Information Security

Entwicklung eines flexiblen Systems für neuronale Netze zur Klassifikation von Malware

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

eingereicht von

Stefan Boubela BSc.
Is161502

im Rahmen des
Studiengangs Information Security an der Fachhochschule St. Pölten

Betreuung
Betreuer: FH-Prof. Dipl.-Ing. Dr. Sebastian Schrittwieser, Bakk.

St. Pölten, 30.06.2018

(Unterschrift Autor/Autorin)

(Unterschrift Betreuer/Betreuerin)

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

St. Pölten, am 30.06.2018

(Unterschrift Autor/Autorin)

Danksagung
Ich möchte zuallererst meinem Betreuer FH-Prof. Dipl.-Ing. Dr. Sebastian Schrittwieser für dieses interessante Thema, die Bereitstellung der NVIDIA Tesla Grafikkarte und konstruktive Gespräche danken.

Weiter möchte ich Dipl.-Ing. Robert Luh für die Bereitstellung der Daten danken, da diese Art von Daten selbst unmöglich erzeugt werden kann.

Ganz besonders danke ich auch Felix für die frühen Ideen wohin die Arbeit führen könnte, für die angeregten Diskussionen und die kritische Musterung meiner Ansätze bei einem Kaffee, sowie für das Korrekturlesen.

Melanie danke ich für die Geduld mir gegenüber und dafür, dass sie mich motiviert hat weiterzuarbeiten.

Meinem Vater danke ich für die kritisch konstruktiven Gespräche und den Versuch, mit mir meine Gedanken zu ordnen, obwohl er inhaltlich wenig Ahnung von der Thematik hatte.

Florian und Johannes danke ich für das aufmerksame Korrekturlesen und die guten Rückmeldungen.

Zusammenfassung

Neuronale Netze sind eine vielversprechende Technologie im Bereich künstlicher Intelligenz, um damit automatisch Klassifizierungen und Regressionen durchzuführen. In diversen Versuchen konnten diese Netze bereits beeindruckende Ergebnisse erzielen und dabei die Fähigkeiten des Menschen übertreffen. Auch in der Malware Erkennung gelten neuronale Netze als zukunftsträchtige Technologie und sollen die klassischen signatur-basierten Verfahren, welche gegen die heutigen Mengen und Methoden von Malware kaum eine Chance haben, ablösen. Während bereits Systeme existieren, die unter Laborbedingungen nahezu perfekte Ergebnisse erzielen fehlt es noch an gemeinsamen Grundlagen, sowie Verfahren für mehr Flexibilität und Alltagstauglichkeit. Diese Arbeit beschreibt daher umfassende Grundlagen zu neuronalen Netzen und häufigen Verfahren und Algorithmen, um eine einheitliche Ausgangslage zu schaffen und Entwicklungen in diesem Bereich zu fördern. Die Grundlagen reichen dabei von einfachen Fragen wie: „Was sind neuronale Netze?“, bis zu modernen Optimierungsverfahren wie Adaptive Moment Estimation und möglichen Problemen wie Overfitting und Covariate Shift, um das Thema möglichst vollständig abzudecken. Danach werden aufgrund der Vielzahl an verfügbaren Netzen und Frameworks einerseits passende Netze zur Erreichung der Zielsetzung dieser Arbeit und andererseits verschiedenste Frameworks zur Implementierung evaluiert. In späteren Kapiteln werden darauf basierend zuerst zwei Systeme theoretisch modelliert, verbessert und zuletzt praktisch in Keras implementiert. Entscheidungen und neue Verfahren werden dabei in simplen Versuchen praktisch untermauert. So werden beispielsweise Versuche zum Vergleich der Leistung von Feed-forward Neural Networks und Deep Belief Networks angestellt, aber auch eine Hyperparameter Optimierung durch automatisierte Tests mit über 2.000 Konfigurationen durchgeführt. Bei der praktischen Umsetzung des Systems wurde der Schwerpunkt auf konsistente Programmierung und ausführliche Kommentierung gelegt, um die möglichst einfache Weiterverwendung des Programmcodes zu ermöglichen. Begleitend dazu werden der Aufbau, die Funktionsweisen und die Bedienung des Programms ausführlich beschrieben. Das resultierende System besteht aus einem Recurrent Neural Network, welches die Daten zu einem kurzen Output verarbeitet, und zwei parallelen Classifiern, welche beide denselben Input erhalten und deren Ergebnis verglichen wird um das endgültige Resultat zu erhalten. Während das Recurrent Neural Network nicht trainiert wird, werden die Classifier mit supervised Learning auf die bestmögliche Rekonstruktion des Inputs trainiert, weshalb sie in dieser Arbeit als Rekonstruktionsnetze bezeichnet werden. Dieses System bietet die Funktionalität eines herkömmlichen Classifiers, während es außerdem mehr Flexibilität bei Training und Implementierung bietet. Zusätzlich besitzt es einen Mechanismus zur Steuerung von False-Positive Rate und Erkennungsrate. Des Weiteren ist das System durch das implementierte One-Hot Encoding Verfahren in der Lage, nahezu jeden Input ohne größere Anpassungen zu verarbeiten. Während die Vorversuche durchgängig positive Ergebnisse liefern, fallen die Ergebnisse des Hauptversuchs negativ aus. Abschließend werden die Ergebnisse zusammengefasst, mögliche Gründe für das negative Ergebnis diskutiert und eine substantielle Anzahl weiterer Möglichkeiten und Ansätze für zukünftige Arbeiten beschrieben.

Abstract

Neural networks are a promising technique in the field of artificial intelligence used for automated classification and regression. In various experiments, they have already achieved impressive results, surpassing human capabilities and performance. Neural networks are also regarded as promising technology in malware detection and should replace the classic signature-based approach, which has little chance against the current quantities and methods of malware. While systems still exist that achieve near-perfect results under laboratory conditions, there is still a lack of common foundations, as well as methods for more flexibility and everyday usability. Therefore, this paper begins by describing the basic principles of neural networks and common procedures and algorithms to create a consistent starting point and promote developments in this area. The basics range from simple questions such as, “What are neural networks?”, to modern optimization methods such as adaptive momentum estimation, as well as problems like overfitting and covariate shift, to cover the topic as completely as possible. Afterwards, due to the large number of available network architectures and frameworks, on the one hand suitable networks for achieving the set goal of this work, but also different frameworks for implementation are evaluated. In later chapters, two systems will be modelled theoretically and improved as to finally be implemented practically in Keras. Decisions and new procedures are practically substantiated in simple experiments. For example, attempts are being made to compare the performance of feed-forward neural networks and deep belief networks, as well as hyperparameter optimization is performed through automated testing with over 2,000 configurations. In the practical implementation of the system, a lot of emphasis was put on proper programming and detailed comments to ensure easy reuse of the program code. Accompanying this, the structure, the functions and the operation of the program are described in detail in this work. The resulting system consists of a recurrent neural network, which processes the data into a short output, and two parallel classifiers, both receiving the same input. By comparing the output of the two classifiers, the final result is determined. While the recurrent neural network is not trained, the classifiers are trained using supervised learning on the best possible reconstruction of the input; thus being, called reconstruction networks in this work. This system provides the same functionality as a traditional classifier, while providing more flexibility in training and implementation as well as having a mechanism for controlling false-positive rate and recognition rate. Additionally, due to the implemented one-hot encoding procedure, it is able to process virtually any input without costly adjustments. While the initial experiments yield consistent positive results, the results of the main experiment are negative. Finally, the results are summarized and possible reasons for the negative results discussed, concluding with descriptions of many other possibilities and experiments for future work.

Inhaltsverzeichnis

1 EINLEITUNG 12

1.1 PROBLEMSTELLUNG 12
1.2 ZIEL DER ARBEIT 12
1.2.1.1 Hypothesen 13
1.3 RELATED WORK 13

2 NEURONALE NETZWERKE 19

2.1 ENTSTEHUNG UND ENTWICKLUNG 19
2.2 AUFBAU 19
2.2.1 Aktivierungsfunktion 21
2.2.1.1 Step Funktion 22
2.2.1.2 Lineare Funktion 23
2.2.1.3 Sigmoid Funktion 23
2.2.1.4 Rectified Linear Unit 23
2.2.1.5 Softmax Funktion 24
2.2.2 Bias-Units 24
2.2.3 Hyperparameter 24
2.2.4 Initialisierung der Gewichtungen 25
2.2.4.1 Xavier Initialization 25
2.2.4.2 RELU/He Initialization 25
2.3 ARTEN VON NETZEN 25
2.3.1 Feed-forward Neural Networks 26
2.3.1.1 Autoencoder 26
2.3.1.2 Restricted Boltzmann Machine 26
2.3.1.2.1 Deep Belief Network 26
2.3.1.3 Pattern Associator 27
2.3.1.4 Recursive Neural Tensor Networks 27
2.3.1.5 Convolutional Neural Networks 27
2.3.2 Fast Neural Networks 28
2.3.3 Recurrent Neural Networks 29
2.3.3.1 Long- Short-Term Memory 30
2.3.4 Adaptive Resonanztheorie 30
2.3.5 Word2Vec 31
2.4 TRAINING 31
2.4.1 Trainingsarten 32
2.4.1.1 Supervised Learning 32
2.4.1.1.1 Kostenfunktionen 32
2.4.1.2 Reinforcement Learning 33
2.4.1.3 Unsupervised Learning 33
2.4.1.3.1 Unsupervised Pre-Training 33
2.4.2 Lernregeln 33
2.4.2.1 Hebb-Regel 33
2.4.2.2 Delta-Regel 34
2.4.2.3 Backpropagation 34
2.4.2.3.1 Backpropagation Through Time 34
2.4.2.4 Real-Time Recurrent Learning 35
2.4.2.5 Competitive Learning 36
2.4.3 Verbesserung 36
2.4.3.1 Allgemein 36

2.4.3.1.1 Online Training versus Offline Training ... 36
2.4.3.1.2 Verbesserungszeitpunkt ... 37
2.4.3.1.3 Lernrate ... 37
2.4.3.2 Optimierungsalgorithmus ... 37
2.4.3.2.1 Gradientenverfahren ... 37
2.4.3.2.2 Stochastisches Gradientenverfahren ... 38
2.4.3.2.3 Mini Batch Gradientenverfahren ... 38
2.4.3.2.4 Momentum ... 38
2.4.3.2.5 Adagrad ... 38
2.4.3.2.6 Adaptive Moment Estimation ... 39
2.4.4 Validation ... 39
2.4.5 Überanpassung ... 39
2.4.5.1.1 Dropout ... 39
2.4.5.1.2 DeCov ... 40
2.4.5.1.3 L1 Regularization und L2 Regularization ... 40
2.5 VERSCHIEDENES ... 40
2.5.1 Schritt, Iteration, Batch, Mini-Batch, Epoche ... 40
2.5.2 Feature ... 41
2.5.3 Dimensionalität ... 41
2.5.4 Covariate Shift ... 41
2.5.4.1 Internal Covariate Shift ... 42
2.5.5 Batch Normalization ... 42
2.6 ANWENDUNG ... 42
2.6.1 Anwendungsbeispiele ... 43
2.6.2 Probleme von neuronalen Netzen ... 43

3 UMSETZUNG ... 45

3.1 PRAKTISCHE ASPEKTE UND HERAUSFORDERUNGEN ... 45
3.1.1 Input Encoding/Embedding ... 45
3.1.2 Daten unterschiedlicher Form ... 46
3.2 EVALUIERUNG VON FRAMEWORKS UND LIBRARIES ... 46
3.2.1 Anforderungen ... 46
3.2.2 Analyse der Frameworks und Libraries für neuronale Netze ... 47
3.2.2.1 Tensorflow ... 47
3.2.2.2 Keras ... 48
3.2.2.3 Caffe2 ... 48
3.2.2.4 Theano ... 48
3.2.2.5 Torch oder PyTorch ... 49
3.2.2.6 MXNet ... 49
3.2.2.7 Gluon ... 49
3.2.2.8 Scikit-learn ... 49
3.2.2.9 Microsoft Cognitive Toolkit (CNTK) ... 49
3.2.2.10 Deeplearning4j (DL4J) ... 49
3.2.2.11 Passage Library ... 50
3.2.3 Fazit ... 50
3.3 EVALUIERUNG EINES PASSENDEN NETZES ... 50
3.3.1 Anforderung ... 50
3.3.2 Mögliche relevante Netze ... 50
3.3.3 Fazit ... 52
3.4 PRAKTISCHE EINARBEITUNG IN TENSORFLOW ... 52
3.4.1 Einstieg und Überblick ... 52

3.4.2 Kompilierung für CPU ... 54
3.5 Erster Ansatz ... 57
3.5.1 Aufbau ... 58
3.5.1.1 Input ... 58
3.5.1.2 Feature Extractor ... 59
3.5.1.3 RNN ... 59
3.5.1.4 Classifier ... 59
3.5.1.5 Ergebnis ... 60
3.5.1.6 Training ... 60
3.5.2 Fazit ... 60
3.6 Zweiter Ansatz ... 61
3.6.1 Erkenntnisse aus dem ersten Ansatz ... 61
3.6.2 Aufbau ... 63
3.6.2.1 Input ... 64
3.6.2.2 RNN ... 64
3.6.2.3 Classifier ... 64
3.6.2.4 Ergebnis ... 65
3.6.2.5 Training ... 65
3.6.2.6 Fazit ... 65
3.7 Praktische Umsetzung des Systems ... 66
3.7.1 Daten ... 67
3.7.1.1 Herkunft der Daten ... 67
3.7.1.2 Verarbeitung der Daten ... 67
3.7.1.3 Auszug aus den Daten ... 67
3.7.2 Erstellung des Codes ... 68
3.7.2.1 Programmstart ... 68
3.7.2.2 Input ... 69
3.7.2.3 RNN ... 69
3.7.2.4 Classifier ... 70
3.7.2.5 System ... 71
3.7.2.6 Training ... 71
3.7.2.7 Ergebnis ... 71
3.7.3 Anwendung ... 71

4 ERGEBNISSE ... 73

4.1 Diskussion über die Ergebnisse ... 74
4.2 Probleme und Einschränkungen ... 75

5 ZUKÜNFTIGE ARBEIT ... 77

5.1 Untrainiertes RNN ... 77
5.2 Unveränderbares RNN ... 77
5.3 False-Positive-Feedback ... 78
5.4 Quelle der Daten ... 78
5.5 One Hot Encoding Varianten ... 78
5.6 Vergleich mit herkömmlichem Classifier ... 79

6 APPENDIX A: PRAKTISCHE TESTS UND VERSUCHE ... 80

6.1 Versuchsreihe 1 ... 80
6.1.1 Aufbau ... 80
6.1.2 Versuch 1 ... 80
6.1.3 Versuch 2 ... 81

6.1.4 Source-Code ... 81
6.1.5 Fazit ... 81
6.2 VERSUCHSREIHE 2 ... 81
6.2.1 Aufbau ... 82
6.2.2 Versuch 1 ... 82
6.2.3 Versuch 2 ... 82
6.2.4 Source Code ... 83
6.2.5 Fazit ... 83
6.3 VERSUCHSREIHE 3 ... 84
6.3.1 Aufbau ... 84
6.3.2 Versuch 1 ... 84
6.3.3 Versuch 2 ... 84
6.3.4 Versuch 3 ... 85
6.3.5 Versuch 4 ... 86
6.3.6 Versuch 5 ... 87
6.3.7 Source Code ... 88
6.3.8 Fazit ... 88
6.4 VERSUCHSREIHE 4 ... 88
6.4.1 Aufbau ... 88
6.4.2 Versuch 1 ... 89
6.4.3 Versuch 2 ... 91
6.4.4 Source Code ... 91
6.4.5 Fazit ... 92
7 GLOSSAR ... 93
8 LITERATURVERZEICHNIS ... 94

Abbildungsverzeichnis

Abbildung 1: Ein schematisches neuronales Netz mit vier Schichten zur Klassifikation in vier Kategorien 20
Abbildung 2: Ein schematisches trainiertes neuronales Netz zur Klassifikation in vier Kategorien 21
Abbildung 3: Grafische Darstellung der notwendigen Elemente um das Signal für Neuron $\boldsymbol{a0(1)}$ zu berechnen 22
Abbildung 4: Die Step-Funktion 23
Abbildung 5: Graph der logistischen Funktion 23
Abbildung 6: Eine ReLU [29] 24
Abbildung 7: Die Trainingsabschnitte eines DBNs 27
Abbildung 8: Durchführung von Filtering eines Features im Convolutional Layer 28
Abbildung 9: Die möglichen Schritte eines CNNs 28
Abbildung 10: Ein RNN(links) aufgerollt über mehrere Zeitschritte(rechts) 29
Abbildung 11: Beispiel für ein SRN 29
Abbildung 12: Beispiel für ein LSTM. Zur besseren Übersicht sind nicht alle Verbindungen eingezeichnet. Die Abbildung ist [39] nachempfunden. 30
Abbildung 13: Zwei Beispiele für ein beschriftetes (engl. „labeled“) Tupel für supervised Learning 32
Abbildung 14: Berechnung der Kosten für den Output von 4 Neuronen 32
Abbildung 15: Links in der Abbildung ein einfaches RNN, rechts dasselbe RNN ausgerollt über 7 Schritte 35
Abbildung 16: So sieht beispielsweise TBPTT aus, die Updates werden hier immer nur 2 Schritte zurückpropagiert[50] 35
Abbildung 17: Verdeutlichung des Zusammenhangs zwischen Schritt, Batch, Epoche und Training. 41
Abbildung 18: One-Hot-Encoding Vektor, der gerade die Instruktion „CreateFile“ anzeigt 45
Abbildung 19: Tensorflow Stack mit den verschiedenen APIs 48
Abbildung 20: Verschiedene Graphen für die erste Layer eines Netzes zur MNIST-Klassifikation 53
Abbildung 21: Der Verlauf der Genauigkeit eines Netzes zur MNIST-Klassifikation beim Training (orange) und beim Test (blau) 53
Abbildung 22: Eine Tabelle für One-Hot Encoding für den ersten Ansatz 57
Abbildung 23: Aufbau des Systems für den ersten Ansatz 58
Abbildung 24: RNN mit einer Hidden Layer und einem Hidden State zurück in die Hidden Layer 59
Abbildung 25: Problem von zu hohen Gewichtungen, W stellt den Wert der Gewichtung dar 62
Abbildung 26: Der Aufbau des Systems im zweiten Ansatz, im Wesentlichen aus einem RNN und zwei Classifiern bestehend 64
Abbildung 27: Umwandlung eines Inputs mittels One-Hot Encoding. 69
Abbildung 28: Verteilung der Werte im Datensatz 75
Abbildung 29: Orange ist die logistische Verteilung, rot die Gaußsche Verteilung. Erstellt mit www.desmos.com 83

Tabellenverzeichnis

Tabelle 1: Hyperparameter des RNNs 64
Tabelle 2: Hyperparameter des Classifiers 65
Tabelle 3: Aufbau des Netzes, Hyperparameter und Trainingskonfiguration 80
Tabelle 4: Hyperparameter für die niedrigsten Kosten im Zuge der Versuche. 86
Tabelle 5: Hyperparameter für den größten Kosten-Unterschied zwischen den Datentypen im Zuge der Versuche. 87
Tabelle 6: Inputdaten, Trainingsziel und Outputdaten der einzelnen Netze 89
Tabelle 7: Zusammenhang zwischen vortrainierten Netzen, Gewichtsmatrizen und FFNN Layers 89
Tabelle 8: Aufbau und Hyperparameter des DBNs 90

1 Einleitung

1.1 Problemstellung

Die Cyber-Kriminalität steht in einem ständigen Kräftemessen mit den Entwicklern von Antiviren-Programmen, Intrusion Detection Systemen, Data Loss Prevention Systemen und vielen anderen Programmen zum Schutz der Systeme und deren Daten. Die darin involvierten Angreifer sind sehr vielfältig, von der organisierten Cyber-Mafia, über staatlich finanzierte Hacker, bis zum Mitarbeiter, der bestochen wurde und dafür Daten abzieht oder Programme installiert. Während die Angreifer bereits Mengen von neuartigen Techniken in Programmen nutzen, die teilweise monatelang in Firmen unentdeckt operieren können, sollten auch die Verteidiger drastisch aufrüsten und die eingesetzten Technologien von der statischen und signatur-basierten Analyse weiterentwickeln zu verhaltensbasierten, intelligenten Systemen. Laut dem M-Trends 2018 Bericht von Mandiant verbleiben Angreifer im Jahr 2017 im Schnitt beeindruckende 101 Tage unbemerkt, bis erste Nachweise für eine Kompromittierung gefunden werden, in EMEA und APAC liegt dieser Schnitt noch weit höher [1]. In Anbetracht dieser Herausforderungen wirken neuronale Netze sehr vielversprechend, da sie in den letzten Jahren stark weiterentwickelt wurden und in verschiedensten Bereichen erstaunliche und herausragende Ergebnisse erzielen. Gerade die schnelle und effiziente Erkennung von Mustern, die teilweise für Menschen nicht erkennbar sind, ist eine der großen Stärken. Wie im Kapitel 1.3 ausgeführt, existieren tatsächlich bereits im Bereich der Malware Erkennung beeindruckende Systeme mit nahezu perfekten Ergebnissen unter Laborbedingungen. Allerdings werden die Entwicklung und der breite Einsatz von Anti-Viren Systemen basierend auf neuronalen Netzen durch verschiedene Hürden eingeschränkt. Diese Probleme umfassen beispielsweise die fehlende Tauglichkeit für reale Anwendungsumgebungen durch generische Trainingsdaten oder zu lange Trainingszeiten, fehlende Flexibilität bei der Aktualisierung und Steuerung der Systeme und fehlende Grundlagen um neuronale Netze entwerfen und umsetzen zu können. Zuletzt mangelt es oft an vorhandenem Expertenwissen um die für die Systeme notwendigen Daten wie API Calls oder Signaturen laufend zu analysieren und zu kategorisieren.

1.2 Ziel der Arbeit

Die wesentlichen Beiträge dieser Arbeit sollen, begründet durch ihre Neuartigkeit, ein umfassender Überblick über Grundlagen der neuronalen Netze, ein flexibles System zur Klassifikation und ein Mechanismus zur Steuerung des Verhältnisses zwischen False-Positive Rate und Erkennungsrate sein. Die Aufarbeitung der Grundlagen soll einen umfangreichen Überblick über die Thematik geben und eine einheitliche Basis schaffen, durch die weitere Forschung und Entwicklung gefördert werden soll. Abgesehen davon sollen die Grundlagen dazu dienen, passende Netze und Verfahren für spätere Phasen der Arbeit zu identifizieren. Ein weiteres Ziel dieser Arbeit ist, ein auf neuronale Netze basierendes System zu entwickeln, welches lokal Systemevents von verschiedensten Vorgängen wie Programmen auswertet und darin Anomalien findet, die auf Malware oder Angriffe hindeuten. Dies soll dadurch bewerkstelligt werden, dass das System einerseits mit einigen Verhaltensmustern von Malware vertraut ist, andererseits einen gewissen Normalzustand des Systems kennt. Das heißt, das System kennt ungefähr die typischen Aktivitäten, typische Anwendungen mit Internetverbindung, wie Spotify und Outlook. Weiter sollen typische Aktionen von Programmen bekannt sein, welche DLLs geladen und welche Prozesse gestartet werden. Wie Lee et al.[2] erwähnen, gibt es empirische Nachweise, dass User Aktivitäten und Programmstarts eindeutig regelmäßige Muster zeigen, so öffnen und starten Programmierer öfter C-Files, greifen Programme mit höheren Rechten meist auf die gleichen Ordner und Files zu. Daher sollten auch die dadurch generierten Events regelmäßige Muster aufweisen. Das System soll durch ein flexibles Design ermöglichen, dass anstatt generischer, gutartiger Daten der Hersteller die tatsächlich beim Anwender vorkommenden gutartigen Daten für das Training verwendet werden können. Da sich herausgestellt hat, dass ein System mit zu vielen Fehlalarmen nicht mehr beachtet wird, besteht eine große Herausforderung darin, ein aufmerksames und kritisches System zu entwickeln und trotzdem möglichst wenige Fehlalarme zu erzeugen. Die Steuerung von False-Positive Rate und Erkennungsrate soll helfen, das System besser an den Anwendungsfall und die Umgebung des Systems anzupassen. Es existieren gerade im Bereich der kritischen

Infrastruktur relativ stark isolierte Systeme, beispielsweise in Kraftwerken, deren ständiger Betrieb wichtig für die Versorgung der Bevölkerung ist. Dort ist es je nach Risiko eventuell besser, weniger Viren zu erkennen und dafür auf keinen Fall eine fehlerhaft erkannte Datei (False-Positive) zu entfernen und damit eventuell das System zu Absturz zu bringen. Der Virus kann weder Daten abziehen, noch einen Remote Zugang herstellen, außerdem ist er durch fehlende Verbindung zu einem C&C Server eventuell sogar harmlos. In vielen Fällen hat ein Virus damit nicht so extreme Auswirkungen, wie es beispielsweise ein gelöschtes System-File hätte. Auf der anderen Seite hat eine Firma ein exzellentes Backup Konzept und führt in sehr kurzen Zeitabständen Backups durch. Für diese Firma ist die False-Positive Rate vermutlich unwichtig, da gelöschte Dateien oder abgestürzte System leicht wiederhergestellt werden können. Viel wichtiger ist hingegen, dass alle Viren gefunden werden, die das ganze Netzwerk infizieren oder Daten absaugen könnten. Das System soll die Erkennung nicht nur auf File oder Prozess-Analyse zu beschränken, sondern flexibel Systemevents verarbeiten und darin Muster für einen Angriff oder Aktivitäten von Malware identifizieren. Dies würde den Fokus von dem leicht veränderbaren und sogar versteckbaren Programmcode von Malware auf die eigentlich wichtigen Punkte lenken, nämlich das, was die Malware am System macht. Damit soll es auch möglich sein, unscheinbarere Vorgänge erkennen zu können, um einen langsam aber beständig voranschreitenden, über Monate angesetzten, Angriff erkennen und beenden zu können. Damit würden sich alle im Moment gängigen Angriffe und Malware abwehren lassen, da alles Systemevents erzeugt: von APTs, Programmcode-loser Malware, manuellen Angriffsschritten, bis Zero-Day Exploits und einer Kombination aus all dem. Einzig ausgenommen natürlich Zero-Day Exploits, welche die aufzeichnende Software umgehen können. Schlussendlich soll dieses System den größten Nachteil von Sandboxes mit verhaltensbasierter Analyse ausgleichen, nämlich dass der Fokus nur auf Files liegt und diese gezielt geprüft werden müssen, da es auch durch Web-Exploits oder über USB Stick in das Unternehmen gelangte Malware erkennt. Diese Entwicklung folgt auch dem Trend, nicht wie früher nur den Perimeter zu sichern und das interne Netzwerk ungeschützt zu lassen, sondern auch das interne Netzwerk und die Systeme gut zu schützen, da es immer Angriffe und Malware geben wird, die den Perimeter mühelos überwinden oder ihren Weg gar nicht über diese Grenze in das Unternehmen finden. Zuletzt soll die Notwendigkeit von Expertenwissen zur Erstellung der Trainingsdaten möglichst geringgehalten werden. Dies ermöglicht, dass für die regelmäßige Überarbeitung und Aktualisierung der Trainingsdaten benötigte Ressourcen leichter verfügbar sind. Die Ergebnisse dieser Arbeit werden in Kapitel 4 behandelt.

1.2.1.1 Hypothesen

Folgende Fragestellungen lassen sich aus dem Ziel dieser Arbeit ableiten und sollen mithilfe dieser Arbeit beantwortet werden:

- Ist es möglich, mithilfe eines speziellen Verfahrens ein Netz zur Klassifizierung nur mittels einer Kategorie x zu trainieren, insofern, dass das Netz danach feststellen kann, ob ein Input zur Kategorie x gehört oder nicht?
- Wie könnte ein System aussehen, mit dem dieses Verfahren zur Klassifizierung umgesetzt wird?
- Ist es möglich, das Verhältnis aus Erkennungsrate und False-Positive Rate bei Netzen zur Erkennung von Malware manuell zu steuern, um damit auf das Bedrohungsszenario des Anwenders eingehen zu können?

1.3 Related Work

Es gibt bereits einige Forschungsarbeiten, die sich mit der Erkennung beziehungsweise Klassifikation von Malware mithilfe von neuronalen Netzen beschäftigen. Dabei reichen die Ansätze von statischer Fileanalyse über API Call Analyse hin zur Analyse von High-Level Events. Es werden eine Vielzahl an unterschiedlicher Netze verwendet und die Verfahren unterscheiden sich mitunter stark in Bezug auf manuellen Aufwand und benötigte Rechenleistung.

Raff et al.[3] versuchen Malware anhand von statischen Analysen des Bytecodes mithilfe von neuronalen Netzen zu entdecken. Dazu steht ihnen ein beachtlicher Datensatz von knapp über 2 Millionen Files zur Verfügung, von

denen circa jeweils die Hälfte Malware beinhaltet und die andere Hälfte ungefährliche Files sind. Um bei den teilweise sehr großen Binaries den Speicher nicht zu überladen, werden bei dem Ansatz sehr aggressive Pooling Layers und ein hoher Stride-Wert eingesetzt, durch die die Datenmenge reduziert werden. Alles in allem enthält das vorgestellte Netz 134.632 trainierbare Parameter. Von der Idee, einen Text-Input in ein Bild mit Graustufen umzuwandeln und dann die weit fortgeschrittene Entwicklung in der Bilderkennung zu nutzen um Muster zu identifizieren, wird in diesem Paper unter berechtigten Argumenten abgeraten, weswegen dieser Ansatz in dieser Arbeit nicht weiterverfolgt wird. Das aufwendige Training wurde auf insgesamt 8 NVIDIA Tesla P100 GPUs durchgeführt.

Eine ähnliche Art der Malware Analyse wird angeblich auch von verschiedenen Firmen wie DeepInstinct[4] oder Invincea[5] kommerziell eingesetzt, wobei Invincea dies laut Angabe mit dynamischer verhaltensbasierter Erkennung kombiniert. Vermutlich lassen sich aber nur durch Analyse von Binaries Angriffe, die mehrere Binaries kombinieren oder bei denen manuelle Schritte durchgeführt werden, nicht erkennen.

Huang und Stokes[6] nutzen ein Netz für die Erkennung von Malware als auch für die Klassifikation, welches mithilfe von insgesamt 6,5 Millionen Files trainiert wird. Die Erkennung wird auf Basis von Daten aus der dynamischen Analyse durchgeführt und das Netz erreicht eine beeindruckende Error Rate von 0,358%. Die dynamische Analyse dürfte sich in diesem Fall jedoch darauf beschränken, dass die Binaries in einer vereinfachten anti-malware engine ausgeführt werden um API Calls aufzuzeichnen, sowie Strings und Code-Ausschnitte („Null-terminated objects“) aus dem Speicher wiederherzustellen. Durch den Fokus auf API Calls müssen in dieser Arbeit allerdings die API Calls klassifiziert werden, zum Beispiel kann man Files erstellen mit CreateFile() im User Mode, ZwCreateFile() im Kernel Mode oder fopen() in C. Diese Analyse ist bei dem in dieser Arbeit beschriebenen Ansatz überflüssig, da alle drei API Calls zu einem Event für File Access/File Creation führen. Dadurch ist der beschriebene Ansatz auch tolerant gegenüber neuer Befehle, mit denen man beispielsweise auch ein File anlegen kann, ohne das Netz erneut trainieren zu müssen. In dem Paper wird ein neuronales Netz mit 12.102 Neuronen, verteilt über Input Layer, Output Layer und 4 Hidden Layers, verwendet, was bei fully-connected Layers zu einer hohen Anzahl von circa 20 Millionen Gewichtungen führt. Die Extraktion der Daten aus den Files dauerte circa zwei Wochen auf einem Computer, das Training des Netzes dauerte auf einer NVIDIA Tesla K40 GPU etwa 18 Stunden.

Erhan et al.[7] beschäftigen sich mit den Vorteilen für ein Deep Neural Network durch die Anwendung von unsupervised Pre-Training. Dieses Verfahren verbessert das teilweise sehr schwerfällige oder unmögliche Training von sehr tiefen Netzen oder ermöglicht es erst. Während die Methode bereits bekannt ist, ist wenig über die Hintergründe für den Erfolg erforscht. Weiter stellen die Autoren fest, dass sich das Pre-Training positiv auf die Fähigkeit des Netzes zu generalisieren auswirkt, sie vermuten, dass es an der Positionierung der Gewichtungen nahe einem Minimum liegt. Den Grund, wieso Pre-Training gerade bei sehr tiefen Netzen gut hilft, sehen sie darin, dass bei tiefen Netzen die Chance viel höher ist, dass der Gradient in ein lokales Minimum sinkt, aus dem er nicht mehr herauskommt und das Netz sich nicht weiter verbessert. Durch Visualisierung der Gewichtungen zeigt sich außerdem am klassischen Beispiel der Analyse MNIST Handschriften-Datenbank deutlich, dass die Features in einem vortrainierten Netz deutlich besser ausgebildet sind und eine deutlich höhere Qualität aufweisen. So sind in den tieferen Schichten bei einem vortrainierten Netz bereits Muster von Zahlen sichtbar, während man bei einem Netz ohne Pre-Training lediglich Rauschen sieht. Außerdem dürfte sich die Anwendung von Pre-Training auch grundlegend wie eine Regularization Methode auswirken, so weisen kleine Netze durch Anwendung von Pre-Training immer schlechtere Ergebnisse auf, als kleine Netze ohne Pre-Training. Dies führen die Autoren auf einen Effekt ähnlich anderer Regularization Methoden wie beispielsweise Dropout zurück.

Auch von Rieck et al.[8] existieren Methoden zur verhaltensbasierten Erkennung von Malware, basierend auf deren ausgeführten Operationen und Aktivitäten vor allem mit Fokus auf System Calls. Allerdings erfordert diese Methode, dass ein File gezielt in einer Sandbox ausgeführt und analysiert wird. Dadurch werden Files, die nicht

über die vorgesehene Verbindung in das Netzwerk kommen, nicht gescannt. Interessant ist jedoch der Ansatz für Classification und Clustering mithilfe von Prototypen im Vektorraum, den sie verwenden.

Ciresan et al.[9] kombinieren verschiedene Erkenntnisse aus dem Bereich Biologie und Data Science und zeigen mithilfe von Multi Column Deep Neural Networks (MCDNN) und Kombination von Aktivierungs-Funktionen wie ein Netz die Leistung von allen bisherigen Netzen und teilweise auch von Menschen übertrifft. Durch eine Implementierung für die GPU ergeben sich schnelle Trainingszeiten und kurze Durchlaufzeiten.

Tan et al.[10] beschäftigen sich mit neuronalen Netzen unter der Verwendung von ART und temporal difference (TD), um ein System zu schaffen, welches über direktes und zeitverzögertes Feedback lernt, also reinforcement Learning. Dieses Modell wird anhand einer Simulation getestet, in der das System ein Minenfeld in gegebener Zeit durchqueren und ein gewünschtes Ziel erreichen muss, ohne eine Mine auszulösen. Dafür erhält das System Informationen aus fünf Sensoren und muss lernen über mehrere Runden eine Strategie zu entwickeln, auch wenn sich das Minenfeld jede Runde ändert. Das verwendete Netz dürfte ein Feed-forward Neural Net mit drei Layers sein, anscheinend 18 Neuronen in dem Input Layer, 36 Neuronen in dem Hidden Layer und ein Neuron in der Output Layer. Als Aktivierungsfunktion wurde eine symmetrische Sigmoid-Funktion gewählt.

Benchea et al.[11] nutzen ein einseitiges Perceptron (OSP), eine Abwandlung des Perceptron-Algorithmus aus einer früheren Forschungsarbeit, und versuchen damit die beiden großen Probleme der Malware Analyse, nämlich eine hohe Erkennungsrate und eine niedrige False-Positive Rate, zu lösen. Da die gute Funktion des Perceptrons davon abhängig ist, dass viele Features vorhanden sind, konzentriert sich das Paper außerdem auf die Erstellung von Features mithilfe einer Restricted Boltzmann Machine (RBM). Von den Autoren wird ebenfalls ein riesiger Datensatz von über 3 Millionen Files genutzt. Für den vorgestellten Ansatz schalten [11] die RBM als erste Schicht und ein OSP als zweite Schicht hintereinander. Das grundlegende Ziel ist es, ein System mit genau Null False-positives für das Training zu erstellen und damit sicherzugehen, dass auch bei den Tests und der praktischen Anwendung die False-positive Rate entsprechend gering ist. Dies wird vor allem dadurch erreicht, dass das OSP so lange auf die ungefährlichen Files trainiert wird, bis alle Files richtig erkannt werden, oder eine Anzahl an Durchläufen erreicht wurde. Von jedem File wurden 3299 Features extrahiert.

Auch Vatamanu et al.[12] konzentrieren sich in ihrem Paper auch auf File-basierte Features, wie Typ des Compilers, Typ des Packers, File Geometrie und ähnliches, und stellen insgesamt über 6.000 Features pro File auf. Zusätzlich benutzen sie aber auch binäre Features wie die Fragen, ob sich das File ins Internet verbindet, Files herunterladet, andere Files modifiziert und ähnliches. File Infectors, Keygens und Adware wurde aus dem Datensatz mit 2,2 Millionen Files entfernt, da sie laut Meinung der Autoren schwer oder nicht zu erkennen sind und daher nur für ein störendes Rauschen in dem Datensatz sorgen. Der von den Autoren vorgestellte, weiterentwickelte One Side Class (OSC) 3 Algorithmus, der auf den Grundlagen eines Perceptrons basiert, verfolgt den Ansatz, dass ein OSC Netz alle bösartigen Files richtig klassifizieren soll und ein zweiter OSC alle gutartigen Files richtig klassifizieren soll, erkennbar durch jeweils eine Hyperebene als Abtrennung. Damit ergibt sich zwischen den beiden resultierenden Hyperebenen ein Raum, mit nicht eindeutig klassifizierten Files, über der oberen Ebene und unter der unteren Ebene aber sind alle Files richtig klassifiziert. Diese richtig klassifizierten Files werden aus der Menge entfernt und der Prozess startet erneut.

