AutoFocus: Interpreting Attention-based Neural Networks by Code Perturbation

Conference or Workshop Item

How to cite:

For guidance on citations see FAQs.

© 2019 ACM; 2019 IEEE

https://creativecommons.org/licenses/by-nc-nd/4.0/

Version: Accepted Manuscript

oro.open.ac.uk
AutoFocus: Interpreting Attention-based Neural Networks by Code Perturbation

Nghi D. Q. Bui  
School of Information Systems  
Singapore Management University  
dqnbui.2016@phdis.smu.edu.sg

Yijun Yu  
School of Computing & Communications  
The Open University, Milton Keynes, UK  
y.yu@open.ac.uk

Lingxiao Jiang  
School of Information Systems  
Singapore Management University  
lxjiang@smu.edu.sg

Abstract—Despite being adopted in software engineering tasks, deep neural networks are treated mostly as a black box due to the difficulty in interpreting how the networks infer the outputs from the inputs. To address this problem, we propose AutoFocus, an automated approach for rating and visualizing the importance of input elements based on their effects on the outputs of the networks. The approach is built on our hypotheses that (1) attention mechanisms incorporated into neural networks can generate discriminative scores for various input elements and (2) the discriminative scores reflect the effects of input elements on the outputs of the networks. This paper verifies the hypotheses by applying AutoFocus on the task of algorithm classification (i.e., given a program source code as input, determine the algorithm implemented by the program). AutoFocus identifies and perturbs code elements in a program systematically, and quantifies the effects of the perturbed elements on the network’s classification results. Based on evaluation on more than 1000 programs for 10 different sorting algorithms, we observe that the attention scores are highly correlated to the effects of the perturbed code elements. Such a correlation provides a strong basis for the uses of attention scores to interpret the relations between code elements and the algorithm classification results of a neural network, and we believe that visualizing code elements in an input program ranked according to their attention scores can facilitate faster program comprehension with reduced code.

Index Terms—attention mechanisms, neural networks, algorithm classification, interpretability, code perturbation, program comprehension

I. INTRODUCTION

Deep learning techniques have been adapted for various software engineering tasks, such as code completion, bug prediction, and program classification [1]–[5]. Despite high prediction accuracy achieved, deep neural networks are mostly treated as black boxes without explanation on why certain outputs are generated for certain inputs [6]–[8], so that users lack of confidence in the results. Attention mechanisms have been proposed [9], [10] for neural networks to focus on certain input elements or features when making predictions, and such elements or features are assumed to reflect certain interpretability of the networks. However, in many cases the features getting higher attentions may be implicit, and the prediction outputs of the attention networks according to the features may disagree with human users’ understanding [11].

In this work, we aim to justify and improve the interpretability of attention-based neural networks with the AutoFocus approach. The key idea of the approach is to reveal correlations between inputs and outputs of attention networks by perturbing inputs and observing the effects of perturbed inputs on the outputs. In this paper, we apply AutoFocus to the attention networks trained for algorithm classification (i.e., networks that classify the algorithm implemented in a given input program [3]–[5], [12]). It helps correlate attention scores of certain code elements (e.g., statements) in a program with the importance of the elements in determining the program’s algorithm class. Such a correlation provides us a strong basis for using attention scores of individual statements as a metric to visualize a program, and helps users in interpreting the networks’ prediction outputs and understanding the program with increased focus, saving the need to read through all code.

We combine two techniques to realize AutoFocus:

1) Syntax-Directed Attention: We adapt attention mechanisms into the neural networks in the context of algorithm classification, and generate attention scores for syntactically meaningful elements in input programs (e.g., statements), instead of arbitrary elements;

2) Code Perturbation: We systematically perturb input programs syntactically (e.g., deleting statements one by one) to observe how the perturbations affect neural networks’ classification outputs and relate to the attention scores.

With respect to tree-based and graph-based algorithm classification neural networks (TBCNN and GGNN [3]–[5], [12]), our key research question here is:

Can the syntax-directed attention scores be used as a proxy to interpret the decisions made by the neural networks?

With evaluation on more than 1000 programs implementing 10 different sorting algorithms, we positively show that the attention scores of individual statements are strongly correlated with the effects of the statements on the classification results, and thus can be used to interpret the input/output behaviour of the networks. Furthermore, the statements in a program can be visualized according to their attention scores to facilitate more focused and faster code comprehension.

More generally, the interpretability produced by the AutoFocus approach technically only depends on the availability of attention scores and the interpretability of code elements that follow certain syntax, and thus AutoFocus is likely applicable to many other neural networks for various code learning tasks.

II. RELATED WORK

Interpretability is important for software mining and analysis in general [13]. In other domains, various techniques have been proposed to interpret machine learning results, such as by projecting outputs of CNN models through hidden neurons to input image pixels [14], by quantifying the effects of different
compositions of English sentences on NLP models [15], and by perturbing inputs for black-box neural networks [16].

Our work is unique in that it adapts the ideas of attention mechanisms and code perturbation to interpret the input/output effects of algorithm classification neural networks via identification and visualization of meaningful code elements.

III. AUTOFOCUS APPROACH OVERVIEW

Figure 1 gives an overview of the six major steps in AutoFocus. Next section explains the steps in more details.

1) Training of attention-based neural networks: We add additional aggregation layers in conventional classification neural networks to generate attention scores for input elements using a global attention mechanism [9], [10]. Given training programs, we obtain trained attention networks.

2) Generation of classification confidence score $c(p)$ for a test program $p$ and attention scores $a(s)$ for each suitable code element $s$ in $p$: Given a test program $p$, the classification confidence score $c(p)$ is derived from the softmax layer of the attention networks, indicating the likelihood for $p$ to belong to a certain class. For multi-class classification tasks (e.g., [3]–[5]), there is a confidence score for each class, while the correct class for $p$ often but not necessarily has the highest confidence score. In this work, we always take the confidence score produced by the trained networks for the correct class of $p$ as the $c(p)$. Meanwhile, the attention networks produce an attention score for each input element, and we aggregate the scores according to $p$’s syntactical structure and produce an attention score for each statement $s$ in $p$, denote as $a(s)$.

3) Perturbation of test program(s): Each test program $p$ is modified into a set of perturbed programs $P' = \{p'_s\}$, where $p'_s$ indicates a perturbed program by deleting the statement $s$ from $p$. For each perturbed program $p'_s$, we apply the attention networks to predict its class and obtain a new confidence score $c(p'_s)$.

4) Impact measurement of perturbing statements: Given a set of perturbed programs $\{p'_s\}$, we have a set of classification confidence scores $\{c(p'_s)\}$. The differences between $c(p)$ and $\{c(p'_s)\}$ are denoted as $\Delta(p) = \{\delta(s) = c(p'_s) - c(p) | s \in p\}$. Intuitively, a higher $\delta(s)$ may indicate a statement $s$ that has more impact on the networks’ classification accuracies and thus may be more important.

5) Correlating statement-level attention scores $\{a(s)\}$ and perturbed confidence scores $\{\delta(s)\}$: We analyze the correlation coefficients between the two kinds of scores for various test programs so that we may use the perturbed classification confidence scores to justify the uses of attention scores to interpret the classification decisions made by the attention networks.

6) Visualization of statements: Given the attention scores $\{a(s)\}$ and perturbed confidence scores $\{\delta(s)\}$ as a proxy for the importance of individual statements in a program $p$, we visualize $p$ with a spectrum of derived colours to facilitate focused view on more important statements for program comprehension.

IV. AUTOFOCUS DETAILS

A. Building Attention Neural Networks

We choose state-of-the-arts tree-based and graph-based neural networks [3], [4], [12], for they yield accurate outputs for algorithm classification.

Fig. 2. Attention mechanism as the aggregation layer for the neural network

Figure 2 illustrates the process of adding attention layers for algorithm classification neural networks. First, source code is parsed as an AST and a graph by connecting tree nodes to dependent ones. Then the neural networks are used as a feature extractor to update the information of each node following the edges. An aggregation layer is used to combine the information about all of the nodes into one single vector as the representation for the code (see Section IV-B).

Since a graph is a more general form of a tree, we summarize the design principle of both TBCNN [3] and GGNN [12] with graph notations. A graph $G = (V, E, X)$ is composed of a set of nodes $V$, a set of node features $X$, and a list of directed edge sets $E = \{E_1, \ldots, E_K\}$ where $K$ is the number of edge types. Initially, we annotate each node $v \in V$ with a real-valued vector $x_v \in \mathbb{R}^d$ representing the features of the node. The node features $X$ come from a pretrained embedding [3]. We associate every node $v$ with a hidden state vector $h_v$, initialised from pretrained feature embedding $x_v$. The process of attention networks can be split into the feature extraction and the aggregation phases. The feature extraction phase aims to propagate information from a node $v$ to its neighbor. Specifically,
• The input to TBCNN is AST which is an undirected graph. A function \( f_{\text{conv}} \) aggregates the information of the direct children of a node \( v \) to update its state vector.

The purpose of this step is to derive attention scores for code elements on the networks’ classification results. Here, we focus on perturbing statements in a program because a statement may be a reasonable level of granularity for developers to examine and understand, and because a recent study [4] shows that splitting ASTs at the statement level achieves better learning results than some other granularity levels.

We work on trees and graphs to perturb statements: we traverse the AST of a program to identify the node sequence corresponding to statements in a post-order, and mutate the trees and relevant graphs to delete the statement nodes and related edges one by one. For simplicity in this paper, when a statement is deleted, all nested substatements are also deleted. Although the deletion can introduce compilation errors in the programs (e.g., undeclared variables), the tree- or graph-based neural networks can still be applied to the perturbed trees and graphs. To limit the time needed for exploring the deletion of various combinations of statements, we delete statements in the greedy post order and only backtracks one statement when deleting the current statement leads to a worse classification.

D. Visualisation

We transform the attention scores of statements into colors to be shown on the foreground of the code tokens contained in the statements. The rules of thumb for the color transformation is to ensure that the statements with higher attention scores get a higher contrast to the background color. Many color transformation schemes are possible. In this work, we use the grey scale to present colors from white (attention=0) to black (attention=1). Since the score of each node ranges from 0 to 1 when we choose the sigmoid function for non-linearity, the darkness increases when the score is increased and vice versa.

V. EMPIRICAL EVALUATION

We verify the capability of AutoFocus with respect to a graph-based neural network trained for multi-class algorithm classification [12]. The data for the evaluation consists of 1023 unique Java programs crawled from GitHub for 10 distinct sorting algorithms [5], where about 70% of the programs were used for training, 10% for validation, and 20% for testing.

First, the settings described in GGNN [12] are used to train a model of 85% accuracy on the 200 test programs; the model is used as the ground truth for interpretation. Then, we follow the steps in Section III to derive attention scores and deltas of deleting statements for each test program. After that, we conducted a statistical analysis on the correlation between the deltas and the attention scores of the statements deleted by code perturbation. Following Step 5 in Section III, we obtain the Pearson correlation ratio for each test program. For all the test programs, a list of Pearson correlation ratios can be seen as a discrete variable \( P \). Figure 3 shows the histogram of
More model visualization and interpretability techniques can be applied to generate attention scores for individual statements in an input program, and (2) inducing syntax-directed code perturbation to observe the effects of individual statements on the network’s classification outputs. It then shows that these two independently derived metrics have a strong correlation and can be used to produce a spectrum visualization of the perturbed program as a recommendation for programmers to identify the most relevant code elements when viewing the program.

REFERENCES