UNIVERSITY OF MOLISE

DEPARTMENT OF BIOSCENCES AND TERRITORY

MASTERS THESIS

Evaluating Curriculum Learning in Code Related tasks

Author:
Carmen ARMENTI

Supervisor:
Prof. Rocco OLIVETO

Co-supervisor:
Prof. Gabriele BAVOTA

Software Analytics
October 27, 2022

"Everything is just a function of the amount of time put into it."

Contents

List of Figures

List of Tables

To my family,
without whom I would not be who I am.

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 education 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. Skinner[1], 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. Practically shaping shares the same idea of the divide-and-conquer concept.
Even though all the previous said looks to be more appropriate to behavior engineering[2] - being it related to the attempts that experts of the field put into practice to change human or animal behaviors - shaping has been related to the context of machine learning.

1.1 Application context

Bengio *et al* [7] 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

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

[2] Behavioral 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.

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 [**?**]. 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 the simplest 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 [30], 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 reinforcement learning was applied by gradually making the learning task more difficult.

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 [7], 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 be of benefit to 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 to *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 given that training

a NMT model is a time-consuming task a priori, we were curious to see results in terms of both the two most significant requirements aforementioned.

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* [16] 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 lenghty 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.
In humans, learning and development interact in a way that is as important as non-obvious; maturational changes might provide the enabling conditions which allow learning to be most effective. It is a matter of fact that 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 psychologists Skinner (1958), Peterson (2004) and Krueger & Dayan (2009)

where it is called *shaping*. In the context of machine learning, such a meaningful learning process is known as *curriculum learning*.

2.1 State of the art

The basic idea of *curriculum learning* - 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* [14] 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. Owning to this, it is fairily intuitive wondering if the curriculum training strategy could ever benefit machine learning. Extensive experiments from early [7], [31], [56] to recent works [17], [20], [23], [41] in various applications of machine learning show that such strategy is of benefit to 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

As the idea of CL provide a general training strategy beyond specific machine learning tasks, its power have been exploited in considerably wide application scopes, including supervised learning tasks within computer vision [22], [28], natural language processing (NLP) [41], [48], healthcare prediction [15], various reinforcement learning (RL) tasks [18], [40], [45] as well as other applications such as graph learning [19], [43] and neural architecture search (NAS) [22]. As already mentioned in 1.2, the advantages of applying CL training strategies to miscellaneous real-world scenarios can be mainly summarized as (i) improving the model performance on target tasks, and (ii) accelerating the training process, which cover the two most significant requirements in most of the machine learning research.
For instance, Platanios *et al.* [41] present a personal framework that consists of a way of deciding which training instances are shown to the model at different times during training, based

on an estimated difficulty of a sample and on the current competence of the model. Thus, filtering training samples prevents the model from getting stuck in bad local optima, making it converge faster and reach a better solution than the common approach of uniformly sampling training examples. Their experiment shows that CL helps the neural machine translation model reduce training time by up to 70%, while at the same time obtaining accuracy improvements of up to 2.2 BLEU points, compared to plain training without any curricula.
To further illustrate, in [18] Florensa *et al.* talk about curriculum learning as a reverse curriculum technique. They propose a method to learn goal-oriented tasks without requiring any prior knowledge, other than obtaining a single state in which the task is achieved. They dimonstrate that their approach is based on a reverse training, where the robot gradually learns to reach the goal from a set of start states increasingly far from the goal. That approach resulted in solving hard problems, not solvable by state-of-the-art reinforcement learning methods.

2.3 Curriculum Learning application in Deep Learning tasks

Training neural networks is traditionally done by providing a sequence of random mini-batches sampled uniformly from the entire training data. Conversely, curriculum learning involves the non-uniform sampling of mini batches, on the training of deep networks. However, understanding why and when *starting small* strategies can benefit machine learning algorithms is a question that every scholar try to contribute to. Bengio *et al.* [7] other than showing several cases where very simple multi-stage curriculum strategies give rise to improved generalization and faster convergence, contribute to the question introducing a hypothesis which might help to explain some of the advantages of a curriculum strategy. They argue that a well chosen curriculum strategy can act as a continuation method. Intuitively, continuation methods are optimization strategies for non-convex criteria which first optimize a smoother and easier version of the problem to reveal the "global picture", and then gradually consider less smoothing versions, until the target objective of interest. Therefore, continuation methods provide a sequence of optimization objectives, starting with an objective for which it is easy to find a global minimum, and tracking the local minima throughout the training. In this way, continuation methods guide the training towards better regions and the local minima learned from easier objectives have better generalization ability and are more likely to approximate global minima. To test this hypothesis, they turn the attention to training of deep architectures, which have been shown to involve good solutions in local minima that are almost impossible to find by *random initialization*. Generally, deep learning methods try to learn feature hierarchies, i.e., features at higher levels are formed by the composition of lower level features. As a consequence, automatically learning multiple levels of abstracion may allow a system to induce complex functions mapping the input to the output directly from data, without depending heavily on human-crafted features. One possible theoretical motivation for deep architectures comes from complexity theory. Some functions, in fact, can be represented with an architecture of depth k but require an exponential size architecture when the depth is restricted to be less than k. Training deep networks, however, involves a potentially intractable non-convex optimization

problem [7]. There were no good algorithms for training fully-connected deep architectures before 2006, when Hinton introduced a learning algorithm that greedily trains one layer at a time, exploiting an unsupervised generative learning algorithm for each layer. It is conceivable that by training each layer one after the other, the network is organized in such a way that it can first learn the simpler concepts, represented in the first layer indeed, then sligthly more abstract ones, represented in the second layer, and so on. Not long after, strategies for building deep architectures from related variants were proposed and these works showed the advantage of those frameworks over shallow ones, and of the unsupervised pre-training strategy in a variety of settings.

2.4 Definition of Curriculum Learning

Based on all the previous works in the first place provided in behavior and cognitive science literature, the concept of CL was first proposed in [7] with experiments on supervised visual and language learning tasks, exploring when and why curriculum could benefit machine learning. The original definition of CL refers to a curriculum as a sequence of training criteria over T training steps $C = \langle Q_1, ..., Q_t, ..., Q_T \rangle$, where each criterion Q_t is a reweighting of the target training distribution $P(z)$: $Q_t(z) \propto W_t(z)P(z)$, for each instance in the training set.
In the definition the following three conditions were considered:

- the entropy of distributions gradually increases, $H(Q_t) < H(Q_{t+1})$;
- the weight for any example increases, $W_t(z) \leq W_{t+1}(z), \forall z \in D$;
- $Q_T(z) = P(z)$.

