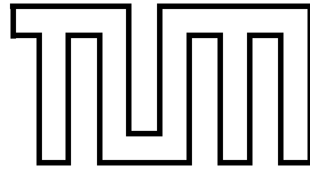


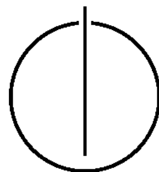
FAKULTÄT FÜR INFORMATIK

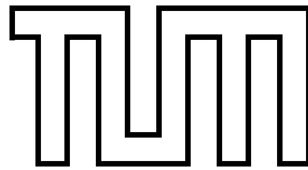
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Multi-task Deep Learning in the Software Development Domain

Silvia Severini





FAKULTÄT FÜR INFORMATIK

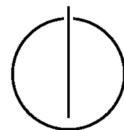
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Multi-task Deep Learning in the Software Development Domain

Multi-Task Deep Learning im Bereich Softwareentwicklung

Author:	Silvia Severini
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Ahmed Elnaggar
External Advisor:	Davide Bacciu
Date:	October 15, 2019



Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den September 22, 2019 Silvia Severini

Acknowledgments

I would like to thank professor Florian Matthes for the opportunity to work at the project in the SEBIS chair.
I would like to thank my advisors, Ahmed Elnaggar and Davide Bacciu, for the supervision offered during all the project development and for the precious feedbacks they gave me.
I want to thank my brother Lorenzo and my parents, Roberto and Antonella, for always supporting me during all the five years of academic studies. Moreover, I want to thank Diego for always being by my side.
Finally, I would like to thank my colleagues and housemates that contributed to improving my university life.

Abstract

Nowadays, almost every aspect of life depends on reliable high-quality software, but its creation is costly and it is hard work for engineers. Indeed, there is a high demand for software tools that could help this development process. One direction of improvement consists on the use of Deep Learning techniques to create them. However, we might face problems related to limited labeled datasets available, model overfitting that prevents the effective generalization, and energy consumption for the training process.
In this thesis, we investigate how Multi-task Deep Learning can tackle these issues in the software development domain applying it to tasks that involve the manipulation of English, a natural language, and four programming languages namely Python, SQL, C#, and Java. We review the software development domain, the techniques used in Natural Language Processing, and the domain adaptation in Deep Learning. Then, we adapt the Transformer model architecture, actual state-of-the-art for sequence-to-sequence manipulation problems, to seven supervised tasks and to a self-supervised language model and we explore whether we get benefits on the training of single tasks compared to multiple tasks together. We show the performance of our models and we compare our results with state-of-the-art counterparts that solved the tasks with the same datasets. We conclude that, given enough computing resources, Multi-task Deep Learning with the Transformer architecture is a promising framework to deal with software development domain tasks. To the best of our knowledge, this is the first work that applies large-scale multi-task models to software development tasks that involve source code and self-supervised and supervised tasks. Moreover, we create two source code corpora, one for C# and one for Java, and we train a model that can be fine-tuned to help the transfer of knowledge for further research.

Zusammenfassung

Heutzutage hängt fast jeder Aspekt des Lebens von zuverlässiger, hochwertiger Software ab. Ihre Erstellung ist aber kostspielig und eine harte Arbeit für Ingenieure.
Tatsächlich besteht ein hoher Bedarf an Softwaretools, die diesen Entwicklungsprozess unterstützen könnten. Die Anwendung von Deep Learning Techniken kann in diesem Fall angewendet werden, um sie zu erstellen. Jedoch könnten wir Probleme haben, wie zum Beispiel mit den verfügbaren, begrenzt beschrifteten Datensätzen, mit dem Model Overfitting, das eine effektive Generalisierung verhindert, und mit dem Energieverbrauch für den Trainingsprozess. In dieser Arbeit wird es untersucht, wie Multi-Task Deep Learning diese Probleme in der Softwareentwicklung angehen kann, indem wir es auf Aufgaben anwenden, die die Manipulation von Englisch, einer natürlichen Sprache und vier Programmiersprachen (Python, SQL, C# und Java) beinhalten. Wir überprüfen den Softwareentwicklungsbereich, die Techniken, die in der Natural Language Processing verwendet werden, und die Domain Adaptation im Deep Learning. Dann passen wir die Transformer-Modellarchitektur, die der aktuelle Stand der Technik bei Sequence-to-Sequence-Manipulationsproblemen ist, an sieben supervised Tasks und an das semi-supervised Language Model an. Dann untersuchen wir, ob wir Vorteile beim Training von Einzelaufgaben im Vergleich zu mehreren Aufgaben bekommen.
Wir analysieren die Performance unserer Modelle und vergleichen die dazu gehörigen Ergebnisse mit den modernsten Counterparts, die die Tasks mit den gleichen Datensätzen gelöst haben. Am Ende können wir die Schlussfolgerung ziehen, dass Multi-Task Deep Learning mit der Transformer-Architektur bei ausreichenden Computerressourcen ein vielversprechender Rahmen für die Bewältigung von Aufgaben in der Softwareentwicklung ist. Nach bestem Wissen ist diese die erste Arbeit, die groß angelegte Multitasking-Modelle auf Softwareentwicklungstasks, self-supervised und supervised Tasks anwendet. Hier ist es zu beachten, dass die Softwareentwicklungtasks Source Code beinhalten. Darüber hinaus erstellen wir zwei Quellcode-Corpora, eine für C# und eine für Java, und wir trainieren ein Modell, das fine-tuned werden kann, um den Wissenstransfer für die weitere Forschung zu unterstützen.

Contents

Acknowledgements **vii**

Abstract **ix**

1. Introduction **1**
1.1. Motivation 1
1.2. Problem statement 2
1.3. Research Questions 2
1.4. Thesis contribution 3
1.5. Research approach 3
1.6. Structure of the document 4

2. Background **5**
2.1. Software Development domain 5
2.2. Natural Language Processing 6
2.2.1. NLP for source code 7
2.3. Deep Learning for NLP 8
2.4. Domain adaptation in Deep Learning 10
2.4.1. Single-task learning 10
2.4.2. Transfer learning 10
2.4.3. Multi-task learning 11

3. Related works **15**
3.1. Software Development and Deep Learning 15
3.1.1. Task-specific related works 17
3.2. Transfer learning 19
3.3. Multi-task learning 21
3.4. Multi-task learning for NLP 22

4. Approach **23**
4.1. Model architecture 23
4.1.1. Transformer model 23

4.1.2. Single-task learning 25
4.1.3. Multi-task learning 25
4.2. Experimental setup 26
4.2.1. Hardware 26
4.2.2. Tensor2Tensor 27
4.3. Evaluation metrics 29

5. Datasets **31**
5.1. Benchmark Datasets 31
5.2. Language model Corpora 35
5.3. Pre-processing 36
5.4. Statistics 37

6. Results and discussion **43**
6.1. Single-task model results 43
6.2. Multi-task model results 45
6.3. Comparison between single and multi-task models 46
6.4. Comparison with state-of-the-art 47
6.5. Partial results 49

7. Conclusions **51**

8. Future Research **53**

Appendix **57**

A. Code sample **57**

B. Multi-task accuracy plot **59**

C. Results table **61**

Bibliography **63**

List of Figures

2.1. Sequence-to-sequence learning model from Sutskever et al. [46] 9
2.2. Soft parameter sharing MTL [42] 13
2.3. Hard parameter sharing MTL [42] 13

3.1. The SE tasks solved by deep learning and participated by industrial practitioners [26] 16

4.1. The Transformer model architecture [47] 24
4.2. Schema of our multi-task model setting. Each record of the input, consisting of seven tasks plus the language task, is augmented with a task id. A single unified dataset is then created and fed to the Transformer model. . 26

5.1. Sample in the Source Code Summarization dataset 32
5.2. Sample from the Code Comment generation dataset 32
5.3. Sample from the Commit message generation dataset 33
5.4. An example of extracting API sequence and its annotation from a Java method [15] 34
5.5. Example of a solution in DSL and problem statement from the AlgoLISP dataset [39] 34
5.6. Distribution of length of the tasks' corpus 39
5.7. Distribution of length of the tasks' corpus 40
5.8. Distribution of length of the corpus for the language model 41

6.1. Single-task model results on the test sets: F1 measure of ROUGE 1, 2 and L 45
6.2. BLEU score results with Single-task models in comparison with State-of-the-art. The BLEU score is not reported for the first and the last task because it is not available in the original papers. 48
6.3. BLEU score results for Single-task and Multi-task models in comparison with State-of-the-art. The BLEU score is not reported for the first and the last task because it is not available in the original papers. 49

6.4. Multi-task models' learning accuracy as function of number of steps. The red curve is the accuracy for the small multi-task model and the green curve is for the bigger model 50

List of Tables

4.1. LRZ's machine specifications 27

5.1. Number of samples of each unsupervised dataset 37
5.2. Number of samples for each task 38

6.1. Single-task models' setting 43
6.2. Single-task models' results for base (B) and tiny (T) model in terms of training steps, accuracy, BLEU score and ROUGE scores calculated on the test sets 44
6.3. Multi-task model setting 45
6.4. Multi-task model results for 630K steps calculated on the test sets 46
6.5. State-of-the-art results available. The "X" means that the metric is not reported in the paper. 47
6.6. Results with bleu.py script 47
6.7. Bigger Multi-task model setting 49

C.1. Summary table for all the results obtained with single-task and multi-task models. In bold there are the best results for each task and for each score also compared to the state-of-the-art (SOTA). 61

1. Introduction

1.1. Motivation

Machine Learning (ML) is the part of Artificial Intelligence (AI) that make the systems automatically learn and improve from experiences without being explicitly programmed. ML became popular in the 90s thanks two the increasing computer power and the growth of data availability. Nowadays, machine learning is shaping and simplifying the way we live, work, travel, and communicate. Google, Amazon, and Netflix are only some of the big companies that are exploiting this technique for image and speech recognition, natural language processing (NLP), spam detection and filtering, and advertising.
Deep Neural Networks have shown great success in various applications because of their ability to progressively extract higher-level features from raw input. There exists a variety of models and architectures that can be applied to different problems.
The NLP field improved substantially thanks to Deep Learning. In particular, Recurrent and Convolutional Neural Networks were discovered to be very effective for text data. These models require big quantities of data to be trained effectively. However, usually limited labeled datasets are available for many tasks and we need to exploit them in the best way to achieve good results and avoid overfitting in the training process. Transfer Learning and Multi-task learning have recently been applied to NLP tasks and not only to images as before, thanks to their ability to work with limited labeled data. They pushed the state-of-the-art performance in various tasks and a famous example is the case of the BERT language model [13].
Motivated also by this advantage, researchers started to investigate the application of NLP and Deep Learning techniques in different fields. One of them is the software development domain that involves problems with source code and not only natural language texts. Nowadays, it is an important field because almost every aspect of life like healthcare, transportation, and entertainment, depends on reliable high-quality software. Unfortunately, developing new software is costly and it is hard work for engineers. Therefore, there is a high demand for software tools that could help this development process in terms of speed, costs and reliability and one direction of improvement consists of the use of Machine Learning techniques. There are many academic researchers and industrial practitioners like DeepMind, Facebook, Google and Microsoft that started to inte-

grate Deep Learning solutions in their software tasks motivated by the success of deep learning in data mining and pattern recognition.

1.2. Problem statement

When applying Deep learning to software development domain tasks, we face three main problems:

Data scarcity: currently, there are limited labeled datasets that are massive and publicly available for research so we need to make the most of the available ones.

Overfitting: this is a direct consequence of the data scarcity. If a task's dataset is small, then the probability of overfitting increases.

Energy consumption: training a deep learning model might require hundreds of millions parameters and, in order to execute the training process, we need powerful hardware and a substantial amount of time.

1.3. Research Questions

Motivated from these improvements in the research community and from the problem that aroused, we investigate how Multi-task learning works with software development tasks that involve programming languages. In this project, we wanted to address the following research questions:

1. Can multi-task deep learning be beneficial for tasks in the software development domain?
2. How far is multi-task deep learning from state-of-the-art solutions in the software development domain?
3. Could the model be trained with the English language and programming languages together?
4. How does training on multiple tasks of the software development domain simultaneously compare to training on each task separately act?

1.4. Thesis contribution

In this project, we apply single-task and large-scale multi-task models to software development domain tasks. We made three main contributions:

Training of transformer architectures on software tasks: we adapt the transformer architectures to tasks involved in the software development domain and we evaluate the resulting performance

Application of Multi-task models to Software Development tasks: we trained a model on software development tasks that involve summarization and generation of text in the form of sentences or programs. The process involves English language as well as source code written in programming languages such as Java, SQL, Python and C#. We evaluate the results also comparing them to state-of-the-art single-task models.

Training of a model that can be fine-tuned: we trained a multi-task model that could be fine-tuned subsequently by other researchers in order to enable the knowledge transfer and save energy consumed by the training process.

To the best of our knowledge, this is the first work that applies large-scale multi-task models to software development tasks that involve source code and self-supervised and supervised tasks.

1.5. Research approach

In order to achieve the goal of this project and answer the research questions above, we follow the steps summarized below:

1. **Literature review:** we started our project with the paper of Li et al. [26] where we can find a review of research papers that apply Deep Learning methods to software development tasks.

2. **Choice of relevant tasks:** we focus on the tasks and papers where deep learning is applied and some of them were also tackled by industrial practitioners. We selected the more interesting tasks based on the availability of the dataset and on the relation between them.

3. **Choice of the model:** we applied the Transformer model of Vaswani et al. [47], actual state-of-the-art for sequence-to-sequence modeling, with few modifications to better fit our problem.

4. **Pre-processing and integration of the datasets:** this stage aims to pre-process the chosen datasets in order to be fed to the model.

5. **Training:** we train the models on all the tasks selected and on each task separately to understand which between single-task and multi-task models perform better.

6. **Evaluation of the results:** we first compared single-task and multi-task results. Then we made a comparison with the respective papers' counterpart. Finally, we verify and validate the research questions.

1.6. Structure of the document

The thesis is divided into 8 chapters, including this introduction. Chapter 2 explains the foundation where this thesis lays on. In particular, it is described the domain of this work, some Natural Language Processing techniques also applied to source code, and the definition of Single-task, Multi-task, and Transfer learning. Chapter 3 explains the state-of-the-art works related to this project with a focus on Multi-task learning. Chapter 4 explains the experimental setup of this work and the models' architectures used. It also briefly describes the metrics computed for the performance evaluations. Chapter 5 describes the datasets that have been used for the supervised and unsupervised training of the models, together with pre-processing and statistics on them. Chapter 6 shows the results achieved and the comparisons between different architectures and with state-of-the-art counterparts. Chapters 7 and 8 conclude the work with a discussion of the overall results and some future experiments and further development of the project.

2. Background

In this chapter, there are the fundamental concepts explored in this project. We first introduce the software development domain. Then, we describe the foundations of Natural Language Processing because the project deals with textual data and we exploited some of these techniques. Finally, we talk about Deep Learning focusing on Transfer Learning and Multi-task learning that are the concepts from where the project was built.

2.1. Software Development domain

Software design, software testing, GUI testing, strategic decision making and automated code generation are only some of the tasks part of the Software Development domain. When we deal with Software Engineering (SE), different aspects should be taken into account and each of them leads to different tasks:

- Requirement analysis: analysts need to extract requirements from natural language texts that have to be fulfilled with the final software;
- Software design: design patterns of software can be recognized;
- Development: some tasks are program learning and program synthesis, automatic software repair, code suggestion, API management, and model visualization;
- Testing: this includes the tasks that aim to test the code and find defects;
- Maintenance: it deals with malware detection and bugs;
- Management: helps the developers to estimate the cost or effort and the software size.

Recently, machine learning methods have been applied to SE tasks. Industrial practitioners expressed some concerns about the practicability of applying deep learning [18][26] due to the inherent nature of source code and to efficiency, understandability and testability of these models. However, as the paper of Li et al. [26] states, deep learning techniques achieved competitive performance on 40 SE tasks and they overcome the previous algorithms. In the literature, there are also many research papers that integrate

deep learning into SE tasks.

2.2. Natural Language Processing

Natural Language Processing (NLP) includes the set of methods used to aid computers to understand the human's natural language. Most of them rely on Machine Learning techniques.
In these years, NLP has become embedded in our daily lives. We can think of automatic machine translation, text classification for our email inboxes to detect spam, search engines that nowadays have a high degree of linguistic sophistication and dialog systems that provide an easy way to get information. Moreover, NLP-based systems enabled a wide range of applications such as Google's powerful search engine and the Amazon's voice assistant named Alexa[1].

There is a wide range of tasks that could be solved using NLP. Some examples are briefly described below:

- **Machine translation:** this task aims to translate a sentence written in a certain language into another one. Statistical phrase-based translation system consisted of many small sub-components that are tuned separately. Nowadays, Neural Machine Translation is taking place. It attempts to build and train a single, large neural network that reads a sentence and outputs a correct translation [3].

- **Text Summarization:** this is a technique for generating a concise and precise summary of texts focusing on the sections that convey useful information without losing the overall meaning. An exhaustive survey of the methods implied to solve these tasks can be found in the paper of Das and Martins [12].

- **Question-answering**: this task is also part of the Information Retrieval field, and it focuses on building systems that automatically answer questions posed by humans in a natural language. In general, these systems are made of three components such as question classification, information retrieval, and answer extraction [16].

- **Sentiment analysis**: also known as opinion mining, it aims to analyze people's opinions, sentiments, evaluations, attitudes, and emotions from written language [35]. They craft unstructured text into structured data using NLP. For example, sentiment analysis is widely used to analyze tweets or Facebook posts over a period of time to see the sentiment of a particular audience.

[1] https://developer.amazon.com/it/alexa

Understanding the human language requires understanding both the words and how the concepts are connected to deliver the intended message. Indeed, it is a difficult task for a computer. First of all, natural language is inherently ambiguous because the meaning of a word depends on the context where it lays. Also, a sentence can contain idioms like ”Better late than never” or ”Break a leg” that imply an underlying meaning. Sometimes we also have to deal with non-standard English sentences like the ones coming from social networks where hashtags and other special symbols are used. Moreover, there are neologism, recent terms that are entering common use, that might be difficult to be found in normal NLP corpus. Finally, there are hundreds of natural languages in the World, each of which has different syntax rules.These are only a part of the reasons why NLP is considered a very challenging task.

NLP techniques

The two macro-groups present in NLP are syntactic analysis and semantic analysis. The first one aims to analyze the syntax of the sentences so it involves lemmatization, morphological segmentation, word segmentation, part-of-speech tagging, parsing, sentence breaking and stemming. The second refers to the meaning that is conveyed by a text. It is more difficult due to the characteristics of languages mentioned before. It includes named entity recognition, word sense disambiguation, and natural language generation.

There are three main techniques to perform these analyses:

- Rule-based: this is the oldest approach to NLP. It focused on pattern-matching or parsing and includes regular expressions and context-free grammars;
- Traditional ML: some techniques are probabilistic modeling, likelihood maximization, and linear classifiers;
- Neural Networks approaches: large training corpus are involved and usually Recurrent and Convolutional Neural Networks are used.

2.2.1. NLP for source code

Natural Language Processing techniques are thought to be applied to languages like English, German, Italian and so on. However, we could think of applying these methods to source code data if we imagine that they are like new languages. In this case, we have to consider that we will deal with sentences that have a particular structure and include new issues [18] that are summarized below:

- **Context awareness:** when dealing with source code, we have to take into account that longer dependencies usually appear in programs. For example, a *class* can be used far away from its import statement.
- **Dynamism:** software systems evolve faster than NLP corpora because bug fixes and new features keep rolling in. If we want to build a language model, we have to take into account these changes and adapt consequently.
- **Unlimited vocabulary:** both natural languages and code have an unbounded vocabulary, but it is also true that for the first one the vocabulary usually saturates quickly and new names pop up rarely. On the contrary, for programming languages, new identifiers appear very often so fixing a vocabulary size, like is done in classic NLP methods, might be a problem. In this project, to reduce the vocabulary explosion, we decided to substitute the strings and numbers with specific tokens and we created a wide vocabulary obtained from a big set of data.
- **Dataset scarcity:** in the research community there are not so many source code corpora available. However, to build a language model for example, we need a huge quantity of unsupervised data. A way to tackle this issue was to use a web scraper which takes programs from an online repository, like GitHub, and based on a certain language extracts the data.
- **Tokenization:** when we work with languages, we might need to split the samples into tokens where a token is an "instance of a sequence of characters in some particular document that are grouped as a useful semantic unit for processing" as Manning et al. [31] stated in their book. We can also split a sentences by spaces and punctuation but most of the time it is required a knowledge on the particular language to handle tricky cases. When we have programming language, however, these particular cases increase because there are more unusual specific tokens that we wish to recognize.

2.3. Deep Learning for NLP

In the past, NLP was addressed with techniques that implied time-consuming handcrafted features and shallow machine learning models. However, from 2013, three main types of neural networks became the most widely used: Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN), and Recursive Neural Networks.
RNNs are a good choice to deal with the dynamic input sequences in NLP. Long Short-Term Memory networks (LSTM) were widely used thanks to their ability to resist more to vanish and exploding gradient problems. Bi-directional LSTMs are used in NLP to deal

with both the left and right context of words. However, their sequential computations preclude the parallelization within training samples, which becomes critical at longer sequence lengths, as memory limit batching across samples.
CNNs were chosen because they are more parallelizable than RNNs, as the state at every timestep depends only on the local context rather than on all the past states as in the RNN.
Recursive Neural Networks came from the linguistically inspired idea of treating sentences as trees rather than sequences, exploiting their inherent hierarchical structure.

In 2014, Sutskever et al. [46] proposed a general framework for mapping one sequence to another one using a neural network: the sequence-to-sequence learning. This model schematized in figure 2.1 contains an encoder module and a decoder.

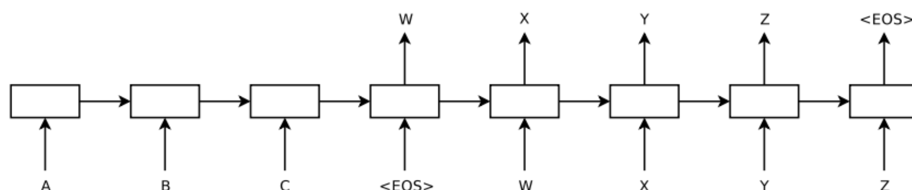


Figure 2.1.: Sequence-to-sequence learning model from Sutskever et al. [46]

The first computes a vector representation s for each input sequence that is read by the decoder module. The latter predicts the output symbol by symbol, taking into account the previously predicted output, and decomposes the conditional probability of mapping an input sequence $x_1, ..., x_n$ into an output sequence $y_1, ..., y_m$ in the following way:

$$log(p(y|x)) = \sum_{j=1}^{m} log(p(y_j|y_{<j}, x, s)) \tag{2.1}$$

This model was particularly successful for Machine Translation [9] and nowadays is the leader in natural language generation tasks.

The main drawback of sequence-to-sequence learning is that it requires to compress the entire content of the source sequence into a fixed-size vector. To reduce this issue, attention-based mechanisms have been proposed. They allow the decoder to look back at the source sequence hidden states, which are then provided as a weighted average and additional input to the decoder. A definition of attention is given:

Definition 2.1 (Attention [47]). *Mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted*

sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

A particular type of attention is called "Self-attention". It looks at the surrounding words in a sentence or document to obtain more contextually sensitive word representations. It can be done at multiple layers and that is exploited on the Transformer architecture of Vaswani et al. [47], the actual state-of-the-art for Neural Machine Translation.

2.4. Domain adaptation in Deep Learning

In this section we explore two main techniques that deal with multiple tasks at the time: Transfer Learning and Multi-task learning. They both exploit the idea that two tasks can share information in order to improve the learning process.

2.4.1. Single-task learning

Single-task learning is the classic machine learning approach that consists of solving one task training a model. The objective is to improve the performances based on a certain metric, specific for the problem, that is optimized by fine tuning the model parameters until there is no more gain.
This paradigm ignores the information that comes from related tasks that could help to improve the performances [42]. If we share the representations between related tasks, we can enable our model to generalize better on our original task, and this is what Transfer Learning and Multi-task learning do.

2.4.2. Transfer learning

Transfer learning methods came up with the real-world example that people can apply knowledge learned previously to solve new problems faster or with better solutions. For example, learning to recognize apples might help to recognize pears [34].
Transfer learning relax the assumption that the domains, tasks, and distributions used in training and testing have to be the same. A definition is given below

Definition 2.2 (Transfer Learning [34])**.** *Given a source domain D_S and learning task T_S , a target domain D_T and learning task T_T , transfer learning aims to help improve the learning of the target predictive function $f_T(\cdot)$ in D_T using the knowledge in D_S and T_S , where $D_S \neq D_T$, or $T_S \neq T_T$.*

A Domain D consists of a feature space χ and a marginal probability distribution $P(X)$, where $X = \{x_1, ..., x_n\} \in \chi$. Given a specific domain $D = \{\chi, P(X)\}$, a task T consists of

a label space Y and an objective predictive function $f(\cdot)$ which is not observed but can be learned from the training data, which consist of pairs $\{x_i, y_i\}$, where $x_i \in X$ and $y_i \in Y$.

The main questions to answer in this scenario regard what, how and when to transfer. The knowledge to be transferred has to be common between different domains and tasks so that it may help improve the performance for the target. Subsequently, learning algorithms have to be developed in order to decide how to transfer this knowledge. Finally, we need to understand when the transfer is beneficial and when, on the contrary, it is unsuccessful and could lead to a ”negative transfer” [50] that is the hurt of performances in the target domain. In general, when the source domain and target domain are not related to each other, brute-force transfer may not work. However, understanding when two tasks are ”sufficiently” related is not trivial and it's an open research topic.

Transfer learning can be categorized in three ways [34]:

- **Inductive**: the target task is different from the source task, independently from the domains. We can have two scenarios: there are a lot of labeled data in the source domain, similar to a *Multi-task learning* setting, or there are not at all, like in a *self-taught* setting.
- **Transductive**: the source and target task are the same while the domains are different. In this case, no data is available for the target.
- **Unsupervised**: the focus is on solving unsupervised tasks in the target domain and no data is available for either of them.

2.4.3. Multi-task learning

Multi-Task Learning (MTL) is a learning paradigm that aims to leverage useful information contained in multiple related tasks to help improve the generalization performance of all the tasks. A formal definition is given:

Definition 2.3 (Multi-task learning [54])**.** *Given m learning tasks $\{T_i\}_{i=1}^m$ where all the tasks or a subset of them are related, multi-task learning aims to help improve the learning of a model for T_i by using the knowledge contained in all or some of the m tasks*

The idea of MTL came also from real life examples. If a child is trained from birth on only a single task like ”play tennis”, he would be unlikely to learn it. In order to achieve the goal, he also has to learn to walk, run, jump, and so on. The complementary tasks are similar but not equal to the main task and this similarity enables to learn also to play

tennis.

The success of MTL is given to the following mechanism also described in the paper of Ruder [42]:

- **Implicit data augmentation:** it effectively increased the sample size. Moreover, given that different tasks have different noisy patterns, learning them simultaneously helps to learn a more general representation averaging them.
- **Attention focusing:** the tasks provide further evidence for the relevance or irrelevance of the features of another task. This helps the model to focus the attention on those features that are relevant for most of the tasks involved.
- **Eavesdropping:** there might be features of a task C that are easy to be learned from a task B but are difficult from a task A. This might be because A interacts with the features in a more complex way or because other features are impeding the model's ability to learn C. In this case, MTL allows the model to "eavesdrop" so to learn C through B.
- **Representation bias:** the model tend to be biased toward representation that other tasks prefer, so if the tasks are sufficiently similar, this will increase the performances.
- **Regularization:** MTL introduces also an inductive bias. If several related tasks are given at the same time, these tasks can be used as valuable sources of inductive bias for each other. This bias helps the model to avoid overfitting.

MTL architectures

The Multi-task architectures can be divided based on how they share the parameters, on the modality of input and output and on the way they give the results.

MTL can have hard or soft parameter sharing policies. The first is the most used and usually is applied by sharing the hidden layers between all task and keeping some task-specific layers at the end. This setting reduces the risk of overfitting and Baxter [5] showed that the risk of overfitting the shared parameters is N times smaller than overfitting the task-specific parameter, where N is the number of tasks involved. A schema of this setting is in figure 2.3. The soft sharing policy (fig. 2.2), on the contrary, states that each task has its own model and parameters but the distance between them is regularized in order to encourage the parameters to be similar.

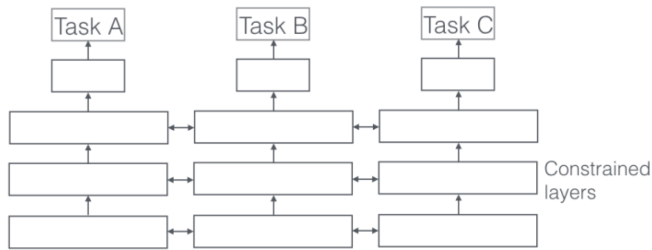


Figure 2.2.: Soft parameter sharing MTL [42]

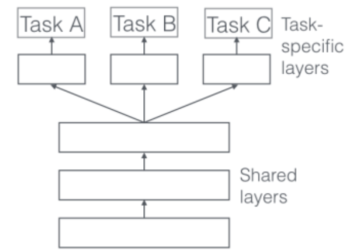


Figure 2.3.: Hard parameter sharing MTL [42]

A second division of architectures can be made based on the setting of input and output. First, a MTL model can have a single input and a single output even if it implies many tasks. In this case, the input data are combined with an identification signal at the beginning. The model is still under development so there is not a formal research paper in the literature yet. On the other hand, some architectures have multiple inputs and multiple outputs [23].

Furthermore, MTL architectures can be divided based on the moment in which they give the results. The classic setting is the one with all the outputs given at the end. An example is the architecture of Kaiser et al. [23] that works with multiple domains and achieved state-of-the-art performances. On the contrary, other models are in a hierarchical form and give the output at every step of the hierarchy. This is the case of the model from Hashimoto et al. [17] and Sanh et al. [43].

3. Related works

This work concerns Software development, Natural Language Processing, and Domain adaptation. This chapter discusses related works for these topics. In the first section, we highlight the recent papers about the application of Deep Learning techniques in Software Development. Subsequently, we describe some interesting works in Transfer Learning and Multi-task learning on NLP.

3.1. Software Development and Deep Learning

In recent years, there has been an increasing trend for industrial practitioners and academic researchers to integrate Deep Learning (DL) into Software Engineering (SE) tasks. The success was also driven by the advances of Deep Learning in data mining and pattern recognition.

The study of Mou et al. [32] is the first to investigate programs by Deep Learning. They first create a new method for representation learning of source code motivated by the claim that existing NLP algorithms were not suitable for programs. Indeed, programs contain a certain structure stronger than natural languages, and they have a different concept of "neighborhood" because neighboring elements in a program source code are not necessarily near to each other. They also explored the feasibility of analyzing programs by deep neural networks with their representation learning method, in particular with the task of program classification.

The paper of Li et al. [26] is a recent bibliographic analysis on almost a hundred research papers where Deep Learning is applied to SE tasks. We used this study as a starting point for the literature review of this project. They extract 41 main tasks solved also thanks to Deep Learning solutions as shown in picture 3.1 and they underline that 13 of them have at least one industrial practitioner involved.

Moreover, they identify the steps that should be done to solve a SE task with Deep learning and they take, as an example, the task of bug reports summarization solved with an Autoencoder [25]. The steps are almost the same that we follow in this thesis and, in order, are:

1. SE data collection

#	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 3.1.: The SE tasks solved by deep learning and participated by industrial practitioners [26]

2. SE data preprocessing
3. model selection and configuration
4. input construction
5. model training
6. model application

From this study, it also emerges that several people admit concerning with the use of Deep Learning in SE. This is also stressed in the paper of Hellendoorn and Devanbu [18] that describes some issues of modeling source code that make the creation of an efficient DL model more difficult with respect to modeling a normal language (eg. English). For example, source code has unlimited vocabulary because you create a new ”word” for each new identifier’s name so a classic NLP strategy of fixing the vocabulary size might not be the correct one in this case.

3.1.1. Task-specific related works

In this section, we highlight the five research papers that have been used as counterparts for the comparisons of the results achieved in this thesis.

Source Code Summarization

There are billions of lines of code in online repositories and it can be extremely useful for applications like code search or tutorial to have short summaries of pieces of code.
Iyer et al. [21] proposed ”CODE-NN”, an end-to-end neural attention model to generate natural language summaries of C# and SQL code.
Their model is based on LSTM and is guided by an attention mechanism on the source code snippets to generate a summary one word at a time.
The dataset to train and test the model is retrieved from StackOverflow[1], and it was used after a process of cleaning that included to strip out of all the comments, to replace too specific tokens, and to insert special start and end tokens.
They outperformed all the other methods for source code summarization on the METEOR[4] and BLEU-4[36] score thanks to the model ability to perform better content selection and to focus on more salient parts of the code given by the attention methods used.
The performances are higher for C# than SQL because the first language contains informative variable names that are directly related to the goal of the code. As drawbacks, this model sometimes gives out-of-context outputs for low frequency tokens, very long dependencies or compositional structures of the inputs.
They used this model also to perform code retrieval from a natural language query where they outperform the baseline.

Code Comment Generation

Code comments are often mismatched, missing or outdated in software projects. There are various techniques to generate them based on manually-crafted features and Information Retrieval but they have some weakness. First, they fail when we want to extract accurate keywords for identifiers and method poorly named. Second, they rely on the similarity of code snippets retrieved to find the right comment.
Hu et al. [20] proposed ”DeepCom”, an approach based on Deep Learning and NLP to generate comments of Java methods automatically using identifier names, formatting, semantics, and syntax features. They outperform the state-of-the-art and they evaluate the model based on the BLEU-4 score [36]. They formulate the problem similar to a Neural Machine Translation [51] problem where the ”translation” is between Java methods and

[1] http://stackoverflow.com

English comments.
The Java code fed to the model is first converted in AST (Abstract Syntax Trees) sequences with a method they built on purpose, and then it is analyzed to treat out-of-vocabulary token. They used a Seq2Seq model with Encoder and Decoder LSTM, and an attention component. They built a language model for the code and one for the English comments, on the contrary of the past state-of-the-art methods "CODE-NN" [21] described above.

Commit message generation

Commit messages are natural language descriptions of changes in source code snippets. The repositories store these messages along with diffs that represent the differences between the current and the previous version of the files affected of changes.
Jiang et al. [22] create a method to automatically translate diffs info commit messages to help programmers and speed up the process. Their intention is not to substitute automatic generators of commit messages that summarize what changed in the versions, but they want to help to insert the reason why something was modified.
They achieved the goal employing a Neural Machine Translation setting with diffs as inputs and commit messages as targets. Their model is an Encoder Decoder with attention added to deal with diffs that are much longer than natural language sentences.
The dataset was extracted by GitHub projects written in Java and it was cleaned and tokenized before the used. They created a QA (Quality Assurance) filter to investigate whether the output of the model generated good commit messages.
They achieved good performances on the BLEU-4 score and contributed to the advance of the state-of-the-art.

API Sequence Recommendation

Developers invoke API sequences in order to reuse existing libraries or frameworks. A method invocation sequence among the APIs is call "API usage sequence" and obtaining it helps the programmers.
In the past, there were methods able to generate such sequences using statistical world alignment models that rely on a bag-of-words assumption [40]. However, these approaches did not consider the sequences of words and fail to recognize the semantics of natural language queries.
Given that, Gu et al. [15] proposed "DeepAPI", a method based on Deep Learning able to generate relevant API usage sequences (JDK APIs) from natural language queries. They formulated the problem as a machine translation one and they used an attention-based RNN Encoder-Decoder model tested on a GitHub Java corpus of more than 7 million samples. To boost the performances of the model, they helped it to consider the indi-

vidual importance of APIs with a penalty term in the loss function with an IDF-based weighting. With this model, they outperform the twp state-of-the-art approach on the BLEU-4 score: "Code Search with Patter Mining"[28] and "SWIM"[40]. This model embeds words into a continuous semantic space on the contrary of the bag-of-words methods. Moreover, it can learn a sequence of words with their relative position. Finally, it can learn common patterns of API sequences instead of searching for specific samples.

Program learning and synthesis

Polosukhin and Skidanov [39] presented an algorithm to synthesize programs from user specifications. They combined methods from the Program Synthesis domain and Deep Learning and created an efficient search algorithm guided by a Seq2Tree [2] model. The model is a sequence encoder followed by a tree decoder augmented with attention that computes the probabilities of each symbol in the AST (Abstract Syntax Tree) tree. The probabilities are used inside a Tree Beam Search.
In particular, they synthesize a program from a short description and several input/output pairs in order to diminish the inherent ambiguity of natural language descriptions. They create a specific dataset to test the model called "AlgoLISP".
They pushed the state-of-the-art with an accuracy of 85.5% on the test set. The performances of the model grows if the search explores more trees and the correct tree is more likely to be found in the early steps of the search. If the depth of the tree increases, the accuracy reduces because there are more nodes to be predicted.

3.2. Transfer learning

"Transfer learning" regards the improvement of learning in a target task through the transfer of knowledge from a related task that has already been learned. In the literature, there are various research paper on this topic that is becoming very popular nowadays.

Transfer learning with language models have demonstrated that unsupervised pre-training is an integral part of many language understanding systems. A big improvement in the NLP world was achieved with the introduction of ELMo, UlmFit, and BERT. ELMo (Embeddings from Language Models) was proposed by Peters et al. [38] and is a new type of deep contextualized word representation that aims to solve the difficulties of learning high-quality representations of words with complex characteristics and different possible uses depending on the context. The main difference between ELMo and traditional word embeddings is that to each token is assigned a representation that is a function of the entire input sequence. They used the vectors coming from a biLSTM

trained with a coupled language model objective on a large text corpus. The higher-level states capture context-dependent aspects of words meaning, whereas the lowers model the syntax. If these representations are added to existing models, they can significantly improve the state-of-the-art on various NLP tasks. In particular, they demonstrated the improvement of question answering, textual entailment, and sentiment analysis tasks.
ULMFiT (Universal Language Model Fine-tuning) is a work from Howard and Ruder [19]. They described a method that can be used to apply transfer learning for any task for NLP like in computer vision tasks previously. They also focus on avoiding catastrophic forgetting during fine-tuning, that is a common problem on language models architectures when fine-tuned with a classifier. They outperform substantially the state-of-the-art on six datasets on the tasks of sentiment analysis, question classification, and topic classification.

A big step forward on NLP was made by Devlin et al. [13] that created BERT (Bidirectional Encoder Representations from Transformers) that beat the state-of-the-art on 11 NLP tasks. Previous approaches used unidirectional language models to learn general language representations. Instead, BERT use multi layers bidirectional Transformer encoder with bidirectional self-attention [47] as pre-training architecture. The bidirectional model was possible thanks to a ”masked language model” pre-training objective that randomly masks some of the tokens from the input and the objective is to predict the original vocabulary id of the masked word based only on its context.
They proposed two model sizes: BERT base with 12 layers and BERT large with 24. The model is pre-trained on the BookCorpus[56] and English Wikipedia.
Recently, it was proposed a new model able to overcome BERT performances on various tasks: XLNet by Yang et al. [52]. They aimed to solve the problems of previous Autoregressive models and BERT. The formers are not suitable for modeling deep bidirectional context. On the other side, as stated by the authors of XLNet, BERT neglects dependency between the masked positions and suffers from a pretrain fine-tune discrepancy. The artificial symbols used by BERT during pre-training are absent from real data at fine-tuning time, resulting in a pretrain-finetune discrepancy. Moreover, the predicted tokens are masked in the input so BERT assumes the predicted tokens are independent of each other given the unmasked tokens, which is an oversimplified assumption.
XLNet solved these issues thanks to the Permutation Language Modeling in the pre-training phase. It keeps the original sequence order, uses positional encodings, and relies on a special attention mask in Transformers to achieve the permutation of the factorization order. The model, Large or Base, is based on Transformer-XL[11] with appropriate modifications to make it works with the permutation operation XLNet outperforms BERT on 20 tasks and achieves state-of-the-art results on 18 tasks including question answering,

natural language inference, sentiment analysis, and document ranking.

3.3. Multi-task learning

The idea of Multi-task learning (MTL) comes from the intuition that tasks could share knowledge and gain benefits.
Caruana [7] proposed one of the first papers on MTL and he investigated the advantages and disadvantages of this paradigm and why it improves the generalization. They also introduce a possible use of MTL on Decision Tree that can be useful when the tasks have a convoluted relationship and one task needs the result of another.
The work from 2008 of Collobert and Weston [10] is an old project that started to use MTL for NLP. They built a Convolutional Neural Network where the tasks are trained jointly. They have both supervised tasks (Part-of-speech tagging, chunking, named-entity recognition and semantic role-labeling) and the language model that is unsupervised and trained on the entire Wikipedia website. This mix of learning methods was a new form of semi-supervised learning. In particular, they focus the work on the semantic role-labeling problem that consists of giving a semantic role to a syntactic constituent of a sentence and it is considered the more difficult task between the other tackled.

MLT has been used in various domains like speech recognition [44], computer vision [14] [55], drug discovery [41] and natural language processing [29]. The latter is discussed in the following section.
MTL achieved good performances also in multiple domain settings like in the paper of Ngiam et al. [33] where they build a model for audio and video data with a focus on learning representations for speech audio which are coupled with videos of the lips. The model used is a deep Autoencoder that was chosen after a comparison with a Restricted Boltzmann Machine architecture. They pushed the state-of-the-art on visual speech classification on the "AVLetters" dataset and they achieved effective shared representation learning.
Another instance of multi domain tasks is in the paper of Kaiser et al. [23] where they deal with speech, images and text data. The different types of data required a way to convert inputs into a joint representation space. They achieved it by means of "Modality networks" that convert the input in the common representation space and then in the output space. The overall network consists of modality-nets, an encoder, an I/O mixer, and an autoregressive decoder. They showed that a transfer of learning is possible even between different type of tasks and that the tasks with fewer data available, like parsing, can achieve better performances with this network.

3.4. Multi-task learning for NLP

In these years, various papers showed that combining multiple NLP tasks in one model can give improvements in performances and it can generate also a better representation of the text involved.

The paper of Luong et al. [30] improved the state-of-the-art for the translation task "English to German" and constituent parsing. They exploit the paradigm of sequence2sequence learning described in section 2.3 and they investigated how to combine it with MTL. In particular, their model is a deep LSTM without attention. They also investigated three MTL settings: one-to-many (encoder shared), many-to-one (decoder shared) and many-to-many (multiple encoders and decoders are shared).

A recent paper from Liu et al. [29] describe the MT-DNN model that combines multi-task learning and language model pre-training for NLP tasks. They showed that these two ways can be coupled to improve the learning of text representations and to boost the performances of various tasks. In particular, they analyzed the tasks of single-sentence classification, pairwise text classification, text similarity scoring, and relevance ranking. The experiments were conducted on three benchmarks: GLUE [49], SNLI [6], and SciTail[24]. The model is made of shared layers at the lower part and task-specific layers at the top. The last lower layer is a Transformer encoder [47] that captures contextual information and generates shared embeddings, similar to BERT [13], but it learns the representation using multi-task objectives and not only pre-training. They overcome the state-of-the-art in all the tasks, except WNLI (Winograd NLI [49]).
In the literature we can also find the work of Hashimoto et al. [17] where they describe a multi-task model for chunking, dependency parsing, semantic relatedness, and textual entailment tasks. They proposed a model that follows a hierarchy having POS tagging at the first layer and textual entailment at the last one. It can be trained end-to-end and they demonstrated that hierarchies improve the performances. Differently from other papers, they investigate the potential of handling different types of NLP tasks, rather than closely-related ones, in a single hierarchical deep model.
The hierarchies were also supported by Sanh et al. [43] that created a hierarchical multi-task architecture combining four different NLP tasks. They achieved state-of-the-art performances on Named Entity Recognition, Relation Extraction and Entity Mention Detection.
Moreover, it has been showed that multi-task learning is effective when we have to deal with tasks of different complexity. Søgaard and Goldberg [45] demonstrated that low-level tasks, like Part-of-speech tagging, are better modeled in the low layers of their architecture based on bidirectional LSTMs.

4. Approach

In this chapter, we discuss the model architecture for the single and the multi-task setting. Then we describe the experimental setup with the hardware and software. Finally, we show the metrics we used to evaluate our models.

4.1. Model architecture

In this project, we adapt the Transformer model by Vaswani et al. [47] to our tasks motivated by the fact that it is the current state-of-the-art for sequence-to-sequence modeling. In the following sections, we describe the general Transformer model and we specify the architectures for the single-task and multi-task learning.

4.1.1. Transformer model

The Transformer model was introduced by Vaswani et al. [47] on 2017 and a TensorFlow implementation is available as part of the Tensor2Tensor package (subsec. 4.2.2).
This Deep Learning model is based on attention to boost the speed with which the model can be trained. It is particularly effective for machine translation tasks and it is the state-of-the-art for various problems.
The main advantages of this model is the possibility to parallelize the computations, that is something RNNs and CNNs suffer from.

The Transformer is an Encoder-Decoder (fig 4.1) model where the encoder maps a variable-length source sequence to a fixed-length vector, and the decoder maps the vector representation back to a variable-length target sequence.

The main components of the model are Encoder, Decoder and attention mechanism:

- **Encoder:** it is made of six identical layers stacked and each of them has two sublayers: a multi-head self-attention mechanism and feed-forward neural network.
- **Decoder:** it is also made of six layers stacked but it has three sublayers, the ones above plus a multi-head attention over the output of the encoder stack. The self-attention sublayer is slightly modified to prevent positions from attending to sub-

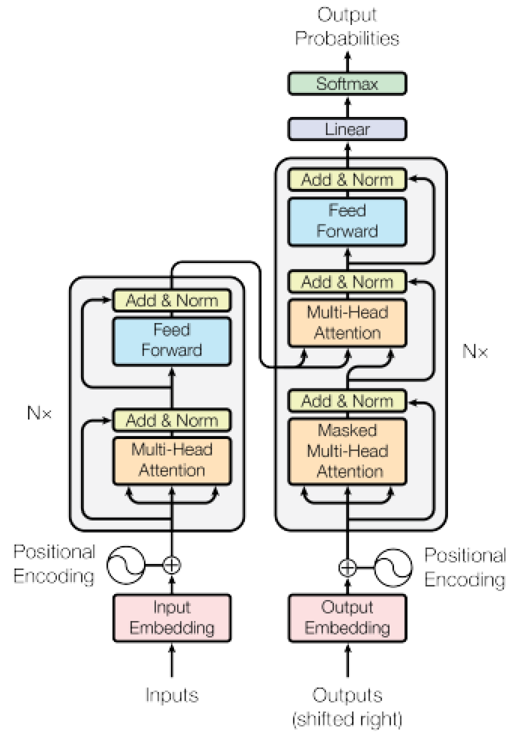


Figure 4.1.: The Transformer model architecture [47]

sequent positions and ensure that predictions for position i can depend only on the known outputs at positions less than i.

- **Attention:** the Transformer uses an attention mechanism, called Scaled Dot-Product Attention, which directly models the relationships between all words in a sentence irrespective of their positions. As the model processes each word, it allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word. The attention function is computed on values and keys, of dimension d_v and d_k, coming from the encoding of the input, and queries coming from the encoding of the translated sentence so far.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{4.1}$$

The scaling factor $1/\sqrt{d_k}$ regularizes the dot product of queries and keys that might grows large in magnitude and pushes the softmax into region where it has extremely small gradients.
In order to jointly attend to information from different representation subspaces at

different positions, the model performs a "Multi-Head attention" where W_i^Q, W_i^K, W_i^V are parameter matrices and they used $h = 8$ parallel attention layers:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^0 \tag{4.2}$$

$$\text{where } head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

4.1.2. Single-task learning

In order to better understand our multi-task model results, we train the model with a single task at the time.
This can be done setting the MODEL variable to "Transformer", the HPARAMS to "transformer_base" or "transformer_tiny" and considering the generator for the single problem only. We don't consider the language model in this case, so we generate the TFRecords files and the vocabulary for the specific task without the language model and dataset.

We will also compare the results with the respective ones in the papers. However, even if some scores are not published, we will check Accuracy, BLEU and ROUGE score for all the tasks for future research purpose. All the metrics are calculated on the same datasets of the papers' counterparts for fair comparisons.

4.1.3. Multi-task learning

The multi-task model is shown in figure 4.2. The eight tasks, seven tasks plus the language model, are given to the data generator script to prepare the TFRecords and add the task IDs. We used the language model in order to make the overall architecture better understand the dependencies between tokens and the structure of the sentences. Moreover, it helps to analyze other sequences that might exist out of the training set distribution.
The transformer model itself is composed of 12 layers of only decoder units. We had to use a decoder-only architecture because we deal with supervised tasks but also the language model that is unsupervised and it is not intended for encoder-decoder models.
To build a multi-task model, T2T defines the "MultiProblems" sub-classes and we can specify the list of problems to include in the training. We also have to specify the vocabulary obtained from the data generation of the language model. In this case, the variable MODEL is set to "transformer" as for the single-task, but the HPARAMS is set to specific parameters.

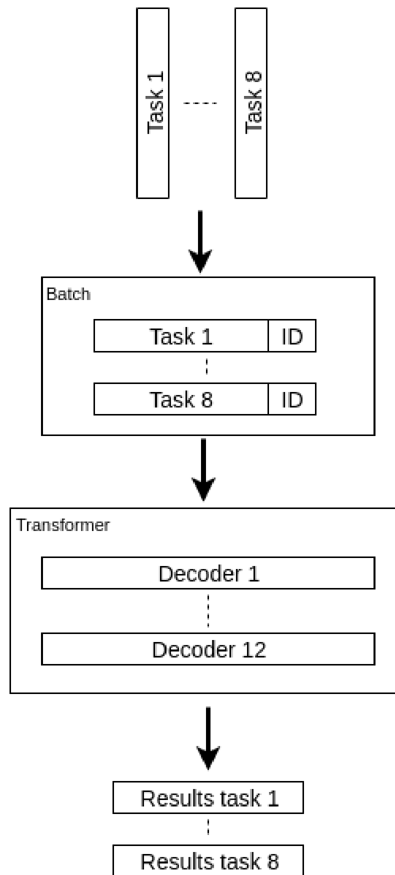


Figure 4.2.: Schema of our multi-task model setting. Each record of the input, consisting of seven tasks plus the language task, is augmented with a task id. A single unified dataset is then created and fed to the Transformer model.

4.2. Experimental setup

In this section, we describe the hardware and software use for this project.

4.2.1. Hardware

The training of multi-task models usually require a substantial amount of time and memory. In order to speed-up the experiments we used mainly two machines offered by the LRZ (Leibniz-Rechenzentrum), the European supercomputing center in Garching near Munich. The specifications are in table 4.1. On the first machine, the one with a single GPU, we trained the single-task models because they require less computation capabilities. The training of the multi-task models was made on the DGX-1 that has 8 GPUs.
We used half precision fp16 to speed-up the training even if we might sacrifice a little the performances in terms of accuracy.

Machine name	P100	DGX-1
GPUs	1x NVIDIA P100	8x Tesla V100
GPU Core	3.5 K	41 K
Memory	16 GB	8 x 16 GB

Table 4.1.: LRZ's machine specifications

4.2.2. Tensor2Tensor

Tensor2Tensor (T2T)[48] is a library with Deep Learning models and datasets created to speed up the research on DL and make it more accessible. It is actively used and maintained by researchers and engineers within the "Google Brain" team and a community of users. The development is done in their GitHub[1]. T2T is based on Tensorflow[1], an end-to-end open source platform for machine learning, and contains various T2T-specific abstractions with focus on performance and usability. Researchers can train models on CPU, GPU, and TPU with minimal device-specific code.
There are different models and datasets available to support Neural Machine Translation tasks and others across multiple media (text, images, video, audio). The models come in different variants and with different hyperparameters sets. They are customizable by the user to fit special demands.
T2T also provides facilities to evaluate the results of the model like the automatic calculation of some metrics (e.g. accuracy, BLEU score, ...).
The most recent version of T2T is the 1.13.4, but in this project we used the 1.12.0 that is older but has better support for multi-task models.

Mode of operation

Tensor2Tensor operations are divided into three main steps:

1. **Data Generation:** in order to train the model, T2T prepares the data in advance. It transforms the input into a specific format that can be directly read by the model at training time. The data generator class define how the raw data is divided in input and target. The pairs are then shuffled and saved into *TFRecord* files. Each of the task involved in the model is associated with a task ID used later for the decoding. If we do not specify the vocabulary file, a new one is created based on the input and output data. For our multi-task model, we specify the vocabulary from the language model data generation, whereas for the single-task experiments we allowed the creation of the vocabularies, one for each task.

[1] http://github.com/tensorflow/tensor2tensor

To run the data generators we can use the following command where the "USR_DIR" contain the generators' files and the "PROBLEM" variable is the name of the class:

```
$ t2t-datagen --t2t_usr_dir=$USR_DIR --data_dir=$DATA_DIR
  --tmp_dir=$TMP_DIR  --problem=$PROBLEM
```

An example of data generator for the task "code comment generation" can be found in appendix A.

2. **Training:** at training time, the model uses the TFRecords created before as input data. We can specify the model and various parameters including the batch size and the maximum sequence length. During the training, T2T saves the checkpoints for the model that contains the training state at a specific step and can be retrieved to continue a training that was interrupted. It also saves the event files for training and validation that can be used for the visualization of the learning process with Tensorboard. After setting some variables like the $MODEL and $HPARAMS, we can start the training with the following command:

```
$ t2t-trainer --t2t_usr_dir=$USR_DIR --data_dir=$DATA_DIR
--problem=$PROBLEM --model=$MODEL --hparams_set=$HPARAMS
--output_dir=$TRAIN_DIR --train_steps=50000
--eval_steps=2000 --worker_gpu=8
```

3. **Decoding:** after the training process, T2T helps to decode a given file from the model. We can do it with the command below where we specified also the "BEAM SIZE" and the length penalty "ALPHA". The first parameter is used for the BEAM search that, instead of greedily choosing the most likely next step as the sequence is constructed, expands all possible next steps and keeps the K most likely, where K is the "BEAM SIZE" parameter and controls the number of beams or parallel searches through the sequence of probabilities. The alpha is needed for a length normalization procedure to account for the fact that we have to compare hypotheses of different lengths. Without it, the Beam search will favor shorter results over longer ones on average [51].
In the case of multi-task models, we must specify the ID of the task we are decoding as an hyperparameter of the "t2t-decode" command. T2T also provides a set of commands to calculate some basic metrics. For example with the "t2t-bleu" command, we can calculate the BLEU score given the decoded file and the labels.

```
$ t2t-decoder --t2t_usr_dir=$USR_DIR --data_dir=$DATA_DIR
--problem=$PROBLEM --model=$MODEL --hparams_set=$HPARAMS
--output_dir=$TRAIN_DIR --decode_from_file=$DECODE_FILE
--decode_hparams="beam_size=$BEAM_SIZE,alpha=$ALPHA"
--decode_to_file=decodedApi.txt
```

4.3. Evaluation metrics

To evaluate our model we used three metrics: accuracy, BLEU score, and ROUGE score. We calculate them for all the tasks for future researches even if the papers' counterparts calculate only some of them.

- **Accuracy:** it is used to measure the performance of the models. The metric is computed as:

$$Accuracy = \frac{correct\ predictions}{predictions} \tag{4.3}$$

- **BLEU score:** this metric was proposed by Papineni et al. [36] and aims to evaluate machine translation performances in a quick, inexpensive and languange-independent way. This metric has the ability to correlates highly with human evaluation. It uses a weighted average of variable length (1-grams to 4-grams) phrase matches against the reference translations.
The score is computed as:

$$BLEU = BP \cdot exp(\sum_{n=1}^{N} w_n \log p_n) \tag{4.4}$$

where p_n is the precision of n-grams, N is the maximum number of grams we consider, usually set to 4, and $w_n = \frac{1}{N}$ is the weight for each p_n.

$$p_n = \frac{\#n - grams\ appear\ in\ the\ reference + 1}{\#n - grams\ of\ candidate + 1} \tag{4.5}$$

BP is the "brevity penalty" which penalties overly short candidates that may have a higher n-gram precision, where r is the length of the reference sequence, and c is the length of the candidate sequence.

$$BP = \begin{cases} 1 & c > r \\ e^{(1-r/c)} & c \leq r \end{cases} \tag{4.6}$$

- **ROUGE score:** ”Recall-Oriented Understudy for Gisting Evaluation” was proposed by Lin [27]. It counts the number of overlapping units (n-gram, word sequences, and word pairs) between the generated summary to be evaluated and the target. There are different types available. Below we show a formal definition where n is the length of the n-grams and $Count_{match}(gram_n)$ is the maximum number of n-grams co-occurring in a candidate summary and a set of reference summaries.
  We focus on 1-grams, 2-grams and longest common subsequences: ROUGE-1, ROUGE-2 and ROUGE-L.

$$ROUGE - N = \frac{\sum_{S \in \{ReferenceSummaries\}} \sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{S \in \{ReferenceSummaries\}} \sum_{gram_n \in S} Count(gram_n)} \quad (4.7)$$

5. Datasets

This project deals with different tasks and for each one we have a specific dataset. There are eight different tasks, seven supervised plus the language model that is unsupervised. In this chapter, we describe these datasets and the techniques we used to pre-process each of them.

5.1. Benchmark Datasets

For each supervised tasks, there is a corresponding research paper where the task is solved with single-task Deep Learning models. We decided to use the same dataset of these papers in order to make comparisons of the results later on.

Source Code Summarization

This task includes three sub-tasks, one for each programming language involved: C#, Python and SQL.
Iyer et al. [21] collected the data from Stack Overflow in December 2014. Using the tag related to the language, they were able to collect almost 1 million posts per language. From each post, they extracted the title and the code of the accepted answers that contain exactly one code snippets in the ”$< code >$” tag. In this way, they added to the corpus the pairs $< code, title >$. Subsequently, they cleaned the code from the queries without relation with the code snippets with a semi-supervised classifier. The final dataset can be downloaded from their GitHub page[1]. An instance of a sample in the dataset relative to Python can be seen in picture 5.1.

Code Comment generation

The code comment dataset has as input a code snippet and as target a description in natural language. Hu et al. [20] created a dataset from 9,714 Java projects from GitHub.

[1] hyperref[]https://github.com/sriniiyer/codenn

Source code Python:
```python
from pygithub3 import Github
username = raw_input("Please enter a Github username: ")
password = raw_input("Please enter the account password: ")
gh = Github(login=username, password = password)
get_user = gh.users.get()
user_repos = gh.repos.list().all()
for repo in user_repos:
    print repo.language
```
Description:
Getting repository information using pygithub3 for Python

Figure 5.1.: Sample in the Source Code Summarization dataset

They used Eclipse's JDT compiler[2] to parse the Java methods into ASTs and extract corresponding Javadoc comments which are standard comments for Java methods. They used only the first sentence appearing in the comments as description because it characterizes the functionalities of the methods. A sample is in picture 5.2 and the whole dataset can be found in their GitHub page[3].

Source code Java:
```java
public void handleEntryExpiredSA(EntryExpiredBusPacket packet) throws Exception {
  handleEntryExpiredCoreSA(packet.getEntryHolder(),packet.getTransaction(),packet.isFromReplication());
}
```
Comment:
Handles EntryExpired packets.

Figure 5.2.: Sample from the Code Comment generation dataset

Commit Message Generation

In this task, the goal is to generate commit messages from diffs that represents the difference between the current and previous version of the affected files.
The dataset of diffs and human-written commit messages come from the top 1k Java GitHub projects. They extracted the first sentence of the commit messages because usually it summarizes the entire message. Then, they removed issue ids from the extracted sentences and removed commit ids from the diffs because they are unique identifiers and they want to limit the size of the vocabulary. They also removed any diff that is larger than 1MB and that included diffs of merges and rollbacks. They set the maximum length of the target sequences, the commit messages, at 30 tokens, because the first sentences from the commit messages tend to be short. The diffs sequence maximum length was set to 100 tokens.
They also created a "V-DO" (Verb-Direct Object) filter to diminish the problem of bad

[2]http://www.eclipse.org/jdt/
[3]https://github.com/xing-hu/DeepCom

quality data because the messages can have different writing styles or can be poor written. In the end, the dataset is composed of 32k pairs (fig 5.3) and is available in their GitHub page[4].

```
Diff:
--- a/core/.../CursorToBulkCursorAdaptor.java
+++ b/core/.../CursorToBulkCursorAdaptor.java
@@ -143,8 +143,7 @@ public final class
CursorToBulkCursorAdaptor ...
public void close() {
maybeUnregisterObserverProxy();
- mCursor.deactivate();
-
+ mCursor.close();
}
public int requery(IContentObserver observer, ...

Message:
Call close ( ) instead of deactivate ( ) in CursorToBulkCursorAdaptor . close ( )
```

Figure 5.3.: Sample from the Commit message generation dataset

API Sequence Recommendation

In this task, we aim to extract an API sequence recommendation from a natural language annotation. Gu et al. [15] constructed the corpus from Java projects of GitHub created from 2008 to 2014 with at least one star. In total, they collected 442,928 projects. Then they extract the pairs $< APIsequence, annotation >$ with a detailed method explained step-by-step in their paper. In picture 5.4 we can see an example of Java code from which they extracted an API sequence and the corresponding annotation. In the end, they obtain a database consisting of more than 7 million pairs.

Program Learning and Synthesis

The goal of this task is to create a program from a natural language specification. To this purpose, Polosukhin and Skidanov [39] created ”AlgoLISP”, a dataset composed of problem statements, solutions in their domain-specific language (DSL) and 10 tests. The solutions are inspired to the LISP programming language with constraints that simplify the automated generation and is presented in DSL form in order to facilitate the conversion into multiple programming languages. A program comprises a set of arguments

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

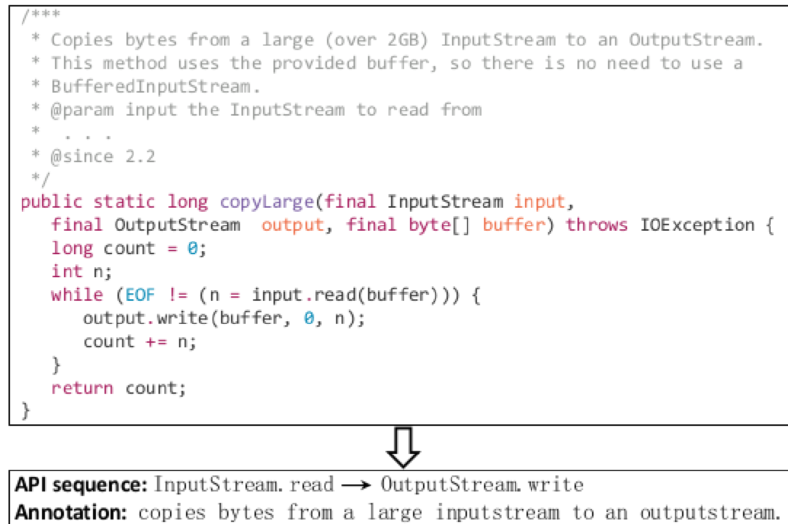


Figure 5.4.: An example of extracting API sequence and its annotation from a Java method [15]

(name and type) and a program tree where each node is argument, function call, function, or lambda. It also has a library of standard functions.
The 10 tests in the dataset are input-output pairs to be used to check the program.
In our case, we used only the problem statement and the solution without considering the tests. However, this dataset is limited in the size and types of problems included due to the expensive cost of collecting human annotation of programs so it is unlikely that the model trained on it would generalize to new types of algorithm. Moreover, we found that train, validation, and test set are very related.

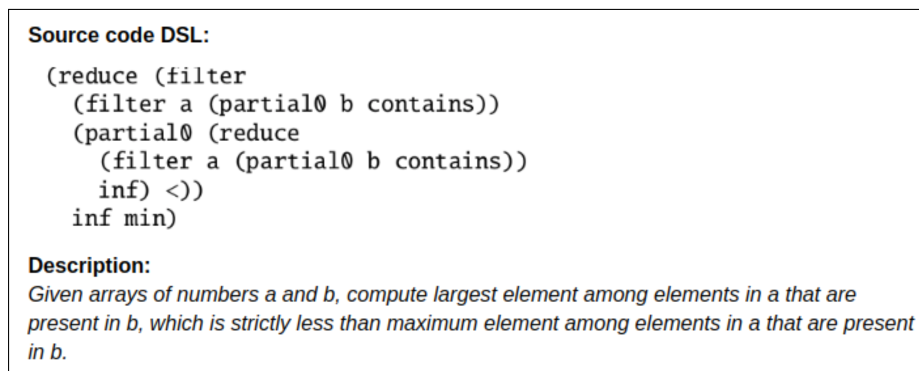


Figure 5.5.: Example of a solution in DSL and problem statement from the AlgoLISP dataset [39]

5.2. Language model Corpora

Our architecture includes a language model. To train it, we need an unsupervised corpus made of samples written in different programming languages involved in the tasks above, plus samples in English.
The corpus has to be of the same size or larger than the corresponding dataset for the specific task. For example, if we have the source code summarization dataset for Python with 14k samples, then the language corpora for Python should be at least the same size. A list of the different datasets we used is shown below:

- **English:** we use the "1 Billion Word Language Model" Benchmark[5] dataset that extract the corpus from the WMT11 website[6]. The sentences in the corpus are already tokenized and ready to use. The paper of Chelba et al. [8] describes how they created the dataset and some application of it to different language models.

- **Python:** we use the "150k Python Dataset"[7] from the "Secure, Reliable, and Intelligent Systems Lab" of the ETH of Zurich. The Python programs were collected from GitHub repositories by removing duplicate files and project forks, keeping only programs that parse and have at most 30000 nodes in the AST. For parsing the programs, they used the Python AST parser included in Python 2.7.

- **SQL:** we use a part of the "StaQC" dataset described in the paper of Yao et al. [53] available in their GitHub page[8]. This is a question-answering dataset but we only extract the code snippets written in SQL. The data was automatically mined from Stack Overflow with the framework they proposed.

- **Java and C#:** for these two languages we could not find a big enough corpus so we decided to use the source code coming from GitHub repositories thanks to a GitHub scraper as described below.

GitHub Scraper

In order to collect a big corpus of unsupervised data for Java and C#, we had to mine the GitHub platform. To do that, we used the Public Git Archive (PGA) tool that helps to list all the repositories with at least one Java or C# file and download them subsequently with the following command.

[5] http://www.statmt.org/lm-benchmark/
[6] http://statmt.org/wmt11/training-monolingual.tgz
[7] https://www.sri.inf.ethz.ch/py150
[8] https://github.com/LittleYUYU/StackOverflow-Question-Code-Dataset

```
$ pga list --lang java | pga get --lang java
```

The output files of this command are in the SIVA (Seekable Indexed Verifiable Archiver)[9] format. It is similar to tar or zip formats, but it is focused on allowing constant-time random file access, seekable access to the contained files and concatenable archive files. The files contain rooted repositories with all the git objects in order to reduce the duplicates. A rooted repository is a standard Git repository that stores all objects from all repositories that share common histories, identified by the same initial commit. To extract a GitHub repo from a siva file, we can use the command line interface:

```
$ siva unpack -v -o file.siva destination
```

Once we have the repository, we can extract the files with programs in the desired programming language and treat each of them as a sample of the corpus.

5.3. Pre-processing

The samples need to be preprocessed before feeding them to the models.
We first remove the comments from the programs because they were not part of the code, paying attention to the different way in which comments are specified in each language. The multi-task model deals with sequence data, so we process the programs to put them in that form simply by removing the newline (\n, \r\n, \n\r) characters. We also substitute characters, strings and numbers (integer, real, hexadecimal) with specific tokens in order to avoid the creation of a too big vocabulary. Finally, we had to tokenize each sample taking into account that it is a language-specific procedure. Thus, we had to treat each language, English included, with a specific tokenizer.

- **English:** we used the Natural Language Toolkit (NLTK) that is a platform for building Python programs that provides easy interfaces to manipulate human language data. In particular, we used the "tokenize[10]" package that provided a simple effective way to tokenize the sentences. All the tasks' datasets contain at least a part of the pairs in English, so this tokenizer was used in all of them. We did not apply it to the English corpus because it was already tokenized. Moreover, we consider commit messages, API sequences, and AST programs as written in English because we did not have a specific tokenizer for them.

[9] https://github.com/src-d/go-siva
[10] https://www.nltk.org/api/nltk.tokenize.html

- **Python:** we use the Python "tokenize[11]" library that contains functions to separate the tokens of a string and returns the value of the token and the type (number, string, ...).

- **Java:** we use the Python library called "javalang[12]" that provides a lexer and a parser targeting Java. The tokenizer can simply be invoked with: *"javalang.tokenizer.tokenize(s)"*.

- **SQL and C#:** for these two languages, we used the tokenizer from Iyer et al. [21] with few modifications. For SQL they used the Python library "sqlparse[13]" that provides support for parsing, splitting and formatting SQL statements. For C# they used the ANTLR (ANother Tool for Language Recognition) parser from Parr [37].

5.4. Statistics

We performed an analysis of the data obtained in order to later set the hyperparameters of the model. The unsupervised datasets contain a number of samples at least equal to the correspondent corpus in that language. A summary is in table 5.1 where we can see that the English corpus contains a huge number of samples, whereas the SQL one contains only 135.000 samples. In table 5.2 we can see the final number of samples for each of

Dataset	# samples
English: 1 Billion world corpus	300.000.000
150K Python Dataset	150.000
SQL corpus	133.000
Java from PGA	700.000
C# from PGA	500.000

Table 5.1.: Number of samples of each unsupervised dataset

the 7 tasks, divided into training and validation set. The division is the same applied by the authors of the respective papers. Here we can immediately see that the "API sequence recommendation" corpus has the biggest number of samples and the "source code summarization" for Python has the smallest.

Moreover, we analyzed the length of the samples, expressed in number of tokens, in the dataset to decide how to set the maximum length hyperparamter of the model. We chose 1024 because we identified this length as suitable for a good representation of the

[11] https://docs.python.org/2/library/tokenize.html
[12] https://github.com/c2nes/javalang
[13] https://github.com/andialbrecht/sqlparse

Task	Training	Validation
SCS Python	12.004	2.783
SCS C#	52.943	6.629
SCS SQL	25.671	3.340
Code comment generation	468.800	58.600
Commit messages generation	26.208	3.000
API sequence recommendation	7.475.850	10.000
Program learning and synthesis	79.214	10.819

Table 5.2.: Number of samples for each task

sample and a satisfying model accuracy. In figure 5.6 and 5.7, we visualize the distribution of the length of the input and target feature for each supervised dataset. There are some very long sequences for each dataset but they can be considered as outliers due to the small number of occurrences and they are not shown in the plot.
As we can see, when the sequences are treated as English sentences, the majority of occurrences are between 20 and 50 tokens. Only the task of ”program synthesis” has longer sequences but anyway they are below 200. This limited size of the English sentences is justified buy the size of a normal sentence in this language.
The programming inputs, in the tasks of ”code comment generation” and ”source code summarization”, are longer but at most they reach 600 tokens with a concentration around 100 tokens.

In figure 5.8 we can see the length of the samples in the unsupervised corpus. Like for the tasks' datasets, the English corpus has a very limited size, mostly limited to less than 60 tokens.
The other languages have different sizes: the SQL dataset has the shortest sequences with a maximum of 300 tokens, whereas the Java and Python have the longest ones with a maximum around 4000 tokens. However, the latter has the majority of occurrences below 1000 tokens.

Given all these findings, we decided to set the maximum sequence length hyperparameter for the multi-task model to 1024 in order to be good for both the tasks' datasets and the unsupervised dataset.

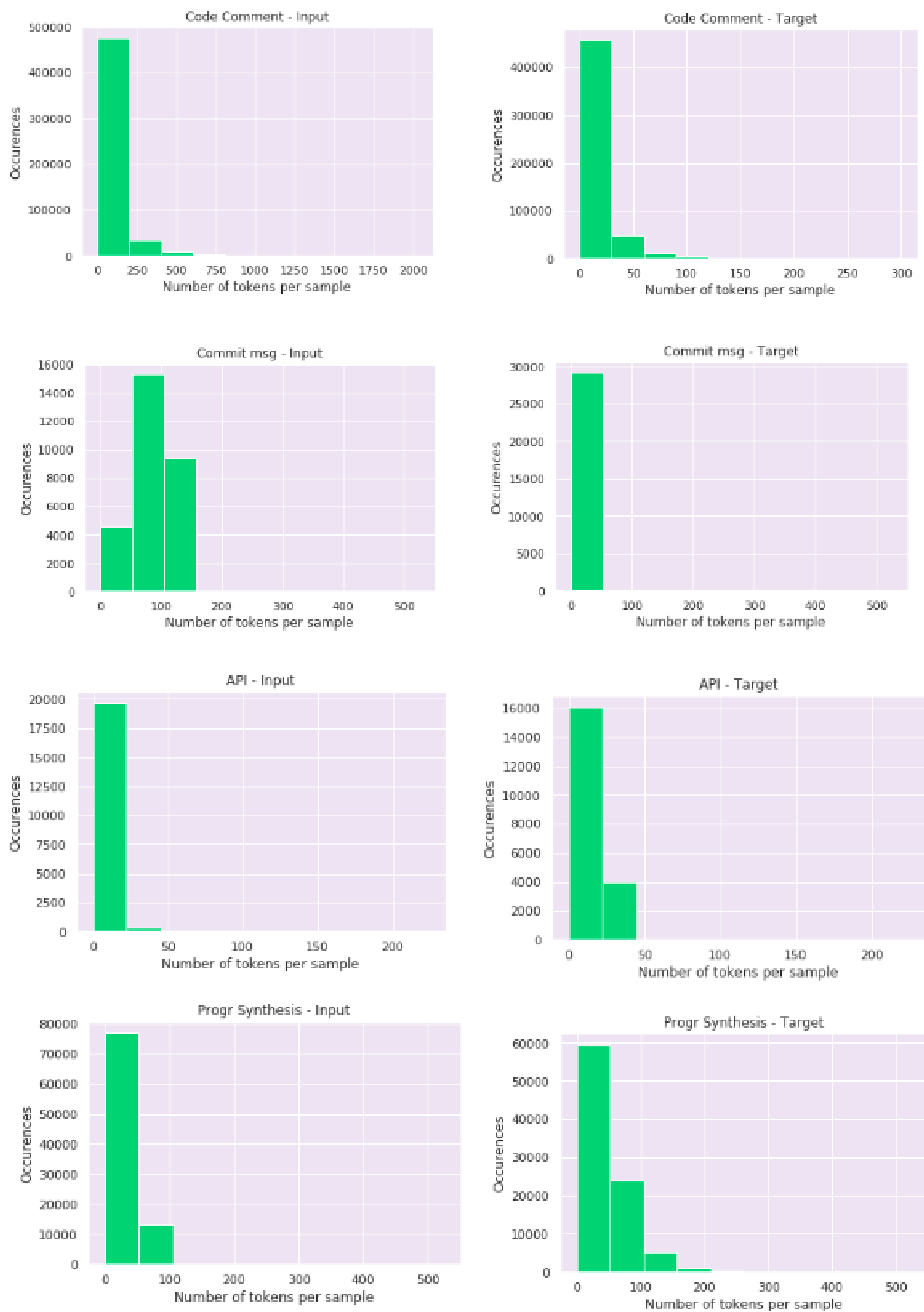


Figure 5.6.: Distribution of length of the tasks' corpus

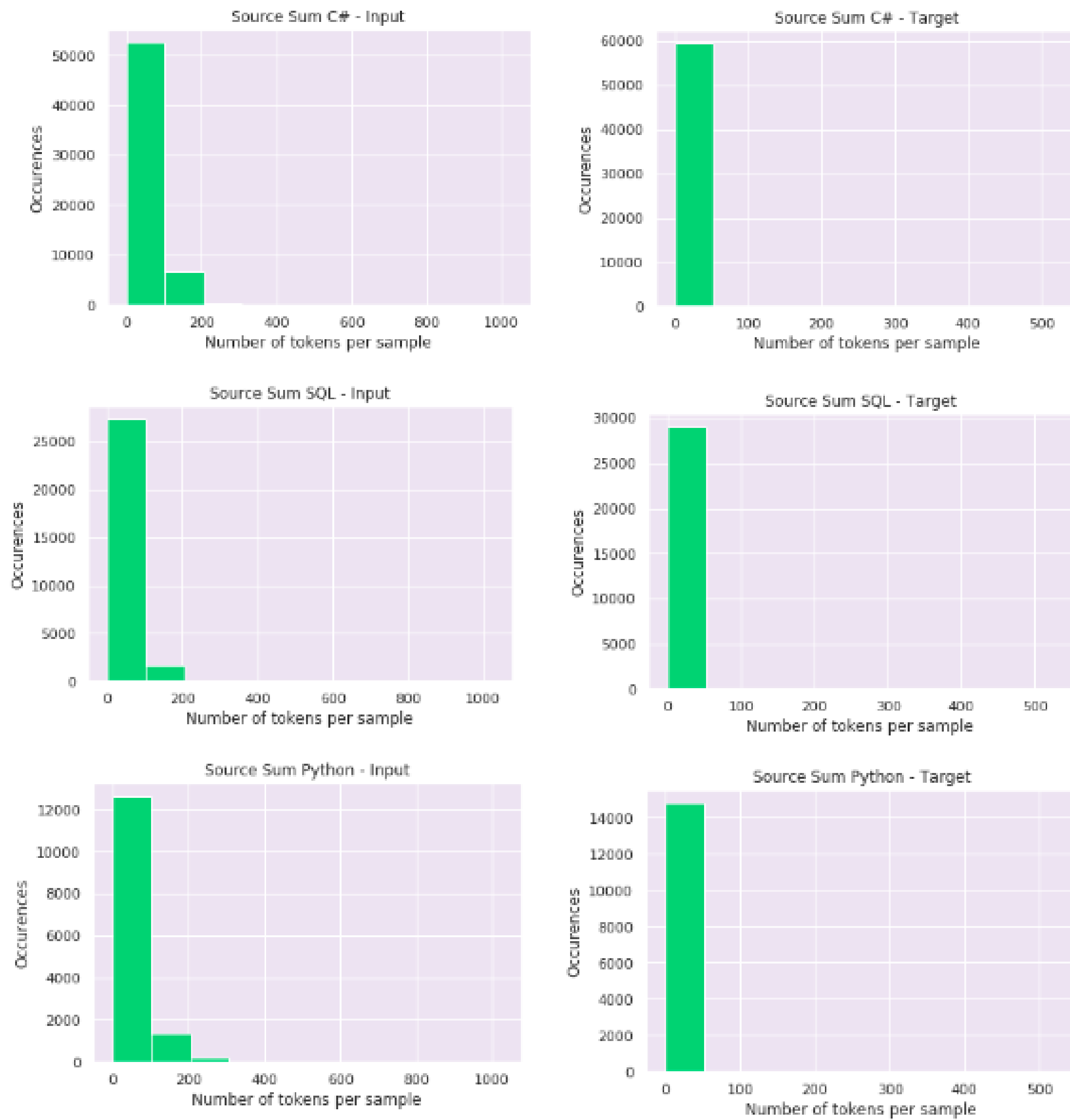


Figure 5.7.: Distribution of length of the tasks' corpus

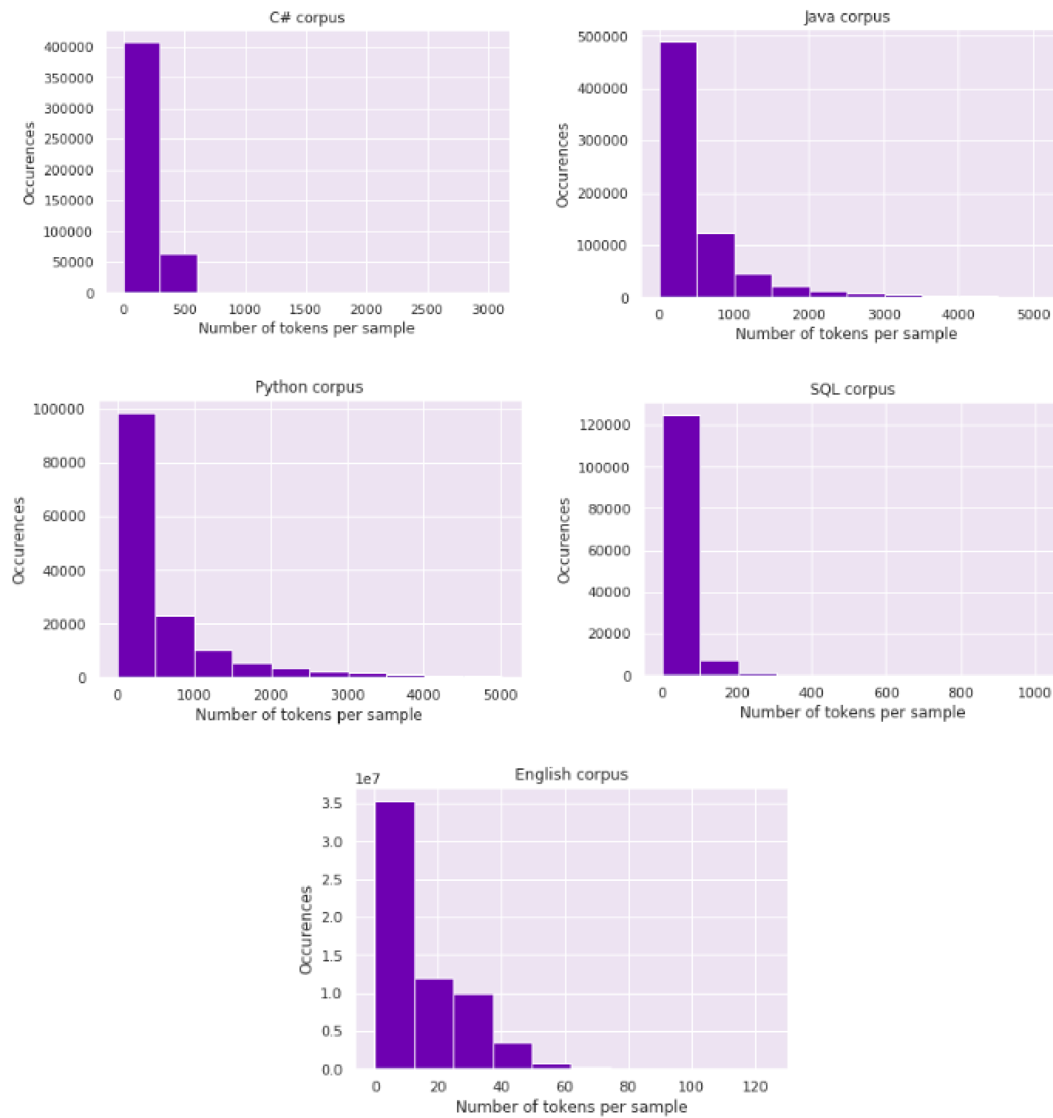


Figure 5.8.: Distribution of length of the corpus for the language model

6. Results and discussion

In this chapter we discuss the experiments conducted and the results achieved with single-task and multi-task learning models. For each model, we calculated the metrics shown in chapter 4. Some of them are not in the papers used for the comparisons, but we show them anyway for future research purpose. All the results are shown base on the metrics discussed in the previous chapter for future research purpose, even if the papers' counterparts used only some of them. A summary table with all the results can be found in appendix C.

6.1. Single-task model results

We performed experiments with single-task learning in order to understand how the transformer performs on each task separately. We use this results for comparisons with the multi-task model and with the original papers' results in the following sections.
We trained two models whose specifications are in Table 6.1, referred to as transformer base and transformer tiny. The second one is smaller than the first in the number of parameters which are set to almost a half of them in the base model. After some experiments, we notice that the tiny model has to perform a larger number of steps for the training in order to reach good performance. For these tasks, the steps are almost doubled with respect to the base model.

Model	Base	Tiny
Vocabulary size	8192	8192
Hidden size	512	128
Filter size	2048	512
Batch size	4096	256
Maximum sequence length	1024	1024
Number of parameters	$\sim$ 50M	$\sim$ 25M
GPUs used	1	1

Table 6.1.: Single-task models' setting

In Table 6.2 we report the results obtained with the two models and we show the number of training steps, the validation accuracy, the BLEU score, and the ROUGE scores

calculated after the decoding. For the ROUGE score, we report the F1 measure for type 1, 2 and L. In most of the cases, the base model outperforms the tiny model.

		SCS Py	SCS SQL	SCS C#	Code C	Commit	API	P synt
Training steps	**B**	5.3K	6K	20K	800K	30K	127K	177K
	T	12K	30K	60K	X	60K	300K	250K
Accuracy	**B**	33.4%	35.06%	41%	40%	56.2%	88.56%	99.85%
	T	33.74%	35.75%	39.88%	X	56.45%	81.1%	**99.87%**
BLEU score	**B**	1.11	1.14	2.06	4.47	**38.93**	**58.85**	99.02
	T	1.35	0.94	2.56	X	38.4	39.74	98.47
ROUGE-1	**B**	18.03	14.63	16.77	34.53	41.18	68.17	99.87
	T	16.94	12.86	18.91	X	41.15	50.60	99.72
ROUGE-2	**B**	3.43	2.25	3.81	15.62	29.65	58.20	99.36
	T	3.70	1.49	4.51	X	29.21	38.66	99.38
ROUGE-L	**B**	16.0	13.13	15.25	33.42	40.88	66.76	99.61
	T	15.05	11.89	17.18	X	40.16	49.44	99.54

Table 6.2.: Single-task models' results for base (B) and tiny (T) model in terms of training steps, accuracy, BLEU score and ROUGE scores calculated on the test sets

The training required approximately one day for each task and for each model on a machine with single GPU.
The models tend to overfit soon for all the tasks. In particular with the task of summarization where, in the case of the SQL language, we could only complete 6000 steps before the loss on the validation starts to increase.
For the task "code comments" we had to use half-precision with a small batch size to avoid out-of-memory problems. This implied an extremely slow training process and a substantial number of steps. Moreover, we did not use the Tiny model because, given that we had to use half precision, it would have been meaningless.

The results of the task "Program synthesis" is absolutely the highest with respect to the others, but we think that this is due to the similarity between the training, validation and test sets.
If we examine the BLEU score, we can see that the highest results are for the last three tasks as well as for the accuracy.
In picture 6.1 we show the three ROUGE score results in term of F1 measure for the transformer base model that outperforms the tiny model most of the times. We can clearly see that ROUGE-2, that refers to the overlap of bigrams between the output and reference summaries, is always lower than the others that instead are very similar. This implies that the output sentences have a substantial number of common words to the ground truth, but they are arranged differently.
All in all, the results of the ROUGE follow the trend of the BLEU with a higher score for

the last three tasks.
We can also see that the tasks with the lowest scores are the ones with as input a program and output an English sentence. The reason could be on the datasets' size of these tasks and not only on the fact that we have source code as input.

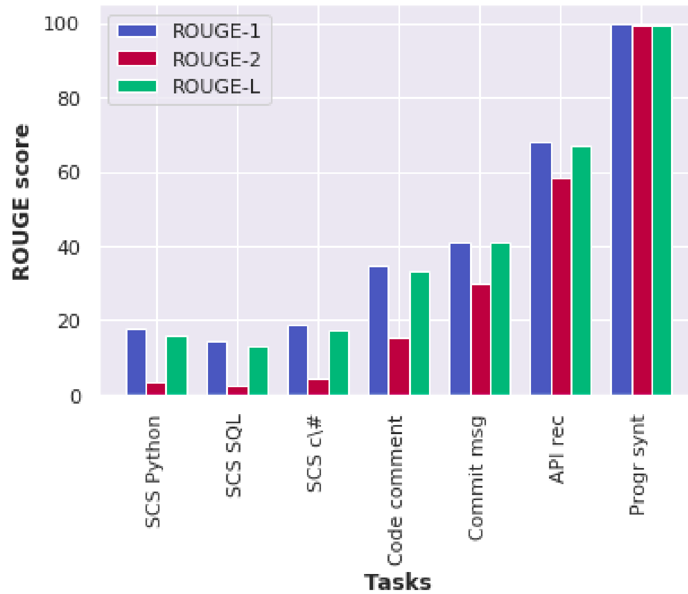


Figure 6.1.: Single-task model results on the test sets: F1 measure of ROUGE 1, 2 and L

6.2. Multi-task model results

In this project, we want to understand whether multi-task models achieve good performances on software development tasks. We trained the multi-task model with the configuration reported in Table 6.3.

Vocabulary size	64K
Hidden size	12
Filter size	8192
Batch size	1024
Maximum sequence length	1024
Number of parameters	360M
GPUs used	8

Table 6.3.: Multi-task model setting

To train it with an 8 GPUs machine, we had to work with half-precision fp16 format and the training process took approximately 13 days.

For the multi-task model, we used as a vocabulary the one obtained from the Language model data generation. It contains approximately 64k sub-words, where the most used are: *a, the, to, :, for, public, string*.

In Table 6.4 we can see the results achieved with this model trained for 630K steps. We stopped the training due to time reasons and to make other experiments like the one explained below (sec. 6.5), even if the model was still learning without any overfitting problem. The accuracy curves of the tasks with the multi-task model can be found in appendix B.

As for the single-task results, we can see that we have better performances for the last three tasks in terms of BLEU score and ROUGE, but not in terms of accuracy that is still low.

In terms of ROUGE score, the 1 and L settings have the highest scores meaning that the model is learning some words with respect to the target sentences, but not always in the right order. This is clear when we look at the ROUGE-2 that is lower.

	SCS Py	**SCS SQL**	**SCS C#**	**Code C**	**Commit**	**API**	**P synt**
Accuracy	41%	39%	39%	47%	55%	77%	94.8%
BLEU	1.57	3.31	4.29	6.32	30.74	26.51	71.02
ROUGE-1	21.60	19.9	22.36	35.74	45.13	41.29	83.30
ROUGE-2	5.27	3.37	4.52	16.37	25.40	22.21	71.44
ROUGE-L	18.87	17.89	20.31	34.44	45.08	40.54	80.66

Table 6.4.: Multi-task model results for 630K steps calculated on the test sets

6.3. Comparison between single and multi-task models

We trained both single-task models and multi-task models in order to understand which perform better for each task involved. We compare the results of each model shown in Table 6.2 and 6.4.

For the "source code summarization" and the "code comment" tasks, the multi-task model reaches a higher accuracy that we were not able to have with the single-task models. This happens because of the great advantage of the multi-task models of diminishing the overfitting possibilities and allow the learning to continue. As a result, we have a better BLEU score for the multi-task model.

For the other tasks, we have an accuracy lower than with the single-task model. Those tasks are made of large datasets and the learning is slower. This implies that they need to perform more steps in order to achieve higher accuracies, but we had to stop the training in advance due to time constraints. However, the accuracy for all the tasks was still increasing slowly whereas for single-task training the accuracy was decreasing due to

overfitting problems and this allows us to assume that the multi-task model would have performed even better with more training steps.

6.4. Comparison with state-of-the-art

We compare the results obtained with the single-task and multi-task model with the respective research papers' results. In Table 6.5 we report Accuracy and the BLEU score available in the original publications.

Task	Accuracy	BLEU
SCS Python	X	X
SCS SQL	X	17.0
SCS C#	X	20.04
Code comment generation	X	38.17
Commit messages generation	X	32.81
API sequence recommendation	X	54.42
Program learning and synthesis	85.8	X

Table 6.5.: State-of-the-art results available. The "X" means that the metric is not reported in the paper.

It is important to specify that the state-of-the-art BLEU score for the "source code summarization" and "code comment generation" tasks were calculated using a slightly different BLEU metrics. The script to calculate them is available on the GitHub page[1] referring to the paper of Iyer et al. [21]. For the following consideration on the affected tasks, we used the same script in order to have a fair comparison and the updated BLEU score are reported in Table 6.6.

	SCS Py	SCS SQL	SCS C#	Code Com
Single-task	7.75	9.99	10.8	15.52
Multi-task	8.61	**18.24**	11.01	16.36

Table 6.6.: Results with bleu.py script

A comparison between transformer base, tiny and state-of-the-art in terms of BLEU score results can be seen in picture 6.2. We observe that the base model outperforms the tiny in most of the tasks even if by small margins. For the task of "commit message generation" and "API sequence recommendation", the base model is able to beat the papers' counterparts with an increase of the BLEU score of 6.12 and 4.43 respectively. The

[1] https://github.com/sriniiyer/codenn

paper related to the task ”program synthesis” does not analyze the BLEU score so we compared it based on an accuracy assessment. In Table 6.2 we see that the accuracy for the transformer tiny model is 99.87% so, also in this case, our model outperformed the paper counterpart by more than 14%.

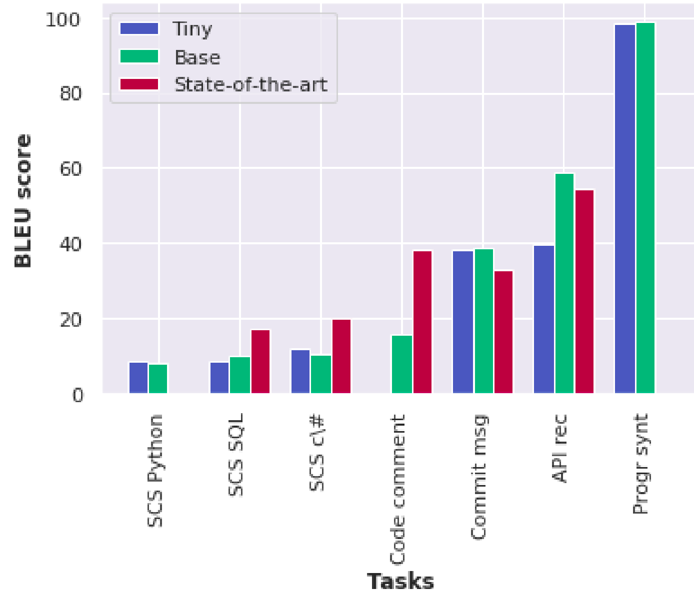


Figure 6.2.: BLEU score results with Single-task models in comparison with State-of-the-art. The BLEU score is not reported for the first and the last task because it is not available in the original papers.

We compare the results obtained with the multi-task model to the State-of-the-art papers' counterparts. In picture 6.3 we can see the BLEU score for Single-task, Multi-task and state-of-the-art.

For the ”source code summarization” tasks, our model beats the correspondent research paper by 1.24% for the SQL language whereas we did not improve it for C#.
The tasks of ”Commit message generation” and ”API sequence recommendation” were improved thanks to the single-task model while no improvement can be noticed on the multi-task. The reason could be found on the fact that we could not take the multi-task model to convergence and it remained with low accuracy, but we conjecture that it would have perform better with more training time.

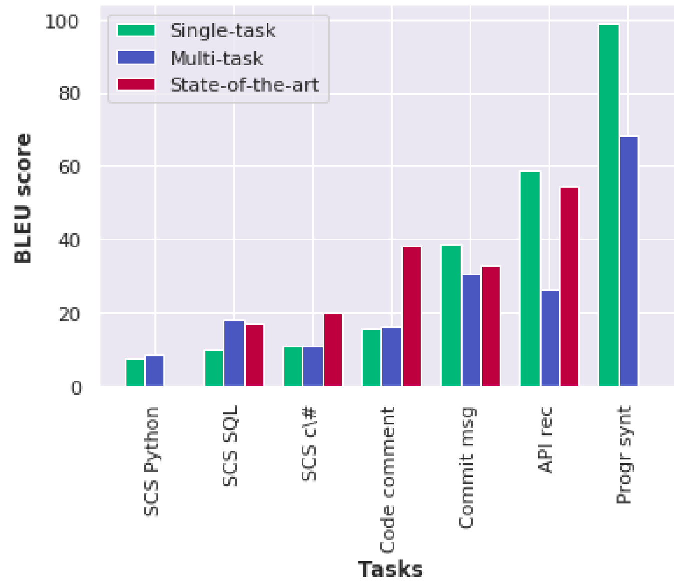


Figure 6.3.: BLEU score results for Single-task and Multi-task models in comparison with State-of-the-art. The BLEU score is not reported for the first and the last task because it is not available in the original papers.

6.5. Partial results

In this project, we deal with large sample sizes so we decided to increase the size of the model in order to better fit the input data. We started training a multi-task model with 62 million parameters more than the previous one. In order to train it with the same machine, we reduced the batch size by eight times. At the time of writing, training has already completed 650K steps.

Vocabulary size	64K
Hidden size	12
Filter size	10240
Batch size	128
Maximum sequence length	1024
Number of parameters	422M
GPUs used	8

Table 6.7.: Bigger Multi-task model setting

In picture 6.4 we can see the learning curve of the previous model and this bigger one for the task of ”API sequence generation”. Given that the batch size is eight time smaller than the previous model, we have to compare each step S of the green curve with the $\frac{S}{8}$ step of the red curve. In this way, we can see that the green curve, so the bigger model, is

giving an higher accuracy promising a better outcome with more training steps.

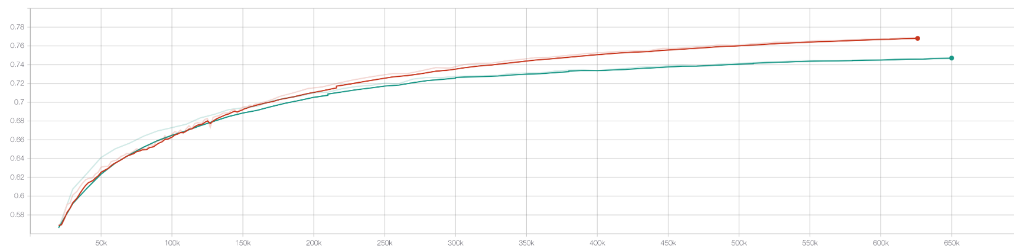


Figure 6.4.: Multi-task models' learning accuracy as function of number of steps. The red curve is the accuracy for the small multi-task model and the green curve is for the bigger model

7. Conclusions

In this thesis, we investigated the application of multi-task learning in the software development domain.
We trained single-task and multi-task models on seven supervised tasks plus a self supervised language model. The tasks are part of the software development domain and involve English and programming languages like Java, SQL, Python and C#.
We first compared the results for single-task and multi-task settings to understand which was more appropriate for each task. We observed that for the task of summarization of code, the multi-task model outperforms the single-task. This is mainly due to the possibility to train the multi-task model for more steps whereas for the single-task we were forced to stop the training early due to overfitting problems. For the other tasks, we can see that the single-task model has better performances but this is due to the fact that we did not train the multi-task model for enough steps to reach a good accuracy because of limited time and because we have preferred to make other experiments. However, the accuracy reached tell us that it is a promising model that could give important results with other 2 weeks of training.
Moreover, we evaluated the results based on the respective state-of-the-art papers' counterparts and we achieved better performances on 4 tasks out of 7.
We can conclude that the multi-task models are suitable for this domain if we have sufficient resources available (memory, GPUs, time, ...).
In this project, we also saw that training English and programming language sentences is possible and that it can give benefits if we take into account the structural information that are part of source codes. All in all, we will experiment more with the multi-task models in order to see which benefit they could give with an extensive training and a better exploration of hyperparameters.

8. Future Research

In this work, we began to study the Multi-task Deep Learning and its applications in the software development domain.
Further interesting experiments could be useful in order to better evaluate the practical impact of our approach.
First of all, we can explore more the hyperparameters configurations in both single-task and multi-task models. We will also dedicate some time on the completion of the training process for the bigger multi-task model described above.
In terms of multi-task learning, different combinations of tasks could be tried out. For example we could experiment with tasks whose input is English and target is a programming language or the other way around instead of mixing different languages. Moreover, we could invert the tasks switching the input with the target, and train all the seven plus seven tasks and the language model together. Finally, we could train subsets of tasks to understand which set gives improvements in performances and which, on the contrary, decreases them.
Another experiment could entail removing the language model from the multi-task training to understand how much benefit it can give.

Through out this project, we generated source code corpora, mainly for C# and Java languages, that could be further used by other researchers for tasks in the software development domain. We also trained a model that can be fine-tuned to help the transfer of knowledge and save energy for the researchers.
All in all, we intend to provide a starting point for future research in the field of software development domain and source code manipulation that include for example code suggestion, bug localization or automatic software repair tasks.
The results reported in this thesis and further experiments will be included on a paper that we planned to publish.

Appendix

A. Code sample

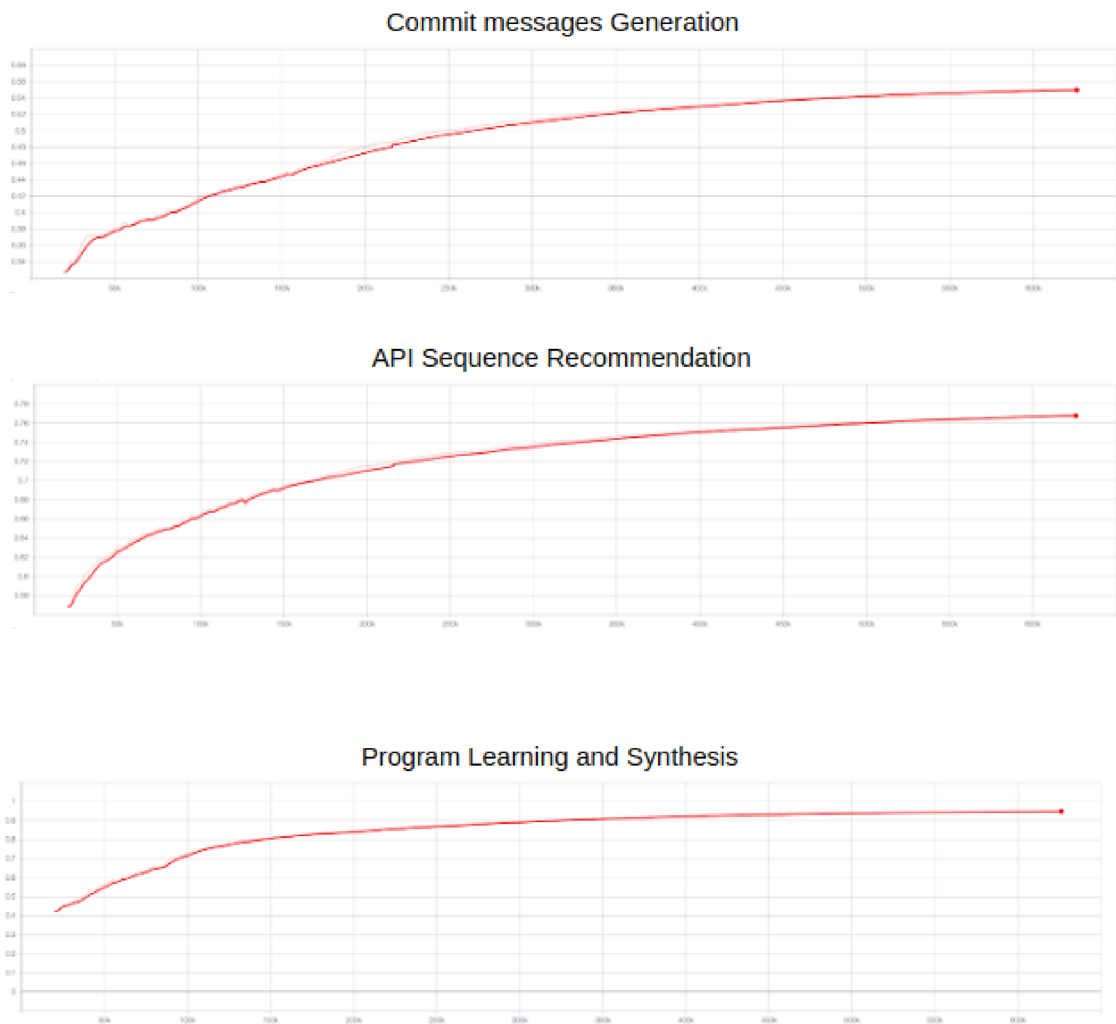
Example of data generator script for the ”Code comment generation” task. The first class is instantiated for the single-task models whereas the second class is for the multi-task models and uses the language model vocabulary.

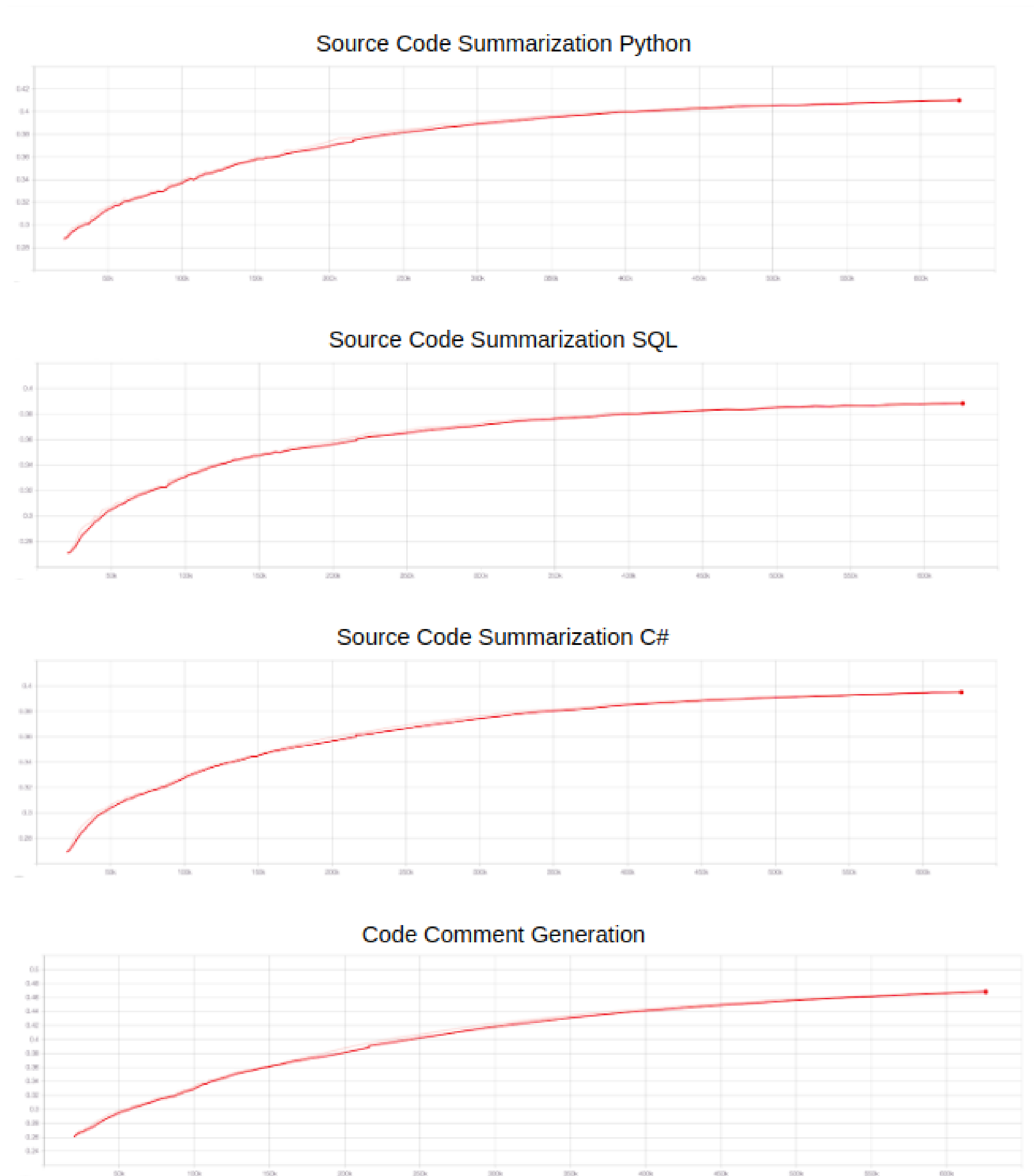
```
import langModel
import re
import os

from tensor2tensor.data_generators import problem
from tensor2tensor.data_generators import text_problems
from tensor2tensor.utils import registry
from tensor2tensor.utils import metrics

@registry.register_problem
class codeComment(text_problems.Text2TextProblem):

    @property
    def approx_vocab_size(self):
        return 2**13

    def is_generate_per_split(self):
        return True

    @property
    def dataset_splits(self):
        return [{
            "split": problem.DatasetSplit.TRAIN,
            "shards": 8,
        },{
            "split": problem.DatasetSplit.EVAL,
            "shards": 1,
        }]

    def generate_samples(self, data_dir, tmp_dir, dataset_split):

        train = dataset_split == problem.DatasetSplit.TRAIN
        dataset = "train" if train else "validation"
```

```
34
35          if dataset == "train":
36              with open('/home/ubuntu/tasks/trainCodeComment.txt') as f:
37                  for line in f:
38                      app = line.split("<sep_silvia>")
39                      try:
40                          yield{
41                              "inputs": app[0],
42                              "targets": app[1],
43                          }
44                      except:
45                          continue
46          else:
47              with open('/home/ubuntu/tasks/validCodeComment.txt') as f:
48                  for line in f:
49                      app = line.split("<sep_silvia>")
50                      try:
51                          yield{
52                              "inputs": app[0],
53                              "targets": app[1],
54                          }
55                      except:
56                          continue
57
58  @registry.register_problem
59  class codeComment_lm(codeComment):
60
61      @property
62      def use_vocab_from_other_problem(self):
63       # to use the vocab file of the language model
64       return langModel.LanguageModel()
65
66      @property
67      def vocab_filename(self):
68       return langModel.LanguageModel().vocab_filename
```

B. Multi-task accuracy plot

Accuracy curve of the tasks with the multi-task model as function of number of steps. The model was stop after 630K steps but all the accuracy curves could have increased with more steps performed.




Source Code Summarization Python
Source Code Summarization SQL
Source Code Summarization C#
Code Comment Generation

C. Results table

		SCS Py	SCS SQL	SCS C#	Code Com	Commit gen	API gen	Progr. synt
Training steps	Tiny	12K	30K	60K	X	60K	300K	250K
	Base	5.3K	6K	20K	800K	30K	127K	177K
	MT	630K	630K	630K	630K	630K	630K	630K
Accuracy	Tiny	33.74%	35.75%	39.88%	X	**56.45%**	81.1%	**99.87%**
	Base	33.4%	35.06%	**41%**	40%	56.2%	**88.56%**	99.85%
	MT	**41%**	**39%**	39%	**47%**	55%	77%	94.8%
	SOTA	X	X	X	X	X	X	85.8%
BLEU NLP score	Tiny	1.35	0.94	2.56	X	38.4	39.74	98.47
	Base	1.11	1.14	2.06	4.47	**38.93**	**58.85**	**99.02**
	MT	**1.57**	**3.31**	**4.29**	**6.32**	30.74	26.51	71.02
	SOTA	X	X	X	X	32.82	54.42	X
BLEU codenn	Base	7.75	9.99	10.8	15.52	X	X	X
	MT	**8.61**	**18.24**	11.01	16.36	X	X	X
	SOTA	X	17.0	**20.04**	**38.17**	X	X	X
ROUGE-1	Tiny	16.94	12.86	18.91	X	41.15	50.60	99.72
	Base	18.03	14.63	16.77	34.53	41.18	**68.17**	**99.87**
	MT	**21.60**	**19.9**	**22.36**	**35.74**	**45.13**	41.29	83.30
ROUGE-2	Tiny	3.70	1.49	4.51	X	29.21	38.66	**99.38**
	Base	3.43	2.25	3.81	15.62	**29.65**	**58.20**	99.36
	MT	**5.27**	**3.37**	**4.52**	**16.37**	25.40	22.21	71.44
ROUGE-L	Tiny	15.05	11.89	17.18	X	40.16	49.44	99.54
	Base	16.0	13.13	15.25	33.42	40.88	**66.76**	**99.61**
	MT	**18.87**	**17.89**	**20.31**	**34.44**	**45.08**	40.54	80.66

Table C.1.: Summary table for all the results obtained with single-task and multi-task models. In bold there are the best results for each task and for each score also compared to the state-of-the-art (SOTA).

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] David Alvarez-Melis and Tommi S Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. 2016.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Satanjeev Banerjee and Alon 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*, pages 65–72, 2005.

[5] Jonathan Baxter. A bayesian/information theoretic model of learning to learn via multiple task sampling. *Machine learning*, 28(1):7–39, 1997.

[6] Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*, 2015.

[7] Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 41–48. Morgan Kaufmann, 1993.

[8] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[10] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[11] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[12] Dipanjan Das and André FT Martins. A survey on automatic text summarization. *Literature Survey for the Language and Statistics II course at CMU*, 4(192-195):57, 2007.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[16] Poonam Gupta and Vishal Gupta. A survey of text question answering techniques. *International Journal of Computer Applications*, 53(4), 2012.

[17] Kazuma Hashimoto, Caiming Xiong, Yoshimasa Tsuruoka, and Richard Socher. A joint many-task model: Growing a neural network for multiple nlp tasks. *arXiv preprint arXiv:1611.01587*, 2016.

[18] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.

[19] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[20] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210. ACM, 2018.

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke 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)*, volume 1, pages 2073–2083, 2016.

[22] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146. IEEE Press, 2017.

[23] Lukasz Kaiser, Aidan N Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit. One model to learn them all. *arXiv preprint arXiv:1706.05137*, 2017.

[24] Tushar Khot, Ashish Sabharwal, and Peter Clark. Scitail: A textual entailment dataset from science question answering. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[25] Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. Unsupervised deep bug report summarization. In *Proceedings of the 26th Conference on Program Comprehension*, pages 144–155. ACM, 2018.

[26] Xiaochen Li, He Jiang, Zhilei Ren, Ge Li, and Jingxuan Zhang. Deep learning in software engineering. *arXiv preprint arXiv:1805.04825*, 2018.

[27] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[28] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.

[29] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. *arXiv preprint arXiv:1901.11504*, 2019.

[30] Minh-Thang Luong, Quoc V Le, Ilya Sutskever, Oriol Vinyals, and Lukasz Kaiser. Multi-task sequence to sequence learning. *arXiv preprint arXiv:1511.06114*, 2015.

[31] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to information retrieval. *Natural Language Engineering*, 16(1):100–103, 2010.

[32] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, and Lu Zhang. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358*, 2014.

[33] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 689–696, 2011.

[34] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[35] Bo Pang, Lillian Lee, et al. Opinion mining and sentiment analysis. *Foundations and Trends® in Information Retrieval*, 2(1–2):1–135, 2008.

[36] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. pages 311–318. ACL, 2002.

[37] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[38] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[39] Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

[40] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. pages 357–367. IEEE, 2016.

[41] Bharath Ramsundar, Steven Kearnes, Patrick Riley, Dale Webster, David Konerding, and Vijay Pande. Massively multitask networks for drug discovery. *arXiv preprint arXiv:1502.02072*, 2015.

[42] Sebastian Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, 2017.

[43] Victor Sanh, Thomas Wolf, and Sebastian Ruder. A hierarchical multi-task approach for learning embeddings from semantic tasks. *arXiv preprint arXiv:1811.06031*, 2018.

[44] Michael L Seltzer and Jasha Droppo. Multi-task learning in deep neural networks for improved phoneme recognition. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6965–6969. IEEE, 2013.

[45] Anders Søgaard and Yoav Goldberg. Deep multi-task learning with low level tasks supervised at lower layers. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 231–235, 2016.

[46] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[48] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, et al. Tensor2tensor for neural machine translation. *arXiv preprint arXiv:1803.07416*, 2018.

[49] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[50] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):9, 2016.

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

[52] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[53] Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pages 1693–1703. International World Wide Web Conferences Steering Committee, 2018.

[54] Yu Zhang and Qiang Yang. A survey on multi-task learning. *arXiv preprint arXiv:1707.08114*, 2017.

[55] Zhanpeng Zhang, Ping Luo, Chen Change Loy, and Xiaoou Tang. Facial landmark detection by deep multi-task learning. In *European conference on computer vision*, pages 94–108. Springer, 2014.

[56] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja 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*, pages 19–27, 2015.