Leibniz Universität Hannover

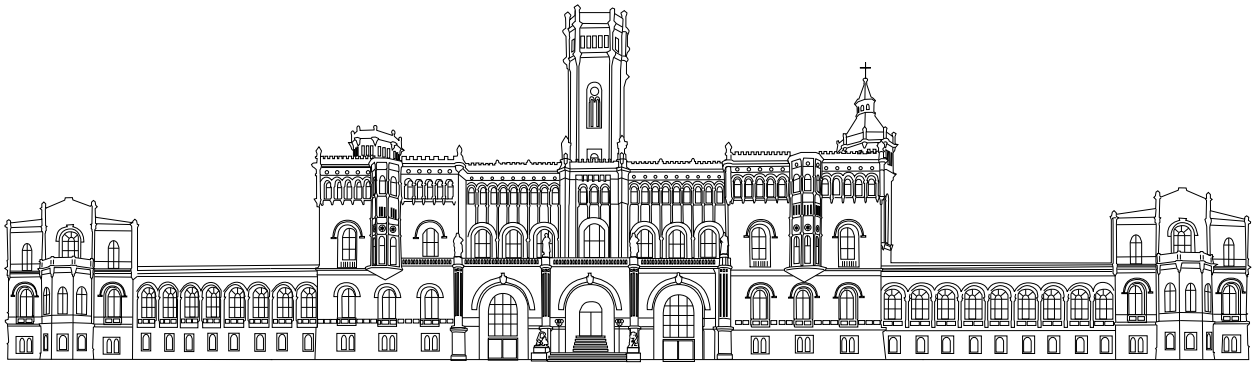
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Code-Schwachstellenanalyse mit Neuronalen Netzwerken

A Neural Network for Code Vulnerability Analysis

Masterarbeit im Studiengang Informatik (M.Sc.),
eingereicht von LEON ROSENTHAL am 26. Oktober 2018

Erstprüfer:	Prof. Dr. Kurt Schneider
Zweitprüfer:	Prof. Dr. Joel Greenyer
Betreuer:	Fabien Patrick Viertel (M. Sc.)



Erklärung der Selbständigkeit

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden, alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind, und die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen hat.

_____ Hannover, den 26. Oktober 2018

Leon Rosenthal

Zusammenfassung

In dieser Masterarbeit wird eine Evaluation verschiedener Typen von neuronalen Netzwerk in dem Bereich der automatisierten Schwachstellenanalyse von Software-Code durchgeführt. Zusätzlich wird ein entsprechendes Programm prototypisch umgesetzt.

Die Sicherheit von IT-Systemen hängt maßgeblich von der Vermeidung von Schwachstellen in Implementierungen ab. Das Verhindern von Schwachstellen erfordert ein weitreichendes Wissen in dem Bereich der IT-Sicherheit. Da hoch frequentiert neue Sicherheitslücken gefunden werden, ist es für einen Entwickler ohne eine Spezialisierung auf IT-Sicherheit schwer, immer einen sicheren Code gewährleisten zu können.

Um Entwickler in sicherheitsrelevanten Bereichen zu unterstützen, wird in dieser Ausarbeitung die Grundlage geschaffen, um automatisiert in Software-Repositories nach Schwachstellen zu suchen.

Für das Fundament der Automatisierung sollen neuronale Netzwerke dienen. Zwecks Gewährleistung der optimalen Leistung werden verschiedene Arten von neuronalen Netzen als Lösungsansatz getestet und mit einander verglichen.

Nach der Evaluation der Netzwerkarten wird die geschaffene Grundlage hinsichtlich ihrer Einsatzmöglichkeiten für Entwickler während der Entwicklungszeit betrachtet.

Abstract

The topic of this master's thesis is the evaluation of different types of neural networks in the field of vulnerability analyses of software code. Additionally, a prototype will be implemented.

The avoidance of known vulnerabilities in implementations is one essential step in software engineering and the depending security of the IT systems. In order to prevent such vulnerabilities a deep knowledge in the field of IT security is mandatory. Due to the high frequency of newly discovered security breaches, it is difficult for software developers without special knowledge in IT security to ensure secure software code at any given time.

To help developers in security related fields, this thesis creates a foundation for automating the search for code vulnerabilities in software repositories.

The groundwork of the automation is based on neural networks. To guarantee the optimal performance, different types of neural networks will be tested and compared.

After having evaluated different network types possible applications for software developers will be discussed.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

KAPITEL 1

Einleitung

In diesem Kapitel wird die vorliegende Thesis einführend beschrieben. Im Zuge dessen wird zunächst die zugrunde liegende Motivation der Arbeit erörtert. Dabei wird auf aktuelle Thematiken Bezug genommen. Fortlaufend sollen die Ziele der Ausarbeitung definiert und erklärt werden. Zuletzt soll ein Überblick über den Aufbau und die Struktur der Arbeit geschaffen werden.

1.1 Motivation

Die Verbreitung von Software-Systemen beschränkt sich schon seit langer Zeit nicht mehr nur auf den Heimrechner. Software ist aktuell an so vielen Stellen zu finden wie noch nie. Ob es nun das Smartphone mit sensiblen persönlichen Daten [63], das täglich genutzte Auto [64] oder der lebenserhaltende Herzschrittmacher [26] ist, der Großteil der elektronischen Geräte wird heutzutage von etwas Quellcode gesteuert oder reguliert. Für den Endverbraucher bedeuten diese Geräte eine Erleichterung des Alltags. Umso leichter der Alltag wird, desto vernetzter wird selbiger. Mit der immer wachsenden Vernetzung steigt allerdings ebenso das Risiko, dass eben diese durch Angreifer ausgenutzt wird, um Geräte entgegen des eigentlichen Verwendungszwecks zu manipulieren. Es wird für Angreifer immer leichter an sensible Daten zu gelangen. Laut einem Fallbericht des *Journal of Forensics Sciences And Criminal Investigation* [51] stiegen die weltweiten Verluste durch Kreditkartenbetrug zwischen den Jahren 2010 und 2015 von ca. 8

Millarden Dollar um 100% auf ca. 16 Millarden Dollar. Auch national ist man sich dieses Problems bewusst. Viele Organisationen bieten ihren Kunden über Webanwendungen diverse Dienste an. Laut des Bundesamts für Sicherheit in der Informationstechnik heißt es in der Veröffentlichung *Die Lage der IT-Sicherheit in Deutschland 2017* [53], dass über 10% der Webanwendungen kritische und über zwei Drittel schwerwiegende Mängel aufweisen.

Diese Mängel bedeuten Sicherheitslücken in einem Software-System. Der Veröffentlichung [53] ist zu entnehmen, dass die Prävention einen großen Teil der Sicherung eines Systems ausmacht. Um ein System präventiv gegen Angriffe zu härten, muss der Vermeidung von bereits bekannten Sicherheitsschwachstellen eine hohe Priorität zugewiesen werden. Dadurch wird versucht, die mögliche Angriffsfläche zu reduzieren.

Damit wird jedoch auch jedem Entwickler eine große Verantwortung zugetragen. Um bereits bekannte Sicherheitslücken vermeiden zu können, ist ein weitreichendes und tiefgehendes Wissen in dem Bereich IT-Security erforderlich. Viele Fachlichkeiten in dem Gebiet Security werden allerdings nur von wenigen Experten durchdrungen [53]. Damit auch Entwickler, welche keine Security-Experten sind, präventiv Sicherheitslücken vermeiden können sollen, müssen sie bereits während der Entwicklungszeit eines Systems automatisierte Unterstützung erhalten. Zur Entlastung von Entwicklern sollten Fachlichkeiten des Security-Bereichs in einen automatisierten Entwicklungsprozess ausgelagert werden.

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist die Entwicklung einer prototypischen Software namens *Security Vulnerabilities Learner* (kurz SecVuLearner), welche Quellcode automatisiert bezüglich Sicherheitslücken testet. Der Prozess wird sich als Grundlage an Algorithmen aus dem Bereich des maschinellen Lernens bedienen. Das Fundament werden verschiedene Arten von neuronalen Netzwerken sein. Diese werden im Laufe der Arbeit evaluiert, um den effizientesten und performantesten Typen von Netzwerken als Kernelement des Prototypen zu implementieren.

Zunächst werden die Anforderungen an die Software definiert und erläutert. Nach dem Entwurf und der Implementierung eines Grundgerüstes, wird näher auf die verschiedenen Typen von neuronalen Netzwerken als austauschbare Kernele-

mente eingegangen. Das hervorgehobene Ziel der Thesis ist somit die Evaluation der Netzwerke bezüglich ihrer Performanz in dem Bereich der automatisierten Schwachstellenerkennung.

1.3 Struktur der Arbeit

Die vorliegende Thesis ist in sieben Kapitel unterteilt. Im Folgenden sollen die einzelnen Kapitel kurz vorgestellt und inhaltlich erläutert werden. In Kapitel 1 werden dem Leser zunächst die Motivation und die Ziele dieser Arbeit vorgestellt. Zusätzlich werden in diesem Kapitel einige verwandte Arbeiten des Themengebiets vorgestellt. Außerdem wird auf die Struktur dieser Arbeit eingegangen. Kapitel 2 umfasst die Aufstellung der diesem Projekt zugrunde liegenden Anforderungen. Hierbei wird zunächst auf die Stakeholder näher eingegangen. Folgend werden die funktionalen, sowie die nicht-funktionalen Anforderungen genauer betrachtet. Anschließend werden die aufgestellten Anforderungen noch einer Priorisierung unterzogen. Kapitel 3 befasst sich mit den Grundlagen, welche zum Verständnis dieser Arbeit erforderlich sind. Entsprechend werden relevante Begriffe aus dem Bereich Security näher beschrieben. Außerdem werden Grundlagen zu neuronalen Netzwerken und deren speziellen Typen geschaffen. Des Weiteren gibt es einen Überblick über Code Clone Detection im Allgemeinen. Das Entwurfskapitel, das Kapitel 4, beschreibt das Design und den Aufbau der Anwendung. Im folgenden Kapitel 5 wird ein genauerer Blick auf die Umsetzung und die Implementierung des vorgestellten Designs geworfen. Kapitel 6 umfasst alle relevanten und interessanten Fakten bezüglich der Evaluation der verwendeten neuronalen Netze. Im Kapitel 7 wird ein Fazit gezogen, sowie ein Ausblick auf folgende Tätigkeiten bezüglich der Ergebnisse getätigt. Zum Schluss wird selbstreflektierend über die Gültigkeit dieser Thesis geschrieben. Das letzte Kapitel befasst sich mit verwandten Ausarbeitungen und stellt diese vor.

KAPITEL 2

Anforderungen

In dem folgenden Kapitel wird ein genauerer Blick auf die Anforderungen an die zu erstellende Software und die Evaluation geworfen. Zunächst werden in Abschnitt 2.1 alle teilhabenden Personen definiert und vorgestellt. Die beiden anknüpfenden Kapitel werden sich mit den funktionalen, sowie mit den nicht-funktionalen Anforderungen befassen (2.2.1 und 2.2.2). Anschließend wird eine Priorisierung der deklarierten Anforderungen getroffen.

2.1 Stakeholder

Die entstehende Software wird sich an drei verschiedene Zielgruppen richten. Da die vorliegende Arbeit allerdings auch einen weiteren Schwerpunkt auf die Evaluation von unterschiedlichen Typen von neuronalen Netzwerken legt, gibt es eine weitere Zielgruppe, welche nicht direkt mit der Verwendung der Software in Beziehung steht. Alle vier Zielgruppen werden folgend beschrieben.

Zunächst wird die Zielgruppe dargelegt, welche nur indirekt von der entwickelten Software profitieren wird. Zu diesen Personen zählen alle, welche sich mit den Einsatzmöglichkeiten von neuronalen Netzen befassen. Durch die am Ende der Entwicklung durchgeführten Evaluation werden Schlüsse über die Tauglichkeit der verschiedenen Netzwerkarten bezüglich einer Klassifikation von Software-Code gezogen werden können. Diese Erkenntnisse sollen optimaler Weise für zukünftige Arbeiten dienlich sein.

Die erste software-spezifische Zielgruppe sind Software-Entwickler. Das entstehende Tool soll die Programmierer während ihrer Entwicklungsphase einer Software unterstützen. Um den Sicherheitsstandard einer neu zu entwickelnden Software möglichst hoch zu halten, ist es zuträglich, wenn Entwickler versuchen Sicherheitslücken in dem Quellcode präventiv zu vermeiden. Hierzu muss die entwickelnde Person während des Programmierens den eigens geschriebenen Software-Code kritisch hinterfragen, um mögliche eingebaute Schwachstellen zu identifizieren. Diese prophylaktische Maßnahme soll mit dem zu entwickelnden Tool automatisiert unterstützt werden.

Des Weiteren werden Experten aus dem Security-Bereich als Zielgruppe angesehen. Firmen mit einer QM[1]-Abteilung haben oft einen Sicherheitsbeauftragten. Dieser ist für die Gewährleistung von sicherer Software verantwortlich. Viele Personen in diesem Aufgabenbereich erlassen eine Security-Policy für den IT-Bereich, an den sich die Software-Entwickler halten sollen, damit kein unsicherer Quellcode entsteht. Unter einer Security-Policy versteht man folgendes:

> „The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information.“ [54]

Damit ein sich Sicherheitsbeauftragter einen schnellen Überblick über die Einhaltung bzw. die Missachtung einer Security Policy verschaffen kann, sollte er schnell und einfach Code bezüglich Sicherheitslücken überprüfen können. Ein Widerhandeln kann beispielsweise das Ignorieren oder Verändern eines vorgegebenen Login-Prozesses sein [22].

In der letzten Zielgruppe befinden sich Mitarbeiter und Studenten der Leibniz Universität Hannover, welche am Fachgebiet für Software Engineering tätig sind. Der mit dieser Masterthesis zusammenhängend entstandene Quellcode kann als ein weiterführbares Forschungsprojekt angesehen werden. Die Ausarbeitung an sich kann für eventuelle anschließende Verwendungszwecke der Software als Dokumentation dienen.

[1] Quality Management

2.2 Anforderungen

Anforderungen an ein Softwaresystem lassen sich in zwei verschiedene Kategorien einteilen: die funktionalen Anforderungen und die nicht-funktionalen Anforderungen[2]. Die funktionalen Anforderungen beschreiben die fachlichen Verhaltensweisen, welche ein System abbilden muss. Folglich kann man sagen, dass diese Anforderungen definieren *was* die Software leisten muss.

Die zweite Art der Anforderungen sind die nicht-funktionalen Anforderungen an das System. Diese beschreiben Qualitätseigenschaften einer Software. Sie beziehen sich somit nicht direkt auf die Funktionalität des Programms an sich. Dies impliziert, dass sie im Gegensatz zu den funktionalen Anforderungen nicht beschreiben *was* ein System leisten muss, sondern viel mehr *wie* eine Software ihre Funktionalitäten anbietet. Unter den Begriff der nicht-funktionalen Anforderungen fallen laut Lawrence Chung [10] mittlerweile unter anderem Leistungen wie die Wartbarkeit, die Verlässlichkeit und die Portabilität der Software.

2.2.1 Funktionale Anforderungen

In diesem Unterkapitel sollen die funktionalen Anforderungen an die zu entwickelnde Software gestellt werden. Diese beginnen alle mit einem **R**. Da sich die vorliegende Arbeit auch tiefergehend mit der Evaluation von unterschiedlichen Typen von neuronalen Netzwerken beschäftigt, werden auch Anforderungen an die Auswertung der verwendeten Modellen gestellt. Diese beginnen in der Aufzählung mit **RE**.

[R1] Die Software soll Schwachstellen in Quellcode erkennen

Der Quellcode, welcher dem Programm als Eingabe übergeben wird, soll auf eventuell bekannte Schwachstellen überprüft werden.

[R2] Klassifikation einer Schwachstelle

Wurde eine potentielle Schwachstelle erkannt, so soll sie der entsprechenden bekannten Schwachstelle zugewiesen werden.

[2] Erstmals 1985 von Roman [47] als solche definiert.

[R3] Nutzung von bestehenden Datenbanken

Die Menge der bekannten Schwachstellen soll durch Elemente der CVE[3]-Datenbank definiert sein.

[R4] Optional: Erweiterung der bekannten Schwachstellen

Falls ein Software-Entwickler das Wissen über eine Schwachstelle hat, welche noch nicht in der CVE-Datenbank enthalten ist, so soll er sie manuell einfügen können, sodass die Menge an bekannten Schwachstellen einfach erweitert werden kann.

[R5] Klone der Typen[4] 1, 2 und 3 sollen erkannt werden

Laut Vorgabe des Fachgebiets für Software Engineering sollen potentielle Schwachstellen erkannt werden, welche Klone der Typen 1, 2 und 3 von bereits bekannten Sicherheitslücken sind.

[RE1] Evaluation der Kernalgorithmen

Jede unterschiedliche Kernalgorithmik soll bezüglich der Tauglichkeit für das Einsatzgebiet der Code-Schwachstellenanalyse evaluiert werden.

[RE2] Datenerfassung während der Trainingsphasen

Die Daten, welche für die Evaluation von Bedeutung sind, sollen während der Trainingsphasen der unterschiedlichen Arten der neuronalen Netze mit erfasst und gespeichert werden.

[RE3] Datenerfassung nach den Trainingsphasen

Nach Ablauf der Trainingsphasen sollen die für die Evaluation wichtigen Daten entsprechend der Konfiguration kenntlich gespeichert werden.

2.2.2

Nichtfunktionale Anforderungen

In diesem Unterkapitel wird ein genauerer Blick auf die nicht-funktionalen Anforderungen geworfen. Diese beginnen mit **NR**. Auch hier wird die Evaluation mit berücksichtigt, sodass diese Anforderungen mit **NRE** anfangen.

[NR1] Software ist in Java geschrieben

[3] In Abschnitt 3.1.2 wird erläutert, was eine CVE-Datenbank ist.
[4] Die verschiedenen Klontypen werden in dem Kapitel 3.4 erläutert.

Laut Vorgabe des Fachgebiets für Software Engineering soll die zu entwickelnde Software in der Programmiersprache *Java* geschrieben sein.

[NR2] Zu überprüfender Quellcode ist in Java geschrieben

Laut Vorgabe des Fachgebiets für Software Engineering soll der Software übergebene Quellcode in der Programmiersprache *Java* geschrieben sein.

[NR3] Strukturierter und kommentierter Quellcode

Laut Vorgabe des Fachgebiets für Software Engineering soll der während der Entwicklung geschriebene Software-Code gut strukturiert und kommentiert sein, sodass eine Weiterverwendung vereinfacht wird.

[NR4] Modularer Aufbau

Das Fachgebiet für Software Engineering fordert, dass die geschriebene Software modular aufgebaut ist. Dies soll zukünftige Arbeiten mit dem erstellten Quellcode erleichtern.

[NR5] Realisierung durch neuronales Netzwerk

Der Kernalgorithmus für den Erkennungsprozess soll mit Hilfe von einem neuronalen Netzwerk realisiert sein.

[NR6] Optional: Erweiterbarkeit der Software

Die Software soll erweiterbar sein, sodass auch andere Programmiersprachen abseits von *Java* als Eingabe für den Erkennungsprozess genutzt werden können.

[NR7] Performanz der Software

Die zu erstellende Software soll möglichst performant arbeiten. Hierbei wird auf die Schnelligkeit und die Abdeckung des Erkennungsprozesses und auf die Dauer der Trainingsphasen besonders großen Wert gelegt. Der Prototyp soll den effizientesten und performantesten Typen der evaluierten Netzwerkarten als Kernalgorithmus implementieren.

[NRE1] Verwendung von unterschiedlichen Konfigurationen[5]

Für jede unterschiedliche Kernalgorithmtik sollen verschiedene Konfigurationen evaluiert werden.

[NRE2] Unterschiedliche Validierungsverfahren

[5] Was eine Konfiguration ist, wird in Kapitel 3.2 beschrieben.

Für die Evaluation der verschiedenen Arten der neuronalen Netzwerke sollen unterschiedliche Validierungsverfahren genutzt werden, damit nicht nur eine Betrachtungsweise verwendet wird.

[NRE3] Verschiedene Kernalgorithmen

Es sollen verschiedene Arten von neuronalen Netzwerken für den Erkennungsprozess realisiert werden.

[NRE4] Modularität der Kernalgorithmen

Die Kernalgorithmen für den Erkennungsprozess sollen modular implementiert sein.

2.3 Priorisierung

Um die erhobenen Anforderungen einer Priorisierung unterziehen zu können, wird das Kano-Modell der Kundenzufriedenheit [28] genutzt. Bei diesem Vorgehen werden alle Kundenanforderungen in drei Kategorien unterteilt, welche verschiedene Grade von Zufriedenheitsmerkmalen abdecken. Diese drei Gruppen sind definiert als die *Basisanforderungen*, die *Leistungsanforderungen* und die *Begeisterungsanforderungen*. Bevor die zuvor deklarierten Anforderungen den Kategorien zugeteilt werden, sollen sie zunächst kurz näher erläutert werden. Der Autor Sauerwein beschreibt das Modell wie folgt:

> „Das Kano-Modell bietet eine Methodik, Produkteigenschaften in mehrere Kategorien zu unterteilen, deren Erfüllung bzw. Nichterfüllung die Kundenzufriedenheit beeinflussen.“ [48].

Die fundamentalen Funktionen eines Programms werden über die Basisanforderungen abgebildet. Sie beziehen sich also auf Funktionalitäten einer Software, welche grundlegend gewährleistet werden müssen, damit das zu entwickelnde Programm richtig funktioniert. Als ein Beispiel für eine Basisanforderung kann die *Speichern*-Funktion in einem Textbearbeitungprogramm angeführt werden. Somit stellen die Basisanforderungen gewisser Maßen die Mindesterwartung eines Kunden dar. Im Prozess der Anforderungserhebung kann diese Kategorie unter Umständen zu Problemen führen. Manche Kunden sehen einige Funktionen eines Programms als selbstverständlich an, sodass es vorkommen kann, dass

Basisanforderungen nicht direkt vom Kunden als Anforderung genannt werden. Folglich ist hier eine gute Kommunikation zwischen dem Kunden und dem Erheber der Anforderung erforderlich. Eben weil viele dieser Funktionen als Selbstverständlichkeit erachtet werden, haben sie keinen großen Mehrwert für die Zufriedenheit des Kunden. Je mehr Basisanforderungen erfüllt werden, führt nicht zwangsweise zu einer höheren Zufriedenheit des Kunden. Dies kann man in Abbildung 2.1 an der roten Linie erkennen.

Für die Kundenzufriedenheit maßgeblich mitentscheidend sind die Leistungsanforderungen. Wie in Abbildung 2.1 durch die blaue Linie dargestellt, wächst die Zufriedenheit des Kunden linear mit der Erfüllung solcher Anforderungen. Diese Teilmenge der Anforderungen werden von dem Kunden explizit definiert. Dies liegt daran, dass diese Funktionalitäten des Programms oft individuell für den Kunden entwickelt werden. Als Beispiel kann hier eine Bearbeitungssicht einer Animationssoftware genannt werden. Diese kann ja nach Kundenwunsch spezifisch als 2D- oder 3D-Ansicht implementiert werden.

Die dritte Kategorie sind die Begeisterungsanforderungen. Diese liefern nach Erfüllung einen Mehrwert für das Produkt, welchen der Kunde selbst noch nicht bewusst war. Folglich kommt es zu einem starken Anstieg der Zufriedenheit. Es handelt sich also um Funktionen, welche die Erwartung des Kunden übertreffen. In der Abbildung 2.1 sind die Begeisterungsanforderungen durch die grüne Linie dargestellt. Als Beispiel kann hier eine automatisierte Kategorisierung von Nachrichten als Spam in einem Mail-Programm genannt werden, welche anhand der ungelesen gelöschten Emails eine Definition für Spam-Nachrichten lernt.

Die Kategorisierung der erhobenen Anforderungen bezüglich der zu realisierenden Software und der durchzuführenden Evaluation anhand des Kano-Modells ist in Tabelle 2.1 zusammengefasst.

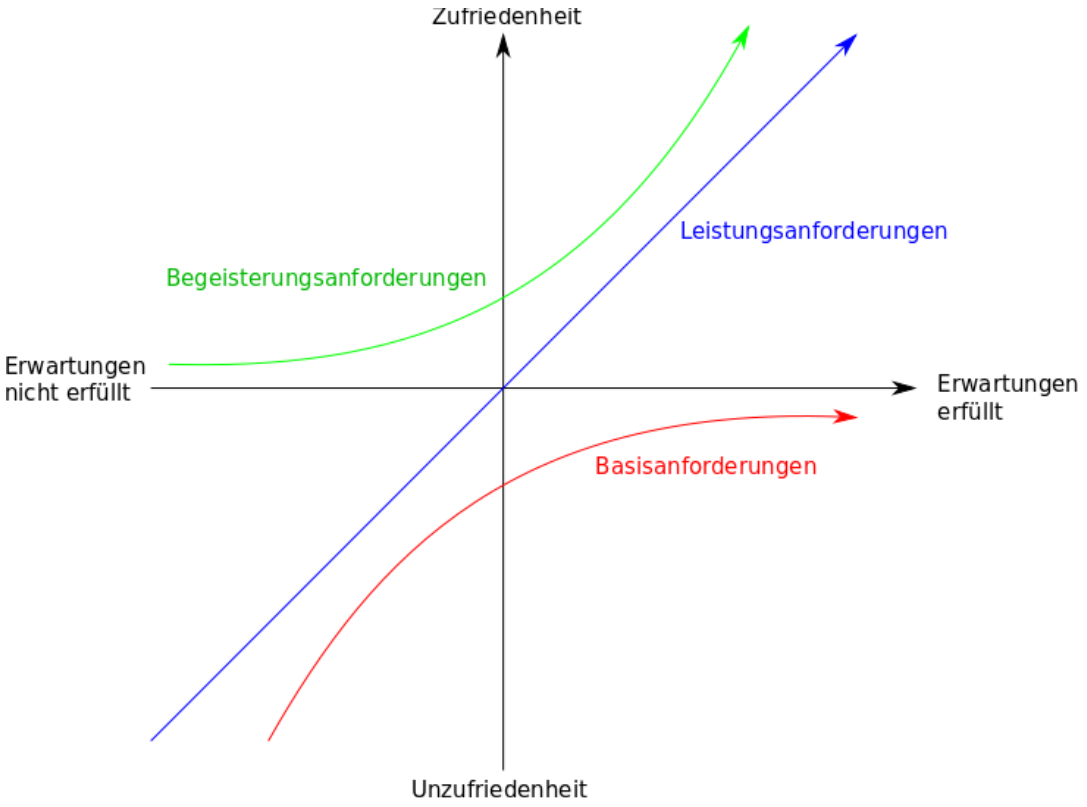


Abbildung 2.1: Kano-Modell in Anlehnung an [28], S.85

Anforderung	Klassifikation	Priorität
R1	Basisanforderung	Hoch
R2	Basisanforderung	Hoch
R3	Leistunganforderung	Mittel
R4	Begeisterunganforderung	Niedrig
R5	Basisanforderung	Hoch
RE1	Basisanforderung	Hoch
RE2	Leistunganforderung	Mittel
RE3	Leistunganforderung	Hoch
NR1	Leistunganforderung	Mittel
NR2	Leistunganforderung	Mittel
NR3	Leistunganforderung	Mittel
NR4	Leistunganforderung	Mittel
NR5	Basisanforderung	Hoch
NR6	Begeisterunganforderung	Niedrig
NR7	Leistunganforderung	Mittel
NRE1	Basisanforderung	Hoch
NRE2	Leistunganforderung	Hoch
NRE3	Leistunganforderung	Hoch
NRE4	Begeisterunganforderung	Niedrig

Tabelle 2.1: Klassifikation der Anforderungen nach Kano-Modell

KAPITEL 3

Grundlagen

Für das Verständnis der weiteren Ausarbeitung werden in diesem Kapitel wesentliche Grundlagen vermittelt. In Abschnitt 3.1 wird mit Begrifflichkeiten aus dem Security-Bereich begonnen. Folgend sollen in Kaptitel 3.2 die Grundzüge von neuronalen Netzwerken beschrieben werden, sodass in Abschnitt 3.3 auf die verschiedenen Arten der Netze eingegangen werden kann. Abschließend werden weitere verwendete Begriffe aus dem Gebiet *Code-Clone-Detection* erläutert.

