TITLE
...
...

Carmen Armenti

October 2022

Supervised by
Prof. Rocco Oliveto

Co-Supervised by

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Application context

1.2 Motivations and Objectives

1.3 Results

1.4 Document Structure

This document is organized as follows:

- In Chapter 2,
- In Chapter 3,
- In Chapter 4,
- In Chapter 5,

Chapter 2

Deep Learning and Curriculum Learning

Inspired by human learning, curriculum learning is an algorithm 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* [7].
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* [1] 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.

2.1 State of the art

2.2 Curriculum Learning related works

2.3 Curriculum Learning application in Deep Learning tasks

Chapter 3

Deep Learning Applications to Software Engineering tasks

3.1 State of the art

3.2 Software Engineering related works

3.2.1 Bug fixing task

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

3.2.2 Code summarization task

3.2.3 Log generation task

3.3

Chapter 4

Evaluating Curriculum Learning in Software Engineering tasks

In a nutshell, curriculum learning means "training from easier data to harder data" [7]. More specifically the core idea is to "start small" [2], 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 [7]. Therefore, Difficulty Measurer together with Training Scheduler constitute a general framework for curriculum design, as illustrated in Figure 4.1. First of all, the Difficulty

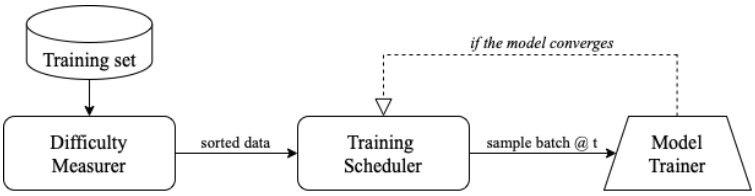

FIGURE 4.1: Predefined Curriculum Design.

Measurer sorts all the training examples from the easiest to the hardest and passes them to the Training Scheduler. Then, at each training epoch t, the Training Scheduler samples a batch of training instances from the easier subset and give them to Model Trainer for training.
As training epochs increase, the Scheduler decide qhen to sample from more harder data, generally until uniform sampling from the whole training set. This schedule either depends on the training loss feedback from the Model Trainer, or on some other parameters that implies that the model would deverge, if left to training for more epochs.
A distinction between **predefined CL** and **automatic CL** must be clarified. The first refers to the framework where both the Difficulty Measurer and Training Scheduler are defined by human prior knowledge, thus with no data-driven algorithms involved; the latter instead, if any - or both - of the components are designed by data-driven algorithms.
Usually, the power of 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 complexity, diversity, or noise estimation definitions.

We adopted the most popular discrete scheduler, known as *Baby Step*, where the complexity of the training data needs to be gradually increasing. Due to this a reliable metric to split the initial dataset of

each task needed to be defined. This approach distributes the sorted data into buckets, from easy to hard, according to the metric and starts training with the easiest bucket. Starting from the easiest bucket, after a fixed number of training epochs or convergence, the subsequent bucket is merged into the current training subset - main characteristic of *Baby Step* approach. Finally, once all the buckets are merged and used, the training process either stops or continues several extra epochs.
Note that, at each epoch the scheduler shuffles the current bucket and samples mini-batches for training.

However, before testing curriculum learning approach, we 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.

In this chapter, we present how we applied Curriculum Learning to some of Software Engineering tasks, i.e. *bug-fixing*, *code summarization*, and *log generation* tasks.
For each of the tasks considered, the following aspects are described thoroughly:

- **Dataset characteristics**: the datasets used are composed by a couple, where the first element is the source of the training, while the second is the target;
- **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 Neural Machine Translation

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

- a RNN Encoder, which encodes a sequence of terms **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: $P(y_1, ..., y_m | x_1, ..., x_n)$, 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. The configuration used by the neural network is composed by 1-layer bidirectional Encoder, 2-layer Attention Decoder both with 256 units, embedding size of 512, and LSTM RNN cells. Bucketing and padding was used to deal with the variable length of the sequences.

4.2 Canonical Training

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

4.2.1 Bug-fixing

As for the bug-fixing task, the datasets used to train the NMT model is the union between *small* and *medium* method-level datasets used by Tufano *et al.* [6]. 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 [reference prese da 2.3.3 michele tufano]. The key idea is that the decoding process keeps track of k hypotheses - being k the beam size or width.
The beam search algorithm selects multiple tokens for a position in a given sequence based on conditional probability. Moreover, the algorithm can take any number of N best alternatives through a hyperparameter known as beam size or width indeed. Conversely to what greedy search does, Beam search broad the search to include other words that might fit better apart from the best word for each position in the sequence. If Greedy search looks at each position in the output sequence in isolation, deciding the word based on highest probability and then moving down to the resto of the sentence, Beam search also takes the N best output sequences and look at the current preceding words and probabilities compared to the current position that is being decoded in the sequence.
We considered the following sizes: 1 (that corresponds to the Greedy Naïve Approach), 5, 10, 15, 20, 25, 30, 35, 40, 45, 50.

4.2.2 Code summarization

Regarding the task of code summarization, we selected a random sample of 200k instances from a dataset of 2 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 correspond to 125.000 steps, and we took the best model over the 20 model configuration evaluated after every epoch, thus using the best checkpoint to run the inference step. This time, to guide the selection of the best configuration, we used the BLEU values computed on the validation set instead of using the loss function. Conversely, the results are computed on the test set. Indeed, the same way it was done for the previous task, after the training step the model is evaluated over the test set of unseen functions to generate code summarizations of such methods.

4.2.3 Log generation

As well as happened in both the previous task experimented, also in log generation we considered the well-known split between training set, test set, and evaluation set. The sets in question are - following the same order - of the amount of 105.985, 12.020, and 13.260. The model was trained for 45 epochs, i.e. 149.625 steps, and it must be specified that it was evaluated every epoch. The best configuration in this case was guided on an early-stopping criterion based on BLEU values. Once the model was trained, it was evaluated against the test set of unseen methods without log statements and messages in order to assess its ability to correctly inject log statements and messages within methods.

4.3 Training with Curriculum Learning

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

4.3.1 Bug fixing task

In this section the way how the bug fixing task is applied in the CL approach is presented.
Dataset Details. The reason for this specific choice was firstly to avoid any type of context dependance, secondly to achive a reasonable number of training instances.

```
public void METHOD_1 ( TYPE_1 VAR_1 ) { VAR_2 . METHOD_2 ( VAR_3 ) ; ( VAR_4 ) ++ ;
METHOD_3 ( ) ; }
```

DATA 4.1: Buggy code

```
public void METHOD_1 ( TYPE_1 VAR_1 ) { if ( VAR_2 . METHOD_2 ( VAR_3 ) ) { ( VAR_4 )
 ++ ; METHOD_3 ( ) ; } }
```

DATA 4.2: Fixed code

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 [4]. It is a string metric for measuring difference between two sequences, specifically the number of insertions, deletions, and substitutions required to transform one string into the other. However, the distance we used for our experiments was token based, thus the granularity is word based instead of being single-character based. Defined the measure, the distance between buggy and fixed method for each of the BFPs was computed

and we observed the data distribution; then we used the quartiles' values to divide the initial dataset in multiples smaller datasets, where each of these represents a different level of difficulty.
By doing so, we clearly obtained 4 level of difficulty; thus we defined an incremental difficulty criteria to feed the model with: the bug-fixing training scheduler described as follows.
Training Scheduler. The training scheduler decides the sequence of data subsets throughout the training process based on the judgment from the difficulty measurer. As mentioned in the introduction of this chapter, Baby Step approach switches to the next difficulty level as soon as the model converges on the previous one.
In the bug-fixing task, the scheduler adjusts the training data subsets based on an early stopping criterion: after a fixed number of steps early stopping is run and it considers the model as being converging if the loss value does not improve after 5K steps.

To guide the selection of the best configuration, we used the loss function computed on the validation set and not on the test set, while the results are computed on the test set.

4.3.2 Code summarization task

Automatic source code summarization is the task of generating short natural language description for source code [3]. The idea is that a brief description allows programmers to understand what a chunk of code does and what is the purpose of the program by and large, without necessarily read the code itself.
In this section CL the application of Curriculum Learning is applied to code summarization task.
Dataset characteristics. The training dataset used to test this software engineering task, is a subset of the data set of 2.1 million Java method-comment pairs used by LeClair *et al* [3]. A random sample of 200K instances was picked from the initial dataset; evaluation and test datasets, however, were used as-is. 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.

```
protected String creatorName() {
    return Texts.getText("solver");
}
```

DATA 4.3: Function

```
//Gives the name of this solver as used to tag new solutions.
//@return the name of this solver
```

DATA 4.4: Comment

Difficulty Measurer. In Natural Language Processing tasks the **sentence lenght** intuitively expresses the complexity of a sentence. 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.
Training Scheduler. Similarly to what happens in bug-fixing task, the training scheduler decides to sample from more harder data only when the model converges on the previous easier bucket. The convergence is assessed after a fixed number of epochs, after which early stopping with patience of 5 epochs is run. Once the model converget on the first bucket, the training proceeds on the second and so forth.

4.3.3 Log generation task

Inserting log messages is a practice broadly used and decide where to inject log statements, what information to report through it, and at which log level is as hard as it is crucial [5]. In the following section Curriculum Learning is applied at log generation task.
Dataset characteristics. 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

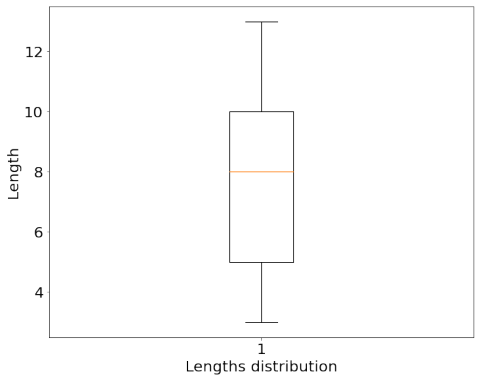

FIGURE 4.2: Code summarization task: data distribution.

message.
Those couples where used to train the model to generate and inject log statements in Java code.

```
public CsvDestination setPath (
final String path )
{ if ( csvFile != null )
{ throw new
UnsupportedOperationException ( ""
Changing the value of path after
opening the destination is not
allowed."" ) ; }
if ( outputChannel != null )
{ try
{ outputChannel . close ( ) ;
outputChannel = null ; }
catch ( final IOException e ) { } }
this . path = path ;
return this ; }
```

DATA 4.5: Method

```
public CsvDestination setPath (
final String path )
{ if ( csvFile != null )
{ throw new
UnsupportedOperationException ( ""
Changing the value of path after
opening the destination is not
allowed."" ) ; }
if ( outputChannel != null )
{ try
{ outputChannel . close ( ) ;
outputChannel = null ; }
catch ( final IOException e )
{ log . error ( String . format ( "
"Could not close file channel with
CSV results for file %s."" ,
csvFile ) , e ) ; } }
this . path = path ;
return this ; }
```

DATA 4.6: Method + log statement

Difficulty Measurer. Given the dataset at our dispose we choose as difficulty measurer the complexity of the instruction set in program execution tasks. More specifically we computed **cyclomatic complexity** through Lizard tool [8]. Cyclomatic complexity is a quantitative software metric developed by Thomas J. McCabe in 1976 used to indicate the complexity of a program. It measures the number of linearly-independent paths through a program module. The measure is computed using the control-flow graph of the program where the nodes of the graph correspond to sets of commands of a program, and a directed edge connects 2 nodes if the second command might be executed immediately after the first command. It can be applied to individual functions, modules, methods or classes within a program. We decided to compute it on each of the source methods.
It must be said that from the initial dataset were removed 397 instances whose cyclomatic complexity was 0. Decided and computed the difficulty measure for each instance, the dataset was ready to be used by the training scheduler.
Training Scheduler. Once again we observed the data distribution and considered the 3 quartiles to divide the whole dataset in 4 subsets. Each of those represented a differen level of difficulty.

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

Chapter 5

Analysis of results

5.1 Bug fixing task

5.2 Code summarization task

5.3 Log generation task

Chapter 6

Conclusion

Appendix A

Appendix

Bibliography

[1] 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.

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

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

[4] 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.

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

[6] 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.

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

[8] T. Yin.