A Grammar-based Evolutionary Approach for Assessing Deep Neural Source Code Classifiers

Martina Saletta, Claudio Ferretti
Dept. of Informatics, Systems and Communication
University of Milano-Bicocca
Milan, Italy
martina.saletta@unimib.it, claudio.ferretti@unimib.it

***Abstract*—Neural networks for source code processing have proven to be effective for solving multiple tasks, such as locating bugs or detecting vulnerabilities.**

In this paper, we propose an evolutionary approach for probing the behaviour of a deep neural source code classifier by generating instances that sample its input space.

First, we apply a grammar-based genetic algorithm for evolving Python functions that minimise or maximise the probability of a function to be in a certain class, and we also produce programs that yield an output near to the classification threshold, namely for which the network does not express a clear classification preference.

We then use such sets of evolved programs as initial populations for an evolution strategy approach in which we apply, by following different policies, constrained small mutations to the individuals, so to both explore the decision boundary of the network and to identify the features that most contribute to a particular prediction.

We furtherly point out how our approach can be effectively used for several tasks in the scope of the interpretable machine learning, such as for producing adversarial examples able to deceive a network, for identifying the most salient features, and further for characterising the abstract concepts learned by a neural model.

***Index Terms*—structured grammatical evolution; evolution strategy; decision boundaries; deep neural networks; source code classifiers**

I. Introduction

In recent years, deep neural networks (DNNs), and especially neural classifiers have become popular and successful for applications in a wide range of domains. Therefore, being able to understand *how* such models make their predictions is of interest, especially with their internal architectures becoming complex and sophisticated.

Ample research studied how to generate adversarial examples for networks which classify images, that is pictures that lead them to wrong predictions. Conceptually, input images can be considered as points in the domain of the function that maps each input to the corresponding classification error, and thus it is possible to move from a correctly classified input to an adversarial example by examining, and following, the gradient of such function [1].

Problems arise when this approach has to be applied to domains where instances, to become input of the neural network, need to be transformed to numerical vectors. This is the case of source code, which can be given as input to neural networks only after being transformed by one of many available embedding methods. In fact, in such a case we can apply known methods for moving on the error function only with respect to the embedding input space. That is, we can for instance find the embedding vector of adversarial examples, but we would need to find a way to build the corresponding adversarial source code [2].

We want to approach this problem by exploring how the output of the network changes, when input source code changes. The output to be considered will not be the categorical classification, binary in our experiments, but the real valued probability that most neural models compute for each possible classification outcome. Also, the exploration will be aimed at finding specific values of the probability, namely the maximal 1, the minimal 0, or the 0.5 value, which in the case of binary classification corresponds to the decision boundary of the network.

Our proposal is to study how the solution space can be explored for finding source code snippets that produce the wanted probability values, and also to find which small perturbations quickly change the output values. This will allow to look for specific features, in the source code, which show strong correlation to the changes in the network output, in a way similar to what is done when studying the adversarial space for numerical input domains.

Our proposal is to efficiently move in the source code input space by using evolutionary methods, to reach the needed network output values. In our method, individuals are source code snippets, fitness is the probability value produced by the neural classifier, and the genomes are evolved as defined by a grammar based evolutionary algorithm.

The overall mechanism involves a first phase generating individuals optimal with respect to the fitness value we look for, and a second phase deriving from them other instances which allow us to explore how – and sensitive to which features – fitness changes around the reached values.

Our contribution is the design of such specific variation on the evolutionary method, good to search the space of source code instances. We demonstrate the results of its use on a state of the art neural classifier. We always obtain source code snippets inducing the required output from the neural network, and we also show how our system easily finds sets of derived snippets, which describe how the classification is changing in a given area of the input space.

II. Related work

Essentially, this paper proposes an evolutionary approach for studying the behaviour of deep neural models in the source code processing domain with bearing on many issues, such as identifying blind spots and salient features, analysing the decision boundary or deceiving a model by generating adversarial examples. To this end, this section provides a concise literature overview on the main topics that concern our work, focusing on the context we are interested in.

A. Machine learning for source code analysis

The recent literature shows how machine learning models dealing with source code are becoming widespread and effective for solving multiple tasks, including code classification [3], code summarization [4], vulnerability detection [5] or for addressing classical software engineering issues [6], such as code completion [7], error fixing [8] or bug location [9].

Besides the use of networks that need as input specific program representations (e.g. vectors [10] or graphs [11]), also the models known as transformers [12], widely used for natural language processing (NLP) applications are becoming popular in the field of source code processing. For instance, we can mention CuBERT [13], a source code-specific BERT [14] model and PLBART [15], a sequence-to-sequence model used for different program comprehension tasks such as code summarization and generation. One of the advantages in the use of such kind of networks is their flexibility: in fact, they can be trained once on generic and large corpora of data and then fine-tuned for different and more specific tasks. In the rest work, we will refer to CuBERT as the baseline model for our discussion.

B. Formal grammars and evolutionary algorithms

Grammatical evolution (GE) [16] is an evolutionary algorithm designed to evolve individuals whose phenotypes are compliant with a given formal grammar. Since programming languages are often specified by a context-free grammar (CFG), this technique is particularly suitable when dealing with source code.

For its inherent nature, GE has been applied for addressing the critical problem of program synthesis [17]. For instance, the authors of [18] show how GE can be effectively used for evolving programs belonging to different domains, although pointing out that the knowledge of the problem to solve is crucial for designing the suitable grammar, while [19] applies GE in the synthesis of programs that solve the problem of sorting sequences of integers.

Despite the insightful results obtained, GE suffers from problems such as redundancy (many genotypes are mapped into the same phenotype) and locality (a small perturbation in a genotype leads to significant changes in the corresponding phenotype) in the genome representation [20]. In order to face these drawbacks, adjustments on GE have been proposed: structured grammatical evolution (SGE) [21] and its improved dynamic version (DSGE) [20] have proven to outperform GE in solving the problems on which they have been compared. Therefore, due to the significant reduction in terms of locality and redundancy, in our experiments we apply DSGE for exploring the solution space of CuBERT.

C. Explaining machine learning models

Machine learning models based on neural networks classify input through a non linear mapping of input instances to output class probabilities. The mapping function is shaped by the trained weights of internal connections, and classification errors arise when this function has areas of input space which generate unexpected outputs.

In the literature, the goal of understanding the overall input output behavior of a trained network, or just of discovering input areas where the model delivers wrong predictions, are pursued by estimating the shape either of the manifold corresponding to the learnt function or of the error function.

In [1] the input space of a network in the regions around known positive instances is studied to look for adversarial examples, while authors of [22] look for adversarial example images against which a defense is harder, by searching input space for points strategically distant from decision boundaries. Also working on images as input, [2] develops a method to generate borderline instances, and also discusses how to generate instances moving far from decision boundaries to explore how the classification behavior of the network changes around that space.

In this work, we sample the input space by evolving source code snippets leading to chosen output values, and then we study how the output varies in their neighborhood, explored by means of small syntactical perturbations.

III. Proposed Method

In this section we detail how we will build and test our method. Essentially, we first choose a neural network classifying source code, then we use its output as the fitness evaluator for an evolutionary algorithm, and finally we look for input programs leading it to output specific class probabilities. A similar approach has been already adopted in [23] to evolve features to be injected in given program instances to produce adversarial examples able to deceive a neural vulnerability detector.

In the literature, classifiers have been developed to predict whether input source code satisfies some given quality properties, related for instance to good software engineering practices or to avoid security flaws. We will use a well known and publicly available neural classifier, based on the transformer architecture, trained to check some properties of Python source code snippets. Another publicly available software platform will be adapted in order to evolve individuals defined through a formal grammar in accordance to the two phase process we are proposing. On top of these software systems, it will be required to integrate them, to make available to the evolutionary system the class probability computed by the neural network.

The behavior of the neural system will be probed by source code snippets evolved by using a simplified grammar, but they will vary enough to discover which code features mostly affect the classification results.

A. *CuBERT source code classifier*

Our benchmark source code classifier, CuBERT [13], is basically a BERT model [14] (the popular transformer for NLP), modified to effectively deal with source code. For our experiments, we considered three of the original fine-tuned classifiers proposed by the authors, to make our results and investigation more robust with respect to the possible bias induced by the single accounted downstream task. In particular, we replicated all our experiments on three fine-tuned models[1] trained on the following binary classification tasks:

1) **Variable misuse:** this task is referred to the mistakes developers could make when dealing with similar code fragments or similar variable names (e.g. when copy-paste code snippets but forget to properly renaming variables) [11]. In this fine-tuned model, it consists in detecting if there is a variable misuse at any location in the Python function given as input.
2) **Wrong binary operator:** this task [24] simply consists in detecting weather there is a binary operator that is improperly used in an expression occurring in the function given as input.
3) **Swapped operands:** this task consists in detecting if there are operands of non-commutative binary operators that are swapped with respect to the correct usage intention.

Our focus, in this work, is to study how different syntactical features most influence the evaluation of the different models, under the hypothesis that the relevance of such features is related to the problem to solve. We remark that, to this end, we are mostly interested in the numerical variation of the output instead of the prediction in terms of classification decision. This is due to the fact that a ground truth for the described problems is difficult to establish for programs that do not have a predetermined functionality, and thus we will not focus on the creation of adversarial examples, but only on evolving programs that lead to arbitrary predictions, no matter their *correct* label.

B. *Two-stage evolutionary search*

In our approach, the exploration of the CuBERT behaviour is performed in two main phases: in the first one, we sample the CuBERT solution space by applying pure DSGE using the simplified Python grammar reported in Figure 1 and different fitness objectives; in the second one, we apply only "constrained" mutations to such evolved individuals, to point out how such mutations affect the fitness.

[1]https://github.com/google-research/google-research/tree/master/cubert

```
<start>          ::= def <funid>(x, y):
                     {:<statements>:}
<funid>          ::= funid | sum_xy | sub_xy
<statements>     ::= <statement>\n
                     | <statement>\n<statements>
<statement>      ::= <varid>=<simpl-expr>
                     | return <simpl-expr>
                     | <if-stmt>
                     | while <atom><condoperator><atom>:
                       {:<statements>:}
<simpl-expr>     ::= <atom> | <atom><operator><atom>
<atom>           ::= <varid> | <int>
<if-stmt>        ::= if <atom><condoperator><atom>:
                     {:<statements>:}
                     | if <atom><condoperator><atom>:
                       {:<statements>:}
                       else:
                       {:<statements>:}
<int>            ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<varid>          ::= x | y | sum_ | sub_ | mult_ | mod_ | div_ | rem_
<operator>       ::= + | - | * | / | % | **
<condoperator>   ::= == | != | >= | <= | < | >
```

Fig. 1: Simplified Python grammar.

a) DSGE and neural fitness functions: in this first experimental phase of our approach, we apply an evolutionary algorithm based on DSGE for sampling the solution space of the fine-tuned CuBERT models described in Section III-A.

Basically, in DSGE, genotypes are represented by lists of ordered derivation steps of a given CFG: each genotype is a sequence $S_1, \ldots, S_n$, where for each $i \in \{1, \ldots, n\}$, S_i is referred to a non-terminal symbol of a context-free grammar $G = (N, T, A, P)$ where N and T are the sets of non-terminal and terminal symbols, respectively, $A \in N$ is the axiom, or the starting symbol of G, and P is the set of production rules. Each S_i is an ordered list of integers $r_1, \ldots, r_k$, where k is the number of times the i-th non-terminal is expanded, and the values correspond to the indices (in the grammar) of the applied expansion rules. For a complete explanation of DSGE, we refer the reader to [20].

In the three considered fine-tuned CuBERT models, which are binary classifiers, the output is a value $0 \leq v \leq 1$, and an individual is predicted to be in the negative class if $v \leq 0.5$ and in the positive class otherwise. Given this premise, using the CuBERT output as the fitness, for each fine-tuned model we evolve programs using DSGE pursuing three fitness objectives: maximization, minimization, and minimization of $|0.5 - v|$. The founding idea is to produce pools of individuals that the network classifies with an high confidence (namely the program instances whose fitness is very close to 0 or to 1) or for which the prediction is uncertain, that is the individuals whose fitness is close to the decision boundary 0.5.

In this experimental phase we evolve programs that comply with the grammar listed in Figure 1. The grammar formalizes an extremely minimal Python function definition in which two variables `x` and `y` are fixed parameters, the only control flow constructs are `if`, `if-else`, and `while` statements, and only assignments and operations on integers are valid instructions. We still remark that the choice of a so simplified Python grammar, which is made for readability reasons, allows us to effectively produce individuals yielding outputs covering all the interval between 0 and 1, and thus it suffices for our discussion. Finally, notice that since the interest here is to analyse the output changes by varying syntactical elements, the evolved programs are not supposed, in general, to be executable or meaningful.

b) Mutations and fitness variations: this second phase is aimed at exploring how the fitness of individuals changes when different mutations are applied, to identify which features are most salient for the network to make its prediction.

In DSGE, the standard mutation operator is defined, for a position $S_i = r_1, \ldots, r_k$ as a random change of an integer r_j into another valid integer r_j^* for that gene. In other words, a mutation is basically the selection of a different expansion option for a non-terminal symbol. It should be noted that, in general, r_j^* could be referred to a production rule that contains non-terminal symbols of the grammar. In that case, for each non-terminal random expansions rules are selected (and thus added to the genome) until all the non-terminals are resolved into terminal symbols. Given this definition, it is possible to specify a different mutation operator in which only some positions can be altered, meaning that only some non-terminals can be changed in their expansion rules.

By means of two constrained mutation operators (whose pseudo-code are reported in Algorithm 1), we explore how the fitness variation is related to the non-terminals for which the mutation is enabled, to identify which syntactical features are the most salient for the network prediction. To this end, we performed two families of experiments:

1) The exploration of the neighborhood of individuals. For this investigation, we applied mutations by following the procedure described by Algorithm 2 and then we measured the average fitness variation of individuals generated by the mutations.
2) Following a $(\mu + \lambda)$ evolution strategy approach [25] (i.e. a population based algorithm in which the best μ individuals of a population are mutated λ/μ times for composing the new generation), we measure how the different constraints affect the fitness variation along a sequence of mutation steps. This procedure is formally described by Algorithm 3.

The core difference between the two experimental investigations is that, while in the first we perform an extensive exploration of the neighborhood of the individuals, in the second one a similar exploration is performed over many mutation steps guided by an evolutionary pressure and controlled by the fitness function. In the first case, the considerations that follow from the results, which will be fully reported and discussed in Section IV, are mainly quantitative: we simply observe how the different constraints in the mutation operator change the fitness evaluation of the mutated individuals when compared to the fitness of the original individual. In the second experiment, indeed, the attempt is more qualitative: we start from a single individual, we mutate (almost) a single token in each step, and we detect the features that most influence the prediction of the network, by observing how the fitness varies along many mutation steps.

Algorithm 1 Constrained mutation operators

```
1: ind ← individual to mutate
2: pmut ← mutation probability
3: $G$ ← list of genes $g_1, \ldots, g_n$
4:   ▷ each $g_i = v_1, \ldots, v_m$ is associated to a non-terminal,
       occurring $m$ times in the sentential form, and each $v_j$
       specifies the grammatical rule used in each position
5:
6:   ▷ The difference between the two operators is that,
       while in CNSTR_MUTATE_1 each mutable position is
       mutated with probability pmut, in CNSTR_MUTATE_2
       a mutation in a gene occurs with probability pmut and
       in that gene only one random position is mutated
7:
8: procedure CNSTR_MUTATE_1(ind, pmut, $G$)
9:     for all $g \in G$ do
10:        $V$ ← values in the genome for the gene $g$
11:        for all $v \in V$ do
12:            if RANDOM() $\leq$ pmut then
13:                mutate $v$ into another valid integer $v^*$
14:                if rule $v^*$ contains non-terminals then
15:                    randomly expand all the non-terminals
16:                end if
17:            end if
18:        end for
19:    end for
20:    return ind
21: end procedure
22:
23: procedure CNSTR_MUTATE_2(ind, pmut, $G$)
24:    for all $g \in G$ do
25:        if RANDOM() $\leq$ pmut then
26:            $V$ ← values in the genome for the gene $g$
27:            randomly choose $v \in V$
28:            mutate $v$ into another valid integer $v^*$
29:            if rule $v^*$ contains non-terminals then
30:                randomly expand all the non-terminals
31:            end if
32:        end if
33:    end for
34:    return ind
35: end procedure
```

IV. Results

In this section we supply the technical details of the performed investigations, and we discuss the obtained results. All the experiments have been performed on a Linux machine with 16GB RAM, 4 CPUs running at 3.60GHz and a Nvidia GTX 1070 GPU. We used tensorflow over CUDA, with sge3 implementation[2] of DSGE.

A. Sampling the solution space

As described in Section III-B, in the first experimental phase we apply DSGE for evolving Python programs using,

[2]https://github.com/nunolourenco/sge3

Algorithm 2 Neighborhood exploration

```
1: $P \leftarrow$ initial population
2: $G \leftarrow$ list of indices of the mutable genes
3: $n \leftarrow$ number of neighbours
4: $E \leftarrow$ empty list                                ▷ list of errors
5:
6: for all $p \in P$ do
7:     $f_p \leftarrow$ FITNESS($p$)
8:     for $i = 1, \ldots, n$ do
9:         $N \leftarrow$ empty list
10:        $newInd \leftarrow$ CNSTR_MUTATE_1($p, 0.5, G$)
11:        append $newInd$ to $N$
12:    end for
13:    for all $ind \in N$ do
14:        $f_{ind} \leftarrow$ FITNESS($ind$)
15:        append $|f_i - f_{ind}|$ to $E$
16:    end for
17: end for
18:
19: output the mean of the values in $E$
```

Algorithm 3 Fitness variation over mutation steps

```
1: $P \leftarrow p_1, \ldots, p_\lambda$                    ▷ initial population
2: $G \leftarrow$ list of mutable genes
3: $\mu \leftarrow$ number of parents
4: $n \leftarrow$ number of mutation steps
5:
6: for $n$ times do
7:     sort $P$ according to the fitness of each $p_i \in P$
8:     $B \leftarrow p_1, \ldots, p_\mu$                ▷ list of best $\mu$ individuals
9:     $P \leftarrow$ empty list
10:    for all $p \in B$ do
11:        append $p$ to $P$
12:        for $\lambda/\mu$ times do
13:            $m \leftarrow$ CNSTR_MUTATE_2($p, \frac{1}{|G|}, G$)
14:            append $m$ to $P$
15:        end for
16:    end for
17: end for
```

as the fitness function, the outputs of the three CuBERT fine-tuned models described in Section III-A, namely the three binary classifiers trained in detecting variable misuse, swapped operands and wrong binary operators.

For each of these models, we considered three fitness objectives (i.e. minimization, maximization and minimization of the distance between 0.5 and the output) and for each of these combinations we performed 10 DSGE runs by using as the reference grammar the one shown in Figure 1. For each run, we evolved populations of 50 individuals for 50 generations with tournament selection (3 individuals per tournament), keeping an elite of 10 individuals at each generation, and by letting crossover and mutation occur with probability 0.9 and 0.1, respectively. Also, we limited the size of individuals by imposing a maximum tree depth of 25.

The obtained results, which are fully reported in Figure 2, show how we are always able to pursue the fitness objective, even if using an extremely simple and minimal grammar. For each model, the targeted fitness that seems to require more generations to be reached is 0.5. This value is indeed the most interesting for the scope of this work, since it represents the decision boundary, that is the classification threshold of the network.

B. Moving across decision boundaries

In this and in the next subsection, we discuss how different syntactical features affect the prediction of the network. Specifically, we defined the two constrained mutation operators detailed in Algorithm 1, we applied them to the individuals evolved with DSGE by following different policies, and we observed the fitness variations.

The first investigation, formally outlined in Algorithm 2, is aimed at a quantitative assessment of how the different mutations affect the predictions of the three considered CuBERT models. For each model, we started from the individuals evolved in the previous phase when pursuing the fitness objective near the decision boundary. In other words, we started from a set of 10 evolved individuals for which the networks outputs a value very close to 0.5. Then, starting from each of individual, we generated 10 neighbours by applying the CNSTR_MUTATE_1 operator, by setting the mutation probability to 0.5 and by imposing three different constraints:

1) mutation allowed only for variable identifiers, that is the only mutable gene is that related to the `<varid>` non-terminal;
2) mutation allowed only for the arithmetic operators, that is the only mutable gene is that related to the `<operator>` non-terminal;
3) a less tied constraints set, in which mutation is allowed for the genes related to `<int>`, `<varid>`, `<operator>` and `<condoperator>` non-terminals. These constrains are labeled as "all" in Figure 4a.

Notice that all these constraints imposed for the mutation operator are related to non-terminal symbols whose expansion rules lead to terminal symbols, meaning that each mutation do not alter the syntactical structure but takes action only on the AST leaves.

We then measured, for each possible pair of constraints and task, the average fitness variation of the mutated individuals, as reported in Figure 4a. Figure 3 shows an example of this neighborhood exploration, along with a graphical representation of the mutation paths studied in the experiments described in Section IV-C.

C. Blind spots and salient features

This last set of experiments, formalized in Algorithm 3, lets the evolutionary machine free to explore how to improve the fitness of individuals, by changing at most one derivation from each occurring non-terminal. In our experiments, only

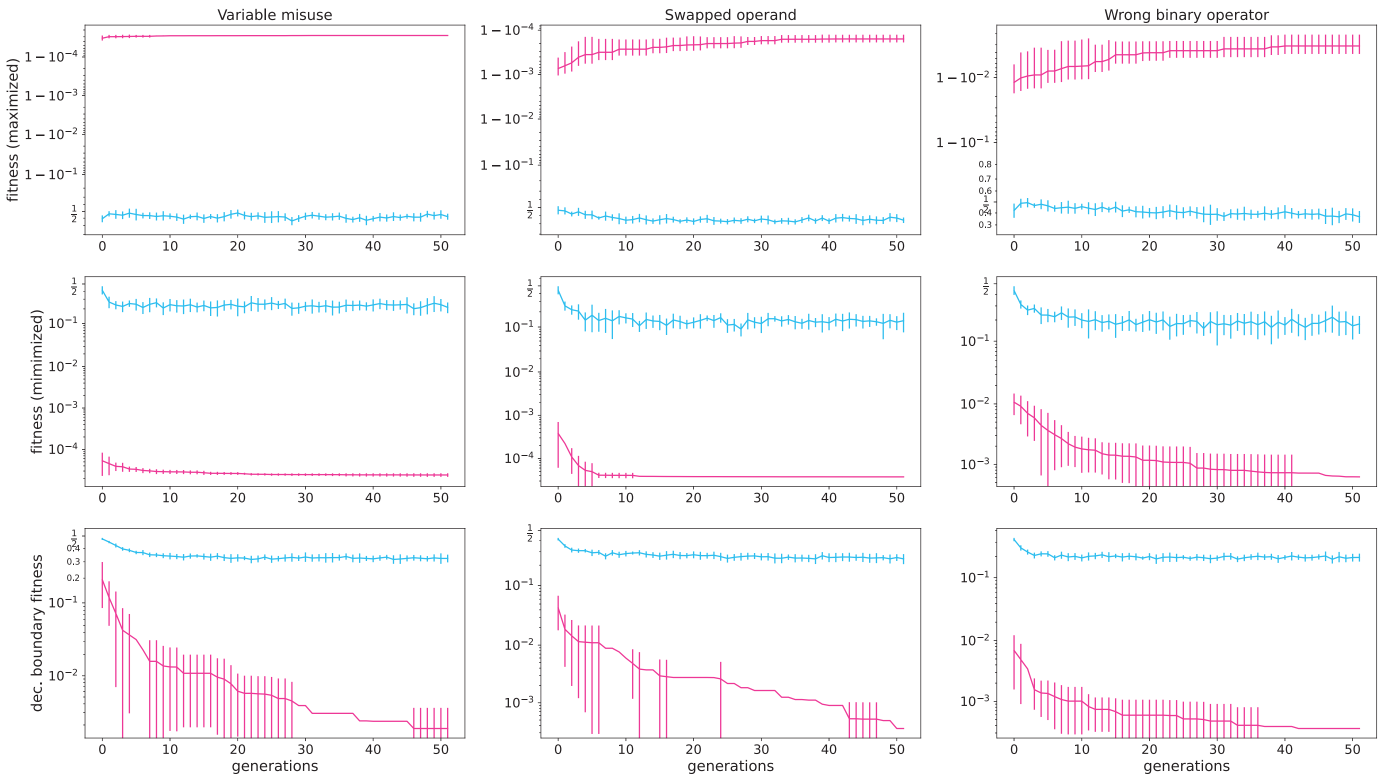
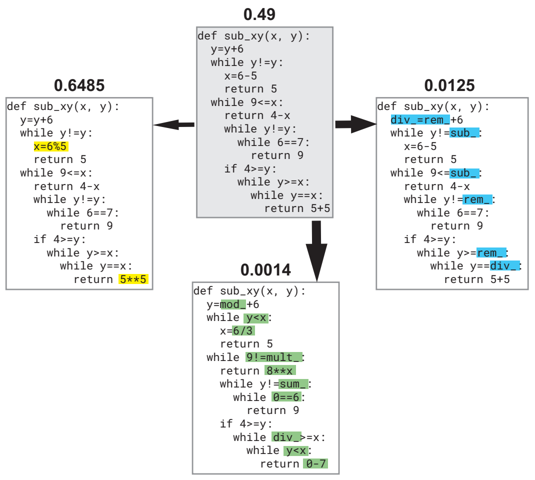


Fig. 2: Plots of the fitness obtained in with DSGE for the considered models and fitness objectives. Blue lines represent the average population fitness, purple lines the fitness of the best individual. Vertical bars report the variance in over the different runs.



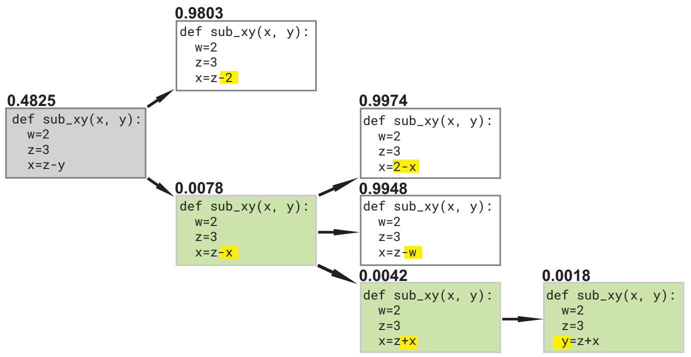
0.49
```
def sub_xy(x, y):
  y=y+6
  while y!=y:
    x=6-5
    return 5
  while 9<=x:
    return 4-x
    while y!=y:
      while 6==7:
        return 9
    if 4>=y:
      while y>=x:
        while y==x:
          return 5+5
```

0.6485
```
def sub_xy(x, y):
  y=y+6
  while y!=y:
    x=6%5
    return 5
  while 9<=x:
    return 4-x
    while y!=y:
      while 6==7:
        return 9
    if 4>=y:
      while y>=x:
        while y==x:
          return 5**5
```

0.0125
```
def sub_xy(x, y):
  div_=rem_+6
  while y!=sub_:
    x=6-5
    return 5
  while 9<=sub_:
    return 4-x
    while y!=rem_:
      while 6==7:
        return 9
    if 4>=y:
      while y>=rem_:
        while y==div_:
          return 5+5
```

0.0014
```
def sub_xy(x, y):
  y=mod_+6
  while y<x:
    x=6/3
    return 5
  while 9!=mult_:
    return 8**x
    while y!=sum_:
      while 0==6:
        return 9
    if 4>=y:
      while div_>=x:
        while y<x:
          return 0-7
```

(a) Neighborhood exploration: single mutation step where several mutations are allowed.



0.4825
```
def sub_xy(x, y):
  w=2
  z=3
  x=z-y
```

0.9803
```
def sub_xy(x, y):
  w=2
  z=3
  x=z-2
```

0.0078
```
def sub_xy(x, y):
  w=2
  z=3
  x=z-x
```

0.9974
```
def sub_xy(x, y):
  w=2
  z=3
  x=2-x
```

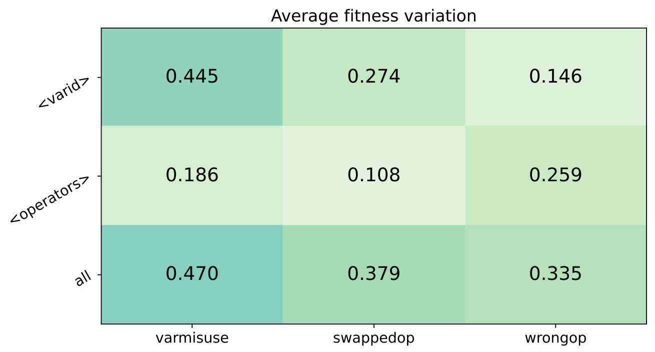
0.9948
```
def sub_xy(x, y):
  w=2
  z=3
  x=z-w
```

0.0042
```
def sub_xy(x, y):
  w=2
  z=3
  x=z+x
```

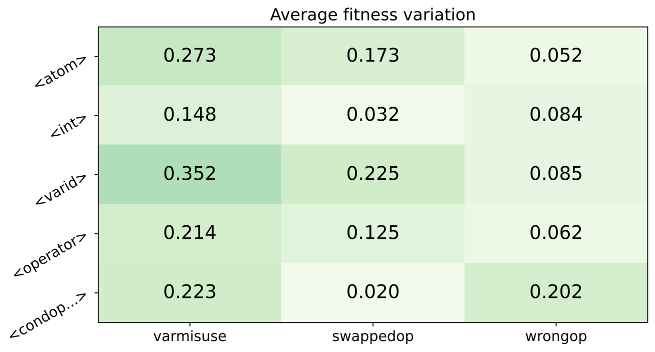
0.0018
```
def sub_xy(x, y):
  w=2
  z=3
  y=z+x
```

(b) Evolution strategy: several mutation steps with one mutation allowed for each step. Green backgrounds highlight a “winning” path.

Fig. 3: Examples for the mutation-based experimental phase. The numbers on top of the boxes represent the fitness value.



(a) Average fitness variation for the three fine-tuned models when different constraints on the mutation operator are imposed.



(b) Average fitness variation for the three fine-tuned models when several steps for mutation operator are possible.

Fig. 4: Results of the mutation-based experimental phase.

a chosen subset of non-terminals were mutable. Also, such exploration spanned a sequence of at most 5 mutation steps.

These choices aimed at assessing whether the neural network had developed specific sensitiveness for some syntactic elements.

In analogy to what can be done when exploring a geometrical space, we can consider that in our runs the evolutionary search moved from a parent individual to a new one by following a specific direction, here represented by a mutation involving a given grammar rule.

For instance, in our experiments we considered the grammar in Figure 1, and our evolutionary exploration moved from the individuals evolved in the first phase, where we aimed at receiving from the network a prediction very close to 0.5. Then, 5 generations of individuals were obtained by mutating their DSGE representation in terms of choices made when applying rules of non-terminals `<atom>`, `<int>`, `<varid>`, `<operator>`, and `<condoperator>`. The details of how such mutation operator works are given in CNSTR_MUTATE_2 of Algorithm 1. For each generation, best individuals have been selected as described in Algorithm 3, with $\lambda = 12$ and $\mu = 3$. The optimization goal of this phase, starting from individuals sited near the decision boundary, has been either to maximise or to minimise the fitness. This amounts to ask our system to find individuals that move far from the decision boundary and across the two sides it separates, always looking at the network's behavioural changes on the individuals belonging to the new generation.

We examined two main resulting sets of data: the fitness values that can be reached, and the grammar rules that are more effective in varying the fitness value. These data offer a view on the behavior of the network, for instance by telling us which syntactic elements the network is more sensitive to, and when they impact more than others on varying the fitness value, namely the probability that the network assigns to instances for belonging to a certain class.

The results of our evolutionary exploration of the input space, around instances classified close to the decision boundary, have been revealing with respect to the assessment of the neural network we tested, namely the CuBERT transformer. We could determine that:

- even when the evolution applied to the starting individuals did not modify their class with respect to the chosen tasks, as guaranteed by the set of changes we allowed in their derivation, our system could always derive individuals with fitness close to the desired value, 1 or 0; this means that we can find adversarial examples for the network, when in context where the ground truth is known;
- each task was a different challenge for our evolutionary system, for instance requiring more steps to reach the optimal fitness when checking for swapped operands, than for variable misuse, or forcing the evolution of longer individuals when aiming to fitness close to 0, than for the opposite goal;
- for each of the three classification tasks, the networks showed higher sensitiveness for specific sets of non-terminals (see Figure 4b), in details:
  - variable misuse: `<atom>` and `<varid>`;
  - swapped operator: `<atom>` and `<varid>`, with overall impact from the other non-terminals different from what happens with variable misuse;
  - wrong operator: `<condoperator>`.

The last remarks, on which non-terminals induced the fastest variation of fitness for a single evolutionary step, deserve some considerations, in relation to the understanding of the actual behavior of the network. In our experiments, we are seeing how the classifier reacts to small, and specific, variations in the input it receives. This information can be compared to what we expect from the trained network, with respect to the task at hand. In our experiment, for instance, we can see that even if we were allowed to evolve individuals by changing integer constants (being allowed to mutate the use of the `<int>` non-terminal), this was mostly not affecting classifications, and this seems correct. Also, having the network of the variable misuse task impacted by changes related to `<varid>` is expected. On the other hand, we discover that the network for the wrong operator task is more sensitive to changes in the choice of conditional operators, than in changes among arithmetical operators (`<condoperator>` VS `<operator>` non-terminals). And this could be somewhat unexpected; it could perhaps point to a bias in the training process.