Pascanu et al.[13] verwenden ein neuronales Netz, um eine Sequenz aus mehreren verknüpften Events in Verbindung zu setzen und auf einen kleineren Output zu reduzieren. Dieser Output wird dann als Input für einen Classifier verwendet, der in der Trainingsphase auf gewisse Malware Samples trainiert wurde. Als Classifier verwenden sie logistische Regression und als Vergleich ein Multi-Layer Perceptron (MLP).

Für die Projection Stage zu Beginn werden RNN und ESN verglichen, jeweils kombiniert mit einer der Methoden Half-Frame, Max-Pooling, Bi-Directional, Leaky-Units und abschließend nur das Netz selbst. Die Bi-Directional Methode soll helfen auch längere Sequenzen verarbeiten zu können, indem ein Netz darauf trainiert ist, die Sequenz von vorne nach hinten zu verarbeiten, während ein zweites Netz darauf trainiert ist, die Sequenz von

hinten nach vorne zu verarbeiten, und das Ergebnis wird kombiniert. Außerdem ist laut Autoren auch der Einsatz von Max-Pooling bei RNNs neuartig, da es sonst meist zur optischen Mustererkennung in CNNs verwendet wird. Sie setzen es so ein, dass das aktivste Neuron der Hidden Layer in jedem Zeitschritt als Ergebnis genommen wird. Dieses Verfahren ist aber für die Problemstellung notwendig, da auch bei langen ungefährlichen Sequenzen ein kurzer auffälliger Teil herausstechen muss. Ein interessanter Ansatz ist auch das Half-Frame Verfahren, bei dem nicht nur das Endergebnis des Vorgangs, sondern auch ein Zwischenstand verwendet wird, da der Zwischenstand besser die Events von Beginn bis dorthin repräsentiert, während der normale Endstand meist besser die Events zum Ende hin repräsentiert. Mit diesen Verfahren soll quasi die Speicherfähigkeit des Netzes erhöht werden. Die Netze in der Projection Stage werden mit unsupervised Learning analog zu Language Modelling trainiert, mit dem Ziel das nächste Event vorhersagen zu können. Genaue Angaben über die Anzahl der Hidden Layers und deren Neuronen fehlen leider. Für die RNNs wird eine Dropout Rate von 0.5 eingesetzt, eine Lernrate von 1.0, allerdings fehlt Information über das verwendete Optimierungsverfahren. Die Autoren verarbeiten zwar High-Level Events, allerdings direkt auf einen Prozess oder ein File bezogen, daher wissen sie, wie viele und welche Events zusammengehören. Weiter dürften sie die zeitliche Dimension nur dadurch berücksichtigen, in welcher Reihenfolge die Events dem Projection Stage Netz präsentiert werden, aber eine Zeitangabe selbst wird nicht verarbeitet. Der verwendete Datensatz aus 500.000 Files war zur Hälfte ungefährlich und zur anderen Hälfte bösartig. Die analysierten Events werden anscheinend durch die Ausführung in einer Anti-Malware Engine gewonnen und es gibt 114 unterschiedliche High Level Events. High Level bedeutet in dem Fall, dass die Autoren für drei verschiedene API Calls um ein File zu öffnen nur ein generisches Event "File Öffnen" benutzen, was zu mehr Flexibilität und besserer Erkennung von geringen Code-Änderungen führt. Diese Definition von High Level Events dürfte manuell durchgeführt worden sein.

Moskovitch et al.[14] richten ihre Arbeit speziell auf die Erkennung von unbekannten Würmern aus, auch wenn die Methode für andere Arten von Malware ebenfalls anwendbar ist. Dafür versuchen sie ähnlich wie ein HIDS den Zustand eines Computers zu beobachten, das Wissen darüber soll sich das System allerdings selbst aneignen, mit einem kleinen Datensatz an bekannten Würmern als Hilfe. Das Ziel ist, dass manuelle Feature Auswahl oder Klassifikationen hinfällig ist, was einiges an Zeit spart. Zu Testzwecken wurde ein vollständiges Netz mit mehreren Computern aufgebaut. Zu Vergleichszwecken wurden vier Mechanismen zur automatischen Feature Auswahl und vier Classifier verwendet. Die vier Classifier sind Entscheidungsbaum, Naive Bayes, Bayesian Network und ANN. Leider beschreiben die Autoren die Features nicht, sondern erwähnen nur, dass für das Experiment 323 Computer Features überwacht wurden.

Schultz et al.[15] extrahieren anscheinend automatisch aus jedem File ein sogenanntes statisches "Binary Profile", aus denen dann Features erstellt werden. Mithilfe der Features trainieren die Autoren drei verschiedene Classifier: RIPPER, Naive-Bayes und ein Multi-Naive Bayes System. Ein RIPPER ist ein inductive Rule Learner, der beispielsweise aus vier grundlegenden Fragen besteht, mit deren Hilfe die Frage nach der Bösartigkeit beantwortet werden können soll. Die vier Fragen bezogen auf eine Executable X sind zum Beispiel "Hat X eine GUI?", "Führt X eine bösartige Funktion aus?", "Kompromittiert X die Systemsicherheit?" und "Löscht X Files?". Durch Analyse des Codes sollen diese Fragen beantwortet und eine finale Entscheidung getroffen werden können. Ihr Datensatz besteht aus knapp über 4.000 Files, was für das Training eines einfachen Classifiers durchaus ausreichend sein kann.

Wastian[16] beschäftigt sich mit Anomalien in Zeitserien. Ziel dieser Arbeit ist es eine Methode zu finden um feststellen zu können, ob ein Event einen großen Einfluss auf die zukünftige Entwicklung einer Zeitreihe haben wird, oder nicht. Ein Beispiel für großen Einfluss wäre, wenn Quartalsgewinne einer börsennotierten Firma weit schlechter ausfallen, als erwartet. Ein Beispiel für kleinen Einfluss wäre, wenn ein Server ausfällt, der durch Redundanz abgesichert ist. Um dies im Zusammenhang auf Server Ausfälle feststellen zu können, werden über 1.400 Variablen, die über den Zustand eines Servers aussagen, von Experten per Hand bewertet. Diese Variablen beinhalten beispielsweise aktuelle Informationen über die verbauten Festplatten oder wie viele Konflikte es in einem Programm gab. Die Variablen werden manuell von 0 bis 3 priorisiert, wodurch die Einschätzung sehr subjektiv aber extrem aufwendig ist. Danach soll ein neuronales Netz mit diesen Mustern

angelernt werden um bei zukünftigen Daten entscheiden zu können, ob die Werte der Variablen auf einen Ausafll hindeuten.

Lee et al.[2] stellen in ihrer Arbeit ein neues Konzept für Intrusion Detection System (IDS) vor, um bisherige Probleme wie eingeschränkte Logdaten-Unterstützung und aufwendige, teure Updates zu beheben. Statt zeitintensiver Modellierung von Angriffen durch Experten wollen sie mithilfe von Data Mining ein System schaffen, dass sich unabhängig der Logdaten-Quelle selbstständig Angriffsmuster aneignet und diese dann korrekt identifiziert. Dafür wollen sie Classification, Link Analysis und Sequence Analysis verwenden. Für Classification werden klassische Classifier verwendet, durch Link Analysis sollen zum Beispiel Gewohnheiten und Tätigkeiten von Usern erkannt werden, da diese oft ähnliche Befehle ausführen und Programme starten. Mit Sequence Analysis wollen sie Events in zeitlicher Abhängigkeit betrachten und beispielsweise Events identifizieren, die gewöhnlich nacheinander oder eben nicht nacheinander stattfinden, damit würde sich beispielsweise ein DOS-Angriff identifizieren lassen. Allerdings erfordert das vorgestellte Konzept, dass die Daten vorverarbeitet werden, so werden Netzwerkdaten beispielsweise zu High Level Events wie Host Sessions und Network Connections verarbeitet. Metadaten wie Verbindungsdauer, übertragene Bytes und ähnliches werden auch erfasst und dazugeschrieben. Anscheinend ist diese Vorverarbeitung aber nicht Teil des Konzepts, sondern eine Anforderung damit das IDS funktioniert und kann beispielsweise mit Systemen von Bro und NFR durchgeführt werden. Durch diese Vorverarbeitung umgehen sie das Problem, dass sie eine Menge an nicht eindeutig verknüpfbaren Events haben und durch Permutation alle Kombinationen ausprobieren müssen. Die Classification wird mit Ripper durchgeführt, der im Paper eigentlich eher statistisch Daten auswertet. So ist anscheinend manuell definiert, dass bei Telnet Verbindungen Zugriff auf System Ordner, Erstellung und Ausführen von Programmen den Hot-Counter erhöht, bis ab einem gewissen Schwellwert ein Alarm ausgelöst wird. Die Link Analysis beruht darauf, dass statistisch nachweisbar User Aktivitäten und Programme regelmäßige Muster aufweisen. Ein Programmierer führt oft C-Files aus und legt damit auch viele Programme an, ein Programm mit hohen Berechtigungen greift meist auf die gleichen System-Ordner und Files zu. Diese Verhaltensmuster sollen durch Link Analysis zu Normal Usage Profiles zusammengefasst werden. Damit diese Daten aber sinnvoll ausgewertet werden können, müssen anscheinend manuell pro Typ von Log Daten essential Features festgelegt werden, das sind Werte, die eine Zeile eindeutig identifizieren. Im Fall von Netzwerkdaten sind das beispielsweise Zeitstempel, Source und Destination Host, Source und Destination Port. Dies wirkt vergleichbar mit Primary Keys in Datenbanken. Aus diesen Daten werden dann automatisch Regeln abgeleitet, die Normalverhalten beschreiben sollen, wie zum Beispiel: „Wenn Benutzer X den Befehl vi nutzt um ein File am Host Test zwischen 8:00 und 16:00 zu öffnen, dann ist es zu 80% ein Shell-File.“ Für die Sequence Analysis wird der Ansatz für Link Analysis verwendet und die Events in eine zeitliche Reihenfolge gebracht. Dadurch werden auch die entstehenden Regeln von der Art her übernommen, bekommen einfach die zeitliche Abfolge dazu. Die Prozedur zur Erstellung dieser Regeln dürfte aber ebenfalls manuell festgelegt worden sein, nur die Durchführung ist automatisch. Dadurch ist auch erhebliches Expertenwissen notwendig, da man Angriffe und ihre Eigenschaften kennen muss. Es wurde zwar nicht explizit erwähnt, aber diese Forschungsarbeit konzentriert sich ausschließlich auf Network Intrusion Detection, die erwähnten Beispiele mit Programmierer und Programmen waren nur über Telnet-Verbindungen relevant.

Da Overfitting ein großes Problem für neuronale Netze darstellt und effektive Gegenmaßnahmen ein großes Potential für leistungsfähigere Netze schaffen, beschäftigen sich Cogswell et al.[17] in ihrer Arbeit mit einer neuen Methode gegen Overfitting, genannt DeCov. Als grundlegende Ursache für Overfitting beschreiben die Autoren Co-Adaption, die auftritt, wenn zwei oder mehr Hidden Units aufeinander angewiesen sind um eine Übereinstimmung mit den Trainingsdaten zu erzielen, diese Units weisen dann eine hohe Korrelation auf. Dropout beispielsweise verhindert dieses Problem, in dem immer verschiedene Neuronen deaktiviert werden, wodurch sich keine hohe Co-Adaption einstellen kann, was in der Arbeit durch zwei Graphen untermauert wird. Im Falle von Dropout dürfte dies allerdings eher zufällig passieren, eine explizite De-Korrelation wird damit nicht versucht. Dieser Zusammenhang wurde allerdings bisher noch nicht untersucht, weshalb die Autoren in dieser Arbeit untersuchen, ob eine Verringerung der Co-Adaption auch immer zu einer Verringerung oder Vermeidung von Overfitting führt. Als Ergebnis stellen sie ein Verfahren namens DeCov Loss vor, welches laut Arbeit zu

jedem existierenden Netz ohne weitere Überarbeitung hinzugefügt werden kann. Bei verschiedenen Experimenten führt die Anwendung von DeCov zu weitaus besseren Ergebnissen als ohne Overfitting-Maßnahme und zu besseren Ergebnissen als mit Dropout. Für den besten Effekt sollte für ein Training Dropout deaktiviert und die Ergebnisse verglichen werden. Leider dürfte DeCov aber noch nicht von Frameworks für neuronale Netze unterstützt werden.

In der Arbeit von Kirda et al.[18] wird ein Konzept vorgestellt, mit dem Spyware auf Basis von statischen und dynamischen Verfahren erkannt werden soll, auch wenn sie bisher unbekannt ist. Dafür konzentrieren sich die Autoren allerdings ausschließlich auf Spyware, die Internet Explorer Browser Helper Objects (BHO) und Tool Bars verwendet, um den User auszuspähen oder die Kontrolle über die Browser Session zu übernehmen. Mittels der dynamischen Analyse soll eigentlich nicht gezielt Spyware entdeckt werden, sondern Spyware-ähnliches Verhalten, wodurch Generalisierung und Unabhängigkeit von einer einzigen Spyware-Art sichergestellt werden soll. Dies wird grundlegend dadurch festgestellt, ob ein Objekt spezielle Aktionen im Browser durchführt oder spezielle API Calls absetzt. Für die Durchführung der dynamischen Analyse wird dem untersuchten Objekt eine Reihe von Browser Events präsentiert, die User Aktivitäten simulieren sollen, und die Reaktionen und darauffolgenden Aktionen des Objekts beobachtet, ähnlich einem Blackbox-Test. Wird ein Objekt als gefährlich eingestuft, kommt noch die statische Analyse dazu, die das File statisch analysiert und dabei nicht nur Web-relevante API Calls und Aktionen berücksichtigt, um beispielsweise auch Zugriffe aufs Filesystem zu entdecken.

Eine Gemeinsamkeit von vielen hier beschriebenen Ansätzen ist, dass sie ein hohes Wissen über Malware voraussetzen, um Events, API Calls und ähnliches richtig für die Trainingsdaten klassifizieren und einordnen zu können. Darüber hinaus bieten die Netze wenig Flexibilität bei der Anwendung, es müssen beispielsweise alle Kategorien für ein Netz gleichzeitig trainiert werden. Gerade durch die häufig anfallenden Updates im Bereich Malware ist dadurch regelmäßig hoher Aufwand von Experten notwendig. Im Gegensatz dazu soll der in dieser Arbeit beschriebene Ansatz nur sehr grundlegend klassifizierte Daten in gutartig und bösartig für das Training erfordern. Dies könnte theoretisch durch andere statische Systeme passieren oder durch getrennte Erhebung der Kategorien automatisch erfolgen.

2 Neuronale Netzwerke

Da es keine aktuelle zusammenfassende Literatur zu neuronalen Netzen gibt, ist es notwendig einen Überblick über den Aufbau, die Typen von Netzen und verschiedene Algorithmen und Verfahren, wie Aktivierungsfunktion, Kostenfunktion und Optimierungsalgorithmus zu schaffen. Durch fehlende Definitionen und Standardisierungen ergibt sich eine Vielzahl von Netzen und Verfahren, die teilweise gut beschrieben oder sogar praktisch implementiert sind, teilweise aber nur kurz umschrieben werden und daher nicht praktisch verwendet werden können. Ein sinnvolles Arbeiten mit neuronalen Netzen ohne die vorhergehende Erarbeitung der Grundlagen ist daher kaum möglich. Es ist zwar Literatur vorhanden, diese ist jedoch beispielsweise aus 2008 und enthält damit viele moderne Verfahren nicht[19], oder behandeln teilweise die Theorie nur sehr kurz und konzentrieren sich dann auf die Umsetzung in einer Programmiersprache, ohne weitere Erklärungen zu liefern[20]. Aus diesem Grund stellt folgender Abschnitt wissenschaftlich betrachtet einen wertvollen Beitrag für weitere Forschung dar.

2.1 Entstehung und Entwicklung

Die Idee von Neuronen wurde erstmals von McCulloch und Pitts[21] 1943 aufgegriffen, die sich mit der Vernetzung von elementaren Einheiten, ähnlich der Vernetzung von Neuronen, zum Berechnen jeder arithmetischen und logischen Funktion beschäftigten. Erst 1947 erwähnten sie aber die mögliche Verwendung für Erkennung von Mustern, wie von Weidtmann und Evers[22] beschrieben wird. Weiterentwickelt wurde das Konzept durch Rosenblatt[23], der dem Netz noch zusätzlich Gewichtungen hinzufügte und vereinfachte neuronale Netze Namens Perceptrons entwickelte. Erste praktische Anwendung fanden die Perceptrons nach weiterer Forschung durch das ADALINE-Modell von Widrow und Hoff[24], das zur Echo-Filterung bei Analog-Telefonie eingesetzt wurde[25]. Ende der 1960er Jahre kam die Forschung in dem Bereich fast vollständig zu erliegen, da Minsky und Papert[26] 1969 mit einer mathematischen Analyse eines damals zweischichtigen Perceptrons bewiesen, dass sich damit wichtige Probleme wie XOR Operationen und nicht linear separierbare Probleme nicht lösen lassen. Durch verschiedene Forschungen und Entdeckungen, wie Backpropagation, Neocognitronen und das Hopfield-Netz wurden zwischen 1972 und 1985 wieder wichtige Entwicklungen gemacht. Diese ermöglichten schlussendlich, dass Rumelhart et al.[27] 1986 mithilfe von mehrschichtigen Perceptrons und Learning by Error Propagation (Backpropagation) einige der ursprünglichen Probleme lösen konnten und neuronale Netze wieder mehr Aufmerksamkeit bekamen. Mittlerweile sind neuronale Netze soweit ausgereift, dass sie in vielen Fällen klassische Algorithmen und Vorgehensweisen oder sogar die menschliche Performance übertreffen[28]. So ist bei der heutigen Rechenleistung beispielsweise die Signalverarbeitung in einem neuronalen Netz weitaus schneller, als im menschlichen Gehirn, denn verglichen mit der hohen Taktrate von heutigen Prozessoren hat das Gehirn mit circa 100 Mal pro Sekunde eine sehr geringe Aktivierungsfrequenz[20].

2.2 Aufbau

Ein neuronales Netz besteht aus einer Vielzahl von Neuronen, die über Verknüpfungen miteinander verbunden sind. Es gibt viele verschiedene Topologien, in dieser Arbeit werden für die Erklärung der Netze allerdings mehrschichtige Feed-Forward-Netze verwendet, wie in Abbildung 1 zu sehen. Bei dieser Topologie sind Neuronen immer nur mit Neuronen aus den höheren Ebenen verbunden, also zeigen alle Verbindungen in eine Richtung, wodurch es den Namen „Feed-forward“ erhalten hat. Dieses Netz besitzt außerdem sogenannte Hidden Layers, die weder für Input noch für Output sichtbar sind und die Abstraktion des Netzes erhöhen. Das Netz in der Abbildung 1 hat 68 Verbindungen und 21 Neuronen und besitzt zwei Hidden Layers. Es gibt 6 Input Neuronen und 4 Output Neuronen. Dieses Netz wird beispielsweise zur Klassifikation genutzt, weswegen jedem Output Neuron eine Klasse zugewiesen wird, für deren Erkennung es verwendet werden soll. Das abgebildete Netz kann also zwischen Katze, Hund, Schaf und Wal unterscheiden. Die Anzahl der Input Neuronen eines Netzes hängt von den Messpunkten ab, will man beispielsweise eine Zahl auf einem Bild von 30x30 Pixel

erkennen, werden 900 Input Neuronen benötigt. Die Anzahl der Output Neuronen hängt von der Anzahl an möglichen Optionen ab, bei der Erkennung von Zahlen daher 10 Neuronen für die Ziffern 0 bis 9. Die Anzahl der Hidden Layers und der Neuronen des jeweiligen Hidden Layers können prinzipiell frei gewählt werden, haben aber Einfluss auf die Leistung des Netzes. Es existieren Optimierungsverfahren um für ein Anwendungsgebiet die optimale Anzahl an Hidden Layers und deren Neuronen festzustellen, beispielsweise genetische und evolutionäre Algorithmen[19]. Einfachere Verfahren sind Growing und Pruning. Bei Growing wird ein zu kleines Netz gebaut und trainiert. Das Netz wird nach jedem Training ein wenig vergrößert und erneut trainiert und die Kosten verglichen. Sobald sich keine positiven Auswirkungen auf die Kosten mehr zeigen, sollte das Wachstum gestoppt werden. Pruning ist der umgekehrte Weg, von einem zu großen Netz ausgehend. Manche neuronalen Netze wie Pattern Associators besitzen keine Hidden Layers, sondern nur Input und Output Layer. Neuronen werden oft auch als Units bezeichnet, analog dazu werden beispielsweise die Neuronen des Hidden Layers auch Hidden Units genannt.

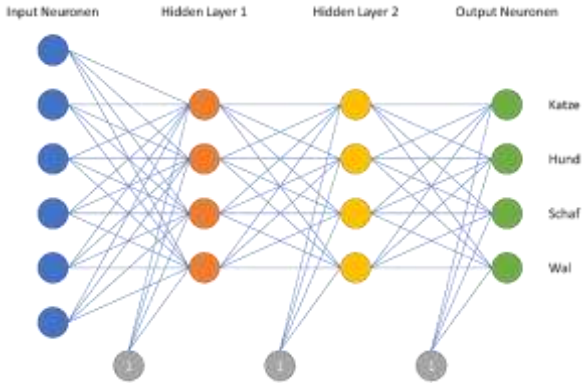

Abbildung 1: Ein schematisches neuronales Netz mit vier Schichten zur Klassifikation in vier Kategorien

Jede Verbindung besitzt eine Gewichtung, die entweder positiv, negativ oder null sein kann. Je nach Netz können Gewichtungen variabel sein und sich während des Trainings ändern oder konstant sein und einen festgelegten Wert haben. Eine positive Gewichtung bedeutet, dass ein Neuron auf das verbundene Neuron einen erregenden Einfluss hat, eine negative Gewichtung zeigt einen hemmenden Einfluss an, eine Gewichtung von null bedeutet, dass die Neuronen im Moment nicht in Beziehung zu einander stehen und sich nicht beeinflussen. Je nach Propagierungsfunktion werden Gewichtung und Signal unterschiedlich verknüpft, um das tatsächliche Signal an ein Neuron zu berechnen. In dieser Arbeit wird bis auf weiteres als Propagierungsfunktion die Linearkombination[19] verwendet. Bei dieser Funktion wird jedes Signal zu einem Neuron mit der entsprechenden Gewichtung multipliziert und anschließend alle Produkte aufsummiert, dies ergibt dann den Netzinput oder Netinput. So eine Funktion wird auch Propagierungsfunktion genannt.

$$Netzinput = \sum a_j * w_{ij}$$

- a_j ist das Ausgangssignal der sendenden Neuronen
- w_{ij} ist die Gewichtung zwischen den sendenden Neuronen und dem empfangenden Neuron

Ein trainiertes neuronales Netz kann beispielsweise wie in Abbildung 2 aussehen. Blaue Linien stellen eine positive Gewichtung dar und orange Linien eine negative Gewichtung, die dicke der Linien soll dabei die Abweichung von 0 darstellen. Die Input Neuronen haben den Wert 0, da sie ihren Wert je nach Input annehmen. Der Wert der Output Neuronen bildet auch das Ergebnis des Netzes, das Ergebnis ist der Typ des Neurons mit dem höchsten Wert. Wenn beispielsweise für einen gewissen Input am Ende das Output-Neuron mit dem Typen Wal den höchsten Wert aufweist, so ist „Wal“ das Ergebnis des Netzes. Neben der Verwendung für Klassifikation können Netze auch für Regression eingesetzt werden. Der Aufbau des Netzes ist aber ident oder zumindest sehr ähnlich.

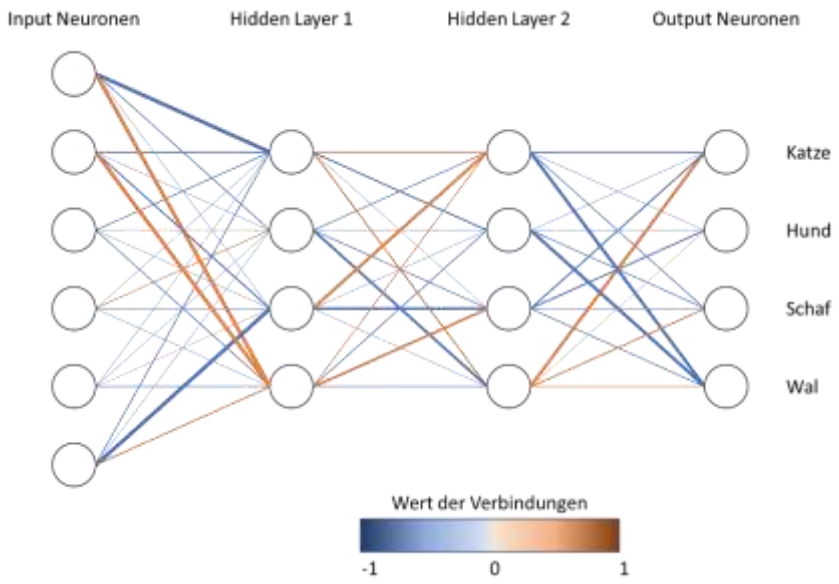

Abbildung 2: Ein schematisches trainiertes neuronales Netz zur Klassifikation in vier Kategorien

2.2.1 Aktivierungsfunktion

Um festlegen zu können, wann ein Neuron aktiviert wird, oder wie oft genannt „feuert“, gibt es verschiedene Aktivierungsfunktionen, die meistens für verschiedene Anwendungsfälle und für verschiedene Netze geeignet sind. Die zwei wichtigsten Eigenschaften von Aktivierungsfunktionen sind die Begrenzung des Aktivitätslevels und die Differenzierbarkeit. Neben den hier beschriebenen Funktionen gibt es noch viele Weitere, diese Auflistung sollte allerdings die gängigsten Funktionen umfassen. Der Netzinput eines Neurons wird als Eingabewert für die Aktivierungsfunktion verwendet, das Ergebnis ist der Output eines Neurons. Die Formel um den Wert eines Neurons zu berechnen besteht daher aus dem Ergebnis der Aktivierungsfunktion, die als Eingangsparameter die Signale der zu dem Neuron verbundenen Neuronen, deren Gewichtung, sowie die Bias des Neurons hat. Unter Verwendung der weiter oben beschriebenen Propagierungsfunktion sieht daher die Formel für die Berechnung des Netinputs von Neuron $a_0^{(1)}$ folgendermaßen aus:

$$a_0^{(1)} = AF(w_{0,0} * a_0^{(0)} + w_{0,1} * a_1^{(0)} + \cdots + w_{0,n} * a_n^{(0)} + b_0)$$

Formel für Berechnung des Signals $a_0^{(1)}$

- AF ist die Aktivierungsfunktion
- $a_0^{(0)}$ ist das erste Neuron der ersten Ebene (Input Layer)
- $a_0^{(1)}$ ist das erste Neuron der zweiten Ebene
- $w_{0,0}$ ist die erste Gewichtung zum ersten Neuron aus der nächsten Ebene
- $w_{0,1}$ ist die zweite Gewichtung zum ersten Neuron aus der nächsten Ebene
- $w_{1,0}$ ist die erste Gewichtung zum zweiten Neuron aus der nächsten Ebene
- b_0 ist die Bias des Neurons, für welches das Signal berechnet wird, also hier $a_0^{(1)}$

In Abbildung 3 sind alle relevanten Elemente für die Berechnung des Signals eingezeichnet, um sich die Zusammensetzung der Formel und die Berechnung besser vorstellen zu können.

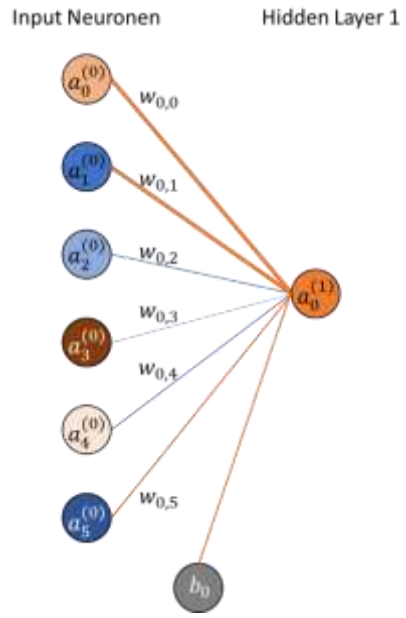

Abbildung 3: Grafische Darstellung der notwendigen Elemente um das Signal für Neuron $a_0^{(1)}$ zu berechnen

Die Funktion kann mithilfe der Summenformel verallgemeinert werden:

$$a_j^i = AF(\sum_k (w_{jk}^i * a_k^{i-1}) + b_j^i)$$

- AF ist die Aktivierungsfunktion
- w_{jk}^i ist die Gewichtung des *k*-ten Neuron in der *(i-1)*-ten Schicht zum j-ten Neuron in der i-ten Schicht
- b_j^i ist die Bias des j-ten Neuron in der i-ten Schicht
- a_j^i ist der Aktivierungswert des j-ten Neuron in der i-ten Schicht

2.2.1.1 Step Funktion

Die Step Funktion ist die einfachste Aktivierungsfunktion, auch vom Rechenaufwand gesehen. Das Ausgangssignal ist 1, wenn das Eingangssignal größer als der festgelegte Schwellwert ist, ansonsten ist das Ausgangssignal 0. Diese Funktion eignet sich aber nur für binäre Klassifikation, da eine Unterscheidung bei mehreren aktivierten Neuronen nicht mehr möglich ist. Weiter ist es mit dieser Funktion nicht möglich, ein Gradientenverfahren als Optimierungsalgorithmus zu nehmen, da die Step Funktion nicht differenzierbar ist.

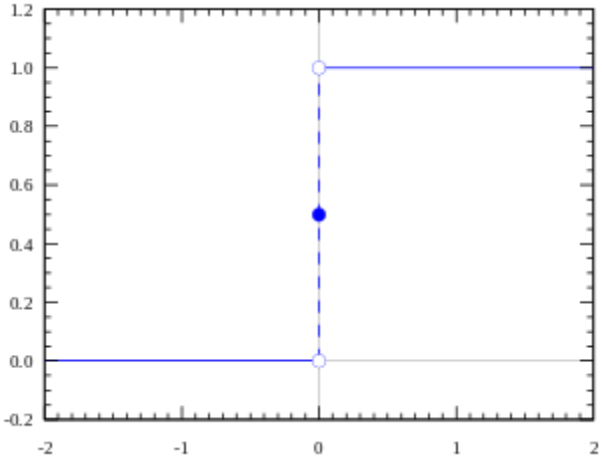

Abbildung 4: Die Step-Funktion

2.2.1.2 Lineare Funktion

Bei dieser Funktion ist das Ergebnis proportional zum Eingangssignal, es ist möglich nicht-binäre Entscheidungen zu treffen, da ein Unterschied zwischen einem oder mehreren aktiven Neuronen festgestellt werden kann. Allerdings kann bei Verwendung dieser Aktivierungsfunktion nicht mit Gradientenverfahren trainiert werden, da dort die erste Ableitung benötigt wird und die erste Ableitung von $f(x) = c * x$ nach x ergibt die Konstante c. Ein weiterer Nachteil ist, dass kein Vorteil durch mehrere Schichten entsteht, da ohnehin alle Signale und alle Kombinationen von Signalen linear sind. Die lineare Funktion ist nach oben und nach unten unbeschränkt, daher wäre die Verwendung bei Recurrent Neural Networks (RNN) problematisch, da sich das Signal über mehrere Zyklen endlos steigert[19].

$$f(x) = c * x$$

Formel für lineare Funktion

- c ist die Steigung der Funktion

2.2.1.3 Sigmoid Funktion

Die Sigmoid Funktion erlaubt binäre und nicht-binäre Entscheidungen, es lassen sich mehrere Schichten kombinieren, da auch Kombinationen davon nicht linear sind. Meist wird eine Sigmoid Funktion jedoch nur für binäre lineare Regression eingesetzt, da beispielsweise im Gegensatz zur Softmax Funktion auch alle Neuronen der Output Schicht den Wert 1 haben könnten. Im Wesentlichen kann die Sigmoid Funktion in die logistische Funktion und die Tangens Hyperbolicus Funktion unterteilt werden, beide Funktionen sind nach oben und nach unten hin begrenzt. Der wesentliche Unterschied zwischen den beiden Funktionen ist, dass der Wertebereich der logistischen Funktion von 0 bis +1 reicht, während der Wertebereich der Tangens Hyperbolicus Funktion von -1 bis +1 reicht. Die Begrenzung sorgt dafür, dass die Signale über alle Schichten hinweg in demselben Bereich bleiben, und sich nicht drastisch in jeder Schicht erhöhen. Dies verhindert zum Beispiel bei Recurrent Neural Networks, dass sich das Signal über mehrere Zyklen so hoch steigern kann, dass kein anderes Signal mehr durchkommt (Gradient Explosion). Durch die Form der Kurve, die sich ab einem gewissen Wert nicht mehr ändert, lässt sich das Netz aber auf Dauer nicht mehr gut trainieren, da eine Änderung des x-Wertes zu immer schwächeren oder zu keiner Änderung des y-Wertes führt (Vanishing Gradient Problem).

Abbildung 5: Graph der logistischen Funktion

Die logistische Funktion wird in der einfachsten Form durch die folgende Funktion berechnet:

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

logistische Funktion

2.2.1.4 Rectified Linear Unit

Die Rectified Linear Unit (ReLU), die in Abbildung 6 zu sehen ist, erlaubt es, obwohl ein Teil davon linear ist, mehrere Schichten zu bilden, da alle Kombinationen nicht-linear sind. Diese Funktion wird auch lineare Funktion mit Schwelle genannt. Eine ReLU ist eine Funktion die für jeden x-Wert ≤ 0 nur 0 zurückgibt, für jeden x-Wert > 0 den Wert $x * c$ zurückgibt, wobei c konstant ist und fast beliebig gewählt werden kann. Diese Funktion ist ohne viel rechnerischen Aufwand umzusetzen und lässt Neuronen mit negativen Werten deaktiviert, was den Rechenaufwand stark reduziert und es beliebt für große Netze macht. Unter anderem wird dadurch wird meist die Trainingszeit beim Einsatz von ReLUs kürzer[6]. Wenn der Gradient im Zuge des Trainings allerdings null

erreicht, reagiert die ReLU auf keine Änderungen mehr und gibt kein Signal mehr zurück, dies wird auch als „Dying ReLU“-Problem bezeichnet und kann für den Ausfall von weiten Teilen des Netzwerkes sorgen. Für dieses Problem gibt es verschiedene Lösungsversuche, wie zum Beispiel die Funktion auf $y = -0{,}01x$ für x<0 zu ändern, auch Leaky ReLU genannt. Der Signal-Wert dieser Funktion ist nur nach unten hin begrenzt, aber nach oben offen. Durch den Vorteil für große Netze und die leichte Berechenbarkeit werden ReLUs sehr oft in Hidden Layers eingesetzt.

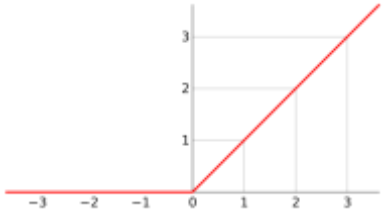

Abbildung 6: Eine ReLU [29]

2.2.1.5 Softmax Funktion

Die Softmax Funktion ähnelt von der Form der Sigmoid Funktion. Diese Funktion wird oft im Output Layer für Classifier eingesetzt, beispielsweise von [13], da die Werte im Bereich 0 bis 1 liegen und alle Werte einer Schicht immer eine Summe von 1 ergeben. Dadurch eignet sich die Funktion gut für nicht-binäre Classifier beziehungsweise Wahrscheinlichkeitsverteilungen.

2.2.2 Bias-Units

Da die korrekte Übersetzung von Bias aus dem Englischen zu Tendenz nicht sehr aussagekräftig ist, wird in dieser Arbeit bis auf weiteres das Englische Wort Bias verwendet. Viele Neuronen besitzen ein „Bias-Neuron“ oder kurz Bias, auf Deutsch Tendenz, die ein Neuron unabhängig vom Input dauerhaft hemmt, dauerhaft verstärkt oder sich gar nicht auswirkt. Das Bias-Neuron strahlt dabei immer den Ausgangswert 1 aus, also das stärkst mögliche Signal. Über die Gewichtung vom Bias Neuron zu dem Neuron selbst wird dann festgelegt, welchen Einfluss es auf das jeweilige Neuron nimmt. Der Ausgangswert des Bias Neurons bleibt unverändert, aber die Gewichtung kann sich, wie jede andere Gewichtung, im Zuge des Trainings verändern. In Abbildung 1 sind die Bias Neuronen in grau dargestellt, meist werden sie aber zur besseren Übersicht in Abbildungen weggelassen oder in schematischen Abbildungen der Einfluss der Bias Neuronen durch Farben der Neuronen selbst dargestellt. Auch Neuronen der Input-Schicht können eine Bias besitzen, dies ist allerdings eher unüblich[19]. Im Gegensatz zu Schwellen in der Aktivierungsfunktion ist die Schwelle durch den Wert der Bias Neuronen variabel und kann damit während des Trainings geändert werden.

