

TITLE

...

...

Carmen Armenti

October 2022

Supervised by
Prof. Rocco Oliveto

Co-Supervised by

Contents

1	Introduction	1
1.1	Application context	2
1.2	Motivations and Objectives	2
1.3	Results	3
1.4	Document Structure	3
2	Deep Learning and Curriculum Learning	5
2.1	State of the art	5
2.2	Curriculum Learning related works	6
2.3	Curriculum Learning application in Deep Learning tasks	6
3	Deep Learning Applications to Software Engineering tasks	7
3.1	State of the art	7
3.2	Software Engineering related works	7
3.2.1	Bug fixing task	7
3.2.2	Code summarization task	7
3.2.3	Log generation task	7
3.3	7
4	Evaluating Curriculum Learning in Software Engineering tasks	9
4.1	Neural Machine Translation	11
4.2	Canonical Training	12
4.2.1	Bug-fixing	12
4.2.2	Code summarization	12
4.2.3	Log generation	13
4.3	Training with Curriculum Learning	13
4.3.1	Bug fixing task	13
4.3.2	Code summarization task	14
4.3.3	Log generation task	15
5	Analysis of results	17
5.1	Bug fixing task	17
5.2	Code summarization task	18

5.3 Log generation task	19
6 Conclusion	21
A Appendix	23

List of Figures

4.1	Predefined Curriculum Design.	9
4.2	Code summarization task: data distribution.	15

List of Tables

5.1	Models' performances	17
5.2	Overlapping metrics: baseline and curriculum learning models.	18
5.3	BLEU values summary	18
5.4	Perfect predictions summary.	19

Chapter 1

Introduction

Humans and animals acquire an extensive and flexible repertoire of complex behaviors through learning. They can perform much more complex tasks than they can acquire using simple trial and error learning. This gap is filled by teaching indeed, and in instruction contexts one important method is *shaping*. Shaping starts with a task analysis in which a desired behavior is broken down into smaller and more manageable steps that would move the learner, namely the child, successively closer to that desired behavior. Once the small approximations of the desired behavior are clearly identified, one must select the reinforcement to be used and make sure that everyone working with the tutee knows which behavior, when, and how to reinforce the approximations. Data on the behavior should be collected and reviewed by the team. The program must continue until the learner demonstrate the desired behavior. In order for shaping to be successful, it is important to clearly define the behavioral objective and the target behavior. Also, in order to gradually achieve the target, a teacher must know when to deliver or withhold reinforcement.

The term *shaping* was first coined by B. F. Skinner¹, who described it as a "method of successive approximations". In shaping indeed, a sequence of intermediate, simple tasks is taught, in order to aid acquisition of an original, complex task. Approximately shaping shares the same idea of the divide-and-conquer concept.

Even though all the previous said looks to be more appropriate to behavior engineering² - being it related to the attempts that experts of the field put into practice to change human or animal behaviors - shaping was related to the context of machine learning.

¹Skinner is an american psychologist, behaviorist, author, inventor, and social philosopher.

²Behavioral engineering, also called Applied behavioral analysis (ABA) is a sceintific discipline that applies empirical approaches based upon the principles of respondend and operant conditioning to change behavior of social significance.

1.1 Application context

Bengio *et al* [1] in their work "Curriculum Learning" are the first to relate the shaping concept to the context of machine learning theory, calling the training strategy of organizing the training in a meaningful order "curriculum learning" indeed. Curriculum learning was originally inspired by the learning experience of humans, since human beings tend to learn better and faster when they are first introduced to simpler concepts and exploit previously learned concepts and skills to ease the learning of new abstractions. However, the basic idea of training a learning machine with a curriculum method can be traced back at the very least to Elman [3]. Elman realized a concept described in terms of *less is more*, in the context of the learning of grammars in simple recurrent networks. The idea was to use an initial phase of training with only simpler incarnations of the rules of the grammar. The experimental results, based on learning a simple grammar with a recurrent network, suggested that successful learning of grammatical structure depends, rather than on innate knowledge of grammar, on starting with a limited architecture that is at first quite restricted in complexity, but afterward expands its resources gradually as it learns. Such conclusions are important for developmental psychology, because they illustrate the adaptive value of starting, as human infants do, with a simpler initial state, and then building on that to develop more and more sophisticated representations of structure. The question of guiding learning of a recurrent neural network for learning a simple language and increasing its capacity along the way was revisited by Krueger & Dayan [5], who used an unelaborated form of the long short term memory (LSTM) model, studying the additional role that shaping might play in generating complex behavior in tasks, and providing evidence for faster convergence using a shaping-like procedure. Similar ideas were also explored in robotics, where by gradually making the learning task more difficult reinforcement learning was applied.

1.2 Motivations and Objectives

Whether machine learning algorithms benefit from a similar training strategy is a question that has been widely answered so far, every time differently depending on the task experimented and on the results aimed. Without surprise, curriculum learning has been found most helpful in end-to-end neural network architectures [1], given that the performance that an artificial network can achieve mostly depends on the quality of training data given to it.

Recently, the application of curriculum learning is also studied for Neural Machine Translation (NMT), that translates text from a source language to a target language in an end-to-end fashion with a single neural network. On top of that, the performance of NMT has been improved significantly in recent years, as these architectures evolved from the initial Recurrent Neural Network (RNN) based models, to convolutional seq2seq models and further to Transformer

models.

Since many clarified when and why a *curriculum* or *starting small* strategy can benefit machine learning algorithms, and given that none of the previous work in the current state of the art involved software engineering tasks, we decided to bridge this gap focusing our attention on three software engineering tasks: **bug-fixing**, **code summarization**, and **log generation**.

Not only are curriculum learning applications of benefit of improving model performances on target tasks - given by the metaphor with the effective human-based learning approach, but also of accelerating the training process. However, since we worked with a NMT and training an NMT model is a time-consuming task *a priori*, we were curious to see results in terms of the first of the two most significant requirements in major machine learning research aforementioned. On the other hand we were not sure about the time needed for the training.

1.3 Results

By and large, the CL training times for the bug-fixing and code summarization tasks were not that much different from the plain-training process times without curricula; quite the opposite, as for the log generation task CL training took longer.

In terms of performances, bug-fixing task is the only one who showed a significant improvement; code summarization task experiment instead reported quite similar results, as for the comparison between plain and curriculum training.

1.4 Document Structure

This thesis is organized as follows:

- Chapter 2 - Deep Learning and Curriculum Learning: related works in the current state of the art;
- Chapter 3 - Deep Learning and Software Engineering tasks: theoretical background and other works;
- Chapter 4 - Evaluating CL in Software Engineering tasks: definition and development of CL approach considered;
- Chapter 5 - Analysis of results: assessment for each of the task of the CL approach used;
- Chapter 6 - Conclusion: summary and possible future developments.

Chapter 2

Deep Learning and Curriculum Learning

Humans are different from other species in many ways, but two of them are particularly noteworthy. First of all, humans display an exceptional capacity to learn; moreover, humans are remarkable for the long time they take to reach maturity. Human beings need about two decades to be trained as fully functional adults for our society and it may be argued that, through culture, learning has created the basis for a non-genetically based transmission of behaviors and habits which might accelerate the evolution of our species. Infancy and childhood are times of great vulnerability for the young, when the first skills are developed and when the adults who must care for and protect their own young go through a severe restriction of their range of activities. Then, why would evolutionary process not prune a long period of immaturity from our species? Previous research carried out by Elman *et al.*[3] at the intersection of cognitive science and machine learning underline that it is important to remember that the evolution looks at the whole individuals rather than at the value of isolated traits; the adaptive success of individuals therefore is determined by the joint interaction of all their traits. Thus, it might be that to understand the persistence of one trait with apparently negative consequences - such as the lengthy period of immaturity, possible interactions with other traits - such as the ability to learn - may need to be considered. The perfect example of such an interaction, is that in human beings the greatest learning occurs precisely in the period of time when they are undergoing major changes, during childhood indeed.

2.1 State of the art

In humans, learning and development interact in an as important as non-obvious way; maturational changes might provide the enabling conditions which allow learning to be most effective. It is a matter of fact that humans need about two decades to be trained as fully functional adults, and the higher the training is organized, the better the knowledge of the person. Humans in fact learn much better when the examples are not randomly presented, but organized

in such a way where the amount of concepts is incremented gradually, and where the complexity of them increases over time. The majority of education systems are organized in order to illustrate different concepts at different times, exploiting previously learned notions to ease the learning of new abstractions. By choosing which examples to present and in which order to explain them to the learning system, one can guide training and consequently increase the speed at which learning can happen. This is the same idea exploited in *animal training* by Skinner (1958), Peterson (2004) and Krueger & Dayan (2009)¹ where it is called *shaping*.

In the context of machine learning, such a meaningful learning-like process is known as *curriculum learning*, whose basic idea - traced back to Elman - is to start small, learn easier aspects of the task, and then gradually increase the difficulty level. Inspired by human learning in fact, curriculum learning (CL) is a training strategy that emphasizes the order of training instances in a computational learning setup. As a feature of human learning, curriculum - or even better learning in a meaningful way - has been transferred to machine learning, thus creating the subdiscipline named *curriculum learning*.

Essentially, human education is organized as curricula, by starting small indeed, and gradually presenting more complex concepts. The paramount hypothesis is that simpler instances should be learned during the first steps as building blocks, to then learn more complex ones. Several experiments on sentiment analysis task and tasks similar to sequence prediction tasks in NLP carried on by Cirick *et al* [2] prove that curriculum learning has positive effects on LSTM's internal states, by biasing the model through building constructive representations. Specifically, the internal representation at the previous timestep is used as building block for the next one, thus contributing at the final prediction.

In traditional machine learning algorithms all the training examples are randomly presented to the model, thus ignoring the different complexities of data instances and the learning status of the current model. Owing to this, it is fairly intuitive wondering if the curriculum training strategy could ever benefit machine learning. Extensive experiments from early [], [], [] to recent works [], [], [], [] in various applications of machine learning show that such strategy is of benefit in this field, but not always, and because of that the power of introducing the curriculum-like strategy depends on how the curriculum for specific applications and datasets is designed.

2.2 Curriculum Learning related works

2.3 Curriculum Learning application in Deep Learning tasks

¹hello

Chapter 3

Deep Learning Applications to Software Engineering tasks

3.1 State of the art

3.2 Software Engineering related works

3.2.1 Bug fixing task

Each instance of the dataset is a pair (m_b, m_f) , where m_b is a buggy code component and m_f is the corresponding fixed code. These BFPs were used to train the NMT model, allowing it to learn the translation from the buggy to the fixed method, thus being able to generate fixing patches.

3.2.2 Code summarization task

Automatic source code summarization is the task of generating short natural language description for source code [6]. The idea is that a brief description allows programmers to understand what a chunk of code does and what is the purpose of the program by and large, without necessarily read the code itself.

3.2.3 Log generation task

3.3

Chapter 4

Evaluating Curriculum Learning in Software Engineering tasks

In a nutshell, curriculum learning means "training from easier data to harder data" [10]. More specifically the core idea is to "start small" [4], train the machine learning model with easier subtasks, to then gradually increase the difficulty level of subtasks until the whole training dataset is used.

Bearing in mind the strategy of training from easier to harder data, to design such a curriculum idea (i), what kind of training data is supposed to be easier than other data, (ii) and when is appropriate to present more harder data for training - and how much more - must necessarily be decided. Technically, those 2 issues can be abstracted in the concepts of a Difficulty Measurer, that decides the "easiness" of each data instance to start the training process from, and a Training Scheduler, that rules the sequence of data subsets during the whole training process [10]. Therefore, Difficulty Measurer together with Training Scheduler constitute a general framework for curriculum design, as illustrated in Figure 4.1. First of all, the Difficulty

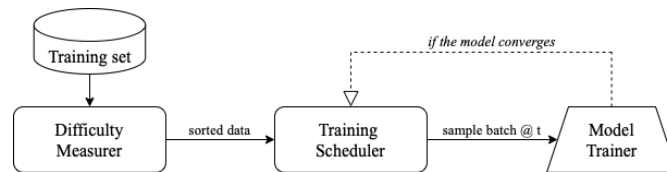


FIGURE 4.1: Predefined Curriculum Design.

Measurer sorts all the training examples from the easiest to the hardest and passes them to the Training Scheduler. Then, at each training epoch t , the Training Scheduler samples a batch of training instances from the easier subset and gives them to the Model Trainer for training.

As training epochs increase, the Scheduler decide when to sample from more harder data, generally until uniform sampling from the whole training set. This schedule either depends on the training loss feedback from the Model Trainer, or on some other parameters that implies that the model would deverge, if left to training for more epochs.

A distinction between **predefined CL** and **automatic CL** must be clarified. The first refers to the

framework where both the Difficulty Measurer and Training Scheduler are defined by human prior knowledge, thus with no data-driven algorithms involved; the latter instead, if any - or both - of the components are designed by data-driven algorithms.

Usually, the power of introducing Curriculum into Machine Learning depends on how the curriculum for specific applications and dataset is designed. Due to this, Difficulty Measurers often rely on the data characteristics of specific tasks, and most of them are defined by complexity, diversity, or noise estimation definitions.

We adopted the most popular discrete scheduler, known as *Baby Step*, where the complexity of the training data needs to be gradually increasing. Due to this, a reliable metric to split the initial dataset of each task needed to be defined. This approach distributes the sorted data into buckets, from easy to hard according to the metric, and starts training with the easiest bucket. Starting from the easiest bucket, after a fixed number of training epochs or convergence, the subsequent bucket is merged into the current training subset - main characteristic of *Baby Step* approach. Finally, once all the buckets are merged and used, the training process either stops or continues several extra epochs.

Note that, at each epoch the scheduler shuffles the current bucket and samples mini-batches for training.

However, before testing curriculum learning approach, we thought it was strictly necessary reproducing - in the case of bug-fixing task - or conducting - within the framework of the other 2 tasks - the experiment on the modelbase.

We worked with a neural network whose configuration is composed by 1-layer bidirectional Encoder, 2-layer Attention Decoder both with 256 units, embedding size of 512, and LSTM RNN cells.

In this chapter, we present how we applied Curriculum Learning to some of Software Engineering tasks, i.e. *bug-fixing*, *code summarization*, and *log generation* tasks.

For each of the tasks considered, the following aspects are described thoroughly:

- **Difficulty Measurer:** a measure to sort the datasets' instances is needed, each task was experimented with a different metric;
- **Training Scheduler:** sequence of data subsets are presented to the model following a training schedule.

The goal of this study is to assess whether Neural Machine Translation, combined with the Curriculum Learning approach, can be used for the tasks experimented. In the following section, the design of our study is described in detail.

4.1 Neural Machine Translation

The experimented models are based on an Recurrent Neural Network (RNN) Encoder-Decoder architecture with attention mechanism, frequently used in Neural Machine Translation. This kind of model is composed by two dominant components:

- a RNN Encoder, which encodes a sequence of terms \mathbf{x} into a vector representation;
- a RNN Decoder, which decodes the vector representation into another sequence of terms \mathbf{y} .

The model learning is based on a conditional distribution, where the output sequence of terms is conditioned by the input sequence: $P(y_1, \dots, y_m | x_1, \dots, x_n)$, where m and n not necessarily have to have the same length. The Encoder takes as input a sequence $\mathbf{x} = (x_1, \dots, x_n)$ and produces a sequence of states $\mathbf{h} = (h_1, \dots, h_n)$. The framework relies on a bi-directional RNN Encoder, which is composed by a backward and a forward RNN, where both are able to create representations taking into account past and future inputs. Specifically, each state h_i is the concatenation of the states produced by the two RNNs when reading the sequence not only in a forward but also in a backward manner.

The RNN Decoder computes the probability of a target sequence $\mathbf{y} = (y_1, \dots, y_n)$ given \mathbf{h} . The probability of each output term h_i is computed based on:

- the recurrent state s_i in the Decoder;
- the previous $i - 1$ terms (y_1, \dots, y_{i-1}) ;
- a context vector c_i , which constitutes the attention mechanism.

The vector c_i is a weighted average of the states in \mathbf{h} , where the weights associated to each state allow the model to pay more attention to some parts of the input sequence than to others:

$$c_i = \sum_{t=1}^n a_{it} h_t$$

Precisely, the weight a_{it} defines how much the model should take into consideration the term of the sequence in input x_i when predicting the target term y_t . Encoder and Decoder are simultaneously trained - instead of sequentially - by minimizing the negative log likelihood of the target terms, using stochastic gradient descent. The configuration used by the neural network is composed by 1-layer bidirectional Encoder, 2-layer Attention Decoder both with 256 units, embedding size of 512, and LSTM RNN cells. Bucketing and padding was used to deal with the variable length of the sequences.

4.2 Canonical Training

Before experiment the curriculum learning approach, we reproduced the baseline approaches for each task experimented that include the conventional technique of randomly ordering the training samples with the aim to investigate how the performance of the original models are, compared to the experiment affected by curriculum learning.

4.2.1 Bug-fixing

As for the bug-fixing task, the datasets used to train the NMT model is the union between *small* and *medium* method-level datasets used by Tufano *et al.* [9]. So, we worked with the standard training, evaluation and test set splits of respectively 99.044, 12.381, and 12.380 instances.

It is vital to use new data when evaluating a model to prevent the likelihood of overfitting to the training set. However, we decided to evaluate our model as we were building it to find the best parameters of a model. To evaluate the model while still building and tuning the model, we used the evaluation set. The neural network was set up to evaluate the model every 1.000 steps. The training was performed for 60k steps as upper bound, out of which we considered the best model, early stopping the model before overfitting considering the loss values computed every 1.000 steps. The best model configuration was then used to run the inference. Indeed, after the model was trained, it was evaluated on the test set of unseen buggy code. Instead of using the classic greedy decoding that selects the output term y_i with the highest probability, however, we used another decoding strategy known as Beam Search. The key idea is that the decoding process keeps track of k hypotheses - being k the beam size or width.

The beam search algorithm selects multiple tokens for a position in a given sequence based on conditional probability. Moreover, the algorithm can take any number of N best alternatives through a hyperparameter known as beam size or width indeed. Conversely to what greedy search does, Beam search broad the search to include other words that might fit better apart from the best word for each position in the sequence. If Greedy search looks at each position in the output sequence in isolation, deciding the word based on highest probability and then moving down to the resto of the sentence, Beam search also takes the N best output sequences and look at the current preceding words and probabilities compared to the current position that is being decoded in the sequence.

We considered the following sizes: 1 - that corresponds to the Greedy Naïve Approach -, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50.

4.2.2 Code summarization

Regarding the task of code summarization, we selected a random sample of 200k instances from a dataset of 2.1 million items. Also here we considered the canonical division between training, test and evaluation of respectively 200.000, 105.832, and 106.153 instances. We trained

the model on the training set for 20 epochs, that corresponds to 125.000 steps, and we took the best model over the 20 model configuration evaluated after every epoch, thus using the best checkpoint to run the inference step. This time, to guide the selection of the best configuration, we used the BLEU values computed on the validation set instead of using the loss function. Conversely, the results are computed on the test set. Indeed, the same way it was done for the previous task, after the training step the model is evaluated over the test set of unseen functions to generate code summarizations of such methods.

The scope of the training here was to make the model able to learn how to summarize pieces of code, starting from a sequence of tokenized methods.

4.2.3 Log generation

As well as happened in both the previous task experimented, also in log generation we considered the well-known split between training set, test set, and evaluation set. The sets in question are - following the same order - of the amount of 106.382, 12.020, and 13.260. The model was trained for 45 epochs, i.e. 149.625 steps, and it must be specified that it was evaluated every epoch. The best configuration in this case was guided on an early-stopping criterion based on BLEU values. Once the model was trained, it was evaluated against the test set of unseen methods without log statements and messages in order to assess its ability to correctly inject log statements and messages within methods.

4.3 Training with Curriculum Learning

As stated in the Curriculum Learning literature, the training instances need to be ordered based on the curriculum. Each of the task considered is related to a different curriculum, specifically the datasets were ordered following 3 different complexity ideas. Since a curriculum approach requires the definition of a **Difficulty Measurer** and a **Training Scheduler**, we defined those for each task. In the following sections details are explained.

4.3.1 Bug fixing task

Based on the entire training dataset at our dispose of couple of buggy-fixed methods, we defined a Difficulty Measurer to divide the dataset and a Training Scheduler to rule the order in which the instances were given to the model. Since both the source and the target were tokenized, we thought that computing the number of changes from the buggy version to the fixed one of each method was a quite reliable way to define the complexity for the type of instances used in this task.

Difficulty Measurer. As stated above, a difficulty measurer definition is the core element of the approach implemented. For the task at issue here the **Levenshtain distance** was used as metric. The *Levenshtein distance*, also known as the *edit distance*, was introduced by Vladimir Levenshtein in 1965 [7]. It is a string metric for measuring difference between two sequences, specifically the number of insertions, deletions, and substitutions required to transform one string into the other. However, the distance we used for our experiments was token based, thus the granularity is word based instead of being single-character based.

Defined the measure, the distance between buggy and fixed method for each of the BFPs was computed and we observed the data distribution; then we used the quartiles' values to divide the initial dataset in multiples smaller datasets, where each of these represents a different level of difficulty.

By doing so, we clearly obtained 4 level of difficulty; thus we defined an incremental difficulty criteria to feed the model with: the bug-fixing training scheduler described as follows.

Training Scheduler. The training scheduler decides the sequence of data subsets throughout the training process based on the judgment from the difficulty measurer. As mentioned in the introduction of this chapter, Baby Step approach switches to the next difficulty level as soon as the model converges on the previous one. In the bug-fixing task, the scheduler adjusts the training data subsets based on an early stopping criterion: after a fixed number of steps early stopping is run and it considers the model as being converging if the loss value does not improve after 5K steps.

4.3.2 Code summarization task

As well as happened for the training on the baseline model, a random sample of 200K method-comment pairs was picked from the initial dataset; evaluation and test datasets, however, were used as-is. This initial dataset was then divided in 4 sub-datasets, each representing a different level of difficulty.

Difficulty Measurer. In Natural Language Processing tasks, the **sentence lenght** intuitively expresses the complexity of a sentence, thus we used the instances' lenghts as measure of complexity. However, instead of focusing on the source of the couple, for this task we decided to sort out the subdatasets computing the difficulty measure on the target, namely the lenght of each method's comment. Once obtained the lenghts, accordingly to the data distribution we took advantage of the quartiles obtained to split the intial dataset in 4 smaller buckets. Accordingly to the training set, the evaluation set was splitted in 4 sub-datasets as well, because the model trained on a defined level (or levels) of difficulty needs to be evaluated on instances with the same complexity, according to the curriculum learning idea.

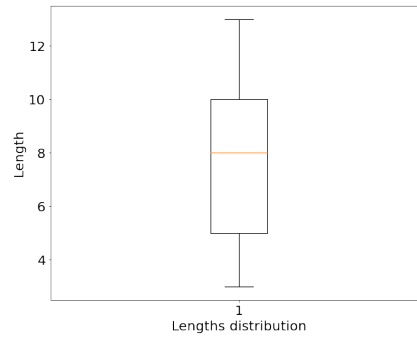


FIGURE 4.2: Code summarization task: data distribution.