3.1

Begriffe des Security-Bereichs

Im bisherigen Verlauf dieser Thesis wurden bereits einige Begrifflichkeiten aus dem Gebiet der IT-Sicherheit verwendet. Um für das weitere Vorgehen ein einheitliches Verständnis der Begriffe gewährleisten zu können, werden die beiden wichtigsten in diesem Kapitel definiert. Zunächst soll erläutert werden, was in dieser Arbeit unter dem Begriff *Schwachstelle* verstanden wird. Anschließend soll der Zusammenhang mit der CVE-Datenbank dargelegt werden.

3.1.1

Schwachstelle

Die eindeutige Definition der Schwachstelle ist ein schwieriges Unterfangen. In dem Bereich der IT-Sicherheit wird dieses Wort als ein Oberbegriff verwendet. Zunächst versteht man unter einer Schwachstelle (engl. *weakness*) lediglich eine Schwäche in einem System [23]. Laut dem Konsortiums des *CWE*[1] fallen darunter Fehler und Verwundbarkeiten (engl. *vulnerabilities*) in der Implementierung, dem Design und der Architektur einer Software, welche zu einem von Angreifern verwundbaren System führen können:

„Software weaknesses are flaws, faults, bugs, vulnerabilities, and other errors in software implementation, code, design, or architecture that if left unaddressed could result in systems and networks being vulnerable to attack.“ [24]

Eine Teilmenge der IT-Schwachstellen sind folglich die Verwundbarkeiten. Darunter versteht man eine Schwachstelle, mit welcher die Sicherheitsdienste eines Systems unautorisiert modifiziert oder umgangen werden können. [23]. Bezieht man sich auf das Common Vulnerabilities and Exposures Konsortium (siehe Kapitel 3.1.2), so lautet die Definition einer Verwundbarkeit wie folgt:

„A «vulnerability» is a weakness in the computational logic (e.g., code) found in software and some hardware components (e.g., firmware) that, when exploited, results in a negative impact to confidentiality, integrity, OR availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety).“ [58]

Eine Verwundbarkeit ist also laut der CVE ein Fehler in der Funktionsweise, welche in der Software sowie der Hardware zugrunde liegt. Diese Fehler können von Angreifern ausgenutzt werden. Ist dies der Fall, so kann das negative Auswirkungen auf die Integrität, die Verfügbarkeit und/oder die Vertraulichkeit haben. Um eine Verwundbarkeit aufheben zu können, ist es typischerweise erforderlich, dass die entsprechende Stellen im Software-Code geändert werden müssen. Die in dieser Thesis zu entwickelnde Software befasst sich explizit mit diesen Quellcode-Änderungen.

[1] CWE - Common Weakness Enumeration

In diesem Abschnitt wurde beschrieben, dass der Ausdruck *Schwachstelle* als Oberbegriff für diverse Arten von Fehlern im Gebiet der IT-Sicherheit interpretiert werden kann. Im Laufe dieser Arbeit wird die Bezeichnung *Schwachstelle* mit der Definition der *Verwundbarkeit* gleichgesetzt. Des Weiteren wird der Terminus *Sicherheitslücke* als Synonym verwendet.

3.1.2

CVE

Die Vermeidung von Sicherheitslücken ist ein wesentlicher Bestandteil für die Erhaltung einer guten IT-Sicherheit. Jede Organisation, die ein Produkt anbietet, welches auf Software angewiesen ist, muss dafür Sorge tragen, dass Software-Entwickler Quellcode schreiben, welcher möglichst keine Schwachstellen implementiert. Um dies zu vereinfachen, ist es von Vorteil wenn man eine Auflistung von bereits bekannten Schwachstellen hat, sodass diese explizit vermieden werden können. Da Angriffe in dem Bereich der Cybersecurity ein Problem für jede Organisation darstellen kann, wäre eine Vereinheitlichung aller bekannten Sicherheitslücken für jede Unternehmung von Vorteil. Durch den Zugriff auf Daten können Sicherheitsbeauftragte Bedrohungen bei der Entstehung identifizieren und schnell reagieren [18].

Einen solchen Zusammenschluss wird durch die gemeinnützige MITRE Corporation realisiert. Diese betreibt die Common Vulnerabilities and Exposures (CVE). Die CVE wurde 1999 mit dem Ziel gegründet, dass Informationen über bekannte Cybersecurity Schwachstellen zentralisiert, öffentlich und vereinheitlicht gespeichert werden sollten [1]. Dies war ein neuer Ansatz im Umgang mit bekannten Sicherheitslücken. Bis zu diesem Zeitpunkt verwendeten viele Cybersecurity Tools eigene Datenbanken und Kennzeichnungen für Sicherheitsschwachstellen. Ohne eine gemeinsame Notation war es oft nicht ersichtlich, ob sich verschiedene Tools auf die gleiche Sicherheitslücke bezogen. Dies erschwerte den Umgang mit Schwachstellen enorm. Die eingeführte Notation hat folgenden beispielhaften Aufbau:

`CVE-2014-10001` [15]

Diese Notation wird auch *CVE-ID* genannt und besteht aus drei Fragmenten. Diese sind ein Präfix, eine Jahreszahl und eine variable Anzahl von Zahlen:

`CVE prefix + Year + Arbitrary Digits` [15]

Jede CVE-ID ist eine eindeutige Kennzeichnung für eine bekannte Sicherheitsschwachstelle. Die CVE führt eine Liste dieser Sicherheitslücken, welche stetig erweitert wird. Der Prozess um einen neuen CVE-Eintrag der Liste hinzuzufügen, beginnt mit der Entdeckung einer potentiellen Security-Schwachstelle [1]. Wie bereits erwähnt, funktioniert die CVE als Zusammenschluss von verschiedenen Organisationen. Jeder Teilnehmer kann der Liste neue Einträge hinzufügen. Ein neues Element besteht aus einer neuen CVE-ID, einer Beschreibung der Sicherheitsschwachstelle und Referenzen[2], an welchen Stellen die Security-Lücke zu finden ist [1]. Jede Organisation wird als *CVE Numbering Authority* (CNA) bezeichnet.

„CVE Numbering Authorities (CNAs) are organizations from around the world that are authorized to assign CVE IDs to vulnerabilities affecting products within their distinct, agreed-upon scope, for inclusion in first-time public announcements of new vulnerabilities." [17]

Jede CNA ist also authorisiert neue CVE-IDs für Schwachstellen zu erstellen, welche die Produkte der Unternehmung betreffen. Dabei wird darauf geachtet, dass die Teilnehmer nur neue Einträge für ihr individuelles und zuvor vereinbartes Fachgebiet erstellen sollen. Teilnehmende Organisationen sind unter anderem `Apple Inc.`, `Android` und `Google Inc.` [45]. Jeder dieser Partizipierenden kann nur CVE-IDs für ihren eigenen Bereich erstellen. So kann `Apple Inc.` nur Fehler bezüglich von Apple-Produkten hinzufügen. Für `Android` gilt das selbe und `Google Inc.` kann nur Einträge bezüglich *Chrome* und *Chrome OS* erstellen.

Aktuell gibt es 106960 CVE-Einträge [16]. Wie bereits erwähnt besteht ein Eintrag lediglich aus der Kennzeichnung, der Beschreibung und den Referenzen. Die CVE enthält also keine Bewertung für das Risiko, welches von einer gelisteten Schwachstelle ausgeht.

Damit Schwachstellen in dieser Hinsicht bewertet und somit auch miteinander verglichen werden können, wurde das Common Vulnerability Scoring System (CVSS) [12] entwickelt und gilt derzeit als Industriestardard [11].

„The Common Vulnerability Scoring System (CVSS) provides a way to capture the principal characteristics of a vulnerability, and produce a numerical score reflecting its severity, as well as a textual representation of that score." [12]

Das CVSS nutzt verschiedene Metriken, um prinzipielle Charakteristiken von

[2]Zum Beispiel ein Pfad zu einem git-Repository, welches die Schwachstelle enthält.

Schwachstellen zu ermitteln [12]. Mit Hilfe dieser Metriken kann folgend zu jeder Sicherheitslücke ein numerischer Wert errechnet werden, welcher zwischen 0.0 und 10.0 liegt [12]. Diese Werte werden in fünf unterschiedliche Kategorien unterteilt, welche die numerischen Werte textuell repräsentieren. Tabelle 3.1 zeigt die Einteilung der Kategorien.

Rating	**CVSS Score**
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

Tabelle 3.1: CVSS v3.0 Rating Scale [12]

3.2 Grundlagen Neuronaler Netzwerke

Der Bereich der neuronalen Netzwerke ist ein weitreichendes und komplexes Themengebiet. Im Laufe der Arbeit wird sich tiefergehend mit diesen Modellen beschäftigt. Um einen etwa gleichen Wissensstand in diesem Bereich voraussetzen zu können, werden in diesem Kapitel die grundlegenden Verhaltensweisen und Begrifflichkeiten bezüglich neuronaler Netze erklärt.

Um die Funktionen von einigen dieser Begriffe präzise zu erklären, werden sie anhand von kurzen Beispielen nachvollzogen.

3.2.1

Arbeitsweise

Die grundlegende Arbeitsweise eines neuronalen Netzwerks ist die Verarbeitung von zuvor deklarierten Eingaben zu einer Ausgabe. Diese Ausgabe eines neuronalen Netzes wird **Prediction** genannt. Dabei soll während der Verarbeitung eine gewisse Funktionalität von dem Netzwerk abgebildet werden. Beispielsweise soll ein Netzwerk die boolsche Funktion *XOR* abbilden. Somit hat das Netz zwei deklarierte Eingaben und eine Ausgabe. Die Funktion an sich wird allerdings nicht

in der Architektur des Netzwerks abgebildet, sondern sie soll von dem Modell *erlernt* werden.

Hierzu gibt es bei jedem neuronalen Netz die sogenannte **Lernphase**[3]. In dieser Phase werden dem Netzwerk Eingaben übergeben, zu welchen man die Ausgaben bereits kennt. Diese Daten werden als **Trainingsdaten** bezeichnet. Sind die gewünschten Ausgaben in den Trainingsdaten enthalten, so spricht man von **supervised learning** [56]. Es gibt auch Trainingsphasen bei denen die Ausgaben nicht bekannt sind. Diese fallen dann unter den Begriff **unsupervised learning** [56]. In dieser Arbeit werden allerdings ausschließlich Trainingsdaten genutzt, bei denen die Ausgaben bekannt sind. Wieviele Trainingsdaten für den Lernprozess benötigt werden, hängt maßgeblich von der abzubildenen Funktion ab. In dem einfach Falle der *XOR*-Funktion gibt es nur vier Datensätze. Diese sind in Tabelle 3.2 aufgeführt.

Datensatz-Nr.	Eingabe 1	Eingabe 2	Ausgabe
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Tabelle 3.2: Trainingsdaten für *XOR*-Funktion

Für die Abbildung einer komplexeren Funktionalität können aber auch einige Millionen Datensätze benötigt werden [32]. Wie das Modell diese Funktionalität mit Hilfe der übergebenen Daten lernt, wird in den folgenden Kapiteln näher beleuchtet.

Ist die Lernphase abgeschlossen, kann das neuronale Netzwerk optimaler Weise die gewünschte Funktionalität abbilden. Um herauszufinden wie gut sie umgesetzt wird, kann das neuronale Netz **evaluiert** werden. Hierzu werden wieder Daten des selben Aufbaus dem Netz übergeben. Diese Daten werden als **Testdaten** bezeichnet. Während der **Evaluation** werden allerdings die gewünschten Ausgaben mit den tatsächlichten Predictions des neuronalen Netzwerks bezüglich der jeweiligen Eingaben miteinander verglichen, um zu überprüfen wie genau die Funktionalität abgebildet wurde. Im Falle der *XOR*-Funktion stimmen die Testdaten mit den Trainingsdaten über ein. Eine mögliche Ausgabe der Evaluation wird in Tabelle 3.3 dargestellt. Die gewünschte Ausgabe wird während der Evaluation als **Target** bezeichnet und die Prediction wird **Actual** genannt.

[3] Wird häufig auch **Trainingsphase** genannt.

Datensatz-Nr.	Eingabe 1	Eingabe 2	Target	Actual
1	0	0	0	-8.881..E-16
2	0	1	1	0.99999999998
3	1	0	1	0.99999999999
4	1	1	0	2.220..E-16

Tabelle 3.3: Testdaten für *XOR*-Funktion

3.2.2

Error

Der **Error** bezeichnet die Differenz zwischen Target und Actual. Ein einfaches Beispiel erklärt das Prinzip. Angenommen ein neuronales Netz soll die Berechnung von Zentimeter zu Zoll erlernen. Bei der Umwandlung besteht eine lineare Abhängigkeit die durch eine Konstante **c** dargestellt werden soll.

```
X Zentimeter = X Zoll * c
```

In der Lernphase muss die Konstante c angenähert werden. Hierzu zeigt die Tabelle 3.4 die Trainingsdaten. Aufgrund der linearen Abhängigkeit der Eingabe zur Ausgabe sind für dieses Beispiel lediglich zwei Datensätze aufgeführt.

Datensatz-Nr.	Zentimeter	Zoll
1	0	0
2	100	39,3701

Tabelle 3.4: Trainingsdaten für Zentimeter-zu-Zoll-Umrechnung

Damit die Phase starten kann, muss die Konstante zufällig initialisiert werden. Gehen wir hier von einem Wert von 0.3 aus. Mit diesem Wert gäbe das neuronale Netzwerk die in Tabelle 3.5 dargestellten Evaluationsdaten aus.

Datensatz-Nr.	Zentimeter	Target	Actual
1	0	0	0
2	100	39,3701	30

Tabelle 3.5: Testdaten für Zentimeter-zu-Zoll-Umrechnung

Wie oben bereits erwähnt ergibt sich der Error aus der folgenden Differenz:

```
Error = Target - Actual

9,3701 = 39,3701 - 30,0
```

Die reine Berechnung ist unverkennlich äußerst trivial. Dennoch spielt der Error eine wichtige Rolle in dem eigentlich Lernprozess eines neuronalen Netzwerks.

Dies wird in den folgenden Kapiteln näher erläutert. Die Errechnung des Fehlerwerts wird auch **Kostenfunktion** genannt. Im Englischen gibt es einige Synonyme. Diese sind **error function**, **loss function** und **cost function**.

Die hier vorgestellte Kostenfunktion ist die einfachste. Es gibt noch den *Absolut Error*, welcher immer nur positive Werte hat und den *Squared Error*, welcher den einfachen Fehlerwert quadriert.

3.2.3

Aktivierungsfunktion

Neuronale Netzwerke bestehen aus künstlichen Neuronen. Ein einzelnes künstliches Neuron bekommt eine gewisse Anzahl von Eingaben $x = a_0 \ldots a_n$ für $n = |Eingaben|$ und produziert daraus eine Ausgabe y. Diese Ausgabe wird gegeben, wenn ein gewisser Schwellwert erreicht wird. Dieser Prozess wird durch die sogenannte **Aktivierungsfunktion** abgebildet.

Die einfachste Art einer Aktivierungsfunktion wird *Stufenfunktion* genannt. Wenn der Schwellwert s_0 erreicht wird, gibt diese Funktion $y = 1$ aus, ansonsten Null. Die Abbildung 3.1 zeigt den Graphen der Stufenfunktion und Gleichung (3.1) gibt die formale Definition.

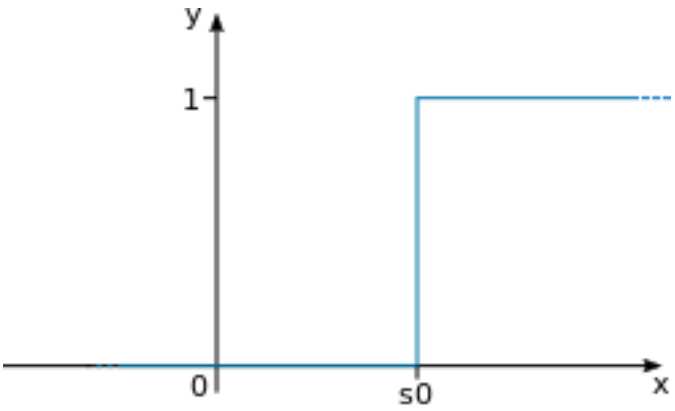


Abbildung 3.1: Graph der Stufenfunktion

$$y = \begin{cases} 1 & \text{für } x \geq s_0 \\ 0 & \text{für } x < s_0 \end{cases} \tag{3.1}$$

Künstliche Neuronen sind den Neuronen aus dem menschlichen Gehirn nachempfunden[4]. Da es in der Natur allerdings selten exakte Übergänge von Null auf

[4]Diese „Originale“werden im Folgenden als *echte Neuronen* bezeichnet.

Eins, wie in der Stufenfunktion, gibt, werden viele verschiedene Aktivierungsfunktionen verwendet, welche das Verhalten von künstlichen Neuronen natürlicher und realistischer beschreiben sollen, sodass sie den echten Neuronen mehr ähneln. Eine für diesen Zweck oft gewählte Funktion ist die *Sigmoidfunktion*. Sie wird in Abbildung 3.2 gezeigt und in Gleichung (3.2) formal definiert.

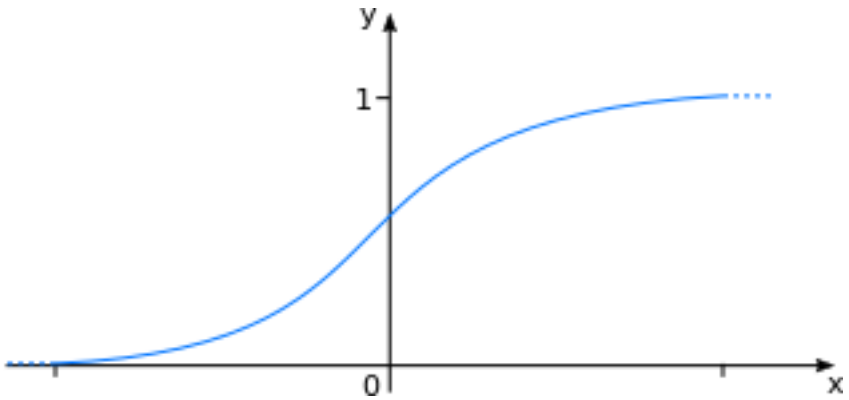


Abbildung 3.2: Graph der Sigmoidfunktion

$$y = \frac{1}{1 + e^{-x}} \tag{3.2}$$

Für die Backpropagation, welche in Kapitel 3.2.5 erklärt wird, ist es wichtig, dass die Aktivierungsfunktion differenzierbar ist [46]. Dies ist durch die Sigmoidfunktion gewährleistet.

3.2.4

Weights

Der Vorteil von neuronalen Netzwerken ist, dass sie lernen können. Sie passen ihr Verhalten in der Lernphase entsprechend der zugeführten Eingaben und den jeweiligen gewünschten Ausgaben an. Dies wird durch die grundlegende Architektur dieser Netzwerke erreicht. Abbildung 3.3 zeigt den Aufbau eines einfachen neuronalen Netzes. Es hat zwei Schichten mit jeweils zwei Knoten. Jeder Knoten einer Schicht ist mit allen Knoten der nachfolgenden Schicht vernetzt und gibt somit die produzierte Ausgabe des Knotens an alle nachstehenden Knoten weiter.

Um ein Netzwerk lernfähig zu machen, hat jede Verbindung zwischen den Knoten ein **Gewicht** (engl. **Weight**). Dieses gibt an wie groß der Einfluss des Knotens auf den Folgeknoten ist. Je größer der Absolutbetrag des Gewichts, desto größer ist auch der Einfluss [41]. Liegt der Wert bei Null, so hat der Knoten momentan

keinen Einfluss auf den Folgeknoten [41]. Eine Momentaufnahme der Werte aller Gewichte in einem neuronalen Netzwerk gibt das *Wissen* an, welches in dem Netz zu diesem Zeitpunkt gespeichert ist [41]. Der Prozess des Lernens wird durch die Änderung der Gewichte definiert. Wie dies genau geschieht, wird in Unterkapitel 3.2.5 beschrieben.

„*Learning* [...] is about finding weights that make the NN exhibit *desired* behavior [...].“ [49]

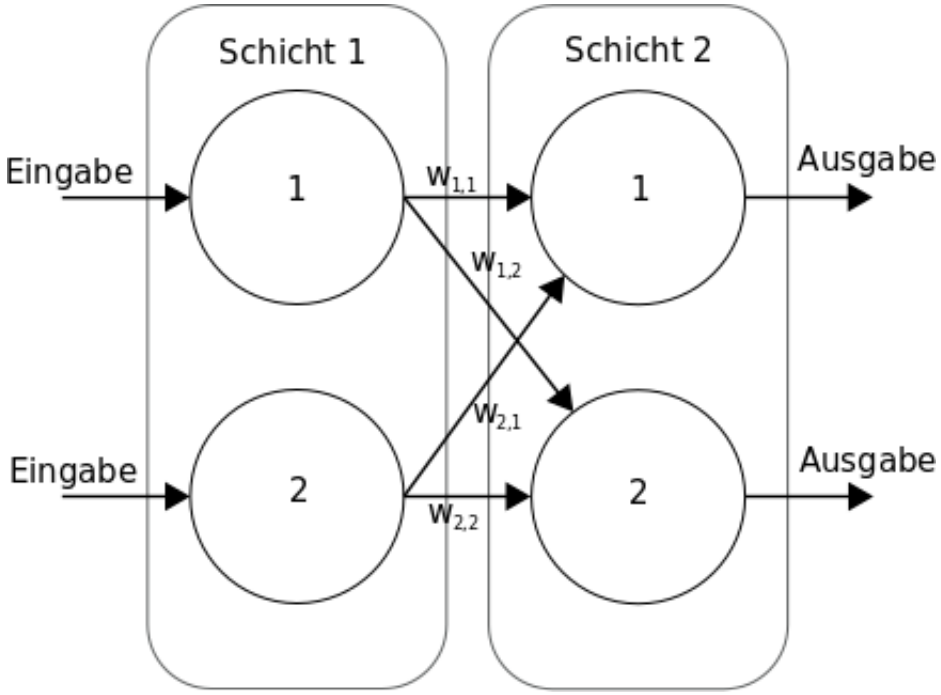


Abbildung 3.3: Architektur eines einfachen neuronalen Netzes

Wie in Abbildung 3.3 an den Verbindungen der Knoten zu erkennen ist, werden die Gewichte wie folgt definiert:

Sei S die Menge und $n \in \mathbb{N}^+$ die Anzahl aller Schichten eines neuronalen Netzwerks. Sei weiter K_i die Menge und $m_i \in \mathbb{N}^+$ die Anzahl der Knoten aus Schicht i mit $i \in \mathbb{N}^+ \wedge i \leq n$. Dann ist das Gewicht definiert als $w_{k,l}$ mit $j \in \mathbb{N}^+ \wedge j < n$, sodass $k, l \in \mathbb{N}^+ \wedge k \leq m_j \wedge l \leq m_{j+1}$.

Man kann aus der Abbildung weiter erkennen, dass jeder Knoten einer Schicht mit jedem Knoten der Folgeschicht in Relation steht. Sei diese Relation W_R. Laut Architektur eines neuronalen Netzes ist W_R links- sowie rechtstotal. Die allgemeine Linkstotalität ist in Gleichung 3.3 definiert.

$$\begin{aligned} &\forall\, k \in \mathbb{N}^+ \wedge k \leq m_j \; mit \; j \in \mathbb{N}^+ \wedge j \; < \; n : \\ &\qquad \exists\, l \in \mathbb{N}^+ \wedge l \leq m_{j+1} : \; (k,l) \in W_R \end{aligned} \tag{3.3}$$

Da dies allerdings nur aussagt, dass es für alle k *mindestens* ein l geben muss, kann die Anforderung an die Relation wie folgt verschärft werden.

$$\begin{aligned} \forall\, k \in \mathbb{N}^+ \wedge k \leq m_j \; mit \; j \in \mathbb{N}^+ \wedge j \; < \; n : \\ \forall\, l \in \mathbb{N}^+ \wedge l \leq m_{j+1} : \; (k,l) \in W_R \end{aligned} \tag{3.4}$$

Durch die Verschärfung der Linkstotalität ist die Relation W_R automatisch auch rechtstotal. Die Definition der Rechtstotalität wird damit als trivial betrachtet und somit weggelassen.

3.2.5

Backpropagation

Die Lernfähigkeit ermöglicht es neuronalen Netzwerken mit ihrem Verhalten komplexe Funktionen abzubilden. In den vorigen Kapiteln wurden wichtige Elemente erläutert, welche für die Trainingsphasen von fundamentaler Bedeutung sind. Wie bereits erwähnt, wird das Lernen durch die Anpassung der Gewichte zwischen den einzelnen Knoten definiert. Die Änderung der Gewichte hat den *Error* als Ausgangspunkt. Dieser soll in das Netz zurückgeführt und rückwärts durch alle Knoten und Verbindungen gereicht werden (engl. **Backpropagation**). In Kapitel 3.2.2 wurde beschrieben, wie der Error berechnet wird. Nun soll gezeigt werden, wie der Error für die Anpassung der Gewichte genutzt wird.

Zunächst ist festzuhalten, dass dieser Fehlerwert durch das gesamte Verhalten des neuronalen Netzes zu Stande kommt. Folglich hat jeder Knoten in dem Modell zu diesem Error beigetragen. Dies impliziert, dass jedem einzelnen Neuron ein Teil dieses Errors zugewiesen werden muss. Die Zuweisung wird über die *Gewichte* der Verbindungen geregelt. Die Fehlerwerte aller Knoten, bis auf die der Endknoten, da diese ja durch die Differenz von Zielwert und tatsächlichem Wert beschrieben sind, werden wie in Gleichung 3.5 definiert.

$$e_{i,j} = \sum_{n=1}^{m_{i+1}} e_{i+1,n} * (\frac{w_{j,n}}{\sum_{l=1}^{m_i} w_{l,n}}) \tag{3.5}$$

mit m_i als Anzahl der Knoten in Schicht i und $j \in \mathbb{N}^+ \wedge j \leq m_i$. Diese Zuweisung soll nun anhand eines kurzen Beispiels verdeutlicht werden. In Abbildung 3.4 sind die letzten beiden Schichten eines neuronalen Netzes zu sehen. Es wird davon ausgegangen, dass das Netzwerk zwei Ausgaben hat. Es wird weiter angenommen, dass die beiden Error bereits errechnet wurden. Dies sind die Werte

0,4 für die erste Ausgabe und 0,7 für die zweite. Diese beiden Fehlerwerte sollen nun rückwärts durch das Netzwerk propagiert werden. Wendet man die Formel aus Gleichung 3.5 an, so erhält man für das erste Neuron der vorletzten Schicht einen Error von 0,44 und für das zweite Neuron einen Wert von 0,66. Alle Dinge, die mit der Rückführung des Errors zu tun haben, sind in rot markiert.