2.2.3 Hyperparameter

Hyperparameter sind Parameter, die das Netz nicht von alleine lernen soll oder kann und die daher noch vor Beginn des Trainings gesetzt werden[30]. Durch diese Parameter wird das Netz als Ganzes parametrisiert und beschrieben, sie beinhalten beispielsweise Lernrate, Kostenfunktion, Optimierungsverfahren, Aktivierungsfunktion. Auch der grundlegende Aufbau des Netzes, wie beispielsweise die Anzahl an Hidden Layers und die Anzahl deren Neuronen zählen als Hyperparameter. Da diese Einstellungen die Funktionsweise des Netztes elementar beeinflussen, wird mithilfe von Hyperparameter Optimierung versucht, diese Parameter möglichst optimal zu wählen. Für die meisten Hyperparameter gibt es aber noch keine Verfahren um sie systematisch zu verbessern, außerdem würden sie damit vom Hyperparameter zum Parameter werden. Daher ist es meist üblich, sich für verschiedene Hyperparameter aufgrund von Erfahrung möglichst sinnvolle Werte zu überlegen, diese nacheinander auf das Netz anzuwenden und die resultierende Performance zu vergleichen. Dadurch kann das Training aber sehr aufwendig werden, da ein Netz vielleicht nicht einmal sondern zehn Mal oder öfter mit verschiedenen Hyperparametern trainiert werden muss. Dies ist notwendig, da es trotz „üblicher“

sinnvoller Werte für Hyperparameter oft vorkommt, dass für einen spezifischen Anwendungsfall andere Werte bessere Ergebnisse erzielen.

2.2.4 Initialisierung der Gewichtungen

Wenn Gewichtungen nicht initialisiert werden, sondern zu Beginn des Trainings alle den Wert null haben, würden alle Neuronen einer Schicht dieselben Werte produzieren und beispielsweise im Zuge von Backpropagation alle dieselben Updates erhalten. Das Netz könnte sich also nichts merken und sich nicht verbessern. Abgesehen von diesem Fall können zu kleine Gewichtungen dazu führen, dass das Signal nicht durch alle Schichten eines Netzes kommt, während zu große Gewichtungen dazu führen können, dass ein sehr großes und einheitliches Signal durchkommt, das zu sehr einheitlichen Änderungen im ganzen Netz führt. Die Initialisierung der Gewichtungen kann dadurch im Extremfall darüber entscheiden, ob ein Netz überhaupt konvergieren kann oder nicht, wie beispielsweise in [31] beschrieben wird. Eine kurze Übersicht über einige Initialisierungs-Verfahren befindet sich unter [32].

2.2.4.1 Xavier Initialization

Xavier Initialization oder Glorot Initialization ist eine Initialisierung von Glorot und Bengio[33], bei der für jede Gewichtung ein Wert aus einer Verteilung mit Mittelwert null und einer speziellen Varianz ausgewählt wird. Dadurch soll eine gute Verteilung rund um Null gefunden werden, welche keine zu großen oder zu kleinen Gewichtungen erzeugt. Die Xavier Initialization ist sehr weit verbreitet und oft die standardmäßige Variante.
Die Varianz ergibt sich dabei durch die folgende Formel:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

- W ist die Verteilung der Gewichtungen für ein Neuron
- n_{in} und n_{out} ist die Anzahl der Input Neuronen, beziehungsweise die Anzahl der Output Neuronen

Oft wird die Formel auch in einer abgeänderten Version verwendet, die sich ausschließlich auf die Input Neuronen beschränkt:

$$Var(W) = \frac{1}{n_{in}}$$

2.2.4.2 RELU/He Initialization

Diese Initialisierung ist von He et al. [31] und wird ReLU Initialization oder He Initialization genannt. Sie eignet sich besonders gut für sehr tiefe Netzwerke und Netzwerke mit ReLUs und unterscheidet sich dadurch von der Xavier Initialization, die bei tiefen Netzwerken teilweise nicht konvergieren kann. Die Formel ist stark angelehnt an die Formel der Xavier Initialization

$$Var(W) = \frac{2}{n_{in}}$$

2.3 Arten von Netzen

Es existiert eine Vielzahl von unterschiedlichen neuronalen Netzen, weswegen hier nur ein paar verschiedene Netze als Beispiel für den unterschiedlichen Aufbau erklärt werden. Weitere besondere Arten von Netzen sind das Kohonennetz oder das Constraint Satisfaction Netz. Da wie bereits erwähnt keine einheitliche Definition für neuronale Netze existiert, existiert auch keine einheitliche Definition der unterschiedlichen Typen. Dieses Kapitel ist daher nicht als absolut zu betrachten, in unterschiedlicher Literatur besitzen manche Typen eine unterschiedliche Anzahl von Schichten, werden in andere Kategorien eingeteilt oder zu einem Typ zusammengefasst.

2.3.1 Feed-forward Neural Networks

Feed-forward Neural Networks, auf Deutsch vorwärts gerichtete neuronale Netze, können meist aus drei oder mehreren Schichten bestehen. In dieser Art von Netz sind die Neuronen einer Schicht immer nur mit Neuronen der nächsten Schicht verbunden, niemals jedoch mit sich selbst, Neuronen derselben Schicht oder der vorherigen Schicht. Es kann jedoch sogenannte Short-Cut Verbindungen geben, die eine oder mehrere Hidden Layers überspringen. Die Netze sind eine der einfachsten Formen und lassen sich leicht konstruieren und trainieren, haben dafür nur einen eingeschränkten Anwendungsbereich. Abgesehen davon, dass der Begriff Feed-forward Neural Network oft als Überbegriff für Netze mit den oben erwähnten Eigenschaften dient, bezeichnet dieser Begriff auch direkt ein einfaches Netz. Oft wird auch der Begriff „fully connected“ verwendet um zwei Schichten oder ein Netz zu bezeichnen, wo alle Neuronen einer Schicht mit allen Neuronen der nächsten Schicht verbunden sind, ohne dass Verbindungen fehlen und ohne Short-Cut Verbindungen.

2.3.1.1 Autoencoder

Autoencoder haben meist flache Netze, oft nicht mehr als ein Input Layer, ein Hidden Layer und ein Output Layer. Sie werden dafür eingesetzt, Input zu komprimieren, indem sie ihn möglichst gut rekonstruieren. Da für den Hidden Layer meist weniger Neuronen gewählt werden als im Input Layer muss das Netz entscheiden, welche Features einen Input dieser Art besonders gut definieren. Daher werden diese Netze auch für Feature Extraction und Reduzierung der Dimensionalität angewendet, indem das Ergebnis des kleinsten Hidden Layers verwendet wird, sie können also wesentliche Merkmale von Daten identifizieren. Autoencoder können mit unsupervised Learning trainiert werden, da für die Kostenfunktion lediglich der Input mit dem Output verglichen werden muss, das Netz darauf ab den Input möglichst genau rekonstruieren.

2.3.1.2 Restricted Boltzmann Machine

Restricted Boltzmann Machines(RBM) sind flache Netze mit zwei Ebenen, genannt Visual Layer und Hidden Layer. Die Anwendung für eine RBM liegt darin einen Input wieder rekonstruieren beziehungsweise klassifizieren zu können. Beim Training wird im ersten Schritt Input in das Netz in den Visual Layer gegeben und der Endwert der Neuronen im Hidden Layer berechnet. Im Zweiten Schritt werden die Werte der Neuronen im Hidden Layer als Ausgangswert genommen und über die Verbindungen zurück in den Visual Layer propagiert, wo sie zu denselben Werten wie die ursprünglichen Inputwerte führen sollten.

2.3.1.2.1 Deep Belief Network

Ein Deep Belief Network (DBN) besteht aus mehreren RBMs in Schichten hintereinander. So ist der Output Layer des ersten RBMs der Visual Layer des zweiten, und so weiter. Außergewöhnlich sind DBNs aufgrund des Trainingsansatzes, sonst gleichen sie prinzipiell einem fully connected Feed-forward Neural Network. Das Training besteht aus einem unsupervised Pre-Training und aus einem supervised Fine Tuning, für das nur ein kleiner beschrifteter Datensatz notwendig ist. Im Pre-Training wird jedes RBM nacheinander und einzeln trainiert, das heißt, immer zwei Layers werden darauf trainiert, den vollständigen Input möglichst gut wiedergeben zu können. Wie in Abbildung 7 beschrieben, wird also zuerst Abschnitt a vollständig trainiert, dann Abschnitt b und dann c. Dies hat damit zu tun, dass bewiesenermaßen diese Art der Anpassung der Gewichtungen durch unsupervised Pre-Training das Netz optimal für supervised Training vorbereitet, da sich die Gewichtungen dann bereits nahe einem Minimum befinden[34].

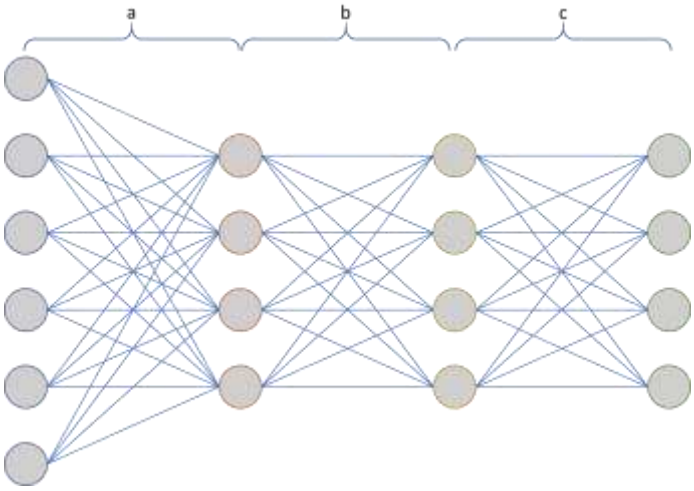

Abbildung 7: Die Trainingsabschnitte eines DBNs

Dies unterscheidet es grundlegend von anderen Netzen, wo alle Layers gleichzeitig trainiert werden, das Wissen über das ganze Netz verstreut ist und jeder Layer unterschiedliche Dinge lernt. Durch diesen Ansatz funktionieren DBNs besser als ein RBM, da mehrere Layers die Möglichkeit haben, dem Input möglichst nahe zu kommen, vorstellbar wie eine Kameralinse die langsam auf ein Bild scharfstellt. Danach wird im Fine-Tuning supervised Learning in klassischer Weise mit Backpropagation durchgeführt. Da das Netz aber bereits sehr gut auf die Merkmale einer Art von Daten trainiert ist, haben die Gewichtungen bereits eine gute Verteilung und das supervised Training funktioniert auch über viele Layers gut, wodurch das Netzwerk auch weniger zu Overfitting neigt[34]. Durch das supervised Fine Tuning kann ein DBN für Classification eingesetzt werden, aufgrund des Pre-Trainings wird nur ein sehr kleiner beschrifteter Datensatz benötigt, was Aufwand und Fehleranfälligkeit reduziert. Neben diesen Vorteilen ist ein DBN im Vergleich zu anderen Netzen auch sehr genau und kann in kurzer Zeit trainiert werden. Erfunden wurden DBNs von Geoffrey Hinton, der auch schon an der RBM beteiligt war. Leider existiert direkt über DBNs keine Forschungsarbeit, die älteste Quelle zu DBNs stellt ein Artikel von Hinton selbst in Scholarpedia dar[35].

2.3.1.3 Pattern Associator

Der Pattern Associator ist ein Netz bestehend aus einem Input Layer und einem Output Layer. Alle Neuronen der Input-Schicht sind mit allen Neuronen der Output Schicht verbunden. Dieses Netz eignet sich beispielsweise zur Erkennung von vorher gelernten Mustern und Formen, wie Gesichtern. Da es keine Hidden Layers besitzt kann es nicht mittels Backpropagation trainiert werden, sondern muss mithilfe der Hebb-Regel oder Delta-Regel[19] lernen.

2.3.1.4 Recursive Neural Tensor Networks

Recursive Neural Tensor Networks (RNTN) eignen sich gut um Semantik zu erfassen. So beachten sie beispielsweise bei der Analyse eines Satzes nicht nur die enthaltenen Wörter, sondern auch die Reihenfolge. Der Aufbau entspricht dabei einem binären Baum, wobei ein Knoten aus einer Menge von Neuronen besteht. Die Daten fließen von den Blättern zur Wurzel, wo ein Wert herauskommt [36]. RNTNs können mittels Backpropagation trainiert werden. Neben der Textverarbeitung können sie auch in der Bild-Analyse zur Erkennung verschiedener Bereiche und Formen eingesetzt werden, auch genannt Image Parsing.

2.3.1.5 Convolutional Neural Networks

Convolutional Neural Networks (CNN) besitzen mehrere Verarbeitungsschichten und eignen sich vorzugsweise zur Erkennung von Bildern beziehungsweise Mustern, die sich in einem zweidimensionalen Raum darstellen lassen. Anhand von Abbildung 8 erklärt, ein so genanntes Feature(A) stellt ein Unterscheidungsmerkmal dar und wird mit allen möglichen Stellen eines Inputs(B) verglichen, die Übereinstimmung mit der jeweiligen Stelle

wird auf einer Karte(C) festgehalten. Die drei farbigen Rahmen bei Input(B) stellen drei der 25 Stellen dar, an denen das Feature mit dem Input verglichen wird. In der unten abgebildeten Karte(C) stellt daher die Zahl 1 in der linken oberen Zelle die Übereinstimmung mit dem orangen Rahmen bei Input(B) dar. Wie sich auf der Karte(C) gut erkennen lässt, sieht man entlang der Diagonale von links oben nach rechts unten eine hohe Übereinstimmung, dieses Feature hilft also Diagonalen dieser Art in einem Bild zu finden. Dieser erste Schritt wird Filtering genannt und wird mit allen Features und jeweils einer eigenen Karte für den selben Input wiederholt. Die Anzahl der Filter kann frei gewählt werden, je mehr Filter, desto feinere Muster können erkannt werden. Die Schicht, in welcher dieser Vorgang durchgeführt wird, heißt Convolutional Layer, dabei wird aus einem Input je nach Anzahl der Filter ein Stapel von gefilterten Inputs, den Karten.

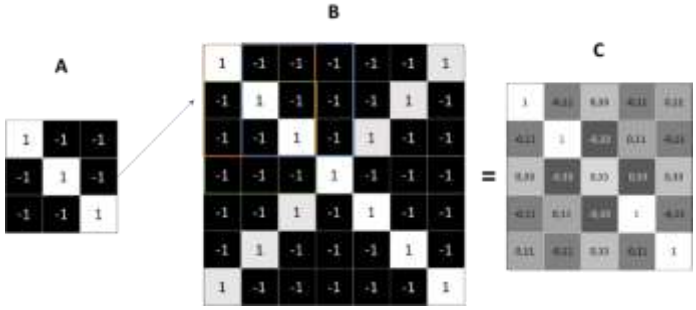

Abbildung 8: Durchführung von Filtering eines Features im Convolutional Layer

Danach durchläuft jede Karte eine ReLU, in der alle negativen Werte auf null gesetzt werden, da sie nicht mehr benötigt werden und dies weitere Berechnungen vereinfacht. Dies findet in dem ReLU Layer statt und wird Normalization genannt. Jede dieser normalisierten Karten wird danach beim Pooling verkleinert. Dabei wird zwar die Genauigkeit reduziert, aber dafür die Flexibilität erhöht, da ein Unterscheidungsmerkmal nicht exakt die selbe Form haben muss. Außerdem wird die Leistung des Netzwerks erhöht, da beispielsweise ein Bild mit 7x7 Pixel auf 4x4 Pixel reduziert werden kann. Dieser Vorgang wird für alle Karten durchgeführt und findet im Pooling Layer statt. Dabei unterscheidet man grundsätzlich zwischen Max-pooling, bei dem der höchste Wert eines Bereiches übernommen wird, oder Average-pooling, bei dem der Durchschnittswert eines Bereichs übernommen wird. Diese drei Schritte können beliebig in unterschiedlicher Reihenfolge wiederholt werden, um die Erkennungsrate des Netzwerkes zu erhöhen. Zuletzt werden die verkleinerten Karten als Input für ein Feed-forward Neural Network verwendet, dabei dient je ein Wert auf einer Karte als Input für ein Neuron. Aus 10 Karten mit 2x2 Feldern ergeben sich daher 40 Input Neuronen. Dieses neuronale Netzwerk trifft dann wie ein herkömmlicher Classifier eine Entscheidung, die Schichten davor dienen quasi lediglich der „Vorverarbeitung“.

Abbildung 9: Die möglichen Schritte eines CNNs

2.3.2 Fast Neural Networks

Fast Neural Networks (FNN) sind Netze, die speziell für Anwendungen geeignet sind, die eine schnelle Durchlaufzeit benötigen. Sie ermöglichen es beispielsweise schnell Gesichter oder Augen (Iris) zu erkennen, da sie im Vergleich zu anderen Netzen weitaus weniger Rechenschritte benötigen[37]. Leider ist über diese Netze nicht viel bekannt, man findet kaum Literatur oder praktische Implementierungen dazu.

2.3.3 Recurrent Neural Networks

In Recurrent Neural Networks (RNNs) haben im Vergleich zu Feed-forward Neural Networks Neuronen auch Verbindungen zu Neuronen aus vorherigen Schichten, zu sich selber oder zu Neuronen die nur zur Informationsspeicherung gedacht sind, sogenannten Hidden States, siehe Abbildung 10. Damit werden Rückkoppelungen und Informations-Rückfluss erlaubt, durch die RNNs Informationen sequentiell über mehrere Zyklen verarbeiten können. Daher können RNNs auch als eine Sequenz aus zusammenhängenden Netzen betrachtet werden, die in einem fixen zeitlichen Ablauf stehen, dieser Vorgang wird auch „aufrollen“ eines Netzes genannt.

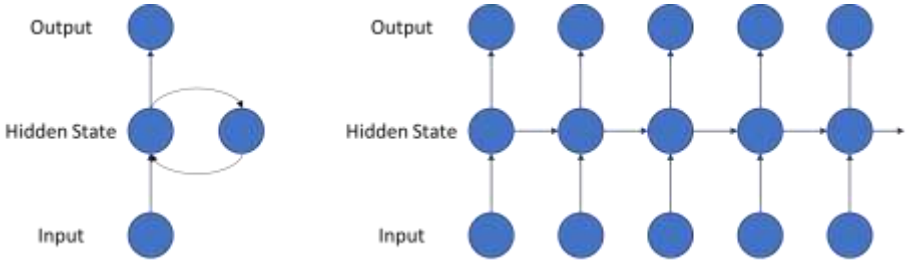

Abbildung 10: Ein RNN(links) aufgerollt über mehrere Zeitschritte(rechts)

Bei steigender Anzahl an Zyklen zwischen zwei zu verknüpfenden Inputs sinkt aber die Wahrscheinlichkeit drastisch, dass das Netz eine richtige Entscheidung trifft, da der „Einfluss“ der ersten Information jeden Zyklus sinkt und von anderen Informationen überlagert wird. So lassen sich zwar gut Buchstaben oder Wörter damit verarbeiten, aber das Netz wird beispielsweise am Ende eines Satzes nur noch schwer feststellen können, mit welchem Buchstaben der Satz begonnen wurde, da dazwischen eine Vielzahl an anderen Inputs passiert sind. Bei Einsatz einer linearen Aktivierungsfunktion kann es passieren, dass der Gradient über mehrere Zyklen extrem stark ansteigt, auch genannt „Gradient Explosion Problem“. Durch die Verzweigungen des Netzes ist es nicht mehr möglich, das herkömmliche Backpropagation Verfahren zur Verbesserung einzusetzen, da dieses Verfahren Rückkoppelungen und Schleifen nicht entsprechend einbeziehen kann. Stattdessen wurden die zwei Verfahren Backpropagation Through Time (BPTT) und Real-Time Recurrent Learning(RTRL) entwickelt. Beispiele für RNNs sind das Simple Recurrent Network (SRN), das Jordan-Netz, Elmann-Netz, Auto Associator und das Attraktoren Netz[19]. RNNs können aufgrund ihrer Komplexität Turing-vollständig sein[38]. Durch ihre Komplexität neigen RNNs aber zu Überanpassung und instabilem Verhalten.

Abbildung 11: Beispiel für ein SRN

2.3.3.1 *Long- Short-Term Memory*

Eine wichtige Art von RNNs sind Long Short-Term Memory-Netze (LSTM), die durch ihren Aufbau Informationen über viele Input-Zyklen hinweg speichern und verwenden können. Sie wurden von Hochreiter und Schmiedhuber[39] 1997 entwickelt und beinhalten neben einem neuartigen Netz auch ein neuartiges Verfahren für die Verbesserung, nämlich Constant Error Carrousel (CEC) anstatt des komplexeren BPTT. Dadurch erhalten sie eine sehr hohe Dynamik und Flexibilität und zeichnen sich darüber hinaus durch eine hohe Lernfähigkeit und Genauigkeit trotz vieler Schritte aus. Sie können durch diesen Speicher Input in beliebiger Reihenfolge verarbeiten und eignen sich daher gut für Daten mit einer zeitlichen Dimension, wie Log Analyse, Sprachanalyse oder Wort-Vorhersage auf Smartphones. Hochreiter und Schmiedhuber zeigen in ihrem Paper unter anderem in fünf Experimenten, dass ein LSTM Netz auf Input je nach zeitlicher Reihenfolge anders reagieren kann und sich wesentlichen Input auch über mehrere hundert Schritte merken kann. Die wesentlichen Bestandteile eines LSTMs sind dabei die Memory Cells, die in Memory Blöcke zusammengefasst werden können, und die zugehörigen Gates. Jeder Memory Block hat ein Input Gate, das dafür zuständig ist den Signalfluss in den Memory zu regeln, also sicherzustellen, dass nur wesentliche Informationen im Memory abgespeichert werden. Außerdem ein Output Gate, welches sicherstellt, dass der Inhalt des Speichers nur ausgegeben wird, wenn er gerade relevant ist. Ein LSTM kann wie in Abbildung 12 gezeigt beispielsweise aus vier Input und vier Output Neuronen, zwei Memory Blöcken zu je zwei Memory Cells und den zugehörigen Gates bestehen. Die Input Neuronen sind mit allen Gates und Memory Cells verbunden, diese sind jeweils unter sich vollständig verbunden, auch mit sich selbst. Mit dem Output Layer sind allerdings in diesem Fall nur die Memory Cells verbunden. Die Memory Cells kann man sich als Neuronen vorstellen, die mit sich selbst verbunden sind und dadurch Informationen abspeichern können.

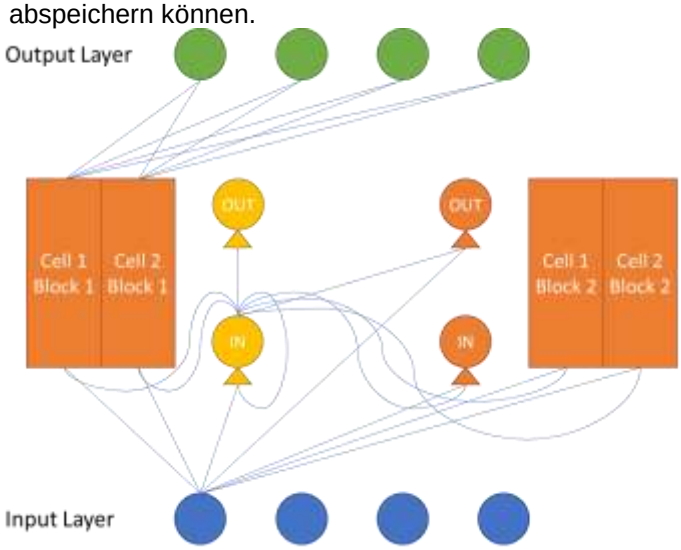

Abbildung 12: Beispiel für ein LSTM. Zur besseren Übersicht sind nicht alle Verbindungen eingezeichnet. Die Abbildung ist [39] nachempfunden.

Die vereinfacht als Input und Output Gates dargestellten Elemente sind eigene neuronale Netze, damit die Gates „lernen“ können, wann sie ein Signal durchlassen und wann sie es blockieren. LSTMs eignen sich, im Gegensatz zu den meisten anderen Netzen, auch zur Zeitreihen Vorhersage, was durch die zeitliche Dimension erschwert ist.

2.3.4 Adaptive Resonanztheorie

Die adaptive Resonanztheorie (ART) will das Dilemma zwischen Plastizität und Stabilität lösen und wurde von Stephen Grossberg und Gail Carpenter erfunden. Dieses Dilemma entsteht bei neuronalen Netzen, weil ein neuronales Netz entweder eine hohe Plastizität hat, wodurch viel Neues gelernt werden kann aber bestehendes Wissen verschwindet da es überlagert wird, oder es hat eine hohe Stabilität, die sicherstellt, dass bestehendes

Wissen erhalten bleibt, aber damit verhindert, dass Neues gelernt werden kann. Aus dieser Theorie ergibt sich ein Modell für ein Netz, das durch Plastizität immer bereit ist Neues zu lernen, aber durch einen Mechanismus sich bei neuem Wissen selbstständig um Neuronen erweitert und dadurch sicherstellt, dass bestehendes Wissen unberührt und damit erhalten bleibt. Diese Theorie wird übrigens auch im Bereich der Neurowissenschaft verwendet[40]. Zur einfacheren Wiedergabe wird in der Literatur für Netze, die auf der adaptiven Resonanztheorie beruhen, ebenfalls die Abkürzung ART oder ART-Netze verwendet, siehe beispielsweise [41]. ART Netze werden grundsätzlich mit unsupervised Learning trainiert. Es existieren mittlerweile eine Vielzahl an ART Varianten, beispielsweise Fuzzy ART[42], Distributed ART[43], ARTMAP[44], Fuzzy ARTMAP[45] und viele weitere. Obwohl das Konzept vielversprechend ist, da es beispielsweise ein großes Problem bei der Klassifizierung von immer größeren Datensätzen löst, wird es leider kaum in der Literatur, Blogs, Vorträgen oder Tutorials behandelt. So stammt die meiste Literatur von einem der beiden oder beiden Erfindern selbst. ART Netze werden anscheinend auch von keinem Framework für neuronale Netze unterstützt. Dabei wäre es beispielsweise gut geeignet für ein Netz, das sich Muster von Malware merken muss, da es ständig mit neu auftauchender Malware erweitert werden kann, ohne bestehende Muster zu überschreiben. Bei anderen Netzen hingegen müsste man von Zeit zu Zeit das Netz mit mehr Neuronen neu aufbauen und grundlegend neu trainieren.

2.3.5 Word2Vec

Word2Vec ist eine Gruppe von neuronalen Netzen die verwendet werden um Word Embeddings zu erstellen, die oftmals zur weiteren Verarbeitung benötigt werden. Word2Vec Netze sind flache Netze und bestehen daher oft nur aus Input und Output Layer, manchmal haben sie auch noch einen zusätzlichen Hidden Layer. Als Input dienen mehrere Wörter, die durch das Verfahren in Vektoren in einem gemeinsamen Vektorraum umgewandelt werden. Diese Vektoren können genutzt werden um ähnliche oder zusammenhängende Wörter zu identifizieren, aber auch um als Input für weitere neuronale Netze zu dienen. Gerade durch letzteres sind sie für neuronale Netze mit Text-Input sehr wichtig, da neuronale Netze Buchstaben und Wörter standardmäßig kaum verarbeiten können, wie in Kapitel 3.1.1 erläutert. Ein effizientes Verfahren namens Skip-Gram wurde unter anderem von [46] präsentiert.

2.4 Training

Nach der Konzeption eines neuronalen Netzes haben alle Neuronen und Verbindungen noch kein Gewicht, sie müssen durch Training erst angepasst werden. Um dieses Training zu erleichtern wird dafür in vielen Fällen für jede Gewichtung und jede Bias-Gewichtung zu Beginn ein zufälliger Wert eingesetzt[47], wobei dafür aus verschiedenen Verfahren gewählt werden kann, wie ein Kapitel 2.2.4 beschrieben. Das Ziel des Trainings ist einfach formuliert, für alle Biases und Gewichtungen den besten Wert zu finden, damit das Netz die Aufgabe optimal lösen kann. Grundsätzlich gibt es drei relevante Ansätze für das Training: supervised Learning, Reinforcement Learning und unsupervised Learning. Zusätzlich existieren verschiedene Lernregeln, um die Verbesserung strukturiert durchzuführen, verschiedene Kostenfunktionen, um die Treffsicherheit des Netzes zu prüfen und verschiedene Optimierungsverfahren, um das Netz über Änderung der Gewichtungen und Biases zu verbessern. Ein besonderes Vorgehen gilt für Netze wie Constraint Satisfaction Netze (CSN), da dort die Verbindungen und deren Gewichtungen im Zuge des Designs festgelegt und nicht mehr geändert werden, man spricht auch von der Direct Design Method[19]. Netze dieser Art überspringen daher die Trainingsphase. Da dieses Anwendungsgebiet dieser Netze aber nicht zum Ziel der Arbeit passt, wird nicht näher darauf eingegangen. Wenn ein Netz auch nach langwierigem Training nicht fähig ist, die Trainingsdaten richtig zu verarbeiten, wird es „Underfitting“ genannt. In diesem Fall müssen Hyperparameter geändert oder grundlegend die Verwendung eines gewissen Netzes oder neuronaler Netze allgemein überdacht werden.

2.4.1 Trainingsarten

2.4.1.1 Supervised Learning

Für diesen Ansatz wird ein großer Datensatz mit Objekten oder Werten benötigt, die erkannt oder verarbeitet werden sollen, denen aber bereits das richtige Ergebnis zugewiesen ist. Am Beispiel von der Ziffern-Erkennung wären das beispielsweise tausende Tupel bestehend aus Bildern von handschriftlichen Ziffern und der darauf dargestellten Zahl als Text.

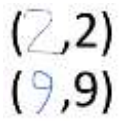

Abbildung 13: Zwei Beispiele für ein beschriftetes (engl. „labeled“) Tupel für supervised Learning

Nun wird nach der Reihe jedes Bild als Input für das Netz verwendet und das Ergebnis des Netzes mit dem verglichen, was auf dem Bild tatsächlich dargestellt wird. Diese Abweichung wird über die sogenannte Kostenfunktion festgestellt und dient dazu, dass sich das Netz je nach Stärke der Abweichung gezielt verbessern kann. Wenn das Netz ein Bild richtig erkennt, sind die Kosten sehr klein, wenn das Netz ein Bild nicht oder nicht gut erkennt, dann sind die Kosten sehr groß. Der Durchschnitt der Kosten für ein ganzes Trainingsset zeigt an, wie gut das Netz bereits trainiert ist beziehungsweise wie erfolgreich es aus einem Input den gewünschten Output erzeugt. Über die entsprechende Lern-Regel werden dann Anpassungen an den Gewichtungen vorgenommen. Dieses Training wird so oft wiederholt, bis die durchschnittlichen Kosten entsprechend gering sind, das neuronale Netz sieht dann wie beispielsweise in Abbildung 2 aus.

2.4.1.1.1 Kostenfunktionen

Um die Kosten eines Netzes zu berechnen, also wie gut es in Bezug auf die Trainingsdaten abschneidet, gibt es verschiedene Funktionen. Kostenfunktionen, oder auch Error- und Loss-Funktionen genannt, sind ausschließlich für supervised Learning relevant. Eine weit verbreitete Kostenfunktion ist die Funktion der quadratischen Kosten, auch bezeichnet als durchschnittlicher quadratischer Fehler, maximale Wahrscheinlichkeit und summierter quadratischer Fehler. Bei dieser Funktion wird im Output Layer für jedes Neuron das gewünschte Ergebnis von dem gegebenen Ergebnis subtrahiert, quadriert und alle Ergebnisse aufsummiert. Damit erhält man den Fehler für einen Trainingsinput als numerische Zahl, wie am Beispiel von Abbildung 14 gezeigt, in der das letzte Neuron das richtige Ergebnis wäre und daher den Wert 1 haben müsste.

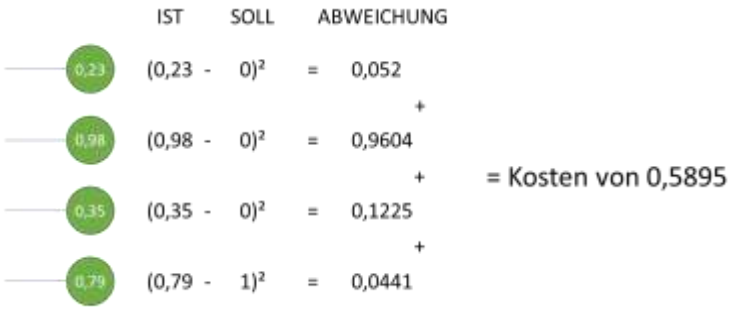

Abbildung 14: Berechnung der Kosten für den Output von 4 Neuronen

Die Formel für die Berechnung der durchschnittlichen quadratischen Kosten (engl. Mean squared error) lautet:

$$E_{Total} = \frac{1}{2}\sum(output - target)^2$$

- *Output* ist der tatsächliche Output vom ersten Neuron im Output Layer

- *Target* ist der gewünschte Output, den das Neuron im Output Layer haben soll, damit der Input richtig erkannt werden würde

Die Funktion der quadratischen Kosten wird sehr oft verwendet und wird daher bis auf weiteres als Kostenfunktion in dieser Arbeit verwendet. Die Entwicklung beziehungsweise die Evaluierung einer passenden Kostenfunktion bietet an sich genug Inhalt für den Umfang einer Diplomarbeit. L1-Norm(Least Absolut Deviations) und L2-Norm(Least Squares Error) sind ebenfalls zwei weitverbreitete Kostenfunktionen. Neben den hier erwähnten Funktionen gibt es beispielsweise noch die exponentielle Kostenfunktion, die Kreuzen Entropie Kostenfunktion, die Hellinger Distanz, die Kullback-Leibler Divergenz und viele mehr.

2.4.1.2 Reinforcement Learning

Dieser Ansatz ist eine Mischung aus supervised und unsupervised Learning. Hier wird dem System mitgeteilt, ob das Ergebnis korrekt ist oder nicht, aber es werden keine näheren Berechnungen zu der Abweichung vom gewünschten Wert durchgeführt und dem System mitgeteilt. Dadurch kann das Netz gezielter lernen als beim unsupervised Learning, es ist aber weniger Aufwand notwendig und es lassen sich leichter Trainingsdaten generieren als beim supervised Learning. Diese Methode eignet sich beispielsweise gut für Simulationen, in denen ein Auto selbstständig lernen soll eine Straße entlang zu fahren. Sobald es durch die Verarbeitung von Input Parametern wie Position, Objekte, Straßenverlauf und ähnlichem zu entscheidenden Parametern wie Lenkeinschlag und Geschwindigkeit kommt, wird überprüft, ob es damit von der Straße abkommt. Sollte dies passieren, wird es zurück auf die Straße gesetzt und dem System mitgeteilt, dass die produzierten Output-Parameter für die präsentierten Input-Parameter falsch waren beziehungsweise zu einem falschen oder unerwünschten Ergebnis geführt haben.

2.4.1.3 Unsupervised Learning

Bei diesem Ansatz lernen Netze „von selbst“, es werden keine vorbeschrifteten Datensätze verwendet. Dieser Ansatz hat den Vorteil, dass der Trainingsprozess ohne größeren Aufwand funktioniert, da wenig Werte berechnet und festgelegt werden müssen. Diese Art wird beispielsweise für Speicher verwendet, wo dem Anwender egal ist, wie das Netz Gewichtungen setzt, solange es eine Information wiedergeben kann. Weitere Möglichkeiten für unsupervised Learning ist beispielsweise Textverarbeitung, bei der das Netz vier Wörter eines Satzes präsentiert bekommt und das letzte Wort vorhersagen muss, wofür nur die Trainingsdaten (Sätze) vorhanden sein müssen, aber keine Labels. Weit verbreitet ist auch der Einsatz solcher Netze zur Rekonstruktion oder Rauschentfernung vom Input, bei der das Netz dadurch trainiert wird, dass die Output Neuronen den Input Neuronen möglichst ähnlich sein sollen. Die Idealvorstellung ist oft, dass ein Netz in der gewünschten Anwendungs-Umgebung laufen gelassen wird, und es sich alle Eigenschaften „von selbst“ beibringt und merkt, was teilweise tatsächlich funktioniert.

2.4.1.3.1 Unsupervised Pre-Training

Unsupervised Pre-Training ist ein Verfahren, das mit den Methoden von unsupervised Learning ein Netz vortrainieren soll und damit auf einfache Art und Weise die Gewichtungen in eine gute Position und in die Nähe eines Minimums bringt. Meist wird es kombiniert mit einem supervised Fine-Tuning, bei dem dann tatsächlich das Minimum erreicht wird und das Netzwerk auf seinen endgültigen Zweck trainiert wird. Außerdem wird damit mithilfe des schnellen unsupervised Learnings die Dauer des langsamen supervised Learnings verkürzt. Zum Einsatz kommt es beispielsweise bei DBNs, Stacked Autoencoders oder normalen Feed-forward Neural Nets. Informationen darüber finden sich in [34], [48], [7].

