3-8	320,000
			Large	TPU v3-8	130,000
Program Synthesis	DSL	79,214	Small	TPU v2-8	16,000
			Base	TPU v2-8	30,000
			Large	TPU v2-8	2,000

Table 5.6.: The multi-task learning fine-tuning experiment setups for each task. We listed here the hardware, and the fine-tuning steps for the best performance of the small, base and large models for each task.

6. Evaluation Results and Discussion

This chapter presents our experiment results for all the tasks using single-task learning, transfer learning, multi-task learning, and fine-tuning. We call our model CodeTrans because the model is based on the Transformer architecture. We compare our CodeTrans results with the baseline and discuss the reasons impacting the results.

6.1. Evaluation Results

In this section, we list the evaluation results of all the CodeTrans models for each task. The models' performance on the validation set can be found in Appendix A. We choose the models we explained in Section 3.2 as our baseline. The performance of most of these models can be considered as the state-of-the-art performance for that specific task. We estimated our models' final performance on the test set. We use the same metric script for evaluation as the baseline models to compare the model performances. Additionally, we also apply the T5 built-in BLEU and ROUGE metrics to get more insights into the results.

6.1.1. Code Documentation Generation

We present the results of six Code Documentation Generation tasks separately.

Table 6.1 compares different model performances on the Code Documentation Generation - Python task. We can see that most of our transfer learning, multi-task learning, and multi-task learning fine-tuning models outperform the state-of-the-art model CodeBERT in this task. Among them, the **CodeTrans base model with the multi-task learning** strategy achieves the best result and has more than one percent higher Smoothed BLEU score than CodeBERT.

Table 6.2 lists evaluation results of the Code Documentation Generation - Java task. Our CodeTrans models with transfer learning, multi-task learning, and multi-task learning with fine-tuning all outperform the baseline CodeBert model. Among them, **CodeTrans multi-task learning large model** performs best and outperforms CodeBERT by more than four percent in Smoothed BLEU. CodeTrans multi-task learning fine-tuning large model has the highest score in T5 built-in BLEU metric. Nevertheless, the multi-task learning base and large models' performances are very similar to the performances of the multi-task learning fine-tuning base and large models on all the scores.

Model	Model Size	Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	17.31	5.92	30.91	10.60	28.80
	Base	16.86	6.97	29.51	9.89	27.31
CodeTrans Transfer Learning	Small	19.93	7.38	35.96	14.09	33.71
	Base	20.26	7.83	36.44	14.66	34.15
	Large	20.35	7.41	36.33	14.59	34.16
CodeTrans Multi-task Learning	Small	19.64	7.12	35.45	13.71	33.2
	Base	**20.39**	**7.99**	**36.82**	**14.82**	**34.34**
	Large	20.18	7.94	36.72	14.53	34.25
CodeTrans Multi-task Learning Fine-tuning	Small	19.77	7.58	35.74	13.91	33.37
	Base	19.77	7.83	35.81	14.04	33.39
	Large	18.94	7.30	35.22	13.42	32.75
CodeBERT		19.06	-	-	-	-

Table 6.1.: The evaluation results for the task Code Documentation Generation - Python. We compare the CodeTrans performance using different training strategies with the state-of-the-art CodeBERT.

Table 6.3 shows evaluation results of the Code Documentation Generation - Go task. The **CodeTrans transfer learning large model** works best and outperforms CodeBERT by more than one percent. The CodeTrans multi-task learning fine-tuning model has the highest score on the T5 built-in BLEU metric. Regarding the ROUGE-L, the large model of multi-task learning performs best. The difference among the different sizes of CodeTrans models is minimal for this task.

The evaluation results of the Code Documentation Generation - Php task are shown in Table 6.4. The **CodeTrans multi-task learning base model** works best and outperforms CodeBERT by more than one percent on the Smoothed BLEU score. Regarding the T5 built-in BLEU score, the multi-task learning fine-tuning large model performs best. The multi-task learning base model works better than the large model. However, if we could train the multi-task large model longer, we may achieve a better result.

The Code Documentation Generation - Ruby task's evaluation results are listed in Table 6.5. The CodeTrans transfer learning, multi-task learning, and multi-task learning fine-tuning models all outperform the CodeBERT. The **multi-task learning base model** also performs best on four metrics and has a three percent better score than the CodeBert model. The transfer-learning large model has the best performance on the T5 built-in BLEU metric.

Table 6.6 presents evaluation results of the Code Documentation Generation - Javascript task. The **CodeTrans transfer learning large model** outperforms the CodeBERT by more than four percent on the smoothed BLEU score. The multi-task learning fine-tuning large model shows a similar performance and achieves the best score on ROUGE-1 and ROUGE-L. Moreover, the multi-task learning fine-tuning base model has the best performance on the T5 built-in BLEU metric. However, the three multi-task learning models have extremely low scores using the

Model	Model Size	Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	16.65	8.60	31.22	12.21	28.95
	Base	17.17	8.92	30.90	12.24	28.74
CodeTrans Transfer Learning	Small	19.48	8.39	36.33	15.91	34.02
	Base	20.19	8.44	36.36	16.43	34.17
	Large	20.06	7.92	36.79	16.51	34.54
CodeTrans Multi-task Learning	Small	19.00	7.20	35.73	15.25	33.51
	Base	21.22	9.93	37.98	17.99	35.80
	Large	**21.87**	12.04	**38.60**	**18.75**	**36.29**
CodeTrans Multi-task Learning Fine-tuning	Small	20.04	7.90	36.37	16.30	34.28
	Base	21.12	9.99	37.86	17.81	35.67
	Large	21.42	**12.46**	38.44	18.49	35.99
CodeBERT		17.65	-	-	-	-

Table 6.2.: The evaluation results for the task Code Documentation Generation - Java. We compare the CodeTrans performance using different training strategies with the state-of-the-art CodeBERT.

Model	Model Size	Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	16.89	5.98	37.14	14.27	35.58
	Base	17.16	9.41	37.41	14.49	35.43
CodeTrans Transfer Learning	Small	18.88	8.60	41.29	17.29	39.23
	Base	19.50	9.52	42.09	18.07	39.86
	Large	**19.54**	9.89	**42.43**	**18.51**	40.29
CodeTrans Multi-task Learning	Small	19.15	7.83	41.90	17.83	39.69
	Base	19.43	9.06	41.94	18.24	39.98
	Large	19.38	8.41	42.20	18.50	**40.33**
CodeTrans Multi-task Learning Fine-tuning	Small	19.36	8.19	41.99	18.32	39.99
	Base	18.86	8.00	41.31	17.52	39.42
	Large	18.77	**10.81**	41.04	16.90	38.73
CodeBERT		18.07	-	-	-	-

Table 6.3.: The evaluation results for the task Code Documentation Generation - Go. We compare the CodeTrans performance using different training strategies with the state-of-the-art CodeBERT.

Model	Model Size	Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	23.05	12.23	37.10	15.23	35.02
	Base	22.98	12.27	36.60	14.98	34.64
CodeTrans Transfer Learning	Small	25.35	10.45	42.04	17.77	39.96
	Base	25.84	14.51	41.61	18.64	39.37
	Large	26.18	14.06	42.29	18.92	40.21
CodeTrans Multi-task Learning	Small	24.68	9.23	41.11	17.06	39.08
	Base	**26.23**	10.85	**43.07**	**19.18**	**41.00**
	Large	26.08	11.50	42.63	18.69	40.53
CodeTrans Multi-task Learning Fine-tuning	Small	25.55	9.20	42.19	17.62	40.24
	Base	25.79	10.83	41.65	18.07	39.65
	Large	26.20	**15.11**	42.44	19.39	40.17
CodeBERT		25.16	-	-	-	-

Table 6.4.: The evaluation results for the task Code Documentation Generation - Php. We compare the CodeTrans performance using different training strategies with the state-of-the-art CodeBERT.

Model	Model Size	Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	9.19	2.12	15.54	3.00	14.47
	Base	8.23	2.08	13.16	2.39	12.23
CodeTrans Transfer Learning	Small	13.15	3.78	26.09	8.07	23.99
	Base	14.07	4.70	28.12	9.35	25.73
	Large	14.94	**5.52**	29.10	10.68	26.90
CodeTrans Multi-task Learning	Small	14.91	3.62	29.00	10.04	27.04
	Base	**15.26**	4.48	**30.28**	**11.26**	**28.21**
	Large	15.00	4.15	29.81	10.74	27.64
CodeTrans Multi-task Learning Fine-tuning	Small	13.70	3.84	26.81	8.42	24.65
	Base	14.24	5.25	28.33	9.37	25.85
	Large	14.19	5.35	28.03	9.77	25.89
CodeBERT		12.16	-	-	-	-

Table 6.5.: The evaluation results for the task Code Documentation Generation - Ruby. We compare the CodeTrans performance using different training strategies with the state-of-the-art CodeBERT.

Model	Model Size	Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	13.70	9.43	20.94	7.31	19.50
	Base	13.17	10.13	18.82	7.19	17.75
CodeTrans Transfer Learning	Small	17.23	12.60	28.52	11.44	26.48
	Base	18.25	14.39	30.34	13.21	28.23
	Large	**18.98**	14.08	31.58	**13.75**	29.26
CodeTrans Multi-task Learning	Small	15.26	3.00	27.63	9.13	25.88
	Base	16.11	3.52	29.34	10.29	27.53
	Large	16.23	4.36	30.05	10.94	28.11
CodeTrans Multi-task Learning Fine-tuning	Small	17.24	12.94	28.55	11.78	26.54
	Base	18.62	**14.58**	30.97	13.61	28.96
	Large	18.83	14.56	**31.82**	13.71	**29.52**
CodeBERT		14.90	-	-	-	-

Table 6.6.: The evaluation results for the task Code Documentation Generation - Javascript. We compare the CodeTrans performance using different training strategies with the state-of-the-art CodeBERT.

T5 built-in BLEU metric. Factors like the output length could influence this evaluation metric. So it is worthwhile to have multiple metrics when comparing results.

6.1.2. Source Code Summarization

We present the results of three Source Code Summarization tasks separately.

		Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	8.45	1.05	15.23	2.55	13.63
	Base	9.12	1.62	16.58	3.80	15.08
CodeTrans Transfer Learning	Small	10.06	1.53	18.97	3.72	16.77
	Base	10.94	2.22	21.44	4.33	18.71
	Large	12.41	2.17	23.54	5.32	20.71
CodeTrans Multi-task Learning	Small	13.11	3.60	26.85	7.46	23.70
	Base	**13.37**	**4.48**	**27.81**	**8.05**	**24.64**
	Large	13.24	4.16	27.57	7.88	24.30
CodeTrans Multi-task Learning Fine-tuning	Small	12.10	2.89	23.92	5.65	21.17
	Base	10.64	2.11	21.07	4.29	18.25
	Large	12.14	2.85	23.73	5.94	20.95

Table 6.7.: The evaluation results for the task Source Code Summarization - Python. We compare the CodeTrans performance among different training strategies.

Table 6.7 shows the results for Source Code Summarization - Python. Because the CODE-NN did not provide the evaluation on this Python task, so we compare the results among different CodeTrans training strategies. From the table, we can observe that the small, base and large CodeTrans multi-task learning models perform better than the rest of the CodeTrans models. The **multi-task learning base model** achieves better results than the small and the large models over all the metrics.

		Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	19.74	2.80	20.02	4.61	18.29
	Base	18.65	2.69	20.55	4.69	18.69
CodeTrans Transfer Learning	Small	20.40	3.60	22.88	5.93	20.98
	Base	21.12	3.84	23.27	6.18	21.36
	Large	21.43	4.02	23.68	6.40	21.89
CodeTrans Multi-task Learning	Small	22.39	3.74	23.43	6.16	21.34
	Base	23.20	4.23	**24.71**	6.65	22.50
	Large	**23.57**	**4.39**	**24.71**	**6.90**	**22.62**
CodeTrans Multi-task Learning Fine-tuning	Small	22.03	3.60	22.67	5.93	20.84
	Base	21.40	4.20	24.33	6.56	22.18
	Large	21.10	3.68	22.84	5.65	20.67
CODE-NN		20.50	-	-	-	-

Table 6.8.: The evaluation results for the task Source Code Summarization - CSharp. We compare the CodeTrans performance using different training strategies with the state-of-the-art CODE-NN.

Table 6.8 presents the evaluation results for the Source Code Summarization - Csharp task. We compare the CodeTrans performances with the CODE-NN as our baseline. **CodeTrans multi-task learning large model** outperforms the CODE-NN by more than three percent on this task. It also outperforms other CodeTrans models. The multi-task learning base model has similar good performance and achieves the same highest scores on the ROUGE-1 metric.

		Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	17.55	1.54	18.10	3.57	16.36
	Base	15.00	1.28	16.27	2.86	14.51
CodeTrans Transfer Learning	Small	17.71	1.75	18.69	3.86	17.06
	Base	17.66	2.25	19.94	4.41	17.97
	Large	18.40	2.17	**20.37**	4.27	18.20
CodeTrans Multi-task Learning	Small	19.15	1.95	19.05	4.20	17.03
	Base	19.24	2.10	19.53	4.10	17.65
	Large	19.49	**2.29**	20.34	**4.49**	**18.21**
CodeTrans Multi-task Learning Fine-tuning	Small	18.25	1.88	18.77	3.97	16.97
	Base	16.91	1.95	19.42	3.98	17.57
	Large	**19.98**	1.97	17.48	4.03	16.26
CODE-NN		18.40	-	-	-	-

Table 6.9.: The evaluation results for the task Source Code Summarization - SQL. We compare the CodeTrans performance using different training strategies with the state-of-the-art CODE-NN.

Table 6.9 shows the models' performance on the Source Code Summarization - SQL task. For this task, the **multi-task learning fine-tuning large model** performs best on the smoothed BLEU score and has more than 1.5 percent than the baseline CODE-NN. The multi-task learning large model has the second-highest score on the smoothed BLEU and outperforms all other models on the T5 built-in BLEU, ROUGE-2, and ROUGE-l metrics. The CodeTrans large transfer-learning model has the highest score on ROUGE-1. Since the smoothed BLEU only took 100 samples from the test dataset, the other metrics consider the whole test dataset, so the CodeTrans multi-task learning large model's performance should be better in general.

6.1.3. Code Comment Generation

The evaluation results of the Code Comment Generation task are presented in Table 6.10. We compare the CodeTrans with the DeepCom model as our baseline. In total, the **CodeTrans transfer learning large model** achieves the best 39.50 smoothed BLEU score among all the models, which is also higher than the state-of-the-art DeepCom by more than one percent. The multi-task learning fine-tuning large model achieves a similar manner of performance and has the highest score for the T5 built-in BLEU metric. For this task, the small, base, and large multi-task learning models perform even worse than those that only applied single-task learning. But the improvement is huge for the multi-task learning when the model size increases.

		Smoothed BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	37.98	36.05	46.61	34.83	45.47
	Base	38.07	36.79	46.77	35.06	45.62
CodeTrans Transfer Learning	Small	38.56	36.31	47.93	35.56	46.71
	Base	39.06	37.38	48.95	36.34	47.66
	Large	**39.50**	37.86	**49.68**	**37.07**	**48.37**
CodeTrans Multi-task Learning	Small	20.15	11.97	34.23	17.38	32.78
	Base	27.44	19.96	40.61	25.25	39.21
	Large	34.69	30.74	46.21	32.53	44.83
CodeTrans Multi-task Learning Fine-tuning	Small	38.37	36.81	47.79	35.59	46.58
	Base	38.90	37.60	48.95	36.38	47.57
	Large	39.25	**38.54**	49.21	36.76	47.91
DeepCom		38.17	-	-	-	-

Table 6.10.: The evaluation results for the task Code Comment Generation. We compare the CodeTrans performance using different training strategies with the state-of-the-art DeepCom.

6.1.4. Git Commit Message Generation

Table 6.11 shows the evaluation results for the task Git Commit Message Generation. We did not apply the smoothed BLEU score for this task because the baseline NMT model was only evaluated using the BLEU metric. All our CodeTrans models outperform the baseline model. The performance of models used transfer learning and multi-task learning fine-tuning are quite similar. The **CodeTrans transfer learning large model** achieves the best BLEU score as

44.41. The CodeTrans multi-task learning fine-tuning large model has the best performance on the ROUGE-2 score.

		BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	39.61	37.91	28.89	37.69
	Base	38.67	37.77	28.30	37.59
CodeTrans Transfer Learning	Small	44.22	47.05	34.74	46.57
	Base	44.17	47.41	35.14	46.84
	Large	**44.41**	**48.36**	35.66	**47.76**
CodeTrans Multi-task Learning	Small	36.17	38.15	25.38	37.84
	Base	39.25	43.71	29.90	43.32
	Large	41.18	46.36	32.42	45.86
CodeTrans Multi-task Learning Fine-tuning	Small	43.96	47.39	35.00	46.94
	Base	44.19	47.96	35.61	47.43
	Large	44.34	48.08	**35.75**	47.52
NMT		32.81	-	-	-

Table 6.11.: The evaluation results for the task Git Commit Message Generation. We compare the CodeTrans performance using different training strategies with the state-of-the-art NMT.

6.1.5. API Sequence Recommendation

Table 6.12 presents the evaluation results of the task API Sequence Recommendation. We compare our results with the DeepAPI model as our baseline. We applied the same BLEU metric script as the DeepAPI used, in addition to the T5 built-in BLEU and ROUGE scripts. All the CodeTrans models outperform the DeepAPI model. Among the CodeTrans models, those trained using only multi-task learning perform worst. The **CodeTrans large model with multi-task learning fine-tuning** has the highest scores across all the models. The CodeTrans transfer learning large model also has a similarly good performance.

6.1.6. Program Synthesis

Table 6.13 presents the evaluation results. We evaluated our models using the absolute Accuracy and comparing them with the code Accuracy of the baseline model Seq2Tree. If the absolute Accuracy is high, then the model could definitely achieve a high code Accuracy. Nine out of Ten CodeTrans models outperform the Seq2Tree model. The **CodeTrans multi-task learning fine-tuning small model** achieves the best score on Accuracy. The CodeTrans transfer learning small model performs best on T5 built-in BLEU and ROUGE scores. For these two training strategies, smaller models perform better. For multi-task learning, the performance increases along with the model size. This task's scores are very high, which means that this is an easy task with very similar validation and test sets, and bigger models may easy to be overfitted.

		DeepAPI BLEU	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	68.71	70.92	77.40	68.72	77.37
	Base	70.45	72.32	79.11	70.55	79.09
CodeTrans Transfer Learning	Small	68.90	70.85	77.80	68.91	77.76
	Base	72.11	73.65	80.64	72.43	80.65
	Large	73.26	74.38	81.67	73.69	81.69
CodeTrans Multi-task Learning	Small	58.43	59.69	67.22	56.97	67.19
	Base	67.97	69.82	76.72	67.66	76.70
	Large	72.29	73.55	80.82	72.50	80.76
CodeTrans Multi-task Learning Fine-tuning	Small	69.29	71.31	78.01	69.18	78.02
	Base	72.89	74.16	81.32	73.20	81.33
	Large	**73.39**	**74.53**	**81.80**	**73.81**	**81.78**
DeepAPI		54.42	-	-	-	-

Table 6.12.: The evaluation results for the task API Sequence Recommendation. We compare the CodeTrans performance using different training strategies with the state-of-the-art DeepAPI.

		Accuracy	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
CodeTrans Single-Task Learning	Small	89.43	94.62	98.93	98.21	98.60
	Base	89.65	94.64	99.04	98.43	98.72
CodeTrans Transfer Learning	Small	90.30	**94.73**	**99.30**	**98.62**	**98.84**
	Base	90.24	94.72	99.13	98.59	98.82
	Large	90.21	94.72	99.13	98.61	98.50
CodeTrans Multi-task Learning	Small	82.88	94.03	98.58	97.57	98.25
	Base	86.99	94.52	98.97	98.25	98.68
	Large	90.27	94.71	99.12	98.62	98.83
CodeTrans Multi-task Learning Fine-tuning	Small	**90.31**	94.72	99.12	98.60	98.82
	Base	90.30	94.72	99.12	98.61	98.83
	Large	90.17	94.70	99.12	98.56	98.81
Seq2Tree		85.80	-	-	-	-

Table 6.13.: The evaluation results for the task Program Synthesis. We compare the CodeTrans performance using different training strategies with the state-of-the-art Seq2Tree.

6.2. Discussion

Our CodeTrans models with the encoder-decoder transformer architecture outperform the baseline models in all the thirteen tasks. This proves the effectiveness of the transformer encoder-decoder architecture for these tasks in the software development domain. Table 6.14 shows one example of the outputs from all CodeTrans models with the input javascript code and the golden reference. From this table, we can see all the models give readable outputs and these outputs are highly similar to the golden reference. The examples for other tasks are listed in the Appendix B.

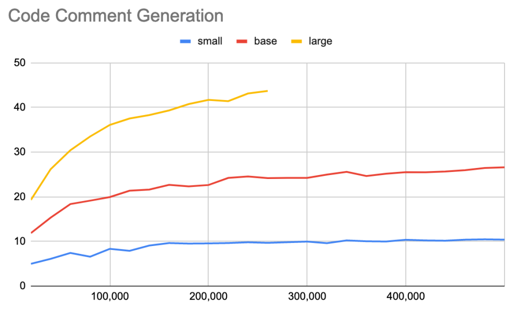
Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Returns true if the browser is a native element .
	Base	Returns whether the givenEnv should be focused .
CodeTrans Transfer Learning	Small	Checks if the current browser is on a standard browser environment .
	Base	Check if browser environment is a standard browser environment
	Large	Check if the environment is standard browser .
CodeTrans Multi-task Learning	Small	Returns true if the browser environment is a standard browser environment .
	Base	Checks if the current browser environment is a standard browser environment .
	Large	Determines if the current environment is a standard browser environment
CodeTrans Multi-task Learning Fine-tuning	Small	Standard browser environment has a notion of what React Native does not support it .
	Base	Check if the browserEnv is standard .
	Large	Checks if the browser is in a standard environment .
Code Snippet as Input		function isStandardBrowserEnv ( ) { if (typeof navigator !== 'undefined' && (navigator . product === 'ReactNative' | | navigator . product === 'NativeScript' | | navigator . product === 'NS')) { return false ; } return (typeof window !== 'undefined' && typeof document !== 'undefined') ; }
Golden Reference		Determine if we re running in a standard browser environment

Table 6.14.: The models' output for an example of the task Code Documentation Generation - Javascript.

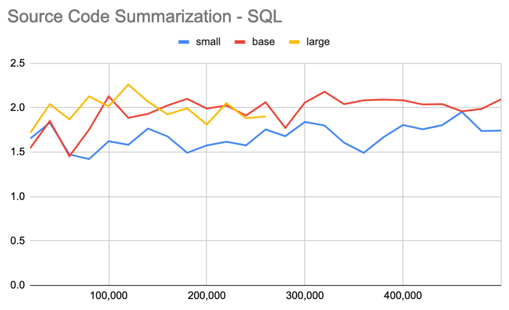
Nevertheless, the models' performance varies a bit when using different training strategies for different sizes of models on different datasets.