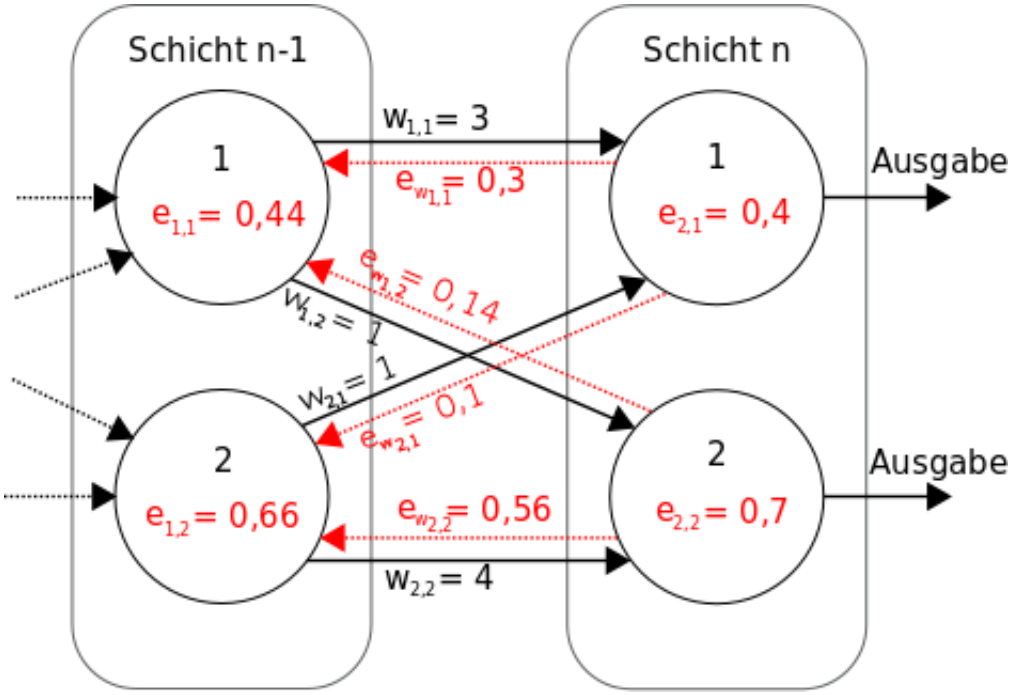


Abbildung 3.4: Beispiel: Rückführung des Errors

Für die Anpassung der Gewichte und somit für die Lernfähigkeit des neuronalen Netzes sind diese errechneten Fehlerwerte von wichtiger Bedeutung. Um das gewünschte Verhalten für ein Modell zu erreichen, muss nun der Error minimiert werden. Dafür muss überprüft werden inwiefern sich der Error in Abhängigkeit zu sich ändernden Gewichten anpasst.

$$\frac{\delta E}{\delta w_{j,k}} \tag{3.6}$$

Ausdruck 3.6 [44] zeigt diese Abhängigkeit. Sie sagt aus wie sich der Error E verändert, wenn sich das Gewicht $w_{j,k}$ ändert. Sie zeigt somit die Steigung der Kostenfunktion, welche minimiert werden muss. Gleichung 3.7 [44] zeigt die durchgeführten Ableitungen.

$$\frac{\delta E}{\delta w_{j,k}} = -(t_k - o_k) * \text{sigmoid}(\sum_j w_{j,k} * o_j)(1 - \text{sigmoid}(\sum_j w_{j,k} * o_j)) * o_j \tag{3.7}$$

Auf eine schrittweise Herleitung wird in diesem Fall aufgrund von Platzgründen

verzichtet. Es wurde die Kettenregel angewendet, sodass zwei Terme auf der linken Gleichungsseite abgeleitet werden mussten. Der erste Faktor auf der rechten Gleichungsseite ist die Ableitung der Kostenfunktion. Hier wurde der *Squared Error* als Funktion gewählt. Die letzten drei Faktoren ergeben sich aus der Ableitung der Ausgabe des Knoten o_k, welche durch die Aktivierungsfunktion gebildet wird. Hier wurde die Sigmoidfunktion verwendet.

Die Berücksichtigung der Kostenfunktion in der Ableitung impliziert, dass sich Gleichung 3.7 auf die Abhängigkeit des Errors in Bezug auf die Ausgabe der Ausgabeschicht des neuronalen Netzes bezieht, denn nur in der letzten Schicht wird die Kostenfunktion für die Errechnung des Errors benötigt, wie in Kapitel 3.2.2 beschrieben wurde. Für die anderen Schichten wird für die Berechnung der rückwärts propagierte Fehlerwert verwendet. Gleichung 3.8 [44] zeigt die entsprechende Anpassung der Gleichung 3.7.

$$\frac{\delta E}{\delta w_{i,j}} = -(e_j) * \text{sigmoid}(\sum_i w_{i,j} * o_i)(1 - \text{sigmoid}(\sum_i w_{i,j} * o_i)) * o_i \tag{3.8}$$

Der errechnete Error für das Neuron j wird hier mit e_j bezeichnet. Mit diesen Gleichungen können nun die Gewichte des neuronalen Netzwerks angepasst werden. Die Gewichte werden entsprechend der errechneten Steigung der Kostenfunktion angepasst. Ist die Steigung positiv, so muss das Gewicht verringert werden, da sonst der Fehlerwert steigt. Wenn die Steigung negativ ist, muss das Gewicht erhöht werden, da so der Error verkleinert werden kann. Wenn die Steigung bei Null liegt, so muss das Gewicht nicht angepasst werden. Formel 3.9 zeigt die Anpassung eines Gewichts $w_{j,k}$.

$$w_{j,k,t+1} = w_{j,k,t} - \beta * \frac{\delta E}{\delta w_{i,j,t}} \tag{3.9}$$

Die Änderung wird durch den Faktor β beeinflusst. Dieser Faktor wird als **Lernrate** (engl. **Learning Rate**) bezeichnet. Dieser Faktor hat in den meisten Fällen einen sehr kleinen Wert (ca. 0,001 o. Ä.) [6]. Der Wert darf nicht zu groß und nicht zu klein sein, da sonst die Sprünge für die Anpassungen zu groß oder zu klein sind [6].

3.3

Typen Von Neuronalen Netzwerken

Laut der Anforderung **[NRE3]** soll der Erkennungsprozess von eventuellen Schwachstellen durch verschiedene Arten von neuronalen Netzwerken realisiert werden, damit die Tauglichkeit dieser Modelle im Anschluss evaluiert werden kann. In diesem Kapitel sollen die unterschiedlichen Arten von neuronalen Netzen dargelegt werden. Es wird auf die Eigenschaften des Aufbaus der Modelle und die damit einhergehenden Verhaltensweisen eingegangen.

3.3.1

Feed Forward

Feed-Forward-Netzwerke gehören zu den einfachsten Arten von neuronalen Netzen. Sie bestehen aus genau drei verschiedenen Schichten (engl. Layer):

Input Layer Die künstlichen Neuronen dieser Schicht nehmen die Eingaben an und geben sie unverändert an die nächste Schicht. Sie werden in den folgenden Diagramm mit einem gelben Punkt dargestellt.

Hidden Layer Hier werden die übergebenen Eingaben entsprechend der Aktivierungsfunktion verarbeitet und eine Ausgabe erzeugt, welche an die nächste Schicht weitergegeben wird. Sie werden mit einem grünen Punkt kenntlich gemacht.

Output Layer Diese Schicht erzeugt die letztendliche Ausgabe des neuronalen Netzes. Hier kann der Error bestimmt werden, welcher durch das restliche Netz rückpropagiert werden soll. Sie sind in den Diagrammen mit einem orangen Punkt dargestellt.

Abbildung 3.5 zeigt ein Feed-Forward-Netz mit je zwei Neuronen in den ersten beiden Schichten und einem Ausgabeneuron.

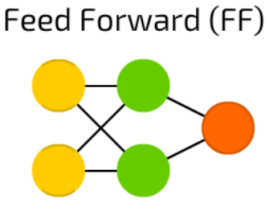


Abbildung 3.5: Beispiel: Feed-Forward-Netzwerk [33]

3.3.2

Deep Feed Forward

Ein **Deep-Feed-Forward**-Netzwerk[5] ist dem Feed-Forward-Netz sehr ähnlich. Die Schichten und Neuronenarten sind die gleichen. Sie unterscheiden sich lediglich in der Menge der Hidden Layer. Ein Feed-Forward-Modell hat immer genau eine solche Schicht. Bei DFF-Netzwerken kann die Menge von Hidden Layern jedoch variieren. Abbildung 3.6 zeigt ein beispielhaftes Modell mit drei Neuronen im Input Layer, zwei Hidden Layer mit jeweils vier Neuronen und einen Output Layer mit zwei Neuronen.

In den bisherigen Erläuterungen und Beispielen zu neuronalen Netzwerken in den vorigen Kapiteln wurde sich immer auf diese Art von Modell bezogen.

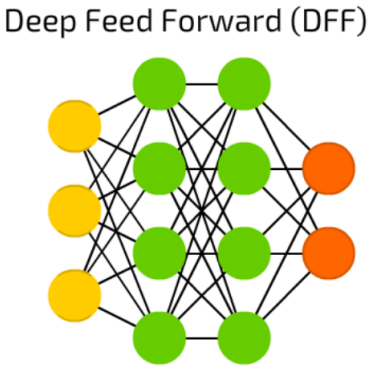


Abbildung 3.6: Beispiel: Deep-Feed-Forward-Netzwerk [33]

3.3.3

Recurrent Neural Network

Die Architektur von **Recurrent-Neural-Network**s[6] ähnelt der von DFF-Netzwerken stark. Dennoch gibt es einen wichtigen Unterschied, welcher die Ar-

[5] Weiterhin nur noch als DFF-Netzwerke bezeichnet.
[6] Im weiteren Verlauf dieser Arbeit werden diese nur noch als RNN bezeichnet.

beitsweise und damit auch die Einsatzgebiete von RNNs fundamental ändert.

Recurrent Neuronen Die Neuronen der Hidden Layer nehmen ihre eigene Ausgabe zeitversetzt als Eingabe. Dabei kann die zeitliche Versetzung variabel gehalten werden. Diese Art von Neuron ermöglicht es dem RNN die Ausgaben der einzelnen Knoten über eine gewisse Schrittanzahl zu speichern, um sie dann wieder in das Netz zurückführen zu können. Dadurch entsteht eine Art von Kontextbewusstsein, da auf vergangene Ausgaben im Nachinein noch reagiert werden kann. Sie werden durch einen blauen Punkt dargestellt.

Abbildung 3.7 zeigt ein beispielhaftes RNN mit jeweils drei Neuronen in den unterschiedlichen Schichten.

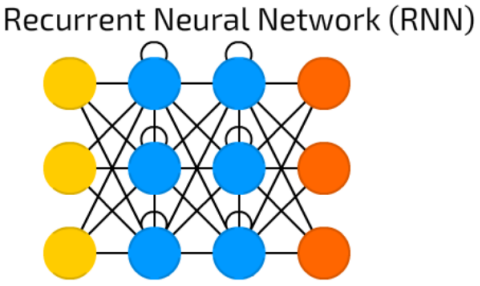


Abbildung 3.7: Beispiel: Recurrent-Neural-Network [33]

Ein großes Einsatzgebiet der RNNs ist die Verarbeitung von Texten. Durch die vorhandene Architektur der Hidden Layer und das damit einhergehende Kontextbewusst sein, sind RNNs prädestiniert für diese Art von Aufgaben, da sie sich durch ihre Verhaltsweise zum Beispiel die letzten fünf Worte "merken" können.

Allerdings stoßen RNNs genau mit dieser Form der Speicherung von Ausgaben an eine Grenze. Und zwar kann sich ein RNN eine Ausgabe nur über eine gewisse Zeit merken und wird dann überschrieben. Somit kann ein RNN zwar auf vergangene Ausgaben reagieren, allerdings nur über einen bestimmten Zeitraum hinweg. Die in dem nächsten Kapitel erläuterte Form von neuronalen Netzwerken ist eine spezielle Form von RNNs, welche dieses Problem nicht haben.

3.3.4

Long Short-Term Memory Neural Network

Ein **Long Short-Term Memory Neural Network**[7] ist eine spezielle Form von RNN. Die grundlegende Architektur ist gleich zu einem RNN. Der Unterschied ist wieder in den einzelnen Neuronen der Hidden Layer zu finden.

Memory Neuronen Genau wie die *Recurrent Neuronen* eines RNN geben sie ihre Ausgaben zeitverzögert an sich selbst weiter. Allerdings haben die Neuronen eines LSTM noch eine neue Eigenschaft. Sie besitzen einen Status. In diesem Status werden Informationen über vergangene Ausgaben und Status gespeichert. Dieser Status wird von Schritt zu Schritt gelesen, angepasst und weitergegeben und befindet sich in der Form eines Vektors. Sie werden durch einen blauen Punkt mit einem Kreis dargestellt.

Abbildung 3.8 zeigt einen beispielhaften Aufbau eines LSTMs. Dabei hat das Modell jeweils drei Neuronen in den verschiedenen Schichten.

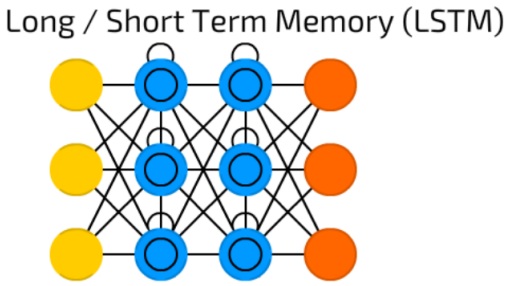


Abbildung 3.8: Beispiel: Long Short-Term Memory-Neural-Network [33]

Die grundlegende Arbeitsweise eines LSTMs mit der Verwendung des Status soll mit Hilfe eines kurzen Beispiels erläutert werden. Angenommen ein LSTM soll für die Textverarbeitung verwendet werden. Gewünscht ist, dass es aufgrund von zuvor gegebenen Worten eines Satzes das nächste Wort im Satz vorhersagen soll. In diesem fiktiven Beispiel soll zu einem Personalpronomen der passende Possessivartikel vorhergesagt werden.

Bei jedem Wort, welches das Netz als Eingabe bekommt, wird zunächst überprüft, welche Informationen des Status nicht mehr benötigt werden. Wird in dem Beispiel ein neues Personalpronomen eingelesen, zum Beispiel `er`, so wird ein eventuell bereits zuvor im Status gespeichertes nicht mehr benötigt und kann "vergessen" werden, zum Beispiel `sie`. Im nächsten Schritt wird dann definiert,

[7] Weiterführend nur noch als LSTM bezeichnet.

welche Informationen angepasst werden sollen. In diesem Fall das Personalpronomen. Dann wird der neue Kandidat für den Statusvektor definiert. Schließlich werden alle Änderungen durchgeführt, sodass das neu eingelesene Pronomen das alte ersetzt hat.

Der Status wird also von Schritt zu Schritt weitergegeben, gelesen und verändert. Er kann als eine Art Speicher angesehen werden. Dieser Speicher kann allerdings im Gegensatz zu dem vom RNN Informationen über eine längere Zeit halten. Dadurch kann ein LSTM bei der Textverarbeitung auch Bezug auf bereits länger vergangene Worte nehmen.

Michael freut sich. *Er* hat ein neues Handy zu Weihnachten bekommen. Auch wenn viele andere Geschenke unter dem Weihnachtsbaum lagen, ist das Handy *sein* liebstes Geschenk.

3.4 Code Clone Detection

Die zu entwickelnde Software soll mögliche Sicherheitslücken in Quellcode erkennen. Dazu wird der Code auf die Ähnlichkeit zu bereits bekannten Schwachstellen geprüft. Können zwischen den Codefragmenten eindeutige Parallelen gezogen werden, so kann es sich um einen Klon handeln. In diesem Kapitel sollen die in dieser Arbeit verwendeten Begrifflichkeiten im Bezug auf Code Clone Detection erläutert werden. Dabei wird auf die Ausarbeitung von Sheneamer et al. [52] Bezug genommen.

3.4.1 Begrifflichkeiten

Code-Fragment Sheneamer et al. definiert ein Code-Fragment wie folgt:

> „A code fragment (CF) is a part of the source code needed to run a program. It usually contains more than five statements that are considered interesting, but it may contain fewer than five statements. It can contain a function or a method, begin-end blocks or a sequence of statements. “[52]

Ein Code-Fragment besteht also aus mehreren Statements und ist Teil des

Quellcodes, welcher zum Ausführen des Programms benötigt wird. Es kann eine Funktion oder eine Methode enthalten und `begin`- und `end`-Blöcke haben. Listing 3.1 zeigt ein beispielhaftes Code-Fragment.

Listing 3.1: Beispiel: Code-Fragment

```
1 public String toString(String[] input) {
2   String result = "";
3   for (int i = 0; i < input.length(); i++) {
4     result += input[i] + ",";
5   }
6   result = result.substring(0, result.length()-1);
7   return result;
8 }
```

Code-Klon / **Klonpaar [52]** Seien *CF1* und *CF2* zwei Code-Fragmente. Wenn syntaktische oder semantische Analogien zwischen diesen Code-Fragmenten bestehen, also in der entsprechenden Relation gleich oder ähnlich sind, so spricht man von einem **Klonpaar** *(CF1,CF2)*. Des Weiteren wird jedes Code-Fragment eines Klonpaars als ein **Code-Klon** des jeweils anderen Fragments des selben Paars bezeichnet.

3.4.2

Typen von Klonpaaren

Nach der Ausarbeitung von Sheneamer et al. [52] können Klonpaare in zwei unterschiedliche Klassen unterteilt werden. Die erste Gruppe beinhaltet Klone, die aufgrund einer syntaktischen und textuellen Ähnlichkeit als Klonpaar definiert werden. Diese Klasse enthält wiederum drei verschiedene Klontypen, auf die später noch ein genauerer Blick geworfen wird. In der zweiten Gruppe sind die Klonpaare enthalten, die wegen ihrer semantischen Analogie als Klonpaar identifiziert werden. Alle in dieser Gruppe enthaltenen Elemente werden als *Typ-4*-Klon bezeichnet. Folglich gibt es vier verschiedene Klontypen. Diese werden nachfolgend näher beschrieben. Außerdem werden die Listings 3.3-3.6 beispielhaft veranschaulichen wie jeder Klontyp zu einem Original-Code-Fragment (Listing 3.2) aufgebaut sein kann. Die jeweiligen Änderungen an dem Original werden zusätzlich textuell herausgestellt.

Typ-1-Klon Ein Klonpaar dieses Typs enthält Code-Fragmente, welche exakte

Kopien voneinander sind. Hiervon ausgenommen sind Whitespaces und Kommentare.

Typ-2-Klon Ein Klonpaar dieses Typs ist bereits ein Typ-1-Klon. Darüber hinaus können sich die Namen der Variablen, Typen, Literale und Methoden der jeweiligen Code-Fragmente unterscheiden.

Typ-3-Klon Ein Klonpaar dieses Typs ist bereits ein Typ-2-Klon. Zusätzlich kann einer der Code-Klone einzelne Statements im Quellcode haben, welche bei dem anderen Klon des Paars fehlen.

Typ-4-Klon Die Code-Fragmente eines Klonpaars aus der zweiten Klasse der Klontypen weisen keinerlei syntaktische Ähnlichkeit auf. Stattdessen findet man bei den Code-Klonen semantische Analogien.

Tabelle 3.6: Beispiele für die Klontypen 1-4

Listing 3.2: Beispiel Klontypen: Original-Quellcode

```
1  switch (month) {
2    case 1:
3      sMonth = "Januar";
4      break;
5    // Weitere Monate
6    case 12:
7      sMonth = "Dezember";
8      break;
9    default:
10     throw new Exception("↩
           Ungültiger Monat");
11 }
```

Listing 3.3: Beispiel Klontypen: Typ-1-Klon

```
1  switch (month) {
2    case 1:
3      sMonth="Januar";// ↩
           Zuweisung 'Januar'
4      break;
5    // Weitere Monate
6    case 12:
7      sMonth ="Dezember"; // ↩
           Zuweisung 'Dezember'
8      break;
9    default:
10     throw new Exception("↩
           Ungültiger Monat");
11 }
```

Listing 3.4: Beispiel Klontypen: Typ-2-Klon

```
1  switch (monat_Zahl) {
2    case 1:
3      monat_Wort ="Januar";//↩
           Zuweisung 'Januar'
4      break;
5    // Weitere Monate
6    case 12:
7      monat_Wort= "Dezember";↩
           //Zuweisung '↩
           Dezember'
8      break;
9    default:
10     throw new Exception("↩
           Ungültiger Monat");
11 }
```

Listing 3.5: Beispiel Klontypen: Typ-3-Klon

```
1  switch (monat_Zahl) {
2    case 1:
3      monat_Wort="Januar";// ↩
           Zuweisung 'Jan'
4      break;
5    // Weitere Monate
6    case 12:
7      monat_Wort= "Dezember";↩
           //Zuweisung 'Dez'
8      break;
9    default:
10     System.out.println("-- ↩
           Ungültiger Monat");
11     throw new Exception("↩
           Ungültiger Monat");
12 }
```

Listing 3.6: Beispiel Klontypen: Typ-4-Klon

```
1  if (month == 1)
2    sMonth = "Januar";
3  // Weitere Monate
4  if (month == 12)
5    sMonth = "Dezember";
6  else
7    throw new Exception("Ungü↩
       ltiger Monat");
```

In Tabelle 3.6 sind die Listings aufgeführt, welche den Originalen Codeausschnitt und die jeweiligen Klontypen enthalten. Listing 3.2 zeigt den originalen Software-Code. Listing 3.3 zeigt das Code-Fragment, welches ein Klon vom Typ 1 ist. Es ist, bis auf die Kommentare, eine exakte Kopie des Originals. Für den Typ-2-Klon, welcher in Listing 3.4 zu sehen ist, wurden zusätzlich die Namen der Variablen geändert. In Listing 3.5 wurde in dem `default`-Fall ein Ausgabe-Statement hinzugefügt. Der semantische Klon aus Listing 3.6 hat augenscheinlich wenig syntaktische Ähnlichkeit mit dem Original-Ausschnitt. Der Quellcode tut allerdings auf semantischer Ebene das gleiche, sodass er ein Klon vom Typ 4 ist.

KAPITEL 4

Entwurf

In diesem Kapitel wird der Entwurf und das Konzept des zu entwickelnden Prototypen definiert und beschrieben. Die getroffenen Entscheidung bezüglich der Architektur des Software werden hierbei näher beleuchtet. Dabei wird auf die nichtfunktionalen Anforderungen **[NR4]** und **[NRE4]** geachtet. Diese fordern einen modularen Aufbau der Software im Allgemeinen, sowie speziell für die verschiedenen Kernalgorithmen.

Aufgrund dieser beiden Anforderungen wird das Entwurfs-Kapitel in verschiedene Unterabschnitte geteilt. Zunächst soll der allgemeine Datenfluss in der Software in Abschnitt 4.1 erläutert werden. Hier werden spezifische Details noch als Blackbox betrachtet. In Kapitel 4.2 wird auf die Wahl der zu evaluierenden Netzwerktypen einegangen. Anschließend soll in Kapitel 4.3 die Schnittstelle für die verschiedenen neuronalen Netzwerke definiert werden (**[NRE4]**). Es wird genau beschrieben, was das Modell als Eingabevektoren erwartet und wie sich diese Vektoren zusammensetzen. Ebenfalls soll in diesem Kapitel gezeigt werden, wie die Eingabevektoren erstellt werden. Es wird definiert, wie Software-Quellcode zu einem Eingabevektor transformiert werden kann. In Kapitel 4.4 werden zwei Frameworks vorgestellt, mit denen Methoden des maschinellen Lernens umgesetzt werden können. In dem darauf folgenden Kapitel 4.5 werden die bis dahin beschriebenen Ergebnisse gebündelt und in Diagrammen dargestellt.

4.1 Datenfluss

Für das System gibt es zwei simultan ablaufende Datenflüsse. Der Grund dafür ist, dass die Software grundlegend zwei verschiedene Arten von Eingaben hat. Zum einen sind dies die Dateien, welche den Code von den bereits bekannten Security-Schwachstellen enthält. Diese werden für die Trainingsphase des neuronalen Netzes benötigt und werden deshalb im weiteren Verlauf als *Trainingsdaten* bezeichnet. Zum anderen gibt es die Eingaben des Nutzers. Dies sind die Dateien, die den Quellcode enthalten, welcher auf mögliche Schwachstellen getestet werden soll. Diese Dateien werden fortlaufend *Testdaten* genannt.

Sowohl die Trainings- als auch auch die Testdaten müssen ein Preprocessing durchlaufen, um aus den Dateien einen Eingabevektor für das Netzwerk zu generieren. Diese Schritte werden in dem nachfolgenden Kapitel 4.3 näher beschrieben. Abbildung 4.1 zeigt den grundlegenden Datenfluss visuell.

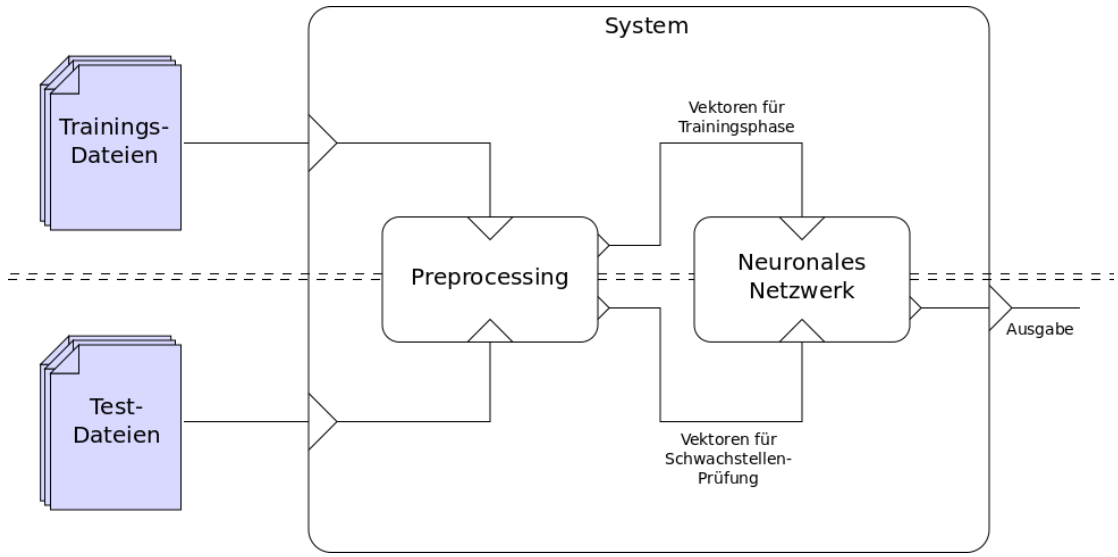


Abbildung 4.1: Datenfluss im System

Die gestrichelte Linie stellt eine Abgrenzung der beiden Flüsse dar, da sie zeitlich nicht parallel laufen müssen. Die Ein- und Ausgaben des Preprocessings wurden hier lediglich für die visuelle Darstellung unterteilt. Selbstverständlich werden die Trainings- und Testdaten dem gleichen Preprocessing unterzogen.

4.2 Typen der neuronalen Netzwerke

Ein wichtiger Bestandteil dieser Masterthesis ist die Evaluation von verschiedenen neuronalen Netzen hinsichtlich ihrer Tauglichkeit im Bereich des Code-Clone-Detections. Da es viele verschiedene Typen von Netzwerken gibt, muss es für diese Ausarbeitung eine Einschränkung geben, da nicht alle evaluiert werden können. Im Speziellen sollen zwei Arten mit einander verglichen werden, welche anschließend als Kernalgorithmus für den Erkennungsprozess dienen soll. Dabei handelt es sich um die zwei wohl am weit verbreitetsten Typen: das Deep-Feed-Forward-Network und das Long Short-Term Memory Neural Network.

Das DFF wurde gewählt, da es aufgrund seiner Architektur sehr gut für Vorhersagen und Klassifikationen verwendet werden kann. Beale und Jackson [7] beschreiben die Eignung für das Erkennen von Mustern in Daten und die damit einhergehende Möglichkeit gute Vorhersagen treffen zu können.

Das LSTM wurde gewählt, da es bereits in der Text- und Dokumentenverarbeitung eine bemerkenswerte Performanz erreichen kann [66]. Um zu überprüfen, ob dies auch für eine Klassifikation von Software-Quellcode genutzt werden kann, soll es in dieser Arbeit mit evaluiert werden.