2.4.2 Lernregeln

2.4.2.1 Hebb-Regel

Die Hebb-Regel ist eine der einfachsten Lernregeln und stammt von Donald Olding Hebb[49]. Diese Regel wird oft vereinfacht durch den Spruch „What fires together, wires together“ erklärt[19]. Das bedeutet, dass Gewichtungen zwischen zwei Neuronen nur verändert werden, wenn beide Neuronen gleichzeitig aktiv sind. Die

Gewichtung wird abhängig vom Lernparameter erhöht. Da für die klassische Hebb-Regel die Neuronen nur die Null oder einen positiven Wert haben können, hat diese Lernregel ein paar Nachteile. So können Gewichtungen immer nur erhöht, aber niemals verringert werden. Außerdem besteht durch den eingeschränkten Wertebereich nur ein geringer Bereich an möglichen Zuständen. Diese Probleme können aber durch verschiedene Anpassungen behoben werden, wie beispielsweise die Gewichtungen für Neuronen, die nicht gleichzeitig aktiv sind, zu senken. Die Hebb-Regel kann für supervised, unsupervised und reinforcement Learning verwendet werden.

2.4.2.2 Delta-Regel

Die Delta-Regel wird auch als Widrow-Hoff-Regel oder Least Mean Square-Regel (LMS-Regel) bezeichnet und stellt eine Regel für supervised Learning dar [19]. Dieses Verfahren ist nur für das Training von zweischichtigen nicht-rekurrenten Netzen geeignet. Das Vorgehen ist, dass das vorhandene Output Signal eines Neurons mit dem gewünschten Output Signal verglichen wird und je nach Abweichung Änderungen vorgenommen werden. Dieses Verfahren erinnert an das Grundprinzip von Backpropagation und ist vermutlich deren Vorgänger. Die Delta-Regel ist prinzipiell für Incremental Training ausgelegt, kann aber einfach für die Verwendung mit Batch Training abgeändert werden.

2.4.2.3 Backpropagation

Als besondere Art der Lernregel ist die Backpropagation (oft „Backprop“ genannt) oder Fehlerrückführung aufzuzählen, da es sich hier eher um ein Verfahren handelt, dessen Ziel es ist auch Netze mit Hidden Layers optimieren zu können. Bei der Anwendung von Backpropagation muss supervised Learning durchgeführt werden und eines der Gradientenverfahren genutzt werden. Die Updates können aber sowohl inkrementell als auch als Batch durchgeführt werden. Dabei wird anhand einer Kostenfunktion festgestellt, wie weit der vorhandene Output eines Netzes neben dem gewünschten Output liegt, und dann nacheinander für jede Schicht und für jede Verbindung die Gewichtung angepasst, von dem Output Layer zurück zum Input Layer. Es kann mit zahlreichen Varianten des Gradientenverfahrens durchgeführt werden, einige davon werden in Kapitel 2.4.3.2 erwähnt. Eine Sonderform stellt Backpropagation Through Time (BPTT) und Truncated BPTT dar, zwei Verfahren zum Trainieren von RNNs. Backpropagation und seine Sonderformen sind für Offline Training gedacht. Da die heutigen Netze meist komplex sind und mehrere Schichten besitzen, auch Deep Neural Network (DNN) genannt, kommt meist Backpropagation zum Einsatz, da dieses Verfahren auch die Veränderung von Gewichtungen der Hidden Layers erlaubt. Abgesehen davon ist das dafür notwendige supervised Learning sehr weit verbreitet.

2.4.2.3.1 Backpropagation Through Time

Backpropagation Through Time (BPTT) ist eine Erweiterung der Backpropagation für RNNs. Es ist für diese Netze nicht ausreichend die Updates mit Backpropagation von den Output Neuronen zurück zu den Input Neuronen zu propagieren, da hierbei die rekurrenten Verbindungen nicht involviert werden und ein RNN dadurch seine besondere Eigenschaft und seinen Mehrwert verliert. Dieser Ansatz wurde laut Literaturrecherche von mehreren Forschern gleichzeitig in ähnlicher Weise entwickelt. Bei diesem Verfahren wird ein RNN so viele Schritte aufgerollt, wie die Input Sequenz lang ist. Das bedeutet, wenn beispielsweise das Wort „EXAMPLE“ mit einer Länge von 7 Zeichen verarbeitet werden sollen, dann wird ein RNN, wie in Abbildung 15 links zu sehen, über insgesamt 7 Schritte ausgerollt, wie vereinfacht in Abbildung 15 rechts dargestellt. Dadurch entspricht das Netz im Grunde einem Feed-forward Neural Net. Nun wird die Input Sequenz, beispielsweise das Wort „EXAMPLE“, dem Netz präsentiert und die Signale bis in den letzten Schritt in den Output Layer berechnet. Dort wird anhand einer Kostenfunktion eine Abweichung berechnet und Updates wieder zurück durch alle Schritte bis in den ersten Schritt propagiert. Für ein Wort mit beispielsweise 12 Zeichen würde das Netz demnach auf insgesamt 12 Schritte aufgerollt werden. Diese Netze heißen auch Many-To-One Netze, da meistens nur der Output des letzten Schritts (in Orange) verwendet wird und alle anderen verworfen (in Grau) werden. Es kann allerdings auch der Output in jedem Schritt verwendet werden, dann wird es Many-To-Many Netz genannt.

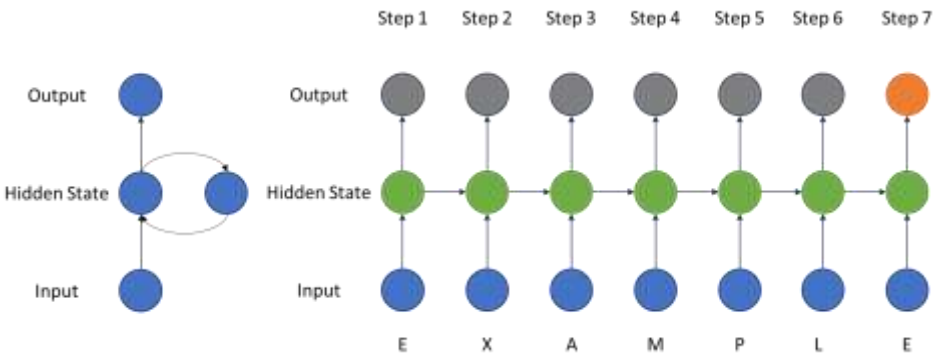

Abbildung 15: Links in der Abbildung ein einfaches RNN, rechts dasselbe RNN ausgerollt über 7 Schritte

Da dieser Ansatz bei längeren Sequenzen, wie bei der Erfassung ganzer Texte mit mehreren tausend Stellen, verschiedene Probleme wie hohe Rechenleistung und verschwindend kleine Updates hat, wurde der Ansatz Truncated BPTT (TBPTT) entwickelt. Dort wird das RNN ebenfalls für die Länge der Input Sequenz ausgerollt, allerdings wird alle k_1 Schritte BPTT für k_2 Schritte zurück durchgeführt, wie zu sehen in Abbildung 16, wo jeden Schritt Updates zwei Schritte zurück propagiert werden. Dadurch hat der jeweilige Hidden State den Wert der langen zusammenhängen Sequenz und Zusammenhänge bleiben erhalten, aber die Updates sind viel kleiner und effektiver, sowie der Rechenaufwand dafür wesentlich geringer. Eine einfachere Methode um dieses Problem mit BPTT zu lösen wäre, die gewünschte lange Input Sequenz auf mehrere kleine Sequenzen herunterzubrechen und jede Teilsequenz als eigene Sequenz zu behandeln und mit BPTT zu verbessern. Hierbei würden allerdings die Zusammenhänge der großen Sequenz verloren gehen.

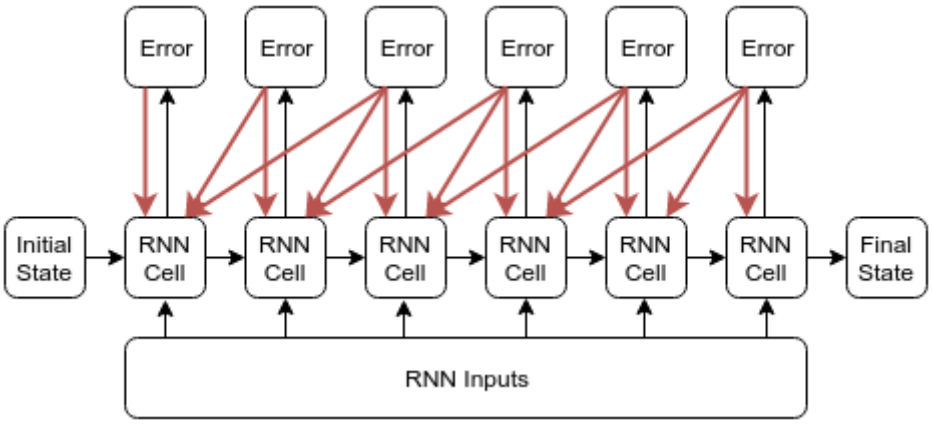

Abbildung 16: So sieht beispielsweise TBPTT aus, die Updates werden hier immer nur 2 Schritte zurückpropagiert[50]

2.4.2.4 Real-Time Recurrent Learning

Real-Time Recurrent Learning (RTRL) ist ebenfalls ein Ansatz zur Verbesserung von RNNs. Dabei wird aber im Unterschied zu BPTT das Signal nicht zuerst vorwärts und dann rückwärts in der Zeit propagiert, sondern ausschließlich vorwärts propagiert. Dadurch müssen im Gegensatz zu BPTT nicht die Hidden States aller vorherigen Schritte für die Rückpropagierung der Updates gespeichert werden. Dieses Verfahren erlaubt Online Training, da die Error-Funktion immer nur für den derzeitigen Schritt berechnet wird. Verglichen mit BPTT ist der Rechenaufwand für dieses Netz relativ hoch, vielleicht findet es deswegen in der Praxis kaum Anwendung. [51]

2.4.2.5 *Competitive Learning*

Diese Lernregel ist nur mit unsupervised Learning möglich, das Netz bekommt als Orientierung also nicht den korrekten Input präsentiert, stattdessen versucht es, ähnliche Daten als Kategorie zusammenzufassen. Diese Lernregel ist vor allem für kompetitive Netze gedacht, die mit einem Pattern Associator vergleichbar sind[19]. Dieses Verfahren setzt sich im Wesentlichen aus drei Schritten zusammen:

- Erregung: Dem Netz wird ein Input gegeben und der Output aller Neuronen wird normal berechnet.
- Wettbewerb: Danach wird der Netinput aller Neuronen in der Output Schicht verglichen und das Neuron, mit dem höchsten Wert, als „Sieger-Neuron“ deklariert
- Adjustierung der Gewichte: In der letzten Phase werden die Gewichtungen aller Verbindungen, die zum Sieger-Neuron führen verändert. Dabei wird die Gewichtung so verändert, dass sie dem Input angeglichen wird. Das heißt, wenn die Gewichtung höher als die auftretende Aktivität ist wird die Gewichtung gesenkt, und umgekehrt. Alle anderen Verbindungen bleiben unverändert, dieses Konzept nennt man daher auch „Winner takes it all“[19].

Ein großer Vorteil dieses Netzes ist neben der Möglichkeit des unsupervised Learnings, dass Redundanzen entfernt werden können, da ähnliche Muster dieselben Neuronen erregen werden, das heißt ein Output Neuron auf mehrere ähnliche Inputmuster reagiert. Ein großer Nachteil ist allerdings, dass es aufgrund der Initialisierung der Gewichtungen und der ersten Trainingszyklen passieren kann, dass fast immer ähnliche Neuronen Phase zwei gewinnen. Dadurch müssten diese Neuronen fast alle Inputmuster nachbilden, was aber nicht möglich ist. Weite Teile des Netzes würden dadurch inaktiv bleiben und die paar aktiven Neuronen würden keine sinnvollen Muster erkennen können. Dies lässt sich jedoch durch verschiedene Mechanismen verhindern oder unterdrücken, zum Beispiel durch eine Begrenzung der Summe der Gewichte zu einem Neuron.

2.4.3 Verbesserung

Grundsätzlich zielen die meisten Lern-Verfahren auf eine Veränderung der Gewichtungen ab, es existieren aber noch einige ungewöhnlichere Möglichkeiten, wie beispielsweise Entwicklung oder Beseitigung von Verbindungen, Änderung der Aktivierungsfunktionen, Aufnahme oder Eliminierung von Neuronen, und Weitere[19]. Um aber bei den gängigen Varianten zu bleiben, wird sich diese Arbeit hier nur auf Verfahren konzentrieren, bei denen das Netz über Veränderung der Gewichtungen verbessert wird.

2.4.3.1 *Allgemein*

2.4.3.1.1 Online Training versus Offline Training

Online und Offline Training unterscheidet prinzipiell, woher die Daten für das Training eines Netzes stammen und wie das Training abläuft. Für Offline Training wird ein statischer, meist großer Datensatz verwendet, mit dem ein System wiederholt trainiert werden kann. Im Gegensatz dazu muss bei Online Training nicht zwingend ein fixer Datensatz existieren, sondern es können die im Normalbetrieb anfallenden Daten verwendet werden um das System zu trainieren. Am Beispiel von einem Spam-Mail Classifier wäre Offline Training, wenn ein Datensatz mit tausenden Mails genutzt wird um den Classifier zu trainieren. Online Training wäre, wenn die eingehenden Mails direkt in das System weiterleitet und zum Training genutzt werden. Dies bietet einerseits den Vorteil, dass kein großer Datensatz vorhanden sein muss und dass durch die sequentielle Art des Trainings auch nach Abschluss des Trainings noch einzelne wichtige Daten trainiert werden können. So kann man beim Online Training beispielsweise dem System noch nach Abschluss des Trainings eine besonders raffinierte Spam Mail geben, die das System zusätzlich lernen soll, während beim Offline Training diese Mail dem Datensatz hinzugefügt wird und der ganze Datensatz erneut trainiert werden muss. Gerade bei großen Datenmengen ist Online Training sehr beliebt, da hier jederzeit das Training gestoppt und wiederaufgenommen werden kann. Teilweise wird Offline und Online Training fehlerhaft mit Batch und Incremental Training vermischt. Online Training ist immer inkrementell.

2.4.3.1.2 Verbesserungszeitpunkt

Es lassen sich prinzipiell zwei Ansätze unterscheiden, wann die Verbesserung im Netz stattfindet: Incremental und Batch Training. Bei Incremental Training werden die Gewichtungen nach jedem Trainings-Input verändert. Im Gegensatz dazu werden beim Batch Training Gewichtungen erst nach dem Durchlauf aller Trainingsdaten eines Batches verändert. Es existieren auch Mini-Batch Verfahren, bei denen ein Trainingsset in mehrere Batches zerlegt wird und Änderungen nach jedem Mini-Batch vorgenommen werden. Die Wahl des Ansatzes hat einen Einfluss auf das Trainingsverhalten des Netzes, so führt beispielsweise die hohe Varianz der Änderungen bei dem Incremental Training dazu, dass das Netz länger benötigt um sich auf einem Gradienten Minimum einzupendeln. Incremental Training kann durch die sofortigen Änderungen am Netzwerk leichter gestoppt und wiederaufgenommen werden. So kann unter anderem das Training leichter beendet werden, wenn die Erkennungsrate optimal ist, was in vielen Fällen Overfitting verhindert. Im Gegensatz kann Batch Training dafür besser parallelisiert werden, da Updates nur nach dem Durchlauf eines ganzen Batchs passieren und bis dahin einzelne Instanzen unabhängig arbeiten können.

2.4.3.1.3 Lernrate

Durch den Optimierungsalgorithmus werden nur die Richtung und die Art der Veränderung auf der Funktion festgelegt, um zu einem Minimum zu gelangen. Die Lernrate legt dabei die Schrittweite fest. Wenn die Lernrate zu hoch ist, kann das Minimum überschritten werden, was nicht nur den Lernprozess verlängert, sondern auch dazu führen kann, dass der Gradient vom globalen Minimum in ein lokales Minimum wechselt. Wenn hingegen die Lernrate zu niedrig ist, kann das Training um ein Vielfaches länger dauern, da nur kleine Änderungen passieren oder das globale Minimum gar nicht erreicht wird. Es existieren basierend auf Erfahrungen verschiedene Vorschläge für Lernraten. Dabei ist es üblich, ein Netz hintereinander mit verschiedenen Lernraten zu trainieren und das Ergebnis zu vergleichen. Die Lernrate ist ein Hyperparameter. Übliche Lernraten sind wie in [3] und [6] verwendet 0.01 als initialer Wert und 0.09 für Momentum, falls verwendet.

2.4.3.2 Optimierungsalgorithmus

Die hier erwähnten Optimierungsalgorithmen sind im Grunde nur für Backpropagation anwendbar. Optimierungsalgorithmen für andere Lern-Regeln sind einfacher und wurden teilweise bereits in Kapital 2.4.2 behandelt. Mit dem Optimierungsalgorithmus wird festgelegt, nach welchem Verfahren die Gewichtungen im Zuge des Trainings geändert werden müssen, um die Kosten des Netzes, also den durchschnittlichen Fehler, zu reduzieren. Eine Schwierigkeit darin besteht, dass das Netzwerk nicht nur auf die Erkennung oder Vorhersage eines Merkmals ausgelegt ist, das heißt die Veränderung von Bias und Gewicht soll zwar die Verarbeitung dieses Merkmals möglichst verbessern, ohne aber die Verarbeitung von anderen Merkmalen zu stören.

2.4.3.2.1 Gradientenverfahren

Gradientenverfahren werden sehr häufig zur Optimierung genutzt, da sie einfach zu berechnen sind. Bei diesem Vorgang werden die Kosten berechnet, in dieser Arbeit mittels der quadratischen Kostenfunktion, um vom Output Layer zurück zum Input Layer bei jeder Verbindung die Gewichtung so anzupassen, dass der Input zum gewünschten Output führt. Dies kann man sich vorstellen, indem die Kostenfunktion als Funktion gesehen wird, deren Input-Werte sich über alle Gewichtungen definiert und für die ein x-Wert mit dem niedrigsten y-Wert gefunden werden soll, es wird also ein Gradienten-Minimum gesucht. Durch die Berechnung der Kosten kann dieses Verfahren nur bei dem supervised Learning verwendet werden. Da die Rückführung, auch Update genannt, durch das Netzwerk beim klassischen Gradientenverfahren nur einmal für das gesamte Trainingsset durchgeführt wird, ist dieses Verfahren sehr langsam und kann bei großen Trainingssets zu Problemen führen, unter anderem aufgrund der vielen Updates die im Speicher bleiben. Dadurch wird dieses Verfahren auch Batch Verfahren genannt, da es sich um Batch Training handelt. Während dieses Verfahren zu einem sehr stabilen Lernverhalten führt, kann es dafür leicht passieren, dass lokale Minima nicht mehr verlassen werden können. Eine Schwierigkeit der Gradientenverfahren besteht in der Auffindung einer passenden Lernrate und darin, dass für alle Datensätze dieselbe Lernrate angewendet wird. Ersteres bewirkt, dass das Netzwerk entweder langsam konvergiert oder sich durch schnelle Änderungen über Minima hinwegsetzt. Zweites führt dazu, dass

Änderungen an den Gewichtungen auch in derselben Stärke passieren, wenn die Input-Information nur sehr selten vorkommt, das Netz wird diese Information also nicht oder nur langsam lernen. Probleme, die aus der Form der Funktion entstehen können, bereiten lokale Minima, flache Plateaus, Überspringen guter Minima, direkte und indirekte Oszillation und Sattelpunkte [19].

2.4.3.2.2 Stochastisches Gradientenverfahren

Das stochastische Gradientenverfahren (oft „SGD“ für engl. stochastic gradient descent) stellt eine Abwandlung des Gradientenverfahrens dar. Es führt Updates für jeden Trainingsinput einzeln durch und ist daher um einiges schneller als das normale Gradientenverfahren. Es handelt sich daher um ein inkrementelles Training. Weiter spricht man bei diesem Verfahren auch von Online Backpropagation, da ein bereits trainiertes Netz leicht mit einzelnen neu hinzukommenden Daten trainiert werden kann. Das Verfahren hat dafür den Nachteil, dass es durch einzelnen Updates eine sehr hohe Varianz hat und leicht über das Minimum hinausschießt, was dazu führt, dass die Konvergenz des Netzes länger dauert.

2.4.3.2.3 Mini Batch Gradientenverfahren

Das Mini Batch Gradientenverfahren ist ebenfalls eine Abwandlung des Gradientenverfahren. Dieses Verfahren versucht die Vorteile von SGD und dem klassischen Gradientenverfahren zu kombinieren, es berechnet zwar den Fehler für jeden Trainingsinput einzeln, führt die Updates aber erst am Ende des Trainingssets aus. Durch die geringere Zahl an Updates ist die Varianz geringer als bei dem stochastischen Verfahren. Da der Fehler pro Trainingsinput berechnet wird, lässt sich dieses Verfahren auch sehr gut parallelisieren. Es stellt eine Mischung aus Batch Training und inkrementellem Training dar.

2.4.3.2.4 Momentum

Die hohe Varianz des Verfahrens SGD führt zu teils starken Oszillationen, die den Gradienten durch „seitliche“ Bewegungen am Weg zum Gradienten Minimum stark verlangsamen können. Dieses Problem soll durch das Momentum Verfahren behoben werden, das den Gradienten in die richtige Richtung beschleunigen und in die falsche, beziehungsweise seitliche, Richtung bremsen soll. Dies wird dadurch erreicht, indem zu dem aktuellen Update ein Wert aus dem vorherigen Update addiert wird. Der dafür verwendete Momentum-Wert wird meist mit 0.9 gewählt, wie beispielsweise von [3] oder [6] verwendet. Dieses Verfahren ähnelt dabei im Grunde den physikalischen Umständen, wenn man einen Ball einen Berg runterrollen lasst, da der Ball immer mehr Fahrt aufnehmen wird und damit eine stabile Bahn den Berg hinunter hat. Ein damit verbundenes Problem ist jedoch, dass der Gradient nach einem langen Abstieg ein so hohes Momentum aufweist, dass er problemlos über das Minimum hinwegschießt und den gegenüberliegenden Berghang wieder hinaufsteigt. Momentum kann als Erweiterung des SGD gesehen werden.

2.4.3.2.5 Adagrad

Dieser Ansatz eignet sich für einen sehr unterschiedlichen Datensatz, in dem manche Daten sehr selten vorkommen, während andere Daten häufig vorkommen. Im Grunde versucht dieses Verfahren, für seltene Daten eine größere Änderung des Gradienten zu bewirken und für häufige Daten kleiner Änderungen. Dies wird dadurch realisiert, dass im Gegensatz zu den meisten anderen Verfahren nicht eine Lernrate für alle Werte verwendet wird, sondern jeder Wert eine eigene Lernrate hat, die sich an den vorherigen Lernraten für diesen Wert orientiert. Ein großer Vorteil dieses Ansatzes ist außerdem, dass die Lernrate nicht mitunter mühsam ausgewählt werden muss, sondern meist auf 0.01 gesetzt wird und sich im Laufe des Verfahrens ohnehin für die Werte selbst anpasst [52]. Auch dieses Verfahren stellt eine Erweiterung des SGD dar. Der größte Nachteil dieses Verfahrens ist, dass durch die Berücksichtigung der vorhergegangenen Änderungen die zukünftigen Änderungen immer kleiner werden, bis der Gradient sich irgendwann nicht mehr oder kaum ändert, das Netz also nicht mehr verbessert werden kann. Dieses Problem wird versucht in dem Verfahren AdaDelta zu beheben [53].

2.4.3.2.6 Adaptive Moment Estimation

Adaptive Moment Estimation (Adam) ist ein weiteres Verfahren, das eine adaptive Lernrate verwendet und für jeden Wert eine eigene Lernrate berechnet. Das Verfahren versucht in einer Mischung aus AdaDelta und Momentum vorhergegangene Probleme zu umgehen und gilt damit heutzutage als eines der besten adaptiven Verfahren, vor allem bei sehr unterschiedlichen Daten, denn während Netze durch Adam sehr schnell konvergieren, gibt es eine geringe Varianz und eine gute Lernrate, die nicht droht auf null zu sinken. Die Adam Optimierung wird auch von führenden Entwicklern in diesem Bereich verwendet[54]. Adam ist eine Erweiterung des SGD Verfahrens.

2.4.4 Validation

Im Zuge der Trainingsdurchläufe wurde die Fähigkeit des Netzes sich zu spezialisieren getestet, indem durch die ständige Berechnung der Kostenfunktion im Falle von supervised Learning oder durch Abgleich von Input und Output bei unsupervised Learning überprüft wird, wie gut das Netz Trainingsdaten erkennt. Die Trainingsdaten sind dem Netz jedoch nach dem ersten Durchlauf „bekannt“. Um damit im Zuge der Anwendung auch neue Daten richtig zu erkennen, muss daher getestet werden, wie gut die Fähigkeit des Netztes zu generalisieren ist. Dafür werden als Input für das Netz neue Daten gewählt und erneut durch Abweichung vom Soll-Ergebnis gemessen, wie gut das Netz für diese neuen Daten abschneidet. Im Unterschied zur Trainingsphase werden allerdings keine Änderungen mehr an den Gewichtungen vorgenommen. Diese Phase wird auch Test- oder Evaluierungs-Phase genannt.

2.4.5 Überanpassung

Wenn ein komplexes Netz zu lange auf ein Trainingsset trainiert wird, kann es zu einer sogenannten Überanpassung (engl. Overfitting) kommen. Dies bedeutet, dass das Netz nahezu perfekt an die Trainingsdaten angepasst ist und diese fast fehlerlos erkennt, durch diese extrem starke Anpassung an diese Daten können aber neue Daten nur mit einer sehr hohen Fehlerquote durch das Netz erkannt werden[17]. Denn durch das wiederholte Training ist das Netz perfekt auf das Trainingsset abgestimmt, aber es fehlt der Spielraum zur Generalisierung, was für die richtige Erkennung neuer Daten absolut notwendig ist. Grundsätzlich kann es drei Gründe für Overfitting geben: Zu viele Neuronen, zu wenig Trainingsdaten oder ein „schlecht“ konzipiertes Netzwerk, wobei diese Gründe nicht immer eindeutig sind und teilweise zusammenhängen, so kann ein Netz sich überanpassen da für die wenigen Testdaten zu viele Neuronen vorhanden sind[54]. Es existieren verschiedene Mechanismen um Overfitting zu verhindern. Diese Methoden umfassen unter anderem einen frühzeitigen Trainingstop, wenn die Error-Rate unter einen gewissen Schwellwert fällt, oder das Hinzufügen von zufälligem Rauschen zu den Trainingsdaten. Weitere Möglichkeiten sind das Trainingsset zu vergrößern und die Komplexität des Netzes zu reduzieren, da zu viele Parameter für eine einfache Aufgabe schneller zu einer Überanpassung führen. Namentlich existieren unter anderem die Konzepte Dropout, Drop-Connect und Maxout[17]. Effektive Methoden gegen Overfitting sind essentiell um leistungsfähigere Netze bauen zu können, denn sie ermöglichen leichteres Training von Netzen, da nicht mehr gigantische Mengen an Daten vorhanden sein müssen, um Overfitting im Vorhinein zu verhindern.

2.4.5.1.1 Dropout

Dropout ist ein bewährtes Verfahren gegen Überanpassung in neuronalen Netzen. Jede Iteration werden x % der Neuronen im Netz deaktiviert, dann wird wie gewohnt der Input Richtung Output propagiert und die Verbesserungen an den Gewichtungen durchgeführt. Die deaktivierten Neuronen werden dabei in keinem Schritt berücksichtigt, Verbindungen zu und von ihnen werden in dieser Iteration nicht verändert. Danach werden die Neuronen wieder aktiviert und für die nächste Iteration zufällig Neue ausgesucht. Eine übliche Dropout-Rate ist 25-50%. Dropout ist heute sehr weit verbreitet, da es einfach angewendet werden kann.

2.4.5.1.2 DeCov

DeCov zielt darauf ab, die Co-Adaption oder die Kovarianz zwischen Units zu stören. Es kann ohne zusätzliche Überarbeitungen auf ein bestehendes Netz angewendet werden, sowohl auf einen Layer, als auch auf alle Layers[17]. Dieses Verfahren wird aber bis jetzt von Frameworks für neuronale Netze laut Recherche nicht unterstützt. (Stand Februar 2018)

2.4.5.1.3 L1 Regularization und L2 Regularization

Sehr weit verbreitet sind auch die beiden Verfahren L1 Regularization und L2 Regularization, die einfach erklärt Gewichtungen „bestrafen“, wenn sie zu groß werden, da die damit verbundenen Neuronen dadurch sämtliche Informationen versuchen abzubilden, während das restliche Netzwerk nichts mehr lernt. Wenn der Bestrafungswert zu hoch ist, kann es allerdings das Lernen des Netzes stilllegen. Dadurch, dass L1 Regularization einzelne Features auf null reduziert, eignet sich diese Methode auch für Feature Selection. Bei beiden Verfahren wird zwar eine spezielle Kostenfunktion verwendet, es sollte aber nicht mit den L1-Norm und L2-Norm Kostenfunktionen verwechselt werden, die keine „Bestrafung“ zu den Kosten addieren.

2.5 Verschiedenes

2.5.1 Schritt, Iteration, Batch, Mini-Batch, Epoche

Diese fünf Begriffe aus dem Training haben eine sehr ähnliche Bedeutung, wodurch es immer wieder zu Verwirrungen kommt. In dieser Arbeit wird eine Verwendung dieser Wörter genutzt, die im Internet weit verbreitet ist. Demnach ist Schritt und Iteration dasselbe und bezeichnet eine Vorwärts-Propagierung von einem Input durch das Netz und falls notwendig eine Rückpropagierung (Backpropagation) durch das Netz, inklusive Berechnung der Änderung der Gewichtungen. Änderungen an den Gewichtungen werden aber bei Batch-Verfahren nur gesammelt am Ende eines Batchs durchgeführt, während sie bei Online Verfahren sofort durchgeführt werden. Ursprünglich wurde mit Batch nur angedeutet, dass Änderungen nicht wie beim inkrementellen Training nach jedem Input, sondern lediglich am Ende des Trainingssets durchgeführt werden, wie in Kapitel 2.4.3.1.2 beschrieben. Meist wird die Bezeichnung Batch aber mit Mini-Batch gleichgesetzt, ein Verfahren, bei dem das Trainingsset nicht nur wie ursprünglich in einen Batch zusammengefasst wird, sondern in mehrere Batches unterteilt wird. Weiter wird statt der korrekten Bezeichnung der Größe des Mini-Batches „Mini-Batch Größe“ meist die Bezeichnung „Batch Größe“ verwendet, weshalb in dieser Arbeit ebenfalls diese Bezeichnung verwendet wird. Ein Batch ist dabei eine beliebig gewählte Unterteilung des gesamten Trainingssets, oft wird eine Batch-Größe von 100 genommen, was beispielsweise bei dem weitverbreiteten MNIST Trainingsset mit 55.000 Datensätzen zu 550 Batches mit je 100 Daten führt. Eine Epoche stellt eine vollständige Verarbeitung des Trainingssets dar, am vorherigen Beispiel der MNIST Daten ist eine Epoche vorbei, wenn alle 55.000 Datensätze, also alle 550 Batches, einmal durch das Netz verarbeitet wurden und entsprechende Änderungen am Netz vorgenommen wurden. Die Anzahl der Epochen wird oft dynamisch an die Kostenfunktion geknüpft, sollten die Kosten einer Epoche unter einen gewissen Schwellwert fallen wird das Training beendet. Ein Training besteht somit aus mehreren Epochen, alle Trainingsdaten werden mehrmals durch das Netzwerk verarbeitet. Umgelegt auf die stark vereinfachte Abbildung 17 würde das Trainingsset aus den vier Daten a, b, c und d bestehen, die Batch Größe beträgt zwei. Insgesamt wurden drei Epochen zu je zwei Batches durchgeführt. Jeder Datensatz wurde also drei Mal verarbeitet. Wie an der Reihenfolge der Buchstaben gut zu erkennen ist, sollten die Daten eines Batchs immer zufällig durchmischt werden, da dies die Konvergenz des Netzes beschleunigt.

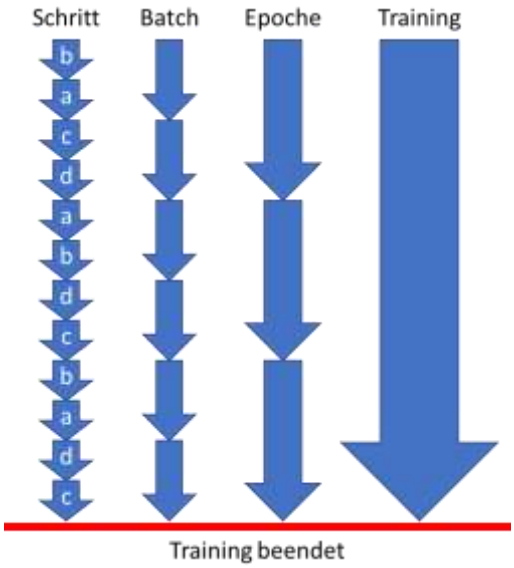

Abbildung 17: Verdeutlichung des Zusammenhangs zwischen Schritt, Batch, Epoche und Training.

2.5.2 Feature

Als Feature oder Merkmal werden meist Eigenschaften von Daten bezeichnet, im Zusammenhang mit neuronalen Netzen vor allem Eigenschaften die relevant für die Auswertung sind. So können beispielsweise in einem Datensatz mit Blumen die Features Blütenfarbe, Blütengröße, Gesamthöhe, Anzahl Seitentriebe relevant sein um Blumen zu verarbeiten, entweder für Regression oder Klassifizierung. Teilweise werden als Feature auch einfach unbestimmte Eigenschaften oder Messpunkte bezeichnet. So wird es beispielsweise als Feature Extraction bezeichnet, wenn man aus einem Datensatz von 1000 Bildern von handschriftlichen Ziffern automatisch Daten oder Messpunkte extrahieren lässt, die verwendet werden können um handschriftliche Ziffern zu klassifizieren. Dabei wird es sich in diesem Fall bei den Features um einzelne Pixelwerte handeln.

2.5.3 Dimensionalität

Die Dimensionalität der Daten ist eine wichtige Eigenschaft, da sie unter anderem darüber entscheidet, wie sich Daten darstellen lassen und welche Möglichkeiten es zur Verarbeitung gibt. Während zweidimensionale Daten meist mit einfacheren mathematischen Verfahren verarbeitet werden können, wird es bei achtdimensionalen Daten schwierig. Außerdem lassen sich zweidimensionale und dreidimensionale Daten relativ einfach darstellen, was bei mehr Dimensionen nicht mehr möglich ist. Oft wird für Dimensionen die eher aus dem Bild- und Videobereich bekannte Bezeichnung xD verwendet, wobei x die Zahl der Dimension ist, also beispielsweise 2D für zweidimensional oder 8D für achtdimensional. Diese Dimensionen sind auch wichtig, da eine Dimension meist einem Neuron entspricht. So sind beispielsweise die Daten einer Schicht mit zwei Neuronen zweidimensional, weil die Werte des einen Neurons als X-Wert und die Werte des anderen Neurons als y-Wert verwendet werden können und damit jede beliebige zweidimensionale Funktion dargestellt werden kann. Aufgrund dieses Zusammenhangs wird bei Autoencodern und einer Reduktion von Neuronen auch oft von einer Reduktion der Dimensionalität gesprochen.

2.5.4 Covariate Shift

Covariate Shift ist eine Veränderung der Verteilung im Definitionsbereich einer Funktion und liegt vor, wenn Trainingsdaten und reale Daten aus einer unterschiedlichen Grundgesamtheit stammen oder eine unterschiedliche Verteilung haben. Vereinfacht gesagt, es ist eine Änderung des Inputs von Training zu Einsatz,

mit der der Algorithmus nicht umgehen kann. Ein klassisches Beispiel[55] für Covariate Shift ist, dass beim Training eines Modells für Sprachsteuerung im Auto eine zufriedenstellend hohe Genauigkeit erreicht wird, sich dann aber im Einsatz zeigt, dass die Erkennungsrate unvertretbar niedrig wird. Dies liegt daran, dass das Modell zwar hervorragend auf Stimmen trainiert wurde, mit den Hintergrundgeräuschen im Auto aber nicht umgehen kann und nicht darauf trainiert wurde, dass diese unwichtig sind. Allgemein können Gründe für Covariate Shift unpassende Trainingsdaten oder nicht ausreichend verteilte Trainingsdaten sein. Ein Beispiel für unpassende Trainingsdaten wäre, wenn künstlich mit einem Algorithmus Daten erzeugt werden, da keine echten Daten verfügbar sind, die künstlichen Daten aber nicht zu den echten Daten passen. Ein Beispiel für nicht ausreichend verteilte Trainingsdaten wäre, wenn ein Netz mit den Gesundheitsdaten der Bevölkerung in den USA trainiert wird, um später Gesundheitsdaten der Bevölkerung von Europa zu verarbeiten.

2.5.4.1 Internal Covariate Shift

Internal Covariate Shift ist ein Problem im Zusammenhang damit, dass sich während des Trainings für jede Schicht die Input Verteilung ändert, ähnlich wie bei Covariate Shift für das ganze Netz. Je mehr Layers das Netz hat, beispielsweise bei einem Microsoft Netz 152 Layers, desto größer die Auswirkungen. Dies wird dadurch hervorgerufen, dass die Änderung der Gewichtungen im Trainingsverlauf zu einer Änderung der Signale an die nächste Schicht führt und damit zu einer Änderung der Verteilung.

