Bauhaus-Universität Weimar
Fakultät Medien
Studiengang Medieninformatik

Analyse und Vergleich von verschiedenen Vorverarbeitungsschritten und Architekturen von künstlichen neuronalen Netzen zur Codegenerierung

Bachelorarbeit

Anne Peter
geboren am 19.10.1995 in Gera

Matrikelnummer: 114421

1. Gutachter: Prof. Dr. Norbert Siegmund
2. Gutachter: PD Dr. Andreas Jakoby

Datum der Abgabe: 13.11.2017

Zusammenfassung

In dieser Arbeit wurden verschiedene künstliche neuronale Netze auf unterschiedlich vorverarbeiteten Datensätze darauf trainiert unfertigen Code zu ergänzen. Das Ziel ist, davon die beste Netzwerkarchitektur und Vorverarbeitung für diese Aufgabe zu finden und einen Ausblick auf die Fähigkeiten eines Netzwerkes zur automatischen Codegenerierung zu geben.

Dazu wurde ein Rohdatensatz von Java-Dateien aus 100 Github-Projekten erstellt und auf verschiedene Weisen vorverarbeitet: Kommentare entfernt; Einrückungen und Leerzeilen entfernt; die Kombination aus beiden Schritten sowie diese Kombination zusammen mit einer Bezeichnerumbenennung mit Hilfe des *JavaParsers*. Diese Eingriffe haben keinen Einfluss auf die Syntax oder Semantik des Quellcodes, aber reduzieren ihn auf das Wesentliche, wodurch die neuronalen Netze besser lernen sollten.

Sieben *LSTM-Netzwerke* mit unterschiedlichen Architekturen und jeweils zwei Optimierern wurden auf den fünf Datensätzen trainiert, um einen kurzen Quelltextausschnitts zeichenweise weiterzuschreiben. Die Qualität des generierten Quellcodes aller Kombinationen aus Netzwerkarchitekturen und Vorverarbeitungsschritten wurde anhand von vier Metriken untersucht. Die Metriken vergleichen den erzeugten Quelltext mit dem Originalcodeausschnitt bzw. mit dem entsprechenden Datensatz und umfassen die *Levenshtein-Distanz*, die *Difflib-ratio*-Funktion und die *Difflib-find_longest_match*-Funktion.

Die Auswirkungen der Vorverarbeitung und der Netzwerkarchitekturen auf die Codegenerierung und welche Kombination die besten Ergebnisse liefert, soll in dieser Arbeit betrachtet werden. Die hier gewonnenen Erkenntnisse sollen künftige Arbeiten zur automatischen Codegenerierung unterstützen.

In der Evaluierung hat sich gezeigt, dass sich bereits durch das Entfernen der Kommentare die Ergebnisse der Metriken sehr verbessert haben. Das Entfernen der Kommentare, Einrückungen und Leerzeilen erzielte noch bessere Ergebnisse. Es wurde nur teilweise durch das Umbenennen der Namen im Code übertroffen. Jedoch konnte das Entfernen der Einrückungen und Leerzeilen nicht als positiv gewertet werden. Die Netzwerke müssen im Einzelnen näher betrachtet werden. Allerdings scheinen kleinere Netzwerke bereits zum Erzeugen von teilweise syntaktisch korrekten Code zu genügen, wobei komplexere Strukturen bei zukünftigen Arbeiten zur semantisch korrekten Codegenerierung nötig sein können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

KNN	Künstliches neuronales Netz
RNN	Rekurrentes neuronales Netz
LSTM	Long short-term memory
API	Application programming interface (dt. Programmierschnittstelle)
FF	Forschungsfrage
IDE	Integrierte Entwicklungsumgebung
AST	Abstract syntax tree (dt. abstrakter Syntaxbaum)

Kapitel 1

Einführung

1.1 Motivation

Eine Vision der Computerwissenschaft ist es, syntaktisch und semantisch korrekten Code basierend auf gegebenen Anforderungen automatisch erzeugen zu können. So bräuchte man nur noch die Spezifikation seines Problems einzugeben und man würde sofort ein Programm zur Lösung erhalten, ohne dass weitere Entwicklungskosten anfallen.

Seit vielen Jahren versucht man Quelltext z.B. mit Hilfe von genetischer Programmierung automatisiert zu generieren und daraus sind Projekte wie *PushGP* [55], Finch [49] und Avida [42] entstanden.

Jedoch konnten bisher nur sehr eingeschränkte Programme erzeugt werden, die meist nur wenige Befehle beinhalteten. Das Hauptproblem dieser Ansätze ist die riesige Menge an kombinatorischen Möglichkeiten, die sich durch die vielen verschiedenen Befehle und deren Anordnung und Schachtelung ergeben.

Einen neuen erfolgreichen Ansatz versprechen *künstliche neuronale Netze (KNNs)*. Ein KNN ist ein Modell in der Informatik, das die Fähigkeit besitzt zu lernen. Dabei liest es Eingaben wie Bilder, Texte oder Vektoren ein und berechnet hierfür eine entsprechende Ausgabe in Abhängigkeit zum Einsatzszenario. Die Verarbeitung erfolgt durch Knoten, die Funktionen über ihre Eingangskanten darstellen. Die Ausgabe eines Knotens dient wiederum als Eingabe für einen anderen Knoten. Durch eine solche Verkettung von Knoten und die parallele Anordnung von Knoten entsteht eine Schichtenarchitektur, die die Eingabe verarbeitet. Generell kann man verschiedene Architekturen von KNNs unterscheiden. Hierbei sind sowohl die Arten der Knoten bzw. deren innere Funktionen unterschiedlich als auch deren Anordnung und Verknüpfung.

Die Forschung zur Generierung von Quellcode mit KNNs steht noch am Anfang, sodass das eingangs beschriebene Szenario der automatischen Ausgabe von funktionsfähigem Code noch weit in der Zukunft liegt.

Allerdings gibt es bereits Ansätze, die basierend auf einen eingegebenen Text neue Texte genieren. So wurden bereits neue Musikstücke basierend auf einer textuellen Beschreibung komponiert [38] und sprachlich variierende Antworten auf gleichbleibende Anfragen gegeben [67]. Diese Arbeiten benutzen eine bestimmte Architektur von KNNs: Long short-term memory. Diese Netzwerke sind in der Lage, Sequenzen von Text basierend auf bereits gegebenen Text zu genieren. Ein solches Szenario ist sehr ähnlich zur Generierung von Quellcode.

Da es, wie bereits erwähnt, zwar heute schon möglich ist mit neuronalen Netzen Text zu generieren, stellen sich die folgenden Forschungsfrage in Bezug auf Codegenerierung: Wie sollte Quelltext vorverarbeitet sein, damit ein KNN optimal darauf lernen kann? Welche Vorverarbeitungsschritte sollten geleistet werden? Welche Formatierungen oder Schreibkonventionen im Code bieten keinen Mehrwert für das KNN und können weggelassen werden? Welcher konkreter Netzwerkaufbau ist am besten für welchen Vorverarbeitungsschritt geeignet? Dies sind die zentralen Forschungsfragen, die in dieser Arbeit behandelt werden und somit einen wichtigen Grundstein in der weiteren Erforschung zur automatischen Generierung von Quellcode darstellen.

Es soll herausgefunden werden, wie der Eingabequelltext und das neuronale Netz geschaffen sein sollte, um möglichst syntaktisch korrekten Quellcode zu erzeugen. In späteren Arbeiten soll die Erzeugung eines syntaktisch und semantisch korrekten Codes auf Basis der hier gewonnenen Erkenntnisse angestrebt werden. Zur Beurteilung dessen werden die generierten Codeabschnitte der verschiedenen Vorverarbeitungsschritte in Verbindung mit den verschiedenen Netzwerkarchitekturen werden miteinander verglichen und evaluiert.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, mit Hilfe neuronaler Netze unfertigen Quellcode zu ergänzen und damit einen Ausblick auf die Fähigkeiten eines KNNs zu geben, Code selbstständig zu erzeugen. Zusätzlich sollen die Auswirkungen der verschiedenen Netzwerkarchitekturen und Vorverarbeitungsschritten auf den generierten Code bestimmt werden.

Der Quellcode zum Lernen des Netzwerkes wird am Anfang vereinfacht und verschieden komplexen Netzwerkarchitekturen als Trainingsdaten zur Verfügung gestellt. Es werden dabei sieben KNNs mit verschieden vielen LSTM-Schichten, Knoten pro Schicht und unterschiedlichen Optimierer betrachtet. Der als Trainingsdaten dienende Quelltext wurde mit Hilfe von vier Methoden vorverarbeitet. Die Netzwerke verarbeiten ihn zeichenweise und ergänzen nach dem Training einen kurzen Codeausschnitt. Der erzeugte Quelltext wird anhand von vier Metriken bewertet.

In der Evaluierung wird betrachtet, welche Kombinationen am besten zur Ergänzung von Quelltext geeignet sind. Es soll herausgefunden werden, welche Vorverarbeitungsschritte das Netzwerk schneller lernen lassen und welche besseren Code erzeugen. Diese Vorarbeit soll in späteren Arbeiten genutzt werden, um die optimale Repräsentation der Trainingsdaten zu ermitteln und damit den Lernprozess anderer KNNs zu beschleunigen. Außerdem soll festgestellt werden, ob die Komplexität und Konfiguration des Netzes die Genauigkeit der Voraussage des Quellcodes verbessert.

1.3 Überblick

In Kapitel 2 werden die Grundlagen zu KNNs und der `Python` [51] Bibliothek `Keras` [12] erklärt, die zur Implementierung der KNNs verwendet wird. In Kapitel 3 wird erläutert, wie der Quellcode für ein KNN verarbeitet werden kann, damit es besser lernt. Die zu untersuchenden unterschiedlichen Netzwerkarchitekturen werden in Kapitel 4 vorgestellt. Im 5. Kapitel erfolgt die Evaluierung der Abstraktionsarten von Code in Zusammenhang mit den verschiedenen Architekturen. Zu diesem Thema verwandte Arbeiten werden in Kapitel 6 gezeigt. Ein Ausblick auf zukünftige Arbeiten und die Zusammenfassung sind im 7. und letzten Kapitel zu finden.

Kapitel 2

Grundlagen

In den folgenden Abschnitten werden künstliche neuronale Netze und die `Python` [51] Bibliothek `Keras` [12] vorgestellt.

2.1 Künstliche Neuronale Netze

Das menschliche Gehirn besteht aus einem komplexen biologischen neuronalen Netz, dass durch ein KNN in einem Computer simuliert werden kann. Die biologischen Vorgänge wie die Aktivierung von Neuronen und chemische Veränderung von Synapsen werden als mathematische Formeln modelliert und somit vom Rechner ausführbar gemacht.

2.1.1 Historie

Die Zusammenfassung der Geschichte der neuronalen Netze stützt sich auf den Arbeiten von Juliane Pestel [50] und Christoph Obenhuber [46].

Die Anfänge von KNNs reichen bis in die frühen 1940er Jahre. Warren S. McCulloch und Walter Pitts beschreiben in ihrer Arbeit *A logical calculus of the ideas immanent in nervous activity* [40] von 1943 Verknüpfungen von Einheiten, die denen von natürlichen neuronalen Netzen gleichen, und jede arithmetische oder logische Funktion berechnen können. Vier Jahre später sprechen sie davon, dass diese Netze zur räumlichen Mustererkennung genutzt werden könnten. Donald O. Hebb entwickelt 1949 in *The organization of behavior: A neuropsychological approach* [24] die *Hebbsche Lernregel*, die den Grundstein für alle später folgenden Lernverfahren (ausgewählte siehe Abschnitt 2.1.7) legte. Der Neuropsychologe Karl Lashley veröffentlichte in seiner Arbeit *In Search of the Engram* von 1950 [37] die These, dass die Informationsspeicherung im Gehirn einer verteilten Repräsentation zu Grunde liegen muss. Dies stellte er anhand von Rattenversuchen fest.

Die Blütezeit begann mit dem von Frank Rosenblatt und Charles Wightman gebauten *Mark I Perceptron*, der erste erfolgreiche Neurocomputer. Er verfügte über einen 20 x 20

Pixel großen Bildsensor und war in der Lage einfache Ziffern zu erkennen. Damals hieß es, dass ein Perzeptron alles, was es repräsentieren kann, auch lernen kann [46]. Eine genaue mathematische Analyse des Perzeptrons von Marvin Minsky und Seymour Papert aus dem Jahr 1969 [44] zeigten jedoch dessen Grenzen auf. Viele Probleme sind mit einem Perzeptron nicht lösbar, so zum Beispiel das XOR-Problem. Daraufhin wendeten sich viele Wissenschaftler von KNNs ab, weil viele Forschungsgelder nach diesem Niederschlag gestrichen wurden. Diese Zeit wurde als *AI winter* (*artificial intelligence winter*), zu Deutsch etwa *Winter der künstlichen Intelligenz*, genannt.

Dennoch beschäftigten sich weiterhin einige Forscher mit neuronalen Netzen. 1972 stellte Teuvo Kohonen den linearen Assoziator, einen speziellen Assoziativspeicher, vor. Paul Werbos entwickelte 1974 das Backpropagation-Verfahren, das in Abschnitt 2.1.7 genauer behandelt wird. Teuvo Kohonen entwickelte 1982 die nach ihm benannten selbstorganisierenden Karten. 1982 beschreibt John Hopfield die Hopfield-Netze, mit denen er drei Jahre später das Travelling Salesman Problem lösen konnte [20]. Die Parallel-Distributed-Processing-Gruppe entwickelt 1985 das Backpropagation-Lernverfahren (siehe Abschnitt 2.1.7), um mehrschichtige Netze trainieren zu können.

Minskys Abschätzung wurde dadurch relativiert, denn mit mehrschichtigen Perzeptrons und Backpropagation sind nicht linear separierbare Probleme lösbar. Seitdem erhält das Fachgebiet der KNNs wesentlich mehr Aufmerksamkeit und die Zahl der daran arbeitenden Forscher ist rasant gestiegen. Seit dem *AI winter* und der Möglichkeit mehrere verschaltete Knoten zu benutzten, wird eher der Begriff *Deep Learning* und *Deep Neural Network* benutzt. Heute werden neuronale Netze in vielen Bereichen eingesetzt. Google nutzt zum Beispiel seit 2013 ein *Convolutional Neural Network* zur Fotoerkennung und -suche[11]. 2016 löste ein tiefes Long short-term memory (LSTM)-Netzwerk das alte Satzglied-basierte System zur automatischen Übersetzung [70] ab. Androids [41] und Apples Spracherkennung basiert ebenso auf *Deep Learning*. Wettervorhersagen können gleichermaßen von KNNs profitieren [53]. Amazon hat 2017 *Amazon Go* angekündigt [29], ein Lebensmittelgeschäft ohne Kassen und Schlangen. Man meldet sich über sein Handy mit seinem Amazon-Konto an, betritt den Laden, nimmt die gewünschten Produkte aus dem Regal und verlässt am Ende die Filiale. Das Amazon-Konto wird kurz darauf belastet. Alles wird über Verknüpfungen von Sensoren, der sogenannten *Sensorfusion*, *Computer Vision* und *Deep Laerning Algorithmen* realisiert. Der Prototyp befindet sich in Seattle, Washington und ist zur Zeit den Amazon-Angestellten im Beta-Programm vorbehalten [2].

2.1.2 Neuron

KNNs bestehen wie ihr menschliches Vorbild aus Neuronen. Diese werden auch Units (engl. *unit* für Einheit) oder Knoten genannt. Ein Unit nimmt Informationen auf (input), verarbeitet sie mit Hilfe von mathematischen Funktionen und gibt sie dann an andere Units weiter (output). Das Weiterleiten von Informationen wird auch Feuern genannt.

Es gibt vier verschiedene Arten von Units: Input-Units, Hidden-Units, Output-Units und Bias-Units. Gleichartige, übereinander angeordnete Units bilden zusammen eine Schicht (engl. *layer*) siehe Abschnitt 2.1.3.

Input-Units. Input-Units erhalten ihre Informationen von der Außenwelt und sind die Anlaufstelle für Signale aus der Umwelt. Sie leiten ihre modifizierten Eingaben an Hidden-Units weiter.

Hidden-Units. Diese verarbeiten ihre empfangenen Daten und übertragen sie an gegebenenfalls vorhandene weitere Hidden-Units oder an Output-Units. Gibt es keine Hidden-Units, werden die veränderten Informationen direkt an die Output-Units übertragen.

Output-Units. Die Output-Units sind die Verbindungen zur Außenwelt und leiten ihre Ergebnisse außerhalb des KNNs.

Bias-Units. Bias-Units (engl. *bias* für Tendenz, Vorurteil) nehmen eine besondere Stellung ein, da sie keine Daten von anderen Units oder von außen erfahren. Sie geben von alleine ihre Informationen an Units weiter und beeinflussen sie ungefragt. Ihre Gewichte zu verbundenen Units können ebenfalls positiv oder negativ sein. Man setzt Bias-Units ein, wenn man für das adressierte normale Unit eine Schwelle benötigt, die andere Units erst über- bzw. unterschreiten (bei einer negativen bzw. positiven Gewichtung) sollen. So neigen betroffene Units eher dazu weniger bzw. mehr als normalerweise zu feuern. Diese Schwelle ist jedoch von der einer Aktivierungsfunktion zu unterscheiden, da diese durch ein Gewicht definiert ist und durch Lernen angepasst werden kann.

2.1.3 Schichten

Alle Input-Units bilden zusammen die Eingabeschicht (engl. *input layer*) und alle Output-Units bilden die Ausgabeschicht (engl. *output layer*). Es ist üblich pro KNN nur jeweils eine Eingabe- und eine Ausgabeschicht anzulegen. Jedoch kann es mehrere verdeckte Schichten (engl. *hidden layer*) geben. Diese sind von Außen nicht einsehbar. Ein beispielhaftes neuronales Netz kann in Abbildung 2.1 betrachtet werden.

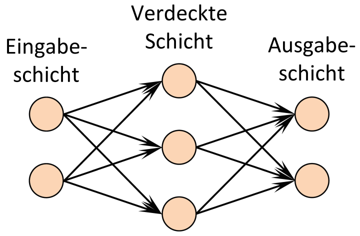

Abbildung 2.1: Ein neuronales Netz

2.1.4 Verbindungen

Units sind untereinander durch gerichtete Kanten verbunden, um Daten auszutauschen. Eine Verbindung leitet die Ausgabe eines Neurons i zur Eingabe eines Neurons j. Wenn Neuron i der Vorgänger von Neuron j ist und Neuron j der Nachfolger von Neuron i, kann jede Verbindung mit einem Gewicht w_{ij} versehen werden.

Gewichte

Wie stark die Verbindung zwischen zwei Units ist, wird durch dessen Gewichtung ausgedrückt. Je intensiver die Verbindung ist, desto höher ist das Gewicht.

Ein positives Gewicht signalisiert, dass ein Neuron auf ein anderes Neuron einen erregenden Einfluss ausübt. Ein negatives Gewicht bedeutet, dass es sich um einen hemmenden Einfluss handelt. Ein Gewicht von Null besagt, dass ein Neuron auf ein anderes Neuron keinen Einfluss ausübt.

Diese Werte werden in Gewichtsmatrizen oder Gewichtsvektoren auf Abruf gespeichert. In den Verbindungsgewichten ist das Wissen des KNNs gespeichert [47]. Die Gewichte können durch Lernregeln verändert werden.

Aktivierungsfunktion

Die Aktivierungsfunktion wird auch als Aktivitätsfunktion oder Transferfunktion bezeichnet. Als Aktivierungsfunktion wird auf die Summe der gewichteten Eingabewerte, die sogenannte Netzeingabe angewendet, um die Ausgabe, das Aktivitätslevel des Neurons zu erhalten. Ist die Eingabe groß genug, ist das Unit aktiviert und feuert. Im einfachsten Fall ist es eine lineare Funktion, andernfalls eine stückweise lineare, eine Sprungfunktion oder nichtlineare. Letztere wird besonders oft benutzt. Eine lineare Funktion bietet wenig Flexibilität und ist nur für einfache KNNs geeignet. Eine stückweise lineare Funktion ist in ihrem Definitionsbereich beschränkt. Eine Sprungfunktion, auch Schwellenwertfunktion genannt, gibt für negative Eingabewerte den Wert null und für positive den Wert eins zurück. An der Stelle x = 0 kann die Funktion nicht definiert sein oder sie gibt einen konkreten Wert, null oder eins, aus.

Eine nichtlineare Funktion, im Zusammengang mit neuronalen Netzen auch oft sigmoide Funktion genannt, weist eine hohe Flexibilität auf und wird häufig in KNNs genutzt. Da die Funktion an allen Stellen differenzierbar ist, kann man diese Eigenschaft für spätere Verfahren wie Backpropagation zur Fehlerminimierung nutzen. Im Gegensatz zu linearen Aktivitätsfunktionen ist das Aktivitätslevel hier sowohl nach oben als auch nach unten begrenzt.

2.1.5 Netztypen

KNNs kennzeichnen sich durch ihren Aufbau. Verschiedene Strukturen sind für verschiedene Probleme/Aufgabenstellungen besser oder auch schlechter geeignet. Man muss selbst herausfinden, welche Zusammenstellung von Knoten und Kanten am Besten für die Aufgabenstellung ist. Eine Eingabe- und Ausgabeschicht besitzt jedes Netz, aber die Anzahl der hidden layer und der Kanten sowie die Vernetzung unter den Schichten variiert. Hier werden einige Netztypen vorgestellt.

FeedForward-Netz

FeedForward-Netze bestehen aus Schichten und Verbindungen zur jeweils nächsten Schicht. Jedes Neuron beeinflusst maximal ein Neuron der nachfolgenden Schicht. Es darf keine Verbindung zu einer vorangegangenen Schicht existieren. Ein Netz ist vollverknüpft, wenn alle Neuronen einer Schicht mit jeden Neuron der nächsten Schicht verbunden sind.

Einschichtiges FeedForward-Netz. Ein einschichtiges FeedForward-Netz verarbeitet seine Informationen nur in der Ausgabeschicht und kommt ohne eine verborgene Schicht aus. Durch diese Einschränkung ist es nur für sehr einfache Modelle geeignet. Ein Beispiel dazu ist in Abbildung 2.2 zu sehen.

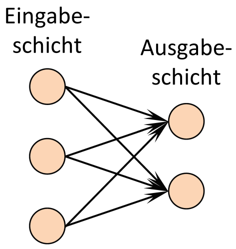

Abbildung 2.2: Ein einschichtiges FeedForward-Netz

Mehrschichtiges FeedForward-Netz. Mehrschichtige FeedForward-Netze verfügen zusätzlich über mindestens eine verdeckte Schicht, die mehr Abstraktion ermöglicht[69]. Mit dem zweischichtigen KNN in Abbildung 2.3 lässt sich beispielsweise schon das XOR-Problem lösen [5].

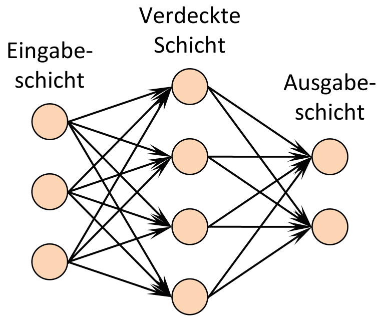

Abbildung 2.3: Ein mehrschichtiges FeedForward-Netz

Rekurrentes Netz

Rekurrente neuronale Netze (RNNs) zeichnen sich durch Rückverbindungen eines Neurons zu sich selbst, zu Neuronen der gleichen Schicht oder der vorherigen Schicht aus. Somit speichert man Erkenntnisse aus dem aktuellen Schritt für den darauf folgenden und ermöglicht das Speichern von zeitlich kodierten Informationen. Auf diese Weise wird dem KNN ein „Gedächtnis“ gegeben, um Aussagen auf Basis von einer vorangegangenen Sequenz von Eingaben zu treffen. In Abbildung 2.4 ist ein beispielhaftes RNN zu sehen.

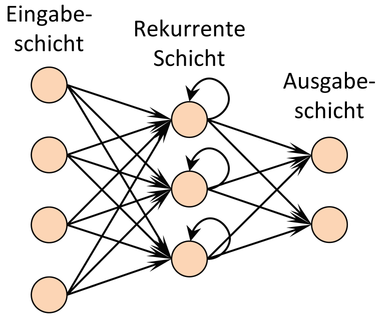

Abbildung 2.4: Ein rekurrentes Netz

Long short-term memory Netze. LSTM Netze sind eine Untergruppe der rekurrenten Netze. Sie wurden 1997 von Sepp Hochreiter und Jürgen Schmidhuber eingeführt [27]. Längere zeitliche Verzögerungseffekte z.B. für Klassifizierungsaufgaben können durch LSTM Netzwerke einbezogen werden und das Training verbessern.

Sie sind in der Lage Langzeit-Abhängigkeiten in Sequenzen zu betrachten und sich Informationen über einen langen Zeitraum zu merken. Sie besitzen Speicherzellen, die Informationen speichern und auslesen können, und sogenannte *Gates*, die geöffnet oder geschlossen werden können. Letzteres beeinflusst, was eine Zelle speichert und wie ihr Inhalt ausgelesen, beschrieben oder gelöscht werden darf [35]. Normalerweise wird zum Trainieren der Backpropagation-Algorithmus benutzt, der im Abschnitt 2.1.7 näher erläutert wird.

Damit die *Gates* für den Backpropagation-Algorithmus differenzierbar bleiben, sind sie analog und durch eine Sigmoid-Funktion über den Bereich von 0 bis 1 definiert. Sie verfügen ebenfalls über Gewichte, die beim Lernen verändert werden. Dadurch passt sich das Verhalten der Speicherzellen an, also wann sie Daten lesen, auslesen oder löschen.

2.1.6 Trainings- und Testphase

In der Trainingsphase soll das Netzwerk anhand von Lernmaterial die gewünschten Ausgaben liefern. Es passt seine Verbindungsgewichte mit Hilfe von Lernregeln entsprechend der gegebenen Zielfunktion an. Die Lernregeln geben an, auf welche Art und Weise die Gewichte modifiziert werden, um die Zielfunktion näher zu approximieren. Das in dieser Phase erworbene Wissen wird in der Testphase am geänderten Lernmaterial oder an neuen Daten überprüft. Hier werden keine Gewichte mehr korrigiert, sondern nur die Ergebnisse des KNNs und dessen Fehlerquote betrachtet. Gegebenenfalls muss danach die Struktur des Netzes, z.B. die Anzahl der verborgenen Schichten oder die Aktivierungsfunktion überdacht werden.

Durch dieses eigenständige Lernen und Verändern der eigenen Gewichte mit jedem Durchlauf kann ein Netz nicht nur das bekannte Problem aus der Trainingsphase lösen, sondern auch weitere mit ähnlicher Beschaffenheit ohne explizit dafür programmiert zu sein [47]. Voraussetzung dafür ist, dass die Anzahl der verdeckten Schichten und Verbindungen drauf abgestimmt sind, dass sich das KNN in der ersten Phase nicht an die Daten "überanpasst", aber auch nicht zu allgemein gehalten wird.

2.1.7 Lernverfahren

Lernregeln, auch Lernverfahren genannt, geben vor, wie in der Trainingsphase die Gewichte verändert werden sollen. Das Verfahren des überwachten Lernens wird hier erläutert, da dieses Verfahren in der späteren Implementierung angewandt wird. Generell werden beim Lernen die Gewichte des Netzwerks zufällig initialisiert und dann nach und nach verbessert.

Überwachtes Lernen

Das Ziel beim überwachten Lernen (engl. *supervised learning*) ist es, eine Funktion zu finden, die am besten eine Auswahl an Eingangsdaten auf die richtigen Ausgangswerte abbildet. Dabei steht ein vorgegebener Lernvektor zur Verfügung, der die gewünschte Ausgabe des Netzes beschreibt. Mit diesem wird die tatsächliche Ausgabe des Netzes verglichen. Die Gewichte werden so angepasst, dass die Abweichung zwischen Vorgabe und Ergebnis möglichst klein gehalten wird. Um den Fehler anzugeben, wird oft die mittlere quadratische Abweichung benutzt. Die mittlere quadratische Abweichung berechnet sich aus der Differenz zwischen den einzelnen Werten und dem arithmetischem Mittel der Daten. Diese Differenz wird quadriert und durch n (Stichprobenumfang) dividiert [64]:

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Um eine neue, bessere Kombination von Werten für die Gewichte zu finden, die den Fehler weiter minimieren, kann man den Backpropagation-Algorithmus verwenden.

Backpropagation. Bei Netzwerken mit mehreren Schichten wird man vor das Problem gestellt, dass man keinen direkten Fehler für Units der verdeckten Schicht bzw. Schichten bestimmen kann [62]. Die Backpropagation-Regel beseitigt diese Komplikation. Sie propagiert den Fehler zurück bis zur Eingabeschicht [62], benötigt dafür jedoch eine differenzierbare Aktivierungsfunktion, um die Ableitung davon zu bestimmen.

2.1.8 Fehlerminimierung mit Backpropagation

Beim Lernen wird die berechnete Ausgabe des Netzes mit dem erwarteten Ergebnis verglichen und jede Abweichung bestraft. Die Zielfunktion hängt von den internen Parametern (Gewichte und Bias) ab, die den Ausgangswert y zur Eingabe x berechnen. Um die Abweichung zu minimieren, gibt es verschiedene Ansätze des Gradientenverfahren wie z.B. Backpropagation. Dies ist Spezialfall des allgemeinen Gradientenverfahrens und basiert auf der Minimierung des mittleren quadratischen Fehlers [68]. Es ist ein effizienter Weg die Gradienten einer Zielfunktion in einem mehrschichtigen Netzwerk zu berechnen. In dieser Arbeit werden jedoch nur die optimierten Versionen `Adam` [34] und `RMSprop` [61] benutzt. Die folgenden Erklärungen orientieren sich an dem Tutorial von Prof. Dr. Guenter Gauglitz und Clemens Jürgens [18].

Alle Gradientenverfahren berechnen den Anstieg einer Zielfunktion, hier der Fehlerfunktion $E(W)$, und folgen ihm nach unten, bis ein lokales Minimum oder im besten Falle das globale Minimum erreicht ist. Es wird versucht den Fehler zu minimieren, indem eine Änderung aller Gewichte δw um einen Bruchteil des negativen Gradienten der Fehlerfunktion vorgenommen wird [17].

$$\Delta W = -\eta \nabla E(W)$$

Die Änderung ΔW des Gewichtsvektors ist proportional zum negativen Gradienten $-\nabla E(W)$ der Fehlerfunktion mit dem Faktor η, auch als Lernfaktor oder Schrittweite bezeichnet. Den Lernfaktor kann man selbst beliebig wählen. Es werden jedoch Werte zwischen 0,01 und 0,5 empfohlen [62]. Wenn die Lernrate zu klein gewählt wird, bewegt man sich zu langsam oder gar nicht aus Plateaus der Fehlerfläche heraus. Generell benötigt man durch eine kleine Rate mehr Trainingszeit. Wenn die Lernrate jedoch zu groß gewählt wird, kann man in Schluchten hin und her springen oder im besten Fall über sie hinweg springen. Für ein einzelnes Gewicht gilt somit:

$$\Delta w_{ij} = \nabla \frac{\delta}{\delta w_{ij}} E(W)$$

Als Fehlerfunktion wird bei dem Backpropagationverfahren der quadratische Abstand zwischen erwarteter und realer Ausgabe verwendet. Der Gesamtfehler E ergibt sich als Summe der Fehler über alle Parameter (Gewichte) p:

$$E = \sum^{p} E_p \text{ mit } E_p = \frac{1}{2} \sum^{j} (t_{pj} - o_{pj})$$

Dabei ist E_p der Fehler für ein Muster (pattern) p, t_{pj} die Lerneingabe, o_{pj} die Ausgabe von Neuron j bei Muster p. Der Faktor $\frac{1}{2}$ wurde verwendet, damit er sich später gegen eine 2 wegkürzt, die durch das Differenzieren entsteht. Zur Bestimmung optimaler Gewichte spielt es keine Rolle, ob man den Fehler oder den halben Fehler minimiert [17]. Ebenso ist es unerheblich, dass hiermit das Quadrat des Fehlers minimiert wird anstelle des Fehlers selbst, der als Quadratwurzel der obigen Summe definiert ist.

Gradientenverfahren haben alle das Problem, dass sie in einem lokalen Minimum der Fehlerfläche stecken bleiben können. Die Schwierigkeit bei KNNs ist, dass die Fehlerfläche mit wachsender Dimension des Netzes (Anzahl an Verbindungen) immer stärker zerklüftet wird und es dadurch wahrscheinlicher wird in einem lokalen statt dem globalen Minimum zu landen [19]. Plateaus in der Fehlerfläche sind ein weiteres Problem von Gradientenverfahren. Da die Größe der Gewichtsänderung von dem Betrag des Gradienten abhängig ist, stagniert Backpropagation auf Plateaus, d.h. das Lernverfahren braucht extrem viele Iterationsschritte um sich von dort zu entfernen.

Das Lernverfahren führt bei einem vollständig ebenen Plateau überhaupt keine Gewichtsänderung mehr durch, weil der Anstieg null ist. Problematisch ist in diesem Fall außerdem, dass man normalerweise nicht erkennen kann, ob das Lernverfahren auf einem Plateau stecken geblieben ist oder sich in einem lokalen oder globalen Minimum befindet, bei dem der Gradient ebenfalls null ist. Für diese Problem existieren jedoch Verfahren, modifizierte Varianten von Backpropagation wie `Adam` oder `RMSprop`, die diese Plateaus überwinden können. Ähnelt die Fehlerfläche einer Schlucht, kann das Lernverfahren oszillieren. Dies geschieht, wenn durch die Gewichtsänderung ein Sprung auf die gegenüberliegende Seite erfolgt. Besitzt der Gradient dort einen genauso großen Betrag, aber die entgegengesetzte Richtung, bewirkt dies einen Sprung zurück auf die erste Seite. Glücklicherweise können `Adam` oder `RMSprop` auch die Oszillationen durch große Gradienten dämpfen oder gar eliminieren.

Das klassische Gradientenverfahren minimiert die Fehlerfunktion, indem deren Anstieg in Bezug zu den Parametern des Netzwerkes berücksichtigt wird. Anhand dessen werden die Parameter so verändert, dass sie die Funktion minimieren. Man berechnet den Gradienten der Fehlerfunktion in Bezug auf die Gewichte des KNNs $E(W)$ und passt sie in die entgegengesetzte Richtung des Gradienten der Fehlerfunktion in Bezug auf die Netzwerkgewichte an [65]. Der Anstieg wird hinsichtlich eines kleines Datensatzes (engl. *batch*) berechnet und die Gewichte werden nur einmal angepasst.

Das stochastische Gradientenverfahren (engl. *stochastic gradient descent*) berechnet im Gegensatz zum klassischen Gradientenverfahren den Anstieg in jedem Schritt für jedes Trainingsbeispiel und nicht nur einmal über den gesamten Datensatz. Die Gewichtsupdates erfolgen in jedem Schritt. Das stochastische Gradientenverfahren ist normalerweise schneller als das klassische [65].

Zum anderen gibt es das Mini Batch Gradientenverfahren (engl. *mini batch gradient descent*), das beide vorherigen Methoden vereint. Es passt die Gewichte für jeden *batch* mit je n Trainingsbeispielen an. `RMSprop` und `Adam` [34] sind Beispiele für das Mini Batch Gradientenverfahren und werden später speziell für `Keras` in Abschnitt 2.2.3 erklärt.

2.2 Keras

`Keras` [12] ist eine in `Python` [51] geschriebene Open-Source Bibliothek für *Deep Learning*. Sie kann entweder auf den *Machine Learning* Bibliotheken `TensorFlow` [39], `CNTK` [43] oder `Theano` [60] aufsetzen. In dieser Arbeit wird `Keras` zusammen mit `TensorFlow` verwendet.

2.2.1 Minimalbeispiel

Um `Keras` zusammen mit `TensorFlow` benutzen zu können, muss vorher `TensorFlow` installiert werden (Anleitung siehe [58]) und danach `Keras` mit dem folgenden Befehl:

```
pip install keras
```

`Keras` wurde entwickelt, um eine höhere Abstraktionsschicht bei der Erstellung von KNNs zu haben, um so schneller Netzwerke erstellen zu können. Man kann sich rasch ein Netzwerk aus vorgefertigten Bausteinen zusammenstellen. Im Folgenden wird ein Beispiel [12] zur Erstellung eines einfachen LSTM Netzes gezeigt: Das Hauptelement ist das `model`. Es enthält die Struktur des KNNs und ist die Schnittstelle, um es zu trainieren oder gelernte Vorhersagen treffen zu lassen. Der einfachste Typ `model` ist das `Sequential model`. Es ist ein linearer Stapel aus Schichten. Für komplizierte Netzwerke gibt es die funktionale Programmierschnittstelle[33]. Hier wird das `Sequential model` genutzt:

```
from keras.models import Sequential
model = Sequential()
```

Neue Schichten und deren Eigenschaften ergänzt man mit *add()*:

```
from keras.layers import Dense, Activation, LSTM
model.add(LSTM(128, input_shape=(40, 100)))
model.add(Dense(100))
model.add(Activation("softmax"))
```

Die Schichten werden in der Reihenfolge miteinander verknüpft, in der aufgerufen werden. Zuerst erstellt man die Eingabeschicht. In dieser wird die Größe der Eingabe mit `input_shape` angeben. Außerdem wird jeder Schicht als erstes Argument die Größe seiner Ausgabe übergeben. Dann folgt die letzte Schicht, in diesem Fall eine `Dense`-Schicht. In dieser ist jedes Neuron mit jedem Neuron der nächsten Schicht verbunden. Hier gibt das Netzwerk 100 Ergebnisse aus. Diese werden als Klassen bzw. in dieser Arbeit als Zeichen interpretiert. Die `softmax`-Aktivierung wird generell bei der Klassifizierung mehrerer Klassen eingesetzt. Sie wandelt das Ausgangssignal eines jeden Units in eine Wahrscheinlichkeit um. Das Beispielnetzwerk könnte also nach dem Training die Wahrscheinlichkeiten für 100 Zeichen berechnen und z.B. das mit der höchsten Wahrscheinlichkeit als Zielzeichen ausgeben.

Den Lernprozess konfiguriert man mit *compile()*

```
model.compile(loss="categorical_crossentropy", optimizer="adam")
```

Man gibt dabei die Ziel- bzw. Fehlerfunktion (`loss`) an und den Optimierer. Die `Keras` Optimierer `Adam` und `RMSprop` werden in Abschnitt 2.2.3 genauer erläutert.

Der Lernprozess wird mit *fit* gestartet. Die Trainingsdaten werden in Teilstücken, sogenannten `batches`, verarbeitet. Insgesamt werden hier 5 Epochen trainiert. `Batches` und Epochen werden in Abschnitt 2.2.4 erklärt.

```
model.fit(x_train, y_train, epochs=5, batch_size=128)
```

Danach kann das `model` auf Basis neuer Eingaben Voraussagen treffen:

```
prediction = model.predict(x_test, batch_size=128)
```

2.2.2 Zielfunktion

In `Keras` wird die Zielfunktion für den Lernprozess `loss function` genannt. Diese gibt den Netz an, wie weit das berechnete Ergebnis von dem tatsächlichen Ergebnis entfernt ist. Dieser Fehler wird an das Netzwerk zurückgegeben. Umso mehr die Ergebnisse von der erwarteten Lösung entfernt sind, desto mehr wird das neuronale Netzwerk bestraft und desto intensiver wird es seine Gewichte ändern. Diese Funktion gilt es zu minimieren.

In dieser Arbeit wird die `categorical crossentropy` Funktion angewendet, sodass die Voraussagen des KNNs als Wahrscheinlichkeiten interpretiert werden können. Aufgrund dieser Wahrscheinlichkeiten werden später die zu voraussagenden Zeichen gezogen.

2.2.3 Optimierer

Die Abweichung der Ausgabe des KNNs vom erwarteten Ergebnis wird durch die Fehlerfunktion E beschrieben. Sie beschreibt die Qualität der Voraussagen des Netzwerkes. Um diese Abweichung zu minimieren, müssen die internen Parameter, wie Gewichte, des Netzes mit Hilfe eines Optimierers verändert werden. Dazu gibt es verschiedene Ansätze des sogenannten Gradientenverfahrens, die in Abschnitt 2.1.8 vorgestellt wurden.

In dieser Arbeit werden die beiden Optimierer (`optimizer`) `Adam` und `RMSprop` in Verbindung mit allen Netzwerkarchitekturen und Quelltextvorverarbeitungsschritten betrachtet.

Das klassische Gradientenverfahren besitzt eine einzige globale Lernrate α für alle Gewichtsupdates, die sich nicht während des Trainings verändert [7]. Doch in einem mehrschichtigen KNN können die geeigneten Lernraten zwischen den Verbindungen sehr schwanken. Um dieses Problem anzugehen, kann z.B. `RMSprop` genutzt werden. `RMSprop` steht für *Root Mean Square Propagation* und besitzt keine globale, skalare Lernrate, sondern einen adaptiven Lernratenvektor, der jedes Gewicht einzeln betrachtet. So kann man eine globale Lernrate festlegen, die jedoch mit einem passenden lokalem Zuwachs, der empirisch für jedes Gewicht festgestellt wird, multipliziert wird. [61]

$$RMS(w,t) = \gamma RMS(w,t-1) + (1-\gamma)\left(\frac{\delta E}{\delta w}(t)\right)^2 \text{ [25]}$$

RMS steht für *Root Mean Square*, was im Deutschen *Wurzel des mittleren quadratische Fehlers* bedeutet. γ ist der Verminderungsfaktor und t indiziert die aktuelle Trainingsiteration.

Es wird ein γ von 0.9 und eine Lernrate η von 0.001 empfohlen [25]. `RMSprop` gleicht damit die Lernrate bzw. Schrittweite der Veränderung an. Die Schritte für kleine Gradienten werden vergrößert, um das starke Verringern des Gradienten (engl. *vanishing gradient problem* [28]) zu vermeiden. Die Schrittweite für große Gradienten werden hingegen verkleinert, um das starke Vergrößern des Gradienten (engl. *exploding gradient problem*) zu verhindern.

`Adam` [34] steht für `Adaptive Moment Estimation` und ist ein Update zum `RMSprop` Optimierer. `Adam` berechnet eine adaptive Lernrate für jeden Parameter des Netzes. Zusätzlich zur Berücksichtigung des exponentiell verfallenden Durchschnitts der quadrierten vergangenen Gradienten behält `Adam` außerdem einen exponentiell abfallenden Durchschnitt der letzten Gradienten. In der Praxis funktioniert `Adam` ausgezeichnet und schneidet gut gegenüber anderen Lernalgorithmen ab, da es schnell konvergiert und so neuronale Netze schnell trainieren kann.

2.2.4 Epochen, Iterationen und batches

Die Epochen in `Keras` geben an, wie oft das KNN auf dem Trainingsdatenset trainiert werden soll. Je mehr Epochen eingestellt werden, desto länger kann das Netzwerk Muster lernen und später bessere Voraussagen treffen. In jeder Epoche wird das gesamte Datenset gelernt. Da dies aber meist zu groß ist, um auf einmal einzulesen und an die Eingabeschicht zu geben, wird es `batch`-weise weitergeben. Wie viele `batches` pro Epoche eingelesen müssen, wird als Iterationsanzahl angegeben.

Bei einem Datensatz von 1280 Daten und einer `batch-size` von 128 werden 10 Iterationen benötigt, um eine Epoche vollständig zu trainieren.

2.2.5 Dropout

`Dropout` ist eine etablierte Technik, die KNNs reguliert und eine Überanpassung des Netzwerkes verhindert. Sie wurde 2014 von Srivastava, et al. [56] vorgestellt. `Dropout` ignoriert zufällig ausgewählte Knoten während des Trainings, wodurch deren Gewichte nicht verändert werden und keinen Einfluss auf das Lernen haben.

Während ein neuronales Netz lernt, nehmen Gewichte ihren Platz im Kontext innerhalb des Netzes ein. Die Gewichte von Neuronen sind auf bestimmte Merkmale in den Trainingsdaten abgestimmt, was eine gewisse Spezialisierung bereitstellt, auf die benachbarte Knoten aufbauen können. Wenn zwei oder mehr Knoten wiederholt das gleiche Muster erkennen, anstatt unabhängig voneinander zu agieren, spricht man von *co-adapting*. Falls dies zu oft vorkommt, kann das zu einem Netzwerk führen, dass sich zu sehr auf die Trainingsdaten spezialisiert hat [6].

Wenn nun aber einige Neuronen im Training wegfallen (engl. *drop out*), werden andere Knoten für sie einspringen müssen, um ihren Verlust zu kompensieren und deren Musterkennung nachzuahmen [6]. Dadurch wird ihre Unabhängigkeit untereinander forciert, was zu einer besseren Generalisierung des Gelernten führt und Überanpassung vermeidet.

Ein KNN, das `Dropout` benutzt, braucht jedoch typischerweise 2-3 mal länger zum Trainieren als ein normales Netz der gleichen Architektur [56]. Dies liegt hauptsächlich daran, dass bei jedem Training eine zufällige Netzwerkarchitektur trainiert wird.

In `Keras` kann man nach jeder Neuronenschicht eine `Dropout`-Schicht einfügen. Die Gleitkommazahl, die übergeben werden muss, gibt an, zu wie viel Prozent Neuronen während des Trainings ignoriert werden. Dabei stehen 0.2 für 20 Prozent.

```python
from keras.models import Sequential
from keras.layers import LSTM, Dropout

model = Sequential()
model.add(LSTM(128, input_shape=(40, 100)))
model.add(Dropout(0.2))
```

2.2.6 Model und Gewichte speichern

In Keras kann man jederzeit das Model mit seiner Architektur, Aktivierungen sowie aktuell gesetzten Gewichten und Bias im `hdf5`-Format speichern.

Das nachfolgende Beispiel zeigt das Speichern und das Laden des vorher konstruierten Models. Diese Funktionen werden in der Implementierung verwendet, um parallel mehrere Architekturen trainieren zu können und die Auswertung vom Training zu trennen.

```python
from keras.models import load_model

model.save("my_model.hdf5")
del model
model = load_model("my_model.hdf5")
```

Nachdem das Model geladen wurde, kann es wieder kompiliert werden und neue Voraussagen zur Auswertung treffen:

```python
model.compile(loss="categorical_crossentropy", optimizer="adam")
prediction = model.predict(x_test, batch_size=128)
```

Kapitel 3

Vorverarbeitungsschritte für Quellcode

In dem folgenden Kapitel werden im Abschnitt 3.1 die Auswahl des Quellcodes zum Trainieren und im Abschnitt 3.2 dessen vorbereitende Maßnahmen besprochen, die ihn vereinfachen und bereinigen sollen, wodurch das KNN besser lernen soll.

3.1 Auswahl der Trainingsdaten

Um ein neuronales Netzwerk auf das Vervollständigen von Quelltext trainieren zu können, benötigt man zu aller erst viele Trainingsdaten mit denen es lernen soll.

Dabei ist es sinnvoll sich auf eine Programmiersprache zu begrenzen, um auf möglichst ähnlichen Quellcode zu trainieren. Dies erleichtert dem KNN das Erlernen von Mustern, da diese öfter vorkommen. Man könnte Quellcode aus jeder beliebigen Programmiersprache nehmen, aber einige eignen sich mehr als andere. Da Java [48] die zur Zeit beliebteste und weitverbreiteste Programmiersprache ist [9], lassen sich viele freie Java-Projekte als Datensatz finden, die für das Trainieren des Netzes erforderlich sind.

Außerdem ist es wichtig, dass die Domäne für alle Dateien gleich ist, damit das neuronale Netz besser lernen kann. Wenn das Netz auf den Trainingsdaten trainiert wird, lernt es Muster zu erkennen und zu rekonstruieren. Wenn der Eingabequellcode zu heterogen ist, kann es keine Muster lernen. Daher sollte der Code relativ homogen sein. Um dies sicherzustellen, werden Java-Projekte mit einer ähnlichen Domäne gesucht, die dadurch wahrscheinlich eine ähnliche Semantik aufweisen. In Java gibt es Bibliotheken, die sich auf bestimmte Anwendungsbereiche spezialisiert haben. Dadurch ist der Quellcode, der eine bestimmte Bibliothek einbindet, oftmals semantisch ähnlich und bedient die gleiche Domäne. Hier wurde sich auf die Java-Bibliothek `javax.swing` festgelegt, mit der grafische Benutzeroberflächen erstellt werden können. Das sollte die Ähnlichkeit und Homogenität der Projekte sicherstellen.

Zusätzlich wird in Java der Sichtbarkeitsbereich von Variablen und Methoden mit Hilfe von Einrückungen bestimmt. Die Einrückungen in Java dienen nur der Übersichtlichkeit für den Programmierer und haben keinen Einfluss auf die Korrektheit des Codes. Solange die Klammern richtig gesetzt werden, kann eine ganze Datei auch in eine Zeile geschrieben werden und ist trotzdem ausführbar. Dies wäre z.B. in Python nicht möglich. Dort werden die Zeilenumbrüche und Einrückungen zum Kompilieren benötigt. Einrückungen werden in Python entweder durch eine Nutzung von Leerzeichen oder Tabs gekennzeichnet. Dabei dürfen beide Formen nicht in einer Datei gemischt werden. Je tiefer die Verschachtelung der Methoden, desto weiter die Einrückung. Diese Konventionen müsste ein KNN mitlernen, was mehr Aufwand bedeuten würde. Beim Trainieren auf Java-Dateien kann dieser Aufwand vermieden werden.

Wenn die Einrückungen wie in Python nötig sind und solch eine Sprache als Datensatz zum Trainieren ausgewählt werden würde, müsste das KNN lernen, diese in der richtigen Anzahl und an der richtigen Stelle zu verwenden. Dies würde einen längeren Lernprozess nach sich ziehen, was mit einer geklammerten Sprache wie Java umgangen werde kann.

Aufgrund dieser Gegebenheiten wurde Java als Programmiersprache für die Trainingsdaten des KNNs ausgewählt.

Nun wurde nach einer Quelle nach Java-Trainingsdaten gesucht, die offen sowie frei verfügbar ist und viel Code zum Trainieren bereitstellt. `Github`, eine Online-Entwicklungsplattform mit Versionsverwaltung, ist als Ausgangspunkt geeignet, da dies die weltweit größte Plattform von Open-Source Code ist. Aufgrund der Popularität von Java war es leicht, viele Java-Projekte auf dieser Website zu finden.

Wie bereits erwähnt, sollte der Trainingsdatensatz relativ ähnlich sein. Deshalb muss die Bibliothek `javax.swing` in den herunterzuladenden Github-Projekten enthalten sein.

Da sehr viele Daten zum ausreichenden Erlernen von Mustern in Java-Code gesammelt werden mussten, wurde ein Shell-Script benutzt. Projekte kann man automatisiert von Github herunterladen, indem man sich der Github-API bedient. Um diese nutzen zu können, muss man zuvor einen API-Token zur Autorisierung erstellen. Dieser dient als eine Art Passwort und wird beim Aufruf des Scripts mit angegeben, um Projekte von Github im Namen des Token-Erstellers herunterladen zu können.

Es wurden die ersten 100 Projekte als *tar.gz* Dateien von Github heruntergeladen, die das Ergebnis der Anfrage *swing* waren. Diese Projekte lagen im Archivformat `tar` vor und beinhalteten alle Versionen des jeweiligen Projektes. Jedes Archiv wurde mit Hilfe eines Scripts entpackt und die Java-Dateien aus jeder Projektversion wurden in einem ausgelagerten Ordner gespeichert. Dieser Datensatz von 39,6 MB wird als Rohdaten bezeichnet und im nächsten Abschnitt weiter bearbeitet.

3.2 Vorverarbeitung

Um den Einfluss der Formatierung des Codes auf die Generierung von Quellcode zu beobachten, wurden verschiedene Vorbereitungsstufen der Trainingsdaten betrachtet.

Den Ausgangspunkt bilden die Rohdaten, der unbearbeitete Quelltext. Dieser enthält alle Kommentare, Leerzeilen und Einrückungen der Autoren. Die beispielhafte Datei *AbstractBandControlPanel.java* sieht folgendermaßen aus:

```java
/*
 * Copyright (c) 2005-2016 Flamingo Kirill Grouchnikov.
 * All Rights Reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 *  o Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 *  o Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 *  o Neither the name of Flamingo Kirill Grouchnikov nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
 * TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
package org.pushingpixels.flamingo.internal.ui.ribbon;

import javax.swing.JPanel;
import javax.swing.plaf.UIResource;

import org.pushingpixels.flamingo.api.ribbon.AbstractRibbonBand;
import org.pushingpixels.flamingo.api.ribbon.JRibbonBand;

/**
```

```
 * Control panel of a single {@link JRibbonBand}. This class is for
 * internal use only and should not be directly used by the applications.
 *
 * @author Kirill Grouchnikov
 */
public class AbstractBandControlPanel extends JPanel implements UIResource
    {
  private AbstractRibbonBand ribbonBand;

  public AbstractBandControlPanel() {
  }

  /*
   * (non-Javadoc)
   *
   * @see javax.swing.JPanel#getUI()
   */
  @Override
  public BandControlPanelUI getUI() {
    return (BandControlPanelUI) ui;
  }

  public void setRibbonBand(AbstractRibbonBand ribbonBand) {
    this.ribbonBand = ribbonBand;
  }

  public AbstractRibbonBand getRibbonBand() {
    return this.ribbonBand;
  }
}
```

Listing 3.1: AbstractBandControlPanel.java unbearbeitet

Da ein KNN jegliche Muster aus dem gegebenen Quelltext lernt, ist es wichtig, ihm nur relevante Muster zu zeigen und beizubringen, um die Genauigkeit der späteren Codevoraussage zu erhöhen. Jedes für die Funktionalität des Codes überflüssige Zeichen raubt dem Netzwerk wertvolle Lernkapazitäten, die es für gehaltvolle Zeichen und Muster nutzen sollte.

Um den Quellcode für das Netzwerk von nicht-relevanten Inhalten zu säubern, wurden vier Vorverarbeitungsschritte bzw. Maßnahmen (M1 bis M4) und damit vier weitere Trainingsdatensätze erstellt. Bei der ersten Maßnahme wurden Kommentare entfernt, bei der zweiten Leerzeilen sowie Einrückungen, bei der dritten eine Kombination aus entfernten Kommentaren, Leerzeilen sowie Einrückungen und bei der letzten wird die Kombination um das Vereinfachen der Methoden- und Feldnamen erweitert.

3.2.1 M1: Kommentare entfernen

Maßnahme 1 beinhaltet das Entfernen von Kommentaren aus den Rohdaten. Da KNNs versuchen alle Muster in den Trainingsdaten zu erlernen, würden sie auch versuchen die Kommentare des Autors zu lernen. Diese dienen im Quelltext jedoch nur der Verständlichkeit und bieten dem Programmierer Raum seine Vorgehensweise zu erklären und Anmerkungen zu machen. Sie haben keinen Einfluss auf das Ergebnis des Codes. Aus diesem Grund können sie entfernt werden und dem Netzwerk ermöglichen den verbliebenen Code besser zu lernen. Der veränderte Quellcode ist immer noch ausführbar und liefert die gleichen Ausgaben wie zuvor. Es bleiben dem Netz dadurch nur mehr Kapazitäten zum Erlernen von Mustern des verkürzten, aber dennoch korrekten Codes.

Die Kommentare wurden aus den Rohdaten mit Hilfe eines Scripts gelöscht, die mit einer Kombination aus folgenden regulären Ausdrücken gefunden wurden. Der erste reguläre Ausdruck findet einzeilige Kommentare:

```
(//[^\n]*)
```

Der zweite Teil identifiziert ein- und mehrzeilige Kommentare:

```
(?s)/\*.*?\*/
```

Zum Schluss werden die gefundenen Zeichenketten gelöscht. Die Beispieldatei *AbstractBandControlPanel.java* sieht nach dem Anwenden der ersten Maßnahme wie folgt aus:

```java
package org.pushingpixels.flamingo.internal.ui.ribbon;

import javax.swing.JPanel;
import javax.swing.plaf.UIResource;

import org.pushingpixels.flamingo.api.ribbon.AbstractRibbonBand;
import org.pushingpixels.flamingo.api.ribbon.JRibbonBand;


public class AbstractBandControlPanel extends JPanel implements UIResource
   {
  private AbstractRibbonBand ribbonBand;

  public AbstractBandControlPanel() {
  }



  @Override
  public BandControlPanelUI getUI() {
    return (BandControlPanelUI) ui;
  }

  public void setRibbonBand(AbstractRibbonBand ribbonBand) {
    this.ribbonBand = ribbonBand;
  }
```

```
  public AbstractRibbonBand getRibbonBand() {
    return this.ribbonBand;
  }
}
```

Listing 3.2: AbstractBandControlPanel.java nach M1

Dieser Vorverarbeitungsschritt führte für dieses Beispiel zu einer Reduktion von 69 Zeilen auf 29 Zeilen und 2546 Zeichen auf 663 Zeichen; ohne, dass sich die Semantik und die Syntax des eigentlichen Quelltextes geändert hat.

3.2.2 M2: Leerzeilen und Einrückungen entfernen

Maßnahme 2 entfernt alle Leerzeilen und Einrückungen aus den Rohdaten.

Leerzeilen dienen nur der Übersichtlichkeit und sind für die Semantik des Quelltextes unbedeutend. Das neuronale Netz muss sie nicht mit lernen, um korrekten Code vorauszusagen. Daher wurden die Rohdaten durch ein Script von ihnen befreit.

Die Einrückungen können entfernt werden, da in Java der Sichtbarkeitsbereich von Variablen und Methoden mit Hilfe von Klammern und nicht von Einrückungen (wie z.B. in Python) bestimmt wird. Würde man die Einrückungen mit lernen, würde das Netzwerk Code generieren, der diese enthält. Durch das Lernen mit leeren Zeilen und Einrückungen besetzen diese Speicher, den das Netzwerk für den eigentlichen Code gebrauchen könnte.

Außerdem würde das KNN sie nicht immer an die richtigen Stellen setzen. Es müsste erst lernen, nur am Zeilenanfang einzurücken und nicht innerhalb einer Zeile. Zudem müsste es lernen, wann man wie weit einrückt und man Leerzeilen benutzt, um den Code klar zu strukturieren. Doch diese Fähigkeit muss es nicht besitzen. Es genügt, wenn es syntaktisch und ggf. semantisch korrekten Code erzeugen kann.

Die Leerzeilen wurden aus den Rohdaten mit Hilfe eines Scripts und dem folgenden regulären Ausdrücken gelöscht:

```
(?m)^[ \t]*\r?\n
```

Die Einrückungen wurden mit Hilfe eines weiteren Scripts gelöscht, das den folgenden regulären Ausdrücken enthielt:

```
^\s+
```

Die Datei *AbstractBandControlPanel.java* sieht ohne Leerzeilen und Einrückungen so aus:

```
/*
 * Copyright (c) 2005-2016 Flamingo Kirill Grouchnikov.
 * All Rights Reserved.
 *
 * Redistribution and use in source and binary forms, with or without
```

```
 * modification, are permitted provided that the following conditions are
 * met:
 *
 *  o Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 *  o Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 *  o Neither the name of Flamingo Kirill Grouchnikov nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
 * TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
package org.pushingpixels.flamingo.internal.ui.ribbon;
import javax.swing.JPanel;
import javax.swing.plaf.UIResource;
import org.pushingpixels.flamingo.api.ribbon.AbstractRibbonBand;
import org.pushingpixels.flamingo.api.ribbon.JRibbonBand;
/**
* Control panel of a single {@link JRibbonBand}. This class is for
* internal use only and should not be directly used by the applications.
*
* @author Kirill Grouchnikov
*/
public class AbstractBandControlPanel extends JPanel implements UIResource
    {
private AbstractRibbonBand ribbonBand;
public AbstractBandControlPanel() {
}
/*
* (non-Javadoc)
*
* @see javax.swing.JPanel#getUI()
*/
@Override
public BandControlPanelUI getUI() {
return (BandControlPanelUI) ui;
```

```
}
public void setRibbonBand(AbstractRibbonBand ribbonBand) {
this.ribbonBand = ribbonBand;
}
public AbstractRibbonBand getRibbonBand() {
return this.ribbonBand;
}
}
```

Listing 3.3: AbstractBandControlPanel.java nach M2

Dieser Vorverarbeitungsschritt führte, ohne die Semantik oder die Syntax des Originalquelltextes zu verändern, für dieses Beispiel zu einer Reduktion von 69 Zeilen auf 62 Zeilen und 2546 Zeichen auf 2509 Zeichen.

3.2.3 M3: Kombination aus M1 und M2

Um dem KNN nur die wesentlichen Bestandteile eines funktionierenden Programmcodes beizubringen, wurden M1 und M2 kombiniert, um die Rohdaten von Kommentaren, Leerzeilen und Einrückungen zu befreien.

AbstractBandControlPanel.java sieht durch Maßnahme 3 folgendermaßen aus:

```
package org.pushingpixels.flamingo.internal.ui.ribbon;
import javax.swing.JPanel;
import javax.swing.plaf.UIResource;
import org.pushingpixels.flamingo.api.ribbon.AbstractRibbonBand;
import org.pushingpixels.flamingo.api.ribbon.JRibbonBand;
public class AbstractBandControlPanel extends JPanel implements UIResource
    {
private AbstractRibbonBand ribbonBand;
public AbstractBandControlPanel() {
}
@Override
public BandControlPanelUI getUI() {
return (BandControlPanelUI) ui;
}
public void setRibbonBand(AbstractRibbonBand ribbonBand) {
this.ribbonBand = ribbonBand;
}
public AbstractRibbonBand getRibbonBand() {
return this.ribbonBand;
}
}
```

Listing 3.4: AbstractBandControlPanel.java nach M3

Die Kombination der beiden vorangegangenen Vorverarbeitungsschritte führte zu einer Verminderung von 69 Zeilen in den Rohdaten auf 20 Zeilen und 2546 Zeichen auf 637 Zeichen. Dabei blieb die Semantik und die Syntax des Originals erhalten.

3.2.4 M4: Kombination aus M3 und Namensvereinfachung

Im Fall von Maßnahme 4 wurden Variablen- und Methodennamen durch simple Bezeichner wie $v1$, $v2$ oder $m1$, $m2$ ersetzt. Dadurch lernt das KNN keine unnötigen, vom Entwickler selbst festgelegte Bezeichner für Variablen und Methoden, die beliebig lang und komplex gewählt werden können sowie keinen Einfluss auf die Syntax des Programms haben. Es bleiben mehr Kapazitäten für das Erlernen der Codestruktur.

Die Umbenennung von Variablen- und Methodennamen wurde durch den `JavaParser` [8] ermöglicht. Er erstellt aus dem gegebenen Quelltext einen sogenannten Abstract syntax tree (dt. abstrakter Syntaxbaum) (AST), anhand dessen er Methodendeklarationen (*MethodDeclaration*), -aufrufe (*MethodCallExpr*), -referenzen (*MethodReferenceExpr*), Felddeklarationen (*FieldDeclaration*), -aufrufe (*FieldAccessExpr*) und Variablendeklarationen (*VariableDeclarationExpr*) durchgeht. Der AST wird durchlaufen und verändert die Bezeichner, wenn einer der oben genannten Knoten erreicht wird.

Am Anfang werden zwei `Hashmaps` erzeugt, die Strings als Schlüssel und Integer als Werte enthalten. Die erste `Hashmap` ist für Variablennamen (`mapVariables`) und die zweite für Methodennamen (`mapMethods`) zuständig. Der Schüssel ist der Name des Bezeichners und der Wert gibt einen fortlaufenden Zähler an, der für die spätere Umbenennung in z.B. $v1$ für $Variable1$, verantwortlich ist. Beim Erreichen eines Namens wird überprüft, ob er schon in einer der beiden `Hashmaps` vorhanden ist. Falls ja, wird er übersprungen. Falls nein, wird der aktuell höchste Zähler der `Hashmap` herausgesucht und um eins erhöht. Dies ist nun der neue Zähler für den aktuellen Namen, der hinzugefügt wird. Der neue Name beginnt mit einem v für Variablem oder einem m für Methoden, gefolgt von dem eben festgestellten Zähler. Nachdem der Name verändert wurde, wird der AST weiter durchlaufen, bis er auf den nächsten relevanten Knoten trifft.

Es ist wichtig, dass durch Bibliotheken festgelegte Methoden und Parameter beibehalten werden. Daher wurden nur die Methoden- und Feldaufrufe verändert, die zuvor durch die Methoden- bzw. Felddeklaration in die `Hashmap` `mapMethods` bzw. `mapVariables` aufgenommen wurden.

Da dies aber nicht alle Namen verändert, wird danach jede Datei zusätzlich nach Einträgen der beiden `Hashmaps` abgesucht. Wenn ein solcher Eintrag gefunden wurde, wird er in seine dazugehörige Kurzform wie beispielsweise $v1$ oder $m1$ umbenannt.

Hier ist ein Beispiel aus der Datei AbstractBandControlPanel.java, deren Variablen- und Methodennamen nur durch den `JavaParser` umbenannt wurden.

```java
public void m1177(AbstractRibbonBand ribbonBand) {
  this.m876 = ribbonBand;
}
```

Listing 3.5: AbstractBandControlPanel.java nach JavaParser-Aufruf

Der Bezeichner *ribbonBand* wurde hierbei nicht umbenannt. Wenn nun die Datei erneut durchgegangen wird und dabei alle gefundenen Einträge der *Hashmaps* umbenannt werden, wird *ribbonBand* erkannt und angepasst.

```
public void m1177(AbstractRibbonBand v876) {
  this.m876 = v876;
}
```

Listing 3.6: AbstractBandControlPanel.java nach JavaParser-Aufruf und zusätzlichem Absuchen nach *Hashmap*-Einträgen

Nun wurden alle Bezeichner in diesem Ausschnitt aus *AbstractBandControlPanel.java* vereinfacht. Dennoch wurden durch den `JavaParser` alle Java-typischen Einrückungen und Leerzeilen eingefügt, auch wenn der Eingangsquelltext bereits von diesen befreit wurde. Aufgrund dessen wurde das Script von M2 zum Schluss angewendet.

Die gesamte Datei *AbstractBandControlPanel.java* sieht nach Maßnahme 4 in dieser Weise aus:

```
package org.pushingpixels.flamingo.internal.ui.ribbon;
import javax.swing.JPanel;
import javax.swing.plaf.UIResource;
import org.pushingpixels.flamingo.api.ribbon.AbstractRibbonBand;
import org.pushingpixels.flamingo.api.ribbon.JRibbonBand;
public class AbstractBandControlPanel extends JPanel implements UIResource
   {
private AbstractRibbonBand v876;
public AbstractBandControlPanel() {
}
@Override
public BandControlPanelUI m652() {
return (BandControlPanelUI) ui;
}
public void m1177(AbstractRibbonBand v876) {
this.m876 = v876;
}
public AbstractRibbonBand m1178() {
return this.m876;
}
}
```

Listing 3.7: AbstractBandControlPanel.java nach M4

Während die Semantik und die Syntax aus den Rohdaten erhalten blieb, reduzierte dieser Vorverarbeitungsschritt 69 Zeilen und 2546 Zeichen der Rohdaten auf 20 Zeilen und 590 Zeichen. Im Vergleich dazu hatte das Entfernen von Kommentaren, Einrückungen und Leerzeile zu 20 Zeilen und 637 Zeichen geführt.

3.3 Zeichenweise Kodierung

Neuronale Netze können auf verschiedenen Granularitäten trainiert werden; z.b. auf Wort-, Silben- und Zeichenbasis. Jedoch haben RNNs in anderen Untersuchungen [31] bemerkenswerte Resultate auf Zeichenbasis erzielt, sodass in dieser Arbeit diese Art des Einlesens und Voraussagens genutzt wird. Die folgende Implementierung beruht auf einem Beispiel des `Keras`-Teams auf Github [13], das Text anhand von Werken von Nietzsche generiert.

Man würde normalerweise damit beginnen, alle Trainigsdaten einzulesen und in eine Form zu bringen mit der das Netz arbeiten kann. Aufgrund von Speicherproblemen ist es aber nicht möglich, den Datensatz einmal im Ganzen vor Beginn des Trainings einzulesen und in einer für das Netzwerk verwendbaren Form abzuspeichern. Daher wird zwar der ganze Quelltext am Anfang als *String* eingelesen, aber noch nicht vektorisiert. Wie die Vektorisierung genau verläuft, wird weiter unten anhand der Methode `load_one_data` erklärt. Auf den hier genutzten Rechnern hat der Speicher nicht für den großen Vektor ausgereicht, der für die gesamte Eingabe nötig gewesen wäre. Daher mussten die Trainingsdaten auf maximal 10 MB große Dateien, im folgenden `chunks` genannt, aufgeteilt werden. Erst während des Trainings wird in jeder Epoche eine Datei gezogen und vektorisiert. Der benötigte Speicher für das Vektorisieren eines `chunks` ist bereits enorm: Bei den Rohdaten z.B. beträgt die Anzahl der `sentences` (siehe weiter unten) 3328552, die `maxlen_window` (siehe weiter unten) 40 und die Anzahl an einzigartigen Zeichen 221. Dadurch muss später ein riesiges dreidimensionales `Array` der Größe 3328552 mal 40 mal 221 erstellt werden.

In einer separaten Datei werden für jede Netzwerkarchitektur und je Optimierer die Parameter wie z.B. Anzahl der Epochen, der Pfad zu den Trainingsdaten und der Pfad zum Speichern der Gewichte gesetzt. Welche verschiedene Netzwerkarchitekturen benutzt wurden, wird in Kapitel 4 erklärt. Dann wird der Datensatz im gesamten als `String` namens `whole_text` eingelesen. Alle einzigartigen Zeichen (`chars`) und deren Anzahl (`len_chars`) werden mit den folgenden 2 Zeilen bestimmt:

```python
chars = sorted(list(set(whole_text)))
len_chars = len(chars)
```

Dies muss nur einmal für alle Architekturen eines Datensatzes durchgeführt werden.

In der Hauptdatei wird dann das jeweilige `Keras` Model konstruiert. Danach wird es mit der Zielfunktion `categorical_crossentropy` und dem Optimierer `Adam` oder `RMSprop` kompiliert. Der Optimierer wird in der separaten Datei als Variable `optimizer` festgelegt und übergeben.

```python
model.compile(loss="categorical_crossentropy", optimizer=optimizer)
```

Dann beginnt das Training. In jeder Epoche wird, wie gesagt, ein Datensatz-`chunk` zufällig gewählt. Dieser wird mit Hilfe der Methode `load_one_data` vektorisiert.

```
for i in range(num_epochs):
  j = random.randint(0, number_of_chunks)
  file = code_file_base+str(j)+".txt"
  text, X, y, char_indices, indices_char = load_one_data(file,
      log_file_data, chars)
```

In `load_one_data` geschieht folgendes. Jedem Zeichen der aktuellen `chunk`-Datei wird eine Ganzzahl (`integer`) zugeteilt und anderes herum.

```
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

Danach werden Textsequenzen (`sentences`) der aktuellen `chunk`-Datei der Länge `maxlen_window` sowie deren nachfolgendes Zeichen (`next_char`) erstellt. Dies sind die Eingabe-Ausgabe-Paare, die zum überwachten Lernen benötigt werden. Die Eingabe ist eine Sequenz von `maxlen_window` vielen Zeichen und die Ausgabe ist das anschließende Zeichen dieser Sequenz. Die nächste Sequenz beginnt `step_size` Zeichen weiter als die vorherige Sequenz. In dieser Arbeit beträgt `maxlen_window` 40 und `step_size` 3. Diese Werte wurden aus dem `Keras` Textgenerierungsbeispiel [13] übernommen, da sie dort zu guten Ergebnissen geführt haben.

```
sentences = []
next_chars = []

for i in range(0, len(text) - maxlen_window, step_size):
  sentences.append(text[i: i + maxlen_window])
  next_chars.append(text[i + maxlen_window])
```

Die Paare müssen jedoch noch ins *one-hot encoding* umgewandelt werden, damit das KNN darauf trainieren kann. Dies geschieht in dem nächsten Schritt.

```
import numpy as np

X = np.zeros((len(sentences), maxlen_window, len_chars), dtype=np.bool)
y = np.zeros((len(sentences), len_chars), dtype=np.bool)

for i, sentence in enumerate(sentences):
  for t, char in enumerate(sentence):
    X[i, t, char_indices[char]] = 1
  y[i, char_indices[next_chars[i]]] = 1
```

Anschließend kann `model.fit()` aufgerufen werden. Für die Netzwerke **1L128**, **2L128** und **2LD128** (Erklärung der Netze siehe Abschnitt 4.5) sieht dies folgendermaßen aus:

```
model.fit(X, y,
  batch_size=128,
  epochs=1,
  shuffle=False,
  callbacks=callbacks_list)
```

Die `batch_size` richtet sich dabei nach der Anzahl der Knoten in der Eingabeschicht des Netzwerkes. Die Netze **1L128**, **2L128** und **2LD128** besitzen je 128 Eingabeneuronen. Für andere Netze mit abweichenden Knotenanzahlen in der Eingabeschicht muss man eine angepasste `batch_size` verwenden. Der Parameter `callbacks` wird genutzt, um nach jeder Epoche das aktuelle `model` samt seinen trainierten Gewichten zu speichern. Die `callbacks_list` wurde zuvor so erstellt:

```
# {epoch:02d} speichert aktuelle Epoche mit 2 Stellen Genauigkeit
# {loss:.3f} speichert aktuellen loss-Wert mit 3 Dezimalstellen
    Genauigkeit
filepath = "checkpoints/raw/checkpoint-"+model_type+"-"+optimizer_name+"
    -{epoch:02d}-{loss:.3f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1,
    save_best_only=False, mode='min')
callbacks_list = [checkpoint]
```

Nachdem eine Epoche trainiert wurde, beginnt man wieder bei der zufälligen Auswahl eines `chunks`, der mit `load_one_data` vektorisiert und nachfolgend trainiert wird, bis insgesamt 20 Epochen pro Netzwerkarchitektur durchlaufen wurden.

Kapitel 4

Netzwerkarchitekturen

In dem folgenden Abschnitt werden die verschiedenen Netzwerkarchitekturen vorgestellt, die in dieser Arbeit genutzt wurden. Sie unterscheiden sich in ihrer Anzahl an Schichten (siehe Abschnitt 4.1), Anzahl der Neuronen pro Schicht (Abschnitt 4.2), der Art des Optimierers (Abschnitt 4.3) und ob die Technik *Dropout* (Abschnitt 4.4) eingesetzt wird oder nicht.

4.1 Anzahl der Schichten

Mit der Größe eines KNNs bestimmt sich auch dessen Speicherkapazität, Einstellungsmöglichkeiten und Fähigkeit zu lernen. Die Anzahl der Schichten in einem neuronalen Netz bestimmen die Komplexität der Muster, die gelernt werden können. Hier wird untersucht, ob sich mit der Anzahl der Schichten in einem Netzwerk auch die Genauigkeit dessen Voraussagen von Code erhöht oder ob es sich überanpasst.

Wenn sich ein Netzwerk überanpasst, bedeutet das, dass es jede noch so unbedeutende Eigenschaft in den Trainingsdaten lernt und nicht den generellen Aufbau. Es generalisiert nicht, sondern stimmt sich genau auf den Datensatz ab. Wenn aber nun neuer Code als Eingabe gegeben wird, der ergänzt werden soll, kann das Netz keine guten Voraussagen treffen. Es wird auf diesen Testdaten viel schlechter abschneiden als auf den Trainingsdaten, da es zu viele irrelevante Muster gelernt hat. Neuronale Netze, die sehr komplex sind, neigen zu Überanpassung.

Zu einfache Netze hingegen können unterangepasst sein und zu wenig generelle Strukturen gelernt haben. Sie liefern schlechte Voraussagen bei den Trainings- sowie den Testdaten. Sie orientieren sich zu wenig an dem realen Trainingsdatensatz und verfehlen die gewünschten Voraussagen.

Daher wird immer versucht einen Mittelweg zwischen Über- und Unteranpassung zu finden, was bedeutet, dass das Netz nicht zu komplex, aber auch nicht zu simpel sein darf.

In dieser Arbeit werden vorrangig LSTM-Schichten benutzt, da sie sich als effektives Modell für sequentielle Daten in vielen verschiedenen Gebieten erwiesen haben [32]. Einige Beispiele sind Textgenerierung, Handschrifterkennung und Handschrifterzeugung [21], maschinelle Übersetzung [57], Spracherkennung [22], Bildüberschriftengenerator [63] und Videoanalyse [15]. Empirische Studien haben gezeigt, dass RNNs besonders gut im Modellieren von Syntaxaspekten wie korrekte Klammerung, Einrückung, etc., sind [32].

Es werden verschieden viele LSTM-Schichten betrachtet: 1, 2, 3 und 5 Schichten. Sie geben zusammen mit der Anzahl der Knoten in einem KNN die Komplexität des Netzes an.

In `Keras` wird eine LSTM-Schicht wie folgt zu einem `model` hinzugefügt:

```
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen_window, len_chars)))
```

Wenn zwei LSTM-Schichten zu einem `model` hinzugefügt werden sollen, muss der Parameter `return_sequences` der ersten Schicht auf wahr gesetzt werden, damit diese ihre volle dreidimensionale Ausgabe an die zweite Schicht weitergibt. Die ersten zwei Dimensionen sind durch `maxlen_window` und `len_chars` gegeben, die dritte Dimension ist die Zeit. Ist `return_sequences` auf falsch gesetzt, wird die erforderliche zeitliche Dimension der Eingaben nicht an die weitere Schicht weitergegeben. Bei der letzten Schicht wird `return_sequences` auf falsch gesetzt (standardmäßig auf falsch [14]), um die vereinfachte Ausgabe weiterzugeben.

```
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen_window, len_chars),
    return_sequences=True))
model.add(LSTM(128, input_shape=(maxlen_window, len_chars)))
```

Hier soll untersucht werden, ob die wachsende Anzahl der Schichten ausschließlich positiv auf das Voraussagen von Code auswirkt oder ob es ab einer bestimmten Anzahl bzw. in einer bestimmten Kombination mit einer anderen Eigenschaft des Netzwerkes (siehe Abschnitt 4.2, 4.3 und 4.4) zu einer Verschlechterung führt.

4.2 Anzahl der Knoten pro Schicht

Die Anzahl an Muster, die ein KNN lernen kann, hängt von der Anzahl der Knoten ab. In Verbindung mit der Anzahl der Schichten geben sie an, wie viele umfangreiche Muster ein Netzwerk lernen kann. Die Größe und Verworrenheit haben Einfluss auf die Über- oder Unteranpassung des Netzes. Zu viele Neuronen verkomplizieren und zu wenig simplifizieren es, was Über- bzw. Unteranpassung (siehe Abschnitt 4.1) zur Folge habe kann. Die Erhöhung der Anzahl der Knoten pro Schicht wird in Verbindung mit der Genauigkeit von Voraussagen von Code betrachtet. Zur Untersuchung wurden unterschiedliche viele Knoten beobachtet: 128, 256, 300 und 500 Knoten.

4.3 Art des Optimierers

Das KNN kann mit verschiedenen Optimierern (siehe Abschnitt 2.1.8 und Abschnitt 2.2.3) trainiert werden. Hier werden `Adam` und `RMSprob` evaluiert.

Beide eignen sich gut [34][52], um die Gewichte in einem neuronalen Netz so anzupassen, dass die Fehlerfunktion minimiert wird, wobei `Adam` etwas bevorzugt wird, aber `RMSprop` dennoch einen Versuch wert ist [30]. Aus diesem Grund werden beide Optimierer in dieser Arbeit betrachtet.

4.4 Dropout

Diverse Arbeiten über die etablierte Regularisierungstechnik `Dropout` [56][26][66] haben gezeigt, dass dessen Einsatz für viele Anwendungsbereiche die Überanpassung eines Netzwerkes verhindert und die Generalisierung fördert. Dieses positive Verhalten soll hier ebenfalls gezeigt werden.

Die Einführung der etablierten Technik `Dropout`, die bereits in Abschnitt 2.2.5 erläutert wurde, soll wie in den anderen Arbeiten zu einer Verbesserung der Generalisierung des Netzwerkes führen.

Es wird eine Vergessensrate von 0.2 bis 0.5 empfohlen, da ein geringerer Betrag keinen Effekt zeigt, aber ein zu hoher in einem unterangepassten KNN resultiert [6]. Die Anwendung von `Dropout` auf jede Schicht hat gute Ergebnisse geliefert [6]. Deshalb wurde in den Architekturen 2LD128, 5LD500 und 10LD500 (siehe Abschnitt 4.5) je eine `Dropout`-Schicht mit einer Rate von 0.2 zwischen zwei LSTM-Schichten eingefügt.

4.5 Vergleich der Architekturen

Alle neuronalen Netze verfügen über eine `softmax`-Aktivierung sowie eine `Dense`-Schicht am Ende und wurden auf der Zielfunktion `categorical_crossentropy` trainiert.

Um später den Quelltext auf Basis eines Codeausschnitts vorauszusagen, wurde die `categorical crossentropy` Funktion als Fehlerfunktion angewendet, sodass die Ausgaben des KNNs als Wahrscheinlichkeiten interpretiert werden können. Aufgrund dieser Wahrscheinlichkeiten werden später die zu voraussagenden Zeichen gezogen. Die verschiedenen Anzahlen von Schichten, Knoten pro Schicht, Optimierungsart und ggf. der Einsatz von `Dropout` ergeben zusammen folgende KNNs:

1L128 Ein Netzwerk mit einer LSTM-Schicht (*1L*) mit 128 Neuronen (siehe Abbildung 4.1).

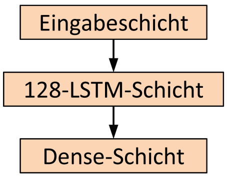

Abbildung 4.1: Neuronales Netz 1L128

Die Erstellung in `Keras` sieht wie folgt aus:

```
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen_window, len_chars)))
model.add(Dense(len_chars))
model.add(Activation("softmax"))
```

2L128 Ein Netzwerk mit zwei LSTM-Schichten (*2L*) mit je 128 Neuronen (siehe Abbildung 4.2).

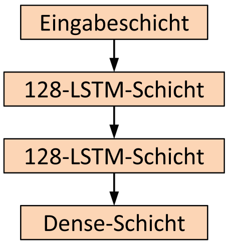

Abbildung 4.2: Neuronales Netz 2L128

In `Keras` wird es wie folgt erstellt:

```
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen_window, len_chars),
    return_sequences=True))
model.add(LSTM(128, input_shape=(maxlen_window, len_chars)))
model.add(Dense(len_chars))
model.add(Activation("softmax"))
```

2LD128 Ein Netzwerk mit zwei LSTM-Schichten (*2L*) mit je 128 Neuronen und einer `Dropout`-Schicht (*D*) zwischen den LSTM-Schichten (siehe Abbildung 4.3).

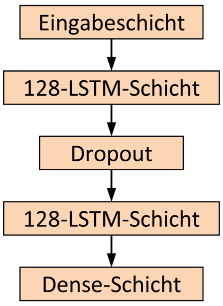

Abbildung 4.3: Neuronales Netz 2LD128

Die Erstellung in `Keras` geschieht folgendermaßen:

```
model = Sequential()
model.add(LSTM(128, input_shape=(maxlen_window, len_chars),
    return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(128, input_shape=(maxlen_window, len_chars)))
model.add(Dense(len_chars))
model.add(Activation("softmax"))
```

2LD256 Ein Netzwerk mit zwei LSTM-Schichten ($2L$) mit je 256 Neuronen und je einer `Dropout`-Schicht (D) zwischen den LSTM-Schichten (siehe Abbildung 4.4).

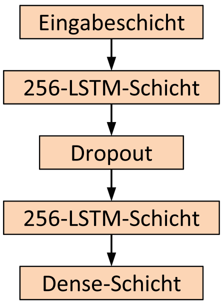

Abbildung 4.4: Neuronales Netz 2LD256

Die Erstellung in `Keras` findet folgendermaßen statt:

```
model = Sequential()
model.add(LSTM(256, input_shape=(maxlen_window, len_chars),
    return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256, input_shape=(maxlen_window, len_chars)))
model.add(Dense(len_chars))
model.add(Activation("softmax"))
```

3LD256 Ein Netzwerk mit drei LSTM-Schichten (*3L*) mit 256 Neuronen in der ersten Schicht, 128 in der zweiten und 64 in der letzten sowie je einer `Dropout`-Schicht (*D*) zwischen den LSTM-Schichten (siehe Abbildung 4.5).

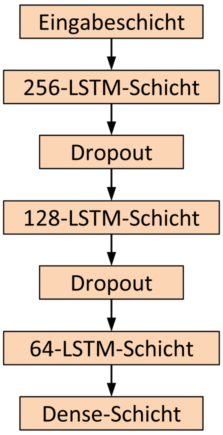

Abbildung 4.5: Neuronales Netz 3LD256

Die Erstellung in `Keras` findet folgendermaßen statt:

```
model = Sequential()
model.add(LSTM(256, input_shape=(maxlen_window, len_chars),
    return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(128, input_shape=(maxlen_window, len_chars),
    return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(64, input_shape=(maxlen_window, len_chars)))
model.add(Dense(len_chars))
model.add(Activation("softmax"))
```

3LD300 Ein Netzwerk mit drei LSTM-Schichten (*10L*) mit 300 Neuronen in der ersten Schicht, 150 in der zweiten und 50 in der letzten sowie je einer `Dropout`-Schicht (*D*) zwischen den LSTM-Schichten (siehe Abbildung 4.6).

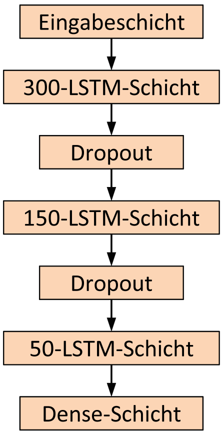

Abbildung 4.6: Neuronales Netz 3LD300

Die Erstellung in `Keras` findet folgendermaßen statt:

```
model = Sequential()
model.add(LSTM(300, input_shape=(maxlen_window, len_chars),
    return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(150, input_shape=(maxlen_window, len_chars),
    return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(50, input_shape=(maxlen_window, len_chars)))
model.add(Dense(len_chars))
model.add(Activation("softmax"))
```

5LD500 Ein Netzwerk mit fünf LSTM-Schichten (*5L*) mit je 500 Neuronen und je einer `Dropout`-Schicht (*D*) zwischen den LSTM-Schichten (siehe Abbildung 4.7).

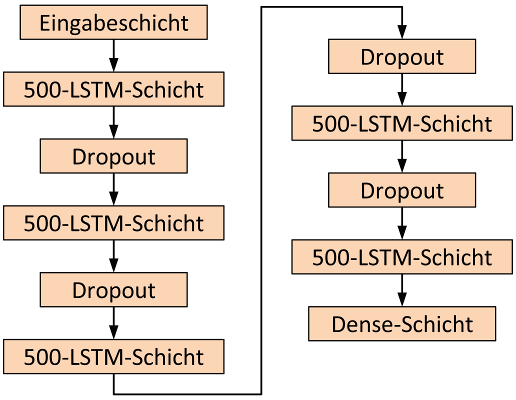

Abbildung 4.7: Neuronales Netz 5LD500

Es folgt die Erstellung in `Keras`:

```
model = Sequential()
for i in range(4):
  model.add(LSTM(500, input_shape=(maxlen_window, len_chars),
      return_sequences=True))
  model.add(Dropout(0.2))
model.add(LSTM(500, input_shape=(maxlen_window, len_chars)))
model.add(Dense(len_chars))
model.add(Activation("softmax"))
```

Alle KNNs wurden je einmal mit dem Optimierer `Adam` und `RMSprop` konfiguriert, wodurch sich am Ende 14 verschiedene Netzwerke ergeben.

Nachdem das `model` in `Keras` wie oben gezeigt erstellt wurde, wird es der Zielfunktion `categorical_crossentropy` und dem Optimierer `Adam` mit einer Lernrate von 0.001 oder `RMSprop` mit einer Lernrate von 0.0001 kompiliert.

```
model.compile(loss="categorical_crossentropy", optimizer=Adam(lr=0.001))
model.compile(loss="categorical_crossentropy", optimizer=RMSprop(lr
    =0.0001))
```

Danach kann das Training begonnen werden.

Zusätzlich zu den obigen Architekturen sollten 2 weitere betrachtet werden: zum einen **5LD1000** mit 5 LSTM-Schichten (*5L*) mit je 1000 Neuronen und je einer `Dropout`-Schicht (*D*) zwischen den LSTM-Schichten (*5LD1000*), zum anderen **10LD1000** mit 10 LSTM-Schichten (*10L*) mit je 1000 Neuronen und je einer `Dropout`-Schicht (*D*) zwischen den LSTM-Schichten (*10LD1000*). Beide konnten jedoch aufgrund eines dafür nicht ausreichenden Arbeitsspeichers nicht trainiert werden, da die Netzwerke zu groß waren. Wie sich aber in der Evaluierung zeigen sollte, liefert schon das Netzwerk **5LD500** wesentlich schlechtere Ergebnisse auf den vorverarbeiteten Daten als die kleinen Netze, zumindest nach den hier betrachteten 20 Epochen Training. Daher hätten die 2 nicht verwendeten KNNs wahrscheinlich noch schlechter abgeschnitten.

Kapitel 5

Evaluierung

Ziel der Evaluierung ist es, herauszufinden welche Art von Eingabe und welche Netzwerkarchitektur beziehungsweise welche Kombination aus beiden, sich am besten auf die Genauigkeit der Voraussage von Quelltext auswirkt.

Die zu untersuchenden Vorverarbeitungsschritte und Abstraktionsarten der Trainingsdaten sowie die verschiedenen Netzwerkarchitekturen wurden bereits in Kapitel 3 und 4 vorgestellt.

5.1 Forschungsfragen

In dieser Arbeit soll betrachtet werden, welche Vorverarbeitungsschritte und Netzwerkarchitekturen sich am besten für das korrekte Voraussagen von Code eignen. Um dies herauszufinden, werden verschiedene Forschungsfragen (FFs) mit spezifischen Metriken zur Evaluierung definiert.

Die Forschungsfragen beschäftigen sich mit den folgenden 3 Gebieten: Vorverarbeitung, Netzwerkarchitekturen und das Zusammenspiel von beiden.

5.1.1 Vorverarbeitung

Die übergeordnete Frage dieses Abschnitts lautet: Welchen Einfluss haben die einzelnen und kombinierten Schritte der Vorverarbeitung auf die Genauigkeit von Voraussagen von Quelltext?

FF 1: Erhöht das Entfernen von Kommentaren die Genauigkeit von Voraussagen von Code? Wenn das KNN als Trainingsdaten Quelltext mit Kommentaren erhält, lernt es die Kommentare mit und würde sie auch mit voraussagen, obwohl sie keinen Einfluss auf die Funktionalität des Codes haben. Sie dienen nur dem Programmierer und den Menschen, die den Quellcode verstehen wollen, als ergänzende Erklärung. Daher können sie entfernt werden. So muss das Netzwerk weniger irrelevante Muster lernen und hat mehr Speicherkapazitäten für den eigentlichen Code.

FF 2: Erhöht das Entfernen von Einrückungen und Leerzeilen die Genauigkeit von Voraussagen von Code? Da in Java Einrückungen und Leerzeilen nur der Lesbarkeit dienen, aber nicht zur Funktionalität des Programms beitragen, können sie entfernt werden. Das KNN belegt durch unnötige Einrückungen und Leerzeilen nur seine begrenzte Speicherkapazität.

FF 3: Erhöht die Kombination vom Entfernen von Kommentaren, Einrückungen und Leerzeilen die Genauigkeit von Voraussagen von Code? Durch das Entfernen aller 3 Eigenschaften verändert sich nicht die Funktionalität des Codes, aber des Netz muss viel weniger irrelevante Muster lernen. Durch diesen Schritt bleibt nur noch der funktionale Rumpf des Programms bestehen, den das KNN lernen soll. Die Kombination von FF 1 und FF 2 könnte sich noch besser auf die Genauigkeit von Voraussagen von Code auswirken als FF 1 oder FF 2 allein.

FF 4: Erhöht die Kombination aus FF 3 in Verbindung mit dem Umbenennen von Methoden- und Feldnamen die Genauigkeit von Voraussagen von Code? Das Netzwerk lernt die teilweise langen Methoden- und Klassenvariablennamen mit, obwohl sie keine Rolle für den Ablauf des Programms spielen. Daher können ihre Namen verkürzt und vereinfacht werden. Dadurch beleibt mehr Speicher für das Lernen von relevanten Mustern im Code übrig.

5.1.2 Netzwerkarchitekturen

Die übergeordnete Frage dieses Abschnitts lautet: Welchen Einfluss haben die einzelnen Architekturen auf die Genauigkeit von Voraussagen von Code?

FF 5: Erhöht sich die Genauigkeit von Voraussagen von Code mit der Anzahl der Neuronen im Netzwerk? Mit der Erhöhung der Anzahl der Neuronen in einem KNN, erhöht sich dessen Speicherkapazität. Mehr Neuronen lassen das Netzwerk mehr Muster lernen. Ob sich dies positiv auf das Voraussagen von Code ausübt, soll untersucht werden.

FF 6: Erhöht sich die Genauigkeit von Voraussagen von Code mit der Anzahl der Schichten im Netzwerk? Mit der Erhöhung der Anzahl der Schichten in einem KNN, erhöht sich dessen Speicherkapazität. Die Komplexität der Muster, die ein Netzwerk lernen kann, ist abhängig von dessen Anzahl an Schichten. Wirken sich mehr Schichten und damit komplexere erlernbare Muster positiv auf das Voraussagen von Code aus?

FF 7: Erhöht die Einführung der Optimierungstechnik `Dropout` [56] die Genauigkeit von Voraussagen von Code? In Studien wie **??** und **??** hat die die Einführung von `Dropout` zur Verbesserung der Generalisierung des neuronalen Netzes geführt. Ob dies hier auch eintreten wird, soll herausgefunden werden.

FF 8: Welcher der beiden Optimierer, `Adam` oder `RMSprob`, ermöglicht eine genauere Voraussage von Code? *Adam* und `RMSprob` sind Standardoptimierer, aber welcher von ihnen eignet sich besser zum Trainieren eines KNNs auf das Voraussagen von Code?

5.1.3 Kombinationen

FF 9: Welche Kombinationen aus Vorverarbeitung, Abstraktionsart und Architektur führen zur höchsten Genauigkeit von Voraussagen von Code? Hier soll festgestellt werden, welche Vorverarbeitungsschritte sich besser für das zeichenweise oder wortweise Voraussagen von Code eignen und mit welche Architektur sie die besten Ergebnisse erzielen.

5.2 Metriken

Um die oben genannten Forschungsfragen zu beantworten, soll das trainierte KNN einen vorgegebenen, unvollständigen Codeausschnitt vervollständigen. Der künstlich erzeugte Quellcode wird anhand der folgenden Metriken bewertet.

5.2.1 Metrik 1: Levenshtein-Distanz

Die *Levenshtein-Distanz* gibt an, wie ähnlich sich zwei Zeichenketten sind. Es werden dabei die Zeichen der zweiten Zeichenkette in Betracht bezogen, die geändert werden müssten, um sie an die erste Zeichenkette anzugleichen.

In dieser Arbeit wird die Levenshtein-Distanz-Implementierung der Python-Erweiterung *Levenshtein* [23] genutzt. Dessen Funktion *ratio* nimmt zwei Zeichenfolgen als Eingabe und berechnet die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um die erste Zeichenkette in die zweite umzuwandeln. Diese absolute Zahl wird in Relation zur Anzahl der Zeichen gesetzt. Das Ergebnis ist eine Zahl zwischen 0 und 1. Der Wert 1 steht für eine völlige Übereinstimmung, 0 steht für gar keine Übereinstimmung.

Zur Bewertung des generierten Quelltextes wird der Originalcodeausschnitt und der erzeugte Code der *Levenshtein*-Funktion *ratio* übergeben. Je näher das Ergebnis an 1 ist, desto ähnlicher sind sich der Originalcode und der generierte Code, desto besser ist die Voraussage des Netzes.

Hier ein Beispielaufruf der Funktion:

```
print(ratio('Levenshtein', 'Leenshten'))
```

Das Ergebnis lautet `0.9` und bedeutet, dass sich die zwei Zeichenketten sehr ähneln.

5.2.2 Metrik 2: Zeichenkettenähnlichkeit

Metrik 2 gibt an, wie ähnlich sich zwei Zeichenketten sind. Jedes Zeichen der ersten Sequenz wird mit der zweiten verglichen. Umso mehr Zeichen gleich sind, desto ähnlicher sind sich die zwei Zeichenketten.

Zur Umsetzung wurde sich dem Python-Modul *difflib*[16] bedient. Es enthält die Funktion `ratio()`, die auf einem `SequenceMatcher`-Objekt aufgerufen wird, das mit zwei zu vergleichenden Zeichenketten instanziert wurde. `ratio` bestimmt die Ähnlichkeit der beiden und drückt sie als Gleitkommazahl zwischen 0 und 1 aus. 0 bedeutet keine Übereinstimmung, 1 hingegen bedeutet, dass beide identisch sind. Wenn T die Anzahl der Elemente beider Sequenzen ist und M die Anzahl übereinstimmender Zeichen, so ergibt sich die Ausgabe aus $2 \cdot \frac{M}{T}$.

Es folgt ein Beispiel der `difflib`-Funktion `ratio()`:

```
s = difflib.SequenceMatcher(None, 'Levenshtein', 'Leenshten')
print(s.ratio())
```

Die Ausgabe dieses Aufrufs lautet `0.9` und bedeutet eine hohe Übereinstimmung der beiden Zeichenketten.

5.2.3 Metrik 3: Difflib-lokal

Die dritte Metrik findet die längste zusammenhängende übereinstimmende Teilsequenz zwischen dem generierten Code und dem lokalen Originalcodeauschnitt siehe Abschnitt 5.4 mit Hilfe der Methode `find_longest_match(a_start, a_end, b_start, b_end)` aus dem `Python`-Modul `difflib`.

Es wird dabei der Block von Sequenz `a` von `a_start` bis `a_end` `(a[a_start:a_end])` und der Block von Sequenz `b` von `b_start` bis `b_end` `(b[b_start:b_end])` verglichen. Wenn mehrere Zeichenketten übereinstimmen, wird nur die längste zurückgegeben. Die Ausgabe enthält die Angabe des Start- und Endzeichens der längsten zusammenhängenden übereinstimmenden Teilsequenz der beiden Eingabesequenzen wie man an diesem Beispiel sehen kann:

```
s = difflib.SequenceMatcher(None, 'Levenshtein', 'Leenshten')
print(s.find_longest_match(0, len('Levenshtein'), 0, len('Leenshten')))
```

Das Ergebnis lautet Match(a=3, b=2, size=6). Es bedeutet, dass ab dem 3. Zeichen der ersten Sequenz und dem 2. Zeichen der zweiten Sequenz die längste zusammenhängende übereinstimmende Zeichenkette mit einer Länge von 6 Zeichen gefunden wurde.

Mit dieser Metrik kann herausgefunden werden, wie sehr das Netzwerk den Code aus dem tatsächlichen Originals voraussagt.

5.2.4 Metrik 4: Difflib-global

Die vierte Metrik findet wie Metrik 3 die längste zusammenhängende übereinstimmende Teilsequenz mit Hilfe der Methode `find_longest_match(a_start, a_end, b_start, b_end)`, aber vergleicht den generierten Quelltext nicht mit dem lokalen Originalcode, sondern mit dem gesamten, globalen Originalcode der Trainingsdaten aus dem jeweiligen Vorverarbeitungsschritts.

Im Gegensatz zu Metrik 3 wird hiermit überprüft, ob das Netzwerk Code aus allen Daten des jeweiligen Datensatzes betrachtet und zur Voraussage benutzt. Wenn dies zutrifft, bedeutet dass, das das Netz Muster aus verschiedenen Bereichen gelernt hat und an anderen Stellen wieder einsetzt. Wenn eine Übereinstimmung gefunden wurde, bedeutet das, dass das Netzwerk echte Variablen, Klassen und Methoden benutzt hat, was sich positiv auf die Bewertung des generierten Quellcodes auswirkt. Je mehr von Menschen erstellter Code benutzt wurde, desto besser.

5.3 Versuchsaufbau

Zur Trainieren des KNNs wurden 2 Rechner mit Windows 10 Pro (64-Bit), einem Intel Core i7-7700K á 4,2 GHz Prozessor, 32 GB RAM und jeweils einer Nvidia GeForce GTX 1080 Ti Grafikkarte mit 11 GB Speicherkapazität verwendet. Es wurden die Anaconda 4.4.0 Python-Version 3.6.1, Keras-Version 2.0.8, TensorFlow-Version 1.1.0, numpy-Version 1.12.1, CUDA® Toolkit 8.0.60, cuDNN-Version 6.0.21, Levenshtein-Version 0.12.0 und Difflib basierend auf Python 3.6.1 benutzt.

Wie in Kapitel 3, Abschnitt 3.1 beschrieben, wurden 100 Java-Projekte inklusive all ihrer Versionen von Github heruntergeladen. Danach wurden die in Kapitel 3, Abschnitt 3.2 vorgestellten Vorverarbeitungsschritte auf die Projektdateien angewandt. Die so entstandenen verschiedenen Datensätze wurden als Trainingsdaten für die Netzwerkarchitekturen aus Kapitel 4 benutzt. Während die diversen Netzwerke lernten, wurden ihre Architekturen und Gewichte gespeichert.

Nachdem alle Architekturen auf allen Datensätze trainiert wurden, wurden die letzten gespeicherten Gewichte jeder Kombination in ein separates Programm geladen, das auf bestimmte Codeausschnitte aus den Datensätzen neuen Code mit Hilfe des trainierten Netzes erzeugte.

Dazu wurde der gewünschte Codeausschnitt (`sentence`) als `String` gespeichert, indem der Startindex seiner Ursprungsdatei `text` genannt wurde und von dort aus die nächsten `maxlen_window` Zeichen gespeichert wurden. Für die Evaluierung wurde der Ausschnitt um die nachfolgenden 50 Zeichen erweitert und als `seed_completed` gespeichert. Danach wird der `String generated`, der noch weitergeschrieben werden soll, mit dem Codeausschnitt initialisiert.

```
sentence = text[start_index: start_index + maxlen_window]
seed_completed = text[start_index: start_index + 200 + maxlen_window]
generated += sentence
```

Dann wird `number_of_generations`-mal ein neues Zeichen mit Hilfe des trainierten KNNs vorausgesagt. Zuerst wird der Codeausschnitt `sentence` dem `one-hot encoding` unterzogen und damit die Eingabe `x` für das Netzwerk erstellt. Danach gibt das `model` mit der Methode `predict` seine Voraussage in Form eines `Arrays` an Wahrscheinlichkeiten aus (`preds`). Mit Hilfe der Methode `sample` wird der Index eines Zeichen aus diesen `Array` gezogen. Danach wird in dem assoziativen Feld `indices_char` das dazugehörige Zeichen nachgeschlagen und an die generierte Zeichenkette angehängt. Danach wird das erste Zeichen des Codeausschnitts `sentence` entfernt und dafür das neu erzeugte Zeichen hinten an gehangen.

```
for i in range(number_of_generations):
    x = np.zeros((1, maxlen_window, len_chars))
    for t, char in enumerate(sentence):
        x[0, t, char_indices[char]] = 1.

    preds = model.predict(x, verbose=0)[0]
    next_index = sample(preds, diversity)
    next_char = indices_char[next_index]

    generated += next_char
    sentence = sentence[1:] + next_char
```

Diese `for-Schleife` wird 3-mal aufrufen, einmal für jeder der 3 Diversitäten 0.5, 1.0 und 1.2. Die `diversity` hat Einfluss darauf, welcher Index in der Methode `sample` aus dem Wahrscheinlichkeitsarray `preds` gezogen wird. Eine niedrige Diversität bedeutet, dass Indices mit einer schlechten Wahrscheinlichkeit öfter gezogen werden als bei einer hohen Diversität. Daraus resultieren Zeichenketten, die eher oft gesehene Muster wiedergeben bzw. Zeichenketten, die aus unüblicheren Kombinationen bestehen.

```
def sample(preds, diversity):
    preds = np.asarray(preds).astype("float64")
    preds = np.log(preds) / diversity
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Die generierten Quelltexte wurden mit dem Originalcode aus dem Datensatz mit der Hilfe der im Abschnitt zuvor besprochenen Metriken bewertet. Anhand dieser Evaluierung konnten Ergebnisgrafiken erstellt werden, die sich im nächsten Abschnitt befinden.

5.4 Ergebnisse

Im Folgenden werden die Ergebnisse der Evaluierung der Quelltextvoraussagen mit Hilfe der im Abschnitt 5.2 erklärten Metriken gezeigt. Dabei sollte der folgende Quelltextausschnitts ergänzt werden:

```
@Override
public void paintIcon(Componen
```

Im Original wurde der Ausschnitt so weitergeschrieben:

```
@Override
public void paintIcon(Component c, Graphics g, int x, int y) {
    Graphics2D g2d = (Graphics2D) g.create();
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
```

Zur Evaluierung sollen nur die nächsten 50 Zeichen generiert werden. Eine exakte Voraussage würde folgendes erzeugen:

```
t c, Graphics g, int x, int y) {
    Graphics2D g2
```

Um einen guten Gesamteindruck von den Resultaten der jeweiligen Netzwerke zu bekommen, wird 5-mal der oben vorgestellte Codeausschnitt ergänzt und von diesen der beste Wert und der Median jeder Metrik zur Auswertung gespeichert. Bei jedem der 5 Durchläufe wird der Quelltext mit den 3 Diversitätsstufen 0.5, 1.0 und 1.2 erzeugt. Jedoch wird für eine Art von Visualisierung pro Durchlauf nur der Wert jeder Metrik gespeichert, der in der besten Diversitätsstufe erzielt wurde. Die beste Diversität wird anhand der kleinsten Anzahl an Abweichungen des vorausgesagten Quelltexts im Vergleich zum Original bestimmt. Diese Abweichung wird unter den Architekturen verglichen und visualisiert. Alle Best- und Medianwerte pro Optimierertyp und pro Metrik werden in den kommenden Abschnitten gezeigt. Zusätzlich werden bei jedem Netzwerk einmal beispielhaft die Abweichungen zum Original für alle 3 Diversitäten angegeben, um den Unterschied zwischen den Stufen zu sehen.

5.4.1 Vergleich der Diversitäten

In Abbildung 5.1 ist exemplarisch das Ergebnis des **1L128** Netzwerks mit dem Optimierer `Adam` und dem Optimierer `RMSprop` auf den Rohdaten für die Diversitäten 0.5, 1.0 und 1.2 gegenüber gestellt. Es werden 50 Zeichen Code vorausgesagt und zeichenweise mit dem Originalcodeausschnitt verglichen. Die x-Achse kennzeichnet den Index eines jeden Zeichens. Wenn ein Zeichen nicht mit dem Original übereinstimmt, erhöht sich y. Je näher der Graph an $y = 0$ ist, desto näher ist die Voraussage am Originalcode und desto besser ist sie. Wenn der Graph von unten links diagonal bis oben rechts verläuft, stimmt kein erzeugtes Zeichen mit dem Original überein. Weitere Abbildungen zu den Diversitäten sind im Anhang A zu finden.

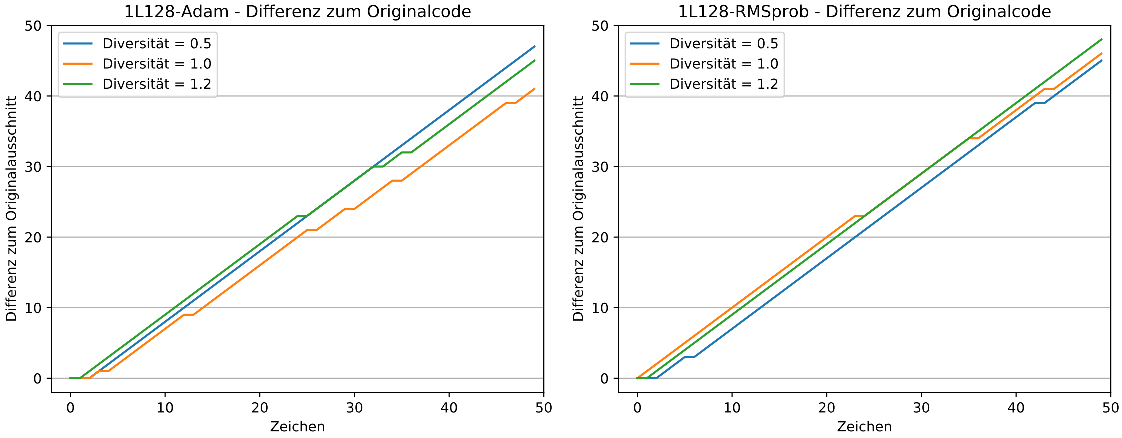

Abbildung 5.1: Vergleich **1L128**, links `Adam`, rechts `RMSprop`

Die Medianwerte der zeichenweisen Abweichung der Architekturen eines Optimierers auf den Rohdaten sind in Abbildung 5.2 dargestellt. Die restlichen Bilder können im Anhang A betrachtet werden.

5.4.2 Ergebnisse der Rohdaten

In Abbildung 5.3 bis 5.6 sind alle Ergebnisse der Rohdaten zu sehen. Pro Abbildung ist das Ergebnis einer Metrik für alle Netzwerkarchitekturen dargestellt. Es wird links der beste Wert der Metrik dem Medianwert auf der rechten gegenübergestellt.

Alle Netzwerke erreichen bei der mittleren *Levenshtein-Distanz* ähnliche Ergebnisse. Die Werte reichen von 0,26 bis 0,4 und es gibt keine großen Unterschiede zwischen der Ver-

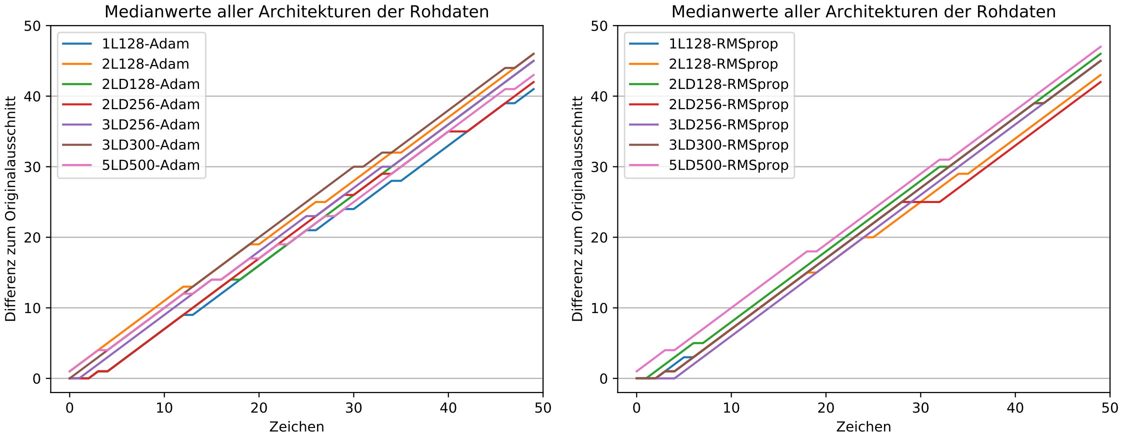

Abbildung 5.2: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

wendung des Optimierers `Adam` oder `RMSprop`. Nur bei den Bestwerten der Metrik erzielt **2LD256** `Adam` mit 0,71 einen erkennbar höheren Wert als die anderen Netze. Damit kommt es dem Originalcodeausschnitt sehr nah. Die restlichen KNNs zeigen in ihren Bestwerten keine deutlich höheren Resultate als bei den Medianwerten.

Bei den *Difflib-ratio*-Bestwerten verhält es sich ähnlich. Alle Netzwerke sind mit geringen Schwankungen gleich, nur **2LD256** `Adam` ist etwas besser. Es erreicht einen Bestwert von 0,71 und die anderen bewegen sich zwischen 0,24 und 0,38. Die Medianwerte fallen hier etwas schlechter aus als bei der *Levenshtein*-Metrik. Die Werte liegen zwischen 0,16 und 0,38. Das Netzwerke **2LD256** `Adam` erreicht dabei mit 0,71 den höchsten Wert.

Die Resultate der mittleren *Difflib-lokal*-Metrik fallen schlecht aus. Sie bewegen sich zwischen 2 und 10 übereinstimmenden Zeichen mit dem Originalcodeausschnitt. Die Netze **2LD256** `Adam` und **3LD256** `RMSprop` schneiden dabei etwas besser ab. Bei den Bestwerten fällt wieder auf, dass **2LD256** `Adam` mit 21 Zeichen viel besser ist als alle anderen. Diese haben nur geringfügig höhere Bestwerte als mittlere.

Die Median *Difflib-global*-Ergebnisse fallen bei einigen Netzwerken wesentlich besser aus als die der lokalen Metrik. Bei den mittleren Werten erreichen **3LD256** `RMSprop` und **1L128** `Adam` 26 bzw. 29 übereinstimmende Zeichen mit dem Rohdatensatz. Die nächst besseren KNNs, **2LD256** `RMSprop` und **2LD128** `Adam`, erzielen 21 bzw. 19 Zeichen. Die restlichen erreichen nur 5 bis 16 Zeichen. Die Bestwerte fallen noch viel besser aus. **1L128** `Adam` und **3LD300** `RMSprop` erzielen 47 bzw. 43 übereinstimmende Zeichen mit den gesamten Rohdaten. Die KNNs **2L128** `Adam`, **3LD256** `Adam`, **3LD300** `Adam` und **5LD500** mit `Adam` und `RMSprop` erzielten die schlechtesten Ergebnisse. Sie liegen zwischen 6 und 7 Zeichen. Die restlichen Netzwerke zeigen hingegen wieder gute Resultate zwischen 20 und 34 übereinstimmenden Zeichen.

Die Resultate aller Architekturen sind über aller Median-Metriken sehr ähnlich. Keins sticht hervor. Dies ändert sich bei den Bestwerten der Metriken *Levenshtein*, *Difflib-ratio* und *Difflib-lokal*. Dort wird sichtbar, dass das KNN **2LD256** `Adam` besser als anderen neuronalen Netze abschneidet.

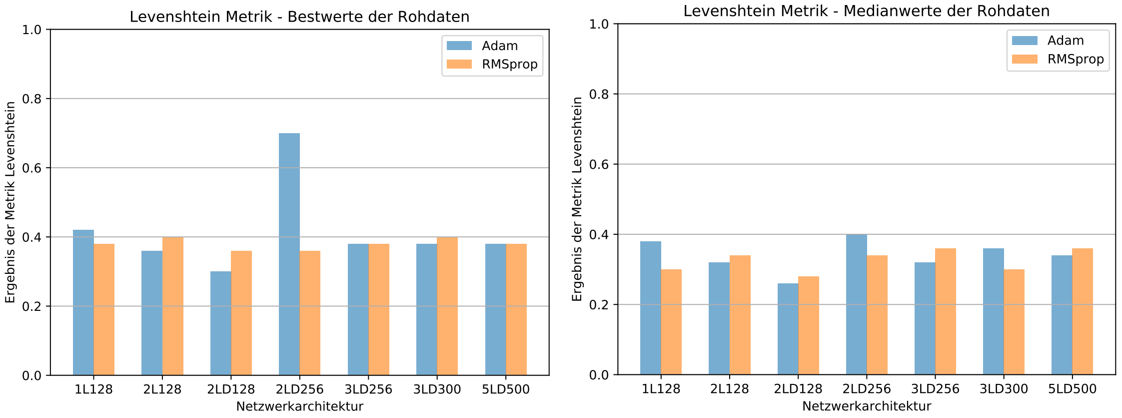

Abbildung 5.3: Bestes und mittleres *Levenshtein*-Ergebnis pro Architektur über 5 Durchläufe auf den Rohdaten

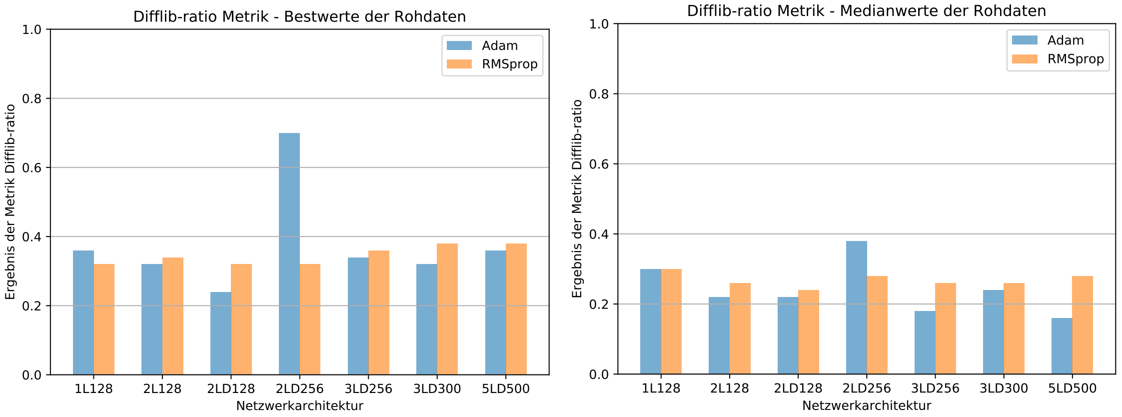

Abbildung 5.4: Bestes und mittleres *Difflib-ratio*-Ergebnis pro Architektur über 5 Durchläufe auf den Rohdaten

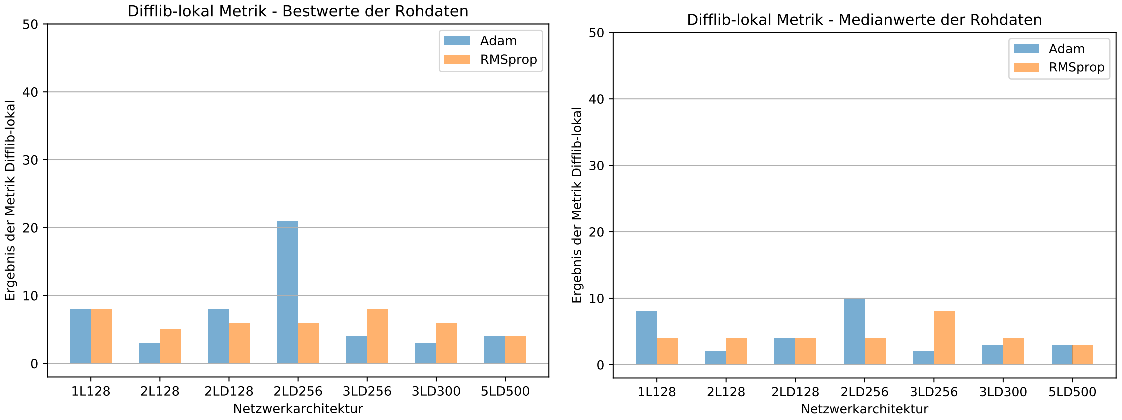

Abbildung 5.5: Bestes und mittleres *Difflib-lokal*-Ergebnis pro Architektur über 5 Durchläufe auf den Rohdaten

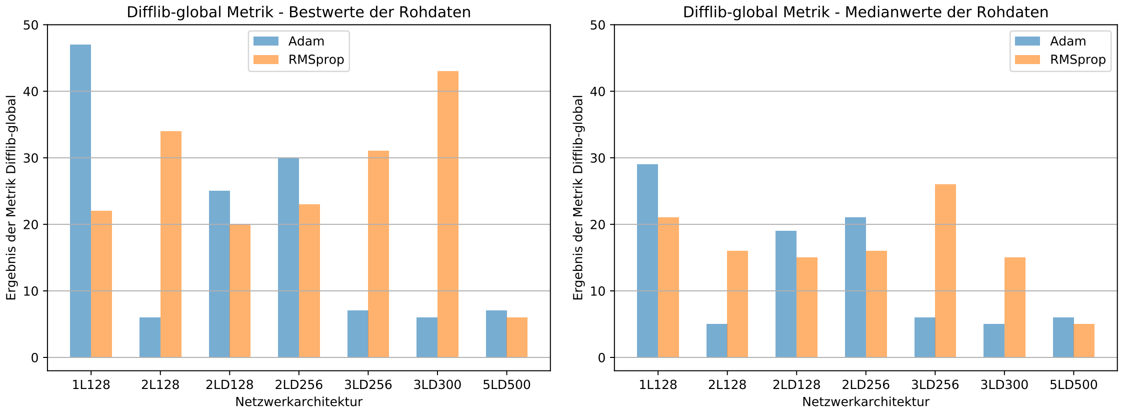

Abbildung 5.6: Bestes und mittleres *Difflib-global*-Ergebnis pro Architektur über 5 Durchläufe auf den Rohdaten

5.4.3 Ergebnisse nach dem Entfernen der Kommentare

Die Ergebnisse nach dem Entfernen der Kommentare befinden sich in Abbildung 5.7 bis 5.10. Jede Abbildung enthält das Ergebnis einer Metrik für alle Netzwerkarchitekturen. Es sind jeweils Abbildungen nebeneinander. Auf der linken Seite ist der beste Wert der jeweiligen Metrik und auf der rechten der Medianwert.

Die Ergebnisse nach dem Entfernen der Kommentare haben sich gegenüber den Rohdaten verbessert. Alle Netzwerke wiesen bei Betrachtung der mittleren *Levenshtein-Distanz* über 5 Durchgänge kleine Verbesserungen auf. Besonders das Ergebnis von **2LD256** `Adam` hat sich am meisten erhöht und ist mit 0,68 das beste. Bei den Bestwerten der Metrik ist **2LD256** `Adam` nun nicht mehr allein die beste Architektur (siehe Rohdaten). Sie kommt zwar auf 0,72, aber die Netze **2L128** `RMSprop` und **3LD300** `RMSprop` erzielen ebenso gute Werte von 0,77 bzw. 0,56. Die anderen Netzwerke bewegen sich zwischen 0,36 und 0,46.

Fast alle Netzwerke haben sich bezüglich des mittleren `ratio`-Ergebnisses nach dem Entfernen der Kommentare verbessert. **1L128** hat sich auf 0,64 gesteigert und ist damit wesentlich besser als alle anderen. Die Bestwerte der *Difflib-ratio*-Metrik verhalten sich ähnlich. Jedoch stechen die KNNs **2L128** `RMSprop`, **2LD256** `Adam` und **3LD300** `RMSprop` mit 0,77 bzw. 0,72 bzw. 0,56 heraus.

2LD256 erzielt bei den mittleren `Difflib-lokal`-Werten 29 übereinstimmenden Zeichen zwischen generierten Code und Originalcode das beste Resultat. Die restlichen Netzwerke haben sich etwas verbessert im Vergleich zu den Rohdaten. Erneut zeigen die Netze **2L128** `RMSprop`, **2LD256** `Adam` und **3LD300** `RMSprop` die besten Ergebnisse der Bestwerte der Metrik. Sie erreichen zwischen 22 und 37 Zeichen.

Alle KNNs haben sich sehr bei den mittleren `Difflib-global`-Werte bezüglich der Rohdatn verändert. Sie schwanken sehr und kommen auf 5 bis 46. Die besten Werte der `Difflib-global`-Metrik haben sich ebenfalls teils verbessert und teils verschlechtert. Allerdings wurde von **2LD256** `Adam` der Maximalwert von 50 Zeichen erreicht, was bedeutet, das eine vollständige korrekte Voraussage getroffen wurde.

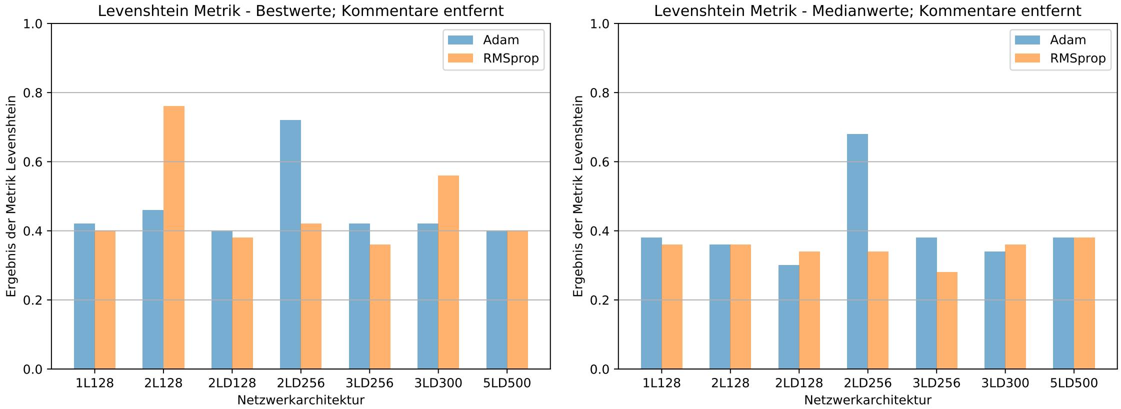

Abbildung 5.7: Bestes und mittleres *Levenshtein*-Ergebnis pro Architektur über 5 Durchläufe; Kommentare entfernt

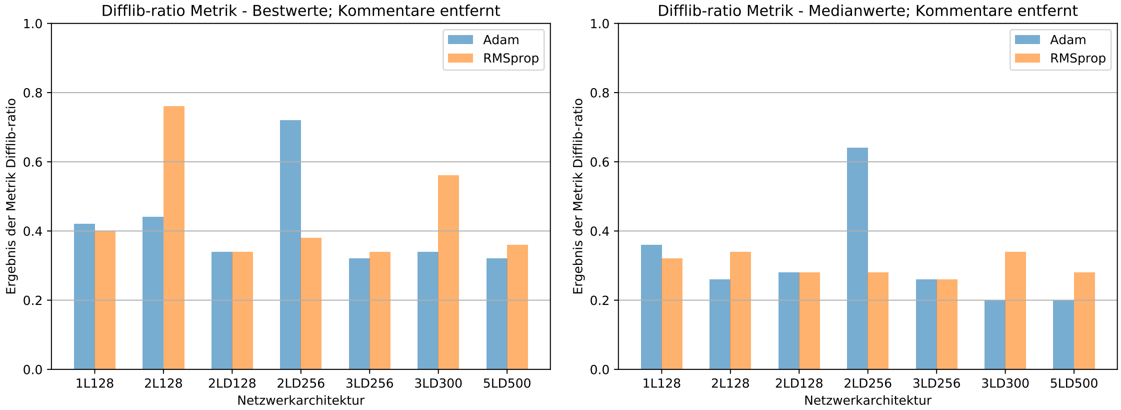

Abbildung 5.8: Bestes und mittleres *Difflib-ratio*-Ergebnis pro Architektur über 5 Durchläufe; Kommentare entfernt

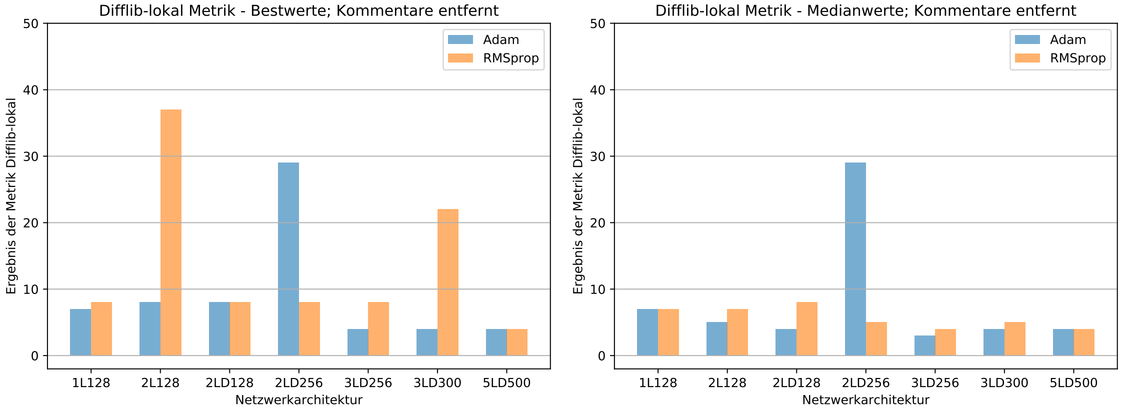

Abbildung 5.9: Bestes und mittleres *Difflib-lokal*-Ergebnis pro Architektur über 5 Durchläufe; Kommentare entfernt

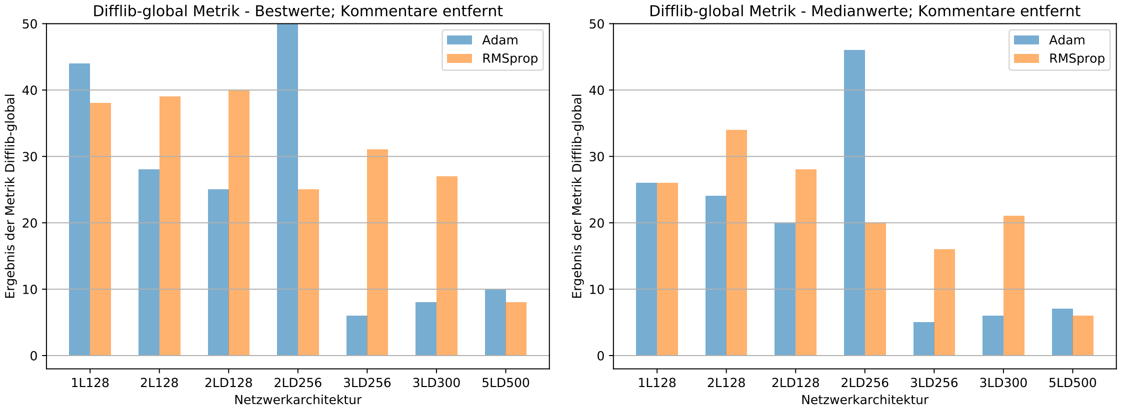

Abbildung 5.10: Bestes und mittleres *Difflib-global*-Ergebnis pro Architektur über 5 Durchläufe; Kommentare entfernt

5.4.4 Ergebnisse nach dem Entfernen der Einrückungen und Leerzeilen

Die Ergebnisse der 4 Metriken nach dem Entfernen der Einrückungen und Leerzeilen sind in Abbildung 5.11 bis 5.14. Das Ergebnis einer Metrik ist in je einer Abbildung für alle Netzwerkarchitekturen dargestellt. Links ist der beste Wert der jeweiligen Metrik und rechts der Medianwert.

Alle Netzwerke erzielen bei der Medianwerten der *Levenshtein*-Metrik ähnliche Ergebnisse wie auf den Rohdaten. Sie haben sich teilweise ein wenig verbessert und teilweise sogar etwas verschlechtert. Die Spanne zwischen den besten und den schlechtesten Resultaten beginnt bei 0,28 und endet bei 0,38. Die Bestwerte der *Levenshtein-Distanz* haben sich gegenüber den Rohdaten wenig verändert und bewegen sich zwischen 0,34 und 0,38. Nur die KNNs **2LD128** `RMSprop` und **2LD256** `RMSprop` sind mit 0,5 bzw. 0,54 minimal besser als die anderen.

Bei der *Difflib-ratio*-Metrik blieb der Spanne der mittleren Ergebnisse ungefähr gleich, jedoch ist diesmal **2LD256** `Adam` schlechter als **2LD256** `RMSprop`. Letzteres erreicht mit 0,34 den besten Wert aller Netzwerke. Die restliche Spanne reicht von 0,16 bis 0,26. Die Bestwerte der *Difflib-ratio*-Metrik haben sich größtenteils etwas verbessert. **2LD128** `RMSprop` und **2LD256** `RMSprop` erzielen ein Ergebnis von 0,44 bzw. 0,5. Die anderen Netzwerke bewegen sich zwischen 0,18 und 0,3. **2LD256** `Adam` ist nicht mehr mit Abstand das beste Netzwerk wie in den Rohdaten. Nach dem Entfernen der Einrückungen und Leerzeilen ist es nur noch das drittbeste.

Die mittleren *Difflib-lokal*-Werte sind wie bei den Rohdaten sehr niedrig und haben sich sogar verschlechtert. Sie sind nun alle annähernd gleich. Es treten nur noch Werte zwischen 2 und 4 auf. Nur die Netzwerke **2LD128** `RMSprop` und **2LD256** `Adam` sowie `RMSprop` haben sich in den Bestwerten der Metrik merklich verbessert und erreichen bis zu 18 übereinstimmende Zeichen. Die restlichen sind sehr niedrig und haben sich teilweise sogar verschlechtert. `2LD256 Adam` war zuvor die beste Architektur, doch nun ist sie die drittbeste wie in den Bestwerten der *Difflib-ratio*-Metrik.

Die 4 Architekturtypen **2L128**, **2LD128**, **2LD256** (mit je `Adam` und `RMSprop`) und **3LD300** `RMSprop` haben sich deutlich in ihren mittleren *Difflib-global*-Werten verbessert. Der Minimalwert dieser liegt bei 20 übereinstimmenden Zeichen und der Maximalwert bei 27 Zeichen. Die `Adam`-Versionen der Netze **3LD256**, **3LD300** sowie beide Versionen von **5LD500** erzielen mit jeweils 5 Zeichen die schlechtesten Resultate. Die *Difflib-global*-Bestwerte schwanken sehr. Die Spanne erstreckt sich von 5 bis 46 übereinstimmenden Zeichen mit dem gesamten Datensatz. **2L128** `Adam` hat sich von 6 auf 46 Zeichen ersichtlich erhöht. Das Netz **5LD500** kommt mit beiden Optimierern nur auf 5 bzw. 6 Zeichen. **3LD256** und **3LD300**, jeweils mit `Adam`, erreichen ähnliche Werte (6 bis 7). Die restlichen Architekturen führen zu guten bis sehr guten Resultaten zwischen 25 und 46 Zeichen.

Die Ergebnisse haben sich im Gegensatz zu den Rohdaten wenig verändert. Der Sprung von den Rohdatenergebnissen zu den Resultaten nach dem Entfernen der Kommentare war beachtlicher als beim Entfernen der Einrückungen und Leerzeilen. Das Netzwerk **5LD500** ist bei den *Difflib-lokal* und *-global*-Metriken immer das schlechteste und liefert sehr niedrige Ergebnisse.

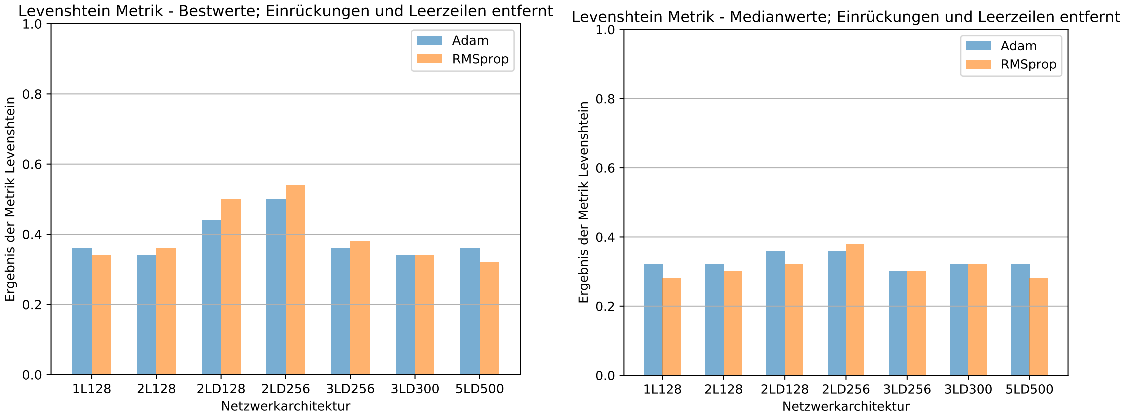

Abbildung 5.11: Bestes und mittleres *Levenshtein*-Ergebnis des Codeausschnitts pro Architektur über 5 Durchläufe; Einrückungen und Leerzeilen entfernt

5.4.5 Ergebnisse nach dem Entfernen der Kommentare, Einrückungen und Leerzeichen

Die Ergebnisse nach dem Entfernen der Kommentare, Einrückungen und Leerzeichen sind in Abbildung 5.15 bis 5.18 visualisiert. Jede Abbildung enthält eine Metrik für alle Netzwerkarchitekturen. Auf der linken Seite ist die Abbildung der besten Werte der jeweiligen Metrik und auf der rechten die Abbildung der Medianwerte über 5 Durchgänge.

Bei Betrachtung der mittleren *Levenshtein-Distanz* fällt auf, dass das KNN **2LD256** wesentlich besser mit `RMSprop` abschneidet im Vergleich zu den Rohdaten und den Rohdaten ohne Kommentare. Es erreicht mit 0,68 den besten Wert aller Architekturen. Die restli-

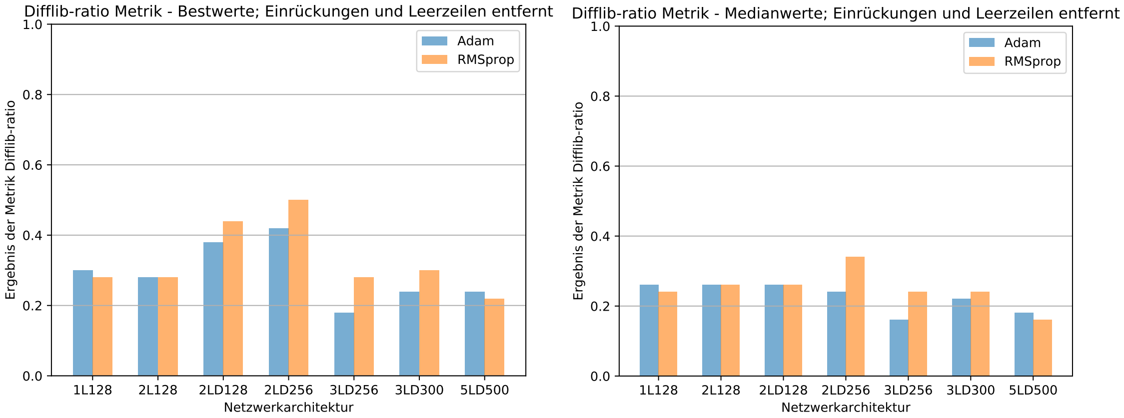

Abbildung 5.12: Bestes und mittleres *Difflib-ratio*-Ergebnis pro Architektur über 5 Durchläufe; Einrückungen und Leerzeilen entfernt

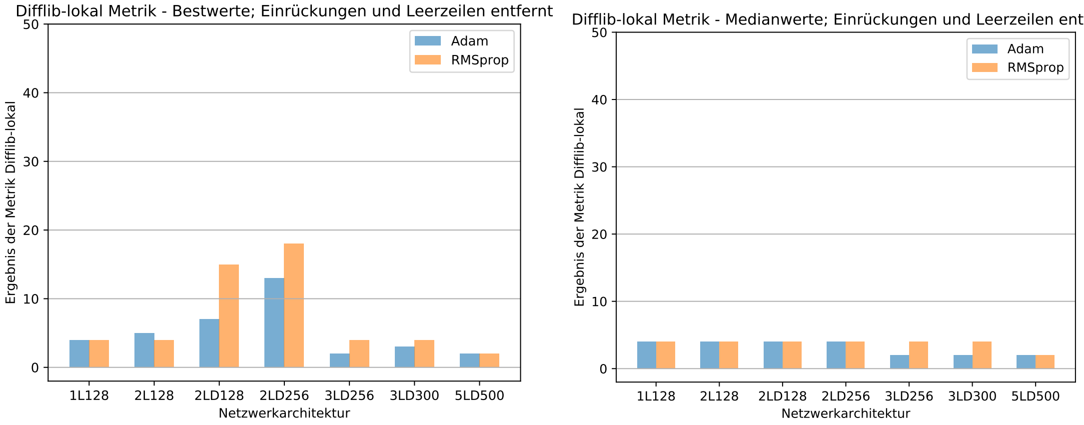

Abbildung 5.13: Bestes und mittleres *Difflib-lokal*-Ergebnis Architektur über 5 Durchläufe; Einrückungen und Leerzeilen entfernt

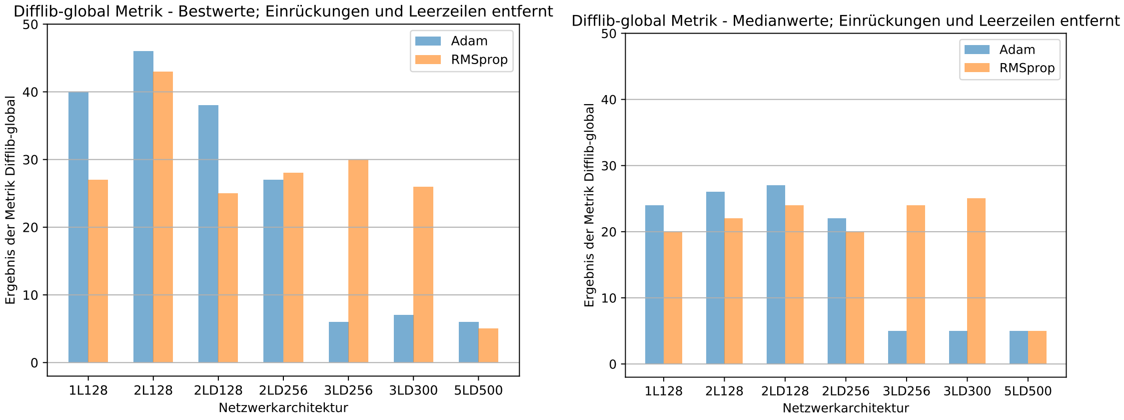

Abbildung 5.14: Bestes und mittleres *Difflib-global*-Ergebnis pro Architektur über 5 Durchläufe; Einrückungen und Leerzeilen entfernt

chen zeigen Resultate zwischen 0,22 und 0,46. **1L128**, **3LD256** und **3LD300** (alle mit `Adam`) haben sich gegenüber den oben genannten Datensätzen leicht verschlechtert. Das

Netzwerk **5LD500** hat sich sogar mit beiden Optimierern verschlechtert. Alle anderen haben sich verbessert. Bei **2LD128**, **2LD256**, **3LD256** und **3LD300** stellt sich `RMSprop` als besserer Optimierer heraus. Die Bestwerte der *Levenshtein*-Metrik erreichen **2L128** `Adam`, **2LD128** `RMSprop` und **2LD256** `RMSprop`, wobei letzteres mit 0,68 wieder das beste ist.

Ein ähnliches Muster lässt sich bei den mittleren *difflib ratio*-Werten beobachten. Nach dem Entfernen der Kommentare, Einrückungen und Leerzeilen weisen **2L128** `Adam`, **2LD128** `RMSprop` und **2LD256** `RMSprop` wieder die besten Werte auf. **2LD256 RMSprop** erzielt mit 0,68 den Höchstwert dieses Vergleichs. Bei den Bestwerten dieser Metrik sind die eben drei genannten Netzwerke, außer dass **2L128** nun mit **RMSprop** besser ist als mit `Adam`, wieder die besten und erreichen 0,46 bis 0,7. Diesmal erlangen alle außer **1L128** mit dem Optimierer `RMSprop` bessere Ergebnisse als mit `Adam`.

Außer für **2LD128** `RMSprop` und **2LD256**, ebenfalls mit `RMSprop`, bewegt sich die Spanne der mittleren Ergebnisse nur zwischen 2 und 6 übereinstimmenden Zeichen der *Difflib-lokal*-Metrik. Dafür erreichen die anderen zwei KNNs 13 (**2LD128**) bzw. 29 Zeichen. Die Bestwerte von *Difflib-lokal* schwanken zwischen 2 und 29 Zeichen. **2L128** und **2LD256**, jeweils mit `RMSprop`, erzielten beide den Höchstwert. Das drittbeste Netzwerk ist **2LD128** `RMSprop` mit 13 übereinstimmenden Zeichen mit den Originalcodeausschnitt.

Bei den mittleren *Difflib-global*-Werten lassen sich insgesamt gute Ergebnisse erkennen. Außer **3LD256** `Adam`, **3LD300** `Adam` und **5LD500** `RMSprop` und `Adam` reichen die Ergebnisse von 18 bis 27 übereinstimmende Zeichen mit dem Datensatz. Die weitere Ausnahme hierbei ist **2LD256** `RMSprop` mit 42 Zeichen. Das KNN **2LD256** `RMSprop` kommt bei den Bestwerten der *Difflib-global*-Metrik auf den Maximalwert von 50 übereinstimmenden Zeichen. Ihm ist somit eine vollständig korrekte Wiedergabe von 50 Zeichen aus dem Datensatz, nicht dem Originalausschnitt, gelungen. Den Netzwerken **2L128** mit beiden Optimierern, **2LD128** `Adam`, **2LD256** `Adam`, **3LD256** `RMSprop` und **3LD300** `RMSprop` haben ebenfalls gute Voraussagen von bis zu 39 übereinstimmenden Zeichen erzielt.

Insgesamt hat sich gezeigt, dass der Optimierer `RMSprop` meist bessere Ergebnisse liefert als sein Gegenstück. **5LD500** zeigt meistens die schlechtesten Resultate.

5.4.6 Ergebnisse nach dem Entfernen der Kommentare, Einrückungen und Leerzeichen sowie der Namensvereinfachung

In Abbildung 5.19 bis 5.22 sind die Ergebnisse nach dem Entfernen der Kommentare, Einrückungen und Leerzeichen sowie der Namensvereinfachung sind veranschaulicht. Pro Abbildung ist eine Metrik für alle Netzwerkarchitekturen dargestellt. Links sind die besten Werte jeder Metrik und auf der rechten die Medianwerte.

Das Ergebnis der mittleren *Levenshtein-Distanz* für die Netzwerke **2L128** (beide Optimierer), **2LD128** `RMSprop` und **2LD256** (beide) haben sich im Vergleich zum vorherigen

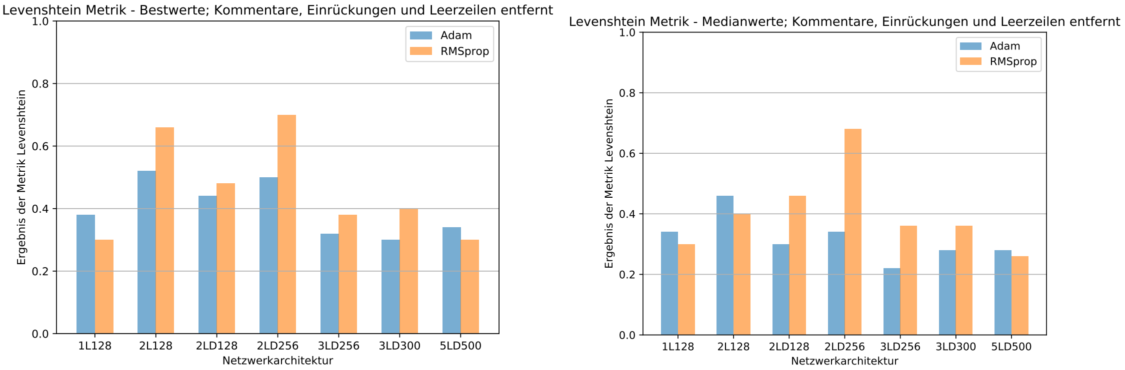

Abbildung 5.15: Bestes und mittleres *Levenshtein*-Ergebnis pro Architektur über 5 Durchläufe; Kommentare, Einrückungen und Leerzeilen entfernt

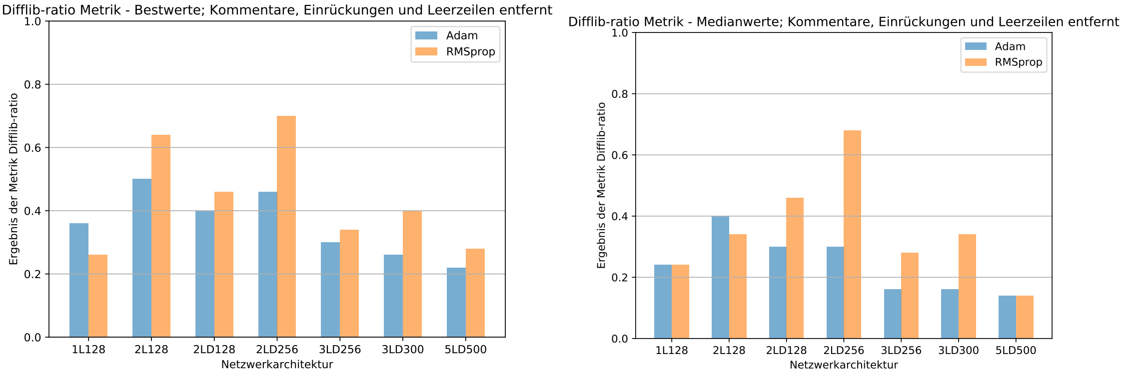

Abbildung 5.16: Bestes und mittleres *Difflib-ratio*-Ergebnis pro Architektur über 5 Durchläufe; Kommentare, Einrückungen und Leerzeilen entfernt

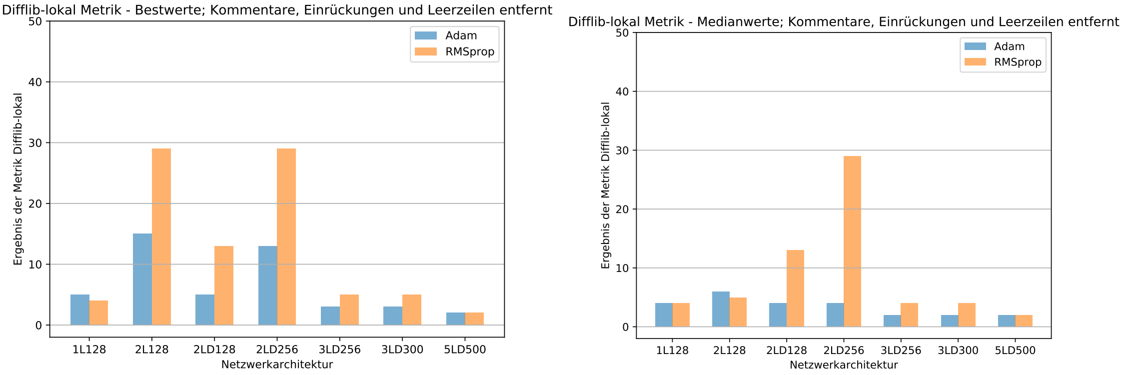

Abbildung 5.17: Bestes und mittleres *Difflib-lokal*-Ergebnis pro Architektur über 5 Durchläufe; Kommentare, Einrückungen und Leerzeilen entfernt

Vorverarbeitungsschritt verschlechtert. Sie waren zuvor die besten drei Architekturtypen. Die anderen Netzwerke haben sich teils etwas verbessert und teils leicht verschlechtert. **3LD300** `RMSprop` ist nun mit 0,46 das beste. Im Gegensatz dazu hatte im vorangegangenen Schritt `2LD256 RMSprop` mit 0,68 das höchste Ergebnis. Jedoch haben sich mehr Netzwerkarchitekturen bei Betrachtung der Bestwerte der *Levenshtein-Distanz* verbessert, im Vergleich zum Median und zum letzten Vorverarbeitungsschritt. Hier liegt der

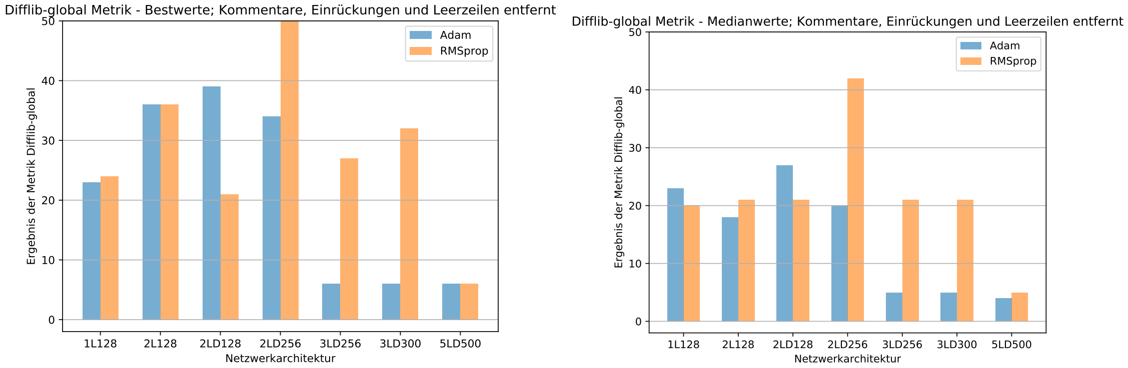

Abbildung 5.18: Bestes und mittleres *Difflib-global*-Ergebnis pro Architektur über 5 Durchläufe; Kommentare, Einrückungen und Leerzeilen entfernt

Höchstwert bei 0,66. Diesen erzielte das Netz **2LD256** `RMSprop`.

Fast alle Netzwerke haben sich von der mittleren *Levenshtein*-Metrik bis zur mittleren *Difflib ratio*-Metrik verschlechtert, verglichen mit dem vorherigen Vorverarbeitungsschritt sogar sehr. Die Ergebnisse reichen nur von 0,1 bis 0,38. Am besten ist das Netz **3LD300** mit dem Optimierer `RMSprop`. Die Bestwerten der *Difflib ratio*-Metrik sind höher und bewegen sich zwischen 0,16 und 0,66. Bei **2LD256**, was das höchste Resultat erzielt, und **3LD300** schneidet `RMSprop` wesentlich besser als `Adam` ab.

Anders als im Datensatz nach dem Entfernen der Kommentare, Einrückungen und Leerzeichen sticht bei den hier betrachteten *Difflib ratio*-Medianwerten kein KNN hervor. Alle erreichen nur niedrige Ergebnisse von 1 bis maximal 6 Zeichen. Die Bestwerte haben sich zwar gebessert, aber es hebt sich immer noch kein neuronales Netz ab. Die Spanne deckt nur Werte von 2 bis 12.

Die Verteilung der mittleren `Difflib-global`-Werte gleicht der des vorherigen Vorverarbeitungsschritt. Allerdings fallen die höchsten Werte hier geringer aus und die Resultate ähneln sich sehr. **2L128** erzielt den Höchstwert von 23 übereinstimmenden Zeichen. Die Netze **2LD256**, **3LD256** und **3LD300** kommen mit dem Optimierer `RMSprop` auf deutlich bessere Werte als mit `Adam`. **5LD500** ist mit beiden schlecht. Es erreicht nur 4 Zeichen. Bei den Bestwerten der Metrik sind **1L128** `RMSprop`, **2L128** mit beiden Optimierern und **3LD300** `RMSprop` die besten Architekturen. Sie konnten 44 bzw. 37 (`Adam`) und 39 (`RMSprop`) bzw. 36 gemeinsame Zeichen voraussagen. Dennoch erzielten die Architekturen des vorangegangen Schritts bessere Höchstwerte.

Insgesamt wird beim Umbenennen deutlich, dass auch hier `RMSprop` öfter höhere Resultate als sein Gegenspieler erreicht, wie beim Datensatz nach dem Entfernen der Kommentare, Einrückungen und Leerzeilen. **5LD500** ist durchgängig das schlechteste KNN.

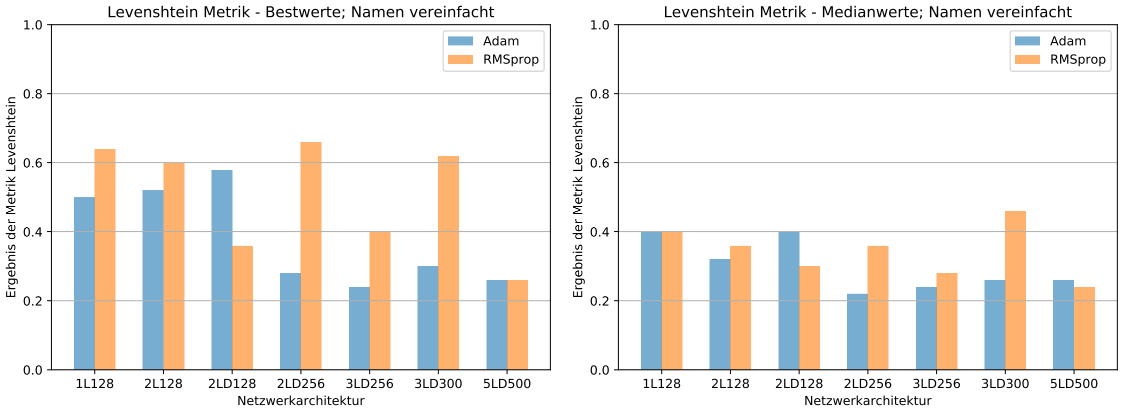

Abbildung 5.19: Bestes und mittleres *Levenshtein*-Ergebnis pro Architektur über 5 Durchläufe; Namen vereinfacht

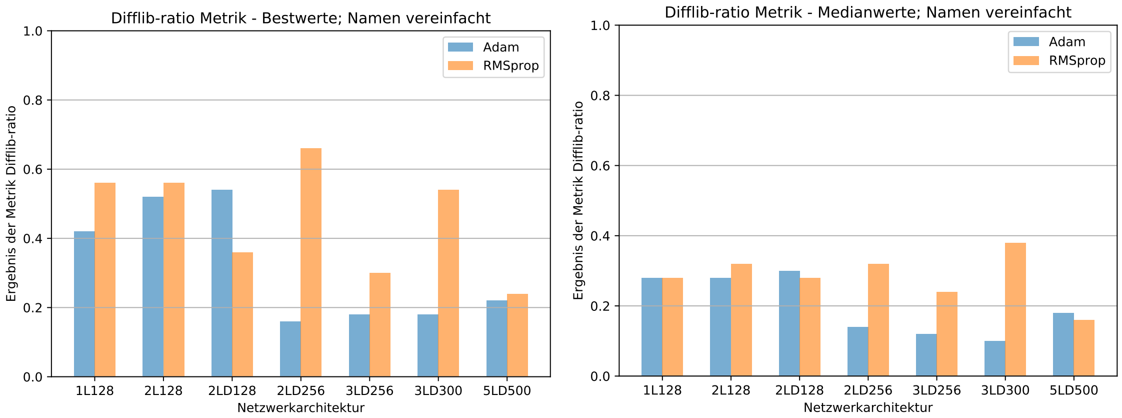

Abbildung 5.20: Bestes und mittleres *Difflib-ratio*-Ergebnis pro Architektur über 5 Durchläufe; Namen vereinfacht

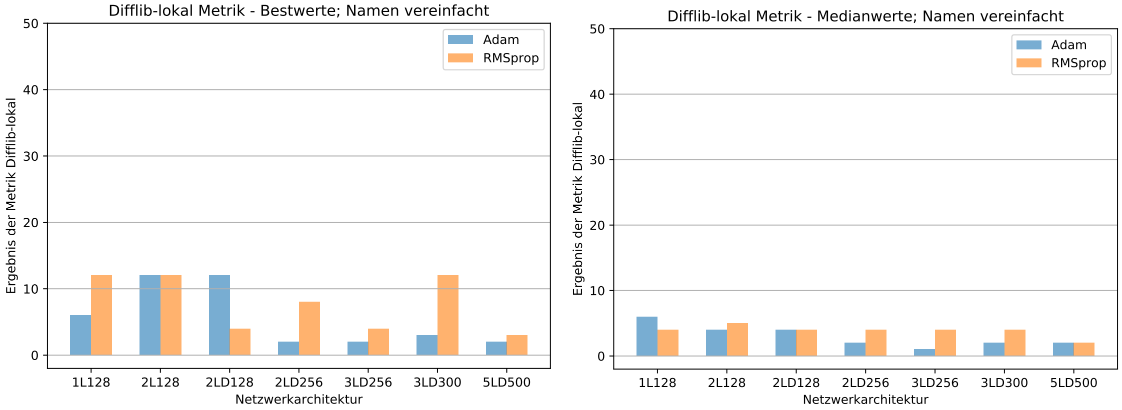

Abbildung 5.21: Bestes und mittleres *Difflib-lokal*-Ergebnis pro Architektur über 5 Durchläufe; Namen vereinfacht

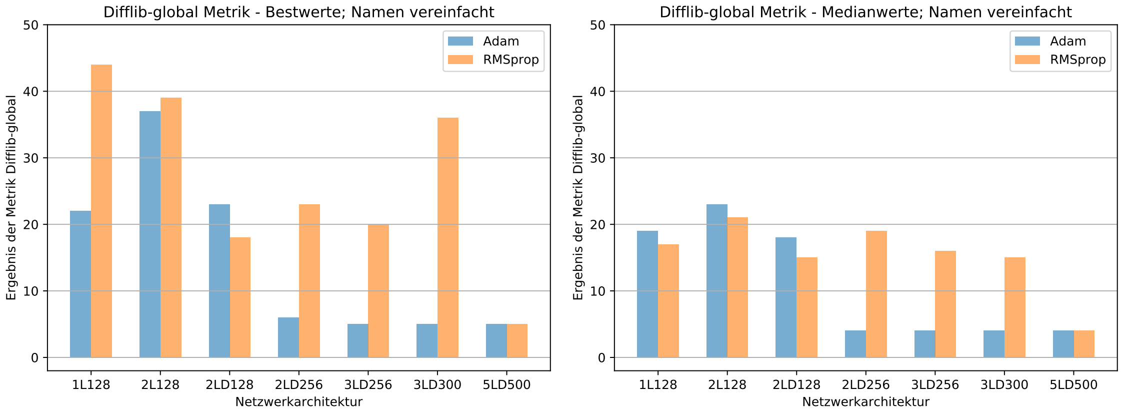

Abbildung 5.22: Bestes und mittleres *Difflib-global*-Ergebnis pro Architektur über 5 Durchläufe; Namen vereinfacht

5.4.7 Ergebnisse aller Vorverarbeitungsschritte

In den Tabellen Abbildung 5.1 bis 5.8 werden die Resultate aller KNNs mit ihren jeweiligen Optimierern pro Metrik zusammengefasst. Die Zeile **Bestwert** gibt den Bestwert der jeweiligen Spalte wieder.

	1L128		**2L128**		**2LD128**		**2LD256**		**3LD256**		**3LD300**		**5LD500**	
	Adam	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**
Rohdaten	0.3	0.3	0.22	0.26	0.22	0.24	0.38	0.28	0.18	0.26	0.24	0.26	0.16	0.28
Kom	0.36	0.32	0.26	0.34	0.28	0.28	0.64	0.28	0.26	0.26	0.2	0.34	0.2	0.28
Ein	0.26	0.24	0.26	0.26	0.26	0.26	0.24	0.34	0.16	0.24	0.22	0.24	0.18	0.16
Kom+Ein	0.24	0.24	0.4	0.34	0.3	0.46	0.3	0.68	0.16	0.28	0.16	0.34	0.14	0.14
Namen	0.28	0.28	0.28	0.32	0.3	0.28	0.14	0.32	0.12	0.24	0.1	0.38	0.18	0.16
Bestwert	0.36	0.32	0.4	0.34	0.3	0.46	0.64	0.68	0.26	0.28	0.24	0.38	0.18	0.28

Abkürzungen: **RMS** - **RMS**prop; **Kom** - **Kom**mentare entfernt; **Ein** - **Ein**rückungen und Leerzeilen entfernt; **Namen** - **Namen** vereinfacht

Tabelle 5.1: Medianwerte der *Levenshtein*-Ergebnisse aller Architekturen auf allen Datensätzen

	1L128		**2L128**		**2LD128**		**2LD256**		**3LD256**		**3LD300**		**5LD500**	
	Adam	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**
Rohdaten	0.42	0.38	0.36	0.4	0.3	0.36	0.7	0.36	0.38	0.38	0.38	0.4	0.38	0.38
Kom	0.42	0.4	0.46	0.76	0.4	0.38	0.72	0.42	0.42	0.36	0.42	0.56	0.4	0.4
Ein	0.36	0.34	0.34	0.36	0.44	0.5	0.5	0.54	0.36	0.38	0.34	0.34	0.36	0.32
Kom+Ein	0.38	0.3	0.52	0.66	0.44	0.48	0.5	0.7	0.32	0.38	0.3	0.4	0.34	0.3
Namen	0.5	0.64	0.52	0.6	0.58	0.36	0.28	0.66	0.24	0.4	0.3	0.62	0.26	0.26
Bestwert	0.42	0.64	0.52	0.76	0.58	0.5	0.72	0.7	0.42	0.38	0.42	0.62	0.4	0.4

Abkürzungen: **RMS** - **RMS**prop; **Kom** - **Kom**mentare entfernt; **Ein** - **Ein**rückungen und Leerzeilen entfernt; **Namen** - **Namen** vereinfacht

Tabelle 5.2: Bestwerte der *Levenshtein*-Ergebnisse aller Architekturen auf allen Datensätzen

	1L128		**2L128**		**2LD128**		**2LD256**		**3LD256**		**3LD300**		**5LD500**	
	Adam	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**
Rohdaten	0.38	0.3	0.32	0.34	0.26	0.28	0.4	0.34	0.32	0.36	0.36	0.3	0.34	0.36
Kom	0.38	0.36	0.36	0.36	0.3	0.34	0.68	0.34	0.38	0.28	0.34	0.36	0.38	0.38
Ein	0.32	0.28	0.32	0.3	0.36	0.32	0.36	0.38	0.3	0.3	0.32	0.32	0.32	0.28
Kom+Ein	0.34	0.3	0.46	0.4	0.3	0.46	0.34	0.68	0.22	0.36	0.28	0.36	0.28	0.26
Namen	0.4	0.4	0.32	0.36	0.4	0.3	0.22	0.36	0.24	0.28	0.26	0.46	0.26	0.24
Bestwert	0.4	0.4	0.46	0.4	0.4	0.46	0.68	0.68	0.38	0.36	0.36	0.46	0.38	0.38

Abkürzungen: **RMS - RMS**prop; **Kom - Kom**mentare entfernt; **Ein - Ein**rückungen und Leerzeilen entfernt; **Namen - Namen** vereinfacht

Tabelle 5.3: Medianwerte der *Difflib-ratio*-Ergebnisse aller Architekturen auf allen Datensätzen

	1L128		**2L128**		**2LD128**		**2LD256**		**3LD256**		**3LD300**		**5LD500**	
	Adam	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**
Rohdaten	0.36	0.32	0.32	0.34	0.24	0.32	0.7	0.32	0.34	0.36	0.32	0.38	0.36	0.38
Kom	0.42	0.4	0.44	0.76	0.34	0.34	0.72	0.38	0.32	0.34	0.34	0.56	0.32	0.36
Ein	0.3	0.28	0.28	0.28	0.38	0.44	0.42	0.5	0.18	0.28	0.24	0.3	0.24	0.22
Kom+Ein	0.36	0.26	0.5	0.64	0.4	0.46	0.46	0.7	0.3	0.34	0.26	0.4	0.22	0.28
Namen	0.42	0.56	0.52	0.56	0.54	0.36	0.16	0.66	0.18	0.3	0.18	0.54	0.22	0.24
Bestwert	0.42	0.56	0.52	0.76	0.54	0.46	0.72	0.7	0.34	0.36	0.34	0.56	0.36	0.38

Abkürzungen: **RMS - RMS**prop; **Kom - Kom**mentare entfernt; **Ein - Ein**rückungen und Leerzeilen entfernt; **Namen - Namen** vereinfacht

Tabelle 5.4: Bestwerte der *Difflib-ratio*-Ergebnisse aller Architekturen auf allen Datensätzen

5.5 Diskussion

Welche Ergebnisse die Vorverarbeitungsschritte aus Kapitel 3 und Netzwerkarchitekturen aus Kapitel 4 beim Voraussagen von Code geliefert haben, werden hier diskutiert. Die dazu in Abschnitt 5.1 definierten Forschungsfragen werden mit den Metriken aus Abschnitt 5.2 beantwortet. Die Betrachtungen lassen sich in die 3 Gebiete Vorverarbeitung, Netzwerkarchitekturen und Kombinationen aus Vorverarbeitungsschritten und Architekturen einteilen.

5.5.1 Vorverarbeitung

Durch das Entfernen der Kommentare ist der Code wesentlich schlanker geworden. Somit fielen viele Muster weg, die sonst von jedem KNN hätten gelernt werden müssen. Dadurch blieb mehr Lernkapazität für das Erlernen des eigentlichen Quellcodes übrig. So mussten beim Vektorisieren (siehe Abschnitt 3.3) weniger Textsequenzen erstellt werden, die den gleichen ausführbaren Code enthielten. Dadurch wurde das Lernmaterial zwar insgesamt weniger, aber gehaltvoller. Es ist möglich, dass die Netzwerke durch den relativ hohen Anteil von Kommentaren nicht nur lernen mussten, wie Code aussieht, sondern auch natürliche Sprache und dadurch insgesamt nicht ganz so gut abschnitten wie ihre Kollegen in den anderen Vorverarbeitungsschritten. Fast alle Netzwerke haben sich stark verbessert bzw. sind zumindest auf dem gleichen Niveau geblieben. Das Entfernen der Kommentare erhöht die Genauigkeit der Codevoraussagen sehr.

	1L128		**2L128**		**2LD128**		**2LD256**		**3LD256**		**3LD300**		**5LD500**	
	Adam	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**
Rohdaten	8	4	2	4	4	4	10	4	2	8	3	4	3	3
Kom	7	7	5	7	4	8	29	5	3	4	4	5	4	4
Ein	4	4	4	4	4	4	4	4	2	4	2	4	2	2
Kom+Ein	4	4	6	5	4	13	4	29	2	4	2	4	2	2
Namen	6	4	4	5	4	4	2	4	1	4	2	4	2	2
Bestwert	8	7	6	7	4	13	29	29	3	8	4	5	4	4

Abkürzungen: **RMS** - **RMS**prop; **Kom** - **Kom**mentare entfernt; **Ein** - **Ein**rückungen und Leerzeilen entfernt;
Namen - **Namen** vereinfacht

Tabelle 5.5: Medianwerte der *Difflib-lokal*-Ergebnisse aller Architekturen auf allen Datensätzen

	1L128		**2L128**		**2LD128**		**2LD256**		**3LD256**		**3LD300**		**5LD500**	
	Adam	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**	**Adam**	**RMS**
Rohdaten	8	8	3	5	8	6	21	6	4	8	3	6	4	4
Kom	7	8	8	37	8	8	29	8	4	8	4	22	4	4
Ein	4	4	5	4	7	15	13	18	2	4	3	4	2	2
Kom+Ein	5	4	15	29	5	13	13	29	3	5	3	5	2	2
Namen	6	12	12	12	12	4	2	8	2	4	3	12	2	3
Bestwert	8	12	15	37	12	15	29	29	4	8	4	22	4	4

Abkürzungen: **RMS** - **RMS**prop; **Kom** - **Kom**mentare entfernt; **Ein** - **Ein**rückungen und Leerzeilen entfernt;
Namen - **Namen** vereinfacht

Tabelle 5.6: Bestwerte der *Difflib-lokal*-Ergebnisse aller Architekturen auf allen Datensätzen

Durch das Entfernen von Einrückungen und Leerzeilen wurden ähnlich wie beim Entfernen der Kommentare weniger Textsequenzen beim Vektorisieren erstellt, was bedeutet, dass weniger unnötige Muster gelernt wurden. Da die Einrückungen und Leerzeilen aber deutlich weniger Anteile am Code haben, als die Kommentare, wirkte sich dieser Vorverarbeitungsschritt weniger stark auf die Ergebnisse aus. Die Netzwerke haben sich teilweise sogar etwas verschlechtert. Die Ursache dafür konnte im Rahmen dieser Arbeit nicht festgestellt werden. Daher empfiehlt es sich nicht, nur die Einrückungen und Leerzeilen ohne die Kommentare zu entfernen, um bessere Ergebnisse zu erhalten.

Denn die Kombination der beiden Vorverarbeitungsschritte; Entfernen der Kommentare, Einrückungen und Leerzeilen, hat sehr gute Auswirkungen auf das Voraussagen von Quelltext. Dadurch haben sich fast alle Netzwerke in allen Metriken verbessert. Durch das Anwenden beider Schritte wurde der Code noch mehr gekürzt, sodass nur noch der reine Quelltext übrig blieb, der zum Ausführen nötig ist. Dieser kombinierte Schritt eignet sich gut, um die Qualität der Voraussagen von Code zu steigern.

Das Umbenennen von Methoden- und Feldnamen hatte nur einen minimalen Einfluss gegenüber dem Entfernen von Kommentare, Einrückungen und Leerzeilen bezüglich der Genauigkeit von Voraussagen von Quellcode. Die Netze zeigen Schwankungen, wurden teilweise besser und teilweise schlechter. Dies lässt vermuten, dass die Netzwerke keine großen Probleme mit langen Bezeichnern haben und diese fast genauso gut wie verkürzte Namen voraussagen können. Das Vereinfachen der Bezeichner hat insgesamt einen geringen positiven Effekt die Voraussagen. Da die Namensänderung durch den `JavaParser` recht

	1L128		**2L128**		**2LD128**		**2LD256**		**3LD256**		**3LD300**		**5LD500**	
	Adam	RMS	Adam	RMS	Adam	RMS	Adam	RMS	Adam	RMS	Adam	RMS	Adam	RMS
Rohdaten	29	21	5	16	19	15	21	16	6	26	5	15	6	5
Kom	26	26	24	34	20	28	46	20	5	16	6	21	7	6
Ein	24	20	26	22	27	24	22	20	5	24	5	25	5	5
Kom+Ein	23	20	18	21	27	21	20	42	5	21	5	21	4	5
Namen	19	17	23	21	18	15	4	19	4	16	4	15	4	4
Bestwert	29	26	26	34	27	28	46	42	6	26	6	25	7	6

Abkürzungen: **RMS** - **RMS**prop; **Kom** - **Kom**mentare entfernt; **Ein** - **Ein**rückungen und Leerzeilen entfernt; **Namen** - **Namen** vereinfacht

Tabelle 5.7: Medianwerte der *Difflib-global*-Ergebnisse aller Architekturen auf allen Datensätzen

	1L128		**2L128**		**2LD128**		**2LD256**		**3LD256**		**3LD300**		**5LD500**	
	Adam	RMS	Adam	RMS	Adam	RMS	Adam	RMS	Adam	RMS	Adam	RMS	Adam	RMS
Rohdaten	47	22	6	34	25	20	30	23	7	31	6	43	7	6
Kom	44	38	28	39	25	40	50	25	6	31	8	27	10	8
Ein	40	27	46	43	38	25	27	28	6	30	7	26	6	5
Kom+Ein	23	24	36	36	39	21	34	50	6	27	6	32	6	6
Namen	22	44	37	39	23	18	6	23	5	20	5	36	5	5
Bestwert	47	44	46	43	39	40	50	50	7	31	8	43	10	8

Abkürzungen: **RMS** - **RMS**prop; **Kom** - **Kom**mentare entfernt; **Ein** - **Ein**rückungen und Leerzeilen entfernt; **Namen** - **Namen** vereinfacht

Tabelle 5.8: Bestwerte der *Difflib-global*-Ergebnisse aller Architekturen auf allen Datensätzen

aufwendig ist und einige Zeit in Anspruch nimmt, würde sich dieser Schritt in zukünftigen Arbeiten im Vergleich zum geringen Nutzen nicht lohnen.

5.5.2 Netzwerkarchitekturen

Die einfacheren Netzwerkarchitekturen wie **1L128**, **2L128**, **2L128** und **2LD256** schnitten öfters besser ab als ihre komplexeren Gegenspieler. Besonders bei dem Bestwerten über die 5 Durchgänge zeigen sich deutliche Unterschiede. Das KNN **5LD500** erzielte fast immer die schlechtesten Ergebnisse.

Wahrscheinlich schnitten die kleinen Netzwerke so gut ab, weil sie weniger Knoten und Schichten haben und dadurch deren Gewichte innerhalb von 20 Epochen gut optimiert werden konnten. Dabei schien die Trainingszeit für eine gute Optimierung von bis einschließlich 2 Schichten zu reichen. Ab 3 Schichten wurden die KNNs schlechter im Vergleich zu ihren Kollegen. Entweder lag es nur an der Epochenanzahl oder mehr Schichten und damit komplexere erlernbare Muster bedeuten nicht zwangsläufig eine Erhöhung der Genauigkeit von Voraussagen von Code. Da **3LD300** öfter besser war als **3LD256**, lässt sich daraus ableiten, dass mehr Knoten und damit mehr Muster die Voraussagen von Quelltext verbessern.

Dies lässt darauf schließen, dass selbst kleine Netzwerke zum Generieren von Code ausreichen, zumindest für Quelltext ohne Semantik. Für spätere Arbeiten mit höheren Ansprüchen kann es natürlich sein, dass komplexere KNNs benötigt werden.

Das Netzwerk **5LD500** konnte voraussichtlich aufgrund der erhöhten Komplexität innerhalb von 20 Epochen nicht so viel lernen wie die anderen Netze. Es wiese häufig die schlechtesten Werten in den Metriken auf. Auch wenn man den generierten Quellcode betrachtet, sieht dieser wesentlich ungeordneter und unlesbarer aus als bei seinen Konkurrenten. Auf den von Kommentaren entfernten Datensatz hat **5LD500** mit dem Optimierer `RMSprop` und einer Diversität von 1,0 den folgenden (längeren) Code geschrieben:

```
iovGet)ei.  ai
nocdpF
in) Bau o.P ;T.itu.t; ln
te zJl(  sPhoe.piiI(Ja  tb topee ieak t
it(rpo
pjbfgclfdvt imauz, r de}iHii    w )  rieg s
hhE(r _aitrn etnAsae2  ewte

Faf nun.air iat iJ {  nm; c
```

Es sind keine Wörter, sondern höchstens Stücke wie `Get` or `air` erkennbar. Die Klammern und Semikolons wurden falsch gesetzt. Im Ganzen sieht es nicht wie Quelltext aus. Die Generierungen auf anderen Datensätzen sehen ähnlich schlecht aus. Das erklärt, warum die Ergebnisse der Metriken für diese Netzwerkarchitektur so niedrig ausfallen.

Vielleicht sind solch große KNNs nicht für die Generierung von Quellcode geeignet und führen auch nach mehr Epochen zu unbefriedigenden Ergebnissen. Dies sollte in späteren Arbeiten getestet werden.

Das neuronale Netz **2L128** erreichte häufiger bessere Resultate als **2LD128**. Dies lässt darauf schließen, dass die Technik `Dropout` keinen positiven Einfluss auf das die Genauigkeit von Voraussagen von Code des KNNs **2L128** hat. Dies bedeutet jedoch nicht, dass `Dropout` nie zu einer Verbesserung führt. Die größeren Netzwerke wie **2LD256**, **3LD256** und **3LD300** erzielen gute Werte. Deshalb kann es sein, dass die Technik nur einen negativen Einfluss auf kleine Netze hat, jedoch nicht auf komplexere.

Auf den Rohdaten und den von Kommentaren befreiten Daten machte sich **2LD256** `Adam` bemerkbar und stellte sich über alle 4 Metriken als eines der besten Netzwerke heraus. Jedoch änderte sich dies nach dem zusätzlichen Entfernen der Einrückungen und Leerzeilen sowie der Namensvereinfachung. Dort erzielten die `RMSprop` Varianten mehrheitlich die besseren Ergebnisse. Beim Netzwerk **5LD500** macht es keinen großen Unterschied, welcher Optimierer eingesetzt wird. Generell scheint `RMSprop` bei einem vorverarbeitetem Datensatz die bessere Wahl zur Codegenerierung zu sein. Er wird für die weitere Forschung empfohlen, jedoch kann man zusätzlich weitere Experimente zum Optimierer `Adam` starten.

5.5.3 Kombinationen

Auf den Rohdaten und dem Datensatz ohne Kommentare erwies sich **2LD256** `Adam` anhand der Metriken als bestes Netzwerk zur Codegenerierung. Beim letzteren Datensatz erzielten **2L128** `RMSprop` sowie **3lD300** `RMSprop` die besten Ergebnisse. Dies konnte man auch nach dem Entfernen der Kommentare, Einrückungen und Leerzeilen sehen, jedoch nicht so deutlich. Allerdings erreichte dort **2LD256** `RMSprop` bei der *Difflib-global*-Metrik (Bestwerte) den Maximalwert, obwohl ihn beim Datensatz ohne Kommentare **2LD256** `Adam` ebenfalls erzielte.

Es ist schwierig zu sagen, welche Kombination am besten war. Aufgrund der Maximalwerte wäre **2LD256** `RMSprop` am besten auf den Daten ohne Kommentare, Einrückungen und Leerzeichen, während **2LD256** `Adam` ohne Kommentare unübertrefflich war. Zusätzlich zeigten die kleinen Netzwerke bis zu 2 Schichten und 128 Neuronen ebenfalls sehr gute Ergebnisse bei den letzten 3 Vorverarbeitungsschritten.

5.5.4 Quelltextbetrachtung

Insgesamt spiegeln die Metriken die Lesbarkeit und Plausibilität des vorausgesagten Codes nur bedingt wieder. Da alle Metriken nur die Zeichen mit dem Original vergleichen, aber nicht die Wörter an sich betrachten, fallen die Bewertungen insgesamt recht schlecht aus. Dabei sieht der generierte Code nicht schlecht aus:

```java
package org.assertj.swing.driver;

import java.awt.event.ActionEvent;
import javax.swing.*;
import javax.swing.border.EmptyBox;
import javax.swing.JPanel;
import javax.swing.JPanel;



public class MouseAdapter extends Rectangle {
```

Dieses erweiterte Beispiel entstand auf dem Datensatz ohne Kommentare bei einer Diversität von 0,5 mit dem Netz **1L128** und dem Optimierer `RMSprop`. Diesmal wurden mehr Zeichen generiert und das Ergebnis sieht auf den ersten Blick wie normaler Code aus. Bei genaueren Hinsehen fällt aber auf, dass z.B. `import javax.swing.JPanel;` zweimal importiert wird, obwohl außerdem `import javax.swing.*;` schon importiert wurde. Die Anordnung innerhalb der Klassendeklaration scheint hingegen in Ordnung zu sein, jedoch macht es keinen Sinn, dass die Klasse `MouseAdapter` von der Klasse `Rectangle` erbt. Dennoch klingen die Bezeichner einzeln stichhaltig.

Hier ein weiteres Beispiel:

```
@Override
public void paintIcon(Component c, Graphics g, int x, int y, int height)
    {
int width = textBox.getSelectedItem();
if (textBox.isSelected()) {
this.parent.setText("target");
}
public void setText(JComponent c) {
return true;
}
public void setDefaultCloseOperation(String text, final int columnIndex)
    {
this.paint = new JTextField();
this.add(new JTextField("Color", "target", "arrow");
}
public void setSelectedItem(String text) {
public class MouseAdapter extends Rectangle {
```

Dieses ebenfalls längere Beispiel stammt von dem Netzwerk **2L128** mit `RMSprop` unter der Verwendung einer Diversität von 0,5 und dem Datensatz ohne Kommentare, Einrückungen und Leerzeilen. Der Anfang ist dem Originalcode sehr ähnlich. Dieser beginnt mit den folgenden Zeilen:

```
@Override
public void paintIcon(Component c, Graphics g, int x, int y) {
```

Es ist erstaunlich wie ähnlich sich die ersten Zeilen sind. Der Codeausschnitt zur Erzeugung ging nur bis `public void paintIcon(Componen`, aber das KNN hat die erste Zeile exakt vorausgesagt und sich dabei noch einen weiteren Parameter (`int height`) für die Methode `paintIcon()` ausgedacht. Der danach folgende Quelltext sieht ziemlich plausibel aus. Die Klammerung ist korrekt, die Aufrufe sehen in Ordnung aus und einige Namen wie z.B. `setSelectedItem` oder `JTextField` ergeben Sinn. Gerade die Zeile `this.parent.setText("target");` sieht sehr realistisch aus. Das Netzwerk hat anscheinend erkannt, dass zu `setText` ein String wie `"target"` gehört, der in Anführungszeichen gesetzt wird. Natürlich ist der Code nicht perfekt und enthält noch keine übergreifende Semantik, aber stellt einen guten Ausgangspunkt für weitere Forschung dar.

5.6 Gültigkeitsbetrachtungen

An dieser Stelle sollen die Ergebnisse kritisch betrachtet werden, ob sich Fehler einschleichen konnten, die die Resultate oder Schlussfolgerungen verfälscht haben. Außerdem wird diskutiert, ob die Ergebnisse verallgemeinert werden können oder nur für diesen Versuchsaufbau gültig sind.

5.6.1 Interne Gültigkeit

Da das Netzwerk `batch`-weise mit Trainingsdaten versorgt wird, muss man überlegen, welche Daten zusammen übergeben werden sollen und welche getrennt. Eine Methode sollte als ganzes übergeben werden, um dem Netzwerk dessen zusammenhängende Bedeutung erkennen zu lassen. Dies wurde jedoch noch nicht in dieser Arbeit getan. Durch die zufällige Aufteilung der Trainingsdaten könnte das Netz den Bezug zwischen den zusammengehörigen Teilen nicht erkannt haben.

Um eindeutigere Ergebnisse zur Voraussage von Code zu bekommen, wäre es besser mehr Epochen zu trainieren. Aufgrund der Zeitbeschränkung, der Dauer des Trainings und der Anzahl der zu untersuchenden KNNs war dies leider nicht möglich. Besonders das komplexe Netzwerk **5LD500** hat viele interne Parameter, die innerhalb von 20 Epochen nicht vollständig optimiert werden konnten.

Außerdem hätte man in der Evaluierung mehr Durchgänge starten sollen, um den Einfluss des Zufalls möglichst gering zu halten. Jedoch ließ das die Zeit nicht zu und es wurde nur 5 mal hintereinander Code generiert. Dies sollte dennoch einen etwas allgemeineren Blick auf die Resultate ermöglichen.

Die 4 Metriken, die zur Evaluierung gewählt wurden, waren nicht so aussagekräftig wie erwartet, da zwar lesbarer und teilweise syntaktisch korrekter Code entstanden ist, aber er nicht genau dem Original entsprach und daher von den Metriken schlecht bewertet wurde. Den Erfolg einiger Voraussagen konnten die Metriken nicht genau abdecken. Sie beurteilten nur die Nähe zum Original und nicht die Qualität des generierten Codes an sich.

5.6.2 Externe Gültigkeit

Es wäre besser gewesen, wenn die KNNs auf mehr Trainingsdaten trainiert hätten. Ein `swing`-Datensatz von knapp 40 MB ist nicht ausreichend, um Aussagen über jedwede Codeerzeugung zu treffen. Da nur eine Domäne in den Daten vertreten war, lassen sich die Ergebnisse nicht zwingend auf andere Themengebiete außer `swing` anwenden. Dazu wurde nur Java-Code als Trainingsdaten verwendet und vorausgesagt. Man kann nicht mit Sicherheit sagen, dass sich ähnliche Ergebnisse bei anderen Programmiersprachen zeigen werden.

Zum Nachstellen dieser Experimente müssen genügend Ressourcen im Sinne von RAM, GPUs und Speicherplatz vorhanden sein. Es sollte darauf geachtet werden, dass genügend RAM zur Verfügung steht, da die vektorisierten Trainingsdaten viel davon benötigen. Zusätzlich nehmen die gespeicherten Checkpoints der Architekturen viel Speicherplatz in Anspruch. Und um nicht wochenlang trainieren zu müssen, sollten möglichst viele GPUs gleichzeitig genutzt werden.

Kapitel 6

Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten zum Thema automatische Codegenerierung vorgestellt.

Ein genereller lückenloser (eng. *end-to-end*) Ansatz zum Sequenzenlernen (engl. *sequence learning*), der minimale Voraussetzungen an die Sequenzstruktur stellt, wird von Sutskever et al. [57] vorgestellt. Es wird ein mehrschichtiges LSTM-Netzwerk benutzt, das die Eingabesequenzen auf einem Vektor einer festen Dimensionalität abbildet und danach ein anderes tiefes LSTM-Netz, das den Vektor in die Zielsequenz umwandelt. Das Ergebnis ist eine automatische Englisch-Französisch-Übersetzung, die ein satzbasiertes, statistisches Maschinenübersetzungssytem übertrifft und keine Probleme beim Übersetzen langer Sätze hat. Das Netzwerk hat sinnvolle Ausdruck- und Satzrepräsentationen gelernt und ist abhängig von Wortanordnungen, aber invariant bei Aktiv und Passiv. Sie haben gezeigt, dass ein tiefes LSTM-Netzwerk mit einem begrenzten Vokabular und fast keinen Ansprüchen an die Struktur des Problems ein Standard-Maschinenübersetzungssytem mit unbegrenzten Vokabular überbieten kann. Dieser Erfolgt lässt vermuten, dass ein solches Netzwerk sich auch in vielen anderen *sequence learning* Problemen gut schlagen würde.

Campbell und Treude [10] haben eine Autovervollständigung (engl. *content assist*) für Codeauschnitte in die integrierte Entwicklungsumgebung (IDE) Eclipse eingebaut. Da Entwickler zunehmend im Internet nach bestimmten Codeausschnitten suchen, die sie in ihre eigenen Programme einbinden können, müssen sie von ihrer Entwicklungsumgebung zum Browser wechseln und dort ihre Anfrage stellen. Um diesen Kontextwechsel zu vermeiden und ihre Produktivität zu erhöhen, wurde das Eclipse plugin `NLP2Code` vorgestellt. Es ist direkt in Eclipse integriert und liefert Codeausschnitte aus der Frage-Antwort-Plattform Stack Overflow unmittelbar an die aktuelle Stelle des Mauszeigers in der IDE. Ihre Evaluierung zeigte, dass die Mehrheit der Aufrufe der Auto-Vervollständigung zu relevanten Codeausschnitten für ein breites Spektrum an Anfragen wie API-Anwendungsbeispiele und Algorithmen führte und hilfreich waren.

Van Kooten [36] hat dieses Jahr *Neural Complete*, ebenfalls eine Code-Autovervollständigung, entwickelt. Es basiert auf einem LSTM-Netzwerk, dass auf `Python` Quellcode, der `Keras`-Importe beinhaltet, trainiert wurde. Das Ergebnis ist ein trainiertes KNN, das hilft Code für neuronale Netze zu schreiben, indem es Vorschläge zum Vervollständigen der aktuellen Zeile macht. Es bezieht dabei den Inhalt der vorherigen Zeilen mit ein und passt den nächsten Vorschlag darauf an. Die aktuelle Implementierung umfasst 2 Modelle, ein zeichenbasiertes und ein `Python token` (Elementarbestandteile der Programmiersprache) Modell. Der Vorteil des Zeichenbasierten ist, dass es jederzeit vervollständigen kann, während das `token` Modell nur mit vollständigen `token` funktioniert, also keine Wörter ergänzen kann. Das `token` Modell ist auf einer höheren, semantischen Abstraktionsstufe angesiedelt als das andere und sollte daher Zusammenhänge besser erkennen. Das zeichenbasiertes Modell betrachtet die letzten 80 Zeichen, das `token` Modell die letzten 20 `token`.

In der Arbeit von Mou et al. [45] wird sich ein (*end-to-end*) Programmgenerierungsszenario ausgemalt, dass auf RNNs basiert. Benutzer geben ihre Programmierintention in natürlicher Sprache an und ein RNN generiert automatisch den passenden Quellcode zeichenweise. Sie weisen dessen Machbarkeit anhand von Fallstudien und empirischer Analysen nach, sind sich aber bewusst, dass noch weitere Schritte zur vollen Umsetzung nötig sind. Man steht vor einigen disziplinenübergreifenden Herausforderungen wie z.B. der Modellierung der Benutzerabsicht, der Datensatzbereitstellung und dem Verbessern der Netzwerkarchitektur. Ihre Version ist es, dass es eines Tages reichlich qualitativ hochwertigen Code gibt, der gut kommentiert und dokumentiert ist und eine mächtige Maschine wie ein neuronales Netzwerk, das die Zuordnung von Problembeschreibungen zu Quelltexten gelernt hat. Entwickler können dann ihr Vorhaben durch natürliche Sprache ausdrücken und die Maschine wird automatisch den gewünschten Code ausgeben.

Alexandru [1] führt in seiner Arbeit die Idee der Geführten Codesynthese (engl. *Guided Code Synthesis*) ein. Sie soll Entwicklern beim Programmieren helfen, indem sie in einem interaktiven Prozess ein trainiertes neuronales Netzwerk zur gewünschten Quellcodeausgabe führen, ohne selbst auf andere Ressourcen wie z.B. Stack Overflow zugreifen zu müssen. Es soll ein Code vervollständigendes System entstehen, dass sich der riesigen Datenmenge im Internet bedient, um kontinuierlich geeignete Voraussagen anzubieten, während der Entwickler gerade an seinem Programm schreibt. Dies soll die Produktivität beim Entwickeln steigern. Bisher wurde nur ein RNN auf 10 MB Java-Quellcode trainiert, um Beispielcode zu generieren, und 7 GB Quelltext von 825 *GitHub*-Projekten gesammelt sowie jeweils 3000 zufällige Codeausschnitte für 24 Programmiersprachen ausgewählt und auf einem RNN trainiert, um die korrekte Sprache mit einer Genauigkeit von 91% zu bestimmen. In der Zukunft soll der Trainingsdatensatz für das Code vervollständigende System erstellt werden und die Experimente mit verschiedenen Netzwerkarchitekturen für kontextsensitive Codesynthese beginnen.

Balog et al. [4] stellen zwei Grundideen vor; zum einen das Erzeugen von Programmen mit

Hilfe von einem Datensatz aus einfachen Programmieraufgabenstellungen, um Strategien zu lernen, die übergreifend für mehrere Problemstellungen gültig sind und zum Anderen das Integrieren von neuronalen Netzwerkarchitekturen mit such-basierten Techniken. Das KNN soll die Suche nach einem Programm in Übereinstimmung mit einer Auswahl an Eingabe-Ausgabe-Paaren leiten anstatt direkt den gesamten Quelltext auszugeben. Es lernt die Abbildung von Eingabe-Ausgabe-Beispielen auf Attribute des Originalprogramms, das diese Paare erzeugt hat. Die Attribute sind als das Vorhandensein oder die Abwesenheit von bestimmten Funktionen definiert. Die Voraussagen des Netzes werden benutzt, um suchbasierte Algorithmen zu unterstützen. Das System *DeepCoder* ist imstande Probleme des Schwierigkeitsgrads verglichen mit den einfachsten Problemen auf Programmierwettbewerbsseiten zu lösen.

Kapitel 7

Zusammenfassung und Ausblick

Die automatische Generierung von Code ermöglicht es, in der Zukunft die Lösungen von Programmierproblemen innerhalb weniger Sekunden oder Minuten auf Knopfdruck zu erhalten. Die Forschung dazu steht noch am Anfang, jedoch wurde in dieser Arbeit gezeigt, dass es möglich ist, mit einfachen KNNs, wenigen Epochen Trainingszeit und einer geeigneten Vorverarbeitung der Trainingsdaten, Code vorauszusagen, der nah am Code des Datensatzes und teilweise syntaktisch korrekt ist. Er sieht menschlichem Code ähnlich, weist aber wenig Semantik auf.

Es hat sich in den Experimenten herausgestellt, dass sich die Vorverarbeitungsschritte Kommentare entfernen und Kommentare, Einrückungen sowie Leerzeilen entfernen positiv für fast alle getesteten Netzwerkarchitekturen auf das Voraussagen von Code auswirken. Die Hinzunahme des Namensvereinfachens hatte keinen so großen positiven Einfluss wie das Entfernen der Kommentare, jedoch haben sich die meisten Architekturen etwas gesteigert. Die Werte des Datensatzes ohne Einrückungen und Leerzeilen haben sich im Vergleich zu den Rohdaten vorwiegend verschlechtert. Die kleineren KNNs bis zu 2 Schichten und 256 Knoten schnitten oft besser ab als die komplexeren und sind durchaus in der Lage syntaktisch korrekten Code zu generieren. Auf den vorverarbeiteten Trainingsdaten zeigte der Optimierer `RMSprop` in der Regel die besseren Resultate. Die Kombination aus **2LD256** `Adam` und Kommentare entfernen sowie **2LD256** `RMSprop` und Kommentare, Einrückungen sowie Leerzeilen entfernen haben die höchst möglichen Ergebnisse der *Difflib-global*-Metrik erzielt.

Für zukünftige Arbeiten wird empfohlen weitere, freie `Java`-Projekte herunterzuladen und zusätzlich zum bestehenden Datensatz zu nutzen. Auf diesen können ebenfalls die in Kapitel 3 vorgestellten Vorverarbeitungsschritte angewendet und getestet werden. Als nächstes Ziel wird die Erzeugung von durchgehend syntaktisch und ggf. semantisch korrekten Quellcode vorgeschlagen. Dazu sollte überprüft werden, ob die gesammelten Trainingsdaten vollständig syntaktisch korrekt sind, um den Netzwerken nur einwandfreien Code zum Lernen bereitzustellen.

Die Repräsentation der Trainingsdaten als Wortvektor (engl. *word2vec*) oder als Syntaxbaum stellt einen fortgeschritteneren Ansatz als die zeichenweise Methode dar. Bei der wortweisen Repräsentation ist anzumerken, dass die Datenstrukturen zum Vektorisieren der Wörter wahrscheinlich sehr groß werden und daher sogenannte *sparse matrices* [54] bzw. *sparse tensors* [59] der `SciPy`- bzw. `TensorFlow`-Bibliothek empfohlen werden. Da das Netzwerk `batch`-weise mit Trainingsdaten versorgt wird, könnte man festlegen, dass eine Methode als Ganzes übergeben werden soll, um dem Netzwerk den semantischen Zusammenhang dieser beizubringen. Würde man dem KNN neben dem Code zum Ergänzen bestimmte Schlüsselwörter oder Kontextinformationen übergeben, ist es denkbar, dass besserer Quelltext erzeugt wird. Wenn man eine Methodendeklaration als Eingabe vorgibt, kann man diese Kontextinformation dem Netz ebenfalls mitteilen. So ist es möglich verschiedene Codeabschnitte z.B. als Methodendeklaration, Methodenrumpf, Klassendeklaration etc. zu kategorisieren und dem Netzwerk als Zusatzinformation dazu zu geben. Dadurch könnte sich die Genauigkeit der Voraussagen von Code verbessern.

Als weiterer Forschungsansatz schließt sich die Anpassung oder Ergänzung der in Kapitel 4 vorgestellten Netzwerkarchitekturen an. Die aktuellen Architekturen können mit weiteren Optimierern, anderen Aktivierungen und Zielfunktionen getestet werden. Laut der Arbeit von Marcin Andrychowicz et al. [3] sind LSTM-Netzwerke die besten Optimierer für LSTM-Netzwerke. Für weitere Arbeiten in diesem Gebiet wäre es interessant, ein LSTM-Netzwerk anstelle eines normalen Optimierers einzusetzen. Die LSTM-Schichten können außerdem gegen andere (wie z.B. `rekurrente`, `convolutional` oder `pooling` Schichten) ausgetauscht werden. Es ist durchaus möglich die Netzwerke größer zu gestalten und mehr LSTM- und/oder andere Schichten sowie mehr Knoten einzubauen. Es können verschiedene anspruchsvolle Schichten innerhalb einer Architektur verwendet werden. Die Anzahl der Schichten und Neuronen kann variiert werden und sich auch innerhalb mehrerer Schichten ändern. Eine Vielzahl von Zusammenstellungen aus verschiedenen Knoten, Schichten, Optimierern, Aktivierungen und Zielfunktionen ist möglich. Zur Verbesserung der Voraussagen bietet es sich an, die bestehenden oder neuen Architekturen länger als bisher auf den Trainingsdaten zu trainieren. Gerade große und komplexe Netzwerke wie `5LD500` müssen nach mehr Epochen betrachtet und evaluiert werden, da sie innerhalb von 20 Epochen nicht so viel lernen können wie kleinere Netze.

Da die hier benutzten Metriken (Abschnitt 5.2) die Qualität des vorausgesagten Quelltexts nur mit bekanntem Code vergleichen, könnte man überdenken in zukünftigen Arbeiten eine Metrik einzuführen, die den generierten Codes an sich abschätzt und z.B. die Anzahl an Berichtigungen zählt, die benötigt werden, um alle Kompilierfehler zu beseitigen.

Es wird darauf gehofft, dass weitere Arbeiten in diesem Gebiet dazu führen, dass zuverlässig syntaktisch korrekter Code generiert werden kann und ihm Semantik verliehen wird, um den Traum der automatischen Programmerzeugung eines Tages wahr werden zu lassen.

Literatur

[1] Carol V. Alexandru. “Guided Code Synthesis Using Deep Neural Networks”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, S. 1068–1070. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2983951. URL: http://doi.acm.org/10.1145/2950290.2983951 (besucht am 05. 10. 2017).

[2] Amazon. *Amazon Go*. URL: http://www.amazon.com/go (besucht am 04. 09. 2017).

[3] Marcin Andrychowicz u. a. “Learning to learn by gradient descent by gradient descent”. In: *CoRR* abs/1606.04474 (2016). arXiv: 1606.04474. URL: http://arxiv.org/abs/1606.04474 (besucht am 13. 10. 2017).

[4] Matej Balog u. a. “DeepCoder: Learning to Write Programs”. In: *CoRR* abs/1611.01989 (2016). arXiv: 1611.01989. URL: http://arxiv.org/abs/1611.01989 (besucht am 05. 10. 2017).

[5] Peter Bradley. *The XOR Problem and Solution*. URL: http://www.mind.ilstu.edu/curriculum/artificial_neural_net/xor_problem_and_solution.php (besucht am 25. 07. 2017).

[6] Jason Browniee. “Dropout Regularization in Deep Learning Models With Keras”. In: *Machine Learning Mastery* (2016). URL: https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/ (besucht am 10. 08. 2017).

[7] Jason Browniee. *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. Juli 2017. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (besucht am 10. 09. 2017).

[8] Danny van Bruggen. *JavaParser*. Projekt-Homepage. URL: http://javaparser.org/ (besucht am 10. 09. 2017).

[9] Tiobe Software BV. *Tiobe Index*. URL: https://www.tiobe.com/tiobe-index/ (besucht am 01. 10. 2017).

[10] Brock Angus Campbell und Christoph Treude. “NLP2Code: Code Snippet Content Assist via Natural Language Tasks”. In: *CoRR* abs/1701.05648 (2017). arXiv: 1701.05648. URL: http://arxiv.org/abs/1701.05648 (besucht am 11. 09. 2017).

[11] Alex Chitu. “How Google’s Image Recognition Works”. In: *Google Operating System* (2013). URL: https://googlesystem.blogspot.de/2013/06/how-googles-image-recognition-works.html#gsc.tab=0 (besucht am 10. 08. 2017).

[12] François Chollet u. a. *Keras*. https://github.com/fchollet/keras. 2015. (Besucht am 25. 07. 2017).

[13] François Chollet u. a. *Keras Examples - LSTM-Text-Generation.* URL: https://github.com/fchollet/keras/blob/master/examples/lstm_text_generation.py (besucht am 05. 09. 2017).

[14] François Chollet u. a. *Recurrent Layers - LSTM.* Projekt-Homepage. URL: https://keras.io/layers/recurrent/#lstm (besucht am 13. 10. 2017).

[15] Jeff Donahue u. a. "Long-term Recurrent Convolutional Networks for Visual Recognition and Description". In: *CoRR* abs/1411.4389 (2014). arXiv: 1411.4389. URL: http://arxiv.org/abs/1411.4389 (besucht am 11. 09. 2017).

[16] Python Software Foundation. *difflib.* Modul-Homepage. URL: https://docs.python.org/3.6/library/difflib.html (besucht am 10. 09. 2017).

[17] Prof. Dr. Guenter Gauglitz und Clemens Jürgens. *Backpropagation - Prinzip der Gradientenverfahren neuronaler Netze.* Tutorial. URL: http://www.chemgapedia.de/vsengine/tra/vsc/de/ch/13/trajektorien/nn_ein.tra/Vlu/vsc/de/ch/13/vlu/daten/neuronalenetze/backpropagation.vlu/Page/vsc/de/ch/13/anc/daten/neuronalenetze/snn8_2.vscml.html (besucht am 10. 09. 2017).

[18] Prof. Dr. Guenter Gauglitz und Clemens Jürgens. *Backpropagation - Probleme des Backpropagation-Lernverfahrens.* Tutorial. URL: http://www.chemgapedia.de/vsengine/tra/vsc/de/ch/13/trajektorien/nn_ein.tra/Vlu/vsc/de/ch/13/vlu/daten/neuronalenetze/backpropagation.vlu.html (besucht am 10. 09. 2017).

[19] Prof. Dr. Guenter Gauglitz und Clemens Jürgens. *Backpropagation - Probleme des Backpropagation-Lernverfahrens.* Tutorial. URL: http://www.chemgapedia.de/vsengine/tra/vsc/de/ch/13/trajektorien/nn_ein.tra/Vlu/vsc/de/ch/13/vlu/daten/neuronalenetze/backpropagation.vlu/Page/vsc/de/ch/13/anc/daten/neuronalenetze/snn8_5.vscml.html (besucht am 10. 09. 2017).

[20] Prof. Dr. Guenter Gauglitz und Clemens Jürgens. *Geschichte Neuronaler Netze.* Tutorial. URL: http://www.chemgapedia.de/vsengine/vlu/vsc/de/ch/13/vlu/daten/neuronalenetze/einfuehrung.vlu/Page/vsc/de/ch/13/anc/daten/neuronalenetze/snn1_6.vscml.html (besucht am 10. 09. 2017).

[21] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: *CoRR* abs/1308.0850 (2013). arXiv: 1308.0850. URL: http://arxiv.org/abs/1308.0850 (besucht am 05. 10. 2017).

[22] Alex Graves, Abdel-rahman Mohamed und Geoffrey E. Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In: *CoRR* abs/1303.5778 (2013). arXiv: 1303.5778. URL: http://arxiv.org/abs/1303.5778 (besucht am 11. 09. 2017).

[23] Antti Haapala. *Levenshtein.* Package Homepage. URL: https://pypi.python.org/pypi/python-Levenshtein/0.12.0 (besucht am 08. 08. 2017).

[24] Donald Olding Hebb. *The organization of behavior: A neuropsychological approach.* John Wiley & Sons, 1949.

[25] Geoffrey E. Hinton. *Lecture 6a: Overview of mini-batch gradient descent.* Lecture. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (besucht am 10. 09. 2017).

[26] Geoffrey E. Hinton u. a. "Improving neural networks by preventing co-adaptation of feature detectors". In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580. URL: http://arxiv.org/abs/1207.0580 (besucht am 10. 08. 2017).

[27] Sepp Hochreiter und Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), S. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735 (besucht am 25. 07. 2017).

[28] Sepp Hochreiter u. a. *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies.* 2001.

[29] Haomiao Huang. “How Amazon Go (probably) makes “just walk out” groceries a reality”. In: *Ars Technica* (2017). URL: https://arstechnica.com/information-technology/2017/04/how-amazon-go-probably-makes-just-walk-out-groceries-a-reality/ (besucht am 10. 08. 2017).

[30] Andrej Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition - Neural Networks Part 3.* URL: http://cs231n.github.io/neural-networks-3/ (besucht am 13. 10. 2017).

[31] Andrej Karpathy. “The Unreasonable Effectiveness of Recurrent Neural Networks”. In: *Andrej Karpathy blog* (2015). URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (besucht am 01. 09. 2017).

[32] Andrej Karpathy, Justin Johnson und Fei-Fei Li. “Visualizing and Understanding Recurrent Networks”. In: *CoRR* abs/1506.02078 (2015). arXiv: 1506.02078. URL: http://arxiv.org/abs/1506.02078 (besucht am 01. 09. 2017).

[33] Keras. *Getting started with the Keras functional API.* Dokumentation. URL: https://keras.io/getting-started/functional-api-guide/ (besucht am 04. 09. 2017).

[34] Diederik P. Kingma und Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (besucht am 10. 09. 2017).

[35] Marina Knabbe. *Erzeugung von Musiksequenzen mit LSTM-Netzwerken.* ger. 2017. URL: http://edoc.sub.uni-hamburg.de/haw/volltexte/2017/3964/pdf/BA_1971279.pdf (besucht am 25. 07. 2017).

[36] Pascal van Kooten. *Neural Complete.* https://github.com/kootenpv/neural_complete. 2017. (Besucht am 11. 09. 2017).

[37] Karl S. Lashley. “In Search of the Engram”. In: *Symposia of the Society for Experimental Biology 4* (1950). URL: http://gureckislab.org/courses/fall13/learnmem/papers/Lashley1950.pdf (besucht am 10. 08. 2017).

[38] Qi Lyu u. a. “Modelling High-Dimensional Sequences with LSTM-RTRBM: Application to Polyphonic Music Generation.” In: *IJCAI.* 2015, S. 4138–4139.

[39] Martín Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (besucht am 03. 09. 2017).

[40] Warren S. McCulloch und Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (Dez. 1943). ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259 (besucht am 10. 08. 2017).

[41] Robert McMillan. “How Google Retooled Android With Help From Your Brain”. In: *Wired* (2013). URL: https://www.wired.com/2013/02/android-neural-network/ (besucht am 10. 08. 2017).

[42] Michigan State University Digital Evolution Laboratory. *Avida Digital Evolution Platform*. Projekt-Homepage. URL: `http://avida.devosoft.org/` (besucht am 25. 07. 2017).

[43] Microsoft. *The Microsoft Cognitive Toolkit*. Projekt-Homepage. URL: `https://docs.microsoft.com/cognitive-toolkit/` (besucht am 04. 09. 2017).

[44] Marvin Minsky und Seymour Papert. “Perceptrons.” In: (1969).

[45] Lili Mou u. a. “On End-to-End Program Generation from User Intention by Deep Neural Networks”. In: *CoRR* abs/1510.07211 (2015). arXiv: `1510.07211`. URL: `http://arxiv.org/abs/1510.07211` (besucht am 05. 10. 2017).

[46] Christoph Obenhuber. *Kursprognose mittels nichtlinearer Regression (MLP) vs. linearer Regression (OLS) am Beispiel der wöchentlichen Rendite des Dow Jones EURO STOXX 50*. diplom.de, 2003.

[47] *Optimierung von KNN*. URL: `http://www.informatik.uni-ulm.de/ni/Lehre/SS02/Evosem/OptimierungKNNFolien.pdf` (besucht am 06. 06. 2017).

[48] Oracle Corporation. *Java*. Homepage. URL: `http://www.oracle.com/technetwork/java/index.html` (besucht am 25. 07. 2017).

[49] Michael Orlov und Moshe Sipper. *Darwinian Software Engineering*. Projekt-Homepage. URL: `http://finch.cs.bgu.ac.il/` (besucht am 25. 07. 2017).

[50] Juliane Pestel. *Einführung und Geschichte neuronaler Netze*. URL: `http://www.desy.de/~guenterg/prosem/Einf_hrung_und_Geschichte_neuronaler_Netze.html` (besucht am 04. 09. 2017).

[51] Python Software Foundation. *Python*. Projekt-Homepage. URL: `https://www.python.org/` (besucht am 25. 07. 2017).

[52] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *CoRR* abs/1609.04747 (2016). arXiv: `1609.04747`. URL: `http://arxiv.org/abs/1609.04747` (besucht am 13. 10. 2017).

[53] Hamed Safikhani. “Modeling and multi-objective Pareto optimization of new cyclone separators using CFD, ANNs and NSGA II algorithm”. In: *Advanced Powder Technology* 27.5 (2016). ISSN: 0921-8831. DOI: `http://dx.doi.org/10.1016/j.apt.2016.08.017`. URL: `http://www.sciencedirect.com/science/article/pii/S0921883116302278` (besucht am 10. 08. 2017).

[54] The Scipy community. *Sparse matrices (scipy.sparse)*. URL: `https://docs.scipy.org/doc/scipy/reference/sparse.html` (besucht am 24. 10. 2017).

[55] Lee Spector. *Evolutionary Computing with Push*. Projekt-Homepage. URL: `http://faculty.hampshire.edu/lspector/push.html` (besucht am 25. 07. 2017).

[56] Nitish Srivastava u. a. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), S. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html` (besucht am 10. 08. 2017).

[57] Ilya Sutskever, Oriol Vinyals und Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *CoRR* abs/1409.3215 (2014). arXiv: `1409.3215`. URL: `http://arxiv.org/abs/1409.3215` (besucht am 05. 10. 2017).

[58] TensorFlow Team. *Installing TensorFlow*. URL: `https://www.tensorflow.org/install/` (besucht am 04. 09. 2017).

[59] TensorFlow Team. *Sparse Tensors*. URL: https://www.tensorflow.org/api_guides/python/sparse_ops (besucht am 24. 10. 2017).

[60] Theano Development Team. *Theano documentation*. Dokumentation. URL: http://www.deeplearning.net/software/theano/ (besucht am 04. 09. 2017).

[61] Tijmen Tieleman und Geoffrey E. Hinton. *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning*. Lecture. 2012.

[62] Christoph Tornau. *Backpropagation of Error*. URL: http://www.informatikseite.de/neuro/node24.php (besucht am 01. 10. 2017).

[63] Oriol Vinyals u. a. "Show and Tell: A Neural Image Caption Generator". In: *CoRR* abs/1411.4555 (2014). arXiv: 1411.4555. URL: http://arxiv.org/abs/1411.4555 (besucht am 11. 09. 2017).

[64] Fabian Wackermann. *Mittlere quadratische Abweichung (Varianz)*. URL: http://www.wege-zum-abitur.de/2006/12/mittlere-quadratische-abweichung-varianz (besucht am 06. 06. 2017).

[65] Anish Singh Walia. *Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent*. URL: https://medium.com/towards-data-science/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f (besucht am 13. 10. 2017).

[66] D. Warde-Farley u. a. "An empirical analysis of dropout in piecewise linear networks". In: *ArXiv e-prints* (Dez. 2013). arXiv: 1312.6197 [stat.ML].

[67] Tsung-Hsien Wen u. a. "Semantically Conditioned LSTM-based Natural Language Generation for Spoken Dialogue Systems". In: *CoRR* abs/1508.01745 (2015). arXiv: 1508.01745. URL: http://arxiv.org/abs/1508.01745 (besucht am 11. 09. 2017).

[68] Wikipedia. *Backpropagation*. URL: https://de.wikipedia.org/wiki/LMS-Algorithmus (besucht am 06. 06. 2017).

[69] Wikipedia. *Künstliches neuronales Netz*. URL: https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz#Typische_Strukturen (besucht am 02. 06. 2017).

[70] Yonghui Wu u. a. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *CoRR* abs/1609.08144 (2016). URL: http://arxiv.org/abs/1609.08144 (besucht am 10. 08. 2017).

Anhang A

Abbildungen der Evaluierung

Hier sind alle Bilder der Evaluierung zu finden, die in der Arbeit noch nicht gezeigt wurden.

Vergleich der Diversitäten

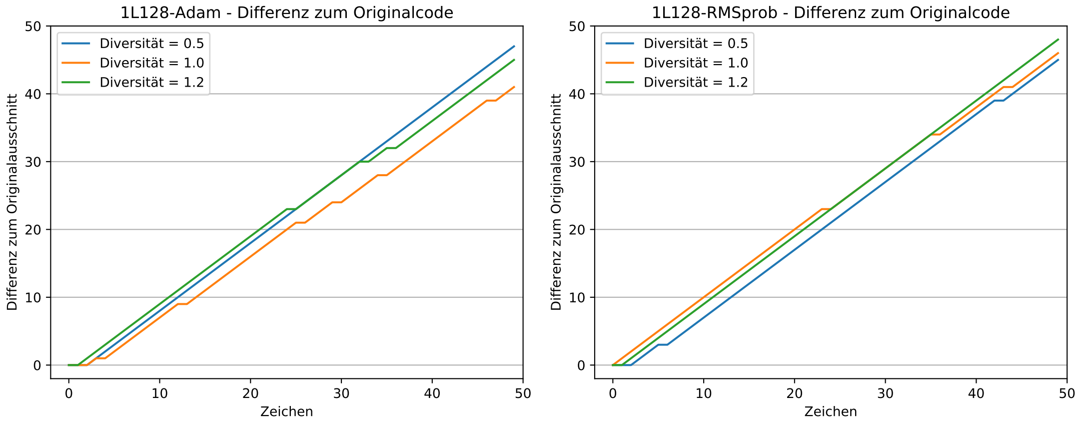

Abbildung A.1: Vergleich **1L128** auf den Rohdaten; links `Adam`, rechts `RMSprop`

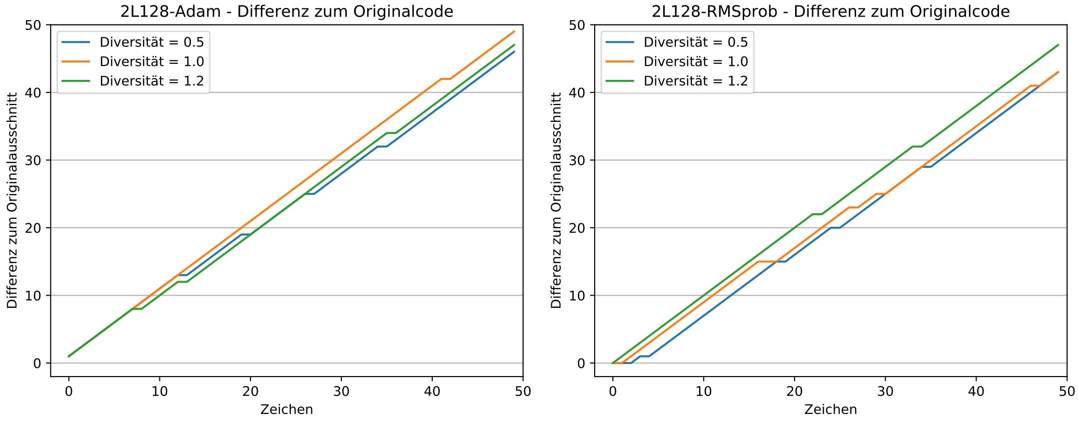

Abbildung A.2: Vergleich **2L128** auf den Rohdaten; links `Adam`, rechts `RMSprop`

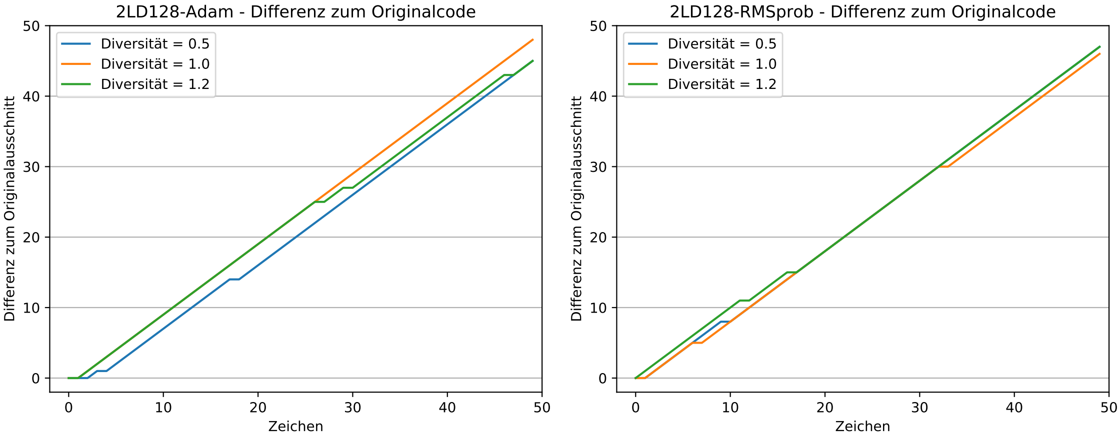

Abbildung A.3: Vergleich **2LD128** auf den Rohdaten; links `Adam`, rechts `RMSprop`

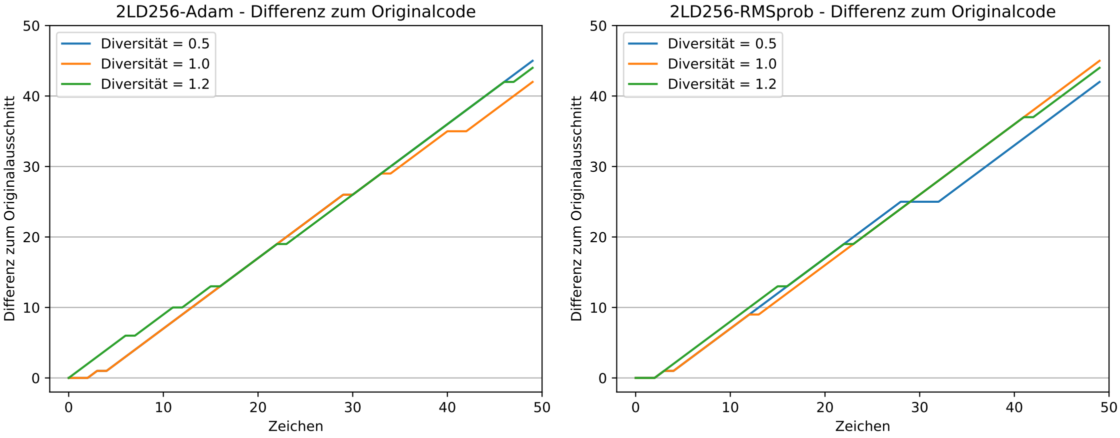

Abbildung A.4: Vergleich **2LD256** auf den Rohdaten; links `Adam`, rechts `RMSprop`

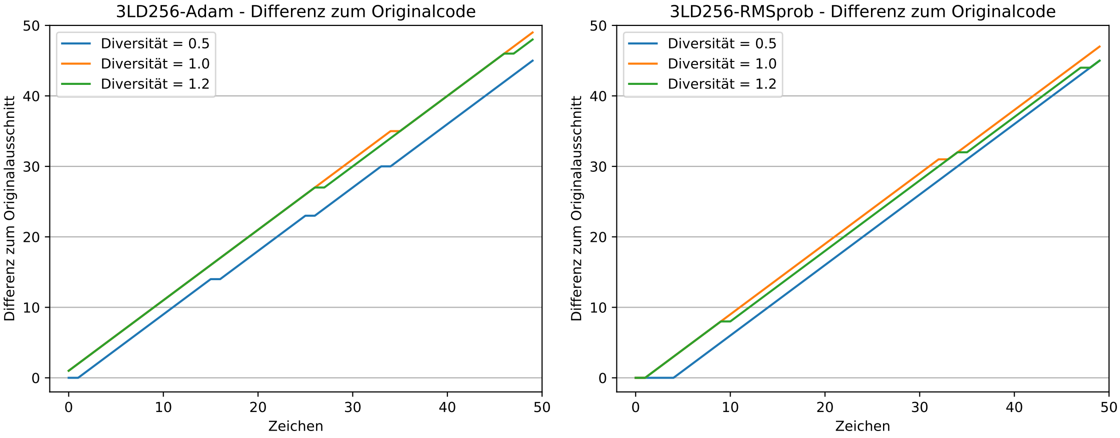

Abbildung A.5: Vergleich **3LD256** auf den Rohdaten; links `Adam`, rechts `RMSprop`

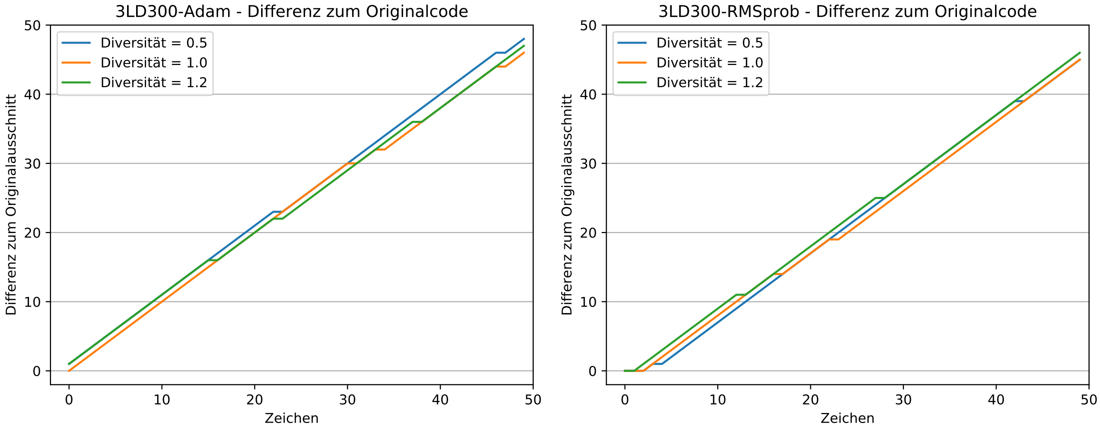

Abbildung A.6: Vergleich **3LD300** auf den Rohdaten; links `Adam`, rechts `RMSprop`

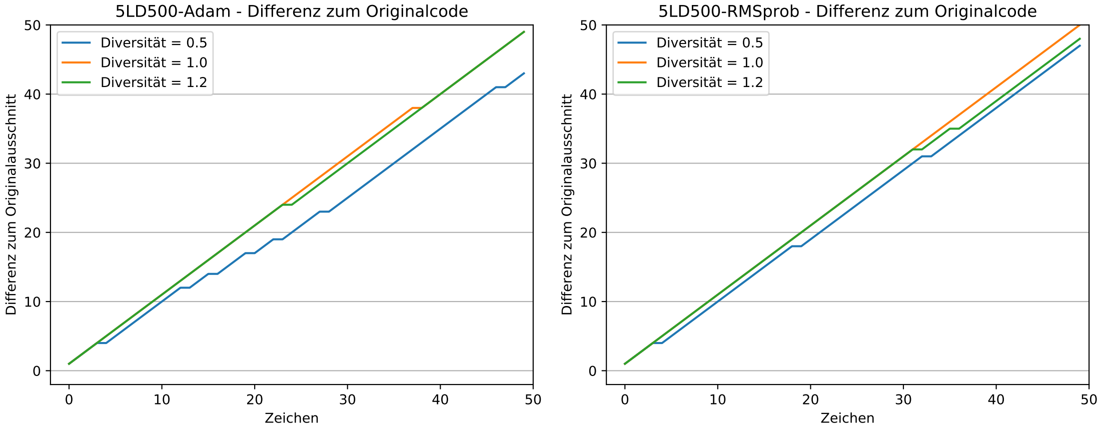

Abbildung A.7: Vergleich **5LD500** auf den Rohdaten; links `Adam`, rechts `RMSprop`

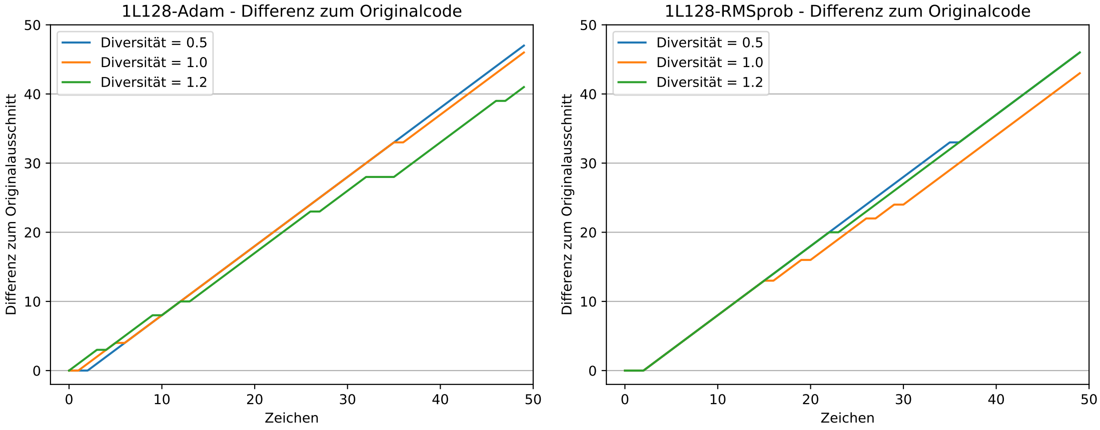

Abbildung A.8: Vergleich **1L128**; Kommentare entfernt; links `Adam`, rechts `RMSprop`

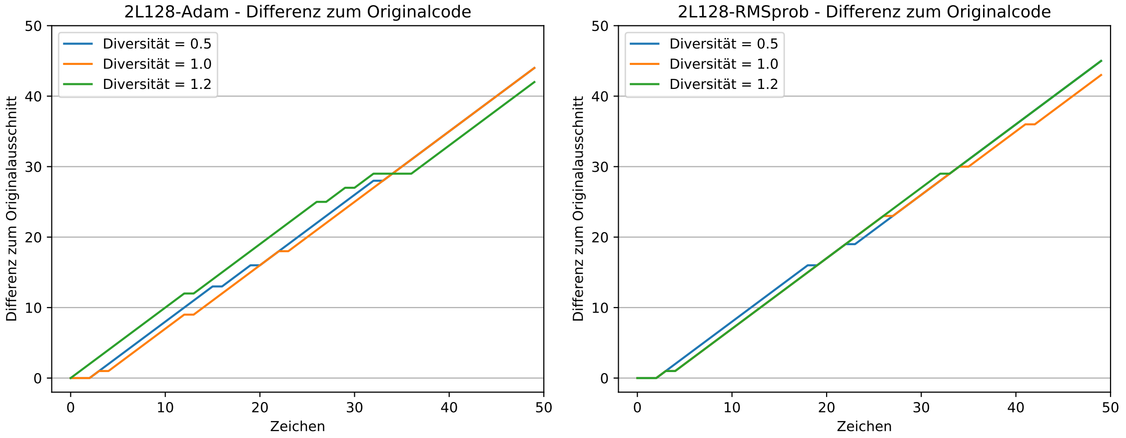

Abbildung A.9: Vergleich **2L128**; Kommentare entfernt; links `Adam`, rechts `RMSprop`

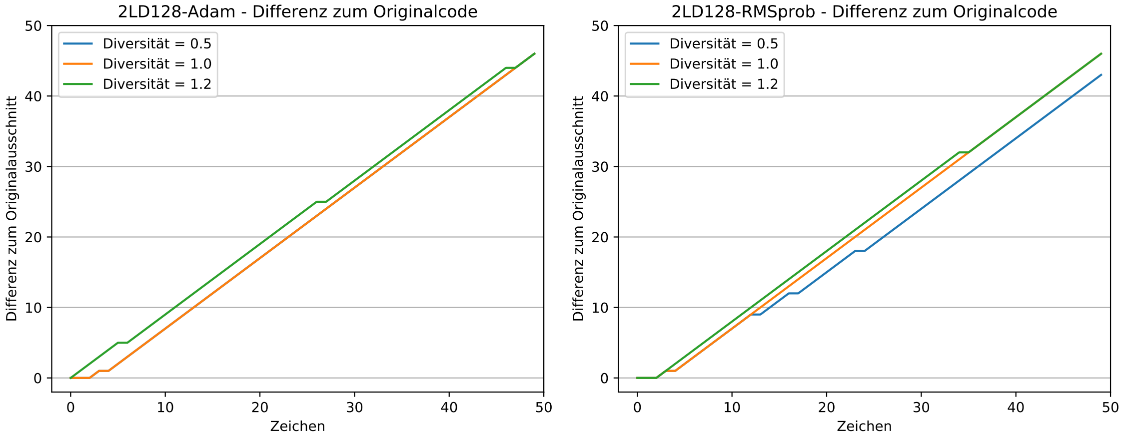

Abbildung A.10: Vergleich **2LD128**; Kommentare entfernt; links `Adam`, rechts `RMSprop`

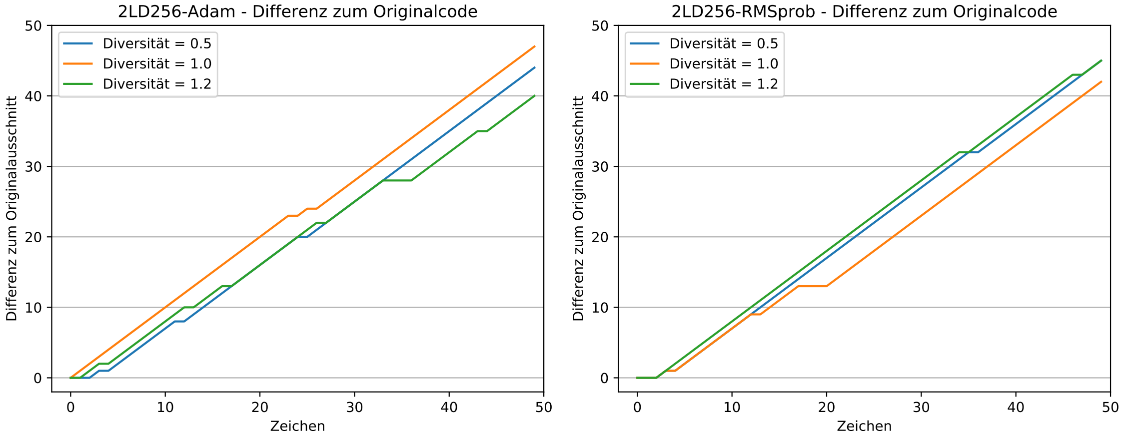

Abbildung A.11: Vergleich **2LD256**; Kommentare entfernt; links `Adam`, rechts `RMSprop`

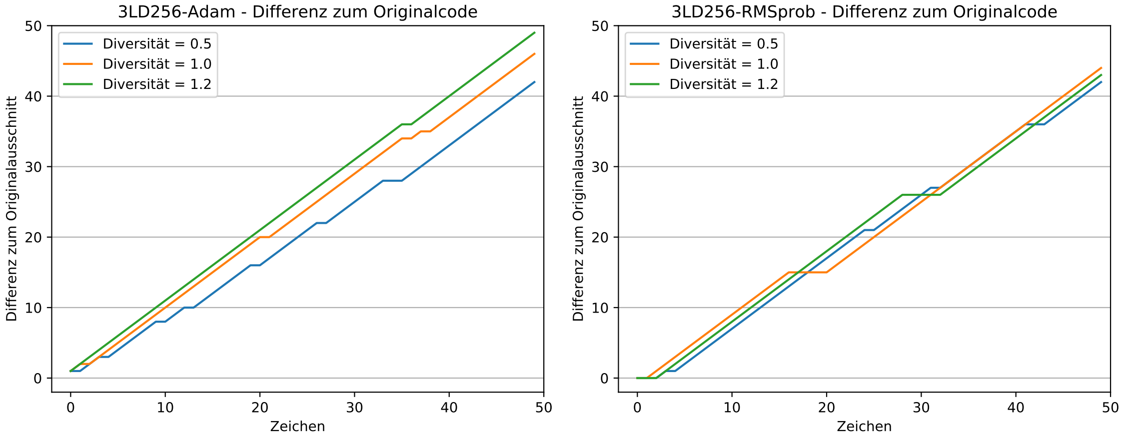

Abbildung A.12: Vergleich **3LD256**; Kommentare entfernt; links `Adam`, rechts `RMSprop`

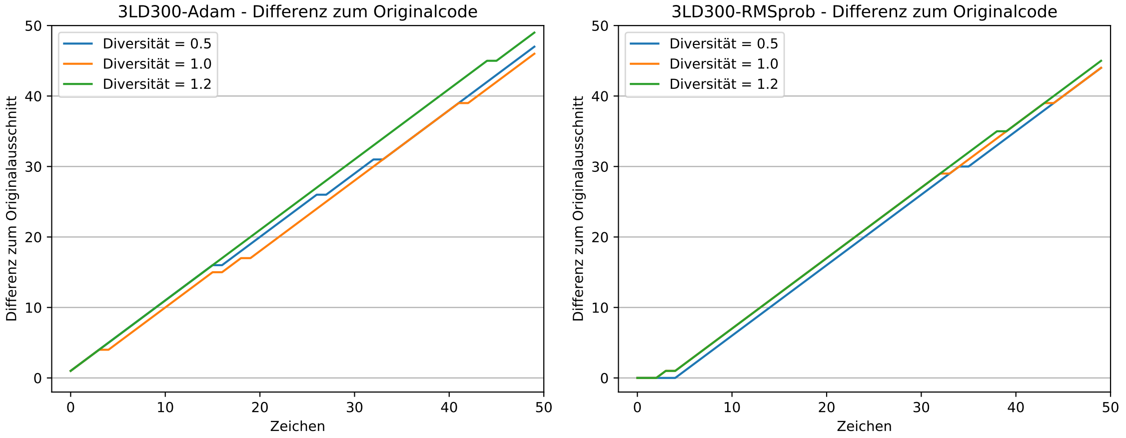

Abbildung A.13: Vergleich **3LD300**; Kommentare entfernt; links `Adam`, rechts `RMSprop`

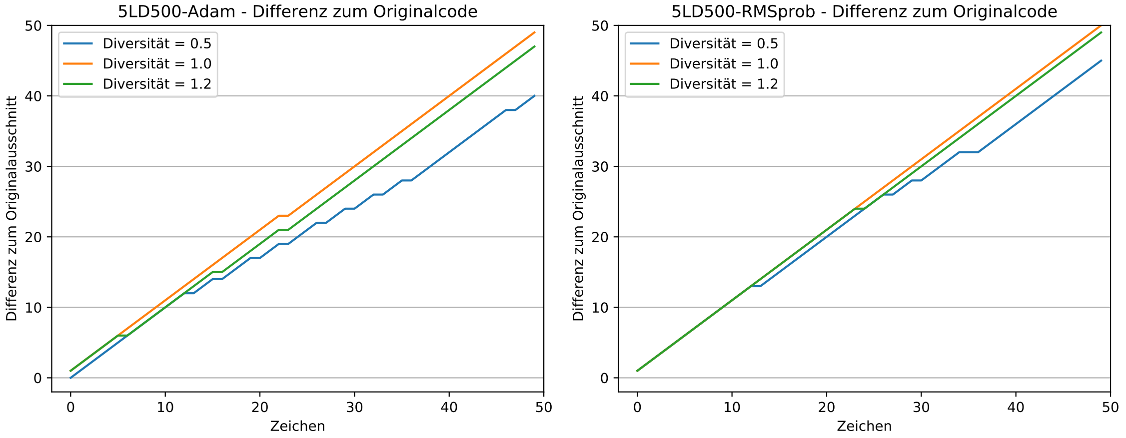

Abbildung A.14: Vergleich **5LD500**; Kommentare entfernt; links `Adam`, rechts `RMSprop`

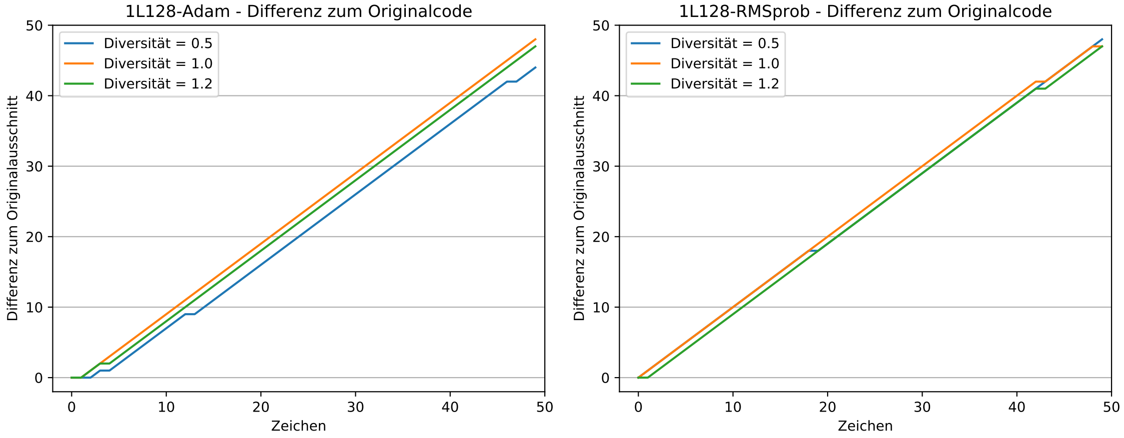

Abbildung A.15: Vergleich **1L128**; Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

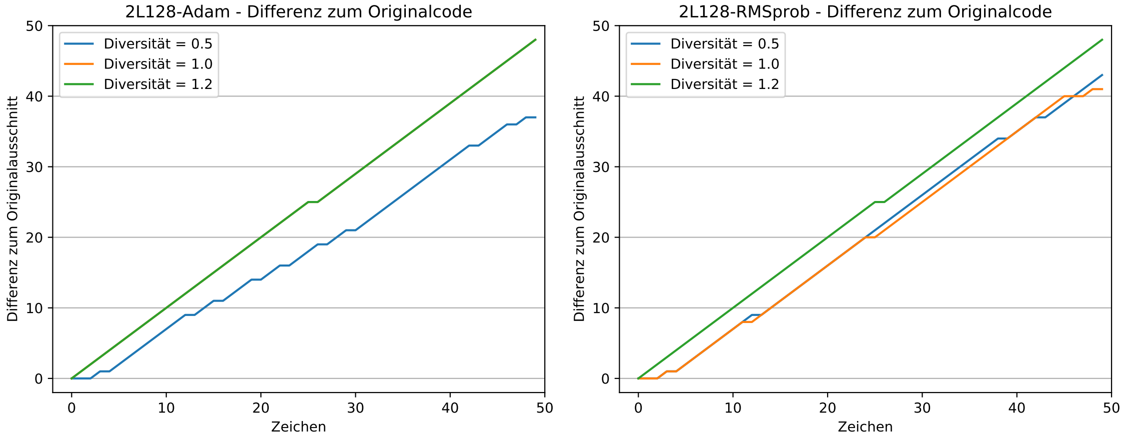

Abbildung A.16: Vergleich **2L128**; Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

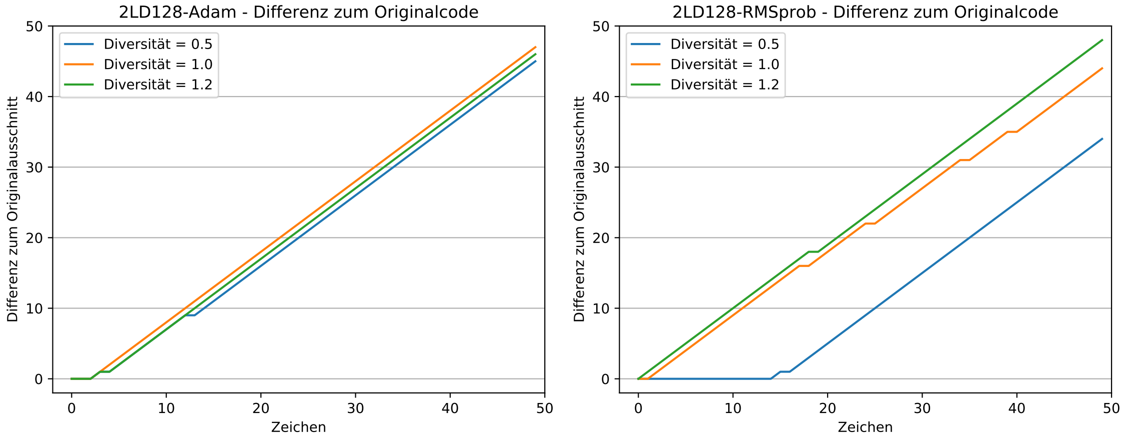

Abbildung A.17: Vergleich **2LD128**; Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

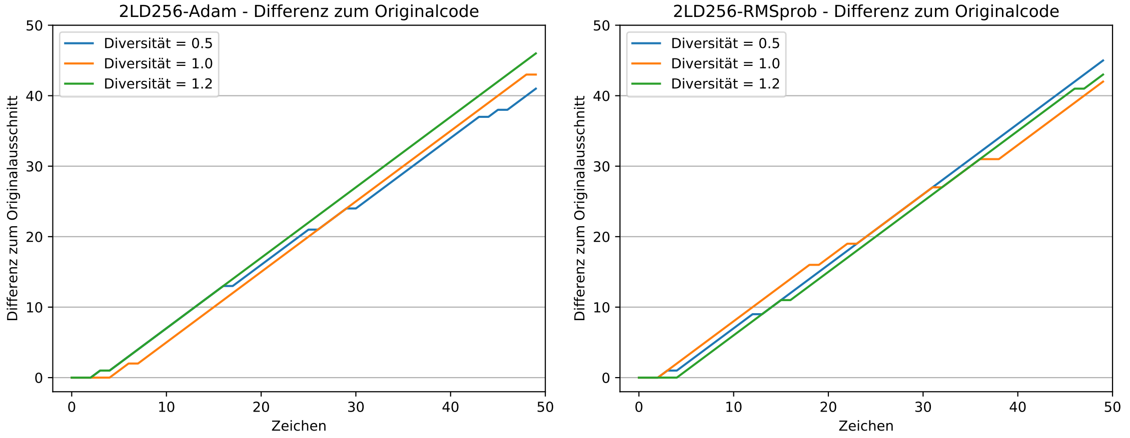

Abbildung A.18: Vergleich **2LD256**; Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

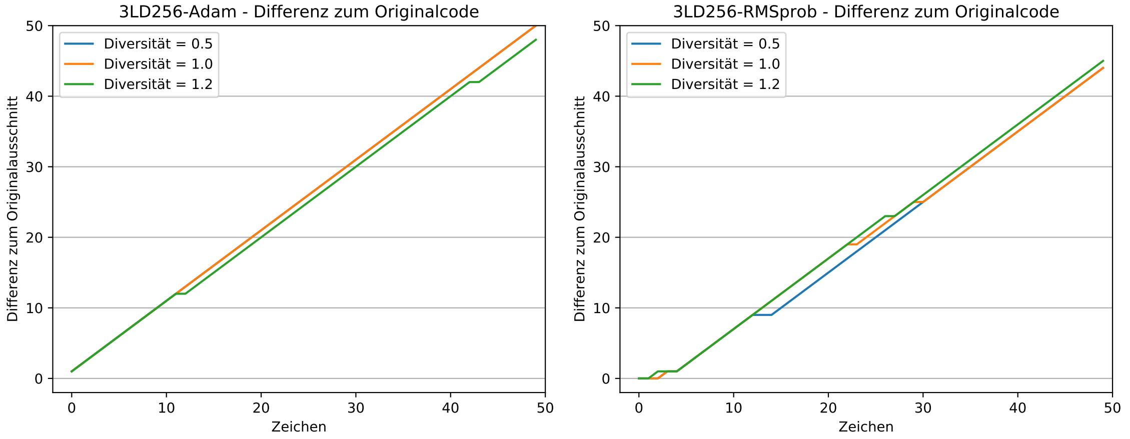

Abbildung A.19: Vergleich **3LD256**; Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

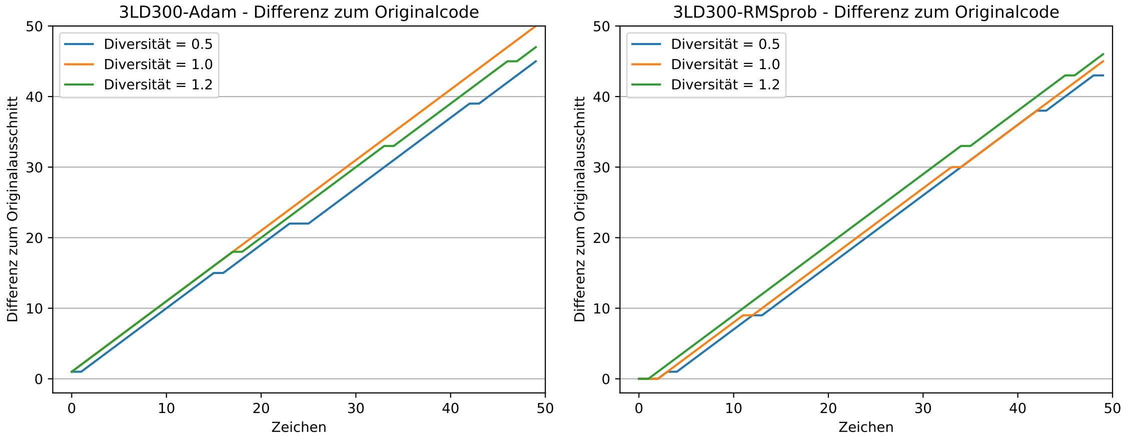

Abbildung A.20: Vergleich **3LD300**; Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

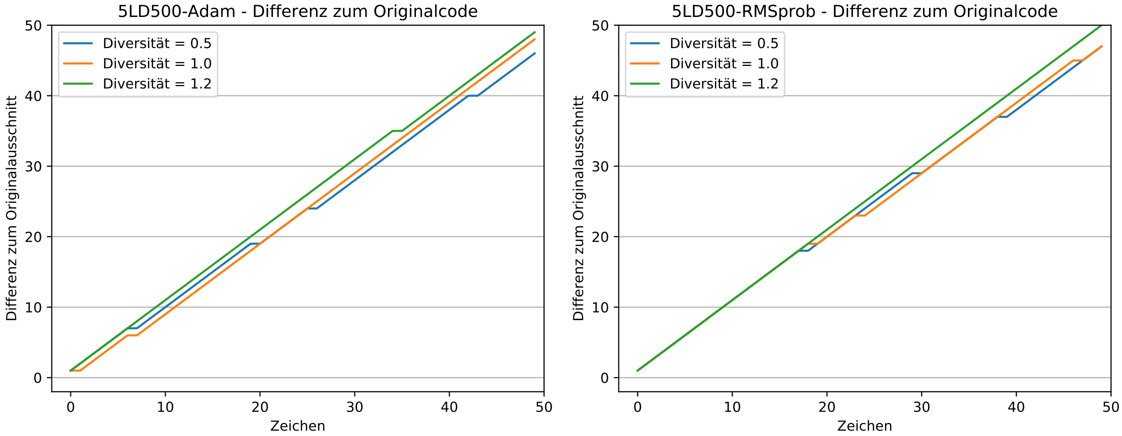

Abbildung A.21: Vergleich **5LD500**; Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

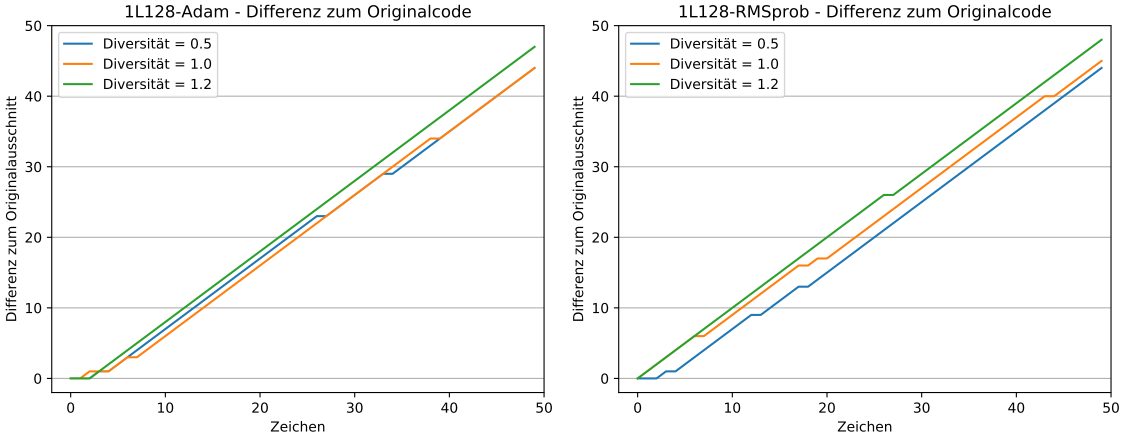

Abbildung A.22: Vergleich **1L128**; Kommentare, Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

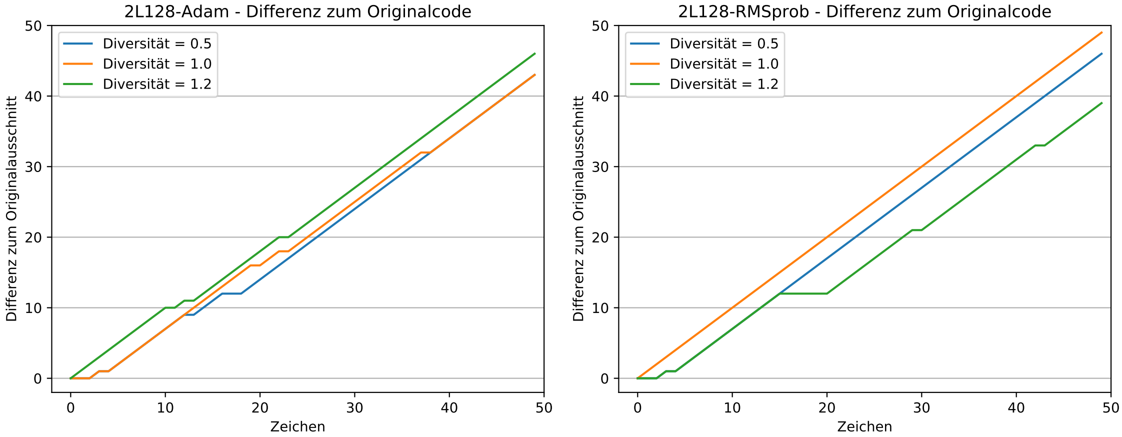

Abbildung A.23: Vergleich **2L128**; Kommentare, Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

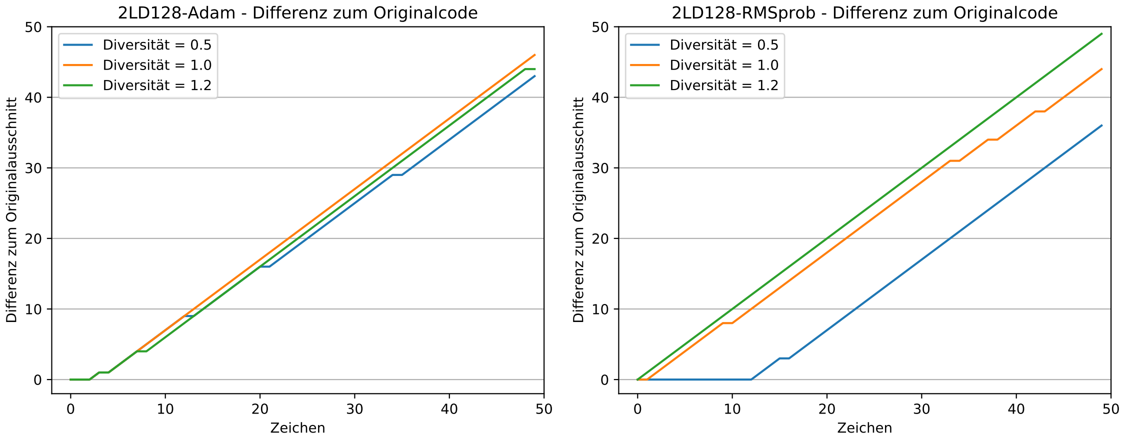

Abbildung A.24: Vergleich **2LD128**; Kommentare, Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

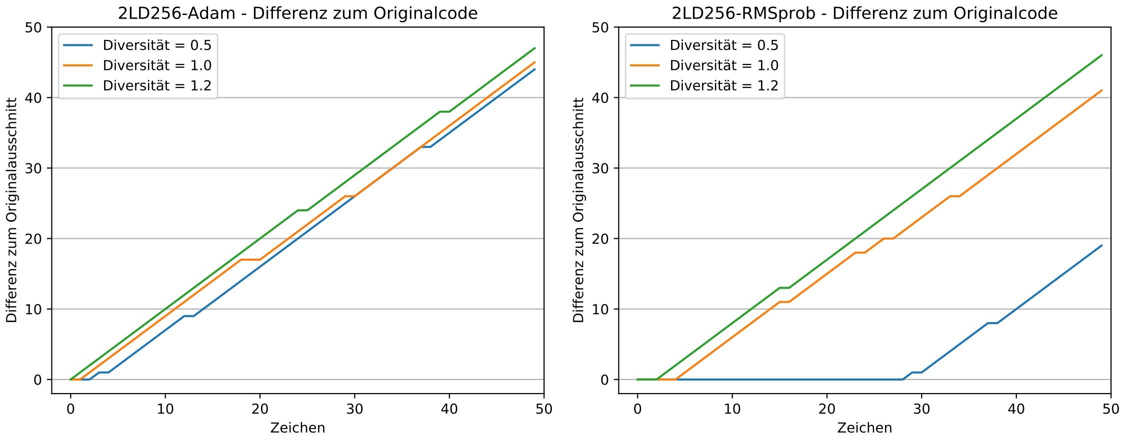

Abbildung A.25: Vergleich **2LD256**; Kommentare, Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

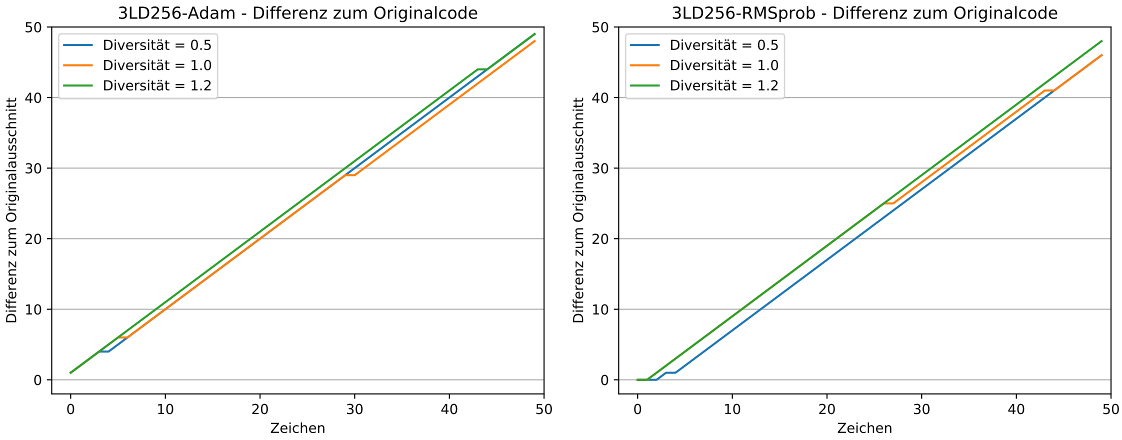

Abbildung A.26: Vergleich **3LD256**; Kommentare, Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

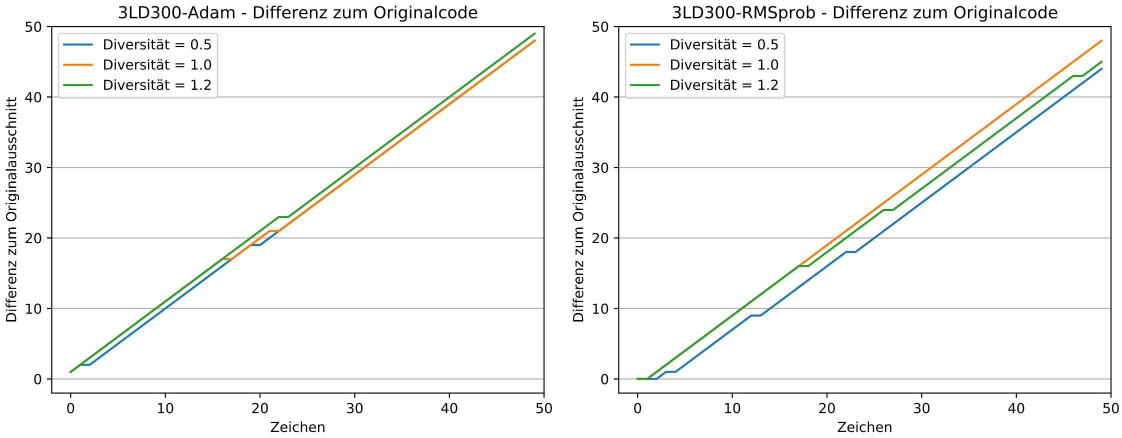

Abbildung A.27: Vergleich **3LD300**; Kommentare, Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

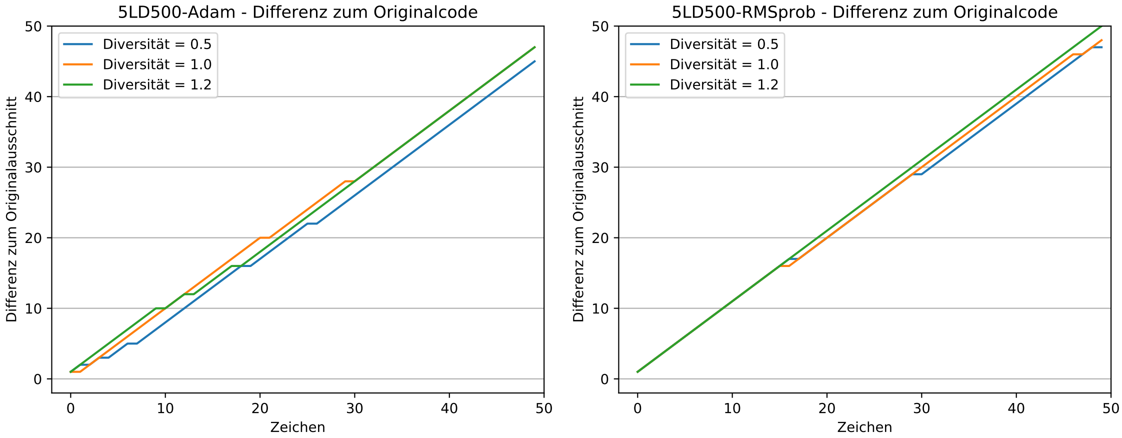

Abbildung A.28: Vergleich **5LD500**; Kommentare, Einrückungen und Leerzeilen entfernt; links `Adam`, rechts `RMSprop`

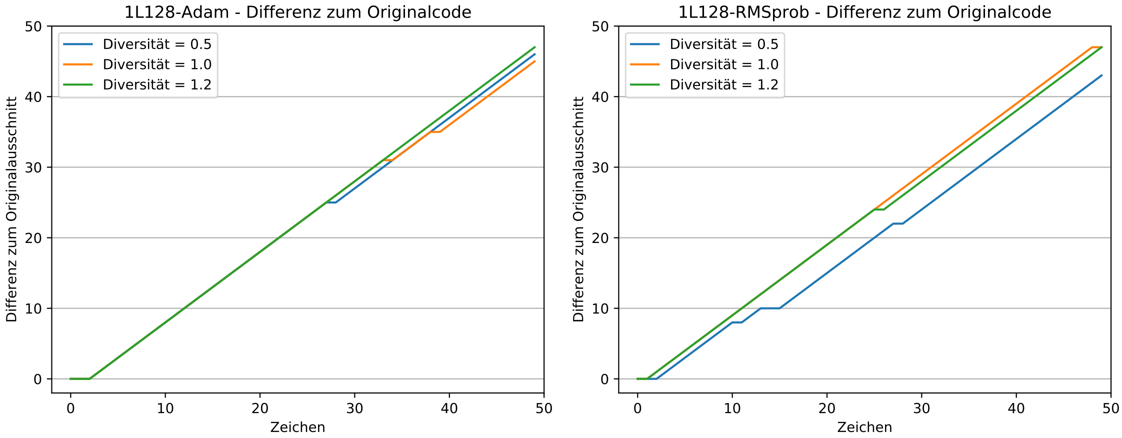

Abbildung A.29: Vergleich **1L128**; Namen vereinfacht; links `Adam`, rechts `RMSprop`

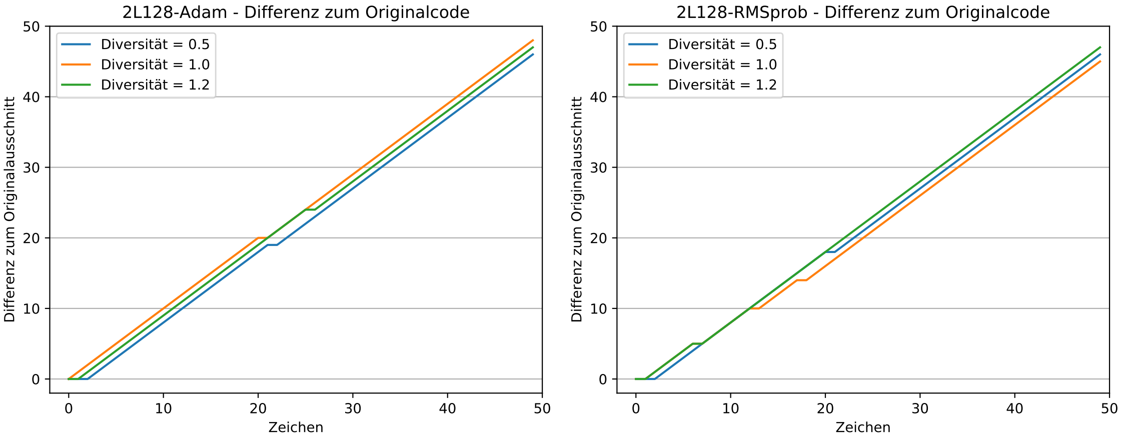

Abbildung A.30: Vergleich **2L128**; Namen vereinfacht; links `Adam`, rechts `RMSprop`

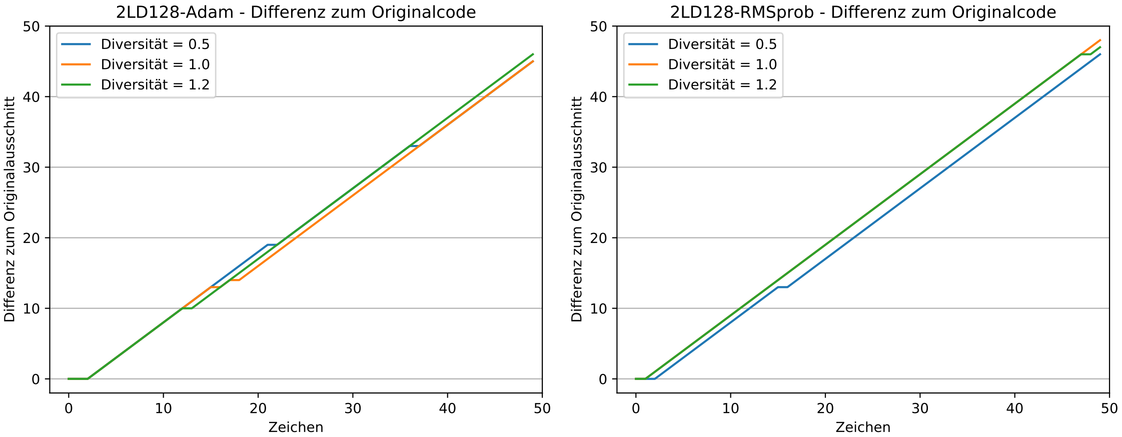

Abbildung A.31: Vergleich **2LD128**; Namen vereinfacht; links `Adam`, rechts `RMSprop`

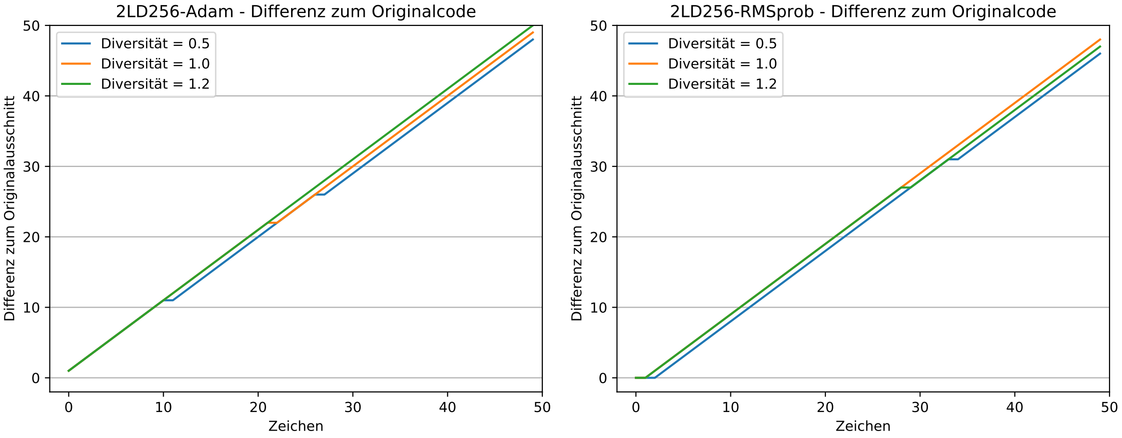

Abbildung A.32: Vergleich **2LD256**; Namen vereinfacht; links `Adam`, rechts `RMSprop`

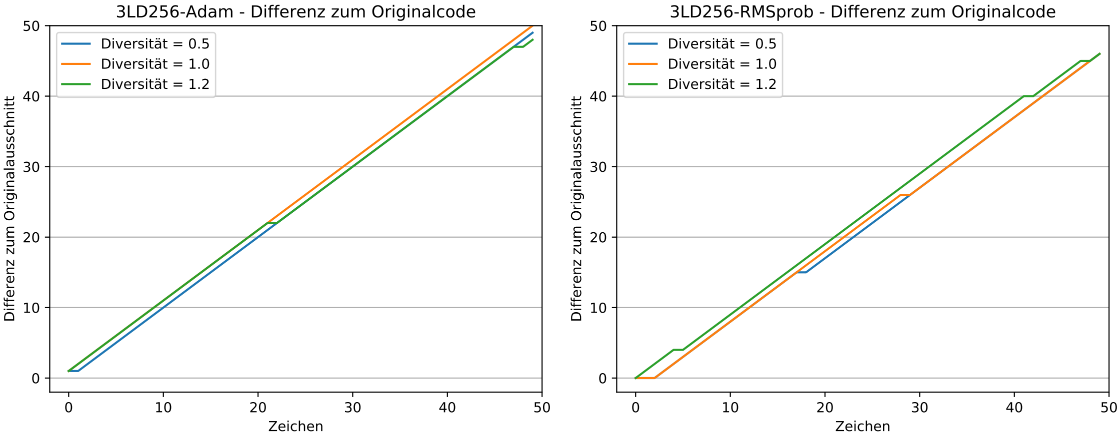

Abbildung A.33: Vergleich **3LD256**; Namen vereinfacht; links `Adam`, rechts `RMSprop`

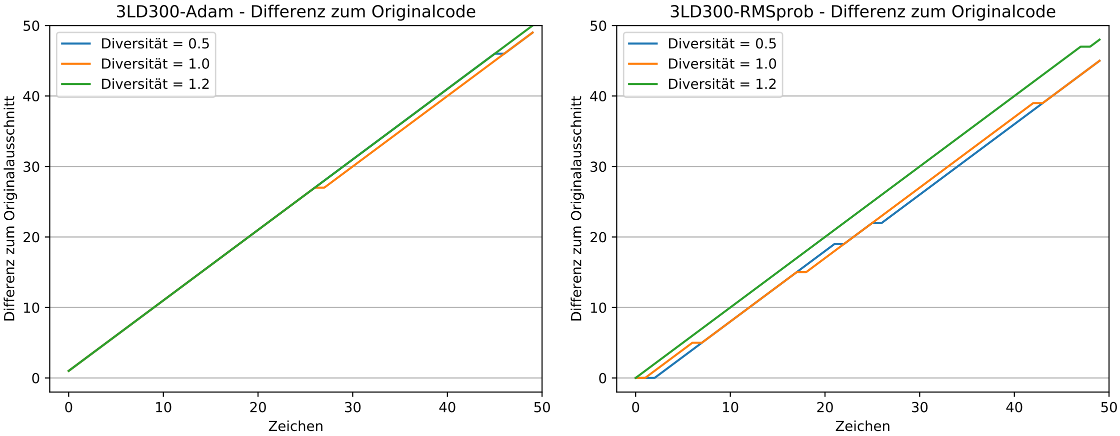

Abbildung A.34: Vergleich **3LD300**; Namen vereinfacht; links `Adam`, rechts `RMSprop`

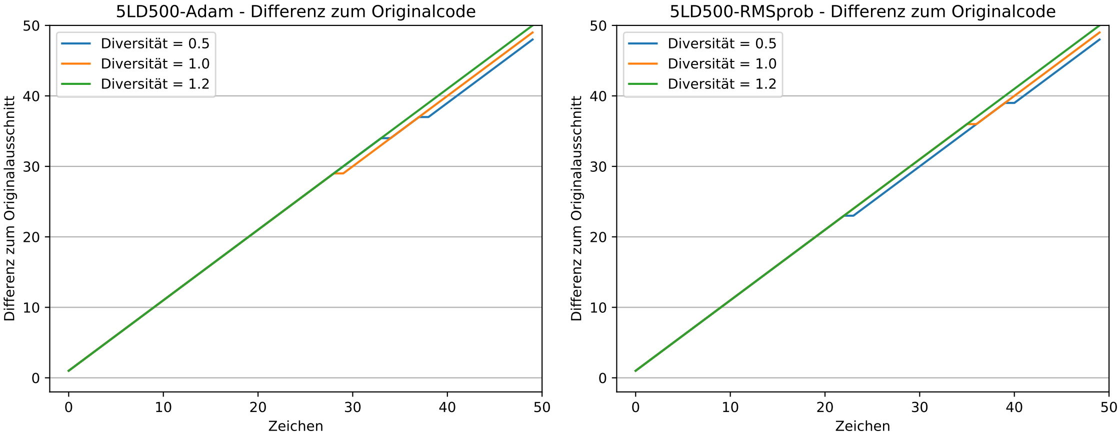

Abbildung A.35: Vergleich **5LD500**; Namen vereinfacht; links `Adam`, rechts `RMSprop`

Zeichenweise Abweichung aller Architekturen

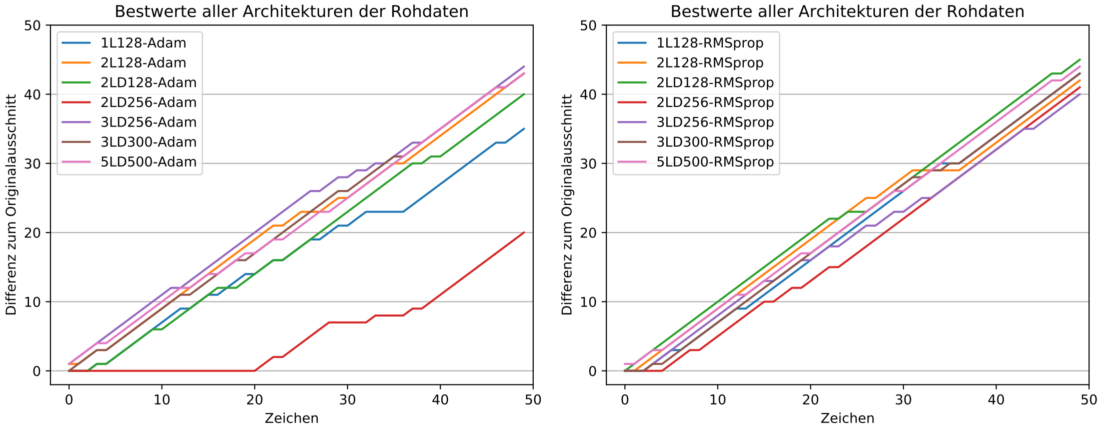

Abbildung A.36: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

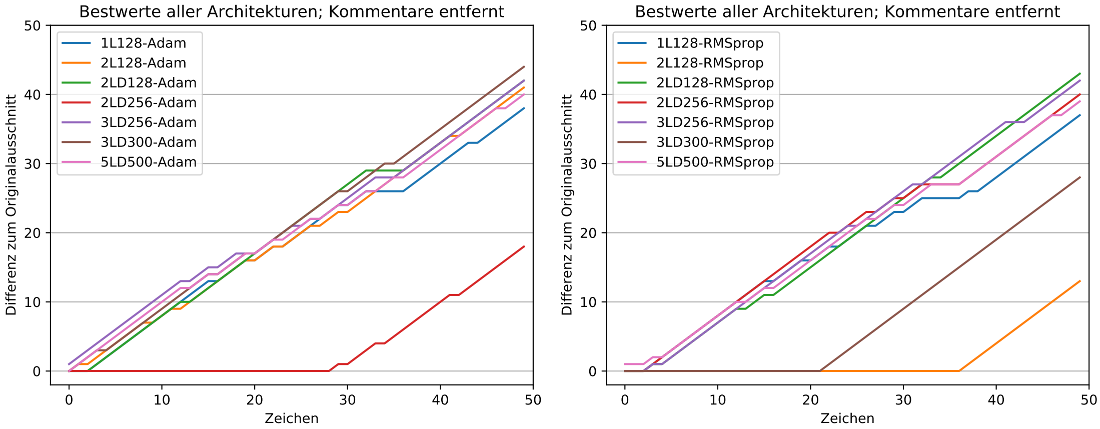

Abbildung A.37: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

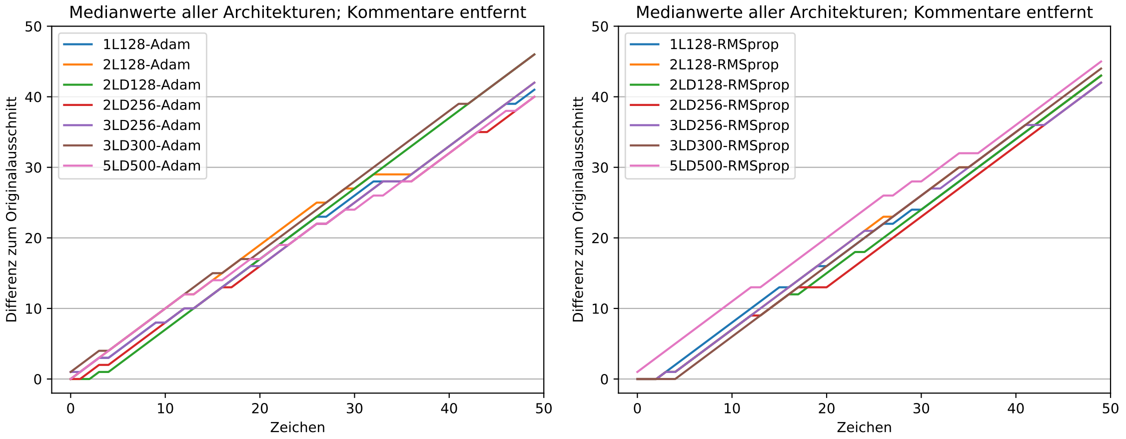

Abbildung A.38: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

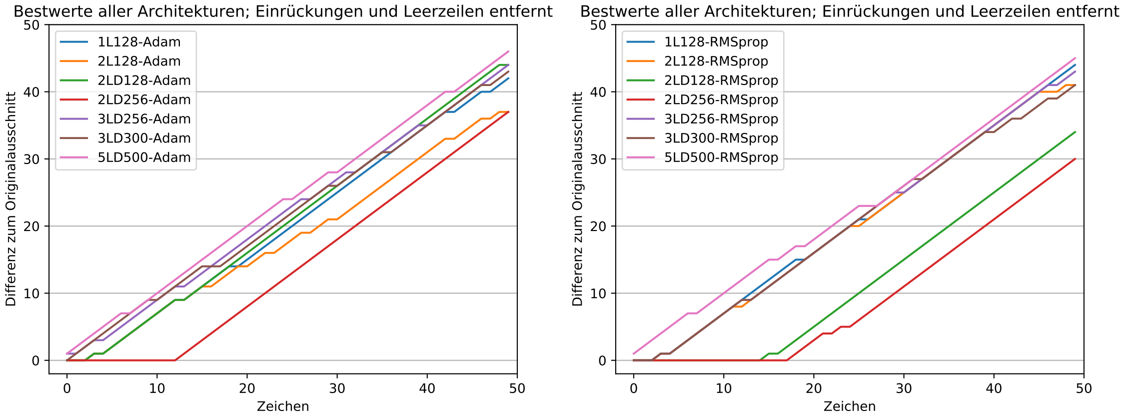

Abbildung A.39: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

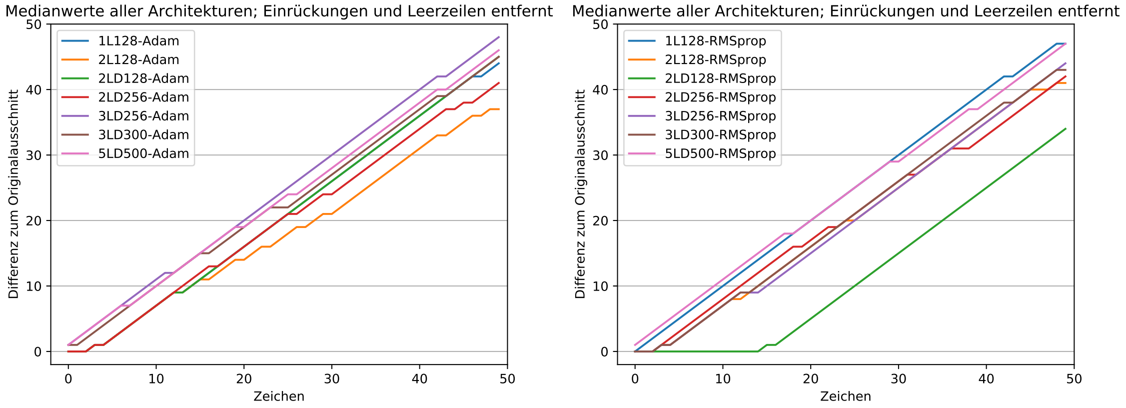

Abbildung A.40: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

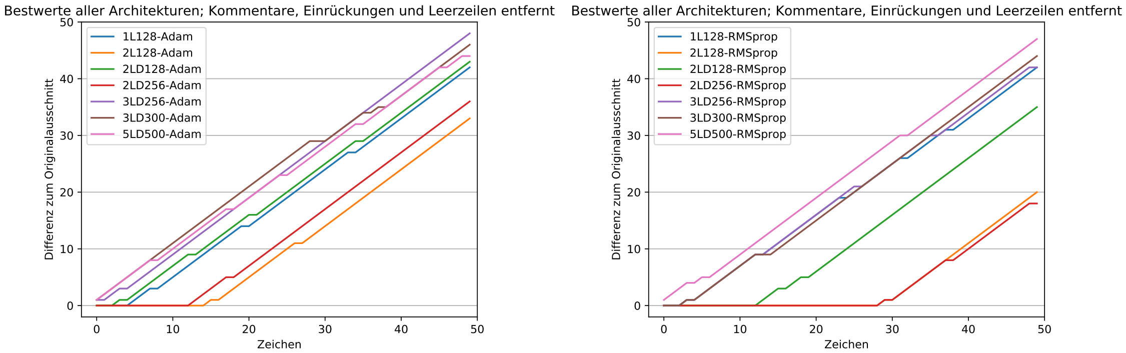

Abbildung A.41: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

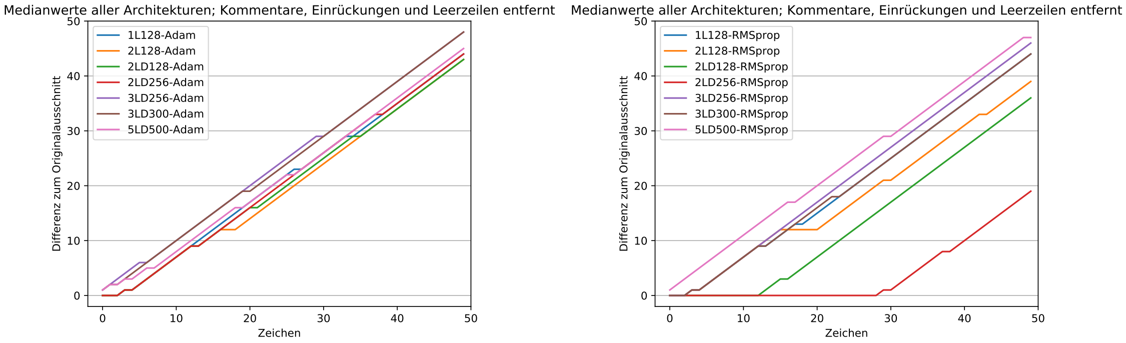

Abbildung A.42: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

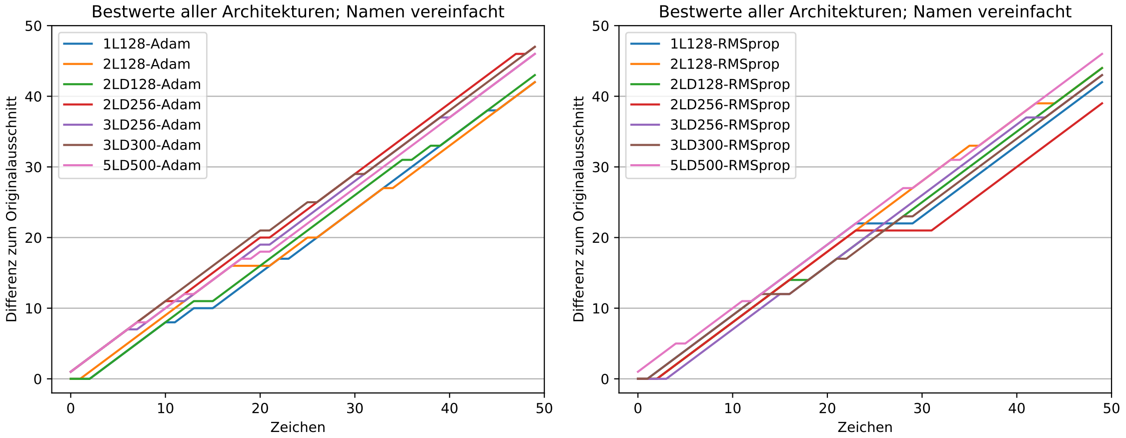

Abbildung A.43: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

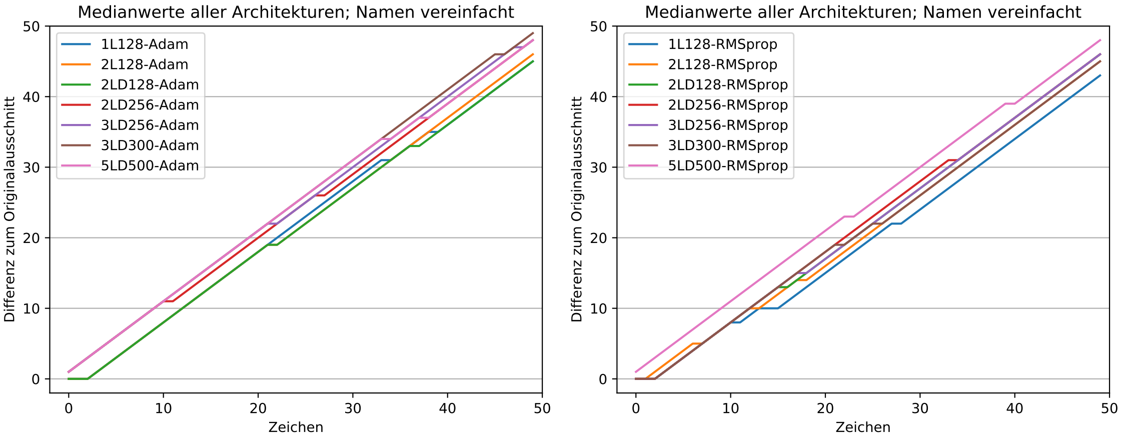

Abbildung A.44: Vergleich aller Architekturen, links `Adam`, rechts `RMSprop`

Eidesstattliche Erklärung

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Anne Peter