4.3 Preprocessing

Neuronale Netzwerke haben ein vielseitiges Einsatzgebiet. Doch egal in welchem Bereich und für welche Funktion sie genutzt werden, die einzelnen Neuronen arbeiten mit numerischen Werten [14], [19]. Dabei ist auch der Typ des Netzes nicht von Bedeutung. Ob ein Netzwerk nun für die Text- oder die Bildverarbeitung verwendet wird, letztendlich werden die Eingaben auf numerische Werte gemünzt. Dieser Prozess der Verarbeitung der Eingaben wird **Preprocessing** genannt. Er ist von fundamentaler Bedeutung für neuronale Netze [36],[37],[60],[8]. Während dieses Vorgangs werden die *Rohdaten* auch auf mögliche Fehler überprüft (z. B. fehlende Werte in einer Daten-Sequenz oder eventuelle Ausschläge von gemessenen Daten). Die Rohdaten werden also Aufbereitet und dann in eine

passende Form für das neuronale Netz gebracht.

Auch für die zu entwickelnde Software muss ein solches Preprocessing implementiert werden. Folglich muss der Code aus den Trainings- und Testdaten in eine Vektorform transformiert werden. In den folgenden Unterabschnitten werden die Grundlagen für den verwendeten Preprocessing-Algorithmus dargelegt.

4.3.1

Eingabevektor

Aufgrund der Anforderung **[NRE4]** sollen die Kernalgorithmen, also die neuronalen Netzwerke, modular austauschbar sein, sodass die Software in Zukunft eventuell durch Implementierungen von anderen Typen von neuronalen Netzen erweitert werden kann. Um dies zu gewährleisten zu können, müssen die Modelle einen einheitlichen Vektor als Eingabe bekommen.

Wie in Abbildung 4.1 zu erkennen ist, bekommt das System als Eingabe Dateien mit Quellcode. Dieser muss also in einen numerischen Eingabevektor für das Modell umgeformt werden. Dieser Vektor muss allerdings eine feste Länge haben, da die Eingaben für das neuronale Netzwerk in ihrer Länge nicht variieren dürfen. Der Grund dafür ist, dass die erste Schicht des Netzwerks (Input Layer) immer eine feste Anzahl von Neuronen hat. Dies ist kein triviales Problem, denn die Eingabe für den Preprocessing-Algorithmus ist Software-Code, welcher in seiner Länge und seinem Aufbau variiert. Es muss gewährleistet sein, dass jeder eingegebene Code in einen numerischen Vektor tranformiert werden kann, ohne Dabei zuviel Metainformationen über den Code zu verlieren. Zu diesen Informationen gehören unter anderem die Namen der verwendeten Methoden und Variablen oder die Anzahl der genutzten Operatoren. Natürlich muss der Vektor auch den syntaktischen Aufbau abbilden können.

Um dieses Problem anzugehen, wird in dieser Thesis die grundlegende Idee und Heransgehensweise der Arbeit von Li et al. [35] aufgegriffen und weitergeführt. In dieser Ausarbeitung wird das Tool `CCLearner` entwickelt. Dabei handelt sich um ein Programm für Code-Clone-Datection, welches ein neuronales Netz als Grundlage nutzt. Hier besteht die Eingabe für das Modell aus einem Vektor, welcher die Häufigkeit bestimmter *Token* zählt.

Token Terminalsymbol der Grammatik einer Programmiersprache [34]. Folglich die kleinste Einheit, die vom Compiler verstanden wird [30]. Ein Token kann auch mit einem Wert beschrieben sein.

Listing 4.1: Beispiel: Token-Stream eines Statements

```
1 double d = 3.5;
```

Listing 4.1 besteht aus fünf Token: `double`, `d`, `=`, `3.5` und `;`.

Li et al. differenzieren die Token in acht verschiedene Kategorien: Keywords, Operatoren, Marker, Literale, Namen von Typen, Namen von Methoden, qualifizierte Namen und Namen von Variablen. Aus diesen Kategorien wird dann ein achtstelliger Vektor generiert, indem die Vorkommnisse der eindeutigen Token gezählt und der jeweiligen Kategorie zugeordnet werden. Tabelle 4.1 zeigt dies an einem Beispiel. Dabei wird die Frequenz eines Tokens durch `<key,value>` dargestellt, wobei `key` ein Token ist und `value` die aufgetretene Häufigkeit.

Index	Category name	An exemplar token-frequency list
C1	Reserved words	<if,2>, <new,3>, <try,2>...
C2	Operators	<+=,2>, <!=,3>...
C3	Markers	<;,2>, <[,2>, <],2>...
C4	Literals	<1.3,2> , <false,3>, <null,5>...
C5	Type identifiers	<byte,2> , <URLConnection,1>...
C6	Method identifiers	<read,2> , <openConnection,1>...
C7	Qualified names	<System.out,6>, <arr.length,1>...
C8	Variable identifiers	<conn,2> , <numRead,4>...

Tabelle 4.1: Beispiel: Kategorisierung der Token [35]

Die Erstellung dieses Vektors durch Li et al. wird in den folgenden Kapiteln näher beschrieben. Hierzu wurde ein Tokenizer und der abstrakte Syntaxbaum (engl. Abstract Syntax Tree oder AST) des umzuwandelnen Software-Codes verwendet.

4.3.2

Tokenizer

Wenn Quellcode in eine Sequenz von Token umgeformt wird, so wird dieser Vorgang als **Tokenization** bezeichnet. Entsprechend werden Frameworks, welche diesen Prozess als Dienst anbieten, als **Tokenizer** definiert. Lit et al. verwenden *ANTLR lexer* [4], um aus dem übergebenen Quellcode einen Tokenstream zu generieren. ANTLR ist ein weit verbreiteter *Parser Generator*, welcher von vielen Firmen (unter anderem Twitter Inc, Hadoop) genutzt wird [42]. Es soll ein Beispiel in Listing 4.2 zur Veranschaulichung dienen.

Listing 4.2: Beispiel: Eingabe für Tokenizer

```
1  public class Person {
2  private int age;
3
4  public Person(int age) {
5    this.age = age;
6  }
7
8  public int getAge() {
9    return this.age;
10 }
11 }
```

Die daraus entstehende Tokensequenz besitzt 35 einzelne Token. Lässt man sich die Token ausgeben, so sieht dies folgender Maßen aus (Listing 4.3). Dabei ist jeder Token wie folgt aufgebaut:

[*Start-Index*:*End-Index*=*Text*,<*Typ-ID*>,*Linien-Nummer*:*Start-in-Linie-Index*]

Listing 4.3: Beispiel: Einzelne Token als String

```
1  [0:5='public',<45>,1:0], [7:11='class',<19>,1:7],
2  [13:18='Person',<115>,1:13], [20:20='{',<70>,1:20],
3  [24:30='private',<43>,2:1], [32:34='int',<37>,2:9],
4  [36:38='age',<115>,2:13], [39:39=';',<74>,2:16],
5  [46:51='public',<45>,4:1], [53:58='Person',<115>,4:8],
6  [59:59='(',<68>,4:14], [60:62='int',<37>,4:15],
7  [64:66='age',<115>,4:19], [67:67=')',<69>,4:22],
8  [69:69='{',<70>,4:24], [74:77='this',<53>,5:2],
9  [78:78='.',<76>,5:6], [79:81='age',<115>,5:7],
10 [83:83='=',<80>,5:11], [85:87='age',<115>,5:13],
11 [88:88=';',<74>,5:16], [92:92='}',<71>,6:1],
12 [99:104='public',<45>,8:1], [106:108='int',<37>,8:8],
13 [110:115='getAge',<115>,8:12], [116:116='(',<68>,8:18],
14 [117:117=')',<69>,8:19], [119:119='{',<70>,8:21],
15 [124:129='return',<46>,9:2], [131:134='this',<53>,9:9],
16 [135:135='.',<76>,9:13], [136:138='age',<115>,9:14],
17 [139:139=';',<74>,9:17], [143:143='}',<71>,10:1],
18 [146:146='}',<71>,11:0]
```

4.3.3

Abstract Syntax Tree

Des Weiteren nutzen Li et al. für die Erstellung des Eingabevektors einen **AST**. Wie bereits erwähnt, sind die kleinsten vom Compiler verstanden Einheiten die Token. Wie diese Token in einer Programmiersprache angeordet werden können, wird durch eine *Grammatik* definiert. Folglich kann man sagen, dass diese Grammatik die Syntax der Sprache bestimmt. Die einzelnen Token werden anhand der Grammatik in deklarierten Ausdrücken verschachtelt. Diese Ausdrücke beschreiben eine valide Syntax und sind immer in einer Baumstruktur angeordnet. Daher werden sie auch **Syntaxbaum** genannt. Wenn das Statement aus Listing 4.1 in einen solchen Syntaxbaum umgewandelt wird, erhält man die Struktur aus Abbildung 4.2.

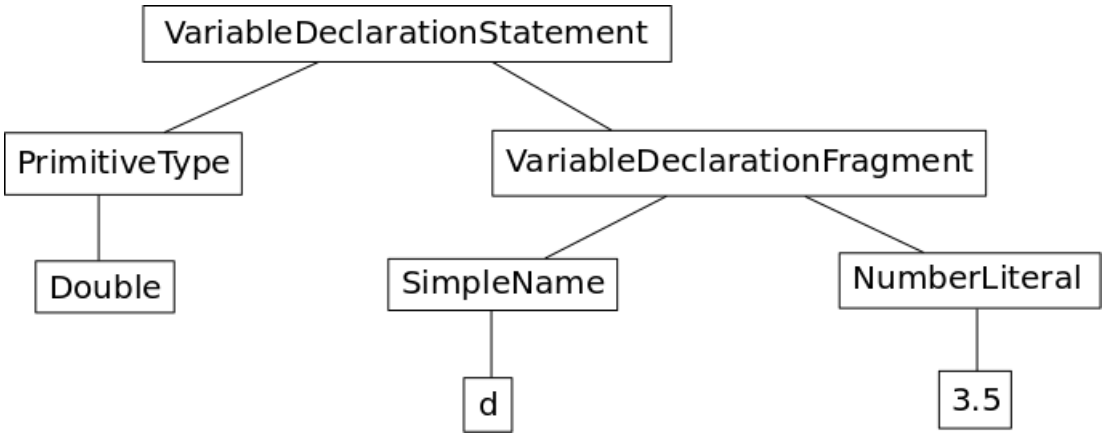


Abbildung 4.2: Beispiel: AST des Listings 4.1

4.3.4

Erweiterung des Eingabevektors

Mit Hilfe der *Token* und des *ASTs* kann der von Li et al. beschriebene achtstellige Vektor durch die Umformung des Codes erzeugt werden. Wie die Transformation genau vollzogen wird, ist in Kapitel 5.1 näher erläutert.

Wenn man sich die Bedeutung der einzelnen Stellen des Vektors vor Augen führt, so erkennt man, dass lediglich die Häufigkeit der Token der acht verschiedenen Token-Kategorien erfasst wird. Wenn man nur die Anzahl der vorkommenen Token mit einander vergleicht, so kann es durchaus geschehen, dass sich zwei Vektoren sehr ähnlich sind, obwohl die beiden umgeformten Code-Fragmente wenig

bis keine Ähnlichkeit aufweisen. Um diesem Problem entgegenzutreten, wird der Vektor nach Li et al. um einige Stellen erweitert.

Die ersten acht Stellen des Vektors werden übernommen. Mit ihnen werden die Metainformationen über die verwendeten Token der einzelnen Kategorien aus dem Code-Fragment gefiltert. Ein entscheidener Teil der Metainformationen fehlt allerdings noch. Und das ist der grundlegende Aufbau des Code-Fragments. Der Eingabevektor für die neuronalen Netzwerke muss ebenfalls die Informationen über den Aufbau des Quellcodes abbilden und nicht nur die Häufigkeit der verwendeten Token.

Für die Abbildung dieser Metainformationen sollen die Namen der in dem Code-Fragment verwendeten Typen und Methoden in ihrer im Code vorkommenden Reihenfolge dem Vektor hinzugefügt werden. Hier stößt man wieder auf das Problem, dass der Vektor nur eine feste Anzahl an Einträgen haben kann. Die Speicherung aller Namen würde den Vektor allerdings mit einer variablen Länge definieren. Umso länger ein Code-Fragment ist, desto mehr Typen und Methoden werden verwendet und desto länger würde der Vektor werden. Folglich muss eine maximale Länge des Vektors festgesetzt werden. Diese Anzahl an Einträgen kann allerdings nicht frei gewählt werden. Wählt man sie zu klein, können längere Code-Abschnitte nicht genügend durch den Vektor abgebildet werden, da wieder viele der Namen nicht in den Vektor aufgenommen werden können. Wählt man sie wiederum zu groß, kann es sein, dass der Vektor niemals ganz gefüllt wird, sodass unnötiger Speicher allokiert wird.

Um die Länge des Vektors bestimmen zu können, wird der *Security Code Exporter für Github* [39] genutzt. Es sollen Beispiele von Code-Fragmenten von bekannten Schwachstellen dahingehend untersucht werden, wie lang die Ausschnitte sind und wie viele Namen von Typen und Methoden verwendet werden. Damit Beispiele von bekannten Sicherheitslücken untersucht werden können, müssen sie zunächst aus dem Internet geladen werden. Dies wird durch das Tool von Christian Müller umgesetzt, welches im Rahmen einer Bachelor-Thesis [39] entwickelt wurde. Der Author beschreibt die Funktionalität des Programms wie folgt:

„Der Security Code Exporter sucht automatisiert auf Github, einem der größten Hosts für Softwareprojekte, nach security relevantem Code und lädt diesen herunter. [...] Mit Hilfe des Security Code Exporters können benutzerdefinierte Schwachstellendatenbanken erstellt werden [...]“[39]

In mit diesem Programm erstellten Repository werden einige Code-Ausschnitte

als Beispiel gesucht. Dabei wird explizit auf die Länge des Codes geachtet und darauf wieviele Typen und Methoden verwendet werden. Die Ergebnisse der untersuchten Code-Fragmente zeigen, dass die längsten Abschnitte etwa 50 Namen verwenden. Daher wird der Eingabevektor um 50 Stellen erweitert, sodass insgesamt 58 Stellen vorhanden sind. Dieser 58-stellige Vektor kann nun die nötigen Metainformationen eines Code-Fragments abbilden.

4.4 Frameworks

Für die Implementierung der modularen Kernalgorithmen werden zwei verschiedene Frameworks in Betracht gezogen: zum einen das Softwarepaket **Waikato Environment for Knowledge Analysis** (kurz **Weka**) und zum anderen **Deeplearning4j** (im weiteren Verlauf durch **DL4J** abgekürzt). Beide Frameworks sind Opensource und bieten Schnittstellen für die Entwicklung im Bereich des maschinellen Lernens an. Sie sollen in diesem Kapitel kurz näher beschrieben werden. Im Anschluss wird das letztendlich gewählte Framework dargelegt.

4.4.1 Weka

Weka ist das Produkt der Universität von Waikato in Neuseeland. In diesem Framework sind viele Algorithmen des maschinellen Lernens implementiert. Dabei beschränkt es sich nicht nur auf neuronale Netzwerke, sondern bildet auch noch andere Funktionen ab, welche für den Aufgabenbereich des Data Mining genutzt werden können.

> „`Weka is a collection of machine learning algorithms for data mining tasks. It contains tools for data preparation, classification, regression, clustering, association rules mining, and visualization.`“[61]

Neben den bereitgestellten Schnittstellen, kann Weka auch über eine Anwendungsoberfläche verwendet werden. Diese Oberfläche vereinfacht die Arbeit mit den Schnittstellen, sodass auch Personen ohne großes Wissen im Bereich der Programmierung das Framework einfach nutzen können. Da sich diese Master-

Thesis jedoch ausschließlich mit neuronalen Netzwerken beschäftigt, sollen nur die relevanten Punkte des Frameworks näher beleuchtet werden. Daher wird nun dargelegt, wie man einen Eingabevektor definiert und wie man die Test- bzw. Trainingsdaten deklariert.

In Weka werden Datensätze durch das *Attribute-Relation File-Format* (kurz *arff*) definiert [5]. Dabei handelt es sich um ASCII-Textdateien, welche eine Liste von Instanzen beschreibt, welche sich eine Menge von Attributen teilen. Diese arff-Datei ist in zwei Bereiche unterteilt. Der erste beschreibt die Metainformationen der im zweiten Bereich folgenden Daten. Ein Beispiel soll den Aufbau dieser Dateien näher beleuchten.

Folglich soll ein Beispiel aus dem Paper *An Approach For Iris Plant Classification Using Neural Network* [57] beschrieben werden. Hierbei werden Pflanzen der Gattung *Iris* anhand ihrer Kelch- und Blütenblätter hinsichtlich ihrer genauen Art klassifiziert. Möchte man dieses Beispiel mit Weka umsetzen, so sieht der Header der arff-Datei wie in Listing 4.4 gezeigt aus.

Listing 4.4: Beispiel: arff-Header für Iris-Beispiel[5]

```
1 @RELATION iris
2
3 @ATTRIBUTE sepallength  NUMERIC
4 @ATTRIBUTE sepalwidth   NUMERIC
5 @ATTRIBUTE petallength  NUMERIC
6 @ATTRIBUTE petalwidth   NUMERIC
7 @ATTRIBUTE class {Iris-setosa,Iris-versicolor,Iris-virginica}
```

Zunächst wird der Name der Relation definiert. Mit `@Attribute` werden die Eingaben des neuronalen Netzes festgelegt. Es wird folglich der Eingabevektor definiert. In dem Beispiel hat jeder Vektor fünf Stellen. Dabei sind die ersten vier Stellen die Eigenschaften der Kelch- und der Blütenblätter. Da die Länge und Breite der jeweiligen Blätter mit numerischen Werten angegeben werden, wurden die Eingabe als `NUMERIC` gekennzeichnet. Das letzte Attribut gibt die drei möglichen Klassen für die Klassifizierung an. Diese sind in dem Beispiel die Klassen `Iris-setosa`, `Iris-versicolor` und `Iris-virgincia`.

Listing 4.5 zeigt den zweiten Teil der Beispieldatei. Hierbei wurde das Beispiel gekürzt, da es im Original zu viele Datensätze beinhaltet. Es soll nur der grundlegende Aufbau der Datei veranschaulicht werden.

Listing 4.5: Beispiel: arff-Daten für Iris-Beispiel[5]

```
1 @DATA
2 5.1,3.5,1.4,0.2,Iris-setosa
3 4.9,3.0,1.4,0.2,Iris-setosa
4 4.7,3.2,1.3,0.2,Iris-setosa
5 4.6,3.1,1.5,0.2,Iris-setosa
6 5.0,3.6,1.4,0.2,Iris-setosa
```

Jede einzelne Zeile in dem Daten-Bereich beschreibt einen Datensatz. Tabelle 4.2 zeigt die Datensätze. Dabei wird Kelchblatt (engl. sepal) durch *KB* und Blütenblatt (engl. petal) durch *BB* abgekürzt. Neuronale Netzwerke werden in Weka

Länge KB	Breite KB	Länge BB	Breite BB	Klasse
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5.0	3.6	1.4	0.2	Iris-setosa

Tabelle 4.2: Werte zum Messen der Performanz

durch die Klasse *MultilayerPerceptron* (kurz *MLP*) definiert. Die einzelnen Parameter des Modells können während der Trainingsphasen gespeichert und geändert werden [40]. Das MLP ist ein Classifier, welches Backpropagation zum Lernen nutzt. Die Dokumentation beschreibt das MLP wie folgt:

„A classifier that uses backpropagation to learn a multi-layer perceptron to classify instances.“[40]

4.4.2

Deeplearning4j

Deeplearning4j (kurz **DL4J**) [21] ist ebenfalls ein Opensource Projekt und wurde von der Firma *Skymind* [20] ins Leben gerufen. Die Entwickler haben versucht, den oftmals schwierigen Einstieg in das Themengebiet des maschinellen Lernens zu vereinfachen und wichtige Schritte genau zu erklären.

„We’ve done our best to explain them, so that Eclipse Deeplearning4j can serve as a DIY tool for Java, Scala and Clojure programmers working on Hadoop and other file systems.“[2]

DL4J unterteilt die Prozesse des maschinellen Lernens in zwei grundlegende Schritte: zum einen das Laden und Aufbereiten der Trainingsdaten und zum anderen die Trainingsphase des Systems an sich [13].

Anders als Weka, nutzt DL4J kein eigenes Datenformat für die Definition der Daten und den Prozess des Einlesens. Die Daten können einfach in einer TXT- oder in einer CSV-Datei übergeben werden. Listing 4.6 zeigt den Inhalt der CSV-Datei, welcher den Datensätzen aus dem Iris-Klassifikation-Beispiel abbildet. Wie man erkennen kann, wurde hier die Klasse *Iris-setosa* mit einer ID abstrahiert, welche in diesem Fall den Wert *0* hat.

Listing 4.6: Beispiel: txt-Datei für Iris-Beispiel

```
1 5.1,3.5,1.4,0.2,0
2 4.9,3.0,1.4,0.2,0
3 4.7,3.2,1.3,0.2,0
4 4.6,3.1,1.5,0.2,0
5 5.0,3.6,1.4,0.2,0
```

Das Einlesen der Daten kann über Klassen gelöst werden, welche in dem Framework mit enthalten sind. Zusätzlich können auf den eingelesenen Daten noch Funktionen ausgeführt werden. Zu diesen Funktionen gehört unter anderem die Möglichkeit Rohdaten in numerische Vektoren umzuformen. Da das Preprocessing allerdings bereits einen solchen Vektor ausgibt, wird diese Funktion nicht benötigt. Es wird allerdings eine in dem Framework mitgelieferte *Normalisierung* genutzt. Dieser Dienst normalisiert die eingelesenen Daten. Dies ist wichtig, damit verschiedene Eingaben, welche sich in den Werten eventuell deutlich unterscheiden, die Gewichtung und Bedeutung der Eingaben nicht beeinflussen. Wenn beispielsweise ein Wert des Input-Vektors immer einen sehr hohen Wert hat und gleichzeitig ein Wert immer einen relativ kleinen Wert, so würde der hohe Wert mehr Einfluss auf das Ergebnis nehmen als der kleine Werte. Es sollen allerdings alle Stellen des Vektors gleich großen Einfluss auf die Ausgabe haben. Auch in dem Fall der zu entwickelnen Software wird dies der Fall sein. Der Grund dafür ist, dass die Hashwerte von Zeichenketten einen sehr großen Wert annehmen können. Darüber hinaus sind die ersten acht Stellen des definierten Vektors eher kleine Zahlen, da sie lediglich die Vorkommen von Elementen aus gewissen Kategorien darstellen. Um einer ungleich verteilten Bedeutung der Eingaben entgegenzuwirken, müssen die Stellen des Vektors normalisiert werden.

Das Framework unterstützt die Prozesse der Trainings- und Testphasen eines

neuronalen Netzwerks. Die zur Verfügung stehenden Daten können bereits während des Einlesens in Trainings- und Testdatensätze unterteilt werden. Dabei findet die Einteilung in die beiden Kategorien zufällig statt.

Die Definition eines neuronalen Netzwerks wird in DL4J in zwei Schritte unterteilt. Der erste Schritt befasst sich mit der Festlegung der Parameter und der zweite Schritt wird das Modell anhand der definierten Konfiguration erzeugt und initialisiert.

Nach der angestoßenen Trainingsphase bietet DL4J eine Evaluation des trainierten Netzwerks an. Dazu werden einfach die zuvor als Testdaten definierten Datensätze in das Netz gegeben. Die Ausgaben werden mit den gewünschten Zielwerten verglichen. Aus den Ergebnissen wird dann eine Evaluationsmatrix aufgestellt, welche die wichtigsten Metriken enthalten. Dazu gehören die bereits angerissenen Werte für die Accuracy, die Precision und den Recall.

4.4.3

Gewähltes Framework

Die Wahl des zu nutzenden Framework muss anhand der Einsatzmöglichkeiten abgewogen werden. Beide vorgestellten Softwarepakete sind Opensource und haben eine Schnittstelle für die Programmiersprache *Java*. Dies ist wichtig für die nicht-funktionale Anforderung **[NR1]**, welche aussagt, dass die zu entwickelnde Software in Java geschrieben werden soll.

Auch die Anforderung **[NR5]** kann mit beiden Frameworks realisiert werden. Sie sagt aus, dass die Kernalgorithmen ein neuronales Netzwerk als Grundlage haben soll. Probleme treten allerdings in Verbindung mit den Anforderungen **[NRE3]** und **[NRE4]** auf. Sie fordern, dass verschiedene neuronale Netze realisiert werden sollen und dass diese modular austauschbar sind. Das Framework Weka bietet zur Entwicklung die Klasse *MultilayerPerceptron* an. Die Standard-Implementierung dieser Klasse bildet allerdings unglücklicherweise nicht den Typ *Long Short-Term Memory Neural Network* von neuronalen Netzen ab. Es ist zwar möglichen, diesen Typen mit dem Framework abzubilden, allerdings erfordert dies eine manuelle Erweiterung der Funktionalitäten. Dies würde einen zusätzlichen Overhead für die Implementierung bedeuten.

Mit Deeplearning4j können alle Ziele für die Implementierung erreicht werden. Das Framework bietet sowohl LSTM- als auch DFF-Netzwerke an. Da mit der Verwendung von DL4J ein großer Implementierungsvorteil geboten wird, ist die-

ses das für die Umsetzung genutzte Framework für die Entwicklung der Kernalgorithmen.

4.5 Software Architektur

In diesem Abschnitt soll auf die grundlegende Architektur des Prototypen eingegangen werden. Dabei sollen verschiedene Aspekte betrachtet werden. Zunächst soll die Entscheidung getroffen werden, ob die Anwendung über die Konsole ausgeführt werden soll oder ob es eine grafische Benutzungsoberfläche geben soll. Danach wird auf die Struktur der Software eingegangen. Hierbei sollen vor allem die Schnittstellen zu dem verwendeten Framework dargelegt werden.

4.5.1 Benutzungsoberfläche

Die Schnittstelle zu dem Benutzer kann entweder durch eine Konsolenanwendung oder einer Anwendung mit grafischer Benutzungoberfläche[1] definiert werden. Die Idee des Tools ist es, dass das Suchen von Sicherheitsschwachstellen in Software-Repositories so einfach wie möglich realisiert wird. Damit dies gewährleistet werden kann, soll die Nutzung des Programm so unkompliziert wie möglich gestaltet werden. Hierfür soll die Anzahl der möglichen Interaktionen des Nutzers so gering wie möglich gehalten werden.

Aufgrund dessen wird für den Prototyp auf eine GUI verzichtet. Eine Konsolenanwendung bietet die perfekte Umgebung für eine unkomplizierte Nutzung durch den Anwender. Er wird anhand von unmissverständlichen Aufforderungen durch den Prozess der Schwachstellenanalyse geführt. Zusätzliche Ausgaben über den aktuellen Fortschritt der Arbeit geben dem Nutzer ein eindeutiges Feedback.

[1] Im weiteren Verlauf als *GUI* (engl. Graphical User Interface) bezeichnet.

4.5.2

Architektur

Folgend soll die Architektur des SecVuLearners beschrieben werden. Hierbei werden die wichtigsten Klassen des Programms dargestellt und wie diese miteinander Interagieren und in Beziehung stehen. Zusätzlich wird auf die Schnittstelle zu dem gewählten Framework näher eingegangen. Die Entscheidung über das Framework wurde in Kapitel 4.4 genauer erläutert. Es werden folglich alle relevanten Entscheidungen über die Organisation des Softwaresystems besprochen [55].

Aussagen über die Qualität einer Software werden oftmals aufgrund des Software-Designs getroffen. Dabei werden Metriken verwendet, welche die Umsetzung von zwei verschiedenen Konzepten messen: die der Kohäsion und die der Kopplung von Modulen und Paketen [43]. Bei der Verwendung von objektorientierten Programmiersprachen (wie Java oder C#) ist es wichtig, diese beiden Punkte im Hinterkopf zu behalten, um ein gutes Software-Design erreichen zu können [31].