2.5.5 Batch Normalization

Batch Normalization wird unter anderem durchgeführt um (Internal) Covariate Shift zu verhindern[56], kann jedoch auch regulierende Effekte haben, die Dropout überflüssig machen. Batch Normalization kann wahlweise auf einzelne Layers eines Netztes angewendet. Zuerst wird der Output der Aktivierungsfunktion in einem Neuron mit der unten beschriebenen Formel normalisiert, danach mit dem Parameter g multipliziert, das Produkt mit Parameter b addiert. Die beiden Parameter g und b werden speziell für Batch Normalization verwendet und werden im Zuge des Trainings trainiert, wodurch das Trainingsverfahren generell geändert wird. Dieses Verfahren verhindert, dass Gewichtungen in einem Netzwerk sehr groß oder sehr klein werden. Zusätzlich kann jeder Trainingsbatch vor Verarbeitung in Bezug auf Varianz und Mittelwert mithilfe der untenstehenden Formel normalisiert werden. Durch dieses Verfahren erhält der Datensatz einen Mittelwert von null und eine Standardabweichung von 1. Damit sind sowohl die Input Daten normalisiert, als auch die Daten in dem Netzwerk, zumindest in den Schichten mit Batch Normalization:

$$z = \frac{x - m}{s}$$

- x ist der Datenpunkt
- m ist der Mittelwert des Training-Batchs
- s ist die Standardabweichung des Training-Batchs

Während dieses Vorgehen in vielen Fällen zu besseren Ergebnissen führt, führt es in einigen Fällen zu schlechteren, da die Daten so unterschiedlich sind und sich durch wichtige Details unterscheiden, die bei der Normalisierung abgeschwächt werden oder verschwinden. Siehe dazu auch [3]. Diese Normalisierung muss natürlich, wenn sie beim Training eingesetzt wird, auch bei der Validierung genutzt werden.

2.6 Anwendung

Ist das neuronale Netz trainiert, kann es für noch nicht kategorisierte Daten verwendet werden um deren Kategorie zu ermitteln oder für Regression verwendet werden. Dafür kann es aber immer nur auf eine Art von Daten verwendet werden, für die es trainiert wurde. Für jeden anderen Anwendungsfall oder für jede andere Art von Daten ist ein eigens trainiertes Netzwerk notwendig. Da neuronale Netze im Normalfall nie so gut trainiert werden können, dass sie alle Erkennungen fehlerfrei durchführen, ist es möglich sie über eine Feedback-

Funktion während der Verwendung laufend zu trainieren. Wenn der Benutzer ein Ergebnis sieht, das nicht dem Input entspricht, kann er angeben zu welchem Ergebnis der Input führen sollte, und das Netzwerk wird durch supervised Learning verbessert.

2.6.1 Anwendungsbeispiele

Neuronale Netze werden für Deep Learning oder Machine Learning eingesetzt und werden grundlegend für Klassifizierung von vorhandenen Daten oder zur Regression eingesetzt. Verschiedene Netze haben durch ihre Eigenschaften unterschiedliche Einsatzgebiete, nicht mit allen Netzen lassen sich dieselben Aufgaben lösen. CNNs eignen sich beispielsweise besonders gut zur Erkennung von Mustern, wie Handschrift, und werden daher auch für OCR oder Bild-Klassifizierung eingesetzt. Eine große Stärke haben neuronale Netze außerdem in Bezug auf unvollständige, fehlerhafte oder verrauschte Daten, da sie nicht auf ein exaktes Abbild der Trainingsdaten angewiesen sind, beziehungsweise sein sollten. Da sowohl Input als auch Output der neuronalen Netze Zahlen sind, lassen sich aber auch Muster in Zahlen erkennen. Weitere Beispiele sind Finanzanalyse, Wettervorhersage, Diagnose in der Medizin, Unterstützung bei Entscheidungen, Planungsaufgaben und Qualitätskontrolle[20]. Heutige Systeme bestehen oft aus einer Menge von kombinierten Netzen, um aufwendige Problemstellungen zu lösen. So verwendete beispielsweise Google für jeden Frame eines Videos ein eigenes, vortrainiertes CNN, verknüpft diese Netze mit einem LSTM und verbindet das Ergebnis mit dem Ergebnis eines LSTMs, welches die Frage „Was passiert in diesem Video“ verarbeitet hat, um genau diese Frage beantworten zu können[57].

2.6.2 Probleme von neuronalen Netzen

Immer eine Antwort

Durch die Art und Weise wie ein neuronales Netz funktioniert, gibt es immer ein Ergebnis, auch wenn das neuronale Netz für den Input nicht ausgelegt ist. Das neuronale Netz erkennt also nicht, wenn es eine Antwort nicht weiß. Wenn man zum Beispiel das auf Klassifikation von Katze, Hund, Schaf und Wal trainierte Feed-forward Neural Network verwendet und ein Bild von einem Adler als Input nimmt, wird das Netz trotzdem ein Ergebnis liefern und für jedes der vier Neuronen einen Wert ausgeben, auch wenn das Ergebnis natürlich falsch ist.

Stabilität-Plastizität-Dilemma

Ein neuronales Netzwerk ist nie „perfekt“, da bei jeder Veränderung von Gewichtung und Biases zugunsten der Erlernung eines Musters die Erkennung eines anderen Musters entweder unmöglich gemacht oder zumindest erschwert wird. Selbst bei einem unendlich großen Trainingsset wird ein neuronales Netzwerk daher nie perfekt sein und selbst im Trainingsset bei wiederholtem Durchlauf nicht alles richtig kategorisieren. Dieses Problem hat auch mit dem Stabilität-Plastizität-Dilemma zu tun, welches ART Netze lösen wollen.

Hohe Komplexität und Rechenaufwand

Ein weiterer Nachteil ist die hohe Komplexität der Netze. Bei dem weiter oben beschrieben Beispiel zur Erkennung von Zahlen in einem 30x30 Pixel Bild würde ein Netz bei zwei Hidden Layers mit jeweils 6 Neuronen insgesamt 922 Neuronen und 5496 Verbindungen besitzen, die zusätzlich eine Bias beziehungsweise eine Gewichtung benötigen. Die Anzahl an Verbindungen steigt circa linear zu den Messpunkten, bei einem Bild mit 60x60 Pixel wären dies bereits 3622 Neuronen und 21696 Verbindungen. Heutige Netze haben dabei teilweise mehr als 1.000.000 trainierbare Parameter und werden wochenlang trainiert[3]. Der damit verbundene Nachteil ist die extrem hohe Rechenleistung, die notwendig ist um ein neuronales Netz für einen Anwendungsfall zu bauen und zu trainieren, auch der Betrieb kann durchaus sehr rechenintensiv sein. Bei dem klassischen Anwendungsbeispiel der Ziffern-Klassifikation mithilfe der MNIST-Datenbank ergeben sich für ein Training ungefähr 431 Millionen Input-Werte, da es ungefähr 55.000 Bilder mit 28x28 Pixel gibt, die insgesamt beispielsweise bei zehn Epochen zehn Mal verarbeitet werden.

Generalisierung-Spezialisierung-Dilemma

Ein letzter großer Nachteil ist der Kompromiss zwischen Spezialisierung und Generalisierung. Ein Netz kann im einen Extremfall hoch spezialisiert sein und den bereits trainierten Input mit absoluter Sicherheit erkennen, dafür keine auch nur wenig veränderten Daten, oder sehr generalisiert sein, und abgeleitet von den Trainingsdaten alle auch nur entfernt ähnlichen, verrauschten oder fehlerhaften Daten einer Klasse zuordnen, auch wenn diese vielleicht aus menschlicher Sichtweise nichts mehr mit einer Klasse zu tun haben. Das richtige Verhältnis hängt oft vom Anwendungsfall ab und ist schwierig zu wählen.

3 Umsetzung

3.1 Praktische Aspekte und Herausforderungen

3.1.1 Input Encoding/Embedding

Da keine Wörter oder Zeichen, sondern nur einen Zahlenwert, als Input für ein Neuron gewählt werden kann, muss der Input für den Anwendungsfall dieser Arbeit vorher bearbeitet werden. Es wäre möglich, wenn der Wertebereich eines neuronalen Netzes 0 bis 1 ist, jedem vorkommenden Zeichen oder Wort einen eindeutigen Wert in diesem Wertebereich zuweisen und dies als Input verwenden. Zum Beispiel könnte man dem Buchstaben A den Wert 0.33, B den Wert 0.66 und C den Wert 1 geben. Dies hat sich allerdings als problematisch erwiesen, da aus Sicht eines Netzes dann beispielsweise A ähnlicher zu B als zu C ist, da ihre Werte knapp zusammenliegen, und C beispielsweise 3*A ist, was beides natürlich falsch ist und zu falschen Rückschlüssen führt. Stattdessen kommen häufig Verfahren wie One-Hot Encoding zum Einsatz, bei dem jedes vorkommende Zeichen einen fixen Platz in einem gemeinsamen One-Hot Vektor erhält. Der Input Layer muss dann ebenso viele Neuronen besitzen, wie dieser One-Hot Vektor lang ist. Je nachdem welches Zeichen oder Wort nun dem Netz präsentiert werden soll, werden alle Stellen in dem Vektor auf 0 gesetzt, nur diese eine Stelle auf 1. Abbildung 18 zeigt beispielsweise einen Vektor, der die Instruktion „CreateFile“ anzeigt. Mit diesen Vektoren lässt sich auch in gewohnter Weise Training durchführen, das Netz lernt diesen Vektor wie jede andere Repräsentation des Inputs. Der Aufbau Vektor darf aber während des ganzen Betriebs nicht verändert werden, es dürfen also keine Zeichen hinzugefügt oder entfernt werden, da das Netz nur mit diesem Vektor funktioniert. Alle für den Vektor notwendige Objekte müssen also bereits zu Trainingsbeginn feststehen.

Abbildung 18: One-Hot-Encoding Vektor, der gerade die Instruktion „CreateFile“ anzeigt

Für den Anwendungsfall dieser Arbeit ist es aber leider nicht möglich, direkt Instruktionen oder Bezeichnungen zu kodieren. Einerseits, da durch viele verschiedene Pfade, Benutzernamen, IP Adressen und ähnlichem riesige Vektoren von mehreren zehntausend oder sogar hunderttausend Zeichen entstehen würden. Wie in [54] erwähnt würde alleine ein Vektor mit üblichen Wörtern einer Sprache bereits circa 30.000 Stellen haben. Andererseits, da es nicht möglich ist, bereits zu Beginn eine vollständige Menge an möglichen Werten zu definieren, da viele Werte erst während des Betriebs auftauchen werden. Dadurch bleibt höchstens die Möglichkeit, jedes Zeichen einzeln zu kodieren, da hier die Menge eher klein ist und bereits vor Trainingsbeginn feststeht und sich nicht mehr ändern wird. Allerdings werden in dieser Arbeit die Schriftzeichen auf alle Buchstaben, Zahlen und Sonderzeichen auf einer Tastatur mit deutschem Layout beschränkt, denn würden beispielsweise alle Zeichen aus Unicode berücksichtigt werden, wäre der Input Vektor über 100.000 Zeichen lang. Dies bedeutet logischerweise, dass die Verarbeitung auf Events mit diesen Zeichen beschränkt ist, der Einsatz in einer Region mit anderen Schriftzeichen wäre nicht möglich. One-Hot-Encoding der Zeichen führt jedoch dazu, dass pro Zeitschritt nicht eine Instruktion, sondern nur ein Zeichen verarbeitet werden kann, was die Menge der notwendigen Zeitschritte und damit die Verarbeitungszeit für Training und Betrieb erhöht. Außerdem kennt das Netz dann im Gegensatz zu dem Input Vektor in Abbildung 18 keine Wörter oder Instruktionen, sondern muss diese erst aus den einzelnen Zeichen zusammensetzen, um Zeichenabfolgen zu unterscheiden oder typische Muster wie den Doppelpunkt in der Uhrzeit zu lernen. Die Trainingszeit ist dadurch um ein Vielfaches höher und es besteht die Gefahr, dass das Netz überhaupt nicht konvergiert. In diversen Arbeiten werden sehr homogene Daten verwendet, wie beispielsweise einen API Call pro Zeile. Dadurch kann man diese als kleinste Einheit festlegen und eine übersichtliche One-Hot-Encoding Tabelle für API Calls

machen, ähnlich wie in Abbildung 18, das Netz muss nicht erst aus einzelnen Zeichen zusammenhängende Wörter identifizieren. Bei der in dieser Arbeit verwendeten Art von Daten ist dies leider in dieser Form nicht möglich.

Ein weiteres mögliches Verfahren ist N-Gram. N-Gram wird für Natural Language Processing (NLP) eingesetzt und es werden je nach dem Wert N Gruppen aus zwei oder mehr nebeneinanderstehenden Worten gebildet, um bei der Verarbeitung den Kontext zu den zusammenhängenden Wörtern zu berücksichtigen. Durch die in dieser Arbeit verwendeten sehr unterschiedlichen Daten (data sparsity) würde aber bereits bei 2-Gram (Gruppen aus zwei Wörtern) kaum eine Gruppe doppelt vorkommen. Das Netz würde die meisten Gruppen im Training damit höchstens einmal sehen und im Betrieb die allermeisten Gruppen zum ersten Mal, dabei wird N-gram erst mit einer höheren Anzahl N effektiv, wie beispielsweise 4 oder 5. Abgesehen davon wäre die Mapping-Tabelle durch die große Menge an einzigartigen Gruppen sehr groß, was wiederrum zu großen Input Vektoren führt.

3.1.2 Daten unterschiedlicher Form

Ein mögliches Problem bei den für das neuronale Netzwerk verwendeten Daten könnte durch die unterschiedliche Form der Daten entstehen. So könnten die unterschiedlichen Trainingsdaten und Live-Daten aus verschiedenen Systemen oder verschiedenen Regionen stammen und damit einen anderen Aufbau besitzen oder beispielsweise Labels in unterschiedlichen Sprachen aufweisen. Dies stellt einerseits bei den Trainingsdaten ein Problem dar, denn wenn sich die Daten der zwei Kategorien bereits durch den Aufbau unterscheiden, wird das Netz die Klassifizierung aufgrund dieser Eigenschaft treffen und nicht aufgrund des Inhalts. Außerdem kann das Problem auch bei den Live-Daten auftreten, wenn diese ebenfalls eine andere Form haben als die verwendeten Trainingsdaten kann das Netz diese eventuell nicht klassifizieren. Daher ist es sehr wichtig, dass alle Trainingsdaten und alle Live-Daten dieselbe Form besitzen. Ein einfaches und sehr abstrahiertes Beispiel soll zeigen, wie unterschiedliche Daten aussehen könnten, die zu den oben beschriebenen Problemen führen:

- Gutartige Daten: „A program xyz was executed at 12:34:59 pm on 7/24/2018“
- Bösartige Daten: “Programm xyz wurde um 12:34:59 am 24.07.2018 ausgeführt “
- Laufende Daten: „不要惊慌12:34:59 在 24.07.2018执行 “

3.2 Evaluierung von Frameworks und Libraries

Da es eine Vielzahl an verschiedenen Frameworks und Libraries für neuronale Netz gibt und die Vor- und Nachteile nicht auf einen Blick greifbar sind, wird das folgende Kapitel einen groben Überblick schaffen um schlussendlich zu entscheiden, was für die praktischen Teile dieser Arbeit verwendet wird.

3.2.1 Anforderungen

Die Anforderungen an Frameworks und Libraries können hauptsächlich nur subjektiv formuliert werden. Wichtig ist unter anderem, dass das Framework einen professionellen und fertigen Zustand aufweist, es in einer fertigen Version vorliegt und der Anwender möglichst nicht mit Bugs und Problemen konfrontiert wird, da die Implementierung selbst ohnehin aufwendig ist. Um den Umgang mit dem Framework zu lernen und bei schwierigen Fragestellungen auf Hilfe zurückgreifen zu können, ist auch eine große Community und viele Tutorials wichtig. Für die ersten Implementierungen wird ein PC mit guter CPU aber schlechter GPU benutzt, während spätere Implementierungen auf einer NVIDIA Tesla Grafikkarte laufen sollen, weswegen sowohl CPU als auch GPU vollständig unterstützt werden müssen und ein Wechsel möglichst ohne Codeänderungen durchführbar sein soll. Unterstützung für mobile Geräte ist für Malware Detection sicher hilfreich, allerdings liegt dies nicht im Fokus dieser Arbeit und wird daher vernachlässigt. Auch sollte sich das verwendete Framework durch hohe Performance auszeichnen, da es in dieser Arbeit nicht darum geht in simplen Netzen das Thema

Neuronale Netze kennenzulernen, sondern das Ziel ist auch größere Datenmengen schlussendlich real-time verarbeiten zu können. Damit die Implementierung von längerem wissenschaftlichem Wert ist stellt eine Anforderung die Kontinuität der verwendeten Lösung dar, die Lösung soll auch die nächsten Jahre noch existieren, was sich möglicherweise durch Faktoren wie Community, mediale Aufmerksamkeit und unterstützende Firmen abschätzen lässt. Das Framework muss bereits gängige Algorithmen und Funktionen implementiert haben, damit nicht alle Funktionen selbst grundlegend implementieret werden müssen, was sich bei komplexeren Netzen und Verfahren wie RNNs und BPTT als sehr zeitaufwendig und fehleranfällig gestaltet. Implementiert sollen aktuelle Netze, Aktivierungsfunktionen, Error Funktionen, Optimierungsfunktionen und Zusatzfunktionen wie Regularization und Batch Normalization sein. Trotzdem sollte durch Framework und Library genügend Möglichkeit für individuelle Anpassung und Änderung der vorhandenen Funktionen und Modelle möglich sein, um das Netz genau auf die notwendigen Bedürfnisse anpassen zu können. Wichtig ist auch, dass State-of-the-art Konzepte und Netze unterstützt werden, da aufgrund der starken Forschung in dem Bereich in kurzer Zeit viele neue Erkenntnisse und Verfahren entdeckt werden. Zuletzt wäre es sehr hilfreich, wenn es Unterstützung für grafische Darstellung der Netze gibt, um Training und Validation besser überwachen zu können und die Arbeitsweise des Netzes gut verstehen zu können.

3.2.2 Analyse der Frameworks und Libraries für neuronale Netze

3.2.2.1 Tensorflow

Tensorflow ist ein Framework von Google, welches Unterstützung für CPU, GPU und Google TPU bietet. Zusätzlich unterstützt das Framework alle gängigen Architekturen und Betriebssysteme, von iOS, über Android, bis Raspberry Pi. Tensorflow unterstützt alle Features in Python, immer mehr Features werden aber auch in C, Java, R, Haskell und Go implementiert (Stand Februar 2018). Es existieren verschiedene APIs, von einer Low-Level API in der vieles selber implementiert werden muss bis zu einer High-Level API mit vortrainierten Netzen, wie in Abbildung 19 zu sehen ist. Das Framework bietet viele Möglichkeiten für out of the box Dashboards, Graphen und Statistiken, die einen guten Einblick in das Netz, den Verlauf von Trainings, die Verteilung von Gewichten und Ähnlichem geben. Bereits im Framework enthalten sind auch verschiedene Libraries zum Importieren und Normalisieren von Daten, wie numpy und Pandas. Gerade bei großen Anwendungen ist die standardmäßige Unterstützung für verteilte Berechnungen sehr hilfreich, mit der sich ohne viel Aufwand ein Netz auf mehrere Cluster/Kerne verteilen lässt, genannt Distributed Tensorflow. Sehr hilfreich sind auch die sogenannten Checkpoints, die ein trainiertes Modell mit allen relevanten Daten abspeichern und beim nächsten Mal eine Wiederaufnahme des Trainings erlauben. Mit Tensorflow Serving lassen sich auch Anwendungen für mehrere Endgeräte mit einem Backend-Server realisieren, durch den sichergestellt wird, dass nicht alle Endgeräte die teilweise aufwendigen Modelle und Daten laden müssen[58]. Tensorflow erlaubt es, Models zu Executables zu kompilieren, wodurch sich die Ausführungszeit drastisch verringert. Unterstützt wird auch Cross-Compilation zwischen verschiedenen Architekturen[59]. Dadurch, dass das Framework offiziell von Google entwickelt und weiterentwickelt wird, gibt es eine große Community und viele Tutorials, Talks und Sample Code, um sich in das Thema einarbeiten zu können. 2017 war Tensorflow laut Google das am besten bewertete Tool für Machine Learning auf GitHub[60]. Im Gegensatz zu verschiedenen anderen Tools unterstützt Tensorflow sämtliche Phasen, von der Entwicklung neuer Modelle, über Training und Deployment, bis hin zum produktiven Betrieb[58]. Tensorflow unterstützt mittlerweile offiziell Keras, wodurch frühere Performance-Probleme bei der Nutzung von Keras und Tensorflow bald behoben werden sein sollten. Mehr zu Keras in Kapitel 3.2.2.2.

Abbildung 19: Tensorflow Stack mit den verschiedenen APIs

3.2.2.2 Keras

Keras ist eine Library für Python, die auf verschiedene Frameworks wie Tensorflow und Theano aufsetzt und dient zur Entwicklung von eigenen Netzen. Wie in einem Block-System gibt es daher bereits Building Blocks für Schichten, Aktivierungsfunktionen, Optimierungsfunktionen, und vieles mehr. Um eine möglichst einfache Implementierung zu ermöglichen, bietet Keras teilweise auch vortrainierte Netze, was außerdem auch die Bearbeitung von kleineren Aufgaben ohne umfassendes Trainingsset ermöglicht. Durch die hohe Abstraktion von der eigentlichen Thematik erlaubt man damit nicht nur, schnell und ohne umfassendes Mathematik-Wissen Netze und komplexe Modelle zu bauen, sondern stellt auch sicher, dass mathematisch sinnvolle Vorgehen automatisch implementiert werden, sodass der User beispielsweise aus 4 sinnvollen Initialisierungsverfahren für ein Netz auswählen kann und nicht selber die Verteilung von Gewichten und Biases bestimmen muss. Laut vielen Meinungen in Foren ist die API von Keras sehr einfach und intuitiv zu verwenden und stellt damit eine bessere Schnittstelle zur Verfügung, als in vielen Frameworks vorhanden ist. Durch die offizielle Unterstützung von Google soll die Integration in Tensorflow verstärkt werden und im Gegensatz zum bisherigen Stand in Zukunft reibungslos funktionieren.

3.2.2.3 Caffe2

Caffe wurde ursprünglich von Berkeley AI Research und der Community entwickelt. Nach erfolgreichem Einsatz auf Facebook zur Klassifizierung von Bildern wurde Caffe2 entwickelt, welches nun offiziell von Facebook weiterentwickelt und unterstützt wird. Laut Website unterstützt Caffe2 ebenfalls CPU und GPU, für die Ausführung werden auch Cluster und Mobile Devices unterstützt[61]. Caffe2 zeichnet sich nach eigenen Angaben durch eine hohe Performance aus, so kann es pro Tag auf einer NVIDIA K40 GPU mehr als 60 Millionen Bilder verarbeiten. Das Framework ist in C++ umgesetzt, unterstützt C++ und Python und beinhaltet Tools, fertige Models, Demos und mehr. Es gibt einen Model Zoo, in dem Models von der Community geteilt werden können, unter anderem befinden sich dort erfolgreich bei Wettbewerben getestete Models, die aber vor allem auf visuelle Klassifikation ausgerichtet sind. Der Fokus liegt abgeleitet vom Inhalt einer offiziellen Präsentation vor allem auf visueller Klassifikation, aber auch auf Sprach und Text-Erkennung[62].

3.2.2.4 Theano

Theano liegt in der Version 1.0 vor, die am 15.11.2017 veröffentlicht wurde. Es ist eine Python Library, die es erlaubt mathematische Ausdrücke zu definieren, zu optimieren und zu evaluieren, auch für mehrdimensionale Arrays[63]. Damit ist es vorwiegend für grundlegende mathematische Berechnungen geeignet, also zum Beispiel zum Bauen von neuronalen Netzen ohne implementierte Bausteine, was gerade bei RNNs und LSTMs schwierig sein könnte. Die Library zeichnet sich durch eine hohe Geschwindigkeit bei der Berechnung mathematischer Ausdrücke aus und unterstützt GPU. Theano besitzt keine vordefinierten Funktionen oder Algorithmen, alle notwendigen Bausteine wie Aktivierungsfunktion, Optimierungsfunktion und ähnliches müssen

selber programmiert werden, was die Fehleranfälligkeit erhöht. Anscheinend unterstützt Theano noch keine verteilte Berechnung (Stand Februar 2018).

3.2.2.5 Torch oder PyTorch

PyTorch ist ein Framework für Python, das derzeit als frühe Beta-Version vorhanden ist (Stand Februar 2018). Dadurch, dass es keine fertige Version gibt und laut manchen Anwendern noch verschiedene Bugs enthalten sind, scheidet PyTorch aus der Evaluierung aus. PyTorch unterstützt Linux und OSx und wird laut Website von einigen Universitäten und Twitter verwendet[64].

3.2.2.6 MXNet

Apache MXNet ist ein Deep Learning Framework, am 4. Dezember 2017 wurde Version 1.0 veröffentlicht. Auf der Website befinden sich neben einer Fülle an Example Source Code und Tutorials auch ein Model Zoo, ähnlich wie bei Caffe. MXNet unterstützt APIs für Python, Scala, R, Julia, C++ und Perl (Stand Februar 2018) und stellt eine umfassende Dokumentation für jede der unterstützten Programmiersprachen bereit. Bis jetzt dürfte MXNet noch nicht gut in der Community verankert sein, da außerhalb der Seite kaum Tutorials, Anleitungen, Vorträge oder Projekte für MXNet vorhanden sind. MXNet beinhaltet bereits definierte Funktionen und Algorithmen, die man ähnlich wie Tensorflow verwenden kann.

3.2.2.7 Gluon

Gluon ist eine Library für Python die unabhängig vom Framework verwendet werden kann, um Models zu erstellen und zu trainieren. Gluon ist in MXNet bereits inkludiert und wird offiziell unterstützt[65]. Im Fall von MXNet dürfte Gluon beispielsweise für komplexere Netze wie LSTMs verwendet werden. Gluon wurde offiziell von Microsoft und Amazon in einer Kooperation veröffentlicht, der Support für Microsofts Cognitive Toolkit soll bald folgen[66]. Bis jetzt dürfte Gluon aber ausschließlich für MXNet verfügbar sein. (Stand Februar 2018)

3.2.2.8 Scikit-learn

Scikit-Learn ist eine Library für Python, im Oktober 2017 wurde die Version 0.19.1 veröffentlicht. Die Library ist vor allem für Einsteiger gedacht und soll erste Erfahrungen mit Deep Learning und neuronalen Netzen vereinfachen. Scikit-Learn hat bereits verschiedene Algorithmen und Funktionen implementiert und unterstützt auch grundlegende graphische Aufbereitung. Der Umfang der Library wirkt im Vergleich zu anderen eher gering, so dürften Modelle für RNNs nicht enthalten sein, was auch damit gerechtfertigt ist, dass von dem Programm noch keine volle Version (>= v1.0) veröffentlicht wurde (Stand Februar 2018).

3.2.2.9 Microsoft Cognitive Toolkit (CNTK)

CNTK liegt aktuell in der Version 2.4 vor und kann als Stand Alone Toolkit verwendet werden oder als Library in Python, C# oder C++ importiert werden. Es sind bereits verschiedene Funktionen für Netze und Algorithmen wie Optimierungsverfahren implementiert und damit können aus einfachen Building Blocks komplexe Netzwerke zusammengesetzt werden. Als großer Vorteil zu Tensorflow kann CNTK relativ einfach auf Windows installiert werden, aber auch auf Linux. Standardmäßig werden Multi-GPU beziehungsweise Multi-Server Berechnungen unterstützt. Microsoft setzt selbst CNTK in zahlreichen Produkten wie Cortana, Bing oder Skype ein. CNTK unterstützt als eines der wenigen Frameworks das Open Neural Network Exchange (ONNX) Format. Anhand der Beispiele in einem YouTube-Video von Microsoft, wofür CNTK eingesetzt wird, zeigt sich, dass der Fokus hauptsächlich auf Bilderkennung und Textbearbeitung liegt.[67].

3.2.2.10 Deeplearning4j (DL4J)

DL4J ist eine Library für Java und Scala und bietet neben MapReduce Funktionalität für verteilte Berechnungen auch weiteren Tools. DL4J unterstützt alle modernen Netz-Typen, von RNNs über CNNs und beinhaltet eine spezielle Library für Vektorisieren, namens Canova. Die Seite bietet Informationen und Guides, auch allgemein zu neuronalen Netzen.

3.2.2.11 Passage Library

Passage ist eine Library in Python für Theano und ist auf Textbearbeitung mithilfe von RNNs spezialisiert, was für die Kodierung von Systemevents oder Logzeilen hilfreich ist. Laut GitHub Repository gab es allerdings seit 2015 keine Updates mehr, wodurch das Produkt ausscheidet.

3.2.3 Fazit

Aufgrund der umfassenden Funktionalität fällt die Wahl für das Framework auf Tensorflow. Es überzeugt durch viele Vorteile wie automatische verteilte Berechnungen, große Community und hat mit Google den möglicherweise besten Entwickler und Unterstützer. Weiter zeichnet sich Tensorflow durch Unterstützung von fast allen Architekturen und Systemen aus. Auch die graphischen Funktionalitäten wie Dashboards, Grafiken und Statistiken helfen, neuronale Netze beziehungsweise das eigene modellierte neuronale Netz besser zu verstehen, den Trainingserfolg mitzuverfolgen oder Probleme zu identifizieren, da die Systeme durch ihre hohe Komplexität anders schwer überschaubar und begreifbar sind. Durch die laufende und schnelle Entwicklung beinhaltet Tensorflow außerdem alle State-of-the-art Funktionalitäten und Netze. Aufgrund der erhöhten Flexibilität und der notwendigen Möglichkeit für Individualisierung wird außerdem die Library Keras verwendet werden, um eigene Netze oder Strukturen zu entwerfen, die standardmäßig in Tensorflow geladen werden können. Durch die offizielle Unterstützung von Keras durch Tensorflow sollten alle Features gut implementiert und Fehler und Probleme kaum vorhanden sein. Da Tensorflow nur die zwei Programmiersprachen Python und C im Moment (Stand Februar 2018) unterstützt, fällt die Wahl eindeutig auf Python. Nicht nur, dass Python mehr Komfort bietet, auch dürfte es aufgrund der vielen zusätzlich verfügbaren Libraries weitaus geeigneter sein. Hinzuzufügen ist, dass die meisten Frameworks und Libraries viele Funktionen haben und gut funktionieren, der Unterschied ist teilweise sehr gering. Es kommt daher vor allem auf den Fokus aber auch auf persönliche Vorliebe an, ein rein objektiver Vergleich ist unmöglich.

3.3 Evaluierung eines passenden Netzes

Ziel dieses Kapitels ist es, aus der Vielzahl an vorhandenen Arten und Typen von neuronalen Netzen das für diese Arbeit passendste Netz auszuwählen, beziehungsweise mehrere passende Netze zu finden. Dafür wird das folgende Kapitel zuerst einen kurzen Überblick über die Einsatzgebiete und Vor- sowie Nachteile der verschiedenen Netze schaffen und abschließend unter Berücksichtigung der Problemstellung dieser Arbeit ein geeignetes Netz ausgewählt, oder zumindest ungeeignete Netze ausgeschlossen. Das Kapitel wird allerdings nur auf ein paar der in den Grundlagen beschriebenen Netze eingehen, da manche Netze aufgrund von ähnlicher Funktionalität oder fehlender Literatur im Vorhinein ausgeschlossen werden können.

3.3.1 Anforderung

Die Anforderungen an die gesuchten Netze können nicht spezifisch definiert werden, da der endgültige Aufbau des Netzes oder der Netze noch nicht feststeht. Zur Lösung des Problems wird aber auf jeden Fall ein Netz notwendig sein, dass sich eine Vielzahl an verschiedenen Mustern sehr gut merkt, um Muster von Malware und von normalen Vorgängen abspeichern zu können. Außerdem wird ein Netz benötigt werden, welches mehrere Logzeilen in Verbindung miteinander setzen kann, also eine Beziehung zwischen mehreren Zyklen herstellen kann, da immer eine unterschiedlich große Zahl an Events verkettet und verarbeitet werden muss.

3.3.2 Mögliche relevante Netze

Feed-forward Neural Networks

Feed-forward Neural Networks eignen sich gut als Classifier. Sie können relativ leicht und schnell trainiert werden, Implementierung und Training sind unkompliziert. Dadurch würde sich das Netz gut als Classifier

eignen. Auch in der Literatur wird durch die nahezu uneingeschränkte Möglichkeit meist ein FFNN als Classifier genutzt, so kann damit von Text bis Bild alles klassifiziert werden.

Deep Belief Network

Das DBN lässt sich angeblich leichter und präziser trainieren als ein normales Feed-forward Neural Network, was am unsupervised Pre-Training liegen soll. Da es sonst in der Funktionalität dem Netz aber sehr stark ähnelt beziehungsweise ident mit diesem ist, bietet ein DBN theoretisch bessere Eigenschaften. Im Gegensatz zu einem FFNN ist es allerdings nicht so weit verbreitet und wird von vielen Frameworks nicht unterstützt.

Convolutional Neural Network

CNNs eignen sich vor allem gut zur Erkennung von optischen Mustern und erzielen dort mitunter bessere Ergebnisse als Menschen. Auch wenn sie bereits zur Erkennung von künstlichen Mustern, beispielsweise in Dateien, benutzt wurden, waren dort die Ergebnisse meist schlechter als die Ergebnisse anderer Netze. CNNs können auch nicht mehrere Input-Zyklen miteinander in Beziehung setzen. Da die zu analysierenden Daten kaum regelmäßige Muster aufweisen werden und vor allem der zeitliche Zusammenhang wichtig ist, wird ein CNN daher nicht die gewünschten Ergebnisse liefern können.

Recurrent Neural Network

RNNs eignen sich gut zur Analyse von Daten mit einem Verlauf beziehungsweise mit einem Zusammenhang zwischen verschiedenen Datensätzen, da sie mehrere Input-Zyklen in Verbindung bringen können. So eignen sie sich beispielsweise zur Analyse von Daten mit einer zeitlichen Dimension und heben sich dadurch von den meisten anderen Netzen ab. RNNs sind aufwendiger zu trainieren als Feed-forward Neural Networks, Probleme wie vanishing gradient oder exploding gradient können auftreten.

Long Short-Term Memory

LSTMs entsprechen im Grunde den Eigenschaften eines RNNs, nur können sie durch ihre Speicherzellen Information gezielter speichern und wieder abrufen. Dadurch lassen sich einzelne Informationen präzise über eine hohe Anzahl an Schritten speichern und dann im richtigen Moment wieder abrufen. Die Menge der speicherbaren Information ist jedoch sehr gering und es bedarf langem Training und eher einheitlichen Sequenzen, damit das Netz die richtige Information speichert und für die benötigte Dauer nicht überschreibt [39]. Da es beim Anwendungsfall in dieser Arbeit aber nicht um die gezielte Speicherung von einzelnen Informationen geht, beziehungsweise wenn doch liegt die Information an sehr unterschiedlichen Stellen, bietet das LSTM keinen Vorteil gegenüber einem einfachen RNN, eher sogar Nachteile. Zum einen erhöht ein LSTM die Komplexität im Design und im Training, zum anderen speichert ein LSTM trotzdem einzelne Informationen, nur wären dies in dieser Arbeit vermutlich unwichtige Informationen, die das Ergebnis unverhältnismäßig beeinflussen.

Adaptive Resonance Theory

ART Netze eignen sich sehr gut zum Lernen von einer Vielzahl von Mustern, da sie bereits gelerntes Wissen nicht vergessen und trotzdem neue Muster erlernen können. Lediglich wird vorhandenes Wissen in Kategorien zusammengefasst. Dadurch würden sie sich gut für ein Netz zur Speicherung von Malware Mustern eigenen, da dafür eine sehr hohe Zahl an Mustern notwendig ist. Leider ist dieses Netz sowohl in der Literatur, als auch in Tutorials, Vorträgen, wissenschaftlichen Arbeiten und Frameworks unterrepräsentiert oder meist nicht vertreten. Eine praktische Implementierung wird dadurch nahezu unmöglich und wäre wenn dann vom Umfang ausreichend für eine eigenständige Arbeit.

Word2Vec

Das Netz eignet sich sehr gut für die Umwandlung von Wörtern in einen Vektor in einem Vektor Raum, in Bezug gebracht mit ähnlichen bereits analysierten Wörtern. Damit eignet sich das Verfahren sehr gut für Natural Language Processing (NLP), da beispielsweise die Zusammenhänge von Wien – Österreich und Madrid – Spanien gelernt werden können. Da es im Anwendungsfall aber kaum um den Zusammenhang der einzelnen

Wörter und Zeichen gut, sowie die Menge der einzigartigen Wörter und Buchstabenkombinationen sehr umfangreich ist, wird ein Word2Vec keine sinnvollen Ergebnisse liefern können.

3.3.3 Fazit

Durch die komplexe Aufgabenstellung stellt sich heraus, dass sich das Problem keinesfalls mit einem einzigen Netz oder mit einem einzigen Netz-Typ lösen lässt. Stattdessen wird eine Kombination aus mehreren Netzen mit verschiedenen Eigenschaften notwendig sein. RNNs haben sich bereits in der Analyse von Log Dateien und sämtlichen Daten mit Zeitablauf bewährt. Sie bieten die unerlässliche Eigenschaft auch über viele Schritte Informationen in Zusammenhang setzen zu können, was sich vortrefflich eignet um mehrere zeitlich gereihte Events in einen einzigen repräsentativen Output zu verarbeiten, der dann weiterverarbeitet werden kann. Explizit wird voraussichtlich eine Architektur ähnlich eines SRNs genutzt werden. Die endgültige Beurteilung soll durch einen Classifier passieren, wozu ein FFNN oder ein DBN geeignet ist. Das DBN besitzt ähnliche Eigenschaften wie ein Feed-forward Neural Network, das sich bereits in vielen Arbeiten als Classifier bewährt hat, führt aber durch den besonderen Trainingsansatz zu angeblich besseren und schnelleren Ergebnissen. Aufgrund der häufigen Verwendung und der vielen Anwendungsfälle wird im ersten Schritt voraussichtlich ein FFNN verwendet werden.