An interesting visualization of what our system allows to describe, concerning how the classification of the network changes when choosing, guided by fitness, instances in the space around the border is shown in Figure 3b, where we can follow the generation of individuals with fitness starting from 0.5 and ending close to 0, during our second phase. Figure 4 shows indeed how different grammar rules act differently on fitness variations.

V. Conclusion and future work

With our grammar-based evolutionary approach, we are able to search the input space of a neural network looking for instances leading to arbitrary probability predictions. Also, we can explore their neighborhood and look for salient variations in the input-output mapping that characterizes the classifier. Our method has been tested on a state of the art source code neural classifier, namely the CuBERT transformer, allowing us to produce the input source code snippets we needed, and to identify which syntactical features of the source code mostly impact the classification. This way to probe the behavior of a network, in important input space areas, can be used to look for adversarial examples, but also to derive deeper information about the sensitiveness of the classifier with respect to features of input instances; the ease is to operate directly on source code and not necessarily on its vectorial representation, as other methods require.

To further this line of research, this approach can be applied to check the robustness of defense proposals in the area of adversarial attacks. A broader application area will be that of neural network understandability, for any neural model and also under a black-box approach, only assuming to have access to the predicted class probabilities. Searching the input space for instances located in key areas, with respect to the neural model decisions, could give insight to what the classifier is actually taking as key feature of the input, or to where it has blind spots or distorted evaluation of source code snippets.

References

[1] F. Tramèr, N. Papernot, I. J. Goodfellow, D. Boneh, and P. D. McDaniel, "The space of transferable adversarial examples," 2017. [Online]. Available: http://arxiv.org/abs/1704.03453

[2] H. Karimi, T. Derr, and J. Tang, "Characterizing the decision boundary of deep neural networks," 2019. [Online]. Available: http://arxiv.org/abs/1912.11460

[3] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016, pp. 1287–1293.

[4] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML*, 2016, pp. 2091–2100.

[5] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proceedings of 17th IEEE International Conference on Machine Learning and Applications, ICMLA*. IEEE, 2018, pp. 757–762.

[6] A. F. Del Carpio and L. B. Angarita, "Trends in software engineering processes using deep learning: A systematic literature review," in *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 445–454.

[7] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings of the 28th International Conference on Program Comprehension, ICPC*. ACM, 2020, pp. 37–47.

[8] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, S. P. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 1345–1351.

[9] X. Huo, F. Thung, M. Li, D. Lo, and S. Shi, "Deep transfer bug localization," *IEEE Trans. Software Eng.*, vol. 47, no. 7, pp. 1368–1380, 2021.

[10] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[11] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *Proceedings of 6th International Conference on Learning Representations, ICLR 2018*, 2018.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems (NIPS)*, 2017, pp. 5998–6008.

[13] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, ser. Proceedings of Machine Learning Research. PMLR, 2020.

[14] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[15] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668.

[16] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Trans. Evol. Comput.*, vol. 5, no. 4, pp. 349–358, 2001.

[17] D. Sobania and F. Rothlauf, "Challenges of program synthesis with grammatical evolution," in *Proceedings of Genetic Programming - 23rd European Conference (EuroGP), held as Part of EvoStar*, ser. Lecture Notes in Computer Science, vol. 12101. Springer, 2020, pp. 211–227.

[18] E. Hemberg, J. Kelly, and U. O'Reilly, "On domain knowledge and novelty to improve program synthesis performance with grammatical evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO*, A. Auger and T. Stützle, Eds. ACM, 2019, pp. 1039–1046.

[19] M. O'Neill, M. Nicolau, and A. Agapitos, "Experiments in program synthesis with grammatical evolution: A focus on integer sorting," in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC*. IEEE, 2014, pp. 1504–1511.

[20] N. Lourenço, F. Assunção, F. B. Pereira, E. Costa, and P. Machado, "Structured grammatical evolution: a dynamic approach," in *Handbook of Grammatical Evolution*. Springer, 2018, pp. 137–161.

[21] N. Lourenço, F. B. Pereira, and E. Costa, "Unveiling the properties of structured grammatical evolution," *Genetic Programming and Evolvable Machines*, vol. 17, no. 3, pp. 251–289, 2016.

[22] W. He, B. Li, and D. Song, "Decision boundary analysis of adversarial examples," in *6th International Conference on Learning Representations, ICLR*. OpenReview.net, 2018.

[23] C. Ferretti and M. Saletta, "Deceiving neural source code classifiers: finding adversarial examples with grammatical evolution," in *GECCO '21: Genetic and Evolutionary Computation Conference, Companion Volume*. ACM, 2021, pp. 1889–1897.

[24] M. Pradel and K. Sen, "Deepbugs: a learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018.

[25] S. Luke, *Essentials of Metaheuristics*, 2nd ed. Lulu, 2013, available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.