Das Konzept der Kopplung beschreibt die Abhängigkeit[2] zwischen den verschiedenen Systemkomponenten, wohingegen mit der Kohäsion Aussagen über die semantische Zusammengehörigkeit von Modulen getroffen werden [55]. Die Kopplung soll so gering wie möglich gehalten werden, sodass die einzelnen Systemkomponenten unabhängig von einander sind. Die Kohäsion soll dagegen möglichst groß sein. Folglich sollen Funktionalitäten, welche inhaltlich zusammengehören, auch von einem Modul oder Paket abgebildet werden. Die Einhaltung dieser Konzepte vereinfachen die Wartung und erhöhen die Wiederverwendbarkeit.

Da in der nicht-funktionalen Anforderung **[NR4]** ein modularer Aufbau gefordert wird, wurde das Programm SecVuLearner in vier logisch getrennte Arbeitspakete geteilt: das Preprocessing, die neuronalen Netzwerke, die Schnittstelle zur Benutzungsoberfläche und die Einstellung für das Tool an sich. Abbildung 4.3 zeigt ein Paketdiagramm, in welchem die relevantesten Klassen mit dargestellt sind. Die einzelnen Klassen sind nicht mit ihren Feldern oder Operationen und Methoden dargestellt, da sonst die Übersichtlichkeit zu stark leiden würde. Hier soll es nur um den grundlegenden Aufbau gehen. Anschließend werden die einzelnen Pakete mit ihren Klassen kurz vorgestellt.

[2] Erzeugungs- und Aufrufabhängigkeit

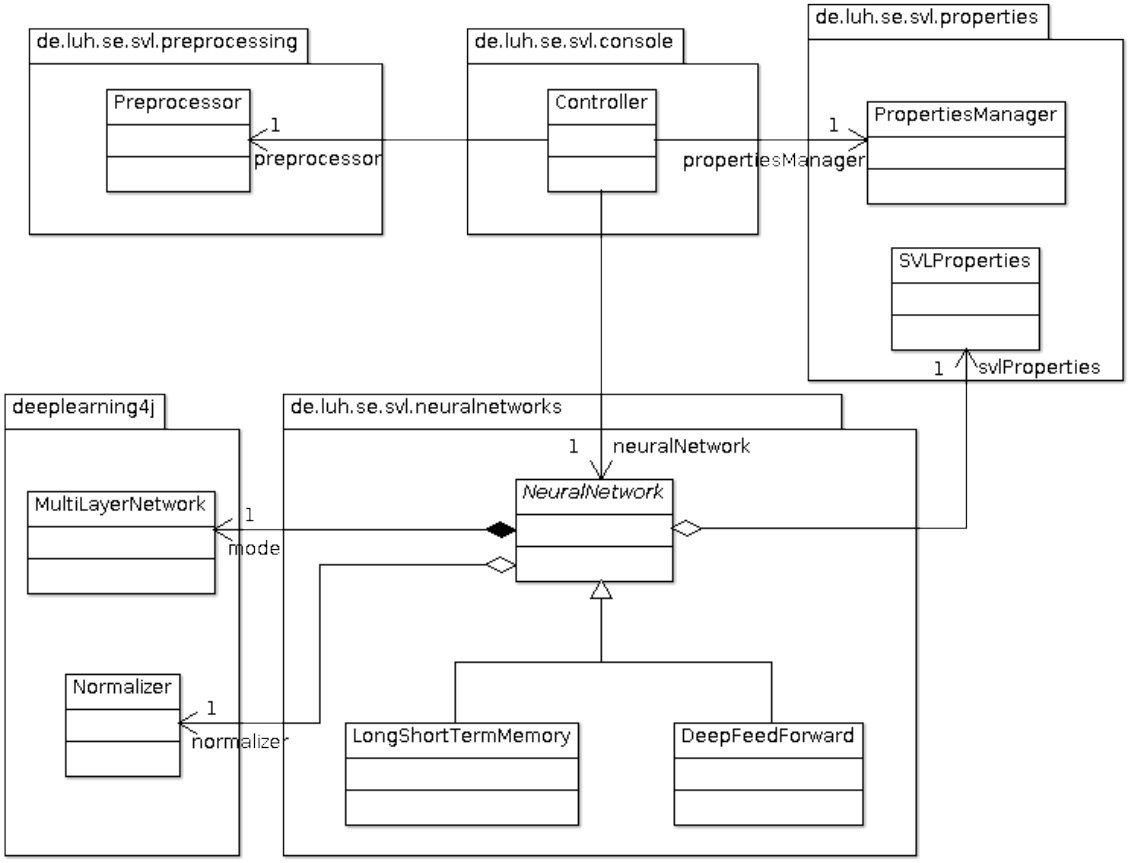


Abbildung 4.3: Paketdiagramm des SecVuLearners

Das Preprocessing wird durch eine Klasse abgebildet: dem `Preprocessor`. Es ist in dem Arbeitspaket `de.luh.se.svl.preprocessing` untergebracht. Dieses Modul ist für alle Funktionalitäten des bereits besprochenen Preprocessings verantwortlich. Die Klasse wird den auf Security Schwachstellen zu prüfenden Software-Code in den entsprechenden Eingabevektor transformieren. Er muss gewährleisten, dass auch ganze Software-Repositories transformiert werden können und nicht nur einzelne Dateien. Er erstellt also für jede zu prüfende Java-Datei einen Eingabevektor. Diese Vektoren werden von dem genutztn Framework durch CSV- oder TXT-Dateien eingelesen. Folglich müssen die Vektoren in eine solche Datei gespeichert werden. Da der Preprocessor die Aufbereitung der Rohdaten vollzieht, fällt auch dies in die Verantwortlichkeit dieses Moduls. Somit ist das Preprocessing so kohäsiv wie möglich gehalten. Die Kopplung ist so gering wie möglich gehalten. Wie in Abbildung 4.3 zu erkennen ist, gibt es lediglich eine Abhängigkeit, welche nicht zu vermeiden ist, da es sich um die Schnittstelle zur Benutzungsoberfläche handelt.

Eine Konsolenanwendung benötigt einen Bereitsteller aller ausführbaren Dienste des Softwaresystems. Dies wurde durch die Klasse `Controller` des Pakets `de.luh.se.svl.console` realisiert. Sie dient als Schnittstelle für alle Aufrufe von

der Benutzungsoberfläche. Dadurch kann auch zu einem späteren Zeitpunkt noch entschieden werden, ob die Oberfläche geändert werden soll, sodass auch eine grafische Komponente nachentwickelt werden kann. Dies ist ein Punkt, welcher die Erweiterbarkeit der Software deutlich steigert, welche in der nichtfunktionalen Anforderung **[NR6]** gefordert wurde. Die Konzepte der Kopplung und Kohäsion werden in diesem Fall vernachlässigt, da die Schnittstelle keine fachliche Komponente des Systems widerspiegelt. Sie wird für die Kopplung der nachfolgenden Pakete nicht mehr berücksichtigt.

Der SecVuLearner hat Einstellungen, welche geladen werden müssen. Dazu zählen Pfade der Verzeichnisse für die Trainingsdaten und für die Ausgaben. Des Weiteren wird die Konfiguration des neuronalen Netzwerks gespeichert. Diese Ansammlung von Daten wird durch die Klasse `PropertiesManager` in dem Arbeitspaket `de.luh.se.svl.properties` geladen. Die Klasse hat keine Abhängigkeiten zu anderen Paketen, sodass ihre Kopplung ein Minimum aufweist. Die Abbildung aller Einstellungen zur Laufzeit wird über die Klasse `SVLProperties` deklariert. Mit der Zugehörigkeit zu dem gleichen Arbeitspaket wird die Kohäsion gesteigert. Die Klasse ist allerdings nicht unabhängig zu anderen Modulen. Doch darauf wird bei der Beschreibung des nächsten Pakets näher eingegangen.

Das letzte Arbeitspaket befasst sich mit den Kernalgorithmen des System und wurde `de.luh.se.svl.neuralnetworks` genannt. Hier werden die nichtfunktionalen Anforderungen **[NR5]**, **[NRE3]** und **[NRE4]** abgebildet. Die besagen, dass der Erkennungsprozess durch ein neuronales Netzwerk abgebildet sein muss und dass verschiedene Typen genutzt werden können, welche modular austauschbar sein sollen. Hierfür wurde eine abstrakte Oberklasse `NeuralNetwork` definiert. Durch die Vererbung an die beiden speziellen Klassen `DeepFeedForward` und `LongShortTermMemory` sind die unterschiedlichen Typen von neuronalen Netzwerken abgebildet und durch die Erweiterung der Oberklasse modular austauschbar. Die abstrakte Klasse dient auch als Schnittstelle für das Framework Deeplearning4j. Sie nutzt zwei Klassen des Frameworks um die Modelle und den Normalisierer zu definieren. Die Kohäsion des Pakets ist aufgrund der inhaltlichen Zusammengehörigkeit möglichst groß gehalten. Ein neuronales Netzwerk hat eine Konfiguration, welche spezifische Werte für gewisse Eigenschaften festhalten. Diese Werte gehören zu den Einstellungen des SecVuLearners und werden wie bereits erwähnt in der Klasse `SVLProperties` gespeichert. Daher muss die Klasse `NeuralNetwork` eine solche Instanz als Feld haben, damit alle nötigen Werte bei der Initialisierung des Netzwerks zur Verfügung stehen. Darunter leidet die Kopplung, da das Paket nicht vollständig unabhängig ist. Dennoch ist die Beziehung notwendig und kann nicht aufgehoben werden.

KAPITEL 5

Implementierung

In diesem Kapitel sollen die wichtigsten Entscheidungen des Entwicklungsprozesses der Implementierung dargelegt werden. Dabei wird zunächst in Abschnitt 5.1 auf das Preprocessing eingegangen. Hier wird beschrieben, mit welchen Entwurfsmustern die Transformation von Quellcode in den in Kapitel 4.3.4 definierten Vektor umgesetzt wurde. In Kapitel 5.2 wird dann erläutert, wie die verschiedenen neuronalen Netzwerke mit dem gewählten Framework umgesetzt wurden.

5.1 *Preprocessing*

In diesem Kapitel soll die Implementierung des Preprocessings dargelegt werden. Wie in Abschnitt 4.3 bereits beschrieben wurde, verwendet der Preprocessing-Algorithmus einen Tokenstream und einen abstrakten Syntaxbaum für die Transformierung. Die Erstellung dieser beiden Elemente werden mit Hilfe von der externen Bibliothek *ANTLR lexer* [4] vollzogen. Die grundlegende Arbeitsweise wurde bereits in Kapitel 4.3.2 erklärt. Daher wird in diesem Kapitel nicht weiter auf die verwendete Bibliothek eingegangen, sondern es wird gezeigt, wie mit den erstellten Elementen die Transformierung von Software-Code zu dem bereits definierten Eingabevektor durchgeführt wird.

5.1.1

Chain Of Responsibility

Die vollständige Vorverarbeitung des Codes wird mit einer `Chain Of Responsibility` gelöst [25]. Dabei handelt es sich um ein Entwurfsmuster, welches zu der Kategorie der Verhaltensmuster gehört. Bei der Anwendung dieses Musters wird der Sender einer Anfrage von dem Empfänger entkoppelt. Im Falle des SecVuLearners ist der Sender die Klasse `Preprocessor`, welche eine Anfrage an die Klasse `FeatureExtractor` schickt. Die Anfrage entspricht hier dem Aufruf des Proprocessing-Algorithmus. Die möglichen Empfänger werden gleich weiter definiert. Abbildung 5.1 zeigt zunächst das Klassendiagramm des angewendeten Musters.

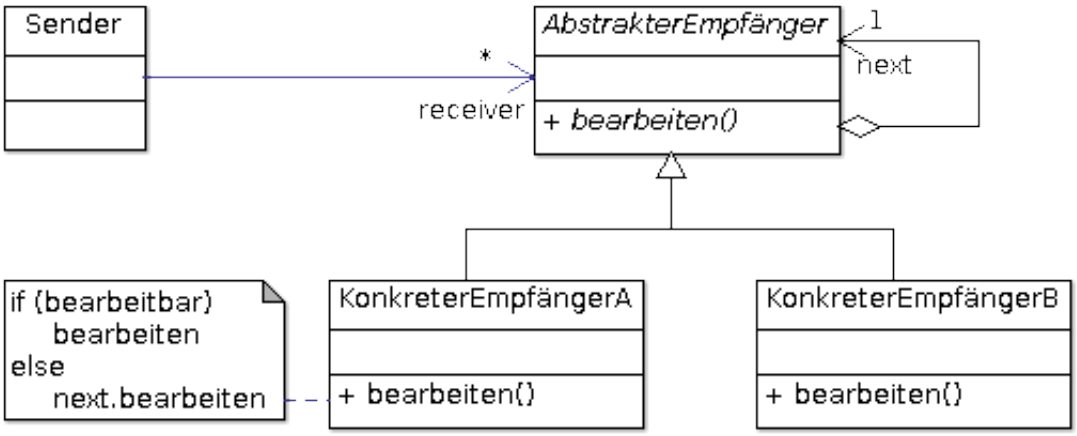


Abbildung 5.1: Klassendiagramm: Chain Of Responsibility

Für die Vorverarbeitung wird eine solche Zuständigkeitskette implementiert. Dabei wird allerdings nicht nur ein Empfänger die Anfrage bearbeiten, sondern alle Klassen, welche den abstrakten Empfänger implementieren. Die Klasse `FeatureExtractor` wird dabei den abstrakten Empfänger des Verhaltensmusters abbilden. Alle speziellen Unterklassen werden dann nacheinander durchlaufen und jede einzelne ist zuständig für die Erzeugung eines oder mehrerer Werte, welche den Stellen des 58-stelligen Vektors entsprechen.

Das Entwurfsmuster wurde an einer Stelle angepasst, um die Initialisierung zu vereinfachen. Die abstrakte Methode, welche in diesem Fall `extract` heißt, wurde in eine Schnittstelle `IFeatureExtractor` ausgelagert, welche von der Klasse `FeatureExtractor` erweitert wird. Zudem muss das Feld `next` nun nur noch die Schnittstelle implementieren und muss kein abstrakter Empfänger sein. Listing 5.1 zeigt die entsprechende abstrakte Klasse.

Listing 5.1: Implementierung: FeatureExtractor

```
1  public abstract class FeatureExtractor implements ↩
       IFeatureExtractor {
2    protected IFeatureExtractor next;
3
4    public FeatureExtractor() {
5      this.next = new ExtractionFinalizer();
6    }
7
8    public FeatureExtractor(FeatureExtractor next) {
9      this.next = next;
10   }
11 }
```

Wie in dem Listing 5.1 zu erkennen ist, bietet die abstrakte Klasse `FeatureExtractor` zwei verschiedene Konstruktoren an. Damit wird die Initialisierung der Zuständigkeitskette vereinfacht. Insgesamt sechs Klassen erben von dem `FeatureExtractor`: der `KeywordsOperatorsAndMarkersExtractor`, der `LiteralsExtractor`, der `MethodsExtractor`, der `QualifiedNamesExtractor`, der `TypesExtractor` und der `VariableNamesExtractor`. Die Klassennamen verdeutlichen für welche Stellen des zu erzeugenen Eingabevektors sie verantwortlich sind. Folglich werden mit diesen Klassen die ersten acht Stellen des Vektors erstellt. Die restlichen 50 Stellen werden von dem `ExtractionFinalizer` transformiert. Dieser implementiert lediglich die Schnittstelle, sodass diese Klasse keinen Nachfolger besitzt. Im Anhang A findet man die Abbildung A.1, welche das entsprechende Klassendiagramm zeigt. Aufgrund der Übersichtlichkeit wurden die Parameter und Rückgabewerte der `extract`-Operation nicht mit aufgegriffen. Listing 5.2 zeigt die Initialisierung der Kette und wie die Transformation angestoßen wird.

Der `VariableNamesExtractor` nutzt den parameterlosen Konstruktor, sodass sein nachfolgender Empfänger automatisch der `ExtractionFinalizer` ist. Die leere übergebene Liste entspricht den bisher gefundenen Stellen des Vektors. Die Ergebnisse werden als Liste und nicht als Vektor übergeben, da das Konkatenieren von neuen Ergebnissen mit einer Liste simpler ist, da ein Vektor immer eine festgelegte Anzahl von Stellen haben muss. Eine der leeren übergebenen Mengen wird während des durchlaufenden Prozesses mit den Zeichenketten der Token gefüllt, damit bekannt ist, welche Token bereits verarbeitet wurden. Die anderen beiden Mengen werden von dem `ExtractionFinalizer` benötigt. Er muss die

Namen von den Methoden und den Typen für die letzten 50 Stellen des Vektors verarbeiten.

Listing 5.2: Initialisierung: FeatureExtractor

```
1  FeatureExtractor fe = new ↩
       KeywordsOperatorsAndMarkersExtractor(
2    new LiteralsExtractor(
3      new TypesExtractor(
4        new MethodsExtractor(
5          new QualifiedNamesExtractor(
6            new VariableNamesExtractor())))));
7
8  List<String> stringList = fe.extract(new ArrayList<String>(),
9    new HashSet<String>(),
10   new HashSet<String>(),
11   new HashSet<String>(),
12   ast,
13   tokens);
```

5.1.2

Visitor

Wie bereits in Kapitel 4.3.3 kurz beschrieben, wird für die Vektorisierung des Software-Codes ein abstrakter Syntaxbaum verwendet. Mit dessen Hilfe werden die Stellenwerte der verwendeten Literale, Typen, Methoden, qualifizierten Namen und Variablennamen berechnet. Für die Extraktion dieser Werte wird ein *Besuchermuster* [25] auf dem übergebenen Syntaxbaum angewendet. Dieses zählt ebenfalls zu den Verhaltensmustern. Es wird verwendet, wenn ein auszuführender Algorithmus `a` von dem Typen einer Klasse `K` abhängig ist und diesbezüglich variiert. Sämtliche Variationen dieses Algorithmus werden in eine sogenannte Besucherklasse `B` ausgelagert. Die Klasse `K`, dessen Typ bestimmt welche Variation der Methode ausgeführt werden soll, wird als ein Parameter dieser Operation definiert. Nun muss nur noch die Klasse `K` eine Methode `akzeptieren` besitzen, welche eine Besucherklasse als Parameter hat. Auf dem übergebenen Parameter wird folglich der Algorithmus `a` aufgerufen und die Instanz der Klasse `K` übergibt sich selbst als Parameter. Die Typsicherheit gewährleistet, dass die richtige Variation des Algorithmus aufgrufen wurde. Abbiildung 5.2 zeigt das allgemeine

Klassendiagramm des Verhaltensmusters.

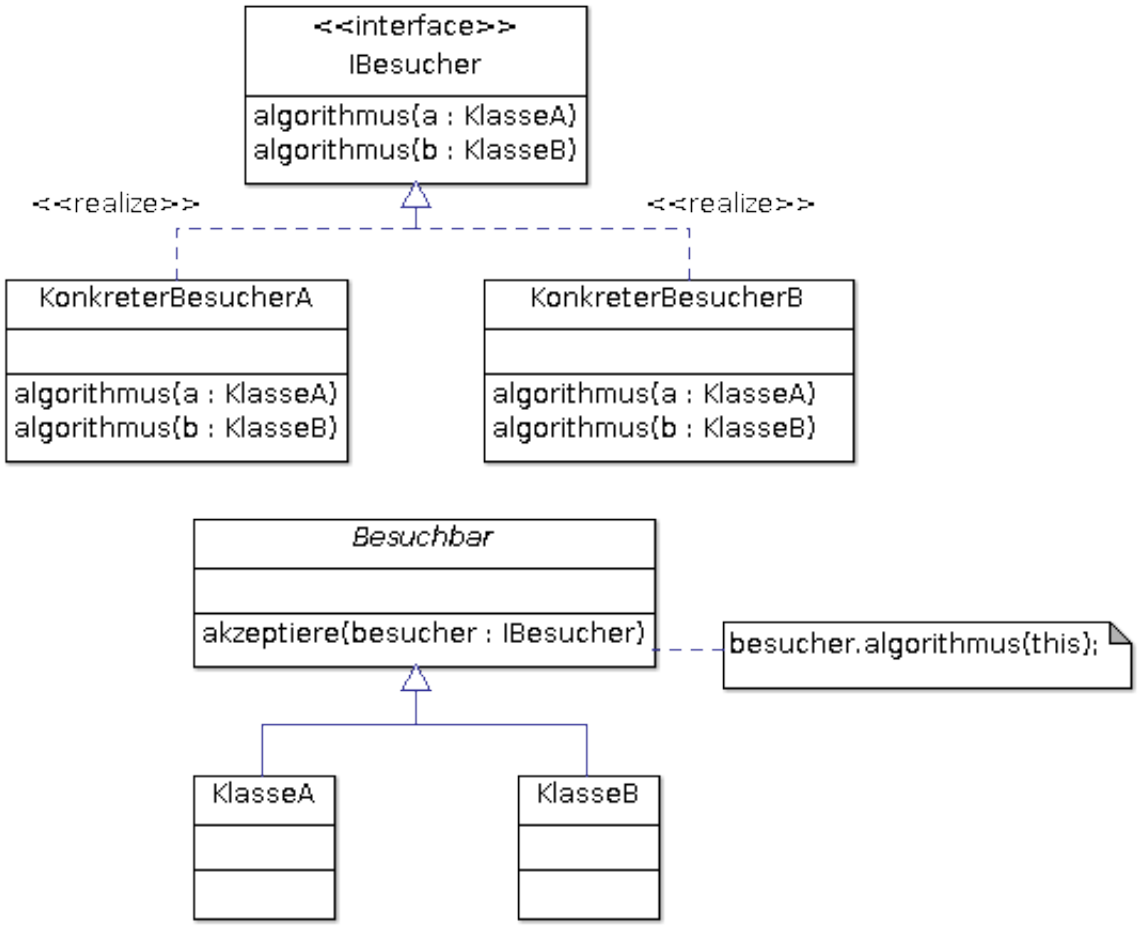


Abbildung 5.2: Klassendiagramm: Besuchermuster

Die nötigen Klassen für die Anwendung dieses Musters werden von dem Framework *ANTLR lexer* [4] bereitgestellt. Da die Implementierung immer sehr ähnlich sind und es zu viel Platz kosten würde, alle aufzuführen, wird lediglich die Umsetzung des Besuchermusters an einem Beispiel gezeigt. Listing 5.3 zeigt die Implementierung für die Extraktion der in dem Code verwendeten Methoden. Der abstrakte Syntaxbaum besteht aus Klassen, welche alle besuchbar sind. Somit definieren die verschiedenen Knoten des Baums, welche Algorithmen ausgeführt werden sollen. Im Falle der Methoden werden die Knoten besucht, welche die Deklaration und den Aufruf von Methoden beschreiben.

Wie in Listing 5.3 ebenfalls zu sehen ist, wird für den Besucher eine anonyme innere Klasse verwendet. Diese Art von Klasse hat keinen Namen und erzeugt immer automatisch ein Objekt [59]. Die Deklaration der Klasse und die Erzeugung des Objekts ist zu einem Sprachkonstrukt verbunden [59]. Der Zähler der Methoden muss in einem Objekt gekapselt werden, da innere Methoden nur auf primitive Datentypen der äußeren Klasse zugreifen können, welche als `final`

markiert sind. Dadurch wird allerdings ein Schreibzugriff untersagt.

Listing 5.3: Anwendung des Besuchermusters

```
1  IntegerContainer methodsCounter = new IntegerContainer(0);
2
3  ast.accept(new ASTVisitor() {
4    public boolean visit(MethodDeclaration methDecl) {
5      methodsCounter.increment();
6      String key = methDecl.getName().getFullyQualifiedName();
7      keys.add(key);
8      methodKeys.add(key);
9      return true;
10   }
11
12   public boolean visit(MethodInvocation methInv) {
13     methodsCounter.increment();
14     String key = methInv.getName().getFullyQualifiedName();
15     keys.add(key);
16     methodKeys.add(key);
17     return true;
18   }
19 });
20 result.add(Integer.toString(methodsCounter.getValue()));
21 return this.next.extract(result, keys, typeKeys,
22   methodKeys, ast, tokens);
```

5.2 Neuronale Netze

Zunächst ist herauszustellen, dass gemäß Anforderung **[NRE3]** verschiedene Typen von neuronalen Netzen für die Evaluation implementiert werden müssen, allerdings wird im Prototyp lediglich eins der beiden Netze verwendet werden. Hierbei wird natürlich das performantere und effizientere Modell gewählt. Welches das sein, kann allerdings erst nach der abgeschlossenen Evaluation gesagt werden.

In dem Kapitel 4.5.2 wurde die Architektur der neuronalen Netzwerke beschrie-

ben. Sowohl das `DeepFeedForward` als auch das `LongShortTermMemory` erweitern die Oberklasse `NeuralNetwork`. Diese ist in Listing 5.4 gezeigt.

Listing 5.4: Implementierung NeuralNetwork

```
1  public abstract class NeuralNetwork {
2    protected MultiLayerNetwork model;
3    protected NormalizerStandardize normalizer;
4    protected SVLProperties svlProperties;
5
6    public NeuralNetwork(SVLProperties svlProperties) {
7      this.svlProperties = svlProperties;
8      initialize();
9    }
10
11   protected abstract void initialize();
12   public abstract void startTraining() throws
13     IOException, InterruptedException;
14   public abstract List<IdFileTuple> ↩
       testCodeForVulnerabilities(List<String> namesOfFiles)
15     throws Exception;
16 }
```

Da sich die Methoden `startTraining` und `testCodeForVulnerabilities` für beide Arten von neuronalen Netzen stark ähneln und nicht sonderlich komplex sind, sollen diese nur kurz beschrieben werden. Die nächsten Kapitel sollen sich vor allem mit der spezifischen Initialisierung des jeweiligen Modells beschäftigen.

Die Methode `startTraining` ist bei beiden Netzwerken gleich aufgebaut. Zunächst werden die Trainingsdaten, welche in einer CSV-Datei abgespeichert wurden, mit Hilfe von Klassen des Frameworks *Deeplearning4j* [21] geladen. Die geladenen Daten werden normalisiert und dann für die Trainingsphase genutzt. Dazu wird die dafür vorgesehende Methode `fit` der Klasse `MultiLayerNetwork`, welche bereits in dem Paketdiagramm 4.3 zu sehen war, genutzt.

Die Methode `testCodeForVulnerabilities` verhält sich ähnlich. Der einzige Unterschied ist jedoch, dass hier noch keine CSV-Datei vorliegt. Folglich werden die zu überprüfenden Dateien entsprechend des Preprocessing-Algorithmus vorverarbeitet und zwischengespeichert. Dann werden die Vektoren geladen und dem Modell als Eingabe übergeben. Dazu bietet die Klasse `MultiLayerNetwork` die Funktion `output` an. Auch hier unterscheidet sich die Semantik zwischen den

Netzwerken nur marginal.

5.2.1

Deep Feed Forward

Die Initialisierung eines `DeepFeedForward`-Netzwerks beginnt mit der Konfiguration. Diese wird definiert und dem Modell zum Laden übergeben. Die Konfiguration wird in Deeplearning4j mit der Klasse `MultiLayerConfiguration` abgebildet. Bei der Erzeugung mit Hilfe eines `Builders` werden alle wichtigen Parameter festgesetzt. In Listing B.1 im Anhang B.1 wird eine beispielhafte Konfiguration erstellt und dem Modell zum Initialisieren übergeben. Es wird ein Netzwerk mit einem Hidden-Layer sein.

5.2.2

Long Short-Term Memory Neural Network

Ein Netzwerk von Typ Long Short-Term Memory wird ähnlich zu einem Deep-Feed-Forward-Modell initialisiert. Der wichtige Unterschied liegt hier darin, dass eine andere Art von Schicht genutzt wird. Um einen andere Art der Definition einer Netzwerkkonfiguration zu zeigen, wurde der Ablauf hier ein wenig geändert. Es soll dargelegt werden, wie ein Modell mit einer variablen Anzahl von Schichten definiert werden kann. Dazu wird die in Listing B.1 fortlaufende Punktnotation unterbrochen, um eine `for`-Schleife zu integrieren. Listing B.2 im Anhang B.2 zeigt die Deklaration der entsprechenden Konfiguration und die anschließende Initialisierung des Netzwerks.

KAPITEL 6

Evaluation

Für das automatisierte Erkennen von Sicherheitsschwachstellen in Software-Repositories wurde das Tool SecVuLearner prototypisch umgesetzt. Dabei wurde entsprechend der Anforderungen ein neuronales Netzwerk für den Prozess genutzt. Im Rahmen dieser Masterthesis soll allerdings nicht nur ein Prototyp entwickelt werden, sondern es sollen ebenfalls verschiedene Arten von Netzwerken mit einander verglichen werden. Der Vergleich soll die Tauglichkeit der Typen für den Bereich des Code-Clone-Detections zeigen. Wie in den vorigen Kapiteln bereits geschrieben, wurden die Typen *Deep Feed Forward* und *Long Short-Term Memory* für die Evaluation ausgewählt. Nun gilt es die ausstehenden Anforderungen an die Evaluation umzusetzen. Dabei sollen die Daten der Konfigurationen und die Ergebnisdaten der Performanz vor und nach den Trainingsphasen gespeichert werden, damit eine Vergleichsgrundlage geschaffen wird. Ebenso muss für den Prototypen entschieden werden, welches neuronale Netzwerk in der Software verwendet werden soll.

Zunächst muss allerdings definiert werden auf welcher Datengrundlage eine solche Evaluation stattfinden soll. Diese soll in Abschnitt 6.1 definiert und beschrieben werden. Anschließend soll in Kapitel 6.2 dargelegt werden, in welcher Form zwei neuronale Netze und die spezifischen Ergebnisse miteinander verglichen werden können. Folgend werden in Abschnitt 6.4 und 6.5 die Ergebnisse der Evaluation der einzelnen Netzwerkarten gezeigt. In dem letzten beiden Kapiteln 6.6 und 6.7 sollen die Ergebnisse dann miteinander verglichen werden, sodass ein Fazit gezogen werden kann.

6.1 Datengrundlage

Um ein neuronales Netzwerk für den Gebrauch des Code-Clone-Detection nutzen zu können, muss das Modell eine Datengrundlage bekommen, mit der das Netz die Trainingsphase durchschreiten kann. Dabei werden bei jeder Iteration durch die Daten die Gewichte des Modells angepasst und verbessert. Die Trainingsdaten müssen also Java-Dateien sein von denen bekannt ist, dass sie eine Sicherheitslücke implementiert haben und es muss auch definiert sein, welche Schwachstelle das ist. Für die ausstehende Evaluation sollen nun diese Dateien agglomeriert werden.

In dem Fachgebiet Software-Engineering der Leibniz Universität Hannover wurde bereits eine Masterthesis geschrieben, welche in das Themengebiet des Code-Clone-Detection fällt. Sie trägt den Titel *Security Code Clone Detection entwickelt als Eclipse Plugin* [9] und wurde von Wasja Brunotte verfasst. In dieser Ausarbeitung wird ein Plugin für die Entwicklungsumgebung *Eclipse* implementiert, welches Software-Code zur Entwicklungszeit auf Sicherheitsschwachstellen prüft. Anders als bei dem SecVuLearner werden allerdings keine neuronalen Netzwerke für den Erkennungsprozess verwendet, sondern es wird ein statischer Ansatz verfolgt. Da beide Softwaresysteme mit den gleichen Aufgaben und Datensätzen konfrontiert werden, bietet es sich an, die Daten für den Trainingsprozess und der Evaluation mit den von Herrn Brunotte verwendeten Daten abzugleichen und anzupassen. Somit kann in dieser Arbeit nicht nur die Eignung der verschiedenen Netzwerktypen verglichen werden, sondern es kann zusätzlich evaluiert werden, wie die neuronalen Netzwerke im Vergleich zu einem traditionellen und statischen Ansatz abschneiden. Folglich wird kurz erläutert, welche Daten der Author für die Evaluation gewählt hat und nach welchen Gesichtspunkten er diese gefiltert hat. Wie der Erkennungsprozesses des Plugins genau funktioniert wird nicht weiter beleuchtet, da dies zu umfänglich wäre.

Alle Schwachstellen wurden mit dem *Security Code Exporter für Github* [39] aus dem Internet geladen. Dabei werden viele verschiedene Arten von Sicherheitslücken gefunden. Aus den agglomerierten Daten wurden dann 20 verschiedene Security-Schwachstellen für die Evaluation ausgewählt. Diese sind in Tabelle 6.1 mit ihrer CVE-ID, dem vergebenen Scoring und dem betreffenden Produkt aufgeführt. Die ausgewählten Daten mussten einige Kriterien erfüllen. Der Author schreibt:

„Nicht geeignet haben sich Schwachstellen, die lediglich eine andere Bibliothek importieren. Ebenfalls für ungeeignet im Rahmen dieser Evaluation wurden Schwachstellen befunden, die Abhängigkeiten dynamisch laden bzw. dies mit Hilfe von Versionsangaben in Literalen tun. Des Weiteren fanden Schwachstellen, die wesentliche Code-Änderungen in einer Vielzahl von Methoden enthielten, keine Berücksichtigung.“[9]

CVE-ID	Scoring	Produkt
CVE-2006-2806	7.8	Apache Java Mail Enterprise Server
CVE-2007-5461	3.5	Apache Tomcat
CVE-2007-6203	4.3	Apache HTTP Server 2.0.x and 2.2.x
CVE-2008-2086	9.3	Sun JDK
CVE-2008-5515	5.0	Apache Tomcat
CVE-2009-2693	5.8	Apache Tomcat
CVE-2009-2901	4.3	Apache Tomcat
CVE-2010-4172	4.3	Apache Tomcat
CVE-2011-1475	5.0	Apache Tomcat
CVE-2011-3190	7.5	Apache Tomcat
CVE-2011-3377	4.3	Ubuntu Linux, Opensuse, Redhat Icedtea-Web
CVE-2012-1621	4.3	Apache Open For Business Project
CVE-2012-2459	5.0	Bitcoind und Bitcoin-Qt
CVE-2016-3897	4.3	Google Android
CVE-2016-6723	5.4	Google Android
CVE-2017-0389	7.8	Google Android
CVE-2017-0846	5.0	Google Android
CVE-2017-1217	4.3	IBM WebSphere Portal
CVE-2017-1591	4.3	IBM DataPower Gateway
CVE-2017-9096	6.8	iTextpdf iText

Tabelle 6.1: Für Evaluation ausgewählte Schwachstellen [9]

Da laut der Anforderung **[R5]** auch Klone der Typen 2 und 3 erkannt werden sollen, müssen die festgelegten Daten allerdings noch angepasst werden. Würde man lediglich die ausgewählten Java-Dateien für die Trainingsphase verwenden, so würden nur Klone des ersten Typen, also exakte Kopien des Software-Codes, erkannt werden. Dies betreffend wird die Datengrundlage, wie folgend beschrieben, erweitert.

Die Erweiterung soll anhand eines Beispiel verdeutlicht werden. In Listing 6.1 zeigt den Java-Code, welcher der CVE-Schwachstelle mit der ID `CVE-2017-1591` zugeordnet ist. Dabei handelt es sich um den originalen Code mit einem eingefügten Kommentar. Folglich ist dieser Code ein Typ-1-Klon dieser Sicherheitslücke.

Listing 6.1: Beispiel: Typ-1-Klon von CVE-2017-1591

```
1  public static String randomString(int length) {
2    if (length < 1) {
3      // Kommentar
4          return null;
5    }
6      // Create a char buffer to put random letters and numbers↩
           in.
7      char[] randBuffer = new char[length];
8      for (int i = 0; i < randBuffer.length; i++) {
9      randBuffer[i] = numbersAndLetters[randGen.nextInt(71)];
10   }
11     return new String(randBuffer);
12 }
```

Damit die anderen Klontypen erkannt werden können, muss der Code des Listings 6.1 entsprechend angepasst werden. Für einen Klon des zweiten Typen müssen die Namen von Variablen, Typen, Literalen und/oder Methoden geändert werden. In Listing 6.2 sind die jeweiligen Änderungen durchgeführt. Der Name des `char`-Arrays und der Name des Methodenparameters wurden geändert.

Listing 6.2: Beispiel: Typ-2-Klon von CVE-2017-1591

```
1  public static String randomString(int len) {
2    if (len < 1) {
3      // Kommentar
4          return null;
5    }
6      // Anderer Kommentar
7      char[] rndBuf = new char[len];
8      for (int i = 0; i < rndBuf.length; i++) {
9      rndBuf[i] = numbersAndLetters[randGen.nextInt(71)];
10   }
11     return new String(randBuffer);
12 }
```

Bei dem dritten Typ von Klonen enthält das kopierte Code-Fragment zusätzliche Statements, welche in dem anderen Code-Ausschnitt des Klonpaars nicht enthalten sind. In Listing 6.3 wurden die Statements in den Zeilen 3 und 11 hinzugefügt.

Listing 6.3: Beispiel: Typ-3-Klon von CVE-2017-1591

```
1  public static String randomString(int len) {
2    if (len < 1) {
3      System.out.println("Länge ist kleiner als 1");
4          return null;
5    }
6      // Anderer Kommentar
7      char[] rndBuf = new char[len];
8      for (int i = 0; i < rndBuf.length; i++) {
9      rndBuf[i] = numbersAndLetters[randGen.nextInt(71)];
10   }
11   String result = new String(randBuffer);
12   return result;
13 }
```

Die ursprüngliche Datenmenge für die Trainingsphase besteht aus 20 Java-Dateien, welche die Security-Schwachstellen aus Tabelle 6.1 enthalten. Für jede dieser Elemente müssen nun die Dateien erstellt werden, welche die weiteren zwei Typen von Code-Klonen abdecken. Wenn für jede Sicherheitslücke drei Dateien erstellt werden, so bestünde die Datengrundlage aus 60 Java-Dateien.

Die Genauigkeit von neuronalen Netzwerken ist stark an das zur Verfügung stehende Fundament von Trainingsdaten gebunden. Umso größer die Menge von Lerndaten, desto präziser kann ein Modell die gewünschte Funktion abbilden. Daher wurde beschlossen, dass die Anzahl an Dateien erhöht wird, um ein besseres Ergebnis gewährleisten zu können. Es wurden zusätzlich 340 Dateien erstellt, welche Klone von den 20 festgelegten Schwachstellen enthalten. Dabei wurde darauf geachtet, dass sowohl Typ-2- als auch Typ-3-Klone erstellt wurden. Folglich stehen 400 Java-Dateien mit den entsprechenden Security-Schwachstellen für die Trainingsphase zur Verfügung.

Wenn das Modell mit diesen Dateien die Trainingsphase durchläuft, so klassifiziert das Netz den übergebenen Code als Sicherheitsschwachstelle. Die Ausgabe wird dabei ein Wahrscheinlichkeitsvektor mit 20 Werten sein. Jede Stelle des Vektors gibt an, mit welcher Wahrscheinlichkeit der zu prüfende Software-Code die entsprechende Schwachstelle implementiert. Beispielhaft zeigt die erste Stelle des Vektors an, wie wahrscheinlich es ist, dass der Code die erste Schwachstelle implementiert hat. Augenscheinlich fehlt noch eine Kategorie für die Klassifikation. Bis jetzt wird lediglich die am wahrscheinlichste Sicherheitslücke dem Code zugewiesen. Allerdings kann auch der Fall eintreten, dass der Software-

Code gar keine Schwachstelle hat und somit vollkommen sicher ist. Damit auch diese Klasse berücksichtigt werden kann, müssen in der Trainingsgrundlage auch Java-Dateien enthalten sein, welche keine Schwachstelle haben. Hierfür wurden nocheinmal 20 Dateien erstellt.

Die gesamte Datengrundlage beläuft sich somit auf 420 Java-Dateien von denen 400 eine Schwachstelle implementieren und 20 ohne eine Sicherheitslücke.

6.2

Vergleichsgrundlagen für Neuronale Netze

Für die Evaluierung von neuronalen Netzwerken sind Metriken notwendig, mit denen die Effektivität der Modelle gemessen werden kann. Laut der Anforderung **[NRE2]** an die Evaluation sollen verschiedene Validierungsverfahren genutzt werden. Dabei ist es wichtig zu beachten, dass der Erkennungsprozess laut Anforderung **[R5]** Klone der Typen 1, 2 und 3 erkennen soll. Hierzu wird zum einen die *Leave-One-Out-Cross-Validation* und zum anderen Metriken aus dem Bereich des *Information Retrieval* genutzt. Beide Verfahren werden hier kurz dargestellt und anschließend durchgeführt. Die angewendeten Methoden aus dem Bereich des *Information Retrievals* sind aus dem Werk *Introduction to Information Retrieval* [50] von den Authoren Hinrich Schütze et al. Die einzelnen Metriken werden anhand einer Konfusionsmatrix definiert und beschrieben. Diese ist in Tabelle 6.2 dargestellt.

	Mit Schwachstelle	**Ohne Schwachstelle**
Schwachstelle gefunden	richtig positiv (rp)	falsch positiv (fp)
Keine Schwachstelle gefunden	falsch negativ (fn)	richtig negativ (rn)

Tabelle 6.2: Konfusionsmatrix in Anlehnung an (8.3), S.155 [50]

Im Folgenden werden die einzelnen Metriken mit den in der Tabelle dargelegten Werten beschrieben. Dabei werden lediglich die Abkürzungen verwendet (beispielsweise *rp* für *richtig positiv*).

6.2.1

Leave-Out-Out-Cross-Validation

Bei der **Leave-Out-Out-Cross-Validation** wird die Datengrundlage wie folgt verwendet. Die Anzahl der Datensätze sei durch $n \in \mathbb{N}$ definiert. Nun werden n-1 Datensätze für die Trainingsphase genutzt und der letzte Datensatz wird für die Überprüfung der Erkennung genutzt. Dieser Vorgang wird n-mal wiederholt, wobei der zu überprüfende Datensatz immer wechselt. Bei den vorliegenden Daten, welche aus 420 einzelnen Datensätzen besteht, wird der Vorgang folglich 420 mal durchgeführt, sodass jeder Datensatz einmal für den Code-Clone-Detection-Algorithmus verwendet wird. Die Überprüfung der Anforderung **[R5]** zur Erkennung aller Klontypen wird damit abgedeckt, da durch die Rotation des zu testenden Datensatzes auch die Erkennung der entsprechenden Klontypen überprüft wird.

6.2.2

Precision

Die **Precision P** gibt das Verhältnis der richtig kategorisierten Schwachstellen zu allen kategorisierten Schwachstellen an. Sie wird durch Formel 6.1 definiert.

$$P = \frac{rp}{rp + fp} \tag{6.1}$$

Es wird ein Precision-Wert von 1 angestrebt. Allerdings zeigt diese Metrik nur auf, wieviele der vorausgesagten Schwachstellen tatsächlich welche sind. Wenn eine von hundert Schwachstellen gefunden wird und gleichzeitig keine falsche Sicherheitslücke vorausgesagt wurde, so liegt die Presicion bei 1. Die 99 übersehenden Security-Schwachstellen werden in dieser Metrik nicht berücksichtigt. Dies wird durch die nächste Methode abgebildet.

6.2.3

Recall

Der **Recall R** setzt die Anzahl der gefundenen Sicherheitslücken mit der Anzahl der Schwachstellen in Beziehung, die hätten gefunden werden können. Die For-

mel 6.2 gibt die formale Definition.

$$R = \frac{rp}{rp + fn} \tag{6.2}$$

Auch hier sollte der Wert möglichst 1 annehmen. Das würde bedeuten, dass alle existierenden Schwachstellen gefunden wurden. Was diese Metrik allerdings nicht berücksichtigt, sind die falsch vorausgesagten Sicherheitslücken. Es kann also vorkommen, dass der Wert bei 1 liegt, allerdings im Gegenzug viel Software-Code als sicherheitskritisch deklariert wurde, welcher tatsächlich keine Schwachstelle enthält.

6.2.4

Accuracy

Die **Accuracy A** wird durch den Anteil der korrekten Klassifikationen beschrieben. Sie wird durch Formel 6.3 definiert.

$$A = \frac{rp + rn}{rp + rn + fp + fn} \tag{6.3}$$

Bei dieser Metrik *kann* es im Allgemeinen allerdings passieren, dass die Aussagekraft aufgrund der Verzerrung der Daten leidet. Da es im Alltag sehr viel mehr Software-Code gibt, welcher keine Schwachstelle enthält, könnte ein Security-Analyse-Programm einen relativ guten Accuracy-Wert erreichen, indem es bei jedem zu prüfenden Code-Fragment ausgibt, dass es keine Sicherheitslücke enthält. Angenommen es sollen 100 Code-Fragmente auf Schwachstellen überprüft werden, wovon ledigliche eins tatsächlich sicherheitskritisch ist. Wenn nun ein Algorithmus ausschließlich ausgibt, dass keine Sicherheitslücken existieren, so läge der Accuracy-Wert immernoch bei 99%.

6.2.5

F1-Measure

Da sowohl die Metrik *Precision* als auch der *Recall* die jeweils andere außer Acht lässt, werden beide verwendet um das sogenannte gewichtete harmonische Mittel zu bilden. Dieses wird auch **F-Measure** genannt und gibt einen Wert für die Performanzrate von zwei Faktoren (Precision und Recall). Bei der Berechnung können für beide Metriken ein Gewicht definiert werden, sodass sie das Ergeb-

nis unterschiedlich beeinflussen können. Formel 6.4[1] zeigt die formale Definition nach Schütze et al.

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \text{ mit } \beta^2 = \frac{1-\alpha}{\alpha} \tag{6.4}$$

Dabei soll gelten, dass $\alpha \in [0,1]$ und $\beta \in [0,\infty]$. Wenn die *Precision* und der *Recall* gleich gewichtet werden sollen, wie es für diese Evaluation der Fall ist, dann muss $\alpha = \frac{1}{2}$ und $\beta = 1$ gelten. Diese Gewichtung wird als **F1-Measure** bezeichnet und wird in Formel 6.5[2] dargestellt.

$$F_{\beta=1} = F_1 = \frac{2PR}{P+R} \tag{6.5}$$

6.3 Verwendete Hardware

Es wird zwischen zwei Evaluationen unterschieden: einer automatisierten und einer manuellen. Im weiteren Verlauf dieses Evaluationskapitels wird deutlich gemacht, worin die Unterschiede liegen. Bei der automatisierten Evaluation wurde eine große Menge an Konfigurationen für neuronale Netzwerke getestet. Um die Performanz dieser Parametervariation messen zu können, wurden unter anderem Zeiten gemessen. Da diese stark von der verwendeten Hardware abhängig ist, wird hier das für die Evaluation genutzte Systeme aufgezeigt. Tabelle 6.3 zeigt die wichtigsten Eigenschaften des Systems.

Betriebssystem	Windows 10 Pro
CPU	Intel(R) Xeon(R) CPU E3-1230 v3 @ 3.30GHz
Eclipse-Version	Oxygen.3a Release (4.7.3a)
Festplatte	Samsung SSD 840 EVO
Java-Version	1.8.0_181
RAM	16 GB

Tabelle 6.3: Systemeigenschaften der automatisierten Evaluation

Die manuelle Evaluation wurde auf einem zweiten System durchgeführt. Die Eckdaten dieser Umgebnung werden in Tabelle 6.4 dargestellt.

[1] Formel (8.5), Seite 156 in [50]
[2] Formel (8.6), Seite 156 in [50]

Betriebssystem	Linux Mint 18.1 Cinnamon 64-Bit
CPU	Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
Eclipse-Version	Oxygen.3a Release (4.7.3a)
Festplatte	Seagate ATA Disk 350
Java-Version	1.8.0_181
RAM	8 GB

Tabelle 6.4: Systemeigenschaften der manuellen Evaluation

6.4 Deep Feed Forward Neural Network

Die Datengrundlage und die Evaluationsmetriken wurden nun festgelegt. Ausstehend sind die verschiedenen Konfigurationen, welche laut Anforderung **[NRE1]** ebenfalls evaluiert werden sollen. Daher wird nun in Abschnitt 6.4.1 beschrieben, wie und welche Konfigurationen für die Evaluation ausgewählt wurden. Anschließend wird in Abschnitt 6.4.2 die Evaluation der Deep-Feed-Forward-Netzwerke anhand der zuvor definierten Metriken und der festgelegten Datengrundlage durchgeführt und ausgewertet.

6.4.1 *Konfigurationen eines DFF-Netzwerks*

Eine Konfiguration definiert alle Parameter eines neuronalen Netzwerks. Diese Einstellungen entscheiden darüber, wie gut ein Netzwerk die gewünschte Funktionalität abbilden kann. Um die bestmögliche Abbildung der Funktion, in diesem Fall die der Schwachstellenerkennung, zu finden, müssen viele Konfigurationen ausprobiert werden. Hierzu wurde ein kurzes Programm geschrieben, welche die definierte Datengrundlage nutzt und entsprechend die Parameter des Modells für jeden Durchlauf ändert. Somit werden unterschiedliche Konfigurationen getestet. Laut den Anforderungen **[RE2]** und **[RE3]** sollen bei den jeweiligen Durchläufen die Werte der Parameter und die Ergebnisse gespeichert werden. Die Ergebnisse werden hierbei durch das verwendete Framework *Deeplearning4j* errechnet. Dieses bietet eine Klasse `Evaluation` an, welche die *Precision*, den *Recall*, die *Accuracy* und den *F1-Measure* für ein getestetes Modell ausgibt. Es wird hiermit deutlich herausgestellt, dass diese Klasse lediglich für die Sichtung von Konfigurationen dient, welche eine hohe Effektivität gewährleisten. Die richtige Evaluati-

on der Netzwerke wird händisch implementiert, damit genau bekannt ist, wie die einzelnen Werte der Metriken zustande kommen.

Um effiziente Konfigurationen zu finden, wurden fünf Parameter definiert, welche für jeden Durchlauf angepasst werden. Dabei handelt es sich um die Anzahl der Iterationen während der Trainingsphase, die Anzahl der Neuronen auf den Hidden-Layern, die Lernrate, die Kostenfunktion und die gewählte Gewichtsinitialiserungsmethode. Die Anzahl der Schichten wurde bei den Deep-Feed-Forward-Netzwerken auf fünf festgelegt, da manuelle Tests zeigten, dass mit dieser Anzahl schnell vielversprechende Ergebnisse zu erzielen sind. Auf diese Weise wurden insgesamt 67790 verschiedene Konfigurationen getestet.

Die Ergebnisse der knapp 68 Tausend Durchläufe sind äußerst durchwachsen. Die Effizienz einiger Konfigurationen ist mehr als unzureichend, wohingegen andere sehr viel versprechende Ergebnisse liefern. In Tabelle 6.5 sind einige Konfigurationen mit ihren Evaluationsergebnissen dargestellt.

Es wurden die zehn besten Konfigurationen herausgefiltert. Dies wurde in Abhängigkeit der Summe der verwendeten Metriken getan.

#	# Iterationen	# Neuronen	Lernrate	Kostenfunktion	Gewichtsinitialisierung	Accuracy	Precision	Recall	F1-Measure
01	100	5	0.1	MSE	ONES	0.0475	0.0475	0.0476	0.0907
02	100	50	0.01	L2	XAVIER_UNIFORM	0.2950	0.3987	0.2937	0.3916
03	100	50	0.001	MSE	XAVIER	0.8975	0.8963	0.8989	0.8926
1	1000	50	0.1	MSE	UNIFORM	0.9975	0.9976	0.9976	0.9976
2	1000	50	0.1	SQUARED_LOSS	UNIFORM	0.9975	0.9976	0.9976	0.9976
3	1000	50	0.1	MEAN_SQUARED_LOGARITHMIC_ERROR	UNIFORM	0.9975	0.9976	0.9976	0.9976
4	1000	50	0.1	MEAN_SQUARED_LOGARITHMIC_ERROR	XAVIER_UNIFORM	0.9975	0.9976	0.9976	0.9976
5	1000	50	0.01	MSE	UNIFORM	0.9975	0.9976	0.9976	0.9976
6	1000	50	0.01	MSE	XAVIER_UNIFORM	0.9975	0.9976	0.9976	0.9976
7	1000	50	0.01	MSE	RELU_UNIFORM	0.9975	0.9976	0.9976	0.9976
8	1000	50	0.01	MSE	LECUN_UNIFORM	0.9975	0.9976	0.9976	0.9976
9	1000	50	0.01	MSE	VAR_SCALING_UNIFORM_FAN_IN	0.9975	0.9976	0.9976	0.9976
10	1000	50	0.01	MSE	VAR_SCALING_UNIFORM_FAN_OUT	0.9975	0.9976	0.9976	0.9976

Tabelle 6.5: DFF-Konfigurationen und dessen Evaluationsergebnisse

Die ersten drei Konfigurationen, welche mit *01* bis *03* gekennzeichnet sind, sollen Stichproben aus den Effizienzergebnissen darstellen, welche als Beispiele für die gemischte Qualität dienen sollen. Bei der ersten Konfiguration liegen sowohl Accuracy als auch Precision und Recall unter 5%. Bei der zweiten liegen die Accuracy und der Recall immerhin schon bei knapp 30% und die Precision erreicht einen Wert von knapp 40%. Die dritte Konfiguration lieferte erstmals eine Effizienz, welche die Werte der Metriken an die 90% brachten. Dennoch sind all diese Konfigurationen nicht ausreichend effizient genug, damit zuverlässig Software-Schwachstellen erkannt werden können.

Die letzten zehn Reihen der Tabelle 6.5 zeigen die Konfigurationen, welche die höchsten Werte für die entsprechenden Bewertungsmetriken erreichten. Wie zu erkennen ist, erreichen die Accuracy, die Precision und der Recall alle beinahe 100%. Auch die Verbindungsmetrik F1-Measure erreicht einen entsprechenden Wert.

Da die Effizienz bei allen Durchläufen gleich ist und somit über die verwendeten Metriken keine eindeutige Aussage darüber getroffen werden kann, welche Konfiguration die beste ist, muss ein weiteres Merkmal untersucht werden. Hierfür wird die zeitliche Dauer der Trainingsphasen betrachtet. Diese wurde für jede getestete Konfiguration gespeichert. Tabelle 6.6 zeigt die Lernzeit in Sekunden.

#	**Trainingszeit in Sekunden**
1	172
2	168
3	161
4	162
5	157
6	158
7	157
8	159
9	159
10	159

Tabelle 6.6: Trainingszeit der zehn effizientesten DFF-Konfigurationen

Wie zu erkennen ist, benötigen die Konfigurationen 5 und 7 aus der Tabelle 6.5 beide 157 Sekuden für die Trainingsphase und sind damit beide die sowohl effizientesten als auf performantesten Konfigurationen für das Deep-Feed-Forward-Netzwerk. Daher wird entschieden, dass die Konfiguration #5 für die händisch durchgeführte Evaluation gewählt wird.

6.4.2

Auswertung

Für die Metriken nach Hinrich et al. wurden 25 Java-Dateien aus der Datengrundlage gefiltert. Dabei wurden acht zufällig ausgesuchte Schwachstellen ausgewählt. Die Auswahl der Dateien kann nicht zufällig geschehen, da laut Anforderung **[R5]** alle Klontypen erkannt werden müssen. Durch die manuelle Auswahl wurde sichergestellt, dass alle Klontypen durch die jeweiligen Dateien abgedeckt sind. Zusätzlich wurden einige Dateien ohne Sicherheitslücke hinzugefügt, damit auch die *richtig-negativ*-Klasse abgedeckt wird. Die entsprechende Menge an Dateien wurde für die Evaluation aus der Trainingsmenge entfernt und ist in Tabelle 6.7 abgebildet.

Schwachstellen-ID	Anzahl Dateien
CVE-2006-2806	4
CVE-2007-6203	1
CVE-2008-5515	3
CVE-2011-1475	3
CVE-2011-3190	2
CVE-2016-6723	2
CVE-2017-0389	3
CVE-2017-9096	4
Keine Schwachstelle	3

Tabelle 6.7: Dateien für Evaluation nach Hinrich et al.

Die manuell durchgeführte Evaluation des Erkennungsprozesses mit dem Deep-Feed-Forward-Netzwerk ergibt die Konfusionsmatrix, welche in Tabelle 6.8 dargestellt ist.

	Mit Schwachstelle	Ohne Schwachstelle
Schwachstelle gefunden	22	0
Keine Schwachstelle gefunden	0	3

Tabelle 6.8: Konfusionsmatrix des Deep-Feed-Forward-Modells

Wie aus der Tabelle zu erkennen ist, wurden alle Dateien ihren jeweiligen Klassen zugeordnet. Alle Schwachstellen wurden als solche indentifiziert und sie wurden der richtigen CVE-ID zugewiesen. Ebenfalls wurden die Dateien, welche keine Sicherheitslücke enthalten, korrekter Weise nicht als sicherheitskritisch klassifiziert. Nun sollen die Werte der verwendeten Metriken errechnet werden.

Precision

Entsprechend der Precision-Formel 6.1 und der erhaltenen Konfusionsmatrix 6.8 ergbit sich die Precision aus Gleichung 6.6. Es wurde ein Wert von 1 erreicht, was bedeutet, dass alle erkannten Schwachstellen tatsächlich Sicherheitslücken beinhalten.

$$P = \frac{22}{22 + 0} = 1 \tag{6.6}$$

Recall

Entsprechend der Recall-Formel 6.2 und der erhaltenen Konfusionsmatrix 6.8 ergbit sich der Recall aus Gleichung 6.7. Es wurde ein Wert von 1 bedeutet, dass alle vorhandenen Schwachstellen erkannt und korrekt kategorisiert wurden.

$$R = \frac{22}{22 + 0} = 1 \tag{6.7}$$

Accuracy

Entsprechend der Accuracy-Formel 6.3 und der erhaltenen Konfusionsmatrix 6.8 ergbit sich die Accuracy aus Gleichung 6.8. Es wurde ein Wert von 1 erreicht, was bedeutet, dass alle getätigten Klassifikationen korrekt waren.

$$A = \frac{22 + 3}{22 + 3 + 0 + 0} = 1 \tag{6.8}$$

F1-Measure

Entsprechend der F1-Measure-Formel 6.5 und der erhaltenen Konfusionsmatrix 6.8 ergbit sich der F1-Measure-Wert aus Gleichung 6.9.

$$F_{\beta=1} = F_1 = \frac{2 * 1 * 1}{1 + 1} = 1 \tag{6.9}$$

Leave-One-Out-Cross-Validation

Bei der Leave-One-Out-Cross-Validation wurden die 420 Durchläufe entsprechend der Definition des Validierungsverfahrens durchgeführt. Hierbei wurde bei jedem Durchlauf der nicht für die Trainingsphase genutzte Datensatz korrekt seiner entsprechenden Klasse (Schwachstelle mit CVE-ID oder keine Schwachstelle

vorhanden) zugeordnet. Somit liegt die Erkennungsrate bei diesem Verfahren bei 100%. Die korrekte Klassifikation benötigte bei den 420 Iterationen zwischen 27 und 421 Millisekunden. Im Schnitt lag die Dauer bei 135 Millisekunden.

6.5 Long Short-Term Memory Neural Network

6.5.1 Konfigurationen eines LSTM-Netzwerks

Genau wie bei den Deep-Feed-Forward-Netzwerken, müssen für den Modelltyp *Long-Short-Term-Memory* die effizientesten Konfigurationen gefunden werden. Dafür wurde auch hier ein kurzes Programm implementiert, welches verschiedene Konfigurationen definiert und damit ein LSTM-Netzwerk initialisiert. Gleichend zu dem Suchprozess für die Konfigurationen der Deep-Feed-Forward-Modelle, wird die Klasse `Evaluation` des Frameworks *Deeplearning4j* verwendet. Aufgrund der komplexeren Struktur von LSTM-Modellen, nehmen die Trainingsphasen der Netzwerke sehr viel mehr Zeit in Anspruch.

Es wurden sechs Parameter definiert, welche für jede zu testende Konfiguration geändert werden können. Diese bestehen aus der Anzahl der Iterationen, der Batchsize (unterteilt die Datensätze während der Trainingsphase in entsprechend große Teilmengen), dem Seed (wird für die Gewichtsinitialisierung benötigt), der Anzahl der Neuronen der Hidden-Layer, der Lernrate und der Anzahl der Hidden-Layer. Die Anzahl der Hidden-Layer wurde hier mit als Parameter aufgenommen, da durch manuelle Tests keine Anzahl ermittelt werden konnte, welche mehrheitlich zu guten Evaluationsergebnissen führte. Dafür wurden die Gewichtsinitialisierungsmethode und die Kostenfunktion durch manuelle Tests ermittelt. Mit der beschriebenen Weise wurden 306 Konfigurationen getestet. Der große Unterschied zwischen den getesteten Anzahlen an Konfigurationen für DFF- und LSTM-Modelle ist auf die Dauer der Trainingsphasen zurückzuführen. Dies wird zu einem späteren Zeitpunkt genauer erklärt.

Tabelle 6.9 zeigt die zehn effizientesten Konfigurationen entsprechend der festgelegten Bewertungsmetriken. Dabei ist bereits ein weitaus größerer Unterschied bei den erzielten Werten zu bemerken als bei den Werten der DFF-Modelle.

#	# Iterationen	Batchsize	Seed	# Neuronen	Lernrate	# Hidden-Layer	Accuracy	Precision	Recall	F1-Measure
1	1000	25	100	50	0.005	2	0.9697	0.9762	0.9786	0.9743
2	10000	10	50	25	0.05	2	0.9545	0.9690	0.9690	0.9626
3	5000	50	50	50	0.05	5	0.9545	0.9722	0.9627	0.9639
4	5000	50	100	25	0.005	2	0.9545	0.9643	0.9690	0.9622
5	1000	25	50	25	0.005	2	0.9545	0.9627	0.9690	0.9610
6	1000	10	100	50	0.005	2	0.9545	0.9667	0.9627	0.9605
7	1000	25	100	50	0.005	2	0.9545	0.9667	0.9627	0.9595
8	5000	25	100	50	0.005	2	0.9545	0.9603	0.9548	0.9489
9	1000	25	100	25	0.05	2	0.9394	0.9595	0.9595	0.9520
10	5000	10	50	25	0.05	5	0.9394	0.9571	0.9595	0.9505

Tabelle 6.9: LSTM-Konfigurationen und dessen Evaluationsergebnisse

Laut der Tabelle 6.9 ist die erste Konfiguration die effizienteste. Wie allerdings bereits oben geschrieben wurde, sind die Trainingszeiten von LSTM-Netzwerken um ein Vielfaches länger als die der DFF-Modelle. Es muss geprüft werden, ob die Trainingszeit der effizientesten Konfiguration auch in einem guten Verhältnis zu der Trainingszeit steht. Es soll gewährleistet werden, dass ein Anstieg der Dauer der Lernphase auch mit einem Anstieg der Effizienz zu rechtfertigen ist. Folglich wurden die Trainingszeiten der Konfigurationen gemessen und werden in Tabelle 6.10 aufgezeigt.

#	in Sekunden	in Minuten	in Stunden
1	6081	101.35	1.69
2	46422	773.7	12.895
3	51667	961.28	24.35
4	10450	174.167	2.903
5	3080	51.33	0.856
6	6174	102.9	1.715
7	4031	67.1833	1.12
8	24461	407.683	6.795
9	4657	77.617	1.29
10	23137	385,617	6.427

Tabelle 6.10: Trainingszeit der zehn effizientesten LSTM-Konfigurationen

Wie in der Tabelle 6.10 zu erkennen ist, sind die Trainingszeiten um ein Vielfaches höher im Vergleich zu der Lerndauer von DFF-Modellen. Dies ist auch der Grund, weswegen entsprechend wenig Konfigurationen getestet werden konnten. Einige Variationen der Parameter ließen die Trainingszeit auf über drei Tage steigen. Die beste Effizienz bietet Konfiguration Nummer 1. Allerdings ist die Trainingszeit fast doppelt so lange wie die der fünften Konfiguration. Da gleichzeitig die Werte der verwendeten Metriken der fünften Parametervariation im Verhältnis nicht viel schlechter sind als die der ersten, bleibt abzuwägen, ob der Gewinn von ungefähr 1% pro Metrik eine Verdoppelung der Trainingszeit rechtfertig. Aufgrund der deutlichen Verschlechterung der Lerndauer wird hier entschieden, dass die fünfte Konfiguration für die manuell durchgeführte Evaluation verwendet wird.

6.5.2

Auswertung

Die Anwendung der Metriken nach Hinrich et al. wird mit den selben 25 Java-Dateien durchgeführt, welche bereits für die manuelle Evaluation der Deep-Feed-Forward-Modelle verwendet wurden. Es müssen die gleichen Dateien sein, damit

der Vergleich zwischen den Netzwerktypten gezogen werden kann. Diese wurden bereits in Tabell 6.7 aufgezählt.

Die manuelle Evaluation des Erkennungsprozesses mit dem Long-Short-Term-Memory ergibt die Konfusionsmatrix aus Tabelle 6.11. Wie bereits bei den DFF-Modellen wurden alle Schwachstellen korrekt erkannt und alle sicherheitsunkritischen Dateien wurden also solche identifiziert.

	Mit Schwachstelle	**Ohne Schwachstelle**
Schwachstelle gefunden	22	0
Keine Schwachstelle gefunden	0	3

Tabelle 6.11: Konfusionsmatrix des Long-Short-Term-Memory-Modells

Precision

Entsprechend der Precision-Formel 6.1 und der erhaltenen Konfusionsmatrix 6.11 ergbit sich die Precision aus Gleichung 6.10. Es wurde ein Wert von 1 erreicht, was bedeutet, dass alle erkannten Schwachstellen tatsächlich Sicherheitslücken beinhalten.

$$P = \frac{22}{22 + 0} = 1 \tag{6.10}$$

Recall

Entsprechend der Recall-Formel 6.2 und der erhaltenen Konfusionsmatrix 6.11 ergbit sich der Recall aus Gleichung 6.11. Es wurde ein Wert von 1 erreicht, was bedeutet, dass alle vorhandenen Schwachstellen erkannt und korrekt kategorisiert wurden.

$$R = \frac{22}{22 + 0} = 1 \tag{6.11}$$

Accuracy

Entsprechend der Accuracy-Formel 6.3 und der erhaltenen Konfusionsmatrix 6.11 ergbit sich die Accuracy aus Gleichung 6.12. Es wurde ein Wert von 1 erreicht, was bedeutet, dass alle getätigten Klassifikationen korrekt waren.

$$A = \frac{22 + 3}{22 + 3 + 0 + 0} = 1 \tag{6.12}$$

F1-Measure

Entsprechend der F1-Measure-Formel 6.5 und der erhaltenen Konfusionsmatrix 6.11 ergbit sich der F1-Measure-Wert aus Gleichung 6.13. Es wurde ein Wert von 1 erreicht.

$$F_{\beta=1} = F_1 = \frac{2 * 1 * 1}{1 + 1} = 1 \tag{6.13}$$

Leave-One-Out-Cross-Validation

Es wurden die 420 Durchläufe des Validierungsverfahrens ausgeführt. Bei jedem Durchlauf wurde der Datensatz, welcher nicht für die Trainingsphase verwendet wurde, richtig erkannt und zugeordnet. Die Erkennungsrate liegt bei diesem Verfahren bei 100%. Die korrekte Klassifikation benötigte bei den 420 Iterationen zwischen 517 und 2107 Millisekunden. Im Schnitt lag die Dauer bei 1167 Millisekunden.

6.6 Vergleich der Netzwerktypen

Die beiden für die Evaluation gewählten Netzwerktypen *Deep-Feed-Forward* und *Long Short-Term Memory* haben trotz des grundlegend ähnliches Aufbaus der Neuronen, zwei sehr verschiedene Arbeitsweisen. Dies spiegelt sich auch in den Ergebnissen der Evaluation wider. Bei dem Vergleich sollen die Effizienz und die Performanz betrachtet werden. Anschließend wird ein Fazit über die Eignung der unterschiedlichen Modelltypen für den Bereich des Code-Clone-Detections gezogen werden.

6.6.1 Effizienz

Zunächst wird ein Blick auf die Effizienz der unterschiedlichen Algorithmen geworfen. Bereits bei der Sichtung der effektivsten Konfigurationen wurde deutlich, dass beide Modelltypen sehr gute Ergebnisse für die Erkennungsrate liefern. Tabelle 6.12 zeigt nocheinmal die fünf besten Konfigurationen beider Netzwerke.

#	Typ	Accuracy	Precision	Recall	F1-Measure
1-5	DFF	0.9975	0.9976	0.9976	0.9976
1	LSTM	0.9697	0.9762	0.9786	0.9743
2	LSTM	0.9545	0.9690	0.9690	0.9626
3	LSTM	0.9545	0.9722	0.9627	0.9639
4	LSTM	0.9545	0.9643	0.9690	0.9622
5	LSTM	0.9545	0.9627	0.9690	0.9610

Tabelle 6.12: Fünf besten Konfigurationen beider Netzwerktypen

Die manuell ausgeführte Evaluation bestätigte den ersten Eindruck. Die beiden verwendeten Validierungsmethoden, die Bewertungsmetriken nach Hinrich et al. und die Leave-One-Out-Cross-Validation, erzielten einen perfekten Wert. Sowohl Accuracy, Presicion, Recall und F1-Measure, als auch das Ergebnis der Leave-One-Out-Cross-Validation ergaben bei der Durchführung 100%. In Tabelle 6.13 kumuliert nocheinmal die Ergebnisse der manuellen Evaluation.

Typ	Accuracy	Precision	Recall	F1-Measure	LOOCV
DFF	100%	100%	100%	100%	100%
LSTM	100%	100%	100%	100%	100%

Tabelle 6.13: Ergebnisse der manuellen Evaluation

6.6.2

Performanz

Bei der Performanz sind erhebliche Unterschiede zu erkennen. Aufgrund der verschiedenen Arbeitsweisen und der Struktur der jeweiligen Netzwerkarten divergieren die Zeiten der Trainingsphasen sehr stark. Während die Dauer für Deep-Feed-Forward-Netze im Minutenbereich liegen, so entsprechen die Zeiten der Lernphase für Long-Short-Term-Memory-Modell Stunden bis Tage. Tabelle 6.14 zeigt noch einmal die kumulierten Ergebnisse im Bezug auf die Zeiten der Trainingsphasen der einzelnen Modelltypen für die jeweils fünf effizientesten Konfigurationen.

#	Typ	Zeit der Trainingsphase in Sekunden
1	DFF	172
2	DFF	168
3	DFF	161
4	DFF	162
5	DFF	157
1	LSTM	6081
2	LSTM	46422
3	LSTM	51667
4	LSTM	10450
5	LSTM	3080

Tabelle 6.14: Trainingszeiten der jeweils fünf effizientesten Konfigurationen

6.6.3

Fazit

Die Evaluation der verschiedenen Netzwerkarten sollte die Vor- und Nachteile der jeweiligen Modelle aufdecken. Die erhobenen Ergebnisdaten dieser Evaluation zeigen deutlich, dass der neuronale Netzwerktyp *Deep Feed Forward* dem eines *Long Short-Term Memory* in jeder Hinsicht überlegen ist. Auch wenn es bei der manuellen Evaluation keinen Unterschied bei der Effizienz der beiden Erkennungsprozesse gab, desto deutlicher unterschieden sich die Modelle in dem Punkt der Trainingszeiten. Ein DFF-Netzwerk benötigt lediglich wenige Minuten für die Initialisierung der Gewichte, wohingegen ein LSTM-Modell bis zu drei Tage benötigt. Ein solch langwieriger Erkennungsprozess kann in dem Bereich des Security-Code-Clone-Detection nicht verwendet werden, da die Initialisierung zu lange dauert. Wenn der Prozess nach der Trainingsphase eine stark höhere Erkennungsrate hätte, welche einen Einsatz dieses Modells rechtfertigen würde, so bliebe abzuwägen, ob es Verwendungsmöglichkeiten dieses Netzwerktyps gibt. Da aber ein DFF-Modell eine gleiche bis bessere Erkennungsrate hat und dabei nur einen Bruchteil der Lerndauer benötigt, ist der Netzwerktyp *Long Short-Term Memory* für diesen Bereich ungeeignet.

Somit wird hiermit definiert, dass für den Prototypen der Netzwerktyp *Deep Feed Forward* für die Umsetzung gewählt wird. Dies gewährleistet es, dass das Software-Tool die höchstmögliche Performanz und Effizienz erreicht. Der Typ *Long Short-Term Memory* wird folglich für das Tool nicht berücksichtigt.

6.7 Vergleich zum SCCD

Aufgrund der zuvor beschriebenen Ergebnisse wird das Modell des *Long Short-Term Memory* in diesem Bereich nicht berücksichtig.

Die bisherige Evaluation und der getätigte Vergleich bezog sich lediglich auf die Prozesse des maschinellen Lernens. Nun soll zusätzlich geprüft werden, wie der dynamische Algorithmus mit der Verwendung des neuronalen Netzes des Typen *Deep Feed Forward* im Vergleich zu einem statischen und traditionellen Ansatz des Code-Clone-Detections abschneidet. Hierzu wird der bereits erwähnte Prozess von Herrn Brunotte des Eclipse-Plugins *Security Code Clone Detector* (kurz SCCD) [9] verwendet. Ein Vergleich kann gezogen werden, da der Author in seiner Masterthesis ebenfalls eine Evaluation des entwickelten Plugins durchgeführt hat. Es bleibt allerdings herauszustellen, dass die Ergebnisdaten nicht auf dem selben System kumuliert wurden. Da jedoch keine große Differenz bei der Rechenleistung besteht, werden die Unterschiede der Systeme vernachlässigt.

6.7.1 Effizienz

Bei dem Vergleich der Effizienz der beiden Erkennungsalgorithmen werden jeweils die besten Konfigurationen miteinander verglichen. Tabelle 6.15 zeigt die erzeugten Ergebniswerte der entsprechenden Metriken. Die Werte des SCCDs wurden aus der Masterarbeit [9] von Seite 67 aus der Tabelle 6.14 gefiltert. Es ist zu erkennen, dass es keinen Unterschied zwischen den Algorithmen im Bezug auf die Effizienz gibt. Somit sind die Erkennungsprozesse in diesem Punkt gleichwertig.

Tool	Precision	Recall	F1-Measure
SecVuLearner	100%	100%	100%
SCCD	100%	100%	100%

Tabelle 6.15: Ergebniswerte der Metriken beider Erkennungsprozesse

6.7.2

Performanz

Für den Vergleich der Performanz müssen die Arbeitsweisen der Algorithmen betrachtet werden. Bei dem maschinellen Lernen werden die Rohdaten tranformiert und als Datengrundlage für die Trainingsphasen der neuronalen Netzwerke verwendet. Das Wissen um bekannte Schwachstellen und die möglichen Code-Klone wurde also abstrahiert und dem System komplett übergeben. Der Wissensstand über Sicherheitslücken wird nach der Lernphase durch die gelernten Gewichte des Modells abgebildet. Dies wird folgend als dynamischer Ansatz angesehen.

Der statische Ansatz hat eine andere Herangehensweise. Hierbei werden die bekannten Sicherheitslücken und die zu testenden Code-Fragmente zunächst einem Tokenizer übergeben, welcher sogenannte *Code Blocks* erstellt [9]. Daraus wird im nächsten Schritt ein *Inverted Index* erstellt, welcher wie eine Datenbank aufgefasst werden kann, bei der zu jedem Token ein Verweis hinterlegt wird, wo dieser Token im Quellcode vorkommt [9]. Weitergehend werden Klon-Kandidaten aus dem erstellten Inverted Index erzeugt. Die Kandidaten werden anschließend mit Hilfe einer Vergleichsfunktion verifiziert [9]. So wird festgestellt, ob zwei Code-Fragmente ein Klonpaar darstellen. Es ist zu erkennen, dass das Wissen um bekannte Sicherheitslücken in der Rohform bzw. auf tokenbasierter Ebene besteht.

In Tabelle 6.16 werden die Zeiten angegeben, welche für eine korrekte Klassifizierung benötigt wurde. Die Daten des SCCD-Tools wurden aus den Tabellen 6.19 bis 6.21 auf den Seiten 70 und 71 der Ausarbeitung [9] ermittelt.

Tool	**Zeit in Millisekunden**
SecVuLearner	27 bis 421
SCCD	3901 bis 8872

Tabelle 6.16: Zeitvergleich einer korrekten Klassifikation

Auf den ersten Blick sieht es so aus, als wäre die Methode des maschinellen Lernens deutlich schneller. Dies beruht darauf, dass das Wissen um die Schwachstellen durch das System selbst abgebildet wird. Allerdings muss vehement bedacht werden, dass der Vorgang des Abstrahierens dieses Wissens eine relativ lange Zeitspanne in Anspruch nimmt. In dem Falle der verwendeten Konfiguration des Netzwerks benötigte die Trainingsphase etwa 2 Minuten und 30 Sekunden. Diese Zeit für die Initialisierung des Modells fällt bei dem statischen Ansatz weg.

6.7.3

Fazit

Der Vergleich der unterschiedlichen Ansätze zeigt, dass der Erkennungsprozess mit neuronalen Netzwerken mit den statischen Methoden mehr als nur mithalten kann. Die Effizienz beider Methoden gleichen sich und erreichen eine sehr gute Erkennungsrate. Die Performanz unterscheidet sich in einigen Punkten, jedoch kann gesagt werden, dass keiner der beiden Algorithmen den anderen bezüglich der Erkennungszeit aussticht. Wenn bei einem Programm das Modell lediglich einmal initialisiert werden muss, so bietet sich die Variante mit einem neuronalen Netzwerk an, da die reine Erkennungszeit deutlich kürzer ist. Muss das Modell bei einer Software häufig neu geladen werden, so hat der statische Ansatz den Vorteil, dass nur wenig Zeit für eine Initialisierung benötigt wird.

KAPITEL 7

Fazit / Ausblick

7.1 Fazit

Die Evaluation der verschiedenen Typen von neuronalen Netzwerken zeigte, dass ein Deep-Feed-Forward-Modell im Vergleich zu einem Long-Short-Term-Memory-Netzwerk bei der automatisierten Schwachstellenerkennung deutlich besser abschneidet. LSTM-Modelle benötigen für eine Trainingsphase eine zu lange Zeitspanne, damit sie effektiv eingesetzt werden können. Die Lernphasen eines DFF-Netzwerk erfordern dagegen nur einen Bruchteil dieser Dauer und haben gleichzeitig eine höhere Erkennungsrate. Diese Methode des maschinellen Lernens ist somit durchaus für den Gebrauch in dem Gebiet des Security-Code-Clone-Detections anwendbar.

Weiter konnte durch die Evaluation bewiesen werden, dass dieser dynamische Ansatz ebenfalls mit den statischen lexikalischen Herangehensweisen hinsichtlich der Erkennungsrate sowie der Ausführungsgeschwindigkeit mithalten kann. Auch wenn sich die Einsatzmöglichkeiten der verschiedenen Arbeitsweisen unterscheiden, so kann eine Anwendung von DFF-Modellen zweifellos gerechtfertigt sein.

Zusätzlich wurde eine prototypische Software namens *SecVuLearner* umgesetzt, welche das evaluierte Netzwerk als Kernalgorithmus für die Erkennung von Sicherheitslücken und deren möglichen Klone nutzt. Die Trainingsdaten werden anhand eines Security-Software-Repositories erstellt. Bei der Suche nach Sicher-

heitslücken werden alle Java-Dateien eines Software-Repositories auf mögliche Schwachstellen getestet. Dazu wird der Inhalt der akquirierten Dateien in einem Vorverarbeitungsprozess vektorisiert und in CSV-Dateien abgespeichert. Diese werden dann dem DFF-Netzwerk als Eingabe übergeben, welches für jede Datei klassifiert, ob eine Schwachstelle vorhanden ist oder nicht. Falls eine Schwachstelle vorhanden sein sollte, so wird gleichzeitig ausgegeben, um welche es sich dabei handelt. Die Menge der erkennbaren Sicherheitslücken kann dabei manuell erweitert werden.

7.2 Ausblick

Das Feld des maschinellen Lernens umfasst eine große Menge von verschiedenen Typen neuronaler Netzwerke. Es wäre interessant zu sehen, wie die Effizienz und die Performanz anderer Modellarten in diesem Gebiet sind. Da der Prototyp die Netzwerke mit einer Oberklasse abstrahiert, ist hier eine Erweiterung von diversen Modelltypen einfach durchzuführen.

Die Evaluation wurde auf verhältnismäßig kleinen Datenmengen durchgeführt. Daher wäre es sehr informativ, wie sich die Erkennungsraten ändern würden, wenn die Menge an Schwachstellen deutlich erhöht wird. Ebenfalls kann weiter geprüft werden, wie sich die Klassifikationsrate im Bezug auf nicht vorhandene Schwachstellen verhält. Da das getestete Netzwerk mit kleinen Trainingsmengen an Negativbeispielen arbeitet, würde die Precision bei dem Test mit vielen Dateien ohne Schwachstelle wahrscheinlich deutlich leiden. Die gälte es zu überprüfen.

Falls der SecVuLearner aus dem Status eines Prototypen kommen sollte, kann die Benutztungsoberfläche durch eine grafische Komponente erweitert werden. Das Hinzufügen von Funktionalitäten wie beispielsweise das Anzeigen von Metadaten der gefundenen Schwachstellen oder der Vorschlag von möglichen Behebungen der Sicherheitslücken würde über eine Konsolenanwendung nicht bedienungsfreundlich sein.

7.3 Gültigkeit dieser Arbeit

Wie bereits im Ausblick angedeutet, wurde die Evaluation auf relativ wenig Daten ausgeführt. Insgesamt lagen 420 Datensätze vor, welche für die Trainings- und Testphasen der Netzwerke verwendet werden konnten. Die Aussagekraft der durchgeführten Methoden bezüglich der Accuracy, der Precision, des Recalls und des F1-Measure beziehen sich somit auf die vorliegende Datenmenge. Im Bereich des maschinellen Lernens ist es allerdings nicht unüblich, dass mehrere Tausend Datensätze für die jeweiligen Phasen zur Verfügung stehen. Da die Datengrundlage allerdings aufgrund der unterschiedlichen Klontypen manuell erstellt werden musste, konnte die Menge der Daten während der Bearbeitungszeit dieser Masterthesis nicht erhöht werden.

Um den Prototypen möglichst modular und erweiterbar zu gestalten, wurde der Vorverarbeitungsalgorithmus so abstrahiert, dass er auf beide Netzwerkarten passt. Da die Arbeitsweisen der Netzwerke allerdings sehr unterschiedlich sind, könnte es angenommen werden, dass die Werte der Evaluationsmetriken anders ausfallen würden, wenn für jeden Typ von Netzwerk ein spezifischer Preprocessing-Vorgang entwickelt werden würde.

Die prototypische Umsetzung des SecVuLearners zeigt eine beispielhafte Anwendung für den Gebrauch von neuronalen Netzwerken in dem Bereich des Security-Code-Clone-Detections. Es herauszustellen, dass dieser Prototyp als ein solcher anzusehen ist und nicht für den tatsächlichen Einsatz verwendet werden kann. Dafür bedürfte es einiger Anpassungen des Quellcodes.

KAPITEL 8

Verwandte Arbeiten

Das Thema des maschinellen Lernens hat in der aktuellen Forschung einen fundamentalen Platz eingenommen und wird immer wichtiger. Im Speziellen finden Deeplearning-Algorithmen in immer mehr und äußerst verschiedenen Bereichen Anwendungen [3]. In vielen Fällen wird für die Lösung einer Aufgabe aus diesem Bereich ein neuronales Netz genutzt. Ein stetig wachsendes Gebiet für neuronale Netze ist die Klassifizierung. Es gibt etwaige Beispiele in denen mit Hilfe von Netzwerken eine Klassifikation von entsprechenden Daten durchgeführt wird. Ein Beispiel ist in dem Paper *An Approach For Iris Plant Classification Using Neural Network* [57] beschrieben. Hierbei werden Pflanzen der Gattung *Iris* anhand ihrer Kelch- und Blütenblätter hinsichtlich ihrer genauen Art klassifiziert.