3.4 Praktische Einarbeitung in Tensorflow

3.4.1 Einstieg und Überblick

Da der Autor dieser Arbeit noch keines dieser Frameworks bisher verwendet hat, bedarf es einiger Zeit zur Einarbeitung. Dafür werden einerseits die Tutorials auf der offiziellen Tensorflow Seite genutzt, andererseits beliebige Tutorials im Internet, da ohnehin an der fehlerfreien Ausführung und am Ergebnis erkannt werden kann, ob die im Tutorial vermittelten Inhalte korrekt sind. Unter anderem wird die beliebte Klassifikation von Ziffern mithilfe der MNIST Datenbank durch verschiedene Verfahren durchgeführt. Einerseits wird die Klassifikation mit einem CNN durchgeführt, andererseits mit einem einfachen Feed-forward Neural Network. Zum besseren Verständnis des Frameworks wird die MNIST-Klassifikation mithilfe der Tensorflow Low-Level API, mit Tensorflow premade Estimators, Tensorflow custom Estimators, sowie mit Keras im sequenziellen und im funktionellen Modus umgesetzt. Bei der Erlernung des Frameworks soll der Fokus vor allem auf Feed-Forward Neural Networks und DBNs als Classifier und auf RNNs liegen, da diese Typen voraussichtlich später in der Arbeit benötigt werden. Abgesehen davon wird versucht, verschiedene Parameter, vor allem Hyperparameter mithilfe von TensorBoard grafisch aufzubereiten. TensorBoard ist sehr wertvoll für die Analyse des gebauten Netzes und des Trainings- und Betriebsverlaufs. So kann man beispielsweise für alle Layers eine Vielzahl an Graphen wie die durchschnittliche Gewichtung pro Schritt anzeigen lassen (Abbildung 20), die Genauigkeit bei Training und Test (Abbildung 21) oder den Aufbau des Netzes.

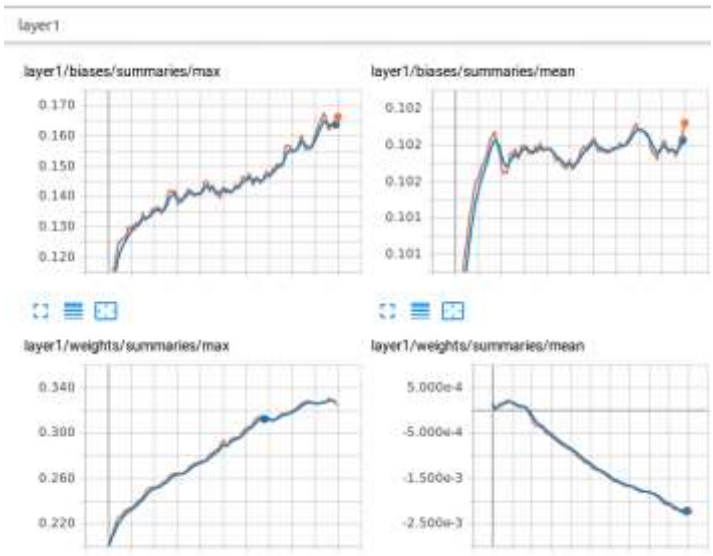

Abbildung 20: Verschiedene Graphen für die erste Layer eines Netzes zur MNIST-Klassifikation

Bei der Verwendung von Tensorflow stellt sich jedoch heraus, dass die Vielfalt und Komplexität zugleich auch ein Problem darstellen. So lässt sich beispielsweise ein Feed-forward Neural Network mit Pre-Made Estimators, mit custom Estimators, mithilfe der Low-Level API, einer Mischung aus den beiden oder mithilfe von Keras bauen. Jeder Ansatz erfordert dabei andere Bausteine, Elemente und Befehle. Zusätzlich können zwar einfache Dinge wie Feed-forward Neural Networks mit all diesen Ansätzen gebaut werden, wenn man aber beispielsweise ein TensorBoard nutzen will, dürfte dies anscheinend nicht mehr mit allen Ansätzen funktionieren. Ebenso können komplexere Netze wie RNNs oder spezielle Feed-forward Neural Networks nicht mit allen Ansätzen umgesetzt werden. So schwankt auch die Codelänge für eine Aufgabe (MNIST Klassifizierung) zwischen 40 Zeilen bei Pre-Made Estimators und 270 Zeilen bei custom Estimators mit TensorBoard. Diese unterschiedlichen Herangehensweisen haben auch das Problem, dass oft unterschiedliche Befehle zur Aufbereitung von Daten oder zur Initialisierung verschiedener Werte benötigt werden, auch wenn es offensichtlich keinen Grund dafür gibt, was dem Framework von verschiedenen Anwendern auch mitunter Kritik einbringt. So müssen beispielsweise für Pre-Made Estimators und custom Estimators die Daten unterschiedlich vorbereiten werden, obwohl beide auf dieselbe API aufbauen. Der benötigte Aufbau ist nicht sehr intuitiv und die Tutorials teilweise schlecht gewartet. So führen Code-Teile des Tutorials zu Warnungen, dass die verwendeten Befehle veraltet (engl. „deprecated“) sind.

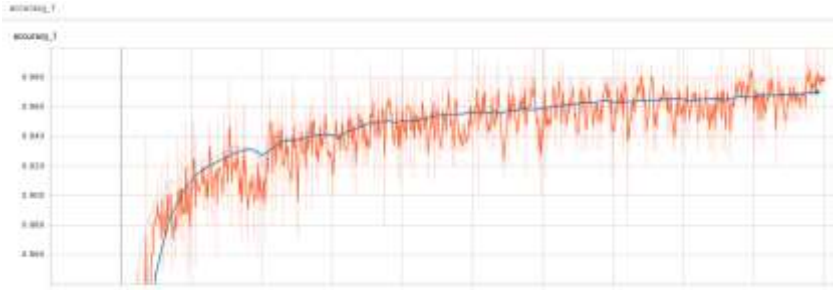

Abbildung 21: Der Verlauf der Genauigkeit eines Netzes zur MNIST-Klassifikation beim Training (orange) und beim Test (blau)

Ein großer Vorteil von Tensorflow sind allerdings die Checkpoints, die automatisch oder manuell angelegt werden können. Jeder Checkpoint enthält dabei

- das Modell,

- alle Gewichtungen im Modell,
- die Trainingskonfiguration mit Kostenfunktion, Optimierungsverfahren und ähnlichen Metadaten,
- den Fortschritt des Trainings, wie absolvierte Schritte und Epochen

Dadurch kann das Training nahtlos nach gewollter oder auch ungewollter Beendigung des Programms wieder aufgenommen werden oder Netze über mehrere Male hinweg weitertrainiert werden. Außerdem wird dadurch erst ermöglicht, dass das Netz zuerst trainiert wird und dann mit beliebigem Abstand danach wieder aus den Checkpoints geladen und verwendet wird. Abhängig von der Trainingsgeschwindigkeit können Checkpoints beliebig gesetzt werden, beispielsweise alle 100 Schritte, nach jeder Epoche oder ähnliches. Die Checkpoints werden auch als Rücksprungpunkt verwendet. Denn sollte beispielsweise nach einer Epoche die Trainings-Erkennungsrate nicht wesentlich besser sein, aber die Test-Erkennungsrate sinken, was klassisch für Overfitting spricht, kann der Checkpoint davor als Trainingsergebnis verwendet und das Training beendet werden. Mit der Verwendung von Keras als API und Tensorflow als Backend wird das weiter oben im Kapitel beschriebene Chaos zwar besser, die API ist strukturierter und intuitiver. Während man in Tensorflow um die Gewichtungen eines Layer zu erhalten anscheinend mit mehreren Befehlen Daten aus dem Speicher laden muss und in einer eigenen Session ausführen muss, das genaue Vorgehen konnte trotz einer Frage auf Stackexchange.com nicht eruiert werden, wird dafür bei Keras eine Zeile benötigt. Allerdings existieren noch Probleme beim Zusammenspiel von Keras und Tensorflow, wie bereits in der Evaluierung des Frameworks beschrieben wurde. Obwohl Keras mittlerweile offiziell von Tensorflow unterstützt wird und in Tensorflow inkludiert ist, führen manche Befehle zu Problemen. So bekommt man bei der Ausführung eines einfachen Feed-forward Neural Networks mit Keras drei „Deprecated“-Warnungen für Funktionen, die außerhalb der Reichweite des Users liegen. Sehr angenehm ist bei Keras im Vergleich zu Tensorflow, dass keine Session oder Interactive Session benötigt wird um Daten zu verarbeiten, da Daten nicht in Tensoren, sondern in Listen und Arrays gespeichert sind. Dies ermöglicht auch, an jeder beliebigen Stelle in den Code zu springen und Daten auszuwerten, ohne sich darum kümmern zu müssen, ob dies zu Problemen mit der Session führt und außerdem die Werte vom Tensor extra evaluiert werden müssen.

3.4.2 Kompilierung für CPU

Tensorflow unterstützt standardmäßig nicht alle Instruktionen von modernen CPUs, mit der Begründung mit möglichst vielen Modellen kompatibel sein zu wollen. Daher wird beim Ausführen eines Scripts immer die Meldung angezeigt, dass diese Version von Tensorflow für verschiedene verfügbare Instruktionen nicht ausgelegt ist und neu kompiliert werden müsste. Da diese Instruktionen angeblich die Performance verbessern und auf dem verwendeten Computer keine Grafikkarte zur Verfügung steht, wurde der Versuch unternommen Tensorflow für alle verfügbaren CPU Instruktionen zu kompilieren, anhand einer Anleitung von Tensorflow [68]. Im Folgenden wird nochmals kurz beschrieben, wie Tensorflow in Ubuntu 16.04 LTS mit Python3.5 kompiliert wurde. User Input ist zur besseren Übersicht in fettgedruckter Schrift abgebildet.

Dafür wurde zuerst das GitHub Repository heruntergeladen:

```
$git clone https://github.com/tensorflow/tensorflow
```

Danach in den Branch der aktuellsten Version von Tensorflow gewechselt, in diesem Fall r1.7:

```
$ cd tensorflow
$ git checkout r1.7
```

Mit folgenden Befehlen wurde zuerst Java JDK und danach Bazel installiert, da damit Tensorflow kompiliert wird:

```
$ sudo apt-get install openjdk-8-jdk
$ echo "deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable jdk1.8" |
sudo tee /etc/apt/sources.list.d/bazel.list
```

```
$ curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
$ sudo apt-get update && sudo apt-get install bazel
$ sudo apt-get upgrade bazel
```

Alle notwendigen Python Packages wurden installiert:

```
$ sudo apt-get install python3-numpy python3-dev python3-pip python3-wheel
```

Über den folgenden Befehl konnten viele Einstellungen für die Version von Tensorflow festgelegt werden, auch wenn für manche Features keine Erklärungen oder Beschreibungen gefunden werden konnte:

```
$ ./configure
Extracting Bazel installation...
You have bazel 0.11.1 installed.
Please specify the location of python. [Default is /usr/bin/python]:
/usr/bin/python3.5
Found possible Python library paths:
  /usr/local/lib/python3.5/dist-packages
  /usr/lib/python3/dist-packages
Please input the desired Python library path to use.  Default is
[/usr/local/lib/python3.5/dist-packages]
/usr/local/lib/python3.5/dist-packages
Do you wish to build TensorFlow with jemalloc as malloc support? [Y/n]:
y
jemalloc as malloc support will be enabled for TensorFlow.
Do you wish to build TensorFlow with Google Cloud Platform support? [Y/n]:
n
No Google Cloud Platform support will be enabled for TensorFlow.
Do you wish to build TensorFlow with Hadoop File System support? [Y/n]:
n
No Hadoop File System support will be enabled for TensorFlow.
Do you wish to build TensorFlow with Amazon S3 File System support? [Y/n]:
n
No Amazon S3 File System support will be enabled for TensorFlow.
Do you wish to build TensorFlow with Apache Kafka Platform support? [y/N]:
n
No Apache Kafka Platform support will be enabled for TensorFlow.
Do you wish to build TensorFlow with XLA JIT support? [y/N]:
n
No XLA JIT support will be enabled for TensorFlow.
Do you wish to build TensorFlow with GDR support? [y/N]:
n
No GDR support will be enabled for TensorFlow.
Do you wish to build TensorFlow with VERBS support? [y/N]:
n
No VERBS support will be enabled for TensorFlow.
Do you wish to build TensorFlow with OpenCL SYCL support? [y/N]:
n
No OpenCL SYCL support will be enabled for TensorFlow.
Do you wish to build TensorFlow with CUDA support? [y/N]:
n
No CUDA support will be enabled for TensorFlow.
Do you wish to build TensorFlow with MPI support? [y/N]:
n
No MPI support will be enabled for TensorFlow.
Please specify optimization flags to use during compilation when bazel option "--
config=opt" is specified [Default is -march=native]:
-march=native
Would you like to interactively configure ./WORKSPACE for Android builds? [y/N]:
```

```
n
Not configuring the WORKSPACE for Android builds.
Preconfigured Bazel build configs. You can use any of the below by adding "--
config=<>" to your build command. See tools/bazel.rc for more details.
        --config=mkl            # Build with MKL support.
        --config=monolithic     # Config for mostly static monolithic build.
Configuration finished
```

Es hätten verschiedene Optimization Flags angegeben werden können, aber laut Tensorflow reicht der Standardwert *-march=native* um die Version automatisch an die vorhandene CPU anzupassen. Andere Werte sind notwendig, wenn das Programm beispielsweise auf einem anderen System laufen soll als es kompiliert wird. Außerdem kann je nach Wunsch die Unterstützung für Plattformen wie Google Cloud aktiviert werden und je nach Verfügbarkeit CUDA oder OpenCL verwendet werden.

Mit dem folgenden Befehl wurde mithilfe von Bazel ein Pip Package für Python erstellt. Dieser Befehl benötigte über eine Stunde und erfordert viel RAM, was bei wenig vorhandenem RAM durch Übergabe eines Parameters optimiert werden könnte.

```
$ bazel build --config=opt //tensorflow/tools/pip_package:build_pip_package
```

Aus der entstehenden Datei build_pip_package wurde mit folgendem Befehl eine .whl Datei erstellt:

```
$ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
```

Abschließend konnte das Pip Paket installiert werden:

```
$ sudo pip install /tmp/tensorflow_pkg/tensorflow-1.7.0-cp35-cp35m-
linux_x86_64.whl
```

Zuletzt sollte noch die erfolgreiche Installation überprüft werden. Einerseits durch Verwendung von Tensorflow in Python, andererseits über die Pip Info:

```
$ python
Python 3.5.2 (default, Apr 23 2018, 16:37:01)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, Tensorflow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
b'Hello, Tensorflow!'
>>> exit()
$ pip show tensorflow
Name: tensorflow
Version: 1.7.0
Summary: TensorFlow helps the tensors flow
Home-page: https://www.tensorflow.org/
Author: Google Inc.
Author-email: opensource@google.com
License: Apache 2.0
Location: /home/stefan/tensorflow_env/lib/python3.5/site-packages
Requires: tensorboard, astor, gast, wheel, numpy, absl-py, termcolor, six,
grpcio, protobuf
Required-by:
```

Danach sollte Tensorflow mit allen vorhandenen CPU Instruktionen verfügbar sein und schneller laufen als zuvor. In diesem Fall verlief der Vorgang problemlos, Tensorflow war tatsächlich danach merkbar schneller. Leider wurden durch die neue Tensorflow Version verschiedene Änderungen notwendig, so musste im Unterschied zu vorherigen Versionen jeder Input ein Numpy Array sein. Laut mehreren Foren kann die

Kompilierung und Installation jedoch zu einer Vielzahl von Problemen führen, spezielle Anforderungen und Versuche zur Optimierung sind sehr schwierig.

3.5 Erster Ansatz

Dieser Ansatz wird nur theoretisch verfolgt, da er lediglich der versuchsweisen Anwendung des Wissens über neuronale Netze dient und daraus hoffentlich für die weitere Arbeit Schlüsse gezogen und Verbesserungen identifiziert werden können, die dazu führen, ein möglichst gut funktionierendes Netz theoretisch entwerfen und praktisch umsetzen zu können. Als Auslöser wird in dieser Arbeit das Programm oder der Vorgang bezeichnet, der eine Kette von Events nach sich zieht. Im Wesentlichen besteht das Ziel also darin festzustellen, ob ein Auslöser gutartig oder bösartig ist. Im ersten Ansatz werden aufgrund der bisherigen Erkenntnisse zwei Classifier verwendet, jeweils für die Typen Malware und Normal. Das RNN bekommt eine Log-Zeile nach der anderen, verknüpft diese und produziert nachdem für einen Auslöser alle Log-Zeilen verarbeitet wurden einen Output, der von den Classifiern weiterverarbeitet werden kann. Die Classifier verarbeiten nun beide denselben Input in ein einziges Output-Neuron. Das Output-Neuron jenes Classifiers, das am aktivsten ist, wird als Ergebnis gewertet. Das Netz sollte daher in der Lage sein zu entscheiden, ob ein Input auf Malware hindeutet oder ob es Normalverhalten darstellt. Dieses Netz ist jedoch nicht in der Lage nähere Klassifizierungen wie Angriffstyp oder Malwareart zu treffen. Da der Input immer kodiert werden muss, wird für diesen Ansatz eine Log-Zeile Zeichen für Zeichen mit One-Hot-Encoding kodiert und verarbeitet. Der Eingangsvektor ist daher circa 70 Zeichen groß, wie in Abbildung 22 zu sehen. Der Vektor umfasst 26 Buchstaben, 10 Ziffern und 34 Zeichen. Des Weiteren wird nicht zwischen Groß und Kleinbuchstaben unterschieden, sondern alle Zeichen als Kleinbuchstaben verarbeitet. Dieses Encoding-Verfahren verlängert zwar die Zeit bis das Netzwerk konvergiert, allerdings ist bei der Art von Daten kein anderes Vorgehen möglich. Für weitere Überlegungen, siehe Kapitel 3.1.1.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t

Index	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Zeichen	u	v	w	x	y	z	0	1	2	3	4	5	6	7	8	9	^	°	!	„

Index	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Zeichen	§	$	%	&	/	(	)	=	?	´	*	,	-	:	;	<	>	,	'	#

Index	60	61	62	63	64	65	66	67	68	69	70	71	72	73
Zeichen	+	`	|	{	}	[	]	\	~	_	<blank>	ö	ä	Ü

Abbildung 22: Eine Tabelle für One-Hot Encoding für den ersten Ansatz

Das RNN produziert über mehrere Input-Zyklen aus mehreren Events einen Output, der damit quasi alle als Input gegebenen Events in zeitlicher Abhängigkeit repräsentiert. Durch One-Hot-Encoding wird dabei pro Zyklus ein Buchstabe verarbeitet, was zwar rechenaufwendiger ist aber ein schlankeres Netz erlaubt und außerdem Padding überflüssig macht, da der Input immer die Größe des One-Hot-Vektors aufweist. Dieser Output dient als Input für den Classifier, der darauf trainiert wurde, anhand eines Inputs zu entscheiden ob es sich um Malware handelt. Daher ist es absolut entscheidend, dass nie die Architektur, die Gewichtungen oder Bias-Units des RNNs verändert werden, da sich sonst die „Signatur“ der Daten verändern würde und der Classifier nichts mehr richtig klassifizieren könnte. Eventuell ist es daher nicht notwendig das RNN zu trainieren, sondern nur den Classifier, da der Output des RNNs prinzipiell beliebig aussehen darf, solange er konsistent ist. Die Idee hinter diesem System ist, dass bedingt durch das Training der Normal Classifier normale Aktivitäten sicherer erkennt, als der Malware Classifier sie als Virus identifiziert, und umgekehrt für bösartige Aktivitäten. Dies sollte relativ leicht möglich sein, da ansonsten eines der beiden Netze nicht richtig funktionieren würde. Daher wird auch, ungewöhnlicherweise, jedes Netz nur mit einer Art von Input trainiert. Der Normal Classifier wird nur auf normales Verhalten trainiert, der Malware Classifier wird nur auf Malware trainiert. Dieser Ansatz soll unter

anderem gegen das Problem der hohen und nicht steuerbaren False-Positive Rate helfen. Diese wird einerseits dadurch hervorgerufen, dass ein neuronales Netz immer einen Output erzeugt, auch wenn es einen Input nicht korrekt klassifizieren kann, da es dafür aufgrund von falscher Trainingsverteilung oder geringer Menge an Trainingsdaten nicht trainiert wurde. Siehe Kapitel 2.6.2, Abschnitt „Immer eine Antwort“. Anstatt einfach ausgedrückt anzuzeigen „Diese Events habe ich noch nie gesehen, ich habe keine Ahnung was das ist“ wird es einen Output generieren, auch wenn dieser falsch ist. Das zweite Problem ist die gewünschte Generalisierung von Netzen, die dazu führen können, dass ein Netz bei dem Versuch zu generalisieren mit vollster „Überzeugung“ einen gutartigen Auslöser als Malware markiert. Siehe Kapitel 2.6.2, „Generalisierung-Spezialisierung-Dilemma“. Dabei stellen gerade False-Positives eine hohe vom System selbst ausgehende Bedrohung für die IT dar, da nur eine einzige fehlerhaft als Malware identifizierte und entfernte Datei ganze Systeme oder sogar Netzwerke zusammenbrechen lassen kann. Eine gute Möglichkeit wäre daher, die False-Positive-Rate gezielt steuern zu können und zugunsten einer niedrigeren False-Positive-Rate eine niedrigere Erkennungsrate wählen zu können. Dies wird durch den Ansatz mit zwei Netzen, die jeweils nur auf Malware oder auf Normalzustand trainiert sind, ermöglicht, da durch einen flexibel wählbaren Faktor das Ergebnis der repräsentativen Neuronen gewichtet werden kann, vergleichbar mit einer Bias Unit. Als Ausgangszustand hat jeder Classifier den Faktor 1, will man beispielsweise eine niedrigere False-Positive-Rate kann der Faktor für den Normal Classifier zum Beispiel auf 1,25 erhöht werden. Dadurch wird ein Auslöser häufiger als Normal eingestuft, da der Normal Classifier häufiger einen höheren Wert erzielt, was wie oben erwähnt im Gegenzug natürlich auch zu einer niedrigeren Erkennungsrate führt.

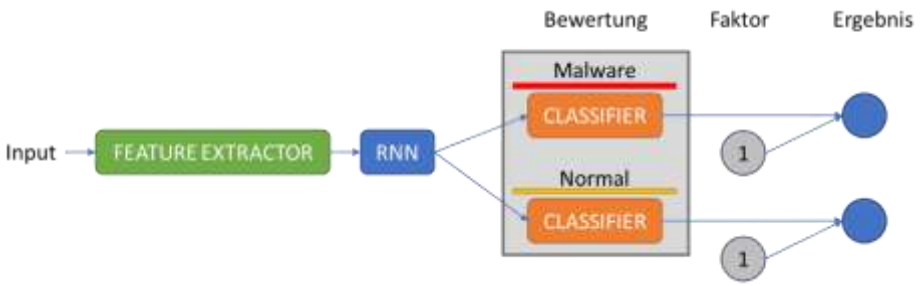

Abbildung 23: Aufbau des Systems für den ersten Ansatz

3.5.1 Aufbau

Das System besteht aus einem Feature Extractor für die Vorverarbeitung, der den großen Input Vektor für ein Event auf die wesentlichen Merkmale und damit einen kleineren Vektor reduziert. Danach kommt ein RNN das nacheinander alle vorverarbeiteten Events als Input bekommt und diese in Zusammenhang setzt und als einen Output Vektor ausgibt. Dieser Output Vektor repräsentiert damit alle Events in genau dieser Reihenfolge und ist der Input für einen Classifier, der je nach Classifier entscheidet ob es sich um Malware beziehungsweise einen Normalfall handelt. Theoretisch können die beiden Classifier unterschiedlich aufgebaut werden, vor allem in Hinblick auf Hyperparameter wie Neuronen Anzahl und Layer, allerdings werden hier beide Classifier ident beschrieben, da sich sinnvolle Unterschiede erst bei der Implementierung und im Zuge erster Tests ergeben werden.

3.5.1.1 Input

Der Input besteht aus Events in zeitlicher Abfolge die einem Auslöser zugeordnet sind und beinhalten oft Pfade und weitere Informationen, wie IP-Adressen, Nutzernamen und Programmnamen. Wie weiter oben beschrieben wird der Input durch One-Hot-Encoding vorverarbeitet werden müssen, dadurch wird sich ein Wertebereich von [0,1] ergeben. Im Gegensatz zu [6] wird hier der Input nicht auf einen Mittelwert von 0 und eine Varianz von 1 normalisiert. Dies verschnellert zwar das Training, führt aber durchaus zu schlechteren Ergebnissen, wie [3]

zeigt, und könnte speziell beim Einsatz von One-Hot Encoding das Ergebnis verschlechtern, da jegliche Veränderung der Daten die Struktur von One-Hot Encoding zerstört.

3.5.1.2 Feature Extractor

Da der Input aus einer Vielzahl an großen Vektoren besteht, könnte man einen Feature Extractor, wie beispielsweise einen Auto Associator, dazwischenschalten, um den großen Input Vektor auf einen kleineren Output Vektor zu reduzieren. Dieser verkleinerte Output Vektor kann in der Datenbank für das Training abgelegt werden und dient im nächsten Schritt als Input für das RNN. In anderen Arbeiten, unter anderem [6], wird Random Projection genutzt um den Input Vektor zu verkleinern. Dies ist quasi eine zufällige Auswahl von Stellen in einem größeren Vektor die weiterverwendet werden. Dadurch gehen jedoch bestimmte Informationen ersatzlos verloren, während sie bei einem Auto Associator zumindest noch leicht das Ergebnis beeinflussen können. Da die Auswahl der Informationen außerdem wie beschrieben dem Zufall überlassen wird, kann mit einem Feature Extractor ein großer Input Vektor gezielter verkleinert werden, als mithilfe von Random Projection.

3.5.1.3 RNN

Die Zahl der Input Neuronen hängt davon ab, auf welchen sinnvollen Vektor der Feature Extractor den Inputvektor reduzieren kann und ergibt sich im Zuge der Implementierung. Die Anzahl der Hidden Layers und deren Neuronen ist schwer zu schätzen, daher wird vorerst eine Hidden Layer mit 40 Neuronen verwendet, die genaue Anzahl ist für die theoretische Modellierung jedoch irrelevant. Von jedem Neuron in dem zweiten Hidden Layer soll eine Verbindung zum Hidden State weggehen, der Daten um einen Zeitschritt verzögert zurück in die Hidden Layer fließen lässt. Da dieses Netz nicht trainiert wird, ergibt sich kein Vorteil aus mehreren Hidden Layers. Die Zahl der Output Neuronen wird danach gerichtet, wie viele Input Neuronen der Classifier benötigt, um sinnvoll eine große Menge an Viren beziehungsweise Normalzuständen erkennen zu können. Als erster Versuch werden 30 Output Neuronen für das RNN vorgesehen. Das RNN ist schematisch mit den richtigen Verbindungen und der richtigen Anzahl an Layers, aber zu wenig Neuronen pro Layer, in Abbildung 24 dargestellt. Da Tensorflow nur Variance Scaling Initialization und Xavier Initialization (Kapitel 2.2.4.1) unterstützt, wird das Xavier Verfahren zur Initialisierung der Gewichtungen verwendet (Stand Februar 2018). Als Aktivierungsfunktion wird eine ReLU verwendet werden, da sie laut Literatur meist zu kürzeren Trainingszeiten führt.

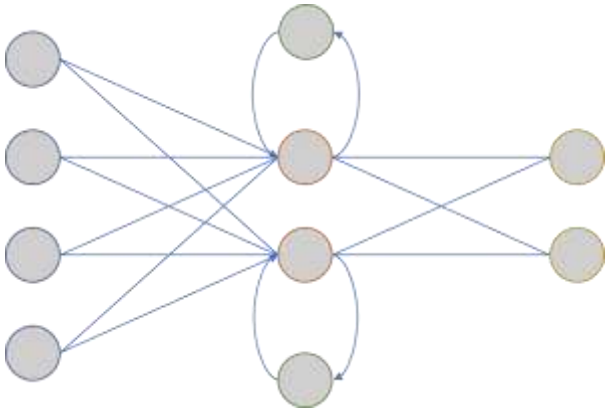

Abbildung 24: RNN mit einer Hidden Layer und einem Hidden State zurück in die Hidden Layer

3.5.1.4 Classifier

Für den Classifier wird ein fully connected Feed-forward Neural Network verwendet, da er weder mehrere Zyklen in Verbindung bringen muss, wie ein RNN, noch spezielle Muster erkennen soll, wie ein CNN. Für die ersten Tests werden circa 30 Input Neuronen verwendet werden, die genaue Zahl richtet sich aber danach, wie viel notwendig ist um sich eine große Menge von Malware beziehungsweise Normalzuständen zu merken. Danach wird eine Hidden Layer mit 20 Neuronen, eine Hidden Layer mit 15 Neuronen und eine Output Layer mit einem

Neuron eingeplant, auch hier werden sich die tatsächlichen Werte erst bei einer praktischen Implementierung zeigen. Wie erwähnt, muss dieses Netz nur eine binäre Aussage treffen, je nach Classifier ob der Input auf Malware beziehungsweise Normalzustand hindeutet oder nicht. Als Aktivierungsfunktion wird eine ReLU verwendet, in der Output Layer allerdings eine Sigmoid Funktion um Ergebnisse im Bereich [0,1] zu erhalten. Auch hier wird Xavier Initialization verwendet.

3.5.1.5 Ergebnis

Das Ergebnis des Systems ist dann durch einen Vergleich der Classifier zu ermitteln. Je nachdem, welches Output Neuron der beiden Classifier aktiver ist, das heißt einen höheren Wert aufweist, deutet der Input auf einen Normalzustand oder auf Malware hin. Sollte der Fall mit geringer Wahrscheinlichkeit eintreten, dass beide Classifier denselben Output produzieren, kann definiert werden, welches Netz entscheidend ist. Da es um die Reduktion der False-Positive Rate geht wird in dieser Arbeit bei Gleichstand der Auslöser als harmlos gewertet.

3.5.1.6 Training

Der Feature Extractor lernt mittels unsupervised Learning und kann eigenständig als erstes trainiert werden. Dafür wird ein Input dem Netz präsentiert, durch kleinere Hidden Layers im Netz propagiert und in einen Output mit gleichvielen Neuronen wie in der Input Layer verwandelt. Nun werden Input und Output Layer verglichen, und das Netz so angepasst, dass der Output Layer besser dem Input Layer entspricht. Ist das Training fertig, kann das Netz an dem kleinsten Hidden Layer „abgeschnitten“ werden und nur von Input Layer bis inklusive zum kleinsten Hidden Layer als Feature Extractor verwendet werden. Der Feature Extractor ist nun in der Lage einen großen Input Vektor in einen kleineren Output Vektor zu verarbeiten, der aber fast alle wichtigen Merkmale beinhaltet wie der große Vektor. Nun lässt man alle Trainingsdaten durch das Netz verarbeiten und speichert den Output Vektor als vorverarbeitetes Event in der Datenbank zu dem entsprechenden Auslöser. Das RNN bedarf basierend auf dem Wissensstand dieser Arbeit keinem zwingendem Training, da es ähnlich einer Hashfunktion vorwiegend darum geht, einen beliebigen Input zu einem Output-Wert zu verarbeiten, in dem Fall mehrere Events. Trotzdem müssen für das weitere Training alle vorverarbeiteten Vektoren der Testdaten durch das RNN verarbeitet werden. Es werden nacheinander alle Events eines Auslösers dem Netz als Input gegeben, immer ein Buchstabe pro Zyklus. Das Ergebnis des Netzes nach Verarbeitung des letzten Events eines Auslösers wird erneut in der Datenbank als Vektor der aggregierten Events abgespeichert. Bis zu diesem Punkt ist das Verfahren für alle Trainingsdaten ident, von da an werden dem Malware Classifier nur noch Malware Beispiele präsentiert und dem Normal Classifier nur Beispiele für Normalverhalten. Der Classifier muss mittels supervised Learning trainiert werden. Dafür werden ihm nacheinander die Vektoren der aggregierten Events der Auslöser präsentiert und der Output mit dem gewünschten Ergebnis verglichen, bei Abweichungen mittels Backpropagation Änderungen am Netz vorgenommen. Das gewünschte Ergebnis ist immer eine 1, da die Netze ausschließlich auf positiv-Beispiele trainiert werden, also das Malware Netz auf Malware und das Normal Netz auf Normalzustände.

3.5.2 Fazit

Dieses Netz soll nun in der Lage sein, eine Reihe von zeitlich gereihten Events, die einem eindeutigen Auslöser zuzuordnen sind, durch ein RNN zu einem einzigen Output zu verarbeiten. Dieser Output berücksichtigt dabei die zeitliche Reihung der Events und dient als Input für die beiden Classifier, die Mithilfe eines Classifiers feststellen, ob es sich eher um Malware oder eher um einen Normalzustand handelt. Dabei ist es möglich manuell einzustellen, ob das System kaum False-Positives erzeugt, aber dafür eine niedrigere Erkennungsrate hat, oder theoretisch umgekehrt eine hohe False-Positive Rate für eine sehr hohe Erkennungsrate in Kauf genommen wird. Dadurch kann das System auf die individuellen Anforderungen und sonstigen Systeme des Anwenders abgestimmt werden, dessen Sicherheitsniveau und eventuelle kompensierende Maßnahmen berücksichtigt werden.

3.6 Zweiter Ansatz

Im zweiten Ansatz wird versucht, durch Erkenntnisse aus dem ersten Ansatz das System noch weiter zu verbessern. Dies ist vor allem dadurch möglich, dass man erst im Zuge der konkreten theoretischen Planung eines Systems auf gewisse Stolpersteine und Widersprüche stößt. Außerdem sollen Vermutungen über mögliche Verbesserungen durch verschiedene Versuche untermauert werden, die in Kapitel 5 dokumentiert sind. Die Versuche im Zusammenhang mit dem zweiten Ansatz verwenden noch nicht die tatsächlichen System Events, da dies für Vorversuche zu aufwendig ist. Stattdessen werden als Gegenstück für gutartige und bösartige Events Daten der Gaußschen Kurve und der logistischen Verteilung verwendet, beide Verteilungen bilden eine sehr ähnliche Glockenkurve. Diese Verteilungen stellen sehr gute Test-Daten dar, da sie präzise sind und keine Messfehler enthalten können, die Daten an sich können aus der jeweiligen Formel generiert werden. Darüber hinaus ist zu berücksichtigen, dass reale Daten ebenfalls immer mathematisch verteilt sind, weswegen sich mathematische Verteilungen wie die Gaußsche Kurve oder die logistische Verteilung sehr gut als authentische Test-Daten eignen.

3.6.1 Erkenntnisse aus dem ersten Ansatz

Feature Extractor

Im ersten Ansatz wurde versucht, einen Feature Extractor wie einen Autoencoder einzusetzen, um den großen Input Vektor auf wesentliche Merkmale zu verkleinern und daher ein kleineres und schnelleres RNN verwenden zu können. Im Zuge der Erarbeitung des ersten Ansatzes stellte sich jedoch heraus, dass die Verwendung eines Feature Extractors zusammen mit One-Hot Encoding in diesem System sowohl problematisch als auch überflüssig ist. Problematisch, da bei One-Hot Encoding die Position der einzelnen Zeichen im Vektor sehr wichtig ist, vor allem die Position des einzigen positiven Zeichens, was durch den beschriebenen Feature Extractor nicht berücksichtigt werden würde. Überflüssig, da auch durch einen verkleinerten Hidden Layer beziehungsweise einen verkleinerten Output Layer im RNN der Input auf einen kleineren Vektor reduziert werden kann und somit zumindest der Input Vektor für den Classifier verkleinert werden. Daher wird für den zweiten Ansatz der Feature Extractor weggelassen und wie in [13] die Daten direkt in ein RNN überführt, das sie zu einem Output verarbeitet und auf die wichtigsten Features reduziert. Dadurch wird im Gegensatz zum ersten Ansatz Zeit für die Vorverarbeitung gespart, sowie Komplexität, Umsetzungs- und Wartungsaufwand reduziert.