We have noticed that the **model size** plays an essential role in the model's performance. For single-task learning, the larger the dataset is, the fewer training steps a bigger model requires. A bigger model reaches a lower loss under the same batch size and the same evaluation steps when applying the multi-task learning or transfer learning strategy. Although the pre-training may cost more time for bigger models, they need fewer iteration steps during fine-tuning for each task than the small models. As a result, for most of the tasks, the bigger the model is, the better evaluation scores the model could achieve with even less fine-tuning time.

The evaluation results also prove that **transfer learning** and **multi-task learning with fine-tuning** strategies outperform the models that only used **single task learning** on all the tasks. The performance of models using transfer learning is very similar to those using multi-task learning fine-tuning. It is hard to say which one is better. However, transfer learning does not require the task dataset to be involved in the pre-training steps. For a new task, the dataset only needs to be trained for relatively few fine-tuning steps, while multi-task learning with fine-tuning needs the new task dataset during pre-training. We can say that transfer learning would save many training steps and times for a new task when only fine-tuning on a pre-trained model checkpoint.



(a) The Code Comment Generation task's training dataset has 470,486 samples.



(b) The Source Code Summarization - SQL task's training dataset has 22,492 samples.

Figure 6.1.: The evaluation of multi-task learning checkpoints on the validation set for two tasks. The x-axis lists the training steps. The y-axis is the T5 built-in BLEU score. Different colors indicate different sizes of models.

The performance of **multi-task learning** depends highly on the **data size** and **attributes** of the task itself. Figure 6.1 illustrates the small, base, and large models' performance on the validation sets for two different kinds of datasets during the multi-task learning. For **large datasets** like the dataset for the task Code Comment Generation and API Sequence Recommendation, multi-task learning models are even worse than the models that only applied single-task learning. Figure 6.1a shows that the model's performance improves a lot when we increase the model size for the Code Comment Generation task with a large dataset. Half a million multi-task training steps are not enough for this task, even using the large model. When the dataset is **tiny and easy to be overfitted**, multi-task learning could achieve the best result, and a bigger model does not lead to a certain better performance. In Figure 6.1b, we can see that the base model performs overall better than the small model for the source code summarization - SQL task, but the large model has several overlaps with the base model. The large model has a sign of overfitting after 120,000 training steps, and the model performance decreases since then.

Table 6.15 lists the outputs of each CodeTrans models comparing with the golden reference extracted from the StackOverflow. The input for the models is "select time (col0) from tab0". We can observe that all the models' outputs are readable sentences. The majority of them have a question format. Because the dataset contains questions and answers from StackOverflow, the models have learned how to ask a question. Outputs from the models with only single-task learning did not make much sense. The other outputs all notice this code is about *time*. All the multi-task learning models also specify the *mysql* database system. The CodeTrans multi-task learning large model mentions the keyword *datetime*, which also appears in the golden reference. Besides, the transfer learning and multi-task learning fine-tuning base models have reasonable outputs as well. The CodeTrans transfer learning and multi-task

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	mysql : how to get the difference of a column in a table ?
	Base	how do i get the average of a date range in sql server 2005 ?
CodeTrans Transfer Learning	Small	how to get the time in milliseconds since the start time of the transaction was taken ?
	Base	how to get current date time in sql server ?
	Large	mysql time ( ) function
CodeTrans Multi-task Learning	Small	how to get the time in mysql ?
	Base	how can i get the time of a date in mysql ?
	Large	how to convert datetime to time in mysql ?
CodeTrans Multi-task Learning Fine-tuning	Small	how to get the correct time from mysql database ?
	Base	how to convert date to time in mysql ?
	Large	select time from mysql table
Code Snippet as Input		select time (col0) from tab0
Golden Reference		datetime implementation in php mysql

Table 6.15.: The models' output for an example of the task Source Code Summarization. We compare different CodeTrans model outputs and the golden reference for the input SQL code "select time (col0) from tab0". The golden reference is the one extracted from the StackOverflow.

learning fine-tuning models focus more on the code function and structure to summarize this code snippet. In total, our judge for the models' performances matches the ranking of our evaluation metrics.

Moreover, most of the Code Documentation Generation tasks achieve the best evaluation performance when using the multi-task learning strategy. It could be that we have two more unsupervised tasks from the same CodeSearchNet corpus during the multi-task learning. These give more similar samples for the supervised Code Documentation Generation tasks, so the model would focus on training these tasks more. Moreover, using different types of tasks during multi-task learning also avoids overfitting efficiently.

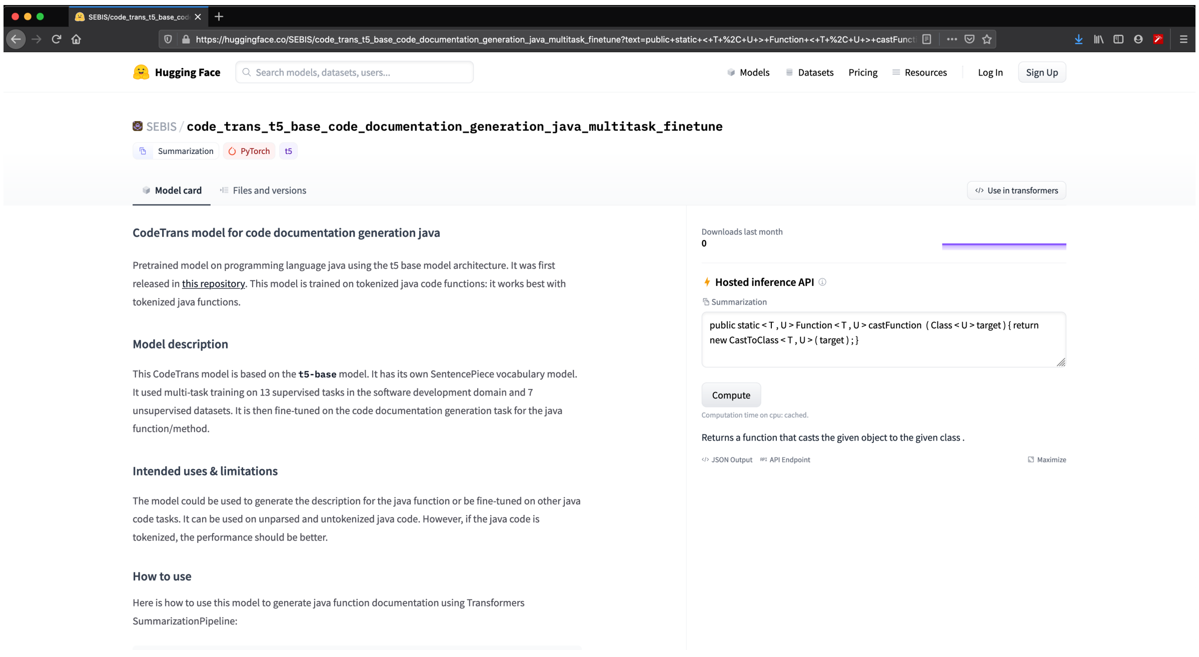
Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	11.94	11.96	11.50	11.13	11.78	11.66	**12.36**	11.83	12.07	11.79
ROUGE-1	**39.59**	38.73	37.79	37.55	37.2	36.66	37.21	37.01	36.93	36.77
ROUGE-2	**19.79**	18.73	17.6	17.39	17.09	16.69	17.02	16.79	16.83	16.57
ROUGE-L	**37.39**	36.26	35.2	35.06	34.66	34.11	34.55	34.43	34.26	34.03

Table 6.16.: The evaluation of the task Code Documentation Generation - Java on the validation set when fine-tuning the multi-task base model. The first row listed the fine-tuning steps. The rest of the rows are the scores on each step regarding different evaluation metrics.

It is worth mentioning that we chose the best checkpoint based on **different metrics** from the models' performance on validation sets. BLEU score and ROUGE scores may point to different best checkpoints. Table 6.16 shows the model performance on the validation set regarding different evaluation metrics when we fine-tuned the Code Documentation

Generation - Java task. The ROUGE scores indicate that the model has the best performance on 2,000 fine-tuning steps, while the BLEU metric achieves the highest score on 14,000 steps. We tested the model using both 2,000 and 14,000 fine-tuning checkpoints. It turns out that checkpoint 2,000 also gives higher BLEU and ROUGE scores on the test set. So it is helpful to decide the best model considering different metrics.

6.3. Models Publication


SEBIS/code_trans_t5_base_cod
https://huggingface.co/SEBIS/code_trans_t5_base_code_documentation_generation_java_multitask_finetune?text=public+static+<+T+%2C+U+>+Function+<+T+%2C+U+>+castFunct
Hugging Face
Search models, datasets, users...
Models
Datasets
Pricing
Resources
Log In
Sign Up
Summarization
PyTorch
t5
Model card
Files and versions
Use in transformers
Compute
JSON Output
API Endpoint
Maximize

SEBIS / code_trans_t5_base_code_documentation_generation_java_multitask_finetune

CodeTrans model for code documentation generation java

Pretrained model on programming language java using the t5 base model architecture. It was first released in this repository. This model is trained on tokenized java code functions: it works best with tokenized java functions.

Model description

This CodeTrans model is based on the `t5-base` model. It has its own SentencePiece vocabulary model. It used multi-task training on 13 supervised tasks in the software development domain and 7 unsupervised datasets. It is then fine-tuned on the code documentation generation task for the java function/method.

Intended uses & limitations

The model could be used to generate the description for the java function or be fine-tuned on other java code tasks. It can be used on unparsed and untokenized java code. However, if the java code is tokenized, the performance should be better.

How to use

Here is how to use this model to generate java function documentation using Transformers SummarizationPipeline:

Downloads last month
0

Hosted inference API

Summarization

```
public static < T , U > Function < T , U > castFunction  ( Class < U > target ) { return
new CastToClass < T , U > ( target ) ; }
```

Computation time on cpu: cached.

Returns a function that casts the given object to the given class .

Figure 6.2.: The multi-task learning fine-tuning base model page for the task Code Documentation Generation - Java from the Hugging Face Model Hub.

We trained the transformer architecture models for tasks in the software development domain in this thesis. These tasks are useful during the software development life cycle, and the performance of the models is very satisfying. We converted the best checkpoints and published all our models on the Hugging Face Model Hub[1], which is a platform containing the largest collection of models, datasets, and metrics about advance AI and NLP. The models are listed under the organization *Software Engineering for Business Information Systems (sebis)*[2]. Users across the world can use these models freely on their demand.

Figure 6.2 presents the website page of the multi-task learning fine-tuning base model for the task Code Documentation Generation - Java from the Hugging Face Model Hub. On the left side of the page is the model card. It describes the background information and the usage of

[1]https://huggingface.co/
[2]https://huggingface.co/SEBIS

this model. The right side of the page is the user input-output interface. By giving the code to summarize the documentation in the text box on the right side and clicking the button "Compute," the documentation generated by our model would show under the button in few seconds. These models can also be downloaded and compute batches of codes using the Python package **transformers**[3] following the description in the model card.

Besides, our transfer learning pre-trained checkpoints are suitable for fine-tuning new tasks in the software development domain, especially those involving Python, Java, Php, Javascript, Ruby, and Go, SQL, CSharp, and Lisp programming languages. Our pre-trained multi-task learning checkpoints are more fit for Code Documentation Generation tasks since more relevant tasks are involved during pre-training. Fine-tuning based on these checkpoints may save users a lot of time and improves task performance. We published these pre-trained checkpoints in our GitHub Repository[4] together with all our training datasets and the training scripts. We also built the Colab Notebooks about preprocessing the datasets and running the models for each task to guide the users using these models.

[3]`https://huggingface.co/transformers/`
[4]`https://github.com/agemagician/CodeTrans`

7. Conclusions and Future Work

In this chapter, we come to the conclusions for this master thesis, and point out the directions for the future work.

7.1. Conclusions

This thesis explores the CodeTrans models with Transformer Encoder-Decoder architecture on six main tasks and, in total, thirteen subtasks in the software development domain covering nine programming languages. We carried out experiments with different training strategies, including single-task learning, transfer learning, multi-task learning, and multi-task learning fine-tuning. We utilized different sizes of the models based on the Google Tensorflow Text-To-Text Transfer Transformer framework by applying the Nvidia GPU and Google Cloud TPUs.

Our CodeTrans models outperform all the baseline models and achieve the state-of-the-art over all the tasks. Our experiments on various tasks have provided us many insights about training a neural network model on software development relevant tasks. We find that, first of all, larger models may bring a better model performance. Secondly, models with transfer learning perform as well as models with multi-task learning fine-tuning, and the pre-training models can be fine-tuned on the new downstream tasks efficiently while saving a lot of training time. Moreover, multi-task learning is very beneficial for the small dataset on which the model will overfit easily. Finally, we also examine the effect of different metrics for the natural language generation tasks, and considering several metrics would be helpful for finding the best model checkpoint. It is also promising that these experiences can be generalized for training natural language processing tasks on different domains.

In addition to these findings, we have published our models on the internet with a friendly user interface and sufficient documentation so that everyone can access our models and use them for their purposes. We also provide the online downloading links to the pre-trained checkpoints generated from our CodeTrans transfer learning pre-training. These checkpoints are suitable for fine-tuning other tasks in the software development domain if the task's programming language is covered in this thesis.

In conclusion, during this thesis work, we gain valuable experiences in training neural network models for natural language processing, give contributions to solve the software development domain tasks, and achieve our research goals.

7.2. Future Work

We involved small, base, and large models in this thesis. However, we only trained the large model for around 250,000 steps due to lack of time, because every 20,000 steps cost almost one week for a large model. It worths continuing to train the large model and obverse the model converge and the model performance in the further iteration steps.

When working on the Code Documentation Generation tasks, we have noticed that a programming language function has two aspects influencing the model performance: the function names/parameter names and the code structure. A well-named function would lower the difficulty for the model to generate the documentation. Further researches about functions with disguised parameter names or function names would be valuable. We considered a function as a sentence during our thesis work. From this aspect, we did not fully make use of the code structure. So how to present the code is also a good research point. Experiments about finding the best way to present the features of code structure can be carried out.

We preprocessed the datasets by parsing and tokenizing the programming codes using different Python libraries for each programming language. So when using our models, applying the same preprocessing way would draw the best results. Nevertheless, not every user is a programming expert, and the preprocessing increases the complexity for users to get the best model performance. It would be meaningful to examine the effect of preprocessing for the software development tasks and train models with good performance but without preprocessing like parsing and tokenizing.

For measuring the performance of models, we only applied the objective human likeness measures of BLEU and ROUGE scores in this thesis. For examining the generated natural languages, subjective human judgments could also be applied. Human annotators could be invited to evaluate the model output and to examine the model performance considering aspects like grammaticality, correctness, or human-likeness.

Moreover, more tasks can be explored using transformer encoder-decoder architecture. It would be interesting to examine our models' performance on the unseen programming languages. Evaluation could be run directly on similar tasks with an unseen programming language using the multi-task learning CodeTrans models. The pre-trained models can also be fine-tuned on tasks with unseen programming languages and examine the model outputs.

Finally, the performance of our CodeTrans models can be the baseline for tasks introduced in this thesis. They can be used to comparing the effectiveness of new natural language processing model architectures in the future.

A. Appendix - Hyperparameter Turning and Evaluation on Validation Set

A.1. Single-task Learning

A.1.1. Code Documentation Generation

Python, small model, batch size: 256

Steps	20000	40000	60000	80000	100000
BLEU	5.646	5.852	**6.261**	6.256	6.242
ROUGE1	**30.14**	29.45	29.41	29.12	28.82
ROUGE2	**10.08**	9.62	9.61	9.57	9.37
ROUGELsum	**28.02**	27.38	27.22	27.07	26.72

Python, small model, batch size: 512

Steps	10000	20000	30000	40000	50000	60000	70000
BLEU	4.645	5.899	6.009	6.329	**6.379**	6.15	6.289
ROUGE1	28.67	**29.03**	29.13	28.8	28.86	28.59	28.79
ROUGE2	9.45	**9.59**	9.37	9.52	9.56	9.31	9.55
ROUGELsum	**27.02**	27.01	26.99	26.71	26.77	26.54	26.76

Python, small model, batch size: 1024

Steps	10000	20000	30000
BLEU	5.372	5.831	5.842
ROUGE1	28.44	28.46	28.3
ROUGE2	9.15	9.28	9.2
ROUGELsum	26.46	26.49	26.33

Python, base model, batch size: 384

Steps	7500	15000	22500	30000	37500	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
BLEU	3.361	5.213	5.594	5.929	5.856	5.949	6.0009	6.025	6.056	6.24	6.303	6.218	6.334	6.153	**6.495**	6.491	6.435
ROUGE1	27.16	27.8	27.45	28.13	27.73	27.77	**28.87**	28.43	27.83	27.93	28.03	27.95	28.69	27.97	28.52	28.25	28.18
ROUGE2	7.87	8.38	8.39	8.81	8.64	8.72	8.89	8.98	8.8	9.03	8.95	8.94	9.18	8.89	**9.24**	9.11	9.16
ROUGELsum	25.41	25.67	25.5	26.1	25.6	25.64	25.85	26.29	25.76	25.82	25.89	25.9	**26.55**	25.88	26.33	26.12	26.13

Python, base model, batch size: 512

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000
BLEU	3.405	4.366	5.202	5.334	5.561	5.641	5.879	5.76	5.935	**6.246**	5.97	6.019
ROUGE1	25.71	26.03	26.89	27.59	27.39	27.51	27.49	27.59	27.64	**27.93**	27.83	27.89
ROUGE2	6.99	7.45	7.96	8.34	8.3	8.58	8.61	8.59	8.63	**8.88**	8.74	8.8
ROUGELsum	23.95	24.24	24.91	25.55	25.27	25.55	25.47	25.52	25.54	**25.83**	25.79	25.77

Python, base model, batch size: 1024

Steps	5000	8000	11000	14000	19000	24000
BLEU	4.493	4.75	4.833	5.379	5.112	5.541
ROUGE1	26.43	26.01	25.98	25.92	26.254	26.87
ROUGE2	7.34	7.66	7.48	7.57	7.671	7.95
ROUGELsum	24.39	24.26	24.22	24.06	24.292	24.76

Java, small model, batch size: 256

Steps	20000	40000	60000	80000	100000
BLEU	6.846	8.071	**8.667**	8.465	8.476
ROUGE1	31.57	31.78	**32.01**	31.76	31.72
ROUGE2	12.05	12.31	**12.58**	12.52	12.54
ROUGELsum	29.4	29.46	**29.69**	29.46	29.5

Java, small model, batch size: 512

Steps	10000	20000	30000	40000	50000
BLEU	6.049	7.45	7.909	**8.484**	7.984
ROUGE1	29.49	29.04	**30.65**	30.44	30.19
ROUGE2	10.3	10.71	**11.52**	11.39	11.3
ROUGELsum	27.39	27.1	**28.6**	28.1	28

Java, small model, batch size: 1024

Steps	10000	20000	30000
BLEU	6.842	**8.792**	8.161
ROUGE1	29.93	**31.08**	30.48
ROUGE2	11.21	**11.76**	11.74
ROUGELsum	28.14	**28.69**	28.46

Java, base model, batch size: 128

Steps	8000	16000	32000	48000	56000	64000	72000	80000	88000	96000	104000
BLEU	3.49	5.052	6.522	6.798	7.003	7.912	7.808	7.482	7.547	7.117	**7.852**
ROUGE1	26.64	27.56	28.69	29.73	29.33	29.7	29.98	29.91	29.64	29.33	**30.32**
ROUGE2	8.19	8.58	10.04	10.6	10.34	10.52	11.03	10.93	10.65	10.46	**11.16**
ROUGELsum	24.98	25.49	26.6	27.61	27.25	27.52	27.79	27.85	27.55	27.21	**28.11**

Java, base model, batch size: 256

Steps	24000	32000	40000	48000	56000	64000	72000	80000
BLEU	8.051	8.259	**8.959**	8.744	8.676	8.647	8.675	8.793
ROUGE1	30.92	30.2	**31.14**	30.72	31	30.98	30.8	30.87
ROUGE2	11.9	11.37	11.93	11.73	11.83	**12.13**	11.88	12.02
ROUGELsum	28.83	28.03	**28.85**	28.49	28.74	28.75	28.68	28.68

Java, base model, batch size: 512

Steps	5000	10000	15000	20000	25000	30000	35000	40000
BLEU	5.949	7.116	8.126	8.555	8.349	**8.874**	8.74	8.822
ROUGE1	28.33	28.1	29.78	30.18	30.43	30.27	30.36	**30.71**
ROUGE2	9.78	9.83	10.95	11.18	11.36	11.25	11.24	**11.52**
ROUGELsum	26.43	26.22	27.82	28.01	28.2	27.99	28.13	**28.44**

Java, base model, batch size: 1024

Steps	3000	6000	9000	11000	13000	15000	17000	19000	23000	27000
BLEU	4.706	7.282	7.248	7.868	8.047	6.819	**8.443**	7.759	8.213	8.414
ROUGE1	27.99	29.09	27.64	28.77	29.17	28.1	29.31	29.34	**29.55**	29.09
ROUGE2	9.24	10.49	9.38	9.92	10.37	9.89	10.41	10.32	**10.66**	10.31
ROUGELsum	26.21	27.19	25.73	26.65	27	26.21	27.09	27.22	**27.36**	26.9

Go, small model, batch size: 256

Steps	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BLEU	5.413	7.792	9.598	9.254	**10.757**	9.363	6.345	10.051	9.63	10.457
ROUGE1	33.11	38.48	40.3	40.46	**41.74**	41.75	36.86	41.58	41.12	41.67
ROUGE2	14.65	17.89	19.22	19.3	**20.39**	20.2	16.38	20.11	19.63	20.36
ROUGELsum	32.04	36.94	38.7	39.02	40.02	40.14	35.41	39.96	39.4	**40.18**

Go, base model, batch size: 128

Steps	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
BLEU	10.985	10.387	11.596	12.155	12.512	12.764	12.676	**12.845**	12.805	12.751
ROUGE1	41.59	41.58	41.64	41.82	42.11	42.65	42.5	**42.67**	42.19	42.38
ROUGE2	20.15	20.09	20.44	20.55	20.64	21.14	20.97	**21.18**	20.9	20.95
ROUGELsum	39.92	39.82	39.85	40	40.23	**40.7**	40.64	**40.7**	40.35	40.48

Go, base model, batch size: 256

Steps	5000	10000	15000	20000	15000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000
BLEU	10.779	11.681	11.534	12.812	12.628	12.826	12.843	12.659	13.238	12.812	12.946	12.912	**13.556**	12.996	13.46	13.267
ROUGE1	41.05	41.21	41.62	42.19	42.23	42.48	42.22	42.2	42.59	42.2	42.53	42.33	42.66	42.2	42.59	**42.81**
ROUGE2	20.17	20.13	20.39	20.79	20.81	21.14	20.89	20.98	21.33	20.83	21.13	20.98	21.34	20.88	21.17	**21.35**
ROUGELsum	39.59	39.46	39.92	40.3	40.28	40.56	40.3	40.26	40.78	40.26	40.55	40.4	40.67	40.29	40.72	**40.84**

Go, base model, batch size: 384

Steps	6000	12000	18000	24000	30000	36000	42000	48000	50000	56000	62000	68000	74000	80000	86000	92000	98000	100000
BLEU	8.822	11.621	11.328	11.706	11.47	12.03	11.833	12.063	12.474	12.018	12.315	12.009	12.171	**12.526**	12.215	12.05	12.048	12.315
ROUGE1	40.51	42.05	41.5	41.81	41.76	**42.19**	41.93	42.09	41.85	41.92	42.13	42.18	41.96	42.15	41.91	41.75	41.9	42.07
ROUGE2	19.55	20.37	20.09	20.41	20.25	20.61	20.38	20.72	20.54	20.53	**20.82**	20.56	20.63	20.79	20.57	20.33	20.44	20.6
ROUGELsum	38.95	40.25	39.64	40	39.83	**40.31**	40.06	40.19	39.9	39.97	40.29	40.18	40.18	40.21	40.04	39.81	39.95	40.25

Go, base model, batch size: 512

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	53000	56000
BLEU	10.813	13.151	13.138	12.951	13.448	13.305	13.334	13.577	13.316	**13.603**	13.579	12.7
ROUGE1	41.1	41.82	42.03	42.24	**42.59**	42.1	42.38	42.38	42.37	42.49	42.56	42.31
ROUGE2	20.24	20.72	20.89	21.01	21.31	21.1	21.08	21.2	21.09	**21.36**	21.31	21.02
ROUGELsum	39.57	39.99	40.28	40.51	**40.69**	40.41	40.47	40.6	40.43	40.67	40.66	40.52

Php, small model, batch size: 256

Steps	20000	40000	60000	80000	100000	120000	140000	160000	180000	200000	220000	240000
BLEU	9.979	12.826	13.479	13.752	14.047	13.546	14.213	14.624	14.022	**14.832**	14.586	14.139
ROUGE1	36.73	37.4	37.08	37.01	37.07	37.34	**37.93**	37.8	37.83	37.69	37.78	37.49
ROUGE2	14.97	15.61	15.73	15.88	16.11	16.06	16.42	16.43	**16.55**	16.36	16.49	16.29
ROUGELsum	35.04	35.56	35.28	35.21	35.25	35.61	36.01	35.89	**36.06**	35.73	35.95	35.74

Php, small model, batch size: 512

Steps	10000	20000	30000	40000	50000	60000	70000	80000
BLEU	9.532	12.729	13.153	13.33	14.032	13.591	**14.441**	13.913
ROUGE1	35.63	36.95	36.41	36.48	**37.09**	37.01	36.99	37.07
ROUGE2	13.89	15.08	15.3	15.48	15.99	15.9	15.86	**15.93**
ROUGELsum	33.94	35.18	34.64	34.74	35.29	35.34	35.17	**35.36**

Php, small model, batch size: 1024

Steps	10000	20000	30000
BLEU	11.748	12.324	13.38
ROUGE1	35.85	35.61	36.2
ROUGE2	14.05	14.56	15.08
ROUGELsum	34	33.94	34.49

Php, base model, batch size: 256

Steps	8000	32000	40000	48000	56000	64000	72000
BLEU	5.783	12.895	14.169	13.517	13.765	13.918	**14.152**
ROUGE1	33.97	35.59	36.76	36.02	36.02	36.42	**37.14**
ROUGE2	11.78	14.63	15.56	15.09	15.13	15.43	**15.89**
ROUGELsum	32.52	33.92	34.98	34.18	34.26	34.66	**35.39**

Php, base model, batch size: 512								
Steps	5000	10000	15000	20000	25000	30000	35000	40000
BLEU	10.369	12.198	13.528	14.041	14.075	13.949	**14.786**	14.598
ROUGE1	34.15	34.94	35.43	35.3	35.79	35.81	35.97	**36.05**
ROUGE2	12.97	13.95	14.36	14.72	14.93	15.02	**15.32**	15.16
ROUGELsum	32.13	33.16	33.55	33.43	33.99	34.1	**34.22**	34.19

Php, base model, batch size: 1024										
Steps	3000	6000	9000	12000	15000	18000	21000	24000	27000	30000
BLEU	9.035	13.999	14.177	14.055	14.484	14.855	14.986	**15**	14.89	14.626
ROUGE1	34.76	36.53	36.5	36.18	36.4	36.61	36.59	36.69	36.7	**36.84**
ROUGE2	13.19	15.42	15.38	15.69	15.65	15.82	15.94	16.04	15.9	**16.06**
ROUGELsum	33.05	34.7	34.72	34.43	34.63	34.86	34.82	34.96	34.99	**35.05**

Ruby, small model, batch size: 128										
Steps	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BLEU	0.857	1.508	1.828	2.098	2.065	1.982	2.226	1.976	2.067	**2.384**
ROUGE1	16.22	15.89	16.71	16.9	16.35	16.09	**16.98**	16.49	15.98	16.93
ROUGE2	2.7	2.59	3.02	3.5	3.13	2.97	3.15	2.96	2.85	**3.56**
ROUGELsum	14.93	14.58	15.51	15.81	15.08	14.87	15.67	15.24	14.79	**15.81**

Ruby, small model, batch size: 256					
Steps	1000	2000	3000	4000	5000
BLEU	0.958	1.756	1.879	**2.216**	2.085
ROUGE1	13.11	13.83	**14.47**	14.41	14.36
ROUGE2	1.81	2.25	2.36	**2.65**	2.42
ROUGELsum	12.37	12.91	**13.27**	13.21	13.23

Ruby, base model, batch size: 32					
Steps	1000	2000	3000	4000	5000
BLEU	0.315	0.739	0.776	0.746	0.932
ROUGE1	10.64	14.06	12.85	13.24	11.55
ROUGE2	0.8	1.66	1.96	1.92	1.38
ROUGELsum	10.05	12.84	11.9	12.22	10.64

Ruby, base model, batch size: 128					
Steps	2000	4000	6000	8000	10000
BLEU	1.1	1.726	1.87	**1.897**	1.853
ROUGE1	13.62	13.1	12.97	**13.57**	12.91
ROUGE2	2.24	2.23	2.13	**2.41**	2.06
ROUGELsum	12.56	12.21	11.82	**12.41**	11.87

Javascript, small model, batch size: 128										
Steps	4000	8000	12000	16000	20000	24000	28000	32000	36000	40000
BLEU	2.756	5.274	5.416	6.946	6.645	7.079	6.778	6.757	**7.256**	7.043
ROUGE1	18.68	19.82	19.3	20.94	20.69	**20.99**	20.44	20.91	20.91	20.48
ROUGE2	4.5	6.15	6.15	7.33	6.97	7.31	7.13	7.09	**7.38**	7.08
ROUGELsum	17.43	18.74	18.17	19.62	19.37	**19.71**	19.25	19.63	19.62	19.29

Javascript, small model, batch size: 256															
Steps	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	12000	14000	16000	18000	20000
BLEU	1.18	1.633	2.26	4.036	5.358	5.255	5.849	6.249	6.589	6.686	6.731	7.345	**7.49**	7.468	7.29
ROUGE1	18.06	16.88	17.87	19.52	19.82	19.65	18.81	19.39	20.55	20.82	20.39	20.71	**21.04**	20.54	20.84
ROUGE2	3.79	3.55	4.15	5.76	6.53	6.29	6.29	6.45	7	7.01	6.86	7.37	7.34	7.29	**7.44**
ROUGELsum	17.04	15.8	16.92	18.42	18.81	18.49	17.74	18.19	19.33	19.53	19.27	19.43	**19.88**	19.29	19.61

Javascript, small model, batch size: 512					
Steps	1000	2000	3000	4000	5000
BLEU	1.849	2.736	4.623	5.982	6.021
ROUGE1	18.64	18.32	18.25	19.65	19.75
ROUGE2	3.8	4.42	5.33	6.56	6.36
ROUGELsum	17.48	17.27	17.12	18.58	18.53

Javascript, base model, batch size: 32								
Steps	1000	2000	5000	6000	7000	8000	9000	10000
BLEU	0.643	0.568	0.92	1.536	1.258	0.941	1.616	1.528
ROUGE1	12.59	11.59	14.55	15.54	14.33	13.74	15.07	14.77
ROUGE2	1.52	1.79	2.45	3.01	2.6	2.39	2.85	2.51
ROUGELsum	11.72	11.16	13.74	14.61	13.54	12.95	14.09	13.78

Javascript, base model, batch size: 128															
Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000	24000	28000	32000	36000	40000
BLEU	1.083	2.67	4.149	5.339	5.685	5.619	5.784	6.051	5.905	5.481	6.52	6.155	6.241	**6.714**	6.206
ROUGE1	15.84	14.87	16.15	17.38	17.14	17.59	17.34	17.85	17.86	17.39	18.34	18.08	18.15	**18.49**	17.79
ROUGE2	2.95	3.47	4.59	5.33	5.67	5.76	5.46	5.9	5.78	5.3	6.19	5.98	5.94	**6.44**	5.77
ROUGELsum	15.03	14.04	15.2	16.39	16.2	16.62	16.25	16.82	16.94	16.28	17.16	17.02	16.95	**17.47**	16.8

Javascript, base model, batch size: 256									
Steps	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	6.312	6.326	6.618	6.555	6.65	7.053	7.169	**7.499**	7.143
ROUGE1	18.31	17.97	18.17	17.99	18.74	18.84	18.53	**18.91**	18.95
ROUGE2	5.92	6.34	6.44	6.25	6.61	6.65	6.65	**6.84**	6.62
ROUGELsum	17.14	16.94	17.14	16.97	17.76	17.78	17.51	**17.79**	17.78

A.1.2. Source Code Summarization

Python, small model, batch size: 233									
Steps	3000	4000	5000	6000	7000	8000	9000	10000	11000
BLEU	0.689	0.628	**1.046**	0.894	0.823	0.785	0.732	0.936	0.884
ROUGE1	15.02	13.65	15.23	15.04	15.36	14.48	14.67	**15.31**	15.04
ROUGE2	2.49	1.86	**2.55**	2.43	2.35	2.14	2.25	2.49	2.32
ROUGELsum	13.39	12.17	**13.63**	13.39	13.53	12.94	13.02	13.51	13.42

Python, small model, batch size: 384														
Steps	3000	9000	15000	20000	23000	29000	32000	38000	44000	53000	59000	65000	68000	70000
BLEU	0.689	0.755	0.717	0.744	0.814	**0.95**	0.813	0.846	0.686	0.869	0.8	0.72	0.698	0.758
ROUGE1	15.02	14.5	14.76	15.03	14.7	15.12	14.91	14.98	14.71	14.64	14.4	14.64	**15.35**	14.5
ROUGE2	2.49	2.14	2.34	2.32	2.31	2.37	2.37	2.43	2.34	2.32	2.23	2.28	**2.48**	2.14
ROUGELsum	13.39	12.8	13.13	13.28	13.07	13.36	13.01	13.36	12.94	12.97	12.77	12.97	**13.56**	12.85

Python, base model, batch size: 32					
Steps	1000	2000	3000	4000	5000
BLEU	**1.615**	0.59	0.737	0.631	0.672
ROUGE1	**16.58**	13.64	13.51	14.01	13.47
ROUGE2	**3.8**	1.95	1.8	2	1.97
ROUGELsum	**15.08**	12.17	12.02	12.51	11.97

Python, base model, batch size: 384													
Steps	4000	8000	12000	16000	20000	24000	28000	32000	36000	40000	44000	48000	50000
BLEU	0.67	0.673	0.691	0.633	0.687	0.74	0.724	**0.729**	0.697	0.714	0.658	0.685	0.669
ROUGE1	**14.19**	13.71	13.46	13.94	13.65	13.68	14.03	13.82	13.81	14.05	14.07	14.01	14.16
ROUGE2	**2.16**	1.95	1.82	1.98	1.86	1.94	1.94	1.97	1.97	2.03	2.05	2.06	2.08
ROUGELsum	**12.44**	12.13	11.81	12.24	12.1	12.17	12.33	12.13	12.15	12.43	12.47	12.31	12.43

SQL, small model, batch size: 64						
Steps	500	1000	1500	2000	2500	3000
BLEU	1.54	**1.685**	1.506	1.028	1.071	0.874
ROUGE1	**18.1**	15.89	16.6	13.97	14.96	14.21
ROUGE2	**3.57**	3.15	3.24	2.19	2.32	2.22
ROUGELsum	**16.36**	14.55	15.16	12.73	13.54	12.81

SQL, small model, batch size: 128						
Steps	500	1000	1500	2000	2500	3000
BLEU	**1.706**	1.172	1.114	0.859	0.724	0.671
ROUGE1	**17.71**	15.9	14.92	14.11	13.29	12.94
ROUGE2	**3.99**	2.77	2.59	2.06	1.73	1.69
ROUGELsum	**16.09**	14.45	13.72	12.67	12.02	11.65

SQL, small model, batch size: 384																	
Steps	3000	6000	9000	12000	15000	18000	21000	24000	27000	30000	33000	36000	39000	42000	45000	48000	50000
BLEU	0.636	0.647	0.787	**0.792**	0.69	0.738	0.741	0.7	0.664	0.607	0.679	0.658	0.791	0.589	0.647	0.735	0.645
ROUGE1	13.3	13.14	13.22	**13.44**	13.23	13.29	13.21	13.27	13.22	13.13	13.1	13.29	13.41	13.2	12.89	12.99	13.26
ROUGE2	1.71	1.7	1.75	**1.78**	1.67	1.7	1.74	1.71	1.64	1.62	1.7	1.6	**1.78**	1.71	1.63	1.68	1.7
ROUGELsum	11.97	11.72	11.87	**12.04**	11.87	11.96	11.9	11.89	11.79	11.79	11.75	11.86	12.01	11.86	11.58	11.7	11.86

SQL, base model, batch size: 32

Steps	500	1000	1500	2000	2500	3000
BLEU	**1.281**	1.138	1.209	1.164	0.828	0.696
ROUGE1	**16.27**	16.12	15	14.44	14.05	12.31
ROUGE2	**2.86**	2.8	2.59	2.36	2.21	1.42
ROUGELsum	14.51	**14.76**	13.7	13.14	12.77	11.25

SQL, base model, batch size: 4000

Steps	1100	2200	3300	4400	5000	3000
BLEU	0.715	**0.752**	0.653	0.586	0.568	0.696
ROUGE1	11.96	**12.36**	12.44	12.3	12.21	12.31
ROUGE2	1.5	**1.58**	1.54	1.56	1.56	1.42
ROUGELsum	10.78	**11.18**	11.16	11.02	10.97	11.25

CSharp, small model, batch size: 32

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	2.423	**2.439**	2.094	1.955	1.719	1.687	1.619	1.75	1.491	1.508
ROUGE1	**20.01**	18.07	17.46	16.52	15.54	15.91	16.19	15.61	15.58	15.28
ROUGE2	**4.53**	4.16	3.54	3.31	2.83	2.92	3.08	2.82	2.76	2.69
ROUGELsum	**18.4**	16.7	15.98	15.24	14.17	14.53	14.71	14.18	14.17	13.89

CSharp, small model, batch size: 64

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	**2.401**	2.12	1.787	1.683	1.648	1.541	1.48	1.649	1.504	1.558
ROUGE1	**18.9**	17.69	16.14	16.64	15.99	15.48	15.59	15.75	15.52	15.56
ROUGE2	**4.44**	3.73	3.05	3.22	2.92	2.91	2.77	2.92	2.77	2.87
ROUGELsum	**17.39**	16.24	14.73	15.17	14.55	14.13	14.19	14.29	14.04	14.15

CSharp, small model, batch size: 128

Steps	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BLEU	2.099	**2.799**	1.975	2.043	1.697	1.631	1.843	1.804	1.638	1.457
ROUGE1	18.17	**20.02**	16.59	17.22	16.03	16.16	16.37	16.01	15.34	14.87
ROUGE2	3.77	**4.61**	3.27	3.52	2.96	2.98	3.22	3	2.74	2.47
ROUGELsum	16.86	**18.29**	15.06	15.72	14.61	14.71	14.95	14.64	13.98	13.54

CSharp, small model, batch size: 384

Steps	2000	4000	6000	8000	10000
BLEU	1.441	1.652	**1.67**	1.409	1.505
ROUGE1	15.54	15.81	**15.97**	15.17	15.66
ROUGE2	2.93	2.9	**2.98**	2.58	2.77
ROUGELsum	14.27	14.37	**14.49**	13.83	14.2

CSharp, base model, batch size: 32

Steps	500	1000	1500	2000	2500	3000
BLEU	**2.686**	1.998	1.857	2.117	2.361	2.105
ROUGE1	**20.55**	17.75	17.67	17.42	19.38	17.61
ROUGE2	**4.69**	3.8	3.49	3.28	4.22	3.9
ROUGELsum	**18.69**	16.76	16.62	16.15	17.6	16.22

A.1.3. Code Comment Generation

Java, small model, batch size: 128				
Steps	1000	5000	15000	40000
BLEU	2.825	14.111	21.437	30.7
ROUGE1	27.34	37.69	43.87	49.17
ROUGE2	12.24	22.27	29.63	36.09
ROUGELsum	26.53	36.55	42.79	48.14

Java, small model, batch size: 256															
Steps	100000	200000	230000	260000	290000	320000	350000	380000	410000	440000	460000	480000	500000	520000	540000
BLEU	43.365	48.313	49.246	49.895	50.041	50.382	50.386	50.827	50.731	50.864	50.872	50.881	**50.934**	50.843	50.871
ROUGE1	55.27	56.94	57.19	57.13	57.52	57.44	57.56	57.48	**57.68**	57.48	57.62	57.64	57.58	57.64	57.59
ROUGE2	43.9	45.98	46.19	46.37	46.64	46.67	46.76	46.8	46.86	46.77	46.84	46.86	46.85	**46.93**	46.87
ROUGELsum	54.31	45.98	56.26	56.15	56.55	56.51	56.63	56.54	**56.72**	56.54	56.66	56.69	56.62	56.66	56.64

Java, small model, batch size: 512						
Steps	5000	10000	20000	30000	40000	50000
BLEU	18.881	26.729	33.301	40.004	39.888	42.135
ROUGE1	43.07	46.64	50.28	52.44	53.09	54.3
ROUGE2	28.84	33.33	38.12	40.55	41.96	43.19
ROUGELsum	42.07	45.61	49.35	51.39	52.18	53.37

Java, small model, batch size: 1024			
Steps	10000	20000	30000
BLEU	32.075	40.29	41.86
ROUGE1	48.85	52.71	54.32
ROUGE2	36.28	41.04	43.23
ROUGELsum	47.82	51.73	53.43

Java, base model, batch size: 256																
Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000
BLEU	21.372	30.134	38.078	42.894	47.233	48.788	49.601	50.379	50.566	50.976	51.108	51.362	51.534	51.516	51.382	**51.561**
ROUGE1	43.1	48.4	51.83	54.11	55.88	56.58	56.87	57.23	57.41	57.58	57.64	57.5	**57.78**	57.59	57.57	57.63
ROUGE2	29	35.59	39.89	43.01	45.03	46.01	46.51	46.8	47.02	47.16	47.18	47.16	**47.38**	47.23	47.26	47.28
ROUGELsum	42.1	47.41	50.85	53.18	54.91	55.72	56	56.29	56.51	56.65	56.72	56.54	**56.84**	56.69	56.65	56.71

Java, base model, batch size: 512										
Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
BLEU	20.967	29.166	45.99	49.678	50.088	50.914	51.022	50.919	**51.314**	51.12
ROUGE1	45.64	52.02	54.47	55.55	55.98	56.54	56.8	56.46	56.53	**56.55**
ROUGE2	33.14	40.44	44.01	45.17	45.89	46.26	46.42	46.41	**46.55**	46.49
ROUGELsum	44.78	51.11	53.65	54.61	55.06	55.64	55.9	55.58	55.62	**55.65**

Java, base model, batch size: 1024										
Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	22.398	24.244	40.488	45.714	48.146	49.783	49.136	50.803	51.14	**51.408**
ROUGE1	41.393	47.27	51.7	54.244	55.49	56.3	56.43	56.41	56.6	**56.87**
ROUGE2	26.997	35.307	40.42	43.722	45.1	45.86	46.15	46.23	46.46	**46.72**
ROUGELsum	40.211	46.352	50.71	53.329	54.62	55.4	55.65	55.54	55.66	**55.92**

A.1.4. Git Commit Message Generation

Java, small model, batch size: 128				
Steps	5000	10000	15000	20000
BLEU	40.073	**40.242**	40.196	39.866
ROUGE1	38.96	39.13	**39.22**	38.76
ROUGE2	29.26	29.6	**29.68**	29.53
ROUGELsum	38.77	38.88	**39.03**	38.54

Java, small model, batch size: 256					
Steps	2000	4000	6000	8000	10000
BLEU	40.018	**40.222**	39.934	40.002	39.819
ROUGE1	38.87	38.89	39.04	39.11	**39.18**
ROUGE2	29.3	29.52	29.49	**29.66**	29.56
ROUGELsum	38.7	38.63	38.81	**38.95**	38.88

Java, small model, batch size: 512					
Steps	1000	2000	3000	4000	5000
BLEU	39.725	40.106	40	**40.123**	40.05
ROUGE1	38.79	39.31	39.16	**39.48**	39.14
ROUGE2	28.87	29.37	29.54	**29.95**	29.82
ROUGELsum	38.6	39.05	38.88	**39.25**	38.96

Java, base model, batch size: 32					
Steps	2000	4000	6000	8000	10000
BLEU	32.854	35.213	36.872	37.247	37.635
ROUGE1	32.78	34.29	35.57	36.97	36.725
ROUGE2	22.55	24.92	26.68	27.43	27.993
ROUGELsum	32.69	34.07	35.29	36.76	36.479

Java, base model, batch size: 128											
Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000
BLEU	39.021	38.92	39.205	39.325	39.29	39.331	39.343	39.372	39.402	**39.667**	39.465
ROUGE1	38.25	38.36	38.58	38.6	38.71	39.04	**39.08**	38.72	38.76	38.83	38.91
ROUGE2	28.82	29.01	29.1	29.29	29.29	29.46	29.41	29.24	29.32	**29.48**	29.42
ROUGELsum	38.05	38.12	38.39	38.31	38.43	38.82	**38.84**	38.45	38.49	38.64	38.77

Java, base model, batch size: 256					
Steps	2000	4000	6000	8000	10000
BLEU	38.603	39.184	36.825	**39.623**	39.433
ROUGE1	37.9	38.79	38.12	**38.82**	38.56
ROUGE2	28.69	29.35	29	**29.52**	29.42
ROUGELsum	37.74	38.48	37.92	**38.64**	38.4

Java, base model, batch size: 512							
Steps	2000	4000	6000	8000	10000	12000	14000
BLEU	39.251	39.206	39.359	39.368	**39.774**	39.716	39.261
ROUGE1	38.76	38.87	38.56	38.81	**39.11**	39.05	38.75
ROUGE2	29.56	29.54	29.52	29.66	**29.98**	29.87	29.51
ROUGELsum	38.47	38.6	38.28	38.56	**38.88**	38.8	38.5

A.1.5. API Sequence Generation

Java, small model, batch size: 256

Steps	200000	400000	440000	480000	520000	560000	600000	640000	680000	720000	760000	800000	840000	880000	920000
BLEU	67.539	69.224	69.581	69.768	69.89	69.949	70.235	70.446	70.568	70.733	70.493	70.567	**70.916**	70.536	70.845
ROUGE1	74.82	76.02	76.3	76.5	76.49	76.69	76.78	76.94	76.98	77.16	77.01	77.23	**77.4**	77.24	77.38
ROUGE2	65.55	67.07	67.42	67.62	67.61	67.81	67.96	68.18	68.3	68.38	68.31	68.43	68.72	68.49	**68.73**
ROUGELsum	74.79	76.01	76.27	76.5	76.46	76.68	76.77	76.95	76.98	77.15	76.99	77.21	77.37	77.24	**77.39**

Java, small model, batch size: 512

Steps	10000	20000	30000	40000	50000	60000	70000	80000	140000	150000
BLEU	55.962	60.503	62.922	64.331	64.945	65.994	66.116	66.336	68.402	68.489
ROUGE1	64.2	68	70.08	71.38	72.26	73.02	73.4	73.66	75.19	75.38
ROUGE2	53.68	57.82	60.27	61.76	62.66	63.51	63.99	64.34	66.08	66.28
ROUGELsum	64.16	67.96	70.08	71.34	72.24	72.98	73.38	73.64	75.15	75.38

Java, small model, batch size: 1024

Steps	10000	20000	30000
BLEU	59.444	62.912	65.27
ROUGE1	66.62	70.15	72.09
ROUGE2	56.47	60.47	62.62
ROUGELsum	66.58	70.15	72.18

Java, base model, batch size: 256

Steps	20000	40000	60000	80000	85000	90000	95000	100000	105000	110000	115000	120000	125000	130000	135000	140000	145000	150000
BLEU	62.376	66.938	68.656	69.559	70.483	70.399	70.425	70.937	71.128	71.366	71.495	71.696	71.719	72.155	71.811	71.989	**72.316**	72.242
ROUGE1	69.59	73.81	75.584	76.76	77.24	77.31	77.57	77.75	78.15	78.16	78.32	78.41	78.5	78.88	78.82	78.73	**79.11**	78.96
ROUGE2	59.55	64.52	66.582	68.11	68.5	68.5	68.88	69.09	69.39	69.56	69.75	69.74	69.99	70.32	70.29	70.3	**70.55**	70.51
ROUGELsum	69.56	73.78	75.604	76.79	77.21	77.3	77.55	77.71	78.11	78.17	78.33	78.42	78.48	78.83	78.8	78.72	**79.09**	78.93

Java, base model, batch size: 512

Steps	8000	10000	12000	14000	16000	18000	20000	24000	26000	28000	30000
BLEU	59.443	60.519	62.519	63.552	64.827	64.855	66.192	67.324	67.583	67.771	68.177
ROUGE1	67.61	68.44	69.92	70.98	72.02	72.39	73.14	74.32	74.68	75	75.29
ROUGE2	57.09	58.33	60.09	61.09	62.41	62.86	63.8	64.86	65.37	65.73	66.05
ROUGELsum	67.56	68.4	69.89	70.94	71.99	72.4	73.13	74.32	74.64	75	75.26

Java, base model, batch size: 1024

Steps	6000	8000	10000
BLEU	33.84	62.4	64.631
ROUGE1	68.62	70.7	71.82
ROUGE2	58.67	60.62	62.29
ROUGELsum	68.6	70.72	71.8

A.1.6. Program Synthesis

DSL, small model, batch size: 128					
Steps	4000	8000	12000	16000	20000
BLEU	93.925	94.327	94.478	**94.48**	94.361
ROUGE1	94.25	**98.9**	97.37	97.1	94.28
ROUGE2	92.87	**98.42**	96.68	96.33	93.06
ROUGELsum	93.97	**98.7**	97.18	96.9	94.09
Accuracy	78.131	80.137	84.869	**87.125**	80.331

DSL, small model, batch size: 256		
Steps	2000	4000
BLEU	93.688	93.983
ROUGE1	94.68	94.05
ROUGE2	93.16	92.72
ROUGELsum	94.33	93.82
Accuracy	75.183	75.183

DSL, small model, batch size: 512			
Steps	2000	4000	6000
BLEU	94.213	93.335	94.401
ROUGE1	99.02	93.68	95.94
ROUGE2	98.47	92.23	94.93
ROUGELsum	98.77	93.44	95.73
Accuracy	78.917	77.641	79.638

DSL, base model, batch size: 32		
Steps	1000	2000
BLEU	75.733	75.54
ROUGE1	87.76	91.49
ROUGE2	72.83	78.12
ROUGELsum	81.71	85.69
Accuracy	13.929	39.967

DSL, base model, batch size: 128					
Steps	4000	8000	12000	16000	20000
BLEU	93.605	**94.503**	94.333	94.133	94.344
ROUGE1	98.51	**99.11**	98.77	94.1	94.26
ROUGE2	97.68	**98.62**	98.27	92.85	93.03
ROUGELsum	98.18	**98.89**	98.55	93.92	94.08
Accuracy	86.246	**91.635**	88.372	80.1	80.59

DSL, base model, batch size: 256								
Steps	2000	4000	6000	8000	10000	12000	14000	16000
BLEU	93.293	94.353	94.187	94.446	94.438	94.236	93.543	**94.525**
ROUGE1	98.4	99	98.9	99.04	**99.16**	98.79	98.81	97.73
ROUGE2	97.04	98.45	98.28	98.43	**98.73**	98.22	98.37	97.11
ROUGELsum	97.79	98.76	98.67	98.77	**98.93**	98.56	98.63	97.55
Accuracy	85.452	90.692	77.909	**91.081**	80.118	79.582	79.185	85.156

A.2. Transfer Learning

Code Documentation Generation - Python, small model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
BLEU	6.724	6.923	6.607	6.284	6.751	6.962	**7.023**	6.853	6.568	6.754
ROUGE1	**34.95**	34.57	33.9	33.51	33.41	33.06	33.12	32.71	32.3	32.49
ROUGE2	13.21	**12.93**	12.59	12.39	12.3	12.09	12.05	11.84	11.62	11.72
ROUGELsum	**32.67**	32.23	31.66	31.39	31.2	30.8	30.85	30.42	30.08	30.21

Code Documentation Generation - Python, base model, batch size: 256

Steps	2000	4000	6000	8000	10000	14000	18000	20000	24000	28000	30000	35000	40000	45000	50000	55000	60000	65000	70000
BLEU	7.247	7.494	7.262	7.569	7.367	7.318	6.982	7.544	7.917	8.099	8.05	8.009	7.957	8.045	8.123	**8.338**	8.041	8.235	8.237
ROUGE1	**35.57**	35.54	35.22	34.35	34.31	33.85	32.98	33.22	33.02	32.38	32.95	32.85	32.55	32.72	32.92	33.1	32.45	32.86	32.88
ROUGE2	**14.02**	13.99	13.63	13.01	12.9	12.74	12.09	12.29	12.12	11.82	12.1	12.11	12.03	12.02	12.24	12.38	11.95	12.16	12.28
ROUGELsum	**33.39**	33.22	32.94	31.95	31.99	31.54	30.77	30.87	30.63	29.81	30.46	30.36	30.16	30.26	30.54	30.64	30.04	30.37	30.48

Code Documentation Generation - Python, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000
BLEU	6.95	7.2	7.137	7.098	**7.618**	7.501
ROUGE1	**35.58**	35.49	35.19	35	35.17	35.14
ROUGE2	**14.07**	14.03	13.75	13.67	13.65	13.51
ROUGELsum	**33.49**	33.26	33.02	32.83	32.68	32.73

Code Documentation Generation - Java, small model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000
BLEU	6.959	8.464	7.379	7.642	8.115	8.543	8.91	8.81	8.969	9.217	9.178	**9.934**	9.922	9.518
ROUGE1	36.07	**36.93**	35.39	35.32	35.63	35.11	35.25	35.49	35.04	35.25	35.23	35.12	34.95	34.4
ROUGE2	16.32	**16.51**	15.08	14.97	15.22	14.81	15.16	15.3	15.05	15.25	15.09	14.99	14.83	14.41
ROUGELsum	34.12	**34.53**	33.17	33.03	33.3	32.8	32.87	33.17	32.73	32.97	32.93	32.77	32.59	32.07

Code Documentation Generation - Java, base model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000
BLEU	8.099	9.247	9.662	10.84	10.803	10.872	11.491	11.242	11.278	11.22	11.677	**11.807**	11.456	11.703
ROUGE1	**36.82**	36.55	35.99	36.06	36.02	35.97	36.23	36.61	36.02	36.31	36.75	36.3	36.5	36.44
ROUGE2	**16.88**	16.09	15.39	15.84	15.7	15.72	15.99	16.27	15.74	16.21	16.53	16.1	16.24	16.07
ROUGELsum	**34.64**	34.09	33.44	33.6	33.41	33.39	33.69	34	33.4	33.86	34.19	33.74	33.93	33.81

Code Documentation Generation - Java, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	8.432	7.479	9.068	9.23	9.012	10.723	10.013	9.285	**11.351**	10.985
ROUGE1	37.75	36.77	37.95	**38.22**	37.17	37.97	37.15	36.39	37.17	37.68
ROUGE2	**18.11**	17.02	17.88	17.98	16.86	17.47	16.93	16.37	16.96	17.47
ROUGELsum	35.68	34.83	35.7	**35.94**	34.78	35.39	34.66	34.13	34.7	35.03

Code Documentation Generation - Go, small model, batch size: 256

Steps	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	110000	120000	130000	140000	150000	160000
BLEU	13.145	13.364	13.718	14.243	13.639	13.824	14.201	14.468	14.275	14.473	14.582	14.679	14.529	**14.702**	14.508	14.543
ROUGE1	**46.69**	46.26	45.5	45.08	45.2	45.48	45.03	44.94	44.76	44.4	44.83	44.7	44.76	44.6	44.39	44.57
ROUGE2	**24.01**	23.67	23.19	22.77	22.92	22.82	22.58	22.58	22.65	22.33	22.47	22.52	22.63	22.6	22.38	22.7
ROUGELsum	**44.68**	44.11	43.31	42.81	43	43.18	42.71	42.66	42.53	42.23	42.55	42.43	42.49	42.37	42.13	42.4

Code Documentation Generation - Go, base model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000
BLEU	13.68	13.789	15.06	15.066	15.225	15.381	15.045	15.34	15.339	**15.624**	15.416	15.371	15.527	15.57
ROUGE1	**47.66**	46.35	46.34	46.08	46.09	46.08	46.15	45.93	46.08	45.84	46.07	45.99	46.08	46.05
ROUGE2	**24.8**	23.95	23.79	23.53	23.72	23.74	23.72	23.62	23.78	23.63	23.72	23.66	23.85	23.86
ROUGELsum	**45.45**	44.22	44.04	43.86	43.86	43.88	43.92	43.83	43.93	43.62	43.87	43.76	43.95	43.84

Code Documentation Generation - Go, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000
BLEU	14.02	14.114	12.862	13.532	**14.38**	13.955
ROUGE1	48.03	**48.24**	47.24	47.06	46.74	46.41
ROUGE2	25.2	**25.41**	24.88	24.7	24.23	24.05
ROUGELsum	46.05	**46.07**	45.38	45.05	44.57	44.35

Code Documentation Generation - Php, small model, batch size: 256

Steps	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	110000	120000
BLEU	11.494	11.241	12.979	13.289	12.899	14.837	14.923	15.43	14.621	**15.79**	15.758	15.294
ROUGE1	**41.34**	40.98	40.7	40.47	40.13	40.36	40.22	40.49	40.23	40.15	40.02	40.48
ROUGE2	17.96	18.04	17.78	17.72	17.79	18.13	18.13	18.23	18.23	18.14	18.26	**18.55**
ROUGELsum	**39.45**	39.11	38.66	38.55	38.28	38.29	38.25	38.44	38.38	38.03	38.04	38.6

Code Documentation Generation - Php, base model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000
BLEU	11.697	14.216	15.171	15.996	17.248	17.133	16.876	16.92	17.973	17.979	17.416	18.098	17.872	17.327	**18.401**	18.302	17.65
ROUGE1	41.9	42.13	42.45	42.14	41.76	41.81	41.96	42.1	42.12	42.35	42.46	41.84	**42.66**	41.95	42.19	42.46	42.43
ROUGE2	19.08	19.42	19.78	19.92	19.6	19.71	20.07	19.92	20.07	20.33	20.4	20.25	**20.57**	20.25	20.36	20.51	20.51
ROUGELsum	40.18	40.19	40.5	40.17	39.66	39.77	40.06	40.11	40.06	40.26	40.49	39.76	**40.64**	40.08	40.13	40.39	40.5

Code Documentation Generation - Php, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	4000	5000	6000	7000	8000	9000	10000	11000	12000	13000	14000	15000	16000	17000	18000	19000	20000
BLEU	8.594	12.038	13.279	12.934	13.799	14.038	14.474	15.143	15.723	15.373	16.62	17.309	17.533	17.805	17.742	17.701	17.782	17.719	17.847	18.266	17.426	**18.515**	18.404
ROUGE1	40.74	42.36	42.61	**43.06**	42	42.18	42.21	42.13	41.93	42.66	42.56	42.84	42.68	42.9	42.73	42.49	42.64	42.54	42.58	42.84	43.05	42.41	42.27
ROUGE2	17.74	19.12	19.92	19.67	19.57	19.67	19.44	19.66	19.74	20.33	20.42	20.49	20.61	20.67	20.68	20.47	20.57	20.77	20.67	20.87	**21.06**	20.63	20.65
ROUGELsum	39.06	40.48	40.66	41.01	40.01	40.14	40.1	40	39.89	40.68	40.58	40.76	40.63	40.86	40.64	40.5	40.6	40.59	40.47	40.8	**41.12**	40.2	40.21

Code Documentation Generation - Ruby, small model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
BLEU	4.144	4.141	4.277	4.398	4.469	**4.663**	4.469	4.329	4.421	4.226
ROUGE1	**28**	27.82	27.94	27.6	27.24	27.75	27.38	27.62	27.41	27.3
ROUGE2	**8.69**	8.33	8.38	8.33	8.08	8.43	8.41	8.3	8.11	8.31
ROUGELsum	**25.85**	25.49	25.54	25.5	25.06	25.58	25.22	25.29	25.14	25.04

Code Documentation Generation - Ruby, base model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
BLEU	**5.309**	4.881	5.294	4.939	5.194	5.148	5.039	5.097	4.603	5.106
ROUGE1	30.04	29.68	**30.23**	29.93	29.92	29.87	29.26	29.41	29.22	29.69
ROUGE2	10.29	10.06	10.18	9.95	10.1	**10.42**	9.75	9.79	9.36	9.78
ROUGELsum	**27.77**	27.19	27.64	27.46	27.54	27.52	26.76	26.86	26.69	27.38

Code Documentation Generation - Ruby, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000
BLEU	4.889	5.39	5.339	5.001	5.09	5.427	5.197	5.426	5.419	5.481	**5.487**	5.104
ROUGE1	29.91	**30.99**	30.57	29.88	30.99	30.02	29.7	30.22	30.43	30.18	30.25	30.18
ROUGE2	10.83	**11.19**	10.82	10.43	10.98	10.27	10.49	10.5	10.88	10.53	10.39	10.21
ROUGELsum	27.79	**28.52**	28.04	27.34	28.31	27.5	27.4	27.81	28.04	27.85	27.83	27.71

Code Documentation Generation - Javascript, small model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000
BLEU	4.859	6.926	7.983	8.736	8.855	9.017	9.201	9.2	9.191	9.323	**9.44**	9.413
ROUGE1	28.18	28.5	28.28	28.57	**28.76**	28.5	28.64	28.82	28.69	28.53	28.33	28.1
ROUGE2	9.74	10.65	10.82	11.2	11.31	11.31	11.29	**11.32**	11.22	11.27	11.22	11.18
ROUGELsum	26.39	26.69	26.23	26.61	26.84	26.57	26.63	**26.89**	26.57	26.49	26.31	26.17

Code Documentation Generation - Javascript, base model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	650000
BLEU	9.1	10.246	10.674	10.773	10.712	10.592	**10.948**	10.723	10.604	10.533	10.644	10.661	10.621
ROUGE1	30.59	31.04	30.71	30.68	**31.14**	30.51	30.72	30.77	30.58	30.3	30.33	30.36	30.45
ROUGE2	12.39	12.94	12.97	12.84	13.15	13.07	**13.25**	13.07	12.97	12.91	12.88	12.78	12.99
ROUGELsum	28.4	28.92	28.66	28.47	**28.97**	28.52	28.6	28.73	28.53	28.19	28.2	28.26	28.28

Code Documentation Generation - Javascript, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	5.156	6.577	8.745	10.372	10.543	10.465	**10.975**	10.551	10.78	10.863
ROUGE1	30.14	30.77	30.7	31.73	**31.75**	31.16	31.22	31.68	31.48	31.29
ROUGE2	11.14	11.89	12.64	13.35	13.44	13.04	13.3	**13.64**	13.42	13.56
ROUGELsum	28.4	28.86	28.87	29.55	**29.66**	29.03	29.06	29.54	29.32	29.29

Source Code Summarization - Python, small model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000
BLEU	1.532	1.681	1.66	**1.796**	1.579	1.464	1.657
ROUGE1	**18.97**	18.86	19	18.91	18.72	18.28	18.85
ROUGE2	**3.72**	3.7	3.66	3.67	3.63	3.32	3.45
ROUGELsum	**16.77**	16.63	16.7	16.57	16.39	16.02	16.43

Source Code Summarization - Python, base model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	6000	7000	8000	9000	10000
BLEU	2.084	**2.223**	2.148	1.973	2.208	2.137	2.161	2.054	1.871	2.001	0.796	1.95	1.898	2.029	1.968
ROUGE1	18.38	**21.44**	20.27	20.78	19.7	20.54	20.45	20.18	19.49	19.95	8.89	19.72	19.6	19.76	19.69
ROUGE2	3.92	**4.33**	4.11	4.23	3.97	4.24	4.22	4.26	3.79	3.95	1.69	3.97	4	4.05	3.94
ROUGELsum	16.63	**18.71**	17.82	18.23	17.38	17.89	17.92	17.77	17.06	17.55	8.11	17.35	17.24	17.49	17.33

Source Code Summarization - Python, large model, batch size: 256

Steps	100	200	300	400	500	600	700	800	900	1000
BLEU	2.165	1.439	1.658	2.073	1.869	**2.221**	2.173	2.188	2.136	2.127
ROUGE1	**23.54**	19.02	20.05	19.86	20.6	21.03	20.25	20.56	20.48	21.05
ROUGE2	**5.32**	3.18	3.61	3.96	4.11	4.42	4.11	4.39	4.12	4.57
ROUGELsum	**20.71**	16.68	17.24	17.36	18.09	18.35	17.9	18.15	17.98	18.51

Source Code Summarization - SQL, small model, batch size: 256

Steps	1000	2000	5000	10000	15000	20000
BLEU	**1.751**	1.513	0.919	1.108	1.1027	0.946
ROUGE1	**18.69**	17.22	14.91	14.34	14.46	13.98
ROUGE2	**3.86**	3.05	2.17	2.14	2.18	2.06
ROUGELsum	**17.06**	15.54	13.43	12.99	13.02	12.66

Source Code Summarization - SQL, base model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	**2.253**	1.512	1.226	1.23	0.885	0.931	0.817	0.86	0.977	1.15
ROUGE1	**19.94**	16.43	15.6	15.98	14.83	14.63	15.04	14.62	14.8	14.52
ROUGE2	**4.41**	3.39	2.69	2.76	2.35	2.21	2.32	2.19	2.34	2.27
ROUGELsum	**17.97**	15.08	14.06	14.33	13.3	13.15	13.48	13.13	13.28	13.04

Source Code Summarization - SQL, large model, batch size: 256

Steps	100	200	300	400	500	600
BLEU	2.025	**2.172**	1.33	1.068	0.819	1.129
ROUGE1	19.2	**20.37**	18.37	14.43	15.56	15.3
ROUGE2	**4.5**	4.27	3.43	2.5	2.25	2.77
ROUGELsum	17.1	**18.2**	16.68	13.41	14.13	13.97

Source Code Summarization - CSharp, small model, batch size: 256

Steps	1000	2000	3000	4000	5000	10000	15000	20000
BLEU	3.291	**3.599**	2.985	3.001	2.949	2.382	2.499	2.325
ROUGE1	22.26	**22.88**	21	20.9	20.86	19.02	18.73	18.28
ROUGE2	5.63	**5.93**	4.98	5.08	4.9	4.03	3.86	3.69
ROUGELsum	20.57	**20.98**	19.19	19.2	18.97	17.25	16.92	16.53

Source Code Summarization - CSharp, base model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	**3.835**	3.704	3.729	2.714	2.401	2.48	2.955	2.957	2.464	2.745
ROUGE1	**23.27**	22.94	23.41	20.72	19.47	20.36	21.02	21.21	19.7	19.68
ROUGE2	**6.18**	5.91	6.17	4.8	4.14	4.68	4.87	4.95	4.31	4.37
ROUGELsum	**21.36**	20.89	21.36	18.9	17.61	18.55	19.01	19.18	17.81	17.85

Source Code Summarization - CSharp, large model, batch size: 256

Steps	100	200	300	400	500	600
BLEU	3.379	**4.024**	3.933	3.304	3.503	3.07
ROUGE1	22.72	**23.68**	23.4	21.64	22.67	20.94
ROUGE2	6.07	**6.4**	6.34	5.59	5.89	4.86
ROUGELsum	21.18	**21.89**	21.59	19.93	20.68	18.96

Code Comment Generation, small model, batch size: 256

Steps	50000	150000	250000	350000	450000	500000	550000	600000	650000	700000	750000	800000	850000	900000	950000	1000000
BLEU	32.449	41.859	45.569	48.47	50.036	50.566	50.948	51.233	51.289	51.475	51.263	51.454	51.569	51.502	**51.697**	51.49
ROUGE1	51.63	55.46	56.76	57.56	58.15	58.15	58.16	58.24	58.3	58.51	**58.71**	58.66	58.51	58.46	58.57	58.58
ROUGE2	38.52	43.82	45.67	46.77	47.36	47.48	47.56	47.62	47.71	47.85	**47.95**	47.89	47.88	47.83	47.85	47.78
ROUGELsum	50.41	54.36	55.67	56.47	57.08	57.11	57.1	57.16	57.23	57.42	**57.66**	57.57	57.43	57.41	57.48	57.52

Code Comment Generation, base model, batch size: 256

Steps	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
BLEU	26.74	36.605	42.734	47.416	50.696	51.565	52.024	**52.41**	52.163	52.351
ROUGE1	49.95	54.11	56.37	57.91	58.85	59.42	49.57	**59.73**	59.59	59.54
ROUGE2	36.35	41.36	44.49	46.46	47.6	48.25	48.53	**48.57**	48.52	48.38
ROUGELsum	48.71	52.84	55.21	56.64	57.66	58.27	58.37	**58.5**	58.43	58.31

Code Comment Generation, large model, batch size: 256

Steps	5000	20000	30000	40000	50000	60000	70000	80000	90000	100000
BLEU	23.148	51.905	52.491	52.566	52.507	52.604	**52.837**	52.551	52.415	52.632
ROUGE1	43.14	59.92	60.25	60.2	59.95	**60.35**	60.23	59.94	60.04	60.1
ROUGE2	28.88	48.79	49.98	49.17	48.93	**49.25**	49.23	48.82	48.95	48.94
ROUGELsum	41.85	58.82	59	59.16	58.86	**59.25**	59.12	58.87	58.97	59.01

Git Commit Message Generation, small model, batch size: 256

Steps	5000	10000	15000	20000	25000
BLEU	**44.762**	44.698	44.178	44.37	44.201
ROUGE1	**47.92**	47.85	47.13	47.76	47.38
ROUGE2	35.57	**35.83**	35.23	35.49	35.34
ROUGELsum	**47.45**	47.26	46.54	47.2	46.79

Git Commit Message Generation, base model, batch size: 256

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	44.8	44.893	44.549	44.719	44.536	44.732	44.897	**45.004**	44.689	44.613
ROUGE1	**48.76**	48.62	48.66	48.72	48.31	48.71	48.65	48.47	48.48	48.25
ROUGE2	**36.14**	35.89	35.91	35.74	35.56	36.02	36.05	35.76	35.72	35.62
ROUGELsum	**48.18**	47.96	48.13	48.17	47.74	48.15	48.01	47.07	47.95	47.73

Git Commit Message Generation, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	44.283	44.953	44.867	44.889	44.963	45.061	44.885	44.989	**45.366**	44.953
ROUGE1	48.57	49.04	48.84	48.94	49.3	48.99	48.7	49.25	**49.41**	49.1
ROUGE2	35.37	36.16	35.81	36.18	36.52	36.09	36.18	36.42	**36.4**	36.39
ROUGELsum	47.95	48.37	48.2	48.24	48.63	48.48	48.12	48.56	**48.83**	48.52

API Sequence Generation, small model, batch size: 256

Steps	50000	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000	1050000	1100000	1150000	1200000	1250000	1300000	1350000	1400000	1450000
BLEU	60.411	63.813	66.706	67.882	68.315	68.952	69.375	69.367	69.971	70.275	70.275	70.52	70.563	70.662	70.464	70.706	70.495	**70.925**	70.849	70.752
ROUGE1	68.24	71.26	73.62	74.69	75.47	75.88	76.24	76.36	76.91	77.23	77.41	77.3	77.35	77.4	77.34	77.6	77.59	77.74	**77.77**	77.72
ROUGE2	57.84	61.43	64.17	65.27	66.21	66.73	67.1	67.24	67.8	68.23	68.41	68.33	68.41	68.49	68.51	68.69	68.7	68.86	**68.92**	68.82
ROUGELsum	68.22	71.24	73.6	74.64	75.45	75.86	76.23	76.34	76.88	77.21	77.41	77.29	77.32	77.38	77.3	77.57	77.56	77.73	**77.75**	77.7

API Sequence Generation, base model, batch size: 256

Steps	10000	20000	50000	100000	150000	200000	250000	260000	270000	280000	290000	300000	310000	320000	330000	340000	350000	360000
BLEU	56.311	61.422	66.306	69.51	71.521	71.625	72.924	72.563	72.618	72.919	73.2	73.068	73.18	73.226	73.075	**73.647**	73.115	73.565
ROUGE1	64.52	69.02	73.72	76.68	78.42	79.01	79.58	79.63	79.8	79.98	80.23	80.17	80.26	80.38	80.49	**80.64**	80.48	80.52
ROUGE2	53.64	58.83	64.15	67.62	69.6	70.39	71.16	71.2	71.43	71.64	71.74	71.7	71.99	72.05	72.17	**72.4**	72.1	72.26
ROUGELsum	64.5	69	73.69	76.66	78.42	78.99	79.58	79.62	79.77	79.97	80.21	80.16	80.24	80.36	80.49	**80.627**	80.49	80.5

API Sequence Generation, large model, batch size: 256

Steps	20000	40000	60000	80000	100000	120000	140000	160000	180000	200000
BLEU	66.752	70.263	71.595	72.949	73.425	74.107	74.061	74.278	**74.379**	74.322
ROUGE1	74.29	77.32	79.11	80.16	80.7	81.1	81.35	81.59	**81.67**	81.58
ROUGE2	64.83	68.46	70.5	71.74	72.44	72.95	73.25	73.44	**73.69**	73.53
ROUGELsum	74.27	77.31	79.11	80.16	80.67	81.09	81.38	81.59	**81.69**	81.58

Program Synthesis, small model, batch size: 256

Steps	5000	10000	15000	20000
BLEU	**94.657**	94.619	94.613	94.639
ROUGE1	**99.21**	99.2	99.2	99.2
ROUGE2	**98.87**	98.83	98.82	98.85
ROUGELsum	**99.03**	99.02	99.03	99.03
Accuracy	**92.384**	92.301	92.319	92.365

Program Synthesis, base model, batch size: 256

Steps	6000	12000	18000	25000	30000	40000	45000	50000
BLEU	94.62	**94.664**	94.578	94.659	94.653	94.644	94.663	94.652
ROUGE1	99.2	**99.21**	**99.21**	99.2	**99.21**	**99.21**	**99.21**	99.2
ROUGE2	98.85	98.89	98.88	98.89	98.87	98.86	**98.91**	98.89
ROUGELsum	99.03	99.05	99.05	99.05	99.04	99.03	**99.06**	99.05
Accuracy	92.319	92.476	90.147	92.476	92.458	92.319	**92.55**	92.448

Program Synthesis, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000
BLEU	94.595	94.641	94.616	94.626	94.635	94.619	**94.642**	94.627
ROUGE1	99.18	**99.2**	99.19	**99.2**	**99.2**	**99.2**	**99.2**	**99.2**
ROUGE2	98.8	**98.86**	98.84	98.84	98.84	98.85	98.85	98.85
ROUGELsum	99.02	99.03	99.02	99.03	99.03	**99.04**	99.03	99.03
Accuracy	91.774	92.301	92.236	92.328	92.328	92.301	**92.375**	92.347

A.3. Multi-task Learning

Multi-task Learning, large model, batch size: 4096

Task	Steps	20000	40000	60000	80000	100000	120000	140000	160000	180000	200000	220000	240000	260000
Code Documentation Generation - Python	BLEU	7.268	7.19	7.136	7.576	**7.907**	7.417	7.19	7.251	7.534	7.607	7.425	7.749	7.639
	ROUGE1	35.12	35.61	35.44	**35.89**	35.42	35.65	34.84	34.87	34.97	34.84	34.43	34.84	34.63
	ROUGE2	13.43	14.01	14.03	**14.04**	13.96	14.01	13.46	13.4	13.58	13.41	13.36	13.43	13.15
	ROUGELsum	32.83	33.42	33.32	**33.6**	33	33.38	32.55	32.5	32.6	32.48	32.18	32.4	32.19
Code Documentation Generation - Java	BLEU	10.36	11.285	11.942	12.286	12.862	12.886	12.871	12.519	**14.398**	13.86	13.136	14.12	14.392
	ROUGE1	38.84	39.09	40.01	39.63	40.13	40.07	39.99	39.18	40.2	40.09	39.27	**40.26**	40.2
	ROUGE2	18.68	19.58	20.35	18.87	20.19	20.35	20.39	19.7	**20.78**	20.51	19.97	20.77	20.6
	ROUGELsum	36.62	36.99	37.96	37.42	37.76	37.86	37.78	37.11	**38**	37.77	37.08	37.99	37.84
Code Documentation Generation - Go	BLEU	12.052	13.023	12.443	**13.662**	13.095	13.642	12.903	13.128	13.103	13.578	12.724	13.609	13.846
	ROUGE1	47.37	47.62	47.76	47.74	47.38	**47.93**	47.36	47.23	46.58	47.09	46.57	46.89	46.85
	ROUGE2	24.8	25.14	**25.27**	24.7	24.84	25.26	24.92	24.59	23.83	24.48	24.35	24.45	24.39
	ROUGELsum	45.45	45.69	**45.9**	45.44	45.32	45.83	45.38	45.23	44.39	44.89	44.59	44.81	44.72
Code Documentation Generation - Php	BLEU	9.745	10.993	10.445	12.318	11.772	12.158	11.798	12.183	13.541	13.111	13.438	13.277	**13.708**
	ROUGE1	41.63	42.13	42.1	42.74	42.69	42.7	42.68	42.2	42.63	42.62	42.39	**42.92**	42.5
	ROUGE2	18.09	18.6	18.88	19.21	19.38	19.53	19.47	19.08	19.57	19.44	19.29	**19.85**	19.4
	ROUGELsum	39.81	40.25	40.36	40.75	40.74	40.82	40.9	40.33	40.62	40.63	40.37	**41**	40.47
Code Documentation Generation - Ruby	BLEU	3.953	4.311	4.051	**4.753**	4.499	4.424	4.128	4.044	4.418	4.436	4.182	4.443	4.741
	ROUGE1	**32.94**	32.77	32.6	32.83	32.69	32.46	31.82	32.13	31.79	31.36	31.99	31.77	32.13
	ROUGE2	**12.7**	12.32	12.57	12.44	12.09	12.33	11.91	11.34	11.49	11.18	11.64	11.35	11.65
	ROUGELsum	**30.8**	30.47	30.54	30.61	30.42	30.21	29.62	29.76	29.54	29.28	29.92	29.4	29.49
Code Documentation Generation - Javascript	BLEU	4.331	4.706	4.536	5.218	5.554	5.139	5.279	5.109	5.687	5.676	5.535	**6.238**	5.917
	ROUGE1	30.07	30.56	30.81	30.82	30.51	31	30.93	30.48	30.7	**31.09**	30.68	31.08	30.77
	ROUGE2	10.3	11.05	11.36	10.99	11.02	11.42	11.46	11.17	11.55	11.32	11.41	**11.56**	11.24
	ROUGELsum	28.21	28.76	29.13	28.88	28.62	**29.19**	29.08	28.61	28.83	29.16	28.75	28.98	28.79
Source Code Summarization - Python	BLEU	3.89	3.867	3.89	4.08	3.793	4.021	3.89	**4.141**	4.045	3.833	3.781	4.058	3.741
	ROUGE1	27.61	26.18	26.27	**27.63**	26.67	27.08	26.22	26.92	26.56	26.06	25.78	26.53	25.55
	ROUGE2	7.86	7.08	6.9	7.88	7.15	**7.94**	7.08	7.35	7.36	7.08	6.89	7.27	6.72
	ROUGELsum	**24.37**	23.35	23.25	24.33	23.6	24.15	23.26	23.6	23.45	22.96	22.94	23.35	22.57
Source Code Summarization - SQL	BLEU	1.717	2.04	1.869	2.128	2.016	**2.262**	2.07	1.926	1.993	1.812	2.052	1.882	1.903
	ROUGE1	19.18	18.84	18.43	**20.57**	19.37	20.21	19.54	19.46	19.27	19.15	19.17	19.47	19.11
	ROUGE2	3.87	3.88	3.49	4.38	3.85	**4.51**	4.24	3.93	3.94	3.94	4.16	3.99	3.86
	ROUGELsum	17.17	17.07	16.61	18.15	17.44	**18.19**	17.39	17.48	17.48	17.36	17.62	17.54	17.31
Source Code Summarization - CSharp	BLEU	4.009	3.771	3.757	4.315	4.059	**4.327**	3.99	4.114	4.208	4.147	3.968	4.19	4.19
	ROUGE1	24.26	23.69	23.26	**24.92**	24.05	24.7	23.72	24.15	24.53	23.98	23.21	23.85	23.85
	ROUGE2	6.55	6.36	5.64	6.83	6.39	**6.91**	6.31	6.45	6.62	6.27	5.91	6.41	6.25
	ROUGELsum	21.91	21.71	21.37	22.57	21.93	**22.62**	21.69	22.06	22.37	21.86	21.33	21.83	21.75
Code Comment Generation	BLEU	19.308	26.184	30.428	33.509	36.096	37.533	38.306	39.342	40.773	41.718	41.421	43.14	**43.712**
	ROUGE1	42.64	46.92	49.75	51.65	53.09	53.06	53.67	54.68	55	55.82	55.6	**56.41**	56.14
	ROUGE2	27.68	33.17	36.55	38.6	40.32	40.58	41.41	42.71	42.85	43.81	43.95	**44.54**	44.39
	ROUGELsum	41.32	45.72	48.6	50.47	51.91	51.92	52.58	53.6	53.87	54.68	54.58	**55.29**	55.01
Git Commit Message Generation	BLEU	37.913	39.705	39.758	40.468	40.124	41.157	41.055	41.181	41.383	41.698	**41.895**	41.729	41.8
	ROUGE1	41.52	43.65	44.14	44.83	44.54	46.35	46.05	46.35	46.76	46.67	**46.94**	46.88	46.9
	ROUGE2	28.8	30.22	30.82	31.36	31.28	32.18	32.29	32.49	32.83	33.1	**33.35**	33.14	32.99
	ROUGELsum	41.14	43.15	43.75	44.34	44.09	45.76	45.54	45.78	46.19	46.1	**46.49**	46.38	46.37
API Sequence Generation	BLEU	65.764	69.491	71.02	71.754	72.225	72.67	73.018	73.076	73.539	73.01	73.457	73.547	**73.548**
	ROUGE1	37.07	76.35	77.86	78.61	79.15	79.54	79.78	80.21	80.48	80.4	80.68	**80.8**	80.73
	ROUGE2	63.56	67.43	69.09	70.02	70.54	71.11	71.41	71.84	72.27	72.01	72.37	**72.54**	72.46
	ROUGELsum	73.05	76.34	77.83	78.6	79.18	79.52	79.78	80.23	80.45	80.42	80.66	**80.78**	70.72
Program Synthesis	BLEU	93.891	94.41	94.6	94.592	94.64	64.605	94.627	94.654	**94.645**	94.616	94.644	94.62	94.609
	ROUGE1	98.47	99.02	99.19	99.16	99.2	99.19	99.2	99.2	99.2	99.2	**99.21**	99.19	99.19
	ROUGE2	97.54	98.53	98.81	98.79	98.87	98.83	98.84	**98.91**	98.87	98.85	98.9	98.81	98.84
	ROUGELsum	98.23	98.84	99.01	98.99	99.03	99.02	99.03	99.05	99.03	99.02	**99.06**	99.01	99.01
	Accuracy	82.79	90.923	91.94	91.866	92.171	92.079	92.171	92.245	92.319	92.18	**92.439**	92.236	92.227

Multi-task Learning, small model, batch size: 4096

Task	Steps	20000	40000	60000	80000	100000	120000	140000	160000	180000	200000	220000	240000	260000	280000	300000	320000	340000	360000	380000	400000	420000	440000	460000	480000	500000
Code Documentation Generation - Python	BLEU	5.932	4.783	5.682	5.178	6.232	5.624	6.067	6.333	6.345	6.481	6.089	6.486	6.098	6.219	6.136	6.114	6.277	6.178	6.134	6.17	**6.614**	6.225	6.397	6.091	6.089
	ROUGE1	32.96	32.28	33.05	32.75	33.92	33.13	33.29	33.87	34.21	34.48	34.02	34.33	34.16	33.76	34.17	33.81	33.73	34.26	34.16	34.24	**34.52**	34.46	34.31	34.12	34.2
	ROUGE2	11.76	11.55	12.25	12.02	12.62	12.41	12.38	12.78	12.93	12.96	12.7	13.03	12.88	12.76	12.89	12.78	12.69	12.95	12.98	13.01	**13.1**	13.03	13.02	13.01	12.99
	ROUGELsum	30.84	30.65	31.15	30.99	31.81	31.27	31.28	31.8	32.16	32.32	32	32.16	32.17	31.69	32.13	31.78	31.7	32.2	32.17	32.22	32.34	**32.36**	32.2	32.17	32.19
Code Documentation Generation - Java	BLEU	6.975	5.479	7.346	6.852	8.422	7.43	8.372	8.709	8.607	8.416	8.583	8.665	8.293	8.634	8.605	8.349	8.679	8.685	8.272	8.675	**8.829**	8.395	8.723	8.458	8.471
	ROUGE1	36.1	34.02	35.98	35.63	36.85	36.31	36.93	37.35	37.25	37.5	37.14	37.34	37.01	37.01	37.16	37.04	37.25	37.64	37.17	**37.65**	37.32	37.02	37.6	37.33	37.31
	ROUGE2	15.97	14.74	16.52	16.36	16.98	17.06	17.02	**17.8**	17.34	17.43	17.44	17.49	17.51	17.2	17.62	17.37	17.6	17.73	17.33	17.71	17.46	17.3	17.61	17.56	17.58
	ROUGELsum	34.04	32.38	34.18	33.86	34.86	34.56	34.94	35.43	35.16	35.36	35.1	35.3	35.08	34.93	35.23	35.15	35.26	35.63	35.18	**35.65**	35.27	35.11	35.52	35.35	35.36
Code Documentation Generation - Go	BLEU	8.226	11.503	11.641	11.682	9.383	11.683	12.368	12.05	**13.173**	12.626	12.134	12.292	12.552	11.845	12.405	12.202	12.484	12.104	11.942	12.08	12.486	12.079	11.536	12.321	12.053
	ROUGE1	43.93	46.11	46.82	46.47	45.43	46.84	47.22	46.82	46.96	46.58	47.53	47.09	47.15	46.69	47.27	47.21	**47.67**	46.8	46.14	47.17	47.4	47.37	47.35	46.29	47.27
	ROUGE2	21.56	24.34	24.84	24.56	23	24.84	24.78	24.75	24.99	24.17	25.32	24.91	25.07	24.8	25	25.07	**25.36**	24.6	24.03	24.89	24.94	25.02	24.92	24.2	25.04
	ROUGELsum	42.18	44.49	45.16	44.9	43.69	45.12	45.33	45.1	45.02	44.69	45.75	45.33	45.45	44.94	45.53	45.47	**45.89**	45	44.49	45.46	45.55	45.6	45.64	44.55	45.46
Code Documentation Generation - Php	BLEU	7.331	6.187	8.243	6.819	8.891	7.093	8.493	8.929	8.28	8.42	8.616	9.46	8.75	8.914	8.01	9.206	8.475	8.449	7.831	8.849	**9.725**	8.288	8.763	7.861	9.019
	ROUGE1	38.22	37.93	38.72	37.91	39.52	38.26	38.24	39.24	39.53	39.61	39.75	39.66	39.59	38.89	39.5	38.73	39.03	39.61	39.42	39.86	40.05	39.72	39.9	39.82	**40.15**
	ROUGE2	15.07	14.4	15.83	15.02	15.98	15.57	15.7	16.58	16.1	16.28	16.28	16.38	16.44	16.13	16.25	16.3	16.37	16.56	16.25	16.55	16.68	16.46	16.56	16.3	**16.76**
	ROUGELsum	36.57	36.52	37.09	36.51	37.73	36.72	36.58	37.49	37.81	37.94	38.03	37.79	37.93	37.13	37.84	37.02	37.37	37.9	37.88	38.17	38.16	38.06	38.1	38.21	**38.44**
Code Documentation Generation - Ruby	BLEU	2.942	2.296	3.285	2.459	3.351	2.818	3.235	3.629	3.583	3.39	3.471	3.671	3.406	3.077	3.362	3.378	3.304	3.276	3.027	3.212	**3.894**	3.055	3.218	3.009	3.248
	ROUGE1	30.41	28.59	29.81	29.09	30.63	29.41	30.19	30.22	30.71	30.66	31.01	30.7	30.76	30.21	30.35	30.6	30.49	30.69	30.78	30.71	**31.31**	30.62	31.19	30.82	30.49
	ROUGE2	9.94	9.32	10.21	10.05	10.79	10.03	10.68	10.7	11.1	10.86	11.09	11	10.84	10.75	10.94	10.58	10.73	11.19	11.09	11.06	11.5	10.99	**11.7**	11.24	11.21
	ROUGELsum	28.25	26.97	27.92	27.48	28.51	27.63	28.4	28.33	28.69	28.7	29.02	28.75	28.87	28.28	28.53	28.74	28.52	28.87	28.96	28.94	29.24	28.7	**29.35**	29.05	28.63
Code Documentation Generation - Javascript	BLEU	3.51	2.385	3.435	2.537	3.602	3.142	3.509	3.867	3.614	3.71	3.459	**4.013**	3.633	3.413	3.599	3.888	3.784	3.359	3.295	3.181	3.959	3.133	3.061	3.119	3.522
	ROUGE1	27.73	26.79	27.62	27.21	28.63	27.96	28.27	28.76	28.89	29.08	28.84	29.07	28.76	28.57	28.74	28.66	28.8	28.93	28.85	28.76	**29.08**	28.85	28.9	28.6	29.01
	ROUGE2	8.8	8.53	9.04	8.91	9.52	9.31	9.54	9.65	9.74	9.9	9.77	**9.96**	9.75	9.61	9.71	9.72	9.77	9.71	9.73	9.7	9.73	9.86	9.69	9.45	9.89
	ROUGELsum	26.04	25.45	26.08	25.93	26.94	26.5	26.69	27.07	27.12	27.34	27.13	27.36	27.22	26.89	27.16	26.91	27.13	27.23	27.21	27.12	27.29	27.28	27.28	27.14	**27.41**
Source Code Summarization - Python	BLEU	3.388	3.054	3.18	3.216	3.709	3.254	3.511	3.678	3.619	3.437	3.56	3.479	3.602	3.377	3.595	**3.712**	3.527	3.474	3.481	3.407	3.571	3.489	3.635	3.531	3.492
	ROUGE1	25.72	24.09	24.62	24.94	26.54	24.46	25.89	26.07	25.68	25.84	25.53	26.28	26.42	24.52	**26.85**	26.25	25.27	25.52	25.77	25.63	25.96	26.13	26.02	24.84	25.67
	ROUGE2	7	6.33	6.28	6.71	7.28	6.37	7.02	7.01	7.09	6.87	6.85	7.24	7.2	6.4	**7.46**	7.31	6.87	6.88	7.14	7.1	7.12	7.02	7.26	6.65	6.74
	ROUGELsum	22.91	21.76	22	22.44	23.49	22.01	22.93	22.93	22.81	22.84	22.66	23.34	23.31	21.84	**23.7**	23.36	22.5	22.72	12.99	23.07	23.08	23.11	23.23	22.36	22.76
Source Code Summarization - SQL	BLEU	1.653	1.831	1.474	1.422	1.623	1.583	1.766	1.677	1.493	1.576	1.618	1.577	1.756	1.68	1.841	1.8	1.606	1.493	1.668	1.806	1.758	1.804	**1.953**	1.739	1.742
	ROUGE1	18.46	17.8	17.32	17.32	17.89	17.65	18.56	18.84	18.08	17.78	17.92	18.12	18.91	17.79	18.6	18.91	18.15	17.9	18.24	18.6	18.57	18.27	**19.05**	18.08	18.4
	ROUGE2	3.66	3.61	3.02	3.08	3.74	3.38	3.83	3.86	3.46	3.45	3.32	3.3	3.8	3.4	3.84	4.08	3.44	3.29	3.55	3.85	3.6	3.77	**4.2**	3.79	3.74
	ROUGELsum	16.32	15.89	15.44	15.65	16.16	16.03	16.53	16.67	16.05	15.87	16.16	16.24	16.84	15.91	16.66	16.8	16.16	16.1	16.34	16.67	16.67	16.48	**17.03**	16.37	16.46
Source Code Summarization - CSharp	BLEU	3.038	2.942	3.233	3.066	3.263	3.183	3.347	3.467	3.502	3.348	3.29	3.654	3.682	3.082	**3.742**	3.596	3.533	3.503	3.469	3.319	3.57	3.599	3.741	3.286	3.539
	ROUGE1	21.82	21.1	21.77	21.63	22.66	21.94	22.48	23.13	22.88	22.64	22.56	23.4	23.24	21.51	23.43	22.84	22.63	22.69	22.86	22.62	23.25	22.95	**23.55**	22.05	22.34
	ROUGE2	5.42	5.11	5.06	5.22	5.71	5.39	5.63	5.8	5.84	5.65	5.55	6.03	6.02	4.89	6.16	5.82	5.59	5.73	5.87	5.57	5.94	5.85	**6.27**	5.46	5.35
	ROUGELsum	19.98	19.51	19.85	19.93	20.75	20.11	20.43	20.95	20.79	20.61	20.56	21.25	21.12	19.73	21.34	20.87	20.6	20.75	20.88	20.6	21.18	20.91	**21.46**	20.23	20.35
Code Comment Generation	BLEU	4.99	6.103	7.426	6.582	8.343	7.899	9.083	9.638	9.523	9.565	9.643	9.812	9.684	9.829	9.968	9.614	10.231	10.041	9.985	10.373	10.228	10.167	10.395	**10.475**	10.404
	ROUGE1	34.11	34.19	36.16	36.46	37.21	37.07	37.8	37.89	37.95	38.53	38.37	38.35	38.48	38.13	38.6	38.74	38.86	**38.95**	38.8	38.91	38.8	38.59	38.81	38.87	38.91
	ROUGE2	17.2	17.94	19.43	19.75	20.33	20.4	20.93	20.97	21.21	21.51	21.48	21.59	21.66	21.62	21.72	22.05	21.88	22.11	21.98	22.14	21.99	21.97	22.25	22.26	**22.37**
	ROUGELsum	32.79	33.06	34.88	35.31	35.93	35.9	36.46	36.55	36.64	37.21	37.07	37.01	37.23	36.89	37.33	37.47	37.55	**37.68**	37.53	37.59	37.52	37.35	37.53	37.61	37.66
Git Commit Message Generation	BLEU	25.458	36.499	36.429	36.389	37.052	36.909	36.907	36.777	37.273	37.09	37.233	37.192	37.252	37.246	37.161	37.219	37.225	37.092	37.256	37.299	37.061	37.025	**37.408**	37.183	37.135
	ROUGE1	35.36	37.61	37.76	37.5	37.88	38.4	38.65	38.8	38.62	38.6	38.91	38.69	38.89	**39.6**	38.78	38.67	39.25	38.98	38.43	39.28	39.24	39.05	39.36	38.97	38.83
	ROUGE2	23.68	25.2	25.55	25.75	25.84	25.65	26.09	26.36	26.27	26.3	26.16	26.03	26.39	**26.89**	26.32	26.07	26.49	26.65	26.14	26.61	26.24	26.38	26.77	26.78	26.28
	ROUGELsum	35.15	37.25	37.43	37.29	37.57	38.14	38.24	38.46	38.25	38.24	38.59	38.51	38.69	**39.33**	38.51	38.35	38.97	38.7	38.14	38.96	38.89	38.76	39.08	38.71	38.6
API Sequence Generation	BLEU	50.508	54.141	55.874	55.675	57.186	57.001	57.304	58.204	58.593	58.307	58.553	58.622	58.772	58.878	58.993	58.983	59.299	59.436	59.242	59.496	59.714	59.445	59.708	**59.807**	59.687
	ROUGE1	58.3	61.63	63.17	63.81	64.6	64.52	65.26	65.46	65.82	65.72	65.85	66.1	66.31	66.46	66.37	66.65	66.83	66.91	66.84	66.96	67.01	66.96	67.15	67.09	**67.22**
	ROUGE2	46.88	50.86	52.66	53.33	54.07	54.18	54.72	54.92	55.39	55.35	55.51	55.84	55.94	55.96	56.05	56.29	56.65	56.59	56.48	56.66	56.75	56.74	56.95	56.73	**56.99**
	ROUGELsum	58.25	61.62	63.15	63.8	64.57	64.48	65.28	65.44	65.77	65.69	65.81	66.08	66.31	66.41	66.34	66.62	66.81	66.89	66.8	66.91	67.01	66.94	67.13	67.06	**67.19**
Program Synthesis	BLEU	91.109	92.338	93.203	93.313	93.591	93.78	93.689	93.785	93.795	93.843	93.912	93.846	93.903	93.925	93.945	93.936	93.906	93.971	93.937	93.707	93.917	**93.986**	93.911	93.942	93.985
	ROUGE1	96.84	97.82	97.94	98.32	98.16	98.24	98.21	98.55	98.51	98.52	98.28	98.67	98.64	98.64	**98.7**	98.25	98.53	98.23	98.42	98.66	98.56	98.6	98.48	98.53	98.55
	ROUGE2	94.98	96.45	96.87	97.25	97.1	97.25	97.18	97.59	97.55	97.55	97.32	97.75	97.72	97.75	**97.78**	97.31	97.6	97.25	97.46	97.73	97.6	97.69	97.54	97.58	97.6
	ROUGELsum	96.43	97.52	97.7	98.07	97.89	98.02	97.97	98.33	98.27	98.29	98.06	98.45	98.43	98.44	**98.48**	98.04	98.31	97.99	98.19	98.45	98.33	98.39	98.26	98.31	98.32
	Accuracy	61.272	78.547	76.819	82.253	82.096	82.143	79.203	82.91	82.558	83.178	82.836	83.76	84.703	84.943	**85.396**	83.584	84.903	83.289	83.658	83.483	83.816	84.675	83.233	83.132	84.684

Multi-task Learning, base model, batch size: 4096

Task	Steps	20000	40000	60000	80000	100000	120000	140000	160000	180000	200000	220000	240000	260000	280000	300000	320000	340000	360000	380000	400000	420000	440000	460000	480000	500000
	BLEU	6.632	6.394	6.908	7.092	6.978	6.992	7.079	**7.57**	6.826	6.64	7.024	7.458	7.362	7.356	6.897	7.122	7.227	7.368	7.263	7.345	7.493	7.012	7.154	7.275	7.448
Code Documentation	ROUGE1	34.25	34.72	35.41	35.35	35.09	35.37	35.42	35.84	35.36	35.44	35.64	35.8	36.13	35.96	35.89	35.68	36	36.12	35.8	36.01	**36.17**	35.89	35.75	36.03	36.13
Generation - Python	ROUGE2	12.92	13.37	13.89	13.84	13.82	13.92	14.08	14.15	14.07	14.03	14.08	14.18	14.34	14.22	14.34	14.26	14.36	14.43	14.39	14.45	**14.5**	14.31	14.25	14.46	14.43
	ROUGELsum	32.03	32.72	33.25	33.12	32.93	33.13	33.3	33.51	33.26	33.41	33.5	33.55	33.86	33.72	33.81	33.55	33.81	33.9	33.62	33.84	**33.91**	33.74	33.57	33.85	33.88
	BLEU	8.66	8.913	9.914	10.837	11.315	11.001	11.439	12.08	12.119	11.118	12.339	12.363	12.414	12.296	11.666	12.17	12.247	12.34	12.7	12.485	12.347	12.244	12.396	**12.873**	12.735
Code Documentation	ROUGE1	37.61	37.36	38.34	38.89	38.87	39.09	39.05	39.41	39.6	38.99	39.58	39.88	39.88	39.72	39.25	39.63	39.59	39.76	40.13	39.9	39.88	39.77	39.92	**40.27**	40.07
Generation - Java	ROUGE2	17.41	17.65	18.47	19.07	19.02	19.53	19.62	19.77	19.98	19.75	19.78	20.17	20.27	20.25	19.77	20.01	20.14	20.16	20.59	20.41	20.23	20.26	20.48	**20.81**	20.49
	ROUGELsum	35.45	35.34	36.23	36.76	36.69	36.97	37.05	37.26	37.47	37.07	37.36	37.65	37.72	37.66	37.17	37.47	37.45	37.61	37.97	37.71	37.67	37.69	37.85	**38.12**	37.82
	BLEU	9.117	11.283	12.646	13.158	12.098	12.308	12.613	13.171	13.079	12.201	12.949	12.534	13.187	13.007	12.679	11.899	**13.733**	12.754	13.232	13.057	12.899	12.868	12.868	13.619	13.319
Code Documentation	ROUGE1	45.3	46.43	47.52	47.27	47.16	47.88	47.74	47.17	48.04	47.72	47.87	47.61	47.27	47.83	47.97	47.22	47.63	48.04	47.69	48.04	48.01	**48.11**	47.89	47.28	47.61
Generation - Go	ROUGE2	23.16	24.32	24.98	24.68	24.79	25.4	25.11	24.69	25.59	25.41	25.55	25.07	24.74	25.47	25.55	24.82	25.35	25.7	25.58	**25.88**	25.62	25.64	25.52	25.1	25.33
	ROUGELsum	43.59	44.65	45.56	45.34	45.39	45.97	45.81	45.23	46.17	45.93	46.05	45.85	45.4	45.98	46.11	45.47	45.77	46.25	45.9	46.27	46.2	**46.28**	46.12	45.44	45.79
	BLEU	7.325	8.753	11.256	10.438	11.182	9.618	10.732	10.868	10.699	9.653	11.587	11.797	11.213	10.625	9.924	11.398	10.892	11.895	10.524	10.641	10.658	10.622	11.669	11.405	**12.01**
Code Documentation	ROUGE1	39.46	40.37	41.67	41.14	41.23	41.2	41.58	41.76	41.95	41.43	42.32	42.01	42.19	41.85	41.72	41.86	42.1	**42.57**	42.12	42.04	42.31	42.27	41.85	42.17	42.52
Generation - Php	ROUGE2	16.16	16.74	18.4	18.25	18.26	18.26	18.45	18.79	18.96	18.37	19.16	19.24	19.14	18.86	18.64	18.98	19.07	**19.35**	19.2	19.02	19.12	18.94	19.05	19.16	19.27
	ROUGELsum	37.94	38.78	39.76	39.41	39.32	39.47	39.78	39.93	40.19	39.82	40.49	40.08	40.34	40.16	40.07	40	40.39	**40.63**	40.33	40.35	40.6	40.55	40.02	40.36	40.54
	BLEU	3.307	3.324	4.125	4.185	4.074	3.721	3.933	**4.905**	3.493	3.91	4.356	4.468	4.464	4.455	3.976	4.408	4.143	3.934	4.043	4.351	4.259	4.012	4.357	4.373	4.301
Code Documentation	ROUGE1	30.84	30.9	32.45	32.33	32.11	32.36	31.87	**33.22**	31.96	31.65	32.46	32.48	32.02	32.66	31.97	32.74	32.1	32.31	32.56	32.88	32.56	32.55	32.55	32.3	32.75
Generation - Ruby	ROUGE2	11.6	11.32	12.37	12.57	12.39	12.5	12.11	**12.61**	12.04	11.98	12.32	12.28	12.01	12.43	12.14	12.55	12.11	11.89	12.42	12.45	12.61	12.42	12.3	12.31	12.59
	ROUGELsum	29.14	29.06	30.39	30.37	30.17	30.41	29.85	**30.92**	29.96	29.68	30.3	30.32	29.84	30.51	29.89	30.34	29.98	30.04	30.45	30.63	30.32	30.27	30.25	30.12	30.6
	BLEU	3.419	3.739	4.572	4.205	4.543	4.289	4.475	**5.156**	4.305	4	4.862	5.069	4.702	4.945	4.121	4.774	4.523	4.512	4.516	4.875	4.732	4.291	4.896	4.517	4.84
Code Documentation	ROUGE1	28.79	29.19	29.8	30.02	30.02	30.2	30.29	30.8	30.44	30.16	30.67	**30.99**	30.61	30.92	30.41	30.59	30.69	30.95	30.71	30.93	30.79	30.95	30.83	30.69	30.78
Generation - Javascript	ROUGE2	9.61	10.35	10.82	10.67	10.8	10.95	10.92	11.08	10.92	11.01	11.01	11.32	11.08	**11.46**	10.96	11.27	11.14	11.25	11.18	11.19	11.09	11.29	11.19	11.03	11.18
	ROUGELsum	27.02	27.65	28.09	28.17	28.23	28.55	28.52	28.86	28.74	28.52	28.87	29.08	28.76	29.05	28.65	28.79	28.88	29.11	29.01	29.1	28.89	**29.12**	28.98	28.81	28.86
	BLEU	3.622	4.071	3.567	3.823	3.865	3.924	4.072	3.777	3.994	3.704	3.991	4.025	**4.48**	3.696	3.884	4.108	3.766	3.824	3.928	3.896	3.85	3.94	3.706	4.024	3.79
Source Code	ROUGE1	25.58	26.35	25.12	26.35	27.41	26.58	27.05	26.98	27.34	25.87	26.67	26.61	**27.81**	26.06	26.54	26.83	26.16	26.37	26.48	25.9	26.1	26.4	25.93	26.67	26.43
Summarization - Python	ROUGE2	6.85	7.41	6.66	7.38	7.76	7.6	7.66	7.55	7.64	7.04	7.2	7.47	**8.05**	7.08	7.42	7.6	7.18	7.11	7.37	7.08	7.23	7.42	6.98	7.51	7.35
	ROUGELsum	22.86	23.41	22.4	23.44	24.18	23.67	23.98	23.88	24.12	23.02	23.53	23.69	**24.64**	23.15	23.56	23.83	23.14	23.39	23.46	22.97	23.14	23.54	23.09	23.64	23.36
	BLEU	1.545	1.851	1.455	1.754	2.128	1.886	1.931	2.026	2.101	1.991	2.025	1.913	2.062	1.774	2.058	**2.18**	2.04	2.084	2.092	2.084	2.037	2.041	1.96	1.987	2.095
Source Code	ROUGE1	17.14	17.58	16.96	18.07	19.19	18.98	18.83	19.15	19.09	18.34	18.75	18.73	19.09	18.65	18.66	19.09	19.23	19.18	19.28	19.32	19.15	19.13	18.39	18.99	**19.53**
Summarization - SQL	ROUGE2	2.84	3.6	3.1	3.6	4.35	4.12	3.97	4.04	4	3.55	3.96	3.8	4.14	3.54	3.86	4.04	3.95	**4.39**	4.12	4.13	3.92	3.88	3.74	3.8	4.1
	ROUGELsum	15.36	16	15.29	16.39	17.29	17.06	16.99	17.24	17.08	16.6	16.85	16.94	17.31	16.65	16.86	17.28	17.41	17.36	17.39	17.49	17.15	17.21	16.63	17.19	**17.65**
	BLEU	3.363	3.646	3.506	3.782	4.025	3.989	3.962	**4.225**	4.088	3.633	3.811	3.82	3.953	3.561	3.759	3.872	3.752	3.76	3.78	3.691	3.658	3.939	3.636	3.919	3.873
Source Code	ROUGE1	22.21	23.02	23.08	23.88	24.37	24.42	23.79	24.71	24.42	23.45	24.11	23.84	**24.82**	23.57	24.18	24.31	24.14	23.77	23.93	23.72	24.04	23.94	23.52	24.12	24.44
Summarization - CSharp	ROUGE2	5.38	6.25	5.51	6.07	6.62	6.41	6.21	6.65	6.29	5.8	6.13	6.2	**6.68**	5.94	6.15	6.36	6.28	5.98	6.13	5.99	6.12	6.16	5.73	6.2	6.28
	ROUGELsum	20.33	21.04	21.1	21.81	22.33	22.24	21.73	22.5	22.22	21.48	21.94	21.76	**22.68**	21.5	22.04	22.14	21.96	21.76	21.84	21.66	21.91	21.94	21.44	22.01	22.23
	BLEU	11.875	15.336	18.379	19.135	19.943	21.359	21.613	22.667	22.336	22.641	24.23	24.542	24.185	24.237	24.233	24.965	25.591	24.655	25.172	25.518	25.504	25.674	25.958	26.448	**26.601**
Code Comment	ROUGE1	38.85	42.14	43.39	44.47	45.01	45.25	45.69	45.93	46.38	46.57	46.92	47.24	47.1	47.65	47.59	47.66	47.83	47.69	47.96	48.23	48.17	48.03	**48.51**	48.4	48.39
Generation	ROUGE2	22.74	26.54	27.7	28.94	29.77	30.05	30.42	30.86	31.2	31.48	31.86	32.32	32.26	32.83	32.91	32.95	32.99	32.99	33.31	33.7	33.59	33.64	33.95	33.84	**34.07**
	ROUGELsum	37.51	40.9	42.1	43.21	43.81	44.03	44.48	44.7	45.22	45.44	45.73	46.03	45.92	46.44	46.42	46.49	46.68	46.59	46.82	47.02	46.98	46.91	**47.38**	47.25	47.24
	BLEU	36.771	38.159	38.462	38.609	38.97	39.013	38.851	38.876	39.476	38.88	39.621	39.517	39.465	39.267	39.781	39.841	**39.894**	39.861	39.726	39.552	39.792	39.815	39.756	39.729	39.743
Git Commit Message	ROUGE1	39.88	41.02	41.83	42.59	42.96	42.97	42.79	43.04	43.57	43.36	43.42	43.98	43.87	43.64	44.1	44.01	44.1	44.2	44.26	44.18	44.11	44.15	44.52	**44.57**	44.27
Generation	ROUGE2	26.64	28.19	28.63	29.15	29.69	29.62	29.64	29.76	30.16	29.94	30.11	30.33	30.36	30.05	30.53	30.42	30.56	30.97	30.59	30.35	30.69	30.6	30.86	**31.13**	30.78
	ROUGELsum	39.46	40.69	41.46	42.2	42.56	42.51	42.41	42.56	43.06	42.89	43	43.48	43.24	43.11	43.58	43.5	43.64	43.73	43.85	43.67	43.65	43.67	44	**44.13**	43.78
	BLEU	59.971	63.724	65.448	65.84	66.262	66.859	67.383	68.14	68.118	68.453	68.181	68.868	68.826	68.995	68.945	69.257	69.086	68.794	69.156	69.347	69.5	69.429	69.45	69.821	**69.843**
API Sequence	ROUGE1	68.05	70.87	72.69	73.16	73.73	74.23	74.55	75.06	75.27	75.51	75.3	75.91	75.74	75.98	75.94	76.37	76.18	76.12	76.22	76.35	76.52	76.47	76.58	**76.71**	76.63
Generation	ROUGE2	57.73	61.13	63.24	63.78	64.33	64.76	65.19	65.84	65.93	66.18	65.99	66.63	66.63	66.88	66.77	67.17	66.95	67.02	67.15	67.2	67.44	67.35	67.56	**67.7**	67.6
	ROUGELsum	68.03	70.87	72.66	73.18	73.69	74.2	74.55	75.07	75.25	75.49	75.26	75.9	75.73	75.97	75.96	76.35	76.13	76.14	76.23	76.34	76.52	76.44	76.54	**76.69**	76.61
	BLEU	93.267	94.001	94.115	94.202	94.25	94.205	94.253	94.291	94.263	94.294	94.242	94.221	94.274	94.235	94.304	94.195	94.278	**94.361**	94.332	94.323	94.256	94.233	94.266	94.342	94.324
	ROUGE1	98.42	98.72	98.78	98.91	98.89	98.93	98.92	98.92	98.91	98.93	98.93	98.92	98.92	98.96	98.96	98.94	98.95	**98.96**	98.94	98.95	98.94	98.96	98.96	98.96	98.95
Program Synthesis	ROUGE2	97.23	97.78	97.94	98.15	98.12	98.18	98.17	98.19	98.2	98.23	98.2	98.16	98.22	98.23	98.25	98.22	98.26	**98.28**	98.25	98.25	98.24	98.27	98.28	98.26	98.28
	ROUGELsum	98.15	98.49	98.57	98.7	98.69	98.75	98.72	98.74	98.73	98.75	98.74	98.73	98.73	98.76	98.77	98.76	98.77	**98.79**	98.77	98.77	98.77	98.78	98.78	98.77	98.78
	Accuracy	83.113	85.304	81.782	87.772	87.346	88.04	88.132	88.243	88.308	87.845	87.707	88.114	88.187	88.058	88.548	88.345	86.681	**88.761**	88.539	88.659	87.873	80.941	88.668	88.548	88.724

A.4. Multi-task Learning with Fine-tuning

Code Documentation Generation - Python, small model, batch size: 256										
Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	6.298	**7.003**	6.505	6.851	6.361	6.364	6.911	6.763	6.639	6.417
ROUGE1	34.08	**34.73**	34.51	34.3	33.66	33.87	33.99	33.69	33.79	33.33
ROUGE2	12.99	**13.18**	13.15	13.03	12.72	12.8	12.76	12.42	12.56	12.28
ROUGELsum	32.05	**32.44**	32.39	32.13	31.52	31.82	31.7	31.33	31.57	31.22

Code Documentation Generation - Python, base model, batch size: 256										
Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	7.88	7.468	6.857	7.499	7.491	7.504	7.919	7.558	7.836	**7.988**
ROUGE1	**35.25**	35.23	34.62	33.77	34.05	33.96	33.86	33.09	33.35	33.1
ROUGE2	**13.83**	13.72	13.3	12.71	12.81	12.84	12.72	12.27	12.5	12.27
ROUGELsum	32.81	**32.95**	32.48	31.39	31.67	31.6	31.44	30.59	31.04	30.57

Code Documentation Generation - Python, large model, batch size: 256						
Steps	500	1000	1500	2000	2500	3000
BLEU	7.028	7.768	7.848	**8.219**	7.878	7.676
ROUGE1	**34.45**	33.96	33.84	33.66	33.66	33.45
ROUGE2	**13.07**	12.54	12.8	12.64	12.65	12.52
ROUGELsum	**32.1**	31.48	31.42	31.16	31.2	31.09

Code Documentation Generation - Java, small model, batch size: 256																				
Steps	2000	4000	8000	10000	14000	18000	22000	26000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000
BLEU	8.11	7.534	7.91	8.348	8.274	8.652	8.928	8.567	9.16	9.34	9.756	9.327	9.252	9.957	9.369	9.613	9.765	**10.108**	9.902	10.026
ROUGE1	**37.47**	35.94	36.28	36.52	36.27	36.4	36.39	35.33	35.19	35.77	34.82	35.28	34.68	34.92	34.72	35.57	35.12	35.14	35.22	34.77
ROUGE2	**17.61**	16.24	16.4	16.26	16.15	16.06	16.19	15.3	15.2	15.64	14.84	15.36	14.55	14.75	14.68	15.65	15.02	14.98	15.07	14.81
ROUGELsum	**35.37**	33.99	34.17	34.17	34.05	33.98	33.95	32.9	32.69	33.34	32.37	32.88	32.24	32.39	32.32	33.14	32.63	32.59	32.84	32.34

Code Documentation Generation - Java, base model, batch size: 256										
Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	11.936	11.959	11.502	11.125	11.778	11.66	**12.361**	11.834	12.072	11.792
ROUGE1	**39.59**	38.73	37.79	37.55	37.2	36.66	37.21	37.01	36.93	36.77
ROUGE2	**19.79**	18.73	17.6	17.39	17.09	16.69	17.02	16.79	16.83	16.57
ROUGELsum	**37.39**	36.26	35.2	35.06	34.66	34.11	34.55	34.43	34.26	34.03

Code Documentation Generation - Java, large model, batch size: 256						
Steps	500	1000	1500	2000	2500	3000
BLEU	13.973	**14.256**	14.141	14.14	13.88	13.728
ROUGE1	**39.68**	38.91	39.13	39.54	38.96	38.64
ROUGE2	**19.95**	19.07	19.32	19.81	18.93	18.67
ROUGELsum	**37.32**	36.53	36.6	37.04	36.4	36.26

Code Documentation Generation - Go, small model, batch size: 256																
Steps	2000	4000	6000	8000	10000	20000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000
BLEU	12.794	11.901	12.605	13.12	13.192	13.367	13.873	13.65	13.788	13.944	14.382	14.505	14.113	**14.967**	14.703	14.718
ROUGE1	**47.92**	46.93	47.19	47.18	47.21	46.61	46.13	45.89	45.61	45.58	45.63	45.27	45.52	45.1	45.47	45.58
ROUGE2	**25.08**	24.26	24.27	24.17	24.27	23.71	23.54	23.36	23.13	23.16	23.12	23.02	23.03	22.84	23.09	23.13
ROUGELsum	**45.86**	45.07	45	44.9	44.82	44.28	43.85	43.63	43.33	43.36	43.31	43.06	43.28	42.78	43.24	43.26

Code Documentation Generation - Go, base model, batch size: 256

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000	22000	24000	26000	28000	30000
BLEU	13.276	13.499	13.445	14.423	15.058	14.563	15.059	15.243	15.352	15.849	15.412	15.358	15.717	**15.949**	15.8
ROUGE1	**46.66**	45.77	45.76	45.59	45.81	45.45	44.89	45.59	45.45	45.8	45.29	45.07	45.31	45.53	45.88
ROUGE2	**24.57**	24.05	24.01	23.74	23.86	23.47	23.31	23.83	23.67	24.05	23.53	23.49	23.58	23.63	24.03
ROUGELsum	**44.86**	43.85	43.89	43.48	43.67	43.3	42.78	43.55	43.32	43.61	43.08	42.98	43.03	43.31	43.73

Code Documentation Generation - Go, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	13.648	13.897	13.825	15.001	15.459	16.103	15.506	16.096	**16.192**	15.895
ROUGE1	46.04	46.25	46.4	46.36	46.48	45.67	46.53	**47.05**	46.78	47.01
ROUGE2	23.66	23.58	24.14	24.06	23.84	23.34	24.1	**24.55**	24.36	24.45
ROUGELsum	43.94	44	44.29	44.06	44.13	43.3	44.31	**44.68**	44.51	44.65

Code Documentation Generation - Php, small model, batch size: 256

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	10.036	8.886	10.895	9.851	11.514	11.764	**12.547**	11.825	11.808	11.667
ROUGE1	**41.43**	40.55	41.04	41.39	40.79	41.47	41.1	41.35	40.79	40.78
ROUGE2	17.49	17.14	18.18	18.02	17.96	**18.24**	18.22	**18.24**	17.99	18.1
ROUGELsum	39.55	38.94	39.17	**39.65**	38.93	39.6	39.13	39.48	38.93	38.95

Code Documentation Generation - Php, base model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
BLEU	12.758	15.454	16.422	16.38	17.299	17.578	16.961	**17.995**	17.943	17.32
ROUGE1	42.15	**42.33**	41.86	41.79	41.84	41.94	41.75	41.97	42.06	41.78
ROUGE2	19.46	19.47	19.38	19.75	19.82	19.86	19.92	20.04	**20.18**	19.99
ROUGELsum	**40.27**	40.26	39.67	39.81	39.85	39.76	39.78	39.83	40.02	39.75

Code Documentation Generation - Php, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500	8000	9000	10000
BLEU	14.343	14.391	15.921	16.197	16.412	17.044	16.126	17.441	16.821	17.502	16.775	17.779	17.652	17.854	17.343	**18.325**	17.804	18.055
ROUGE1	42.32	42.77	42.49	42.83	41.78	42.6	42.67	42.72	41.96	42.56	42.44		42.77	42.67	42.6	**43.08**	42.1	42.58
ROUGE2	19.49	19.8	19.81	20.14	19.6	20.23	20.25	20.43	19.79	20.44	20.48	20.46	20.72	20.5	20.51	**20.96**	20.15	20.61
ROUGELsum	40.13	40.86	40.21	40.8	39.75	40.49	40.51	40.67	39.88	40.57	40.52	40.4	40.76	40.56	40.68	**41.08**	40.02	40.55

Code Documentation Generation - Ruby, small model, batch size: 256

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	3.794	3.82	3.949	4.521	4.277	4.46	**4.603**	4.552	4.111	4.284
ROUGE1	**29.39**	28.42	27.93	28.4	28.05	27.59	27.75	27.67	27.71	27.49
ROUGE2	**9.08**	8.7	8.53	**9.08**	8.66	8.75	8.95	8.53	8.65	8.58
ROUGELsum	**26.89**	26.07	25.43	25.81	25.39	25.42	25.36	25.19	25.2	25.1

Code Documentation Generation - Ruby, base model, batch size: 256

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	4.657	5.019	4.845	5.216	5.018	**5.352**	4.755	4.811	5.071	5.014
ROUGE1	**30.3**	30	29.64	29.92	29.67	29.74	29.35	29.36	29.85	29.45
ROUGE2	**10.29**	9.94	10.02	10.14	9.86	9.85	9.55	9.48	9.74	9.93
ROUGELsum	**27.82**	27.65	27.3	27.46	27.28	27.2	26.87	26.87	27.24	27.04

Code Documentation Generation - Ruby, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000
BLEU	5.46	5.159	5.509	5.389	**5.551**	5.359	5.548	5.425
ROUGE1	30.15	29.97	30.2	30.29	30.49	**30.56**	30.39	30.38
ROUGE2	10.59	10.58	10.75	**10.86**	10.62	10.4	10.49	10.45
ROUGELsum	27.79	27.58	27.78	27.79	**28.04**	**28.04**	27.91	27.85

Code Documentation Generation - Javascript, small model, batch size: 256

Steps	2000	4000	10000	12000	14000	16000	18000	20000	22000	24000	26000	28000	30000	32000	34000	36000
BLEU	4.248	4.669	7.861	8.644	8.795	9.089	9.252	9.365	9.46	9.401	9.569	9.663	9.423	**9.86**	9.483	9.476
ROUGE1	28.92	28.4	28.77	29.26	29.11	29.12	29.1	28.88	28.95	28.8	28.8	28.92	28.57	**29.27**	28.69	28.83
ROUGE2	9.86	9.81	11.14	11.56	11.55	11.42	11.84	11.8	11.72	11.65	11.62	11.94	11.47	**12.03**	11.65	11.69
ROUGELsum	27.17	26.57	26.86	**27.25**	27.02	26.96	27.02	26.92	26.84	26.84	26.75	26.92	26.58	**27.25**	26.72	26.89

Code Documentation Generation - Javascript, base model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
BLEU	10.448	10.679	10.61	10.983	11.01	10.612	10.828	**11.043**	10.901	10.998
ROUGE1	30.85	**31.19**	30.84	30.77	30.99	30.7	31.18	30.87	30.7	30.67
ROUGE2	13	13.07	13.1	13.04	13.28	12.96	13.18	**13.41**	13	13.07
ROUGELsum	28.77	**29.09**	28.73	28.57	28.86	28.75	29.03	28.74	28.64	28.68

Code Documentation Generation - Javascript, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000
BLEU	8.796	10.036	10.397	**10.969**	10.78	10.363
ROUGE1	30.98	31.83	31.76	31.43	**31.98**	30.93
ROUGE2	11.83	13.37	13.34	13.39	**13.72**	13.19
ROUGELsum	28.81	29.86	29.77	29.3	**30.02**	29

Source Code Summarization - Python, small model, batch size: 256

Steps	600	1200	1800	2400	3000	3600	4200	4800	5000
BLEU	**2.891**	1.741	1.906	2.015	1.809	1.863	1.596	1.943	1.836
ROUGE1	**23.94**	19.25	20.08	19.63	19.85	19.44	18.88	19.85	19.55
ROUGE2	**5.64**	4.13	3.98	3.99	3.91	3.75	3.58	3.98	3.86
ROUGELsum	**21.16**	17.27	17.73	17.34	17.31	17.08	16.61	17.41	17.12

Source Code Summarization - Python, base model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	1.94	2.11	**2.254**	2.034	2.171	1.855	1.968	1.577	1.831	1.694
ROUGE1	19.53	**21.07**	20.11	19.99	20.49	19.51	20.09	19.43	20.42	19.84
ROUGE2	4.02	4.29	4.31	4.2	**4.38**	3.9	4.24	3.65	4.18	3.99
ROUGELsum	17.53	**18.25**	17.82	17.66	18.04	17.16	17.64	16.98	17.91	17.43

Source Code Summarization - Python, large model, batch size: 256

Steps	100	200	300	400	500	600
BLEU	**2.848**	2.217	1.752	1.762	2.05	2.536
ROUGE1	**23.73**	21.43	19.8	20.05	20.91	20.75
ROUGE2	**5.94**	5.05	3.58	3.62	4.45	4.45
ROUGELsum	**20.95**	19.13	17.27	17.25	18.34	18.31

Source Code Summarization - SQL, small model, batch size: 256

Steps	600	1200	1800	2400	3000	3600	4200	4800	5000
BLEU	1.602	**1.877**	1.688	1.406	1.11	0.871	1.013	0.881	0.906
ROUGE1	18.2	**18.79**	17.92	16.64	15.25	14.52	14.96	14.39	14.3
ROUGE2	3.6	**3.96**	3.63	3.06	2.62	2.2	2.46	2.17	2.08
ROUGELsum	16.4	**16.96**	16	15.02	13.81	13.12	13.58	12.93	12.94

Source Code Summarization - SQL, base model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	**1.953**	1.382	0.921	1.105	0.942	0.952	0.938	0.941	0.896	1.016
ROUGE1	**19.42**	16.79	15.19	14.93	15.06	14.41	14.88	14.82	14.64	14.83
ROUGE2	**3.98**	3.1	2.22	2.14	2.14	2.08	2.22	2.03	2.03	2.14
ROUGELsum	**17.57**	15.36	13.82	13.54	13.53	12.98	13.36	13.27	13.13	13.26

Source Code Summarization - SQL, large model, batch size: 256

Steps	100	200	300	400	500	600
BLEU	**1.965**	1.693	1.443	1.103	0.947	0.819
ROUGE1	17.48	**18.77**	17.8	15.31	15.07	14.57
ROUGE2	**4.03**	3.67	3.4	2.5	2.35	1.95
ROUGELsum	16.26	**16.96**	15.82	13.77	13.53	13.07

Source Code Summarization - CSharp, small model, batch size: 256

Steps	600	1200	1800	2400	3000	3600	4200	4800	5000	5000
BLEU	3.406	**3.601**	3.472	3.328	3.123	3.051	2.579	2.677	2.675	1.016
ROUGE1	22.18	**22.67**	22.34	22.01	21.68	20.95	19.69	20.06	20.17	14.83
ROUGE2	5.49	**5.93**	5.68	5.5	5.35	4.96	4.45	4.74	4.63	2.14
ROUGELsum	20.47	**20.84**	20.5	20.22	19.92	19.13	18.12	18.41	18.37	13.26

Source Code Summarization - CSharp, base model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
BLEU	**4.199**	3.818	3.683	3.414	2.55	2.405	2.771	2.701	2.712	2.699
ROUGE1	**24.33**	23.18	22.25	21.77	19.25	18.65	20.69	20.14	20.02	20.15
ROUGE2	**6.56**	5.95	5.47	5.23	3.96	3.78	4.66	4.33	4.29	4.47
ROUGELsum	**22.18**	21.15	20.17	19.69	17.29	16.77	18.86	18.12	18.08	18.21

Source Code Summarization - CSharp, large model, batch size: 256

Steps	100	200	300	400	500	600
BLEU	**3.68**	3.544	3.287	3.035	2.602	3.482
ROUGE1	**22.84**	22.63	22	20.81	19.25	22.05
ROUGE2	**5.65**	5.5	5.71	4.72	4.22	5.5
ROUGELsum	**20.67**	20.62	20.21	18.79	17.49	20.06

Code Comment Generation, small model, batch size: 256

Steps	50000	100000	200000	300000	400000	500000	550000	600000	650000	700000	750000	800000	850000	900000
BLEU	33.415	39.303	45.094	47.435	50.037	51.048	51.27	51.339	51.518	51.448	**51.605**	51.471	51.532	51.574
ROUGE1	51.85	54.59	56.65	57.44	58.27	58.49	58.35	**58.53**	58.38	58.36	58.45	58.3	58.49	58.39
ROUGE2	39.12	42.68	45.42	46.64	47.42	47.68	47.59	47.76	47.66	47.7	47.75	47.63	47.75	**47.77**
ROUGELsum	50.76	53.53	55.53	56.42	57.18	57.36	57.28	**57.45**	57.29	57.32	57.36	57.21	57.41	57.32

Code Comment Generation, base model, batch size: 256

Steps	10000	30000	40000	50000	60000	70000	80000	90000	100000
BLEU	36.02	47.477	50.61	51.888	52.108	52.413	52.362	**52.653**	52.255
ROUGE1	52.95	57.75	59.03	59.54	**59.62**	59.51	59.58	59.43	59.58
ROUGE2	40.33	46.66	47.96	48.54	48.78	48.63	**48.79**	48.65	48.73
ROUGELsum	51.83	56.65	57.89	58.42	**58.49**	58.38	58.47	58.3	58.43

Code Comment Generation, large model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000
BLEU	50.474	52.189	52.653	52.642	**52.826**	52.781
ROUGE1	59.01	59.84	59.79	**60.11**	59.89	59.48
ROUGE2	47.77	48.9	48.92	**49.01**	49.09	48.9
ROUGELsum	57.92	58.75	58.68	**59.01**	58.76	58.38

Git Commit Message Generation, small model, batch size: 256

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000	22000	24000	26000	28000	30000
BLEU	43.262	44.47	44.432	44.59	44.505	44.426	44.478	44.591	44.411	**44.593**	44.165	4.487	44.507	44.304	44.301
ROUGE1	47.52	47.73	47.68	**48.12**	47.71	47.51	47.63	47.76	47.83	47.83	47.18	47.8	47.72	47.59	47.68
ROUGE2	34.83	35.37	35.67	35.65	35.61	35.35	35.68	35.6	35.57	**35.71**	35.21	35.66	**35.71**	35.44	35.56
ROUGELsum	47.08	47.06	47.15	**47.56**	47.12	46.95	47.07	47.32	47.32	47.27	46.74	47.29	47.31	47.09	47.25

Git Commit Message Generation, base model, batch size: 256

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	44.584	**44.851**	44.777	44.798	44.507	44.673	44.539	44.813	44.448	44.747
ROUGE1	48.61	48.34	48.41	48.54	48.11	48.28	48.31	**48.75**	48.49	48.59
ROUGE2	35.89	35.45	35.76	**36.01**	35.77	35.53	35.33	35.81	35.54	35.9
ROUGELsum	48.06	47.75	47.91	48.03	47.55	47.83	47.73	**48.13**	47.99	48.05

Git Commit Message Generation, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000	3500	4000
BLEU	44.525	**44.864**	44.793	44.433	44.61	44.858	44.779	44.783
ROUGE1	48.62	48.95	48.84	48.83	48.84	**49.06**	48.6	48.89
ROUGE2	35.48	35.88	35.85	36.22	36.33	36.35	35.98	**36.36**
ROUGELsum	47.94	48.34	48.11	48.2	48.2	**48.53**	48.05	48.31

API Sequence Generation, small model, batch size: 256

Steps	50000	100000	200000	300000	400000	500000	600000	700000	800000	900000	950000	1000000	1050000	1100000	1150000	1200000	1250000
BLEU	63.276	65.28	67.277	68.616	69.079	69.566	69.738	70.125	70.544	70.424	70.921	70.766	71.045	70.876	**71.312**	71.09	71.23
ROUGE1	70.66	72.59	74.55	75.49	76.18	76.62	76.77	77.13	77.38	77.31	77.68	77.63	77.87	77.81	**78.05**	77.95	77.91
ROUGE2	60.81	63.02	65.06	66.35	66.95	67.59	67.72	68.22	68.45	68.35	68.7	68.55	69.02	69	**69.24**	69.12	69.16
ROUGELsum	70.63	72.58	74.57	75.48	76.18	76.6	76.78	77.11	77.33	77.31	77.66	77.61	77.9	77.79	**78.01**	77.95	77.89

API Sequence Generation, base model, batch size: 256

Steps	20000	40000	100000	120000	140000	160000	180000	200000	220000	240000	260000	280000	300000	310000	320000	330000	340000	350000
BLEU	69.933	70.643	72.153	72.245	72.412	73.036	73.351	73.563	73.457	73.476	73.51	74.057	74.239	**74.256**	74.156	74.003	74.065	74.237
ROUGE1	76.92	77.7	79.01	79.24	79.59	79.93	80.01	80.43	80.67	80.68	80.84	80.99	81.3	81.31	81.32	81.32	81.39	**81.41**
ROUGE2	67.89	68.76	70.43	70.72	71.02	71.49	71.62	72.12	72.32	72.5	72.6	72.79	73.1	73.09	**73.2**	73.17	73.19	73.13
ROUGELsum	76.92	77.69	78.98	79.22	79.55	79.95	79.99	80.43	80.65	80.69	80.82	81	81.26	81.294	81.33	81.29	**81.4**	**81.4**

API Sequence Generation, large model, batch size: 256

Steps	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000	110000	120000	130000	140000
BLEU	72.371	73.372	74.109	73.643	74.036	74.268	74.164	74.324	74.354	**74.628**	74.567	74.543	74.528	74.231
ROUGE1	79.6	80.45	80.83	80.74	81.18	81.36	81.41	81.45	81.45	81.73	81.67	81.75	**81.8**	81.61
ROUGE2	71.15	72.08	72.67	72.46	73	73.3	73.34	73.4	73.45	73.74	73.63	73.71	**73.81**	73.41
ROUGELsum	79.6	80.43	80.81	80.77	81.18	81.33	81.39	81.46	81.46	81.72	81.64	81.72	**81.78**	81.6

Program Synthesis, small model, batch size: 256

Steps	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
BLEU	94.599	94.628	94.622	94.625	94.627	94.638	94.651	**94.662**	94.652	94.641
ROUGE1	99.17	99.16	99.2	99.2	99.2	99.2	**99.21**	99.2	**99.21**	99.2
ROUGE2	98.82	98.8	98.85	98.82	98.85	98.86	**98.88**	**98.88**	98.87	98.86
ROUGELsum	99.01	99	99.04	99.02	99.03	99.04	**99.05**	99.04	99.04	**99.05**
Accuracy	91.811	92.245	92.402	92.273	92.338	92.439	**92.458**	92.439	92.402	92.439

Program Synthesis, base model, batch size: 256

Steps	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
BLEU	**94.671**	94.634	94.635	94.66	94.646	94.669	94.65	94.637	94.652	94.647
ROUGE1	99.21	99.2	99.21	99.21	99.2	99.21	99.21	99.2	99.21	99.2
ROUGE2	98.91	98.86	98.88	98.21	98.89	**98.92**	98.89	98.86	98.89	98.88
ROUGELsum	99.05	99.04	99.05	**99.06**	99.05	99.06	99.05	99.03	99.05	99.05
Accuracy	92.541	92.43	92.448	92.513	92.541	**92.606**	92.513	92.43	92.55	92.495

Program Synthesis, large model, batch size: 256

Steps	500	1000	1500	2000	2500	3000
BLEU	94.596	**94.646**	94.639	94.641	94.628	94.615
ROUGE1	99.13	99.2	**99.21**	99.2	99.2	99.2
ROUGE2	98.77	98.84	**98.86**	98.85	98.84	98.84
ROUGELsum	98.97	99.02	99.04	**99.05**	99.03	99.04
Accuracy	92.19	92.393	92.439	**92.467**	92.393	92.402

B. Appendix - Model Output Examples for each Task

B.1. Code Documentation Generation

B.1.1. Code Documentation Generation - Python

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Prints an error and exits with an optional exit code .
	Base	Prints the e .
CodeTrans Transfer Learning	Small	Prints a message and exits with the specified exit code .
	Base	Prints an error message and exits .
	Large	Print an error message and exit .
CodeTrans Multi-task Learning	Small	Prints a message and exits if exit_code is not None .
	Base	Prints an error message and exits with the given exit code .
	Large	Print an error message and exit .
CodeTrans Multi-task Learning Fine-tuning	Small	Print a message and exit .
	Base	Prints a message to the log and exits with an optional exit code .
	Large	Print error message and exit .
Code Snippet as Input		def e (message , exit_code = None) : print_log (message , YELLOW , BOLD) if exit_code is not None : sys . exit (exit_code)
Golden Reference		Print an error log message .

B.1.2. Code Documentation Generation - Java

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Casts a Cast to another type
	Base	Casts a value into a function that returns the same type as the supplied parameter .
CodeTrans Transfer Learning	Small	Casts the given class to a function .
	Base	Casts a function to a class
	Large	Cast a value to the given target type .
CodeTrans Multi-task Learning	Small	Casts the given class to the given target class .
	Base	Returns a function that casts the given value to the given type .
	Large	Creates a function that performs a cast to the given class .
CodeTrans Multi-task Learning Fine-tuning	Small	Creates a function that casts the given object to the target type .
	Base	Returns a function that casts the given object to the given class .
	Large	Returns a function that casts its input type to the target type .
Code Snippet as Input		public static < T , U > Function < T , U > castFunction (Class < U > target) { return new CastToClass < T , U > (target) ; }
Golden Reference		Returns a function that cast the incoming values via a Class object .

B.1.3. Code Documentation Generation - Go

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	needSnapshotAbort returns true if we need to roll a snapshot .
	Base	needSnapshotAbort returns true if we need to call snapshot and false otherwise .
CodeTrans Transfer Learning	Small	needSnapshotAbort returns true if the Progress . Match field of ProgressState == ProgressStateSnapshot and pr . Match < pr . PendingSnapshot .
	Base	needSnapshotAbort returns true if the progress is in a snapshot and the match exceeds the pending snapshot .
	Large	needsnapshotabort returns true if we need to abort a snapshot .
CodeTrans Multi-task Learning	Small	needSnapshotAbort returns true if the current state is a snapshot .
	Base	needSnapshotAbort returns true if we need to abort the snapshot .
	Large	needsnapshotabort returns true if we need to abort a snapshot .
CodeTrans Multi-task Learning Fine-tuning	Small	needSnapshotAbort returns true if the progress needs to be aborted .
	Base	needSnapshotAbort returns true if we need to abort the snapshot
	Large	needsnapshotabort returns true if we need to abort the snapshot .
Code Snippet as Input		func (pr * Progress) needSnapshotAbort ( ) bool { return pr . State == ProgressStateSnapshot && pr . Match >= pr . PendingSnapshot }
Golden Reference		needSnapshotAbort returns true if snapshot progress s Match is equal or higher than the pendingSnapshot .

B.1.4. Code Documentation Generation - Php

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Update table with specific data
	Base	Call when table is constructed
CodeTrans Transfer Learning	Small	Updates a table .
	Base	Calls the updateTable method to update a table .
	Large	Updates a table .
CodeTrans Multi-task Learning	Small	Update a table .
	Base	Updates a table .
	Large	updates a database table .
CodeTrans Multi-task Learning Fine-tuning	Small	Update a table .
	Base	Updates a table .
	Large	updates a database table .
Code Snippet as Input		public static function update ($ table) { if (! is_array ($ table)) { $ table = json_decode ($ table , true) ; } if (! SchemaManager :: tableExists ($ table ['oldName'])) { throw SchemaException :: tableDoesNotExist ($ table ['oldName']) ; } $ updater = new self ($ table) ; $ updater -> updateTable ( ) ; } }
Golden Reference		Update the table .

B.1.5. Code Documentation Generation - Ruby

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Log a message at the given level if the logger is present
	Base	Writes a log message if the current log level is at or below the supplied severity .
CodeTrans Transfer Learning	Small	Create an IO object at the given level with the given block . This method will not be called directly if the IO object at that level .
	Base	Writes a message of the given severity to the log .
	Large	Writes a message to the log device .
CodeTrans Multi-task Learning	Small	Add a message to the log .
	Base	Add a message to the logger .
	Large	Add a message to the log .
CodeTrans Multi-task Learning Fine-tuning	Small	Writes a message to the log with a given severity .
	Base	We need to overload the add method . Basibally it is the same as the original one but we add our own log format to it .
	Large	Writes a message if the severity is high enough . This method is executed asynchronously .
Code Snippet as Input		def add (severity , progname , & block) return true if io . nil? || severity < level message = format_message (severity , progname , yield) MUTEX . synchronize { io . write (message) } true end
Golden Reference		Add a message to the logger with the given severity and progname .

B.1.6. Code Documentation Generation - Javascript

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Returns true if the browser is a native element .
	Base	Returns whether the givenEnv should be focused .
CodeTrans Transfer Learning	Small	Checks if the current browser is on a standard browser environment .
	Base	Check if browser environment is a standard browser environment
	Large	Check if the environment is standard browser .
CodeTrans Multi-task Learning	Small	Returns true if the browser environment is a standard browser environment .
	Base	Checks if the current browser environment is a standard browser environment .
	Large	Determines if the current environment is a standard browser environment
CodeTrans Multi-task Learning Fine-tuning	Small	Standard browser environment has a notion of what React Native does not support it .
	Base	Check if the browserEnv is standard .
	Large	Checks if the browser is in a standard environment .
Code Snippet as Input		function isStandardBrowserEnv ( ) { if (typeof navigator !== ’undefined’ && (navigator . product === ’ReactNative’ || navigator . product === ’NativeScript’ || navigator . product === ’NS’)) { return false ; } return (typeof window !== ’undefined’ && typeof document !== ’undefined’) ; }
Golden Reference		Determine if we re running in a standard browser environment

B.2. Source Code Summarization

B.2.1. Source Code Summarization - Python

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Python : Compare files containing a folder
	Base	How to copy a text file into a variable ?
CodeTrans Transfer Learning	Small	includes within a line ’ with python 2.7
	Base	Python : Include specific lines in a file
	Large	Changing a line in a text file
CodeTrans Multi-task Learning	Small	How to write a string to a file in Python ?
	Base	How to write a text file to a text file in Python ?
	Large	How to remove a line from a file in python ?
CodeTrans Multi-task Learning Fine-tuning	Small	How to include text in a file using Python ?
	Base	Python : ’Include this text in a file ’
	Large	How to remove empty string in python
Code Snippet as Input		with open (CODE_STRING , CODE_STRING) as in_file : buf = in_file . readlines ( ) with open (CODE_STRING , CODE_STRING) as out_file : for line in buf : if line == " ; Include this text " : line = line + " Include below " out_file . write (line) ’
Golden Reference		Adding lines after specific line

B.2.2. Source Code Summarization - CSharp

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	How to convert C # DateTime to C #
	Base	How to convert DateTime to DateTime format in C # ?
CodeTrans Transfer Learning	Small	How to convert double to DateTime in C # ?
	Base	Convert double to DateTime
	Large	How to convert double to datetime ?
CodeTrans Multi-task Learning	Small	How to convert unix timestamp to local time in C # ?
	Base	How to convert a double to a DateTime in C # ?
	Large	How to convert double to datetime in C # ?
CodeTrans Multi-task Learning Fine-tuning	Small	How to convert double to DateTime in C # ?
	Base	How to convert double to DateTime in C # ?
	Large	Convert double to datetime in C #
Code Snippet as Input		public static DateTime ParseUnixDateTime (double unixTime) { var dt = new DateTime (CODE_INTEGER CODE_INTEGER , CODE_INTEGER , CODE_INTEGER , CODE_INTEGER , CODE_INTEGER , CODE_INTEGER , System . DateTimeKind . Utc) ; dt = dt . AddSeconds (unixTimeStamp) . ToLocalTime ( ) ; return dt ; }
Golden Reference		C # Display compare timestamp to current month

B.2.3. Source Code Summarization - SQL

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	mysql : how to get the difference of a column in a table ?
	Base	how do i get the average of a date range in sql server 2005 ?
CodeTrans Transfer Learning	Small	how to get the time in milliseconds since the start time of the transaction was taken ?
	Base	how to get current date time in sql server ?
	Large	mysql time ( ) function
CodeTrans Multi-task Learning	Small	how to get the time in mysql ?
	Base	how can i get the time of a date in mysql ?
	Large	how to convert datetime to time in mysql ?
CodeTrans Multi-task Learning Fine-tuning	Small	how to get the correct time from mysql database ?
	Base	how to convert date to time in mysql ?
	Large	select time from mysql table
Code Snippet as Input		select time (col0) from tab0
Golden Reference		datetime implementation in php mysql

B.3. Code Comment Generation

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Renders the URI into its textual representation .
	Base	Renders the URI .
CodeTrans Transfer Learning	Small	Renders the given URI into a string . Override this if you want to do something special to convert the URI into ASCII .
	Base	Render a URI
	Large	Renders the given URI using the encoding scheme in ISO-8859-1 .
CodeTrans Multi-task Learning	Small	Renders a URI .
	Base	Renders the given URI into a string .
	Large	Renders the URI as an ASCII string .
CodeTrans Multi-task Learning Fine-tuning	Small	Render the URI as ASCII string .
	Base	Renders a URI as a string , escaping special characters .
	Large	Render the given URI .
Code Snippet as Input		protected String renderUri (URI uri) return uri . toASCIIString ( ) ;
Golden Reference		Render the URI as a string

B.4. Git Commit Message Generation

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Added missing Jungtaeks to log4j . jar
	Base	Added missing jar file
CodeTrans Transfer Learning	Small	Adding missing joscar jar
	Base	Added joscar . jar to the lib directory of the Gateway project
	Large	added missing joscar jar
CodeTrans Multi-task Learning	Small	add joscar jar
	Base	Add joscar jar
	Large	Add joscar
CodeTrans Multi-task Learning Fine-tuning	Small	Adding missing jar file
	Base	Added joscar jar
	Large	Started work on the icq / aim protocol provider implementation
Code Snippet as Input		new file mode 100644 index 000000000 . . 892fda21b Binary files / dev / null and b / src / plugins / gateway / lib / joscar . jar differ
Golden Reference		Added joscar JAR

B.5. API Sequence Generation

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	Color . getRed Color . getGreen Color . getBlue
	Base	Color . getRed Color . getGreen Color . getBlue
CodeTrans Transfer Learning	Small	Color . getRed Color . getGreen Color . getBlue
	Base	Math . max Math . min
	Large	Math . max Math . min
CodeTrans Multi-task Learning	Small	Math . min Math . max
	Base	Color . getRed Color . getGreen Color . getBlue
	Large	Renders Math . round Math . max Math . round Math . max Math . round
CodeTrans Multi-task Learning Fine-tuning	Small	Color . getRed Color . getGreen Color . getBlue
	Base	Color . getRed Color . getGreen Color . getBlue
	Large	Color . getRed Color . getGreen Color . getBlue
Code Snippet as Input		Convert from normal rgb to java hsb
Golden Reference		Color.RGBtoHSB Color.getHSBColor

B.6. Program Synthesis

Model	Size	Model Output
CodeTrans Single-Task Learning	Small	[map a [partial1 b -]]
	Base	[map a [partial1 b -]]
CodeTrans Transfer Learning	Small	[map a [partial1 b -]]
	Base	[map a [partial1 b -]]
	Large	[map a [partial1 b -]]
CodeTrans Multi-task Learning	Small	[map a [partial1 b -]]
	Base	[map a [partial1 b -]]
	Large	[map a [partial1 b -]]
CodeTrans Multi-task Learning Fine-tuning	Small	[map a [partial1 b -]]
	Base	[map a [partial1 b -]]
	Large	[map a [partial1 b -]]
Code Snippet as Input		you are given an array of numbers a and a number b , compute the difference of elements in a and b
Golden Reference		[map a [partial1 b -]]

List of Figures

2.1. The architecture of a single-layer feed-forward neural network. 7
2.2. The word2vec models architecture. 8
2.3. The architecture and details of model RNN and LSTM. 9
2.4. An example of the Sequence to Sequence model. 9
2.5. The transformer architecture. 10
2.6. The computational steps of the multi-head self-attention layer. 11
2.7. Bert model with one additional output layer for fine-tuning tasks. 13
2.8. Two different multitask learning methods for deep learning. 14
2.9. The architecture of Multi-Task Deep Neural Network (MT-DNN) model. . . . 15
2.10. The numbers of publications about deep learning in software engineering and development domain per year. 17
2.11. Software engineering tasks are categorized into six software engineering steps. 17
2.12. Data type used in software engineering tasks. 18
2.13. Deep learning architectures used by software engineering tasks. 19

3.1. The evaluation result of CodeBERT for Code Documentation Generation tasks. 23
3.2. CODE-NN Architecture and the evaluation result. 23
3.3. DeepCom Architecture and the evaluation result. 25
3.4. DeepCom SBT method example. 26
3.5. NMT models evaluation results by metrics and one human evaluation example. 27
3.6. An example for extracting an API sequence and its query from a Java method. 28
3.7. The architecture of Seq2Tree encoder-decorder model. 29
3.8. Different transformer architectures. 30
3.9. The different denoising objective examples. 31
3.10. Experiment steps on unsupervised objectives and the selected decisions. . . . 31

4.1. The Boxplot of the sample length in unsupervised data corpus 38
4.2. A Python example with code and documentation from CodeSearchNet Corpus Collection. 39
4.3. A SQL example with the original code snippet, the code snippet after preprocessing and the three related summarizations. 40
4.4. A Java example for Code Comment Generation task. 41
4.5. An example for the Git Commit Message Generation task. 41
4.6. An example for the API Sequence Recommendation task. 42
4.7. An example for the Program Synthesis task. 43

4.8. Input and output sample length in Boxplot for supervised datasets. 45
4.9. The transformer architecture. 47

5.1. The Tensorboard of the T5 small model during the transfer learning pre-training. 56
5.2. The Tensorboard of the T5 base model during the transfer learning pre-training. 57
5.3. The Tensorboard of the T5 large model during the transfer learning pre-training. 58
5.4. The Tensorboard of the T5 small model during the multi-task learning. 61
5.5. The Tensorboard of the T5 base model during the multi-task learning. 61
5.6. The Tensorboard of the T5 large model during the multi-task learning. 62

6.1. The evaluation of multi-task learning checkpoints on the validation set for two tasks. 75
6.2. The multi-task learning fine-tuning base model page for the task Code Documentation Generation - Java from the Hugging Face Model Hub. 77

List of Tables

4.1. Statistics of the unsupervised datasets 37
4.2. Unsupervised programming language datasets attributes 37
4.3. The summarization of supervised datasets. 44
4.4. Important hyperparameters for the architecture of T5 models 48

5.1. Specifications of two kinds of Google Cloud TPUs we used in this thesis. . . . 52
5.2. The single-task learning experiment setups for each task 55
5.3. The transfer learning fine-tuning experiment setups for each task. 59
5.4. Evalutation on the validation set of the Source Code Summarization SQL task for fine-tuning 1,000 steps using the interval of 100. 60
5.5. The best multi-task learning checkpoints (training steps) for different tasks in the small and base model. 62
5.6. The multi-task learning fine-tuning experiment setups for each task. 64

6.1. The evaluation results for the task Code Documentation Generation - Python. 66
6.2. The evaluation results for the task Code Documentation Generation - Java. . . 67
6.3. The evaluation results for the task Code Documentation Generation - Go. . . . 67
6.4. The evaluation results for the task Code Documentation Generation - Php. . . 68
6.5. The evaluation results for the task Code Documentation Generation - Ruby. . . 68
6.6. The evaluation results for the task Code Documentation Generation - Javascript. 69
6.7. The evaluation results for the task Source Code Summarization - Python. . . . 69
6.8. The evaluation results for the task Source Code Summarization - CSharp. . . . 70
6.9. The evaluation results for the task Source Code Summarization - SQL. 70
6.10. The evaluation results for the task Code Comment Generation. 71
6.11. The evaluation results for the task Git Commit Message Generation. 72
6.12. The evaluation results for the task API Sequence Recommendation. 73
6.13. The evaluation results for the task Program Synthesis. 73
6.14. The models' output for an example of the task Code Documentation Generation - Javascript. 74
6.15. The models' output for an example of the task Source Code Summarization. . 76
6.16. The evaluation of the task Code Documentation Generation - Java on the validation set when fine-tuning the multi-task base model. 76

Bibliography

[1] W. W. Royce. "Managing the development of large software systems: concepts and techniques". In: *Proceedings of the 9th international conference on Software Engineering*. 1987, pp. 328–338.

[2] R. Raina, A. Madhavan, and A. Y. Ng. "Large-scale deep unsupervised learning using graphics processors". In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 873–880.

[3] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[4] S. SEVERINI. "Multi-task Deep Learning in the Software Development domain". In: (2019).

[5] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. URL: `http://jmlr.org/papers/v21/20-074.html`.

[6] J. W. Tukey. "The teaching of concrete mathematics". In: *The American Mathematical Monthly* 65.1 (1958), pp. 1–9.

[7] P. Bourque, R. E. Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[8] E. D. Liddy. "Natural language processing". In: (2001).

[9] K. S. Jones. "A statistical interpretation of term specificity and its application in retrieval". In: *Journal of documentation* (1972).

[10] L. Deng and Y. Liu. *Deep learning in natural language processing*. Springer, 2018.

[11] R. Quiza and J. Davim. "Computational Methods and Optimization". In: Jan. 2011, pp. 177–208. ISBN: 978-1-84996-449-4. DOI: `10.1007/978-1-84996-450-0`.

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[13] J. Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural networks* 61 (2015), pp. 85–117.

[14] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*. 2013, pp. 3111–3119.

[15] T. Mikolov, W.-t. Yih, and G. Zweig. "Linguistic regularities in continuous space word representations". In: *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*. 2013, pp. 746–751.

[16] T. Mikolov, S. Kombrink, L. Burget, J. Černock, and S. Khudanpur. "Extensions of recurrent neural network language model". In: *2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2011, pp. 5528–5531.

[17] S. Hochreiter and J. Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[18] I. Sutskever, O. Vinyals, and Q. V. Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[19] M. Sundermeyer, R. Schlüter, and H. Ney. "LSTM neural networks for language modeling". In: *Thirteenth annual conference of the international speech communication association*. 2012.

[20] S. Kombrink, T. Mikolov, M. Karafiát, and L. Burget. "Recurrent neural network based language modeling in meeting recognition". In: *Twelfth annual conference of the international speech communication association*. 2011.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[22] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. "How transferable are features in deep neural networks?" In: *Advances in neural information processing systems*. 2014, pp. 3320–3328.

[23] S. J. Pan and Q. Yang. "A survey on transfer learning". In: *IEEE Transactions on knowledge and data engineering* 22.10 (2009), pp. 1345–1359.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[25] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler. "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 19–27.

[26] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. "Albert: A lite bert for self-supervised learning of language representations". In: *arXiv preprint arXiv:1909.11942* (2019).

[27] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. "Roberta: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[28] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[29] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le. "Xlnet: Generalized autoregressive pretraining for language understanding". In: *Advances in neural information processing systems*. 2019, pp. 5753–5763.

[30] R. Caruana. “Multitask learning”. In: *Machine learning* 28.1 (1997), pp. 41–75.

[31] S. Ruder. “An overview of multi-task learning in deep neural networks”. In: *arXiv preprint arXiv:1706.05098* (2017).

[32] X. Liu, P. He, W. Chen, and J. Gao. “Multi-task deep neural networks for natural language understanding”. In: *arXiv preprint arXiv:1901.11504* (2019).

[33] R. Mihalcea, H. Liu, and H. Lieberman. “NLP (natural language processing) for NLP (natural language programming)”. In: *International Conference on Intelligent Text Processing and Computational Linguistics*. Springer. 2006, pp. 319–330.

[34] M. Stenmark and P. Nugues. “Natural language programming of industrial robots”. In: *IEEE ISR 2013*. IEEE. 2013, pp. 1–5.

[35] X. Li, H. Jiang, Z. Ren, G. Li, and J. Zhang. “Deep learning in software engineering”. In: *arXiv preprint arXiv:1805.04825* (2018).

[36] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[37] C. A. Watson. “Deep Learning in Software Engineering”. PhD thesis. The College of William and Mary, 2020.

[38] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample. “Unsupervised Translation of Programming Languages”. In: *arXiv preprint arXiv:2006.03511* (2020).

[39] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. “Codebert: A pre-trained model for programming and natural languages”. In: *arXiv preprint arXiv:2002.08155* (2020).

[40] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. “Codesearchnet challenge: Evaluating the state of semantic code search”. In: *arXiv preprint arXiv:1909.09436* (2019).

[41] C.-Y. Lin and F. J. Och. “Orange: a method for evaluating automatic evaluation metrics for machine translation”. In: *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*. 2004, pp. 501–507.

[42] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. “Summarizing source code using a neural attention model”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 2073–2083.

[43] M.-T. Luong, H. Pham, and C. D. Manning. “Effective approaches to attention-based neural machine translation”. In: *arXiv preprint arXiv:1508.04025* (2015).

[44] S. Banerjee and A. Lavie. “METEOR: An automatic metric for MT evaluation with improved correlation with human judgments”. In: *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 2005, pp. 65–72.

[45] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, et al. “Moses: Open source toolkit for statistical machine translation”. In: *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*. Association for Computational Linguistics. 2007, pp. 177–180.

[46] A. M. Rush, S. Chopra, and J. Weston. “A neural attention model for abstractive sentence summarization”. In: *arXiv preprint arXiv:1509.00685* (2015).

[47] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. “Deep code comment generation”. In: *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE. 2018, pp. 200–20010.

[48] D. Bahdanau, K. Cho, and Y. Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[49] S. Jiang, A. Armaly, and C. McMillan. “Automatically generating commit messages from diffs using neural machine translation”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 135–146.

[50] R. Sennrich, O. Firat, K. Cho, A. Birch, B. Haddow, J. Hitschler, M. Junczys-Dowmunt, S. Läubli, A. V. M. Barone, J. Mokry, et al. “Nematus: a toolkit for neural machine translation”. In: *arXiv preprint arXiv:1703.04357* (2017).

[51] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. “BLEU: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[52] X. Gu, H. Zhang, D. Zhang, and S. Kim. “Deep API learning”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 631–642.

[53] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[54] M. D. Zeiler. “Adadelta: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012).

[55] P. Koehn. “Pharaoh: a beam search decoder for phrase-based statistical machine translation models”. In: *Conference of the Association for Machine Translation in the Americas*. Springer. 2004, pp. 115–124.

[56] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. “Mining succinct and high-coverage API usage patterns from source code”. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 319–328.

[57] M. Raghothaman, Y. Wei, and Y. Hamadi. “Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 357–367.

[58] D. Alvarez-Melis and T. S. Jaakkola. “Tree-structured decoding with doubly-recurrent neural networks”. In: (2016).

[59] I. Polosukhin and A. Skidanov. “Neural program search: Solving programming tasks from description and examples”. In: *arXiv preprint arXiv:1802.04335* (2018).

[60] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. “Robustfill: Neural program learning under noisy i/o”. In: *arXiv preprint arXiv:1703.07469* (2017).

[61] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. *Improving language understanding by generative pre-training*. 2018.

[62] P. J. Liu, M. Saleh, E. Pot, B. Goodrich, R. Sepassi, L. Kaiser, and N. Shazeer. "Generating wikipedia by summarizing long sequences". In: *arXiv preprint arXiv:1801.10198* (2018).

[63] V. Markovtsev and W. Long. "Public Git archive: A big code dataset for all". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 34–37.

[64] V. Raychev, P. Bielik, and M. Vechev. "Probabilistic model for code with decision trees". In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 731–747.

[65] Z. Yao, D. S. Weld, W.-P. Chen, and H. Sun. "StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow". In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2018, pp. 1693–1703.

[66] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. "One billion word benchmark for measuring progress in statistical language modeling". In: *arXiv preprint arXiv:1312.3005* (2013).

[67] T. Kudo. "Subword regularization: Improving neural network translation models with multiple subword candidates". In: *arXiv preprint arXiv:1804.10959* (2018).

[68] M. Post. "A call for clarity in reporting BLEU scores". In: *arXiv preprint arXiv:1804.08771* (2018).

[69] C.-Y. Lin. "Rouge: A package for automatic evaluation of summaries". In: *Text summarization branches out*. 2004, pp. 74–81.

[70] Y. E. Wang, G.-Y. Wei, and D. Brooks. "Benchmarking tpu, gpu, and cpu platforms for deep learning". In: *arXiv preprint arXiv:1907.10701* (2019).