Da sich die Einsatzgebiete von neuronalen Netzwerken sehr unterscheiden, divergieren ebenso die Anforderungen an die lernenden Algorithmen. Bei den meisten Anwendungsbereichen sind speziell für die Problemstellung angepasste Netzwerke erforderlich. Folglich werden verschiedene Arten von neuronalen Netzen für die unterschiedlichen Lösungsansätze genutzt. Für die vielfältigen Arten von Netzwerken gibt es widerum mannigfaltige Konfigurationen von Variablen, die die Arbeitsweise der Modelle[1] beeinflussen.

Diese Arbeit befasst sich mit dem Zusammenspiel von den verschiedenen Arten der neuronalen Netze, mit deren Kofigurationen und der Klassifikation von Software-Code-Fragmenten. Diese Code-Ausschnitte sollen mit Hilfe von Deeplearning-Algorithmen als Schwachstelle in einer Software erkannt werden. Um diesen Vorgang zu ermöglichen, muss das Modell in der Lernphase Code-

[1] Im Laufe dieser Thesis wird das Wort *Modell* als ein Synonym für *neuronales Netzwerk* verwendet.

Beispiele durchlaufen, welche zuvor bereits als Schwachstellen deklariert wurden. Für bekannte Schwachstellen gibt es die CVE-Datenbank, welche jedem Schwachpunkt eine ID und weitere Informationen zuordnet (wie zum Beispiel einen Verweis auf einen Code-Ausschnitt in einem Repository wie Git-Hub). Die unterschiedlichen Netze werden mit diesen hinterlegten Code-Beispielen ihre Lernphase durchlaufen, sodass sie ähnliche Code-Fragmente möglichst eindeutig zu der jeweilgen ID zuordnen können. Falls Code-Ausschnitte eindeutig als Schwachstelle deklariert wurden, allerdings noch kein entsprechender Eintrag in der CVE-Datenbank existiert, so soll es dennoch eine Möglichkeit geben diesen Schwachpunkt zu erkennen.

Abstrahiert man die fachliche Zuordnung einer ID zu einem Code-Fragment und den technischen Vorgang des Lernens, ist diese Klassifizierung von Software-Code folglich ein Algorithmus, welcher als Code-Clone-Detection bekannt ist. Bei diesem Vorgang werden ähnliche und gleiche Code-Fragmente mit vorgegebenen Beispielen mit einander verglichen und als Klon erkannt.

Es gibt einige Ausführungen, die sich mit dem Thema von Code Clone Detection im Bereich des maschinellen Lernens befassen. Ein damals neu verfolgter Ansatz ist die Ausarbeitung *Deep Learning Code Fragments for Code Clone Detection* [62]. Hier werden rekurrente neuronale Netzwerke (RNN) als Lösungsansatz verwendet. Die Code-Fragmente werden zunächst zu Vektoren mit kontinuierlichen Werten verarbeitet. Ein neuronales Netz charakterisiert dann die Code-Fragmente anhand von Sequenzen der erstellten Vektoren. Nach Aussagen der Autoren ist die Präzision ihrer Methode vergleichbar mit bisherigen Code-Clone-Detection-Ansätzen. Allerdings gäbe es noch Optimierungsbedarf bei den Lernphasen von rekurrenten neuronalen Netzen in dem Bereich der Klonklassifikation.

Ein weiterer Ansatz wird in dem Tool *CCLearner* [35] verwendet. Hier werden die Code-Fragmente untersucht und die Methoden extrahiert. Die extrahierten Methoden werden von einem Tokenizer zu einer Sequenz von Token verarbeitet. Diese Sequenzen werden dann untereinander verglichen und es wird ein Vektor gebildet, der die Ähnlichkeit der verglichenen Methoden beschreibt. Diese Vektoren werden dann für die Trainingsphase eines Convolutional Neural Networks (CNN) genutzt, eine andere Art von neuronalen Netzwerken. Ebenso dienen solche Vektoren zum Erkennen von Klonen. Die Ergebnisse zeigen, dass die Effektivität des Ansatzes mit CNNs gleich gut oder besser ist, als von Ansätzen mit herkömmlichen Methoden, welche kein maschinelles Lernen verwenden. Ein Vorteil von neuronalen Netzen in diesem Gebiet sei, dass man keine speziellen Algorithmen für zuvor bestimmte Klontypen entwickeln muss, sondern das Netz

jeden Klontyp erkennt, welcher nicht auf Semantik beruht.

Für die Problemstellung der Code-Clone-Detection mit Hilfe von neuronalen Netzen gibt es folglich viele verschiedene Ansätze. Jede Art von Netzen ist aufgrund der unterschiedlichen Struktur der Modelle für ein gewisses Gebiet prädestiniert. Rekurrente neuronale Netzwerke werden bei Problemen bevorzugt, in denen Sequenzen als Eingaben verwendet werden. Hierbei spielen die Längen der Sequenzen keine große Rolle, da immer nur einzelne Werte verarbeitet werden und die Sequenz nach und nach abgearbeitet wird. Eine variable Länge der Eingaben können durch Deep-Feed-Forward-Netzwerke (DFF) nur schwer verarbeitet werden, da hier immer ein fester Eingabevektor von Nöten ist. Sind die Daten jedoch in solch einer Form werden hier oftmals die CNNs bevorzugt, da die Trainingsphasen im Vergleich zeitlich meistens kürzer ausfallen, um eine akzeptable Präzision zu erreichen.

Um festzustellen, welche Art von neuronalem Netz für ein bestimmtes Problem verwendet werden sollte, um die optimale Effektivität gewährleisten zu können, ist ein Vergleich der Netze erforderlich. Eine Ausführung, die diesen Vergleich zeigt, ist *Comparative Study of CNN and RNN for Natural Language Processing* [65]. In diesem Paper wird auf die Funktionalität von neuronalen Netzwerken in dem Bereich des Natural Language Processings (NLP) eingegangen. In diesem Gebiet werden Texte in der natürlichen Sprache unter verschiedenen Gesichtspunkten verarbeitet. Es wird beispielsweise der Tonus des Textes automatisch bestimmt. Des Weiteren werden zu vorgegebenen Fragen Antworten generiert und überprüft, wie gut diese Antworten auf die gestellten Fragen passen. Für diese Bereiche des NLPs werden in dieser Arbeit die verschiedenen Arten der Netze miteinander verglichen. Die Ergebnisse zeigen, dass RNNs in vielen Fällen den CNNs überlegen sind. Allerdings gibt es auch einzelne Bereiche, wie das Problem der automatisch generierten Antworten, in denen die CNNs deutlich besser agieren.

Der Vergleich der Netzwerkarten ist folgens ein wichtiger Bestandteil im Umgang mit maschinellem Lernen. Da sich die Anforderungen an die Modelle bei jeder Problemstellung unterscheiden, muss ein Vergleich für jedes neues Einsatzgebiet erbracht werden. Auch wenn CNNs und RNNs schön länger in dem Bereich der Code-Clone-Detection verwendet wird, so gibt es nach bestem Wissen und Gewissen des Autors keinen publizierten Vergleich von den Netzwerkarten in dem Gebiet der Klassifikation von Software-Code-Fragmenten.

Maschinelles Lernen wird bereits in vielen sicherheitsrelevanten Bereichen angewendet. Nicht nur während der Entwicklungszeit können Deeplearning-Algorithmen die Sicherheit von IT-System verbessern. In dem Paper *A Deep Lear-*

ning Approach for Network Intrusion Detection System [29] werden neuronale Netzwerke dazu verwendet, um ein Network Intrusion Detection System (NIDS) zu entwickeln. Diese Systeme nutzen die neuronalen Netze zur Laufzeit, um vermeintliche Angriffe auf ein Rechnernetzwerk zu erkennen. Nach Aussage der Autoren können neuronale Netzwerke hier einen enormen Mehrwert erzeugen, da sie auch auf Angriffsarten reagieren können, welche von Administratoren bei der Sicherung des Netzwerks eventuell nicht vorhergesehen wurden.

Smartphones bieten heutzutage die Möglichkeit viele verschiedene Dienste zu nutzen. Einige dieser Funktionen beinhalten auch die Transaktionen von sensiblen Daten, beispielsweise die persönlichen Login-Informationen für einen Online-Banking-Account. Laut der Veröffentlichung *Mobile Threat Report* [38] der Firma McAfee ist die Anzahl der Malware für mobile Endgeräte seit dem letzten Quartal 2015 bis zum dritten Quartal 2017 von ca. sieben Million auf fast das Dreifache gestiegen. In dem Paper *DL4MD: A Deep Learning Framework for Intelligent Malware Detection* [27] wird der Entwurf einer intelligenten Malware-Detection-Software für mobile Geräte beschrieben. Zunächst werden alle Aufrufe der Windows-API jeder PE[2]-Datei identifiziert. Somit können alle API-Aufrufe einem laufenden Programm eindeutig zugeordnet werden. Mit dieser Zuweisung können alle Aufrufe für eine unüberwachte Lernphase (genauer in Kapitel 3 beschrieben) eines neuronalen Netzwerks genutzt werden. Laut den Autoren ist eine Depplearning-Architektur für eine Maleware-Detection durchaus brauchbar und kann mit traditionellen Methoden verglichen werden.

Die vorgestellten Arbeiten befassen sich mit den Möglichkeiten des maschinellen Lernens und teilweise mit dem Gebiet der IT-Security. Allerdings behandeln sie die Sicherheit von Software zu einem anderen Zeitpunkt während einer Entwicklung (nach dem Deployment) oder behandeln einen anderen Bereich der Software-Security (Verhinderung von Störungen des laufenden Betriebs) als diese Thesis. Die vorliegende Arbeit soll einen Entwickler bereits während der Entwicklungsphase helfen Schwachstellen in einem IT-System zu vermeiden. Des Weiteren werden in dieser Thesis verschiedene Arten von neuronales Netzwerken bezüglich ihrer Tauglichkeit miteinander verglichen.

[2]Portable Executable

ANHANG A

Weitere Abbildungen

A.1

Klassendiagramm FeatureExtractor

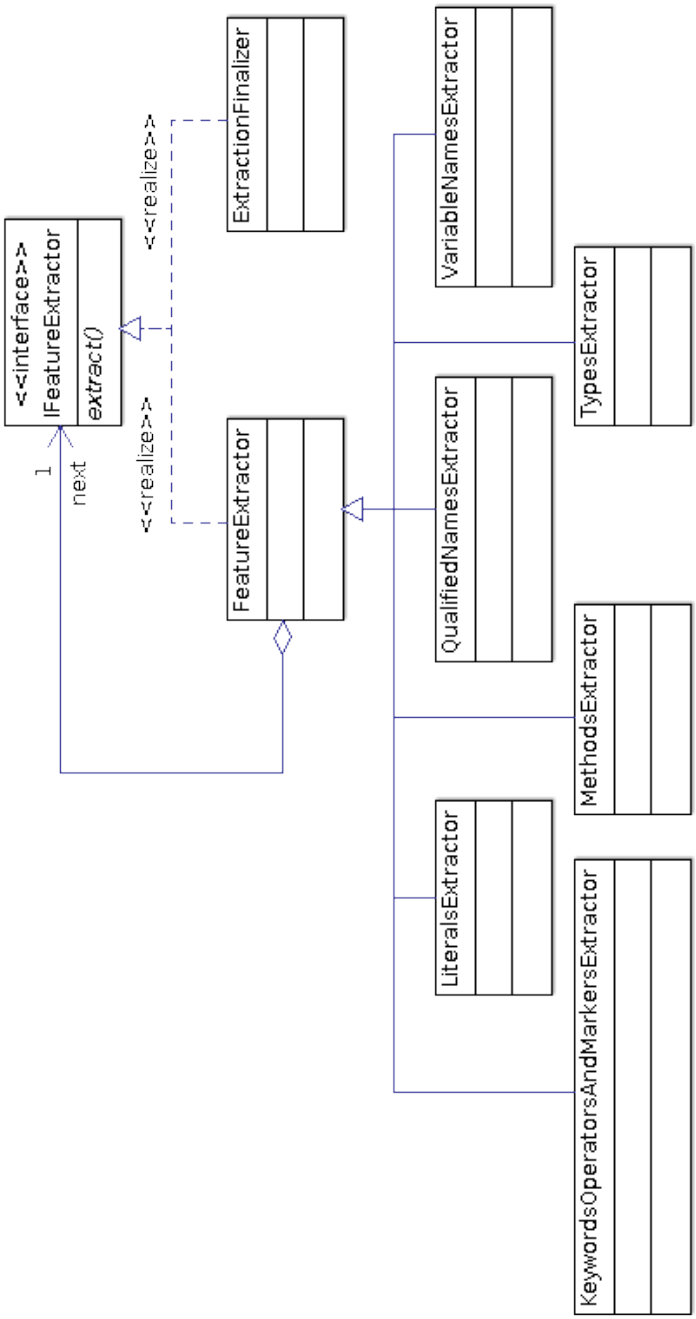


Abbildung A.1: Klassendiagramm: FeatureExtractor

ANHANG B

Weitere Listings

B.1

Deep Feed Forward

Listing B.1: Initialisierung DeepFeedForward

```
1  MultiLayerConfiguration conf = new NeuralNetConfiguration.↩
       Builder()
2          .seed(this.svlProperties.getDff_seed()).updater(
3              new Nesterovs(this.svlProperties.↩
                   getDff_learningRate(),
4                   this.svlProperties.getDff_momentum())).list()
5          .layer(0,
6              new DenseLayer.Builder().nIn(this.svlProperties.↩
                   numInputsAndIndexOfLabel).nOut(this.↩
                   svlProperties.getDff_numHiddenNodes())
7                   .weightInit(this.svlProperties.↩
                        getDff_weightInit()).activation(Activation↩
                        .TANH)
8                   .build())
9          .layer(1,
10             new DenseLayer.Builder().nIn(this.svlProperties.↩
                   getDff_numHiddenNodes())
11                  .nOut(this.svlProperties.↩
                        getDff_numHiddenNodes()).weightInit(this.↩
```

```
                    svlProperties.getDff_weightInit())
12                .activation(Activation.TANH).build())
13          .layer(2,
14              new OutputLayer.Builder(this.svlProperties.↩
                   getDff_lossFunction()).weightInit(this.↩
                   svlProperties.getDff_weightInit())
15                .activation(Activation.TANH).nIn(this.↩
                     svlProperties.getDff_numHiddenNodes())
16                .nOut(this.svlProperties.getNumClasses()).↩
                     build())
17          .pretrain(false).backprop(true).build();
18
19  this.model = new MultiLayerNetwork(conf);
20  this.model.init();
```

B.2

Long Short-Term Memory

Listing B.2: Initialisierung LongShortTermMemeory

```
1  NeuralNetConfiguration.Builder builder = new ↩
       NeuralNetConfiguration.Builder();
2       builder.seed(this.svlProperties.getLstm_seed()).↩
           weightInit(this.svlProperties.getLstm_weightInit())
3           .updater(new Nesterovs(this.svlProperties.↩
               getLstm_learningRate())));
4
5       ListBuilder listBuilder = builder.list();
6       listBuilder.layer(0, new LSTM.Builder().activation(↩
           Activation.TANH).nIn(1)
7           .nOut(this.svlProperties.getLstm_numHiddenNodes()).↩
               build());
8
9       for (int i = 1; i <= this.svlProperties.getLstm_numLayer↩
           (); i++) {
10        LSTM.Builder hiddenLayerBuilder = new LSTM.Builder();
```

```
11          hiddenLayerBuilder.nIn(this.svlProperties.↩
               getLstm_numHiddenNodes());
12          hiddenLayerBuilder.nOut(this.svlProperties.↩
               getLstm_numHiddenNodes());
13          hiddenLayerBuilder.activation(Activation.TANH);
14          listBuilder.layer(i, hiddenLayerBuilder.build());
15      }
16
17      listBuilder.layer(this.svlProperties.getLstm_numLayer()↩
           +1,
18           new RnnOutputLayer.Builder(this.svlProperties.↩
              getLstm_lossFunction()).activation(Activation.↩
              SOFTMAX)
19               .nIn(this.svlProperties.getLstm_numHiddenNodes())↩
                    .nOut(this.svlProperties.getNumClasses())
20               .build());
21      listBuilder.pretrain(false).backprop(true);
22
23  MultiLayerConfiguration conf = listBuilder.build();
24
25  this.model = new MultiLayerNetwork(conf);
26  this.model.init();
```

Quellenverzeichnis

Literaturquellen

[3] Rayner Alfred. „The rise of machine learning for big data analytics“. In: *Science in Information Technology (ICSITech), 2016 2nd International Conference on*. IEEE. 2016, S. 1–1.

[7] Russell Beale und Tom Jackson. *Neural Computing-an introduction*. CRC Press, 1990.

[8] Christopher M Bishop u. a. *Neural networks for pattern recognition*. Oxford university press, 1995.

[9] Wasja Brunotte. „Security Code Clone Detection entwickelt als Eclipse Plugin“. Master Thesis. Gottfried Wilhelm Leibniz Universität Hannover, 2018.

[10] Lawrence Chung u. a. *Non-functional requirements in software engineering*. Bd. 5. Springer Science & Business Media, 2012.

[14] Zhihua Cui u. a. „Detection of malicious code variants based on deep learning“. In: *IEEE Transactions on Industrial Informatics* 14.7 (2018), S. 3187–3196.

[19] Neil Davey u. a. „The development of a software clone detector“. In: *International Journal of Applied Software Technology* (1995).

[22] Premkumar T. Devanbu und Stuart Stubblebine. „Software Engineering for Security: a Roadmap“. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE ’00. New York, NY, USA: ACM, 2000, S. 227–239. ISBN: 1-58113-253-0.

[23] Claudia Eckert. *IT-Sicherheit - Konzepte - Verfahren - Protokolle*. Berlin: Walter de Gruyter, 2013. ISBN: 978-3-486-73587-1.

[25] Erich Gamma u. a. *Design Patterns - Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. Heidelberg: MITP-Verlags GmbH & Co. KG, 2015. ISBN: 978-3-826-69904-7.

[26] Daniel Halperin u. a. „Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses“. In: *2008 IEEE Symposium on Security and Privacy* (2008).

[27] William Hardy u. a. „DL 4 MD: A Deep Learning Framework for Intelligent Malware Detection“. In: *Proceedings of the International Conference on Data Mining (DMIN 16)*. Athens, GA, USA: CSREA Press, 2016, S. 61–67. ISBN: 1-60132-431-6.

[28] Jörg Hölzing. *Die Kano-Theorie der Kundenzufriedenheitsmessung - Eine theoretische und empirische Überprüfung*. Berlin Heidelberg New York: Springer-Verlag, 2008. ISBN: 978-3-834-99864-4.

[29] Ahmad Javaid u. a. „A Deep Learning Approach for Network Intrusion Detection System“. In: *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. Brüssel, Belgien: ICST, 2015, S. 21–26. ISBN: 978-1-63190-100-3.

[30] Stephen C Johnson u. a. *Yacc: Yet another compiler-compiler*. Bd. 32. Bell Laboratories Murray Hill, NJ, 1975.

[31] V Krishnapriya und K Ramar. „Exploring the difference between object oriented class inheritance and interfaces using coupling measures“. In: *Advances in Computer Engineering (ACE), 2010 International Conference on*. IEEE. 2010, S. 207–211.

[32] Alex Krizhevsky, Ilya Sutskever und Geoffrey E Hinton. „ImageNet Classification with Deep Convolutional Neural Networks“. In: *Advances in Neural Information Processing Systems 25*. Hrsg. von F. Pereira u. a. Curran Associates, Inc., 2012, S. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[35] Liuqiung Li u. a. „CCLearner: A Deep Learning-Based Clone Detection Approach“. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017).

[36] A Marciniak, J Korbicz und J Kuś. „Data pre-processing“. In: *Biocybernetics and Biomedical Engineering* 6 (2000), S. 62.

[37] Timothy Masters. *Practical neural network recipes in C++*. Morgan Kaufmann, 1993.

[38] McAfee. „Mobile Threat Report - The Next 10 Years“. In: (2018).

[39] Christian Müller. „Security Code Exporter für Github“. Bachelor Thesis. Gottfried Wilhelm Leibniz Universität Hannover, 2018.

[42] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[43] K Patidar, R Gupta und Gajendra Singh Chandel. „Coupling and cohesion measures in object oriented programming“. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 3.3 (2013).

[44] Tariq Rashid. *Make Your Own Neural Network - A Gentle Journey Through the Mathematics of Neural Networks, and Making Your Own Using the Python Computer Language*. CreateSpace Independent Publishing Platform, 2016. ISBN: 978-1-530-82660-5.

[46] Raul Rojas. *Neural Networks - A Systematic Introduction*. Berlin Heidelberg: Springer Science & Business Media, 2013. ISBN: 978-3-642-61068-4.

[47] Roman. „A taxonomy of current issues in requirements engineering“. In: *Computer* 18.4 (Apr. 1985), S. 14–23. ISSN: 0018-9162. DOI: 10.1109/MC.1985.1662861.

[48] Elmar Sauerwein. *Das Kano-Modell der Kundenzufriedenheit - Reliabilität und Validität einer Methode zur Klassifizierung von Produkteigenschaften*. Wiesbaden: Deutscher Universitätsverlag, 2000. ISBN: 978-3-824-47070-9.

[49] Jürgen Schmidhuber. „Deep learning in neural networks: An overview“. In: *Neural Networks* 61 (2015), S. 85–117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0893608014002135.

[50] Hinrich Schütze, Christopher D. Manning und Prabhakar Raghavan. *Introduction to Information Retrieval*. Cambridge: Cambridge University Press, 2008. ISBN: 978-0-521-86571-5.

[51] Mukesh Sharma und Shailendra Jha. „Digital Data Stealing from ATM using Data Skimmers: Challenge to the Forensic Examiner“. In: *Journal of Forensics Sciences And Criminal Investigation* 1.4 (2017).

[52] Abdullah Sheneamer und Jugal Kalita. „A survey of software clone detection techniques“. In: *International Journal of Computer Applications* 137.10 (2016), S. 1–21.

[53] Bundesamt für Sicherheit in der Informationstechnik (BSI). „Die Lage der IT-Sicherheit in Deutschland 2017“. In: (2017).

[54] Department of defense standard. „TCSEC: Department of defense trusted computer system evaluation criteria“. In: (1985).

[55] Gernot Starke. *Effektive Softwarearchitekturen - Ein praktischer Leitfaden*. M: Carl Hanser Verlag GmbH Co KG, 2017. ISBN: 978-3-446-45420-0.

[57] Madhusmita Swain u. a. „An Approach For Iris Plant Classification Using Neural Network“. In: *International Journal on Soft Computing (IJSC)* 3.1 (2012), S. 79–89.

[59] Christian Ullenboom. *Java ist auch eine Insel - Einführung, Ausbildung, Praxis*. Bonn: Galileo Press, 2014. ISBN: 978-3-836-22873-2.

[60] Zenon Waszczyszyn. *Advances of soft computing in engineering*. Bd. 512. Springer Science & Business Media, 2010.

[62] Martin White u. a. „Deep Learning Code Fragments for Code Clone Detection“. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016, S. 87–98. ISBN: 978-1-4503-3845-5.

[63] Fan Wu, Chung-han Chen und Dwayne Clarke. „Sensitive Data Protection on Mobile Devices“. In: *Editorial Preface* 5.9 (2014).

[64] Oliver Wyman. „Car Innovation 2015: A comprehensive study on innovation in the automotive industry“. In: (2015).

[65] Wenpeng Yin u. a. „Comparative Study of CNN and RNN for Natural Language Processing“. In: (2017).

[66] Chunting Zhou u. a. „A C-LSTM neural network for text classification“. In: *arXiv preprint arXiv:1511.08630* (2015).

Internetquellen

[1] *About CVE*. Aufgerufen am 14.09.2018. Jan. 2018. URL: http://cve.mitre.org/about/index.html.

[2] *About Eclipse Deeplearning4j | Deeplearning4j*. Aufgerufen am 08.10.2018. Sep. 2018. URL: https://deeplearning4j.org/about.

[4] *ANTLR*. Aufgerufen am 06.10.2018. Juli 2018. URL: http://www.antlr.org/.

[5] *ARFF (stable version) - Weka Wiki*. Aufgerufen am 08.10.2018. Aug. 2018. URL: https://waikato.github.io/weka-wiki/arff_stable/.

[6] *Backporpagation*. Aufgerufen am 03.10.2018. Oktober 2018. URL: https://brilliant.org/wiki/backpropagation/.

[11] *Common Vulnerability Scoring System SIG*. Aufgerufen am 19.09.2018. Feb. 2018. URL: https://www.first.org/cvss/.

[12] *Common Vulnerability Scoring System v3.0: Specification Document*. Aufgerufen am 19.09.2018. 2017. URL: https://www.first.org/cvss/cvss-v30-specification-v1.8.pdf.

[13] *Core Concepts in Deeplearning4j | Deeplearning4j*. Aufgerufen am 08.10.2018. Sep. 2018. URL: https://deeplearning4j.org/docs/latest/deeplearning4j-concepts.

[15] *CVE ID Syntax Change (Archived)*. Aufgerufen am 16.09.2018. Sep. 2018. URL: https://cve.mitre.org/cve/identifiers/syntaxchange.html.

[16] *CVE List Home*. Aufgerufen am 16.09.2018. Feb. 2018. URL: http://cve.mitre.org/cve/.

[17] *CVE Numbering Authorities*. Aufgerufen am 16.09.2018. Sep. 2018. URL: http://cve.mitre.org/cve/cna.html.

[18] *Cybersecurity Leaders Must Work Together to Take on Tomorrow's Threats*. Aufgerufen am 14.09.2018. Oktober 2017. URL: https://www.cyberthreatalliance.org/cybersecurity-leaders-must-work-together-take-tomorrows-threats/.

[20] *Deep Learning for Enterprise | Skymind*. Aufgerufen am 08.10.2018. Oktober 2018. URL: https://skymind.ai/.

[21] *Deeplearning4j*. Aufgerufen am 16.10.2018. Oktober 2018. URL: https://deeplearning4j.org/.

[24] *Frequently Asked Questions*. Aufgerufen am 14.09.2018. März 2018. URL: https://cwe.mitre.org/about/faq.html.

[33] *Künstliche Neuronale Netze - Aufbau & Funktionsweise*. Aufgerufen am 03.10.2018. Oktober 2018. URL: https://jaai.de/kuenstliche-neuronale-netze-aufbau-funktion-291/.

[34] *Lexikalische Analyse*. Aufgerufen am 06.10.2018. Oktober 2018. URL: https://www.tcs.ifi.lmu.de/lehre/ws-2011-12/compiler_alt/folien-02.pdf.

[40] *MultilayerPerceptron*. Aufgerufen am 08.10.2018. Sep. 2018. URL: http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/MultilayerPerceptron.html.

[41] *Neuronale Netze - Eine Einführung*. Aufgerufen am 02.10.2018. Mai 2018. URL: http://www.neuronalesnetz.de/downloads/neuronalesnetz_de.pdf.

[45] *Request CVE IDs*. Aufgerufen am 16.09.2018. Sep. 2018. URL: http://cve.mitre.org/cve/request_id.html.

[56] *Supervised and unsupervised learning*. Aufgerufen am 19.09.2018. Sep. 2018. URL: https://www.nnwj.de/supervised-unsupervised.html.

[58] *Terminology*. Aufgerufen am 14.09.2018. Dezember 2015. URL: https://cve.mitre.org/about/terminology.html.

[61] *Weka 3: Data Mining Software in Java*. Aufgerufen am 08.10.2018. Oktober 2018. URL: https://www.cs.waikato.ac.nz/ml/weka/index.html.