Problem einseitiges Trainieren

Die im ersten Ansatz verwendeten Classifier besitzen jeweils nur ein Output Neuron und werden auf exakt eine Kategorie trainiert. Classifier nur auf eine Kategorie zu trainieren wird in dieser Arbeit auch als „einseitiges Training“ bezeichnet. In Hinblick auf die Funktionsweise von neuronalen Netzen führt es jedoch zu einem Fehlverhalten, wenn jeder Classifier nur ein Output Neuron hat beziehungsweise einseitig trainiert wird, da sonst der Classifier dazu übergeht immer denselben Output zu liefern, egal wie der Input lautet. Dies passiert dadurch, dass das Netz im einseitigen Training darauf trainiert wird, für den aus einer Kategorie stammenden Input immer beispielsweise den Wert 1 auszugeben. Bedingt durch den Umstand, dass auch Daten einer einzigen Kategorie einer gewissen Verteilung unterliegen und nicht ident sind wird das Netz trotzdem meistens nicht genau 1 als Output liefern, sondern beispielsweise 0,95. Damit ergibt die Kostenfunktion ein Ergebnis größer Null und Änderungen werden am Netz vorgenommen. Da aber nur ein Output Neuron existiert ist die einzige Verbesserungsmöglichkeit um für alle Daten des Netzes tatsächlich das Ergebnis 1 zu liefern, nicht auf den Input zu achten, sondern immer als Ergebnis 1 zu produzieren. In der Validierung führt dies dazu, dass das Netz auch für alle neuen Daten einer anderen Kategorie den Wert 1 produziert ohne sie tatsächlich zu verarbeiten, und sie damit falsch klassifiziert. Dieses Problem soll durch Abbildung 25 veranschaulicht werden, die ein stark vereinfachtes Feed-forward Neural Network mit Sigmoid Aktivierungsfunktion zeigt, bei dem aufgrund hoher Gewichtungen der Input kaum noch entscheidend ist. Diese Abbildung zeigt das Problem stark überzeichnet, natürlich steigen im Normalfall die Gewichtungen nicht so hoch, dafür gibt es meistens mehr Layer wodurch bereits geringere Gewichtungen zu diesem Effekt führen können. Es wurde auch versucht, dieses Problem

praktisch in Tensorflow nachzubauen, Details dazu sind in Kapitel 6.1 beschrieben. Dabei zeigte sich, dass unter Verwendung von Bias-Units dieses Problem am schnellsten auftritt, aber auch ohne Bias-Units war ein einseitig trainiertes Netz nicht verwendbar.

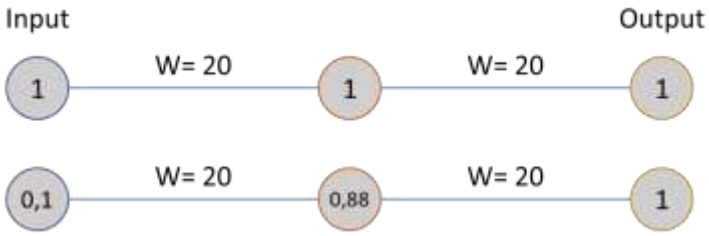

Abbildung 25: Problem von zu hohen Gewichtungen, W stellt den Wert der Gewichtung dar

Lösungsidee 1: Unterklassen bilden

Jeder Classifier könnte auf verschiedene Kategorien trainiert werden, was verhindert, dass ein Input ohne Analyse einer Klasse zugeordnet werden kann. Damit würde dieses Problem gelöst werden und gleichzeitig ein Netz zur Malware-Klassifikation entstehen. Die Unterklassen sind möglich, da sich Normalzustand und Malware wieder in Klassen zerlegen lassen, wie beispielsweise Spyware und Dropper. Durch diese Unterklassen kann das Netz dazu gezwungen werden den Input zu berücksichtigen und möglichst gut einer dieser Klassen zuzuordnen. Dieser Classifier gleicht damit den üblichen Classifiern, die auf mehrere Kategorien trainiert werden und meiste mehrere Output Neuronen besitzen. Diese Zuordnung zu Klassen ist dabei grundsätzlich nur ein Behelf um das Training durchführen zu können, bietet jedoch hilfreiche Zusatzinformationen über den analysierten Auslöser. Dieser Ansatz funktioniert auf jeden Fall, dafür erfordert dieses Vorgehen detailliertere Beschriftung der Daten, da nur Malware oder Normal nicht mehr ausreichend ist, was gerade bei den Daten für den Normalzustand schwierig werden kann. Diese Kategorien könnten dann pro Classifier mittels Max-Pooling auf ein Neuron zusammengeführt werden, das wie bei dem ersten Ansatz den Classifier repräsentiert und als Vergleich der beiden Classifier genutzt werden kann. Die Idee ist, das Netz dadurch zu zwingen, mehr zu generalisieren, ähnlich wie die Idee hinter dem Multi-Task Netz.[6] Diese Lösungsidee wird allerdings durch den sehr hohen Aufwand für Labeling nicht weiterverfolgt. Gerade weil die Netze regelmäßig mit neuen Daten trainiert werden sollten, wäre der hohe Aufwand regelmäßig notwendig.

Lösungsidee 2: L2 Regularization

L2 Regularization verhindert in Netzen zu hohe Gewichtungen, wodurch die Gewichtungen klein gehalten werden und der Input nicht wie in Abbildung 25 vernichtet wird. Allerdings ist dies eher eine Maßnahme zur Symptom-Bekämpfung und keine stabile und sinnvolle Lösung. Unter anderem verändert es als einzige Lösungsidee die Gewichtungen „manuell“ und verfälscht somit das Ergebnis. Durch eine unterschiedliche Entwicklung und Bestrafung der Gewichtungen in den beiden Classifiern könnte die Vergleichbarkeit leiden. Die Lösungsidee wird daher nicht weiterverfolgt.

Lösungsidee 3: Inverses Training

Man könnte einen Classifier mit dem additiven Invers der Trainingsdaten trainieren, damit das Ergebnis nicht mehr unabhängig vom Input sein darf. Beispielsweise würde es für einen Vektor [0,0,1,0], der das Ergebnis 1 liefern soll, ebenso einen Vektor [1,1,0,1] geben, der das Ergebnis 0 liefern soll. Dieser Ansatz ist eher experimentell und soweit dies ermittelt werden konnte noch nicht in der Literatur beschrieben, hätte aber den großen Vorteil, dass der Aufwand dafür kaum steigt, da sich die inversen Trainingsdaten automatisch generieren lassen und im Gegensatz zu Lösungsidee 1 kein hoher manueller Aufwand notwendig ist. Im Gegensatz zu Lösungsidee 1 werden dadurch keine näheren Informationen zum Auslöser geliefert, es reicht aber für viele Anwendungsfälle feststellen zu können, ob ein Datensatz aus der trainierten Kategorie stammt, oder nicht. Details zu Versuchen, ob dieser Ansatz funktioniert befinden sich in Kapitel 6.2. Dort wird zwar gezeigt, dass

der Ansatz funktioniert, allerdings ist der Effekt selbst unter perfekten Testbedingungen zu schwach um ihn sinnvoll nutzen zu können.

Lösungsidee 4: Rekonstruktions-Netz

Die letzte Lösungsidee beschäftigt sich mit Netzen, die auf die Rekonstruktion von Input trainiert sind, daher kurz Rekonstruktions-Netze genannt. Man trainiert ein Netz beispielsweise darauf, Daten der Kategorie x möglichst gut zu rekonstruieren, der Output soll also möglichst dem Input gleichen. Die Vermutung ist, dass das Netz danach Daten der Kategorie x besser rekonstruieren kann, als Daten der Kategorie y oder z, weil es das Muster der Daten kennt. Besser rekonstruieren bedeutet in dem Fall, dass die Abweichung zwischen Soll-Output und Ist-Output, also die Kosten, geringer sind. Damit würde sich feststellen lassen, ob Testdaten aus der Kategorie x sind oder aus einer anderen beliebigen Kategorie. Diese Idee angewendet auf das System mit zwei Classifiern würde bedeuten, dass ein Classifier auf die Rekonstruktion von Kategorie x trainiert ist, während der andere Classifier auf die Rekonstruktion der Kategorie y ist. Mit diesem Verfahren würde sich das System problemlos um ein drittes Netz erweitern lassen, welches wiederum auf eine eigene Kategorie trainiert ist, jedoch müssten sich die Kategorien aller Netze ausschließen. So könnte aber beispielsweise das System aus einem Netz für die Erkennung von Normalzustand, einem Netz für Spyware und einem Netz für restliche Malware bestehen. Mit steigender Anzahl an Netzen wird vermutlich jedoch die Erkennungsrate schlechter.

Feed-forward Neural Network

Anstatt eines Feed-forward Neural Networks als Classifier könnte ein DBN im zweiten Ansatz verwendet werden, welches im Grunde zwar im Aufbau ident ist, aber anders trainiert wird, wodurch ein gezielteres und schnelleres Training möglich sein soll, aber auch die Menge an benötigten beschrifteten Daten für das supervised Learning reduziert wird. Daher wurden zu DBNs an sich und dem Vergleich zwischen DBNs und FFNNs verschiedene Versuche durchgeführt, die in Kapitel 6.3 beschrieben sind. Dort zeigt sich jedoch, dass die Performance von DBNs in diesem Fall vergleichbar mit der Performance des FFNNs ist, die Trainingszeit jedoch deutlich länger ausfällt. Deswegen wird auch weiterhin ein Feed-forward Neural Network verwendet.

ReLU in Classifier bei negativen Inputwerten

Im ersten Ansatz werden im Classifier als Aktivierungsfunktion ReLUs verwendet. Auch wenn die Performance von ReLU als Aktivierungsfunktion in vielen Netzen überzeugend sein mag, stellt sich erst auf den zweiten Blick heraus, dass sie für den Classifier in diesem System ungeeignet ist. Durch das vorgeschaltete RNN werden die Inputwerte für den Classifier negativ sein, diese würden allerdings von der ReLU alle auf null gesetzt werden, wodurch wertvolle Informationen verloren gehen. Stattdessen soll daher für den Classifier in jedem Layer eine Sigmoid-Funktion verwendet werden. Für diesen Umstand sind aufgrund der Eindeutigkeit keine Versuche notwendig.

3.6.2 Aufbau

Da der Feature Extractor wegfällt, aber die restlichen Änderungen in den Netzen selbst passieren, ändert sich der Aufbau des Systems kaum im Vergleich zum ersten Ansatz, wie in Abbildung 26 zu sehen ist.

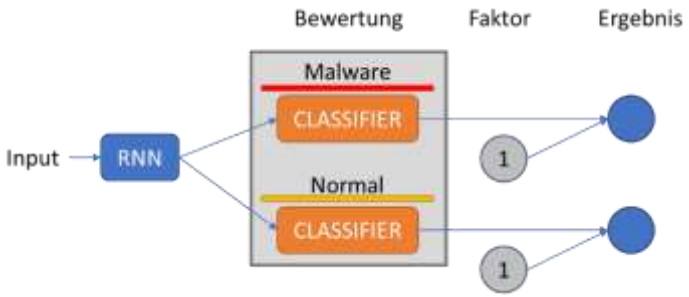

Abbildung 26: Der Aufbau des Systems im zweiten Ansatz, im Wesentlichen aus einem RNN und zwei Classifiern bestehend

3.6.2.1 Input

Der Input besteht wie im ersten Ansatz aus Events in zeitlicher Abfolge, die mit One-Hot Encoding vorverarbeitet wurden. Die Events sind einem Auslöser zugeordnet, in einem Durchlauf werden immer nach der Reihe alle Events eines Auslösers durch das Netz verarbeitet und dann ein einziges Ergebnis erzeugt.

3.6.2.2 RNN

Wie auch im ersten Ansatz sollen die einzelnen One-Hot Vektoren der Events, beziehungsweise die Events an sich, durch ein RNN zu einem einzigen Output verknüpft werden, der dann weiterverarbeitet werden kann. Daher würde sich ein Input Vektor von der Länge des Alphabets mal der Anzahl der Zeichen aller Events eines Auslösers ergeben. Dadurch werden alle Events eines Auslösers durch eine einzige Zahlen-Abfolge repräsentiert. Die Zahl der Output Neuronen wird danach gerichtet, wie viele Input Neuronen der Classifier benötigt, um sinnvoll eine große Menge an Viren beziehungsweise Normalzuständen erkennen zu können. Als erster Versuch werden 30 Output Neuronen für das RNN verwendet. Sonst gleicht der Aufbau dem RNN im ersten Ansatz, es besteht aus drei Schichten und besitzt einen Hidden State, wie in Abbildung 24 abgebildet. Die Hyperparameter des Netzes werden in Tabelle 1 dargestellt.

Tabelle 1: Hyperparameter des RNNs

Netz	
Typ	Recurrent Neural Network
Struktur	73-30-30 / Input Layer – Hidden Layer – Output Layer
Initialization	Xavier
Kostenfunktion	Binary Crossentropy
Optimierungsverfahren	Adam
Aktivierungsfunktion	ReLU
Menge Testdaten	1000000
Batchgröße	100
Epochen	5
Durchmischung des Trainingssets pro Epoche	True

3.6.2.3 Classifier

Der nächste Teil besteht aus zwei Classifiern, denen jeweils der Output des RNNs für einen Auslöser gegeben wird. Als Classifier wird ein Feed-forward Neural Network mit den Hyperparametern aus Tabelle 2 verwendet, die an die Ergebnisse in Kapitel 6.3.5 angelehnt sind und beispielsweise Spezialisierung fördern sollen. Aus den weiter oben beschriebenen Gründen wird für diesen Classifier allerdings keine ReLU, sondern ausschließlich Sigmoid-Funktionen verwendet. Jedes Netz besitzt X Output Neuronen, wobei für jedes Neuron der Output Wert mit dem Input Wert verglichen wird und die Abweichung aller X Neuronen aufsummiert wird. Dieser Wert stellt

dann quasi die Gesamtabweichung vom Input dar, dies entspricht prinzipiell den Kosten, wird aber hier als Ergebnis bezeichnet.

Tabelle 2: Hyperparameter des Classifiers

Netz	
Typ	Feed-forward Neural Network
Struktur	30-40-35-30
Initialization	Xavier
Kostenfunktion	Mean Absolute Error
Optimierungsverfahren	Adam
Aktivierungsfunktion	Sigmoid
Menge Testdaten	1000000
Batchgröße	150
Epochen	10
Durchmischung des Trainingssets pro Epoche	True

3.6.2.4 Ergebnis

Das Ergebnis beider Netze wird nun herangezogen, um den Input zu kategorisieren. Zuerst wird jedes Ergebnis mit dem jeweiligen Faktor des Netzes multipliziert, der im Normalfall 1 ist aber einen beliebigen Wert zwischen 0.1 und 5 annehmen könnte. Das Ergebnis der jeweiligen Multiplikation wird nun als gewichtetes Ergebnis bezeichnet. Die Theorie, dass ein Netz, welches auf die Rekonstruktion der Kategorie X trainiert wurde, im Betrieb Daten der Kategorie X besser rekonstruieren kann als Daten einer anderen Kategorie, wird als Ausgangspunkt für das weitere Vorgehen genommen. Es werden nun die beiden gewichteten Ergebnisse verglichen und analog zu der oben beschriebenen Theorie das Netz, welches das niedrigere gewichtete Ergebnis erzielt hat, als „Sieger“ festgelegt. Das heißt, der Input entstammt wahrscheinlich der Kategorie, auf die das Sieger-Netz trainiert wurde. Diese Berechnungen und Vergleiche lassen sich problemlos vollautomatisch durchführen.

3.6.2.5 Training

Die Trainingsphase wird vom RNN bis auf weiteres übersprungen, da dem RNN lediglich eine Hash-ähnliche Funktion zukommt, es aber für das Ergebnis fast irrelevant ist, wie das RNN die Werte verknüpft. Um den Classifier trainieren zu können, muss das RNN zuerst alle Trainingsdaten vorverarbeiten und wieder in der Datenbank ablegen. Jeder Classifier wird nun ausschließlich mit Daten der dem Classifier zugewiesenen Kategorie trainiert, wofür die vom vorverarbeiteten Daten verwendet werden. Das Training ist unsupervised, da das Netz den Input verarbeitet, durch Abgleich von Input und Output die Abweichung feststellt und durch Backpropagation Änderungen am Netz vornimmt. Das Netz ist nach Absolvierung der festgelegten Epochen fertig trainiert. Zur Evaluierung sollte das gesamte System getestet werden, inklusive des automatischen Abgleichs zwischen den beiden Netzen, und je nach Anspruch festgelegt werden, welche Trefferquote ausreichend ist. Sollte diese nicht erreicht werden, kann man das Training weiterführen oder die Hyperparameter ändern und das Training neu durchführen.

3.6.2.6 Fazit

Dieses Netz, oder aufgrund der vielen Netze eher System, sollte nun in der Lage sein, mehrere in zeitlichem Zusammenhang stehende Events eines Auslösers zu verarbeiten und feststellen, ob der Auslöser mutmaßlich Malware ist oder ob es sich um einen normalen Vorgang handelt. Der Output sollte binär sein und kann dadurch auch leicht weiterverarbeitet werden. Abgesehen davon sollte das Netz durch die Struktur sehr schnell arbeiten und je nach Hardware auch Realtime Analyse der anfallenden Daten ermöglichen. Schlussendlich bietet theoretisch der Aufbau mit zwei parallelen Classifiern, die jeweils auf gutartig oder bösartig trainiert sind, eine Vielzahl an Vorteilen. So sind die beiden Classifier sowohl zeitlich als auch örtlich unabhängig voneinander

trainierbar. Dies ermöglicht nicht nur eine Verteilung beim Training, sondern es macht möglich, jederzeit einen der beiden Classifier oder beide Classifier durch eine neuere oder bessere Version zu ersetzen, ohne den jeweils anderen Classifier zu stören. Dies ist zum Beispiel regelmäßig notwendig, um den Classifier für bösartiges Verhalten auf neue Malware zu trainieren. Auch sollte der Classifier für gutartiges Verhalten in regelmäßigen Abständen neu trainiert und ersetzt werden, um wieder das übliche normale Verhalten in der Umgebung abzubilden. Wichtig ist jedoch, dass beide Classifier immer dieselbe Technologie und dieselben Hyperparameter nutzen und mit demselben RNN trainiert werden müssen. Durch den abschließenden Vergleich der Ergebnisse würde das System nicht mehr funktionieren, wenn ein Classifier durch andere Hyperparameter auch bei unbekannten Kategorien noch weit bessere Ergebnisse liefert, als der zweite Classifier bei bekannten Kategorien. Auch würde das System nicht mehr funktionieren, wenn die Daten mit einem anderen RNN vorverarbeitet werden. Theoretisch ist das System so modular, dass beispielsweise der Classifier für bösartiges Verhalten durch einen Classifier für Spyware und einen Classifier für ≠Spyware (Nicht-Spyware) ersetzt werden kann, oder für jede Malware-Art ein eigener Classifier eingesetzt wird. Damit lässt sich der Fokus gut auf die Ziele des Systems richten, beispielsweise vom einfachen zwei-Classifier System für Anti-Malware Systeme bis zum 5-Classifier System für Malware Analysten, SIEM Systeme und Ähnlichem. Der letzte große Vorteil, den das System eingebaut hat, ist das Faktor-System. Jeder der Classifier hat einen Faktor, der mit dem Endergebnis des Classifiers multipliziert wird und standardmäßig 1 beträgt. Soll beispielsweise erreicht werden, dass ein Netz so wenig False Positives wie möglich produziert und nimmt dafür eine niedrigere Erkennungsrate in Kauf, kann der Faktor des gutartigen Classifiers beispielsweise auf 0.9 oder 0.8 gesetzt werden. Damit wird das gewichtete Ergebnis geringer und geht öfter als „Sieger“ hervor. Wenn beide Netze zu einem ähnlichen Ergebnis kommen, wird dadurch ein Input eher als gutartig klassifiziert, und eher nicht als bösartig, was genau zu dem gewünschten Effekt führt, weniger False Positives bei niedrigerer Erkennungsrate. Die Nachteile dieses Systems belaufen sich auf die bisher nur theoretische Modellierung des Systems. So kann theoretisch nicht geklärt werden, ob die Kapazitäten des Systems ausreichen, die Verarbeitung des RNNs unterschiedlich genug ausfällt und ähnliches. Diese Nachteile können also erst bei der praktischen Implementierung hervortreten.

3.7 Praktische Umsetzung des Systems

In diesem Kapitel soll das System aus dem zweiten Ansatz in Python3 umgesetzt und Aufbau und Struktur erklärt werden. Um definierte Schnittstellen zu erhalten wird es eine eigene Klasse für das RNN und eine eigene Klasse für das Zwei-Classifier System geben. Dadurch soll das System einerseits als Ganzes, aber auch die einzelnen Komponenten getrennt gesteuert werden, was sowohl die Entwicklung als auch die Testphase erleichtert. Eine weitere Anforderung stellt das ausführliche Logging dar. So soll jede Komponente als auch das System an sich in ein eigenes Logfile schreiben, ein Abgleich wird durch eine einheitliche Identifikationsnummer pro Durchlauf ermöglicht, die sogenannte Run ID. In den Logfiles sollen vor allem Hyperparameter, Aufbau und Testergebnisse protokolliert werden. Fehlermeldungen werden ebenfalls in ein eigenes Logfile ausgegeben, auch wenn das durch das Tensorflow Backend nur bedingt möglich ist. Es soll die Möglichkeit geben eine Debug Funktion zu aktivieren, die dazu führt, dass umfassende Informationen auf die CLI ausgegeben werden, beispielsweise Auszüge aus den verarbeiteten Daten. Das Programm soll über die CLI gesteuert werden können, im Wesentlichen sind also keine Änderungen in den Files notwendig. Dies erhöht einerseits den Komfort, verhindert aber auch Fehler beim Anpassen von Werten. Die letzte wesentliche Anforderung an das Programm ist, den Code mit ausführlichen Kommentaren zu versehen, um ihn im Zuge der Erstellung leichter überarbeiten zu können und um auch langfristig damit einen Mehrwert zu generieren. Für eine bessere Übersichtlichkeit wird das System auf vier Files aufgeteilt. Das erste File ist das Hauptprogramm, in dem sich die Logik für die Bedienung über CLI befindet und die Klassen instanziiert und gesteuert werden. Das zweite File beinhaltet die Klasse für das Zwei-Classifier System, das dritte File die Klasse für das RNN. Das vierte File stellt Hilfsfunktionen zur Verfügung, wie beispielsweise eine erleichterte Ausgabe der Hyperparameter. Ebenfalls sollen zur besseren Übersicht verschiedene Arten von Files streng voneinander getrennt werden, so liegen im Hauptordner nur die vier Programmfiles, im Ordner Log alle Logfiles, im Ordner data alle Daten und im Ordner model alle gespeicherten Modelle. Ein sonst wichtiger Punkt, der bei dieser Umsetzung allerdings wenig Priorität

hat, ist die einfache und fehlerfreie Bedienung des Programms. Da dieses Programm nicht für Endanwender, sondern für Forschungszwecke gedacht ist, wird wenig Aufwand in die einfache Bedienung investiert. Außerdem wird vom Anwender Hintergrundwissen erwartet, da beispielsweise nicht immer sichergestellt werden kann, dass der Anwender nicht die Möglichkeit hat ein Netz zu verwenden, bevor es trainiert wurde.

3.7.1 Daten

3.7.1.1 Herkunft der Daten

Die Daten bestehen aus Systemevents, wie beispielsweise Laden von DLLs oder Netzwerkverbindungen, aufgeteilt in einerseits normale Events, andererseits bösartige Events verursacht von Malware oder Angriffen. Die Prozesse, aus denen diese Events stammen, reichen dabei von Windows Systemprozessen wie rundll32 bis Benutzerprozesse wie Chrome oder 7Zip. Die alltäglichen, ungefährlichen oder „normalen“ Events wurden im Forschungszentrum der FH St. Pölten über mehrere Wochen gesammelt. Diese Daten sollen dafür verwendet werden, einem Netz den „Normalzustand“ anzulernen. Die Annahme ist, dass diese Daten am besten den Normalzustand eines Systems im Forschungszentrum der FH St. Pölten repräsentieren und im Zuge des Betriebs durch eine genauere Abgrenzung von bösartigen Files zu einer niedrigeren False-Positive Rate führen. Die bösartigen Events wurden in der FH St. Pölten gewonnen, in dem nach der Reihe Malware Samples in einer dafür ausgelegten Sandbox ausgeführt und alle Events aufgezeichnet wurden. Dadurch existiert hier natürlich die von vielen Forschern beschriebene Einschränkung, dass Malware nicht in der natürlichen Umgebung beobachtet wird und sich daher eventuell anders verhält oder generell alle Aktivitäten einstellt. Diese gesammelten Events sind Kernel Events und lassen sich grundlegend in die Kategorien Process Events, Thread Events, Image Load Events, File Events und Network Events gliedern, näher beschrieben von Luh et al.[69] Diese Events sind dabei über Zeit und Prozess-ID zu sogenannten „Traces“ verknüpft. Daher können damit nahezu sämtliche Vorgänge auf einem System detailliert erfasst und beschrieben werden. Die in Kapitel 3.1.2 beschrieben möglichen Probleme aufgrund unterschiedlicher Herkunft der Daten dürfte nicht gegeben sein, da sowohl die gutartigen Events als auch die bösartigen Events mit derselben Anwendung erhoben wurden.

3.7.1.2 Verarbeitung der Daten

Da einerseits im Forschungszentrum der FH St. Pölten zum Zeitpunkt der Aufzeichnung Malware aktiv gewesen sein könnte, andererseits auf einer virtuellen Maschine neben dem Malware-Code auch eine Vielzahl an gutartigen Prozessen läuft, wie beispielsweise Windows Update, wurden die Daten von Luh et al. [69] noch weiterverarbeitet. Es wurden aus den Daten Bi-Grams gebildet, also immer zwei miteinander auftretende Events gemeinsam zu einer neuen Einheit verknüpft, und dann mithilfe von statistischen Verfahren geprüft, ob ein Bi-Gram eher bösartig oder eher gutartig ist, je nachdem ob das Bi-Gram häufiger in den gutartigen Daten oder häufiger in den bösartigen Daten vorkommt. Daraus ergibt sich ein Wert zwischen -1 und 1, der aussagt, wie wahrscheinlich ein Bi-Gram böse (-1) oder gut (1) ist. Diese Verknüpfung war notwendig, da ein Token alleine kaum Aussagen über Bösartigkeit oder Gutartigkeit zulässt. Auch die Bi-Grams kommen meist in beiden Datenquellen vor, lassen sich jedoch bereits besser einteilen. Details zu dieser Verarbeitung befinden sich in der dazugehörigen Forschungsarbeit von Luh et al.[69] Von diesen Daten werden in dieser Arbeit 95% für das Training des Systems verwendet, 5% für die Evaluierung. Die bösartigen Daten bestehen aus circa 220.000 Bi-Grams, die gutartigen Daten aus circa 20 Millionen Bi-Grams, werden aber aus technischen Gründen auf dieselbe Menge reduziert. Würde nämlich ein Datensatz wesentlich größer sein, würde der jeweilige Classifier viel besser trainiert werden und alle Daten besser rekonstruieren können.

3.7.1.3 Auszug aus den Daten

Die folgenden Zeilen zeigen einen kurzen Auszug aus den Daten. Wie bereits erwähnt handelt es sich um Bi-Grams, es wurde immer für zwei Events gemeinsam ein Wert errechnet. Jedes der Events kann auch in Kombination mit anderen Events als Bi-Gram auftreten. Im Datensatz sind die beiden Events und der Wert in einer Zeile und werden durch einen Strichpunkt getrennt, die Darstellung hier dient nur der besseren Lesbarkeit.

Event1: file-3,/namedpipe/davo.41348.54686228

Event2: network-6,132.44.198.14
Wert: 0.03391838473849820

Event1: file-3,/namedpipe/davo.58621.58762168
Event2: network-6,158.12.216.93
Wert: -0.20159235184087124

Event1: network-6,56.138.19.144
Event2: file-3,/users/benjamin/appdata/local/google/chrome/user.data/profile.5/media.cache/data
Wert: 0.00115923925520161

3.7.2 Erstellung des Codes

Der Code wurde in Tensorflow 1.7.0 mit Keras geschrieben, die genaue Version von Keras konnte leider nicht herausgefunden werden. Aufgrund der weiter oben beschriebenen Probleme mit der Verwendung von Tensorflow APIs wird die praktische Umsetzung ausschließlich in Keras erfolgen, Tensorflow dient nur als Backend. Es wurden außerdem noch verschiedene Libraries wie numpy oder libhdf5-dev verwendet. Damit im Code keine Änderungen an verschiedenen Stellen vorgenommen werden müssen, wurden alle Parameter herausgehoben und können ganz oben im Programm geändert werden. Dort befinden sich neben globalen Konfigurationen wie dem Debug Modus auch die Hyperparameter für die beiden Netze. Der Code wird Großteiles analog zum zweiten Ansatz in den folgenden Unterkapiteln erklärt. Es wird hier bewusst kein Auszug aus dem Code abgebildet, da der ganze Code dafür zu lange ist und es keine einzelnen wichtigen Funktionen gibt, sondern fast jede Funktion wichtig ist.

3.7.2.1 Programmstart

Vor Programmstart sollten im oberen Teil des Hauptprogramms die Einstellungen wie Hyperparameter, Debug Modus und Ausgabe angepasst werden. Der Debug Modus gibt umfassende Informationen während des Betriebs aus, die Ausgabe Einstellung ermöglicht es den Standard Output in ein File umzuleiten, falls man das Programm im Hintergrund laufen lassen will. Danach kann das Programm vollständig über die Command Line bedient werden. Dafür muss als erstes Argument ein System angegeben werden, entweder „RNN“, „Classifier“ oder „System“. Bei Verwendung von „RNN“ und „Classifier“ wird jeweils nur das entsprechende Netz angelegt, alle folgenden Argumente beziehen sich dann nur auf das Netz. „System“ legt sowohl RNN als auch Classifier an und alle weiteren Argumente beziehen sich auf beide Netze, Abhängigkeiten werden automatisch geregelt. Nach dem System können noch ein oder mehrere Aktionen angegeben werden:

- Create: Erstellt das Netz.
- Train: Trainiert das Netz mit dem Trainingsset, nur für Classifier und System verfügbar.
- Save: Speichert das Netz mit Gewichtsmatrix ab.
- Load: Ladet das Netz mit Gewichtsmatrix.
- Eval: Evaluiert das Netz mit dem Evaluierungsset, nur für Classifier und System verfügbar.
- Preprocess_train: Führt Vorverarbeitung der Trainingsdaten durch, nur für RNN und System verfügbar.
- Preprocess_eval: Führt Vorverarbeitung der Evaluierungsdaten durch, nur für RNN und System verfügbar.

Wie zu sehen ist stehen für das System „System“ alle Befehle zur Verfügung. Dadurch, aber auch durch die automatische Berücksichtigung der Abhängigkeiten sollte das System daher möglichst so gesteuert werden. Da beim Speichern und Laden von Netzen kein Ziel angegeben werden kann, können immer nur ein RNN und ein Classifier gleichzeitig gespeichert werden. Weitere Speicher-Vorgänge überschreiben das jeweilige Modell. Wichtig ist jedoch die richtige Reihenfolge der Parameter, die, um die maximale Flexibilität zu gewährleisten, nicht eingeschränkt wurde. So sollte beispielsweise die Aktion „create“ vor der Aktion „train“ stehen. Einige Beispiele für die Bedienung des Programms:

- *„Keras_system.py rnn create save preprocess_train“*: Erstellt RNN und speichert es, führt Vorverarbeitung der Trainingsdaten druch.
- *„Keras_system.py classifier create train save“*: Erstellt Classifier, trainiert und speichert ihn.
- *„Keras_system.py rnn load preprocess_eval“*: Ladet zuvor gespeichertes RNN und führt Vorverarbeitung der Evaluierungsdaten durch.
- *„Keras_system.py classifier load eval“*: Ladet zuvor gespeicherten Classifier und evaluiert ihn.
- *„Keras_system.py system create train save eval save“*: Führt all die obigen Befehle gemeinsam aus, in dieser Reihenfolge:
  - *Create*: Erstellt RNN und Classifier.
  - *Train*: Führt Vorverarbeitung Trainingsdaten mit RNN durch, trainiert damit Classifier.
  - *Save*: Speichert RNN und Classifier.
  - *Eval*: Führt Vorverarbeitung Evaluierungsdaten mit RNN durch, evaluiert Classifier damit.
  - *Save*: Überschreibt gespeicherte Modelle mit aktuellen Modellen.

3.7.2.2 Input

Als Input dienen die weiter oben beschriebenen aufbereiteten Daten. Diese Daten werden circa zu 95% für das Trainingsset verwendet, zu circa 5% für das Evaluierungsset. Beide Datenarten, also gutartige und bösartige Events, sind jeweils gleich häufig vertreten. Wie weiter oben beschrieben sind die Daten Bigrams, das heißt eine Zeile ist im Format Event1;Event2 und wird in dieser Form mittels One-Hot Encoding pro Zeichen vorverarbeitet.

3.7.2.3 RNN

Das One-Hot Encoding erfolgt im Programm in der Klasse des RNNs, da dieses Netz anschließend die Daten ebenfalls vorverarbeiten muss. Aus einer Zeile mit 100 Zeichen werden durch das One-Hot Encoding circa 8.000 Zeichen. Durch die Vorverarbeitung der Daten durch das RNN sollen diese großen Vektoren wieder reduziert werden. Das RNN hat für die Vorverarbeitung die Funktionen Model_preprocess_train um die Trainingsdaten vorverarbeiten zu können und Model_preprocess_eval um die Evaluierungsdaten vorverarbeiten zu können. Dabei wird dem RNN pro Zyklus ein One-Hot Vektor als Input gegeben, bis alle Vektoren der Inputzeile verarbeitet sind. Multipliziert mit der Größe des Alphabets reduziert das RNN so die Input Daten von circa 8.000 Dimensionen auf circa 300 Dimensionen, je nach Hyperparameter. Die Umwandlung von einem Input mittels One-Hot Encoding wird vereinfacht in Abbildung 27 dargestellt, für ein Alphabet aus vier Buchstaben. Wie gezeigt ergeben sich dort aus fünf Input Zeichen nach Umwandlung 15 Zeichen.

Abbildung 27: Umwandlung eines Inputs mittels One-Hot Encoding.

Das Ergebnis wird in eine Datei im Ordner data geschrieben, sodass auch größere Datenmengen problemlos eingelesen werden können und nicht im Speicher bleiben. Außerdem hat dies den Vorteil, dass die Daten bei Programmabsturz nicht verloren gehen und die Vorverarbeitung leicht von der eigentlichen Klassifizierung zeitlich und räumlich getrennt durchgeführt werden kann. Da die Daten für die beiden Classifier unterschiedlich

sind, werden sie auch in unterschiedlichen Dateien gespeichert, einmal für gutartige Events und einmal für bösartige Events. Eine vollständige Liste der Funktionen der Klasse RNN:

- Save_model: Diese Funktion speichert das Netz und die Gewichtsmatrix ab. Dies ist sehr wichtig, da durch eine Änderung der Gewichtsmatrix das RNN die Daten anders verarbeiten würde und der Classifier nicht mehr korrekt funktionieren würde.
- Load_model: Diese Funktion ladet das Netz und die Gewichtsmatrix.
- Create_model: Diese Funktion erstellt das Netz anhand der angegebenen Hyperparameter.
- __Check_status: Diese Funktion dient der sicheren Anwendung und versucht sicherzustellen, dass beispielsweise ein Netz erst nach dem Erstellen verwendet werden kann.
- __Convert_array_to_onehot: Wandelt ein Array in ein Array aus One-Hot Vektoren um.
- __One_hot_encode_char: Wandelt jedes Zeichen in einen One-Hot Vektor um.
- Model_preprocess_train: Wie oben beschrieben zur Vorverarbeitung der Trainingsdaten.
- Model_preprocess_eval: Wie oben beschrieben zur Vorverarbeitung der Evaluierungsdaten.
- Model_predict: Diese Funktion verwendet das fertige Netz um damit Daten zu verarbeiten. Wird daher beispielsweise von Model_preprocess_train benötigt.

Die Funktionen mit zwei Unterstrichen zu Beginn sind ausschließlich als interne Funktionen gedacht und sind nicht dafür ausgelegt, von außerhalb der Klasse aufgerufen zu werden. Die Klasse besitzt keine Funktion zum Trainieren des RNNs, da das RNN wie im zweiten Ansatz beschrieben nicht trainiert wird. Mehr Informationen dazu auch im Kapitel 5. Die RNN Klasse besitzt außerdem noch eine untergeordnete Klasse namens FileIterator, die ein File öffnet und immer eine Zeile zurückgibt, ähnlich wie ein Generator in Python. Die Klasse RNN legt außerdem bei der Instanziierung ein eigenes Logfile an und vermerkt dort die globale Run ID, um einen Abgleich zwischen den Logfiles zu ermöglichen. Sollte im Hauptprogramm der Debug-Modus aktiviert sein, werden in der Klasse RNN umfassendere Informationen ausgegeben, beispielsweise ein Datensatz in den verschiedenen Verarbeitungsschritten, um die Korrektheit sicherstellen zu können.

3.7.2.4 Classifier

Die Klasse TwoClassifier ist sehr ähnlich zur Klasse des RNNs aufgebaut. Die Klasse hat ebenfalls ein eigenes Logfile, in das neben Hyperparameter auch die Ergebnisse ausgegeben werden. Diese Klasse kann nun mithilfe der vorverarbeiteten Daten trainiert und danach evaluiert werden. Dafür werden die Daten aus den Files Zeichen für Zeichen eingelesen und in Float-Werte konvertiert. Für das Training sind keine Labels notwendig, da das Netz auf Rekonstruktion trainiert wird, folglich ist der gewünschte Output der Input selbst. Die beiden Classifier werden hintereinander nur mit den für den Classifier bestimmten Daten trainiert. Der Classifier „gut“ nur mit gutartigen Events, der Classifier „bad“ nur mit bösartigen Events. Die unterschiedliche Benennung der Classifier auf Deutsch und Englisch kommt daher, das nur so die beiden Namen gleich lang sind, was beim Programmieren den Komfort deutlich hebt und damit die Fehleranfälligkeit senkt. Eine vollständige Liste der Funktionen der Klasse TwoClassifier:

- Save_model: Diese Funktion speichert das Netz und die Gewichtsmatrix ab, und damit den Trainingsfortschritt.
- Load_model: Diese Funktion ladet das Netz und die Gewichtsmatrix.
- Create_model: Diese Funktion erstellt das Netz anhand der angegebenen Hyperparameter.
- __Check_status: Diese Funktion dient der sicheren Anwendung und versucht sicherzustellen, dass beispielsweise ein Netz erst nach dem Erstellen verwendet werden kann.
- Train_model: Diese Funktion koordiniert das Training der beiden Classifier mit den jeweiligen Daten.
- __perform_train: Diese Funktion führt das tatsächliche Training durch.