So, curriculum learning is the training strategy that trains a machine learning model with a curriculum. The first of the above conditions means that the diversity and the information of the training set should gradually increase. More specifically, the reweighting of examples in later steps increases the probability of sampling slightly more difficult examples. The second condition instead means to gradually add more training examples, letting the size of the training set increase. Finally, the last condition means that the reweighting of all examples is uniform and the training is carried on the target training set.
At a more abstract level, a curriculum can be seen as a sequence of instance selection or example reweighting along the training process in order to achieve faster convergence or better generalization, which is beyond the "easy to hard" or "starting small" principles.

2.5 A general CL framework

In a nutshell, curriculum learning means "training from easier data to harder data" [50]. More specifically the core idea is to "start small" [16], 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 [50]. Therefore, Difficulty Measurer together with Training Scheduler constitute a general framework for curriculum design, as illustrated in Figure 2.1. First of all, the Difficulty

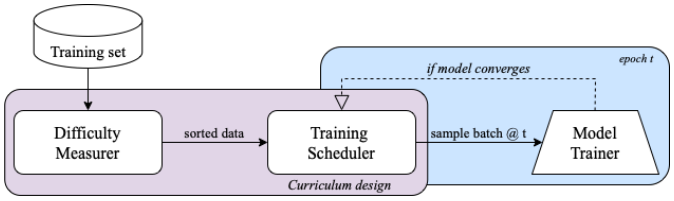

FIGURE 2.1: Predefined Curriculum Design.

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.
Moreover, 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 introducting Curriculum into Machine Learning depends on how the curriculum for specific applications and dataset is designed. Due to this, Difficulty Measurers often relies on the data characteristics of specific tasks, and most of them are defined by a definition of complexity, diversity, or noise estimation.

2.5.1 Predefined Curriculum Learning

To begin with, the *difficulty measurer* depends on the data characteristics, and consequently on the specific task. By and large, common types of difficulty measurers are designed for image and text data, as usually happens in computer vision and natural language processing tasks, but also for other data types such as audio data and programs. As we mentioned before, difficulty measurers can rely on data complexity, diversity, or noiseness. In the first place, *complexity* refers to the structural complexity of a data example, in a way that the higher the complexity,

the harder for the model is to capture the samples. Some of the most popular difficulty measurers are: the *sentence length* - the most used in NLP tasks; the *parse tree depth* - where the complexity is based on the grammar; the *nesting operations* in a program - that computes the complexity based on the number of instructions in program execution tasks. As for *diversity*, the distributional diversity of elements in a group data is meant to define a type of difficulty measurer. The more various the data, the harder the learning process is. When more - or rare - types of data is included in the training data set, training is more difficult for the model. One of the most used measure of diversity is information entropy, which in text data is exploited as the Part-Of-Speech (POS) entropy. One other measure is the word rarity. Last but not least, another definition of difficulty is the *noise extimation*, which estimates the noise level of data examples and classifies cleaner data as easier. To provide an illustration, in [12] the authors present an approach to train a webly supervised convolutional neural networks (CNN), inspired by curriculum learning. They suppose that images retrieved by a search engine like Google are supposed to be cleaner, thus easier, while images posted on photo-sharing website like Flickr are more realistic, therefore noiser and consequently harder. So, examples with lower local density are considered to be harder to predict. On top of that, there are also other difficulty measurers that rely on characteristics such as signal intensity or humanly-annotated image difficulty scores.

If predefined difficulty measurers vary over different data types and tasks, on the other hand the existing predefined *training scheduler* are usually independent from the data - and then from the task. However, training schedulers can be differentiated as well; they are usually divided into *discrete* and *continuous* schedulers. Discrete schedulers adjust the training data subset following a specific criterion - be it a fixed number of epochs or the convergence on the current data subset - whereas continuous schedulers adjust the training data subset at every epoch. Thanks to their semplicity and effectiveness *discrete schedulers* are widely adopted and the most popular discrete scheduler is named as *Baby Step*.

Algorithm 1 Baby Steps Curriculum [13], [51]

Input: **C**: training dataset; **D**: Difficulty Measurer;
Output: $\mathbf{M}^*$: optimal model.

$D' = sort(D, C)$;
$\{D^1, D^2, ..., D^k\} = D'$ where $C(d_a) < C(d_b), d_a \in D^i, d_b \in D^j, \forall i < j$;
$D^{train} = \emptyset$
for $s = 1$ to k **do**
    $D^{train} = D^{train} \cup D^s$;
    while not converged for p epochs **do**
        train(M, D^{train});
    end while
end for

Algorithm 1 first distributes the sorted data into buckets - from easy to hard - and starts training with the easiest one, shuffling the current buckets and the data in each bucket and sampling mini-batches for training. Then, when convergence is reached or after a defined number of

epochs, the next bucket is merged in the current one. This process continues until all the buckets are merged in one unique training set; at this point the training process either stops or continues several extra epochs.
Another discrete scheduler is called *One-Pass* which uses a similar strategy for data bucketing and for starting the training from the easiest one; however, if Baby Step merges the buckets as soon as the model converged on the previous bucket, One-Pass scheduler when updating the training data discards the current bucket and switches to the next harder one.

Algorithm 2 One-Pass Curriculum [13]

Input: **C**: training dataset; **D**: Difficulty Measurer;
Output: $\mathbf{M}^*$: optimal model.

$D' = sort(D, C)$;
$\{D^1, D^2, ..., D^k\} = D'$ where $C(d_a) < C(d_b), d_a \in D^i, d_b \in D^j, \forall i < j$;
for $s = 1$ to k **do**
    while not converged for p epochs **do**
        train(M, D^{train});
    end while
end for

One-Pass is less used than Baby Step in curriculum learning literature due to the lower performance in many tasks. In fact, the complexity or diversity of the training data if gradually increasing helps improve generalization - as happens in Baby Step scheduler; on the other hand, One-Pass scheduler (see Algorithm 2) is like training on a sequence of independent tasks as in continual learning - which faces the problem of forgetting even though the early tasks are easier (we also faced this issue as we explain in Chapter 4). Other discrete schedulers are also based on data bucketing, but take different sampling strategies. For example, some [29] modify the Baby Step to unevenly divide the examples into buckets such that easier buckets have more data examples; then they sample instances without replacement from the easiest bucket only until there remain the same number of examples as in the second easiest bucket. Afterward, they uniformely sample from the first two buckets until the size is the same as that of the third bucket.
In [56] instead, the researchers base their experiments on 2 measures to define a difficulty measurer: *length* and *nesting* of a dataset of programs. They evaluate 4 different curriculum learning strategies, two of which proved to be better than using no curriculum strategy or the naive curriculum strategy - the latter was tested in [7] as well. The first of the two approach - both successful in the context of evaluating short computer programs - is a *mixed strategy* where the training samples are picked randomly over a set of random lengths $[1, a]$ and random nestings $[1, b]$, thus using a balanced mixture of easy and difficult examples at every point during training. The second one instead is a *combined strategy* between the naive curriculum strategy and the mixed strategy. In this approach, every training case is obtained either by the naive method or by the mixed one. Thence, the combined strategy always exposes the network at least to some difficult examples, which is the key element in which it differs from the naive curriculum strategy. As explained in their piece of work, both their curriculum strategies outperform the naive curriculum strategy, especially the combined one, in a matter of hidden state allocation.