Training Scheduler. Similarly to what happens in bug-fixing task, the training scheduler decides to sample from more harder data only when the model converges on the previous easier bucket. The convergence is assessed after a fixed number of epochs, after which early stopping with patience of 5 epochs is run. Once the model converges on the first bucket, the training proceeds on the second and so forth.

4.3.3 Log generation task

Inserting log messages is a practice broadly used and decide where to inject log statements, what information to report through it, and at which log level is as hard as it is crucial [8]. In the following section Curriculum Learning approach is applied at log generation task. The dataset for this task is composed by couple of methods, where the source is the method without the log statement, the target instead has not only the log statement but also the log message.

Those couples where used to train the model to generate and inject log statements in Java code. If the training set for the training on the baseline model was composed by 106.382, the training set for the experiment with curriculum learning is of 105.985 instances. This was because of the Difficulty Measurer we choose to use: according to the latter, 397 instances had a null difficulty, hence were excluded from the set. The same happend in the case of the evaluation set, 13.197 instances were used instead of 13.260. The test set ???

Difficulty Measurer. Given the dataset at our dispose we choose as difficulty measurer the complexity of the instruction set in program execution tasks. More specifically we computed **cyclomatic complexity** through Lizard tool [11]. Cyclomatic complexity is a quantitative software metric developed by Thomas J. McCabe in 1976, used to indicate the complexity of a program. It measures the number of linearly-independent paths through a program module. The measure is computed using the control-flow graph of the program where the nodes of the graph correspond to sets of commands of a program, and a directed edge connects 2 nodes if

the second command might be executed immediately after the first command. It can be applied to individual functions, modules, methods or classes within a program. We decided to compute it on each of the source methods.

It must be said that from the initial dataset were removed 397 instances whose cyclomatic complexity was 0. Decided and computed the difficulty measure for each instance, the dataset was ready to be used by the training scheduler.

Training Scheduler. Once again we observed the data distribution and considered the 3 quartiles to divide the whole dataset in 4 subsets. Each of those represented a different level of difficulty.

As soon as the model converged on the current bucket, BLEU values after each epoch were assessed, and early stopping was applied. We took the best model before divergence to restart the training from the following bucket.

Chapter 5

Analysis of results

In the following section the results achieved by NMT Encoder-Decoder with curriculum learning and by the baselines for the three tasks we considered are reported. We used different metrics for each of the tasks, depending on the metric used in the works that introduced the baselines.

5.1 Bug fixing task

To begin with, as assessed by Tufano *et al.* [9] we used perfect predictions as a metric to evaluate the model performances. In the paper taken as reference to test curriculum learning on, the authors differentiated 2 types of datasets; (i) the first one constituted by methods whose length was up to 50 tokens and (ii) methods with length between 50 and 100 tokens. However, as stated in the previous sections we considered the merging of the two datasets, therefore firstly we reproduced the canonical learning with the merged dataset and then we tested curriculum learning approach. Table 5.1 reports the percentage of bug fixing pairs correctly predicted by the models for different beam sizes. Increasing the beam size, and generating more candidate

BEAM	Baseline	Baby-step
1	5.22 %	5.34 %
5	13.00 %	19.41 %
10	16.55 %	25.16 %
15	18.38 %	28.64 %
20	19.60 %	30.84 %
25	20.51 %	32.39 %
30	21.29 %	33.69 %
35	21.93 %	34.93 %
40	22.26 %	35.72 %
45	22.73 %	36.47 %
50	23.02 %	37.39 %

TABLE 5.1: Models' performances

patches accordingly, the percentages of BFPs for which the models can perfectly generate the

corresponding fixed code - starting from the input buggy code - increases. If the baseline model can predict the fixed code of 5.22% of the BFPs with only one attempt, the same model together with curriculum learning approach performs predicting 5.34% of the same BFPs. It is a decent better result, but looking at bigger beam sizes, the model with curriculum learning performs definitely better, almost triplicating the percentage of perfect predictions when 5 patches are generated, reaching 37.39% when 50 candidate patches are considered. Overall, it can be seen that the improvement margin is constant.

On a second evaluation, we carried on a complementary analysis based on perfect predictions obtained when the model generated 10 candidate patches for each prediction. As can be seen in Table 5.2, the combination between the two models leads to a reasonable percentage of perfect predictions, i.e. 42.38%. However, the model with curriculum learning approach performs even better and on its own. Indeed, 44.63% of BFPs are correctly predicted only by the model with baby-step. On the other hand, only 12.97% of perfect predictions are from the baseline. This result not only indicates that the two approaches are complementary for the bug-fixing

$Dataset(d)$	$Shared_d$	$OnlyBL_d$	$OnlyCL_d$
BF_{all}	42.38 %	12.97 %	44.63 %

TABLE 5.2: Overlapping metrics: baseline and curriculum learning models.

task, but also that the model with curriculum learning reports even better outcomes. Considering that the approach implemented is one of the easiest one, those results recall the need to better tuning the approach with the aim of other improvements in the ability of such a model to exploit the knowledge acquired on this specific task.

5.2 Code summarization task

In the study taken as reference the authors used different neural networks to assess the quality of their work. So, the comparison between our results and theirs is not really straight - forward. However, as said before, we reproduced the experiment on a baseline model, where baseline means the model without any curriculum learning approach applied. The experiment on the baseline model totalized a bleu score of 13.70 that is ... On the other hand, the bleu value

	BLEU-A	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Baseline	13.70	38.8	18.4	10.6	7.1
Baby-step	13.29	38.0	17.8	10.2	7.0

TABLE 5.3: BLEU values summary .

after the experiment on the model together with the curriculum approach scored a value of 13.29, which is lower than the results achieved by the baseline. This means that in this case

the curriculum approach does not work in the view of collect better results. However, it must be said that the models used are different; this might mean that the differences observed may depend on the models used and not on the technique applied.

Also on perfect predictions we do not have better results, but we can observe - as showed in Table 5.4 - that the values are not that much distant one from another.

	Baseline	Baby-step
BEAM 1	4.66 %	4.64 %
BEAM 5	6.68 %	6.53 %
BEAM 10	7.75 %	7.52 %
BEAM 15	8.36 %	8.07 %
BEAM 20	8.84 %	8.51 %
BEAM 25	9.17 %	8.83 %
BEAM 30	9.47 %	9.06 %
BEAM 35	9.74 %	9.29 %
BEAM 40	9.90 %	9.49 %
BEAM 45	10.06 %	9.67%
BEAM 50	%	9.83%

TABLE 5.4: Perfect predictions summary.

5.3 Log generation task

Chapter 6

Conclusion

Appendix A

Appendix

Bibliography

- [1] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.
- [2] V. Cirik, E. H. Hovy, and L.-P. Morency. Visualizing and understanding curriculum learning for long short-term memory networks. *ArXiv*, abs/1611.06204, 2016.
- [3] J. Elman. Learning and development in neural networks: the importance of starting small. *Cognition*, 48:71–99, 08 1993.
- [4] J. L. Elman. Learning and development in neural networks: the importance of starting small. *Cognition*, 48(1):71–99, 1993.
- [5] K. A. Krueger and P. Dayan. Flexible shaping: How learning in small steps helps. *Cognition*, 110(3):380–394, 2009.
- [6] A. LeClair, S. Haque, L. Wu, and C. McMillan. Improved code summarization via a graph neural network. *CoRR*, abs/2004.02843, 2020.
- [7] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, feb 1966. *Doklady Akademii Nauk SSSR*, V163 No4 845-848 1965.
- [8] A. Mastropaolo, L. Pascarella, and G. Bavota. Using deep learning to generate complete log statements. *CoRR*, abs/2201.04837, 2022.
- [9] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4), sep 2019.
- [10] X. Wang, Y. Chen, and W. Zhu. A comprehensive survey on curriculum learning. *CoRR*, abs/2010.13166, 2020.
- [11] T. Yin. **Lizard**.