- __Read_ppfile_to_intlist: Diese Funktion liest die vorverarbeiteten Daten aus den Files Zeichen für Zeichen ein und verwandelt sie in eine Liste aus Integer Werten.
- Model_predict: Diese Funktion verwendet das fertige Netz um Daten zu klassifizieren.
- Model_Eval: Diese Funktion ist sehr ähnlich zur Funktion Model_predict aufgebaut, generiert aber automatisch Labels um den Erfolg der Evaluierung zu messen.

Sollte im Hauptprogramm der Debug Modus aktiviert sein, werden in der Klasse umfassende Informationen vor allem zur Evaluierung ausgegeben. So wird für jeden Datensatz ausgegeben wie hoch die Kosten der beiden Classifier für die Rekonstruktion ist, wie sich daher das System entscheidet und was das richtige Ergebnis ist. Außerdem wird am Ende angegeben wie hoch die Kosten über das ganze Evaluierungsset für die beiden Classifier für jeweils beide Datentypen waren. Dadurch lassen sich unter anderem die Aussagekraft des Ergebnisses als auch der einwandfreie Ablauf ableiten. Diese Klasse verwendet ebenfalls ein eigenes Logfile um Hyperparameter und sonstige Informationen zu protokollieren.

3.7.2.5 System

Das ganze System wird über das Hauptprogramm gesteuert. In diesem werden die Command Line Parameter verarbeitet und überprüft, das Logging und die Run ID initialisiert, notwendige Ordner angelegt und Klassen instanziiert. Wenn das Programm über die Command Line mittels des Systems „System“ gesteuert wird, befindet sich außerdem in diesem Hauptprogramm eine Klasse dafür, wodurch das Programm genau wie bei RNN oder Classifier nur über eine Klasse gesteuert werden kann. Diese Klasse regelt intern alle notwendigen Abhängigkeiten, wie beispielsweise, dass die Trainingsdaten zuerst durch das RNN vorverarbeitet werden bevor sie von den Classifiern für das Training genutzt werden. Das System besitzt ebenfalls ein eigenes Logfile in dem Startpunkt des Durchlaufs, Run ID und Start Parameter dokumentiert werden. Außerdem wird im Hauptprogramm auch das Error Logfile und je nach Konfiguration das StdOut Logfile angelegt. Weiter wird jede Fehlermeldung im ganzen Programm bis zum Hauptprogramm zurückverfolgt, wodurch eine gut lesbare Spur vom Hauptprogramm bis zur Fehlerquelle entsteht, erleichtert durch die jeweiligen Fehlercodes. Jeder Fehler führt zu einem Abbruch, um sicherstellen zu können, dass die Daten nicht in der falschen Form oder Reihenfolge verarbeitet werden.

3.7.2.6 Training

Das Training wird wie bereits erwähnt nur für die beiden Classifier durchgeführt. Das RNN muss die Trainingsdaten jedoch vorverarbeiten, bevor sie genutzt werden können. Diese vorverarbeiteten Trainingsdaten werden als Dateien abgespeichert, damit man das Training mit verschiedenen Hyperparametern wiederholen kann, ohne die Vorverarbeitung erneut durchführen zu müssen. Dies ist nur bei Änderungen am RNN notwendig.

3.7.2.7 Ergebnis

Das Ergebnis wird im Logfile der TwoClassifier Klasse ausgegeben. Es steht dort wie viele Daten von welchem Netz korrekterweise und falscherweise erkannt wurden und wie gut die beiden Netze insgesamt die Daten ihrer Kategorie rekonstruieren konnten. Sollte der Debug Modus aktiviert sein werden außerdem umfassendere Informationen zu den Ergebnissen auf die Command Line ausgegeben.

3.7.3 Anwendung

Nach Fertigstellung des Codes wurde das Programm ausführlichen Tests unterzogen, bevor die eigentlichen Versuche starteten. Die für das Training und die Evaluierung verwendeten Daten sind näher im Kapitel 3.7.1 beschrieben. Aufgrund der Art der Daten wurden für einen ersten Durchlauf alle Daten mit einem Label zwischen -0.1 und 0.1 ausgefiltert, da diese sehr nahe an 0 sind und demnach schwerer zu unterscheiden sind, als Daten die näher an -1 oder 1 sind. Durch diese Selektion entstand ein Trainingsset von jeweils 26.000 gutartigen und bösartigen Events, sowie circa 5.000 Daten für die Evaluierung. Die Versuche zeigten jedoch, dass die Größe des Trainingssets zu klein ist, da auch bei einer hohen Anzahl an Epochen keine guten Ergebnisse erzielt

werden konnten. So wurden in einem durchschnittlichen Durchlauf circa 50% der Daten richtig klassifiziert. Daher wurde für darauffolgende Versuche das vollständige Trainingsset von jeweils circa 200.000 Datensätzen verwendet. In weiteren Versuchen wurde außerdem die Größe der Netze stark angehoben, von Anfangs circa 160.000 Neuronen im RNN und 230.000 Neuronen in jeweils beiden Classifiern auf circa 770.000 Neuronen im RNN und circa 1.200.000 Neuronen in jeweils beiden Classifiern. Die Anzahl der Epochen wurde auf 300 erhöht, auch wenn sich meist ab Epoche 200 keine Verbesserung mehr einstellt. Leider bleiben die Ergebnisse durchgehend unbefriedigend und sind eher zufällig verteilt, erkennbar daran, dass die beiden Classifier Daten circa gleich gut rekonstruieren können und daher eher zufällig entscheiden. Es wurden dementsprechend circa 50% der Daten richtig klassifiziert. Dies kann sowohl an der geringen Menge an Trainingsdaten liegen, als auch an der schlechten Verteilung der Trainingsdaten, die zum größten Teil im Wertebereich von -1 bis +1 nahe an Null liegen. Ergebnisse und mögliche Probleme werden näher im Kapitel 4 erläutert. Bedingt durch schlechte Ergebnisse können manche geplanten Versuche mit dem fertigen System nicht durchgeführt werden, da keine Verbesserungen oder Änderungen identifiziert werden können. Diese und weitere Versuche sind daher im Kapitel 5 erwähnt.

Durch die Art des Systems können nahezu alle Daten verarbeitet und klassifiziert werden. Voraussetzung dafür ist einerseits, das Alphabet im Programm um alle in den Daten vorkommenden Zeichen zu ergänzen und andererseits die Trainingsdaten entsprechend zu klassifizieren. Je nach Datenformat sind eventuell kleinere Anpassungen an der Funktion zum Einlesen der Daten notwendig. Theoretisch wäre daher damit auch Spracherkennung in Texten möglich, auch wenn es dafür geeignetere Netze gibt. Der Code des ganzen Programms ist auf der beigelegten CD im Ordner Hauptversuch enthalten.

4 Ergebnisse

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst, die neben der Beschreibung von Grundlagen und der Entwicklung von Verfahren und Systemen auch die Ergebnisse der Durchläufe des Hauptprogramms umfassen. Zuerst wurden in Kapitel 2 Grundlagen beschrieben, wie verschiedene Netz Typen, Optimierungsverfahren, Aktivierungsfunktionen und mehr. Dieser Teil stellt eine wichtige Zusammenfassung für weitere Entwicklungen dar indem er eine einheitliche Grundlage schafft und einen Überblick über die wichtigsten Themen im Bereich neuronale Netze gibt. Im ersten Ansatz in Kapitel 3.5 wurde ein System vorgestellt, das bestehend aus einem Feature Extractor, einem RNN und zwei parallelen Classifiern Daten klassifizieren kann. Die Besonderheit in diesem Ansatz liegt einerseits daran, dass die Classifier jeweils nur auf eine Kategorie trainiert werden, in dieser Arbeit „einseitiges Training“ genannt, und nicht wie üblich auf zwei oder mehr. Dies ermöglicht ein flexibleres, räumlich und zeitlich getrenntes Training der einzelnen Komponenten, was gerade bei der Malware Erkennung viele Vorteile bringt. Außerdem verfügt das vorgestellte System über einen Mechanismus zur gezielten Steuerung des Verhältnisses zwischen False-Positive Rate und Erkennungsrate, was erlaubt das System an den Anwendungsfall und die Umgebung anzupassen. Im zweiten Ansatz in Kapitel 3.6 wurden Schwächen des ersten Ansatzes identifiziert und durch verschiedenste Versuche probiert, diese Schwächen gezielt zu verbessern und Entscheidungen zu untermauern. Details zu diesen Vorversuchen befinden sich in Kapitel 6. Dabei wurde unter anderem mit den Rekonstruktionsnetzen ein konkretes Verfahren entwickelt und getestet, mit dem das einseitige Training ermöglicht wird, da dies, wie praktisch gezeigt wurde, mit herkömmlichen Verfahren nicht möglich war. Weiters wurde dort das System mit mathematischen Verteilungen getestet und es konnte gezeigt werden, dass die Daten wie geplant klassifiziert werden können. Auch die Steuerung der False-Positive Rate und Erkennungsrate konnte praktisch durchgeführt und deren Funktion nachgewiesen werden. Schlussendlich wurde der zweite Ansatz praktisch in Keras mit Tensorflow als Backend umgesetzt und versucht, die Funktion mit tatsächlichen Events von gutartigen und bösartigen Prozessen zu zeigen. Das Ergebnis ist jedoch sehr schlecht, was auf verschiedene Probleme und Einschränkungen zugeführt werden kann, dazu mehr in Kapitel 4.1. Das umgesetzte System ist genauer in Kapitel 6 beschrieben und erlaubt auch die Verarbeitung und Klassifizierung von anderen Textdaten.

Als die zwei großen Ergebnisse dieser Arbeit kann man daher die Grundlangen über neuronale Netze und das System zur Klassifikation von Malware sehen. Die Grundlagen stellen eine Zusammenfassung vieler Netze, Algorithmen, Funktionen und Verfahren rund um das Thema Neuronale Netze dar, etwas Vergleichbares gibt es derzeit (Stand Februar 2018) in der Literatur nicht in aktueller und umfassender Form. Es muss berücksichtigt werden, dass auch diese Arbeit nicht alles beschreibt, sondern sich eher auf für die Arbeit relevante oder interessante Themen fokussiert. Mehr dazu im nächsten Kapitel. Das hier beschriebene System ist die Grundlage für ein System aus neuronalen Netzen, die zusammen fähig sind laufend die Systemevents eines Clients zu analysieren um automatisch auffällige Verhaltensmuster, die auf Angriffe oder Malware hinweisen, zu erkennen. Der Fokus von vielen Arbeiten auf Malware Features wie File Size, Sections, API Calls und ähnlichem hat neben dem Problem des aufwändigen Feature Engineerings und dem dafür notwendigen Wissen auch den Nachteil, dass teilweise Millionen von Features identifiziert werden. Da dies nicht in einer sinnvollen Zeit verarbeitbar ist, werden meistens einzelne Features zufällig ausgewählt und andere verworfen („Random Projection“). Das hier beschriebene System macht Feature Engineering jedoch überflüssig, wodurch einerseits kein umfassendes Domänen Wissen notwendig ist und andererseits ein schlanker und vollständiger Ansatz ermöglicht wird, der keine Eigenschaften auslassen muss. Die fehlende Notwendigkeit für Expertenwissen entsteht dadurch, dass die gutartigen Daten automatisch bei dem Anwender gesammelt werden können, während bösartige Daten zentral beim Hersteller klassifiziert und erstellt werden. Durch die Verwendung von System Events die durch ein System zu Process-Traces zusammengefasst wurden, ist es auch möglich lange laufende Angriffe zu finden, da die einzelnen Schritte im Process-Trace trotzdem eindeutig hintereinander liegen und nicht in anderen Informationen untergehen. Das System würde beispielsweise auch "Malware-Lose" Angriffe erkennen, wie präparierte USB Sticks, die als Tastatur verschiedene Befehle durchführen. Es ermöglicht auch Malware wie den Code Red Wurm zu erkennen, die sich nur im Speicher befindet, womit viele File Ansätze

an ihre Grenzen stoßen. Ein weiterer großer Vorteil von der Analyse der System Events anstatt der API Calls oder des Codes direkt ist, dass veränderte Viren mit gleichbleibender Funktionalität leichter erkannt werden können, da es für den hier beschriebenen Ansatz beispielsweise irrelevant ist, mit welchem API Call oder in welcher Sprache ein File geöffnet wird, das Event ist immer dasselbe. Abgesehen davon soll der Ansatz dadurch langlebiger werden, da es nicht notwendig ist das Netz neu zu trainieren, wenn für eine API ein neuer Befehl hinzugefügt wird, dessen Funktionalität bereits in einer anderen API vorhanden ist. Wie bereits erwähnt ist das System dafür ausgelegt, einen Computer nicht nur durch gezielte und eingeschränkte Tests zu überwachen, sondern laufend, wodurch zusätzliche Angriffsvektoren abgesichert werden. Beispielsweise beim Einsatz herkömmlicher Sandboxes werden oft nur Files untersucht, die über Mail oder Websites heruntergeladen werden, und es wird außer Acht gelassen, dass Files auch über ungeplante Wege wie USB Sticks oder durch sonstige Tricks wie gezippte Files in das Netz gelangen können. Abgesehen davon entwickeln sich Anti-Sandbox Technologien bei Malware stark weiter und die Chance, dass Malware sich während der Analyse ruhig verhält steigt. Durch den in dieser Arbeit beschriebenen Ansatz mit zwei parallelen Netzen ist es außerdem möglich, viel besser auf die Umgebung des Anwenders einzugehen. Anti-Malware Hersteller können das System auf Erkennung von Malware in der Firmen-Umgebung trainieren und liefern das System dann aus. Das Erkennen von Normalzustand wird dem System erst vor Ort mit den individuellen und real auftretenden Events des Anwenders antrainiert. Dadurch, dass Daten aus der Umgebung des Anwenders verwendet werden, soll das System wesentlich besser funktionieren und eine niedrigere False-Positive Rate aufweisen, als wenn generische Auslöser oder Files für den Normalzustand verwendet werden, die bei einem Anwender vielleicht nicht oder zumindest in einer anderen Verteilung auftreten. Abgesehen davon können Updates leichter durchgeführt werden, da der Anti-Malware Hersteller lediglich in seiner Umgebung ein Netz mit neuen Malware-Daten trainieren muss, es an die Anwender ausliefert und diese das alte Malware-Netz durch das neue Netz ersetzen. Das Netz ist dann sofort einsetzbar, weitere Trainings sind nicht notwendig.

4.1 Diskussion über die Ergebnisse

Wie in den Ergebnissen bereits erwähnt sind die Ergebnisse des ersten und zweiten Ansatzes sehr zufriedenstellend und konnten auch durch erfolgreiche Vorversuche bekräftigt werden. Das Resultat des Hauptversuchs mit realen Events war nicht erfolgreich, identifizierte Probleme und Vermutungen werden hier näher beschrieben, aber auch die eingangs beschriebenen Grundlagen müssen diskutiert werden.

Grundlagen

Die in Kapitel 2 beschrieben Grundlagen über neuronale Netze stellen eine in dieser Form noch nicht vorhandene Zusammenfassung dar. Es wurde versucht, die gängigsten Namen zu verwenden und eine Vielzahl an Netzen, Verfahren und Algorithmen zu erklären. Trotzdem fehlen in der Zusammenfassung noch viele Begriffe und viele Zusammenhänge wurden nur kurz beschrieben. So wurde es beispielsweise unterlassen für einen Begriff auch weniger verbreitete Namen zu erwähnen, um die Vereinheitlichung voranzutreiben. Dies führt auch dazu, dass die Grundlagen für manche Personen nicht korrekt wirken, da sie andere Benamungen und Kategorien kennen. Der Bedarf ist nach wie vor sehr groß für eine zusammenfassende wissenschaftliche Literatur, um eine einheitliche Grundlage und einheitliche Begriffe für weitere Forschungen und Entwicklungen zu schaffen und damit auch den Einstieg in die Thematik zu erleichtern.

Verteilung der Daten

Wie im Kapitel 3.7.1 beschrieben, ist jeder Zeile im Datensatz ein Wert zwischen -1 und 1 zugewiesen, wobei -1 für bösartig und 1 für gutartig steht. Bei der Analyse der Verteilung dieser Werte in dem Datensatz zeigte sich, dass über 98% der Daten sehr knapp an 0 liegen, wie in Abbildung 28 zu sehen ist. Im Gegensatz dazu liegen am negativen und positiven Ende circa jeweils 0,005% der Daten. Das System, mit dem die Vorverarbeitung dieser Daten durchgeführt wurde, konnte offensichtlich den Großteil der Daten nicht wirklich klassifizieren, da der Wert 0 quasi für Unentschieden steht. Daraus ergibt sich der Schluss, dass diese Daten ungeeignet sind um damit ein neuronales Netz zu trainieren, da dafür eindeutig unterscheidbare und klassifizierte Daten notwendig sind. Unter Berücksichtigung dieser Probleme relativieren sich auch die schlechten Ergebnisse des Hauptversuchs mit dem in dieser Arbeit beschriebenen System.

Abbildung 28: Verteilung der Werte im Datensatz

Geringe Menge an Daten

Während bei Vorversuchen bis zu drei Millionen Daten für das Training verwendet wurden, umfassten die realen Daten nur circa 400.000 Datensätze zur Verfügung. Dies führt dazu, dass das Netz nicht ausreichend auf eine Kategorie trainiert wird, es liegt Underfitting vor und das Netz kann keine Kategorie eindeutig erkennen. Auch eine Erhöhung der Anzahl der Epochen ändert dieses Problem nicht, da nach einigen Epochen das Netz keine neuen Informationen aus den Daten mehr lernen kann, wie auch versucht wurde. Um grundlegend positive Ergebnisse erzielen zu können sind vermutlich mindestens eine Million Datensätze notwendig.

System nicht optimiert

Zusätzlich lässt sich das schlechte Ergebnis auch zu einem kleinen Teil auf die fehlende Optimierung des Systems zurückführen. Es wurden verschiedenste Hyperparameter für Durchläufe verwendet, eine systematische Optimierung ist aber erst möglich, wenn das System grundlegend funktioniert und soweit trainiert werden kann, dass es eindeutig unterschiedlich gute Ergebnisse liefert. Da dies in diesem Fall durch fehlende Daten nicht möglich war, können auch nicht die Ergebnisse bei Verwendung verschiedener Hyperparameter getestet werden, da alle Konfigurationen durchgehend schlechte Ergebnisse liefern. Abgesehen davon hätte man das System an sich noch optimieren können, zum Beispiel durch Vergleich von verschiedenen Encoding Ansätzen. Dazu mehr in Kapitel 5.

4.2 Probleme und Einschränkungen

Im Zuge der Durchführung dieser Arbeit wurden verschiedenste Einschränkungen dieser Arbeit identifiziert, oft ausgelöst durch unlösbare Probleme.

Probleme

- Ein großes Problem ist die Beschaffung von ausreichend Systemevents von Angriffen und Malware zu Trainingszwecken. Huang und Stokes[6] verwenden bei ihrem Paper über 6,5 Millionen gekennzeichnete Files, die ihnen von Microsoft zur Verfügung gestellt wurden. Solche Daten in großer Menge und guter Qualität zu beschaffen ist sehr schwer und stellte auch in dieser Arbeit ein unlösbares Problem dar.

- Wie auch Benchea et al.[11] erwähnen, ist eine große Herausforderung die Erkennung von Grayware, wie Adware, Spyware, Keygens und ähnliches, die durchaus bösartiges oder zumindest unerwünschtes Verhalten aufweisen können, aber denen grundlegende Merkmale und Aktivitäten von klassischer Malware fehlen. Dadurch werden Aktivitäten von Grayware meist als gutartig eingestuft, auch wenn sie streng genommen bösartig ist.

- Ein weiteres Problem ist die fehlende Hardware für umfangreiche Netze, die ein sehr rechenintensives Training benötigen. Bei einem ähnlichen Paper von Raff et al.[3] wurden pro Netz 8 NVIDIA Tesla P100 GPUs verwendet und selbst da hat laut Angabe das Training viele Tage oder sogar ein Monat benötigt. Das dort verwendete Setup kostet für ein Netz mehr als 45.000€ (Stand April 2018). In dieser Arbeit wurde eine NVIDIA Tesla K40 verwendet, damit dauerten Trainingsdurchläufe auch bei kleineren Netzen bereits über einen Tag.

- Ein allgemeines Problem im Bereich Machine Learning und Deep Learning ist die fehlende zusammenfassende Literatur. Unterschiedliche Bezeichnungen und Kategorisierungen in der Literatur erschweren es, sich einen gründlichen Überblick über die Thematik zu verschaffen. So werden zum Beispiel von Kaffka[20] Feed-Forward Netze als Backpropagation Netze bezeichnet, während diese Bezeichnung sonst kaum gefunden wurde. Das mag daran liegen, dass es keine allgemein anerkannte Definition von neuronalen Netzen in der Literatur gibt, sondern neuronale Netze eher als Oberbegriff zu verstehen sind, siehe Rey und Wender[19]. Daher können alle hier beschriebenen Bezeichnungen, Eigenschaften und Kategorisierungen mitunter in unterschiedlicher Literatur stark abweichen. In vielen Fällen ist es daher empfehlenswert, sich hauptsächlich entsprechende Bezeichnungen, Funktionen und Algorithmen in der Dokumentation des verwendeten Frameworks oder der verwendeten Library anzuschauen, da dies letztendlich für die Umsetzung entscheidend ist.

Einschränkungen

- Dieses System ist nicht bereit für die Verwendung, sondern eher als experimentelles System für weitere Forschung zu betrachten. Es sollte daher auf keinen Fall für produktive Implementierungen verwendet werden, bevor nicht weitere Verbesserungen und Anpassungen vorgenommen wurden.
- Der Input-Vektor ist nur auf Buchstaben, Zeichen und Sonderzeichen auf einer Tastatur mit deutschem Layout beschränkt. Damit funktionieren viele Sonderzeichen und Sprachen nicht. Erweiterungen auf einzelne andere Sprachen sind problemlos durch Erweiterung des Alphabets möglich. Eine Erweiterung auf Unicode würde das System lahmlegen, da sich aus einem Input von 100 Zeichen nach One-Hot Encoding circa 8.000.000 Zeichen ergeben würden.
- Damit die Rekonstruktions-Fähigkeit der Netze verglichen werden kann, müssen die jeweiligen Trainingsdaten circa gleich groß sein und die gleiche Qualität besitzen. Wird beispielsweise ein Classifier mit Daten schlechter Qualität trainiert, wird er im Betrieb Daten immer schlechter rekonstruieren können, als der zweite Classifier. Allerdings ist die Datenqualität nur schwer messbar.

5 Zukünftige Arbeit

Diese Arbeit stellt aufgrund des Umfangs das beschriebene System nur grundlegend vor. Im Zuge der Erarbeitung wurden jedoch eine Vielzahl von weiterführenden Versuchen und Entwicklungen identifiziert, die in zukünftigen Arbeiten behandelt werden können. Auch bedingt durch schlechte Ergebnisse konnten manche weiterführenden Versuche nicht durchgeführt werden, mehr zu den Gründen in Kapitel 4.1. Da diese Arbeit ohnehin nur die Grundlagen für ein System schaffen soll, bedarf es weiterer ausführlicher Forschung um diesen Ansatz weiter zu verbessern, bevor dieser tatsächlich eingesetzt werden kann. Ein großer Punkt wäre beispielsweise die Umsetzung von automatischem Training, durch die das gutartige Netz regelmäßig auf alle gutartigen Events seit dem letzten Training trainiert wurde, wodurch die Vorteile des Systems optimal zur Geltung kommen würden. Weitere Möglichkeiten für Verbesserungen in zukünftigen Arbeiten werden in den folgenden Unterkapiteln zusammengefasst.

5.1 Untrainiertes RNN

Im ersten Ansatz in Kapitel 3.5 wurde angenommen, dass das RNN nicht trainiert werden muss, da es lediglich mehr oder weniger eine Hashing-Funktion erfüllt. Ein untrainiertes RNN zu verwenden kann aber problematisch werden, da bei der Verarbeitung vieler Events zu einem Output alle Teile der Events als gleich wichtig angesehen werden und gleich verarbeitet werden. Genau genommen werden die Teile der Events je nach zufälliger Initialisierung der Gewichtungen als wichtig angesehen. Das heißt, es kann passieren, dass wichtige Teile von Events immer wenig beachtet werden, und unwichtige Teile immer viel beachtet werden. Durch Training wird sonst versucht, dass das Netz wichtige Teile besonders beachtet und über die Hidden States bis zum Output transportiert. Da aber weder klassisches supervised Learning verwendet werden kann, der Output wird ja nicht klassifiziert, aber auch nicht mit unsupervised Learning versuchen kann, dass der Output dem Input möglichst ähnelt, ist es schwierig das RNN sinnvoll zu trainieren. Eine Möglichkeit wäre, es analog zu einem Language Modelling Task darauf zu trainieren die nächsten Zeichen oder Wörter richtig vorherzusagen. Dadurch müsste sich das Netz auf wichtige Unterscheidungsmerkmale zwischen den Events konzentrieren und den Input zu einem aussagekräftigeren Output verarbeiten können als im ersten Ansatz. Auch dies könnte jedoch zu Problemen führen, da die System Events nicht so starren Regeln wie Sprache folgen, wodurch es für ein RNN vermutlich nahezu unmöglich wird, die nächsten Zeichen vorherzusagen. Es existieren auch keine längeren Sequenzen wie in ein Gedicht, auf die das Netz trainiert werden kann. Abgesehen davon ist es eher sinnlos, auf Werte wie IP Adressen und Pfadnamen zu trainieren, da die Vielfalt zu groß ist und die Zusammenhänge oft unbedeutend sind. Daher würde man das Netz vermutlich nur auf einen Bruchteil der vorhandenen Werte trainieren, was ähnliche Ergebnisse wie ein untrainiertes Netz liefert. Wie schlecht ein NLP Modell mit untrainierten Daten abschneidet zeigt folgendes reales Beispiel, in dem ein Netz auf ein Gedicht trainiert wurde und es nachher rekonstruieren soll. Als Negativbeispiel sollte das Netz auch die Sequenz „Hello Worl“ vervollständigen. Der fettgedruckte Teil war jeweils der Input:

Input war Teil des Gedichts: **Sing a son**g of sixpence, A poc → Richtig: 100%
Input war Teil des Gedichts: **king was i**n his counting house → Richtig: 100%
Input war nicht Teil des Gedichts: **hello worl**ki mh as, bor eed th → Richtig: 0%

5.2 Unveränderbares RNN

Wie bereits im ersten Ansatz in Kapitel 3.5 beschrieben, dient der Output des RNNs als Input für den Classifier. Daher darf das RNN während und nach dem Training nie verändert werden, weder Architektur, noch Gewichtungen, da es sonst die Events nach der Veränderung nach einem anderen Schema verarbeiten würde als vor der Veränderung. Da der Classifier aber nach wie vor auf das alte Schema trainiert ist, würde er keine Daten mehr richtig klassifizieren können. Dies hat zur Folge, dass der Classifier mit dem neuen RNN erneut trainiert werden muss. Das bedeutet, dass die Methoden zum Speichern und Laden des Modells samt

Gewichtungen eine hohe Priorität haben und in jedem Fall fehlerfrei funktionieren müssen. Da es durch das System ermöglicht wird, die einzelnen Netze des Classifier-Systems unabhängig und auf verschiedenen Computern zu trainieren, muss darauf geachtet werden, dass die Gewichtsmatrix des RNNs auf allen Systemen ident ist. Der Aufwand, bei Problemen regelmäßig den Betrieb zu unterbrechen und das gesamte Netz neu zu trainieren wäre unverhältnismäßig. Das RNN sollte daher sehr einfach konzipiert sein. Eine theoretische Lösung wäre eine systematische Initialisierung der Gewichtungen des RNNs, die bei jedem Durchgang dieselbe Gewichtsmatrix erstellt. So ein Verfahren existiert bis jetzt noch nicht und würde auch nur ohne Training funktionieren, da ein Training die Gewichtsmatrix verändern würde. Eine Lösung für die Aufhebung dieser Abhängigkeit wurde bis jetzt auch noch nicht gefunden.

5.3 False-Positive-Feedback

Eines der größten Probleme von neuronalen Netzen, wie in der Arbeit mehrfach erwähnt, ist eine hohe Anzahl an False Positives. Daher wäre eine mögliche Verbesserung ein Feedback System einzubauen, das es dem Anwender erlaubt Rückmeldungen zu Alarmen zu geben, die dann bei nachfolgenden Trainings berücksichtigt werden. Das bedeutet, wenn das System Alarm meldet werden dem User die Events angezeigt, die diesen Alarm ausgelöst haben. Der User hat nun die Möglichkeit die Meldung zu ignorieren, auf „Richtig“ oder „Falsch“ zu klicken. Je nach Antwort werden die Events automatisch gelabelt und zum Trainingsset hinzugefügt. Gerade durch die häufigen Trainingszyklen, die das hier vorgestellte System erlaubt, wird das Feedback zeitnahe eingearbeitet. Damit lassen sich in erster Linie False Positives beheben, es wäre aber auch möglich ein Feedback System für False Negatives umzusetzen, dies würde jedoch einen sehr hohen manuellen Aufwand erfordern und ist daher auch für weitere Forschung nicht relevant.

5.4 Quelle der Daten

Wie in der Arbeit beschrieben ist eines der Probleme von Netzen zur Malware Erkennung, dass sie vollständig beim Hersteller trainiert werden und daher für die gutartigen Daten generische Informationen genutzt werden. Da diese generischen Informationen nicht oder mit einer anderen Häufigkeit beim Anwender auftreten, dafür vorkommende gutartige Daten dem System nicht bekannt sind, führt dieses Vorgehen zu einer höheren False-Positive Rate. Das in dieser Arbeit beschriebene System soll dem entgegenwirken, in dem die Netze unabhängig voneinander trainiert werden können und somit das Netz für gutartige Daten leicht vor Ort beim Anwender trainiert werden kann. Aufgrund des Umfangs wurden in dieser Arbeit keine weiteren Versuche zu diesem Punkt gemacht. Um das System aber weiter zu verbessern wäre ein wichtiger Versuch zu zeigen, dass ein Netz trainiert mit generischen gutartigen Daten bei einem Anwender tatsächlich schlechter abschneidet, als ein Netz trainiert mit lokal vorkommenden Daten. Außerdem sollten Versuche durchgeführt werden um zu zeigen, um wie viel die Trainingszeit des Netzes für gutartige Daten in dem vorgestellten System kürzer ist, als die gesamte Trainingszeit in einem herkömmlichen binären Classifier. Dieser Versuch würde vermutlich den Vorteil des Systems unterstreichen, dass die beiden Netze getrennt trainiert werden können und die Trainingszeit für nur ein Netz in einem niedrigen Bereich liegt, der es tatsächlich ermöglicht, das Netz erst vor Ort beim Anwender zu trainieren.

5.5 One Hot Encoding Varianten

In dem beschriebenen System wird das Verfahren One-Hot Encoding verwendet, um Text in Zahlen umzuwandeln, Details dazu in Kapitel 3.1.1. Bei der hier gewählten Variante wurde ein Alphabet erstellt, in dem jedes in den Daten vorkommende Zeichen einmal enthalten ist. Danach wurden die Daten Zeichen für Zeichen umgewandelt. Da auch andere Varianten möglich wären, aber in dieser Arbeit nicht verwendet wurden, wird nachfolgend ein kurzer Vergleich zwischen drei Verfahren angestellt, Zeichen Encoding, Token Encoding und Event Encoding. Die Verfahren unterscheiden sich grundlegend dadurch, was als kleinste Einheit gewählt wird, ein Zeichen, ein Token oder das Event. In zukünftigen Arbeiten könnten Versuche durchgeführt werden um Performance und Ergebnis der drei Varianten zu vergleichen. Wie auch in den folgenden Abschnitten erwähnt wird, muss im Betrieb eine kleinste Einheit, die nicht im Alphabet enthalten ist, verworfen werden und kann nicht verarbeitet werden. Dies hat den Grund, dass dem Netz die One-Hot Darstellung dieser Einheit nicht bekannt

ist, da sie offensichtlich auch nicht im Training vorgekommen ist. Während des Trainings sollte es nicht passieren, dass eine Einheit nicht im Alphabet vorkommt, da ein vollständiges Alphabet für die Trainingsdaten leicht erstellt werden kann.

Zeichen Encoding

Diese Variante wird in dieser Arbeit verwendet, dabei werden alle Zeichen einzeln und voneinander unabhängig kodiert. Bei dieser Variante lernt ein Netz schnell einzelne Muster, da sich die einzelnen Zeichen oft wiederholen und oft in den Daten auftreten. Durch die einzelne Verarbeitung der Zeichen kann es dafür schwierig oder unmöglich sein, Zusammenhänge aus den einzelnen Zeichen herzustellen. Ein Netz zur Texterkennung wird beispielsweise lange brauchen, bis es erste Wörter lernt. Ein großer Vorteil dieser Variante ist, dass man leicht alle möglichen Zeichen in den Daten herausfiltern und daraus ein Alphabet bilden kann. Sollte trotzdem im Betrieb ein Zeichen nicht im Alphabet enthalten sein, gehen wenig Informationen verloren, wenn es ignoriert wird.

Token Encoding

Die Idee hinter dieser Variante ist die Events an gewissen Stellen in einzelne Token zu unterteilen. Als Trennzeichen könnten sich zum Beispiel Sonderzeichen wie Schrägstrich, Punkt oder Leerzeichen eignen. Im Gegensatz zu Zeichen Encoding besteht das Alphabet hier im Idealfall aus allen möglichen Token. Es wird jedoch schwer möglich sein, alle möglichen Token, die auftreten können, zu identifizieren, da die Menge an möglichen Token im Gegensatz zu den Zeichen bereits nahezu unendlich groß ist. Sollte im Betrieb ein Token nicht im Alphabet enthalten sein und muss verworfen werden, ist der Informationsverlust noch vergleichsweise gering, es kann aber bereits die Entscheidung des Classifiers wesentlich verändern. Die Daten werden dann Token für Token kodiert. Diese Tokens kommen je nach Art der Daten bereits weitaus seltener vor, bis das Netz einen Token gelernt hat dauert es länger als bei Zeichen Encoding. Ein großer Vorteil ist allerdings, dass ein Token bereits Zusammenhänge beinhaltet, das Netz muss die einzelnen Zeichen eines Tokens nicht lernen, sondern nur die Zusammenhänge zwischen den Token.

Event Encoding

Als letzte Variante und absoluter Gegensatz zum Zeichen Encoding wäre ein Event Encoding möglich. Hier wird ein ganzes Event als kleinste Einheit gesehen, das Alphabet besteht daher im Idealfall aus allen möglichen Events, die auftreten können. Hier ist das Problem noch größer, möglichst alle Events im Alphabet zu haben, da die Anzahl an möglichen Events und deren Streuung sehr groß ist. Durch die große Streuung werden Events in den Trainingsdaten kaum doppelt vorkommen, es ist daher eine große Herausforderung für das Netz ein Event überhaupt zu lernen. Der große Vorteil liegt jedoch darin, dass alle Zusammenhänge vollständig erhalten bleiben, auch nach dem One-Hot Encoding. Der größte Nachteil ist, dass während des Betriebs vermutlich eine sehr hohe Anzahl an Events nicht verarbeitet werden kann und verworfen werden muss, was unter Berücksichtigung der Umstände, dass es sich um ein System zur Malware Klassifizierung handelt, durchaus kritisch ist.

5.6 Vergleich mit herkömmlichem Classifier

Da sich das in dieser Arbeit beschriebene System vor allem durch seine Vorteile von einem herkömmlichen Classifier abheben will, soll in zukünftiger Arbeit weitere Versuche mit herkömmlichen Classifiern und diesem System durchgeführt werden, um Vorteile und Nachteile zu identifizieren. Man kann beispielsweise das in dieser Arbeit beschriebene System mit einem üblichen binären Classifier vergleichen, vor allem in Hinsicht auf Trainingsgeschwindigkeit und benötigte Trainingsdaten für ein sinnvolles Ergebnis. Dies würde eventuelle Vorteile oder Nachteile des hier beschriebenen Systems hervorheben und Verbesserungsmöglichkeiten aufzeigen.

6 Appendix A: Praktische Tests und Versuche

In diesem Anhang werden verschiedene Versuche durchgeführt um Entscheidungen zu treffen oder zu untermauern. Die Versuche werden größtenteils an der entsprechenden Stelle in der Arbeit referenziert.

6.1 Versuchsreihe 1

Diese Versuchsreihe beschäftigt sich mit „einseitigem“ Training. Darunter wird in dieser Arbeit verstanden, wenn ein Classifier, nicht wie üblich auf mehrere Kategorien trainiert wird, sondern nur auf eine Kategorie. Im Betrieb soll der Classifier dann zeigen, ob Testdaten aus der trainierten Kategorie stammen, oder nicht. Die Vermutung ist, dass ein neuronales Netz mit herkömmlichen Verfahren nicht einseitig trainiert werden kann, da es dann jeden Testinput der trainierten Klasse zuordnet. Siehe zu Details Kapitel 3.6.1. Meist ordnen Classifier Daten einer genauen Kategorie zu, beispielsweise ein Bild einer Tierart. Für viele Anwendungsfälle reicht es aber aus, lediglich eine Kategorie erkennen zu können, beziehungsweise feststellen zu können, dass die Daten nicht aus dieser Kategorie stammen. Als praