Erste Überlegungen zur Erklärbarkeit von Deep-Learning-Modellen für die Analyse von Quellcode

Tim Sonnekalb Thomas S. Heinze Patrick Mäder
{tim.sonnekalb,thomas.heinze}@dlr.de patrick.maeder@tu-ilmenau.de
Deutsches Zentrum für Luft- und Raumfahrt Technische Universität Ilmenau

Zusammenfassung

Die meisten Deep-Learning-Verfahren haben einen entscheidenden Nachteil: Sie sind Black-Box-Verfahren. Dadurch ist die Auswertung der Ergebnisse mit der Frage nach dem Warum oftmals nicht oder nur bedingt möglich. Gerade bei der Analyse von Software möchte ein Entwickler aber Ergebnisse mit zusätzlicher Begründung, um sie schnell filtern zu können. Erklärbare und interpretierbare Methoden sollen helfen, die Rückverfolgbarkeit von Analyseergebnissen sowie Erklärungen zu liefern.

1 Einleitung

Deep Learning (DL) wird aktuell bei vielen Anwendungen vor allem zur Bild- und Textverarbeitung eingesetzt. Ein weiteres, neueres Forschungsgebiet ist die Analyse von Softwareartefakten wie Quellcode mithilfe von DL-Methoden. Dazu wird Code von Software-Repositorien gesammelt, in Fragmente geteilt und in maschinenlesbare numerische Repräsentationen umgewandelt, um mit einem trainierten Modell im Kontext des Reengineerings Aussagen über die Qualität des Quellcodes zu treffen und z.B. Schwachstellen aufzuzeigen.

Im DL-auf-Code-Bereich gibt es aktuell verschiedene Herausforderungen, um diese Methoden als praxisnahes nutzbares System zu etablieren, welches Softwareentwicklern bei deren Arbeit unterstützt. Aktuell sind Ansätze noch mit einer hohen Rate an Fehlalarmen behaftet, weshalb erhöhter manueller Aufwand notwendig ist, um die Ergebnisse zu validieren [2]. Im Hinblick auf Nutzerfreundlichkeit solcher Systeme können Erklärungen fördern, dass Entwickler Analyseergebnisse schnell verstehen können.

Im Vergleich zu bewährten Methoden wie statische und dynamischer Analyse, die zur Analyse von Software eingesetzt werden, können DL-Methoden das Expertenwissen durch repräsentationsbasiertes Lernen von Schwachstellenmustern ersetzen.Als primäre Analysemethode können sie ressourcensparend Schwachstellen in Softwareprojekten erkennen, nachdem ein effektives Modell trainiert ist. Zusätzlich ist es mit DL-Methoden möglich, Informationen aus nutzerspezifischen Codefragmenten in die Analyse mit einzubeziehen. Nutzerspezifische Codefragmente können direkte Bestandteile des Codes wie Variablen- und Funktionsnamen sein oder indirekte und für die Ausführung nicht relevante Bestandteile, wie Kommentare oder Dokumentation.

2 Problembeschreibung und Forschungsansatz

Eine Analyse auf Code lässt sich als Klassifikationsaufgabe in einer maschinellen Lernumgebung darstellen. Beispielhaft wollen wir uns auf das Lokalisieren von Schwachstellen fokussieren. Diese wird oft im aktuellen Stand der Forschung als Zwei-Klassen-Aufgabe für die Einteilung in sichere und nicht sichere Codebestandteile gestellt, weniger gebräuchlich ist die Darstellung als Multiklassenproblem spezifisch für den Typ der Schwachstelle. DL eignet sich gut, um komplexe Strukturen, so wie die von Code zu analysieren. Aus aktuellen Publikationen geht auch hervor, dass es ein Zielkonflikt zwischen Genauigkeit und Erklärbarkeit dieser Methoden gibt, beispielsweise liefert ein tiefes neuronales Netz als Black-Box-Modell oft die besten Ergebnisse hinsichtlich Genauigkeit der Vorhersage, aber die Ergebnisse sind schlecht bis nicht interpretierbar.

Erklärbarkeit oder Interpretierbarkeit, als Begriffe oft gleichgesetzt in der Literatur, ist die Fähigkeit, Dinge zu erklären oder zu präsentieren, sodass sie für einen Menschen nachvollziehbar sind [3]. Erklärbare maschinelle Lernmethoden gibt es in zwei verschiedenen Richtungen: post-hoc ist die Aufbereitung von bereits trainierten Black-Box-Modellen, wo hingegen ante-hoc Modelle direkt trainierte und erklärbare Modelle sind. Post-hoc Modelle können zusätzlich unterschieden werden in solche mit globalen oder mit lokalen Erklärungen, in anderen Worten die Unterscheidung, ob das gesamte Modell zur Erklärung herangezogen oder nur eine einzelne Vorhersage nachvollziehbar sein soll. Weiterhin kann man orthogonal dazu erklärbare Methoden auch unterscheiden in Methoden, deren Modelle man schrittweise debuggen muss und Methoden, die direkt und nutzerorientierte Erklärungen geben. Die genannten Kategorien sowie ausgewählte DL-Architekturen sind in Abb. 1 dargestellt. Die wichtigsten Bestandteile einer Prozesskette für DL sind die Architektur sowie die Coderepräsen-

tation, auf die wir im Folgenden eingehen werden.

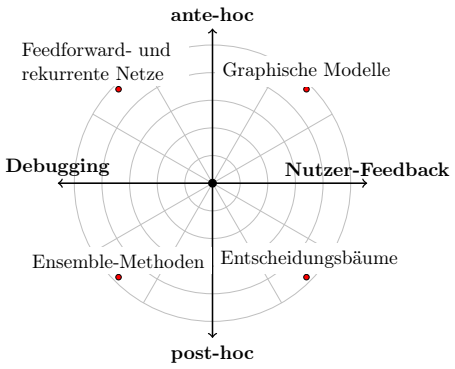

Abbildung 1: Einordnung von maschinellen Lernmethoden hinsichtlich Erklärbarkeitseigenschaften

2.1 DL-Architektur

Wie aus Abb. 1 ersichtlich, weisen DL-Architekturen unterschiedliche Eigenschaften hinsichtlich Erklärbarkeit auf. Ein tiefes neuronales Netz beispielsweise wird direkt/ante-hoc auf ein Klassifikationsproblem angewendet, mit zunehmender Komplexität des Netzes durch mehrere versteckte Schichten oder eine hohe Anzahl an verschalteten Neuronen steigt die Schwierigkeit der Rückverfolgbarkeit von den Ausgabe- zu den betroffenen Eingabeneuronen. Möchte man die Black-Box-Modelle debuggen, könnte man die Testdaten solange leicht modifiziern, bis eine Veränderung der Eingabe auch zu einer Abweichung der Klassifikation führt und so relevante Entscheidungsmerkmale des Modells identifizieren kann [1]. Generell sollte man eine Balance zwischen Komplexität des Netzwerks und Grad an Interpretierbarkeit finden. Daher kann es sinnvoll sein, aggregierte Informationen in die Coderepräsentation zu inkludieren, sodass ein weniger komplexes DL-Netz notwendig ist. Eine weitere oder zusätzliche Möglichkeit wäre eine schichtenweise Relevanzpropagierung zur Hervorhebung der wichtigen Stellen im Code, die zu den Entscheidungen geführt haben [5]. Um unterstützend in Richtung Nutzer-Feedback zu wirken (Abb. 1), könnten vordefinierte Erklärungen gelernt werden, die im passenden Ereignis ausgegeben werden. Die Erklärungen können zudem entsprechend des Wissens- und Erfahrungsstands des Nutzers in der Länge angepasst werden.

2.2 Coderepräsentation

Eine Coderepräsentation beschreibt die Verarbeitung des Quellcodes zu Zwischenformaten, die anschließend mittels eines Embeddings zu einer Menge aus numerischen Vektoren gleicher Länge konvertiert werden, um sie als Eingaben in eine DL-Architektur zu verwenden. Für die Quellcodeanalyse mit DL werden verschiedene Graphformate aus dem Compilerbau verwendet, wie beispielsweise ein abstrakter Syntaxbaum, Kontrollflussgraph, Datenflussgraph oder auch ein kombinierter Graph, wie in [6]. Diese Graphen werden direkt in ein Embedding überführt oder in eine Sequenz aus Tokens weiterverarbeitet. Eine Tokensequenz beinhaltet Wörter und Zeichen, kann aber auch unterschiedlich granular sein, indem mehrere Wörter, die beispielsweise eine Operation bilden, zu einem einzelnen Token zusammengefasst werden.

Bei der Transformation der Token mittels eines Embeddings zu numerischen Vektoren sollte optimal nicht die Semantik der Tokens verloren gehen, sodass durch Operationen die entstandenen numerischen Vektoren in ein Verhältnis gesetzt werden können wie ein Mensch die Tokens als ähnlich erkennen würde. Dafür wird aktuell oft ein word2vec Modell [6] verwendet, welches für die Textverarbeitung mit DL entwickelt wurde. Beispielhaft für ein Code-Embedding sei das Framework code2seq [4] genannt, welches zur Aggregation von unterschiedlich langen Codefragmenten zu einem Vektor begrenzter Länge eingesetzt werden kann. Ein Embedding ist oftmals programmiersprachenspezifisch, da es spezifisch für die Sprache in einem Training erzeugt wird.

3 Zusammenfassung und Ausblick

Die Entwicklung von erklärbaren Methoden für die Analyse von Quellcode mit DL ist aktuell in den Anfängen und braucht noch viel Entwicklungsarbeit. Wie wir in den vorherigen Abschnitten beschrieben haben, ist es notwendig, dass zukünftige Arbeiten dieses Thema berücksichtigen, sodass die entwickelten Methoden in der Praxis eingesetzt werden können. Aktuell gilt es, eine Balance zwischen der Komplexität des Netzwerkes und dem Grad an Interpretierbarkeit zu finden.

Literatur

[1] M. T. Ribeiro, S. Singh und C. Guestrin. “"Why Should I Trust You?": Explaining the Predictions of Any Classifier”. In: (16. Feb. 2016). arXiv: 1602.04938v3 [cs.LG].

[2] M. Allamanis u. a. “A Survey of Machine Learning for Big Code and Naturalness”. In: (18. Sep. 2017). arXiv: 1709.06182v2 [cs.SE].

[3] F. Doshi-Velez und B. Kim. “Towards A Rigorous Science of Interpretable Machine Learning”. In: (28. Feb. 2017). arXiv: 1702.08608v2 [stat.ML].

[4] U. Alon u. a. “code2seq: Generating Sequences from Structured Representations of Code”. In: (4. Aug. 2018). arXiv: 1808.01400v6 [cs.LG].

[5] A. Warnecke u. a. “Evaluating Explanation Methods for Deep Learning in Security”. In: (5. Juni 2019). arXiv: 1906.02108v3 [cs.LG].

[6] Y. Zhou u. a. “Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks”. In: (8. Sep. 2019). arXiv: 1909.03496v1 [cs.SE].