Predefined CL limitations. Even if predefined curriculum learning is simple and effective, there are some limitations in terms of (i) applicability, (ii) difficulty measurer definition, (iii) training scheduler definition. First of all, an expert domain knowledge is needed not only to find the most suitable combination of difficulty measurer and training scheduler for a specific task and its dataset, but also for designing both of them in such a way that are predefined before the training starts. Moreover, defining a predefined difficulty measurer means humanly decide what should be the difficulty boundaries of a model - which are different from those of humans. Finally, when a training scheduler remains fixed it is not flexible enough, since it ignores the feedback of the current model to some extent. On top of the previous, the best hyperparameters of training scheduler are hard to find and there are no other methodologies other than exhaustive trials. In addition, taking Baby step as example, a basic problem in its scheduler is the decision of the number of buckets and how to divide them. In fact, deciding by thresholds on difficulty scores makes it hard to assign each bucket with roughly the same number of instances; on the other hand, division by size might result in fluctuations in difficulty within a bucket or not enough difference between different buckets.

These limitations of predefined curriculum learning has led to the development of more data-and-model-driven approaches rather than human-driven, in such a way that they can be more dynamically adaptive to the training in process: automatic CL approaches.

2.5.2 Automatic Curriculum Learning

The automatic curriculum learning approaches try to break the limits of predefined curriculum learning approaches. Taking as reference the issues mentioned before, i.e., (i) applicability, (ii) difficulty measurer definition, and (iii) training scheduler definition, by and large it can be said that: (i) in terms of *applicability* automatic CL strategies tend to be general and domain agnostic; (ii) *difficulty measurers* are model decided thus dynamic; (iii) *training schedulers* are dynamically defined and redefined by model feedbacks.
In predefined CL who designs the curriculum is a human expert, whereas the entity that is trained by the curriculum is the machine learning model, metaphorically the two parties can be referred as a *teacher* and a *student*. So, automatic CL is meant to reduce the need for *human* teachers, and in [51] the authors summarize four methodologies of predefined CL that take different ideas, among all the ideas.

Firstly, **Self-paced Learning** (SPL) is a primary branch of CL that automates the Difficulty Measurer by taking the example-wise training loss of the current model as criteria. The SPL methods let the students themselves act as the teacher and measure the difficulty of training examples according to its losses on them. This concept is analogous to the self-study human approach, in fact originates from human education, where the student can control the learning curriculum , including what, how, when, and how long to study. In machine learning context, SPL refers to a training strategy - initially proposed by Kumar *et al.* [31] - which trains the model at each iteration with the proportion of data with the lowest training losses. Such a proportion of easiest examples gradually increase until the whole training set is considered, so

using a predefined Training Scheduler. Even if in the literature of SPL, CL and SPL are usually mentioned as two different strategies, where the CL refers to the predefined CL, in the piece of work aforemetioned SPL is considered as a branch of automatic CL, since it shares the spirit with CL and fits with the general framework as explained by Kumar *et al.* in [31] and shown in Figure 2.2. The most valuable advantages of SPL over predefined CL are twofold: (i) SPL is

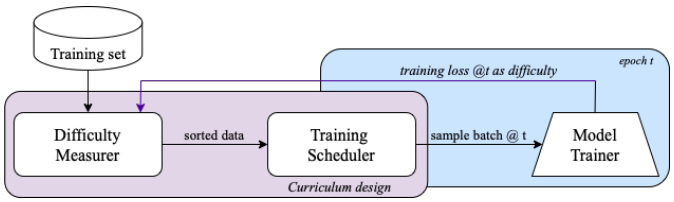

FIGURE 2.2: Automatic CL: Self-paced Learning Design.

semi-automatic CL with a loss-based automatic Difficulty Measurer and dynamic curriculum, which makes the strategy more flexible and adaptive for various tasks; (ii) SPL includes the curriculum design into the learning objective of the original machine learning tasks, thus making it applicable as a tool.
The authors in [51] also specify that other than the original version of SPL, there also exist not only theories for the convergence, robustnessm and essence of SPl to support applications, but also enhanaced versions from different perspectives. Moreover, they argue that as a student-driven weighting strategy on the learning objective, the core design of SPL is the Self Paced regularizer, which directly determines the optimal weights at each training epoch. Therefore, most of the existing improvements on SPL have been focused on SPL regularizers. Without going into details, they differentiate (i) Soft SP-regularizers, comparing them to Hard Regularizers; (ii) Prior-embedded SPL; (iii) other enhancements of SPL.

Secondly, there exists a reliable teacher-driven difficulty strategy, that is the **Transfer Teachers** approach. SPL takes the current student model as an automatic Difficulty Measurer, however, this strategy has a risk of uncertainty at the beginning of the training, when the student model is not sufficiently trained. As happens in human education in fact, if students understand little about - or misunderstand - the learning materials, it would be hard for them to measure the difficulty of the materials and find out the easy ones. So, an idea is to ask a mature tacher for help: an educated person can assess the materials and form an easy-to-hard curriculum. This idea leads the CL approaches to the Transfer Teachers strategy (see Figure 2.3). The

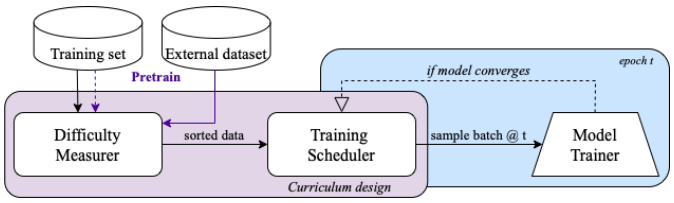

FIGURE 2.3: Automatic CL: Transfer Teacher Learning Design.

latter is a semi-automatic method that first pretrains a teacher model on the training datasat

or an external dataset, ad then transfers its knowledge to compute the example-wise difficulty, based on which a predefined Training Scheduler can be applied to finish the CL design. Transfer Teachers can be helpful to the tasks where the example-wise eaasiness is hard to define. The most general Transfer Teachers are loss-based, thus they do not need any domain knowledge and are closely related to SPL. Concretely, these methods take the example-wise losses calculated by a teacher model as the example difficulty and assume that the lower the loss, the easier the example. The teacher model can either be different from the student model [52] - so being more complex, or share the same structure with it [23], [53].

Then, **Reinforcement Learning (RL) Teacher** methods are exposed, which involve a student model and a reinforcement-learning-based teacher model. The SPL and Transfer Teacher only automate the Difficulty Measurer and still use predefined Training Scheduler, thus they only consider one side of the *curriculum* or *teaching* scenario. On one hand SPl taked the student feedback, i.e., the losses to adjust the curriculum, whereas Transfer Teacher leverages the teacher's knowledge to determine the order of presenting learning materials. However, in a common and ideal human-based education a teaching strategy usually involve both the teacher and the student, where the student can interactively provide feedback to the teacher, and the teacher can adjust the teaching action accordingly. By doing so, the teacher and the student make progress together. Even if this strategy costs more than SPL and Transfer Teacher, is the most flexible and completely automatic. As can be seen in Figure 2.4, it is clear that with

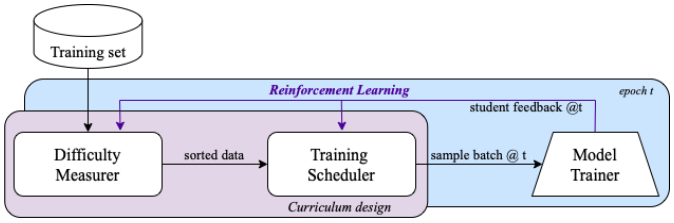

FIGURE 2.4: Automatic CL: Reinforcement Learning teacher Design.

the teacher-student interactive stratey, RL Teacher achieves the fully-automated CL design. At each training epoch, the RL teacher will dinamically select examples for training according to the student feedback. The RL Teacher sets the teacher model as both the Difficulty Measurer and Training Scheduler by dynamically considering the student feedback.
On top of that, RL Teacher methods make it possible to set different student feedback according to different goals, in fact, they are also suitable for multi-tasking learning, where the teacher model selects the most valuable tasks for the student training. Some examples are AutoCL [20] and TSCL [37], and for both of them the goal is to learn a student model that achieves high performance on all the tasks.

Finally, the scholars report **other Automatic CL** approaches. In fact, besides RL Teacher, there exist some other fully-automatic CL designs. These designs should require the generation of the curriculum to rely only on the dataset, the student model, and the goal of the task. Thus, accordingly to this idea and recalling the CL definition in Section 2.4, from the optimization perspective the authors summarize that the objective is optimize the mapping between

{the data, the current state of the student model, the task goal} and the training goal. To this end, RL Teacher methods typically adopt a RL framework to learn the policy for data selection, but as the authors report, there are more optimization methods such as Bayesian Optimization, Stochastic Gradient Descent, Meta-learning, and Hypernetwork, which also demonstrated to have great potential to learn this mapping.

To sum up, even though there are a lot of different CL methodologies, there is no systematic conclusion with regard to howo to choose amon them in real-world applications. One useful principle for selecting a proper CL category is to consider how much prior knowledge one have about the dataset and the task goal. If sufficient expert domain knowledge is available, then predefined CL methods are more preferable to design a *knowledge-driven* curriculum specifically suitable to the exact scenario. On the other hand, if there is no prior assumptions on the data, then automatic CL methods are more prefereable to learn a *data-driven* curriculum adaptive to the underlying dataset and task goal.

Chapter 3

Deep Learning Applications to Software Engineering tasks

The advent of deep learning (DL) has fundamentally changed the landscape of modern software and in order to cope with the increasing complexity of digital system programming, deep learning techniques have recently been proposed to enhance software deployment by analysing source code for different purposes, ranging from performance improvement to debugging and security assessment. Driven by the success of deep learning in data mining and pattern recognition, recent years witnessed an increasing trend for researchers to integrate deep learning with software engineering (SE) tasks. In the most typical SE tasks, deep learning helps to generate or summarize source code, predict defects in software, extract requirements from natural language text, and many more tasks. Generally, a DL system is made of several interconnected computational atomic units that form *layers* which perform mathematical transformations, according to sets of learnable parameters, on input data. These architectures can be trained for specific tasks updating the parameters according to model configuration on a specific set of training data.
Over the years, deep learning has developed advancements in many complex tasks often associated with artificial intelligence, within software engineering is now comprehended. DL is intertwined with SE, in fact, DL techniques allow to automate or improve existing software development tasks nowadays.

3.1 State of the art

Given the effectiveness by which DL systems are able to learn representations from large data corpora, there is ample opportunity to leverage DL techniques to help automate or improve a wide range of code related tasks. Software engineering research investigates questions related to the design, development, maintenance, testing and evolution of software systems. Previously, the software engineering community has applied traditional machine learning techniques to identify interesting patterns and unique relationships within the data to automate or enhance many tasks typically performed by developers. Due to recent improvements in computational power and the amount of memory available in modern computer architectures, the rise of deep learning has led to a new class of learning algorithms suited for large datasets. Deep learning represents a fundamental shift in the way by which machines learn patterns

from data by automatically extracting salient features for a given computational task, as opposed to relying upon human intuition. Given the immense amount of data in software repositories that can serve as training data, deep learning techniques have ushered in advancements across a range of tasks in software engineering research, including automatic software fixing, code suggestion, defect prediction, feature location, among many others. This field of research shows clear potential for transforming the manner by which a variety of specific software development task are performed.

3.2 Software Engineering related works

Taking as reference the piece of work published by [6], in this section some of the ML and DL techniques to ssource code analysis are presented, briefly described in chronological order by year of publication.
Between 2013 and 2015, machine learning techniques were explored on high-level code, starting from a manual definition of features to the first deep learning applicactions capable of extracting features from code on their own. The techniques used are strongly inspired by the background of natural language processing - being *most software natural* [25] since it is created by human beings - and include n-grams, decision threes, and recurrent neural networks (RNN). RNNs in particular have been extensively tested to assess their effectiveness; Rayachev *et al.* [44] proposed a code completion strategy that compares n-gram model and recurrent neural networks. They implement a technique that first extract sequences of API calls from the dataset, then apply n-gram to these sequences, and finally use the RNN to take the last word in the sequence as input and uses one-hot-encoding to predict probabilities for the most likely next word. They argue that the n-gram technique can discover regularities between the last n - 1 elements of fucntion-calls sequences, whereas RNN can discover relations at longer distances.
Zaremba *et al.* in [56] try instead to test the limits of Long Short Term Memory (LSTM) based networl with a task considered difficult. Specifically, given a code fragment, inferring the result as if one was running the code. They defined a simple class of programs and used a LSTM cell in a sequence-to-sequence model to obtain the result of the code execution. They reached 99% of accuracy. The LSTM network receives direct input of the characters, thus it was not necessary for them to develop a complex embedding layer.

Shortly after, between 2016 and 2018, ML techniques for code started to be explored more in depth. Together with the RNNs and their variants, such as LSTM and GRU - Gated Recurrent Units - cells, complex model based on a sequence-to-sequence structure emerged [27], [8], alongside an adaptation of CNN on tree structures [39], [10] and the introduction of a new neural network structure: the Graph Neural Network (GNN) [1], [47]. The techniques were applied on Sequences of Tokens extracted from the source code, Abstract Syntax Tree [39] and API call sequences. The problems addressed span from code description and code comprehension, to syntax error recognition [46] and code fixing as well.

Recently, the topic has become more mature and more works have tried to exploit the Encoder-Decoder structure [4], the GNN models [3], [9], or other models based on hierarchical structures. The problems addressed are, for instance: function and type inferring [5], [42], performance prediction [38], and loop-optimization [24], [9].

The topic described approaches code analysis with machine learning and deep learning. However, the spectrum of problems in these areas is actually very broad and the works mentioned lead to the necessity to apply ML techniques to approach code related tasks. In the following sections, tree code related tasks are described thoroughly.

3.2.1 Bug fixing task

Currently, there are millions of open source projects with numerous bug fixes available in code repositories thanks to the proliferation of software development. This can be leveraged to learn how to fix common programming bugs. Localizing and fixing bugs is an effort-prone and time-consuming task for software developers; thus, to support programmers in this common activity, researchers have proposed a number of approaches aimed to automatically repaire programs. The work we took as reference is motivated by three main considerations. First of all, automated repair approaches are based on a limited and hand-crafted set of transformations or fitting patterns. In the second place, the work done by [32] shows that the past history of existing projects can be successfully leveraged to understand what a *meaningful* program repair patch is. Finally, several works have recently demonstrated the capability of advanced machine learning techniques, such as deep learning, to learn from relatively large software engineering datasets. In the work, the authors expanded upon the original idea of learning bug-fixes and evaluate the suitability of a NMT network to automatically generate patches for buggy code. Actually, software developers can access to plenty of change history and bug-fixing commits from a large number if software projects, from GitHub for example, and a machine-learning based approach can use this data to learn about bug-fixing activities. In fact, automatically learning from bug-fixes provides the chance to emulate real patches written by developers. Moreover, the scholars harness the power of NMT - originally meant for translation purposes - to attempt indeed the *translation* of buggy code into fixed code emulating the combination of Abstract Syntax Tree (AST) operations performed in developers written patches.

The Approach. They mined from GitHub Archive [21] bug-fixing commits from thousands of repositories, considering every public GitHub event between March 2011 and October 2017. For each bug-fixing commit, the authors extracted the source code before and after the bug-fix, discarding commits related to non-Java files. In this way, they collected method-level bug-fixing pairs (BFPs), that is the pre and post commit code, ending having $2.3M$ bug-fixing commits. 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. Since learning bug-fixing patterns is highly challengin by working at the level of raw source code, they abstracted each BFP in isolation using a Java lexer, a parser and a list of idioms to represent each buggy and fixed method within a BFP as a stream of tokens. After that, they filtered out: (i) BFPs that contained lexical or syntactic errors - either because of lexer or parser failure in processing them; (ii) the pairs whose buggy and fixed abstracted code resulted in equal strings; (iii) BFPs that performed more than 100 AST actions between the buggy and fixed version - so to eliminate outliers of the distribution which could have hindered the learning process. Subsequently, they considered BFPs based on their size measured in the number of tokens and disregarded methods longer than 100 tokens and focused on two sizes thus creating two datasets: BFP_small composed by BFPs with maximum 50 tokens and BFP_medium composed by BFPs made up of a token number between 50 and 100. Then, given those datasets they used their instances to train an Encoder-Decoder model, where the instances were randomly partitioned into: training (80%), validation (10%) and test (10%) sets. They also made sure to discard any possible duplicate.

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 **x** into a vector representation;
- a RNN Decoder, which decodes the vector representation into another sequence of terms **y**.

The model learning is based on a conditional distribution, where the output sequence of terms is conditioned by the input sequence: $P(y_1, ..., y_m | x_1, ..., x_n)$, and where m and n not necessarly have to have the same length. The Encoder takes as input a sequence $\mathbf{x} = (x_1, ..., x_n)$ and produces a sequence of states $\mathbf{h} = (h_1, ..., h_n)$. 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 $\mathbf{y} = (y_1, ..., y_n)$ given **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 $(y_1, ..., y_i - 1)$;
- a context vector c_i, which constitutes the attention mechanism.

The vector c_i is a weighted average of the states in **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 weigth 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.
In terms of hyperparameter search, for both models build on the BFP_small and BFP_medium datasets, the scholars tested 10 encoder-decoder architecture configurations. The configurations tested different combinations of RNN cells - LSTM and GRU; number of layers - 1, 2,3; units for both the encoder and the decoder - 256, 512; embedding size - 256, 512. Bucketing and padding was used to deal with the variable length of the sequences. They trained their models for a maximum of 60k epochs and selected the model checkpoint before overfitting the training data. Moreover, to guide the selection of the best configuration, they used the loss function computed on the validation set; the results were computed on the test set. Once the model was trained, it was evaluated against the test set of unseen buggy code and apart from using the classic greedy decoding approach that selects at each timestep the output with highest probability, they also used Beam Search decoding in order to generate up to 50 potential patches for a given buggy code.

The scope of their work was to answer to 3 research questions: (i) if the NMT was a viable approach to learn how to fix code; (ii) what kind of operations were perfomed by the models; (iii) what was the training and inference time of the models. The results obtained showed that: (i) training a model both on small BFPs and medium BFPs, NMT showed to be a viable approach to learn how to fix code; (ii) the models exhibited a very high syntactic correctness of the generate patches ranging between 99% and 82% and the learned operation are the most representative ones giving the chance to theoretically fix a large percentage of bugs; (iii) after training for less than 15 hours, the models were able to generate 50 candidate patches for a single bug in less than a second.

In the end, they empirically investigated the potential of NMT to generate candidate patches that are identical to those implemented by developers. Their results indicate that the trained NMT model can successfully predict the fixed code - given the buggy code - in 9% of the cases (as for the small dataset) and in 3% of the cases (as for the medium dataset).

3.2.2 Code summarization task

Automatic source code summarization is the task of generating short natural language description for source code and is a rapidly expanding research area [33]. Automation has been desired as an effective alternative for the manual effort of writing summaries. Since the term *code summarization* was coined the field has proliferated and at first, the predominant strategy was based on sentence templates and heuristics derived from empirical studies. Then, around 2016, data-driven strategies based on neural networks began to rise, leveraging gains from both the Artificial Intelligence (AI)/Neural Language processing and mining software repositories research communities. The data-driven approaches were inspired by neural machine

translation from NLP, where originally a sentence in one language is translated into another language. As well as a neural architecture is used to learn the mapping betweek words and even the correct grammatical structure from one language to the other, in code summarization the analogy consists in treating source code as one language input and summaries as another. Thus, code is the input to the same models encoder, and summaries to the decoder. Unfortunately though, the metaphor to NMT has some major limits. Firstly, source cose has fewer words that map directly to the corrispondent summary. Secondly, source code tends not to be of equal length to summaries, but it is much longer. Finally, source code is not only composed by words. Moreover, code is a complex structure made of interacting components such as classes, routines, statements, and identifiers connected through different relationships. During the year, software engineering have recognized that code is more suited to graph or tree representations, yet the typical application of NMT to code summarization treats code as a sequence to be fed into a RNN designed for sequential information. Around 2020 the literature begun to recognize the limits to sequential representation of code for code summarization and some scholars [26], [34], [4] showed how neural networks can be effective in extracting information from source code better in a graph or tree form than in a sequence of tokens. What is missing though is *how* graph neural networks - first proposed by [2] to learn representations of code for the problem of code generation - improve representations of code based on the AST. GNN-based representations improve performance indeed, but the degree of that improvement for code summarization has not been exploited thoroughly. LeClair *et al.* [33] then presented an approach for improving source code summarization using GNNs.

The approach. The idea of code summarization tasks 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, the authors targetted the problem of summarizing program subroutines. The approach of the authors is based on the `graph2seq` model presented by Xu *et al.*[54], to which they applied few modifications to customize the model to a software engineering context.
Their model is based on Convolutional GNNs (ConvGNNs). ConvGNNs take graph data and learn representations of nodes based on the initial node vector and its neighbors in the graph. The process of combining the information from the neighboring nodes is called *aggregation*. By aggregating infomration from neighboring nodes, a model can learn representations based on arbitrary relationships which can be structures of a sentence, parts of speech, dependency parsing trees, or sequence of tokens. ConvGNNs also allow nodes to get information from other nodes that are further than just a single edge - called also hop - away. Each time a hop is performed, the node gets information from its neighboring nodes, accordingly to that hop.
The neural model used by LeClair *et al.* in [33], is based on the model proposed in [34] and uses ConvGNNs. Their approach follows 5 steps:

- Embed the source code sequence and the AST node tokens;
- Encode the embedding output with a recurrent layer for the source code token sequence and a ConvGNNs for the AST nodes and edges;

- Use an attention mechanism to learn tokens in the source code and AST;
- Decode the encoder outputs;
- Predict the next token in the sequence.

The dataset they used was provided by LeClair *et al.* in [11] and is composed by 2.1 million Java method-comment pair. The authors used its tokenized version.

Their research objective is to determine if their proposed approach of using the source code sequence together with a graph based AST and a ConvGNN outperform the current baselines and the reason why that happens if that is the case As for their resulting quantitative evaluation, they tested 3 model configurations and they found that the model that uses an encoder for the source code tokens, a ConvGNN encoder for te AST and a Bilateral-LSTM on the output of the ConvGNN resulted in having the highest performing approach obtaining a BLEU-A score of 19.93 and ROUGE-LCS score of 56.98. This model outperformed the nearest graph-based baseline by 4.6% BLEU-A and 0.06% ROUGE-LCS and the flattened AST baseline by 5.7% as for BLEU-A and 12.72% as for ROUGE-LCS. They attribute this increase in performance to the use of the ConvGNN as an encoding for the AST, this allows the model to learn better AST node representations than it can with onlly a sequence model [33].

3.2.3 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 [36]. This is a practice that helps developers in several software maintaninance activities in several phases of the software lifecyle. Usually developers use log statements to expose and register information regarding the internal behavior of a software tool in a human-comprehensible way and most importantly they insert those in strategic positions, specifying appropriate log levels, and defining compact but also compregensible text messages. Yet deciding where, what and at which level to log is hard and demanding.

The approach. The piece of work carried on by Mastropaolo *et al.* presents the first approach supporting developers in the aforementioned decisions, called LANCE. This tool uses a Text-To-Text-Transfer-Transformer (T5) model that was trained on 6,832,859 Java methods. LANCE takes in input a Java method and injects in it a full log statement, including a logging message, properly choosing the needed log level and the statement location. The approach presented by the authors was aimed at exploiting the T5 model [] to automatically generate and inject complete logging statements in Java code. They started with a double-goal pre-training of the model on the entire dataset: (i) the first is the classic "masked token objective, where they masked 15% of the code tokens in the Java methods asking the model to guess them; (ii) the second provides as input to the model Java method grom which log statements originally present in it have been removed, in this case the T5 had to guess where the log statement was needed.

Once pre-trained, the model was fine-tuned to generate complete log-statements. LANCE automatically generated 12.020 log statements and compared them to the ones manually written by developers. The results show that LANCE was able to: (i) predict the appropriate location of a log statement in 65.9% of cases correctly; (ii) select a proper log level for the statement in 66.2% of cases; and (iii) generate a completely correct logging statement, messages included, in 15.2% of cases.

Among the automated support provided to developers for logging activities, LANCE is the first technique able to generate complete logging statements and to inject them correctly.

Chapter 4

Evaluating Curriculum Learning in Software Engineering tasks

Before deciding to use for all our experiments the *Baby Steps* approach, we tried also the *One-Pass* algorithm. However, after some attempts we noticed that - as stated in Section 2.5.1 - LSTMs tend to forget what they learnt in previous steps when the training dataset change completely. So we decided to test *Baby Steps*.
We adopted the most popular discrete scheduler, known as *Baby Steps*, 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 Steps* 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 thougth 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; this configuration is the one that in [49] allowed the authors to achieve the best results. So, we used same architecture used by Tufano *et al.* in [49] (piece of work that we took as starting point for our first task experiment).

In this chapter, we present how we applied Curriculum Learning to some of code related 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 asses 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 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.

4.1.1 Bug-fixing task

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.* [49]. 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 greeding 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 broads 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 rest 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.

4.1.2 Code summarization task

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 configurations 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.

4.1.3 Log generation task

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 instances. The model was trained for 45 epochs, i.e. 149.625 steps, and it must be specified that it was evaluated after each 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.

4.2 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 ideas of complexity. 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.

4.2.1 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 approch 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 [35]. 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; so we defined an incremental difficulty criteria to feed the model with: the bug-fixing training scheduler described as follows. Not only did we split the training dataset in 4 sub-datasets, but also the evaluating set.

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. Once the model performed the maximum steps set in the beginning, early stopping is run and we set a patience of 5 checkpoint. For patience we mean the number of steps with no improvement after which training will be stopped. We set a patience of 5, meaning that if after 5.000 steps - since evaluation is run every 1.000 steps - the loss value does not improve, then the training is stopped. We repeated this process for each of the subsets identified by the difficulty measurer, i.e. for 4 subsets.

4.2.2 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.

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 the initial 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.

4.2.3 Log generation task

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, where 13.197 instances were used instead of 13.260. The test set was used in its whole, out of 12.020 instances none of them had null cyclomatic complexity.

Difficulty Measurer. Given the dataset at our dispose we choose as difficulty measurer the complexity of the instructions set in program execution tasks. More specifically we computed **cyclomatic complexity** through Lizard tool [55]. 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 training and evaluation datasets were removed 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 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 that specific 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 baseline models.

5.1 Bug fixing task

To begin with, as assessed by Tufano *et al.* [49] we used perfect predictions as a metric to evaluate the model performances. In the paper we took 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 merge 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 better

result, but looking at bigger beam sizes, the model with curriculum learning performs even better, almost triplicating the percentage of perfect predictions of beam size 1 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 curriculum 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 took 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 straigh-forward. However, as said before, we reproduced the experiment on a baseline model, i.e. on the model without any curriculum learning approach applied. The experiment on the baseline model totalized a BLEU-A score of 13.70 that is way lower than each of the scores obtained by the authors of LeClair [33]. On the other hand, the BLEU value after the experiment on the model

	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 .

together with the curriculum approach scored a value of 13.29, which is lower than the results achieved by the baseline, then lower than the BLEU values in [33]. This means that in this case the curriculum approach does not work in the view of collecting better results. However, it must be said that the model used in our experiment is different from the model used by the authors of the paper we took as reference; this might mean that the differences observed may be caused by the fact that the original model should have been used and not on the technique applied.

Also on perfect predictions we do not have better results, and 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	10.24 %	9.83%

TABLE 5.4: Perfect predictions summary.

5.3 Log generation task

Chapter 6

Conclusion and Future Works

In this work we wanted to test the curriculum learning approach on code related tasks. In order to test this new technique we decided to take as reference 3 pieces of work and to compare our findings to the results achieved, by using deep learning models. As explained in Chapter 1.2 there are well-known advantages that curriculum learning brougth to the current state of the art. However, it is also true that advantages and benefits depend highly on the data that the model takes in input; consequently, the results with curriculum applied might not always be better than those obtained on the plain models. In fact, we could notice that if on bug-fixing task the model with curriculum learning took better results not only in terms of performance but also of accuracy, in the case of the second task, i.e. code summarization, the results achieved in [33] are definetly better. Nevertheless, it must be said that we focused our attention on the technique for the training process, but in order to achieve better results other enhancements migh be done. To provide an illustration, if for the first task we used the very same model to train both the baseline model and the model with curriculum learning approach integrated, in the second case the model used as for the training was not the one used by the authors in [33]. The same happened for the last task experimented. Given that, the negative results achieved might be happened for different reasons. Firstly, it might be that the *Baby Steps* curriculum approach used is not suitable for code-summarization-like tasks; or secondly, sorting the initial dataset following different levels of difficulties and shuffling the instances may not be enough for the tasks considered; nevertheless it might also be that the results obtained strictly depends on the kind of model used. As for the second and third tasks we decided to use the same neural network used for the first one - both in the baseline and in the model with curriculum learning; however, reproducing the experiments on the same model used in [33] and [36] may possibly bring different results. If that was the case, it would mean that curriculum learning - at the very least for the *Baby Steps* approach - would be a model-independent approach likely suitable for more than one deep learning model. In any case, reproducing code summarization and log generation tasks on the very same models used in the referenced papers with *Baby Steps* curriculum approach would be a first move towards possible future works. In fact, not only will this experiment answer some of the questions that our results lead to, but also it would make it possible to compare reliably the results obtained with curriculum learning to those obtained by the scholars on the baselines. Regardless to the results collected, we can very well acknowledge that curriculum learning outcomes highly depend on the training data and that the approach needs to be settled differently for each task.

Acknowledgements

Five years ago, when I decided to start studying to be part of the IT industry someday, I had no idea what was waiting for me. I did not get into computer science before starting university, and because of that I wanted to ask Prof. Rocco Oliveto - which at that time already was the chair of the bachelor program - for his advice in this regard. Even though I did not know the why, talking to him I immediately felt the path I was about to choose was right for me. I trusted him and I trusted my instincts.
Today, after five years - one of which I studied abroad - I think my room in the world is starting to rise. Never have I ever felt more confident on the decision I made. Yet I am a person who more often than needed struggles with self-confidence.

Just for what has been said so far, and it is not all, I owe Prof. Rocco Oliveto - my advisor and one of the most inspiring teachers I have encountered in my course of study - a huge thank you. Not only was he the reason why I started my journey in one of the most impactful and topic-broad field, but he also gave me a lot of opportunities to grow educationally and personally trough both his academic lectures and life-lessons. Thank you for supporting me during the last years and for being part of the professional future I just started to create.

I would also like to thank Prof. Rocco Oliveto, Prof. Gabriele Bavota, and all the UNIMOL and USI staff who gave me the paramount chance to achieve the double degree. Spending the last year of my Masters at Università della Svizzera Italiana, was pivotal. Even though it was truly tough, it came at a moment in my life when I needed it. Experiencing that extremely challenging and demanding period was genuinely essential to me to understand what really matters not only in career, but also in life. Thank you for the unique opportunity and impeccable backing.

I thank Prof. Gabriele Bavota who since the very first moment I met him made himself available for any kind of support. Thank you for not missing opportunity to remind me that there was someone on the other side who believed in me. Also, thank you for giving me the chance of working on the master thesis with your advising. I learned a lot.

This piece of work would not have been possible without the help of Antonio Mastropaolo. Thank you for all your time and your words of advice, I will not foget them.

I owe a huge thank you to Tania Clarke. Not only is she the best English teacher I have ever had, but for the past two years she has been a very important figure for me. A source of knowledge of a language that I have always wanted to perfect and learn more about. Thank you for always being supportive. Studying aside, your letting me talk and your simple and direct advice helped me.

There are no words to thank my family, source of unconditional love and endless support. The only thing in the whole world which I can not not help with; the source of strength, courage, foresight, wisdom; the reason why I am who I am. Thank you mom and dad for have gave me the right discipline; thanks for being my pillars, empathic objectivity on one side and hopeful rationality on the other. Thank you Ugo and Vito, brothers as annoying as they are attentive and loving. Thank you for always being by my side. What would I do without you?

My abroad experience would have not been the same if I had not met Ottavio, Daniel, Gianlorenzo, Federico, and Isabella. Thanks for all the sharing views we talked about, the fun we had together, and the memories we collected. I feel very grateful to have met you. You now take up a part of my heart.

Special thanks go to Martina, Antonio, and Silvia. You could see how I really am and I truly feel extremely grateful for standing there for me, and not only when I really needed it. I will never forget your constant support, that has emerged a little more in the past year, but that was just a testament to a solid and sincere frienship. I am glad of having you on my side and I will always be by yours. You can bet.

Thank you all.

Bibliography

[1] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs.(2018). 2018.

[3] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876, 2021.

[4] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[5] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[6] F. Barchi, E. Parisi, A. Bartolini, and A. Acquaviva. Deep learning approaches to source code analysis for optimization of heterogeneous systems: Recent results, challenges and opportunities. *Journal of Low Power Electronics and Applications*, 12(3):37, 2022.

[7] 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.

[8] S. Bhatia and R. Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*, 2016.

[9] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 201–211, 2020.

[10] N. D. Bui, L. Jiang, and Y. Yu. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[11] J. Burstein, C. Doran, and T. Solorio. Proceedings of the 2019 conference of the north american chapter of the association for computational linguistics: Human language technologies, volume 1 (long and short papers). In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019.

[12] X. Chen and A. Gupta. Webly supervised learning of convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1431–1439, 2015.

[13] V. Cirik, E. Hovy, and L.-P. Morency. Visualizing and understanding curriculum learning for long short-term memory networks. *arXiv preprint arXiv:1611.06204*, 2016.

[14] 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.

[15] R. El-Bouri, D. Eyre, P. Watkinson, T. Zhu, and D. Clifton. Student-teacher curriculum learning via reinforcement learning: predicting hospital inpatient admission location. In *International Conference on Machine Learning*, pages 2848–2857. PMLR, 2020.

[16] J. L. Elman. Learning and development in neural networks: the importance of starting small. *Cognition*, 48(1):71–99, 1993.

[17] Y. Fan, F. Tian, T. Qin, X.-Y. Li, and T.-Y. Liu. Learning to teach. *arXiv preprint arXiv:1805.03643*, 2018.

[18] C. Florensa, D. Held, M. Wulfmeier, M. Zhang, and P. Abbeel. Reverse curriculum generation for reinforcement learning. In *Conference on robot learning*, pages 482–495. PMLR, 2017.

[19] C. Gong, J. Yang, and D. Tao. Multi-modal curriculum learning over graphs. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(4):1–25, 2019.

[20] A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu. Automated curriculum learning for neural networks. In *international conference on machine learning*, pages 1311–1320. PMLR, 2017.

[21] I. Grigorik. The github archive. *URL: https://githubarchive. org*, 2012.

[22] S. Guo, W. Huang, H. Zhang, C. Zhuang, D. Dong, M. R. Scott, and D. Huang. Curriculumnet: Weakly supervised learning from large-scale web images. In *Proceedings of the European conference on computer vision (ECCV)*, pages 135–150, 2018.

[23] G. Hacohen and D. Weinshall. On the power of curriculum learning in training deep networks. In *International Conference on Machine Learning*, pages 2535–2544. PMLR, 2019.

[24] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255, 2020.

[25] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[26] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. in 2018 ieee/acm 26th international conference on program comprehension (icpc). *IEEE, 200ś20010*, 2018.

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

[28] L. Jiang, D. Meng, T. Mitamura, and A. G. Hauptmann. Easy samples first: Self-paced reranking for zero-example multimedia search. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 547–556, 2014.

[29] T. Kocmi and O. Bojar. Curriculum learning and minibatch bucketing in neural machine translation. *arXiv preprint arXiv:1707.09533*, 2017.

[30] K. A. Krueger and P. Dayan. Flexible shaping: How learning in small steps helps. *Cognition*, 110(3):380–394, 2009.

[31] M. Kumar, B. Packer, and D. Koller. Self-paced learning for latent variable models. *Advances in neural information processing systems*, 23, 2010.

[32] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE, 2016.

[33] A. LeClair, S. Haque, L. Wu, and C. McMillan. Improved code summarization via a graph neural network. *CoRR*, abs/2004.02843, 2020.

[34] A. LeClair, S. Jiang, and C. McMillan. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE, 2019.

[35] 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.

[36] A. Mastropaolo, L. Pascarella, and G. Bavota. Using deep learning to generate complete log statements. *CoRR*, abs/2201.04837, 2022.

[37] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman. Teacher–student curriculum learning. *IEEE transactions on neural networks and learning systems*, 31(9):3732–3740, 2019.

[38] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*, pages 4505–4515. PMLR, 2019.

[39] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[40] S. Narvekar, J. Sinapov, and P. Stone. Autonomous task sequencing for customized curriculum design in reinforcement learning. In *IJCAI*, pages 2536–2542, 2017.

[41] E. A. Platanios, O. Stretcu, G. Neubig, B. Poczos, and T. M. Mitchell. Competence-based curriculum learning for neural machine translation. *arXiv preprint arXiv:1903.09848*, 2019.

[42] M. Pradel, G. Gousios, J. Liu, and S. Chandra. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 209–220, 2020.

[43] M. Qu, J. Tang, and J. Han. Curriculum learning for heterogeneous star network embedding via deep reinforcement learning. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 468–476, 2018.

[44] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[45] Z. Ren, D. Dong, H. Li, and C. Chen. Self-paced prioritized curriculum learning with coverage penalty in deep reinforcement learning. *IEEE transactions on neural networks and learning systems*, 29(6):2216–2226, 2018.

[46] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE, 2018.

[47] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[48] Y. Tay, S. Wang, L. A. Tuan, J. Fu, M. C. Phan, X. Yuan, J. Rao, S. C. Hui, and A. Zhang. Simple and effective curriculum pointer-generator networks for reading comprehension over long narratives. *arXiv preprint arXiv:1905.10847*, 2019.

[49] 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.

[50] X. Wang, Y. Chen, and W. Zhu. A comprehensive survey on curriculum learning. *CoRR*, abs/2010.13166, 2020.

[51] X. Wang, Y. Chen, and W. Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[52] D. Weinshall, G. Cohen, and D. Amir. Curriculum learning by transfer learning: Theory and experiments with deep networks. In *International Conference on Machine Learning*, pages 5238–5246. PMLR, 2018.

[53] B. Xu, L. Zhang, Z. Mao, Q. Wang, H. Xie, and Y. Zhang. Curriculum learning for natural language understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6095–6104, 2020.

[54] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823*, 2018.

[55] T. Yin. Lizard.

[56] W. Zaremba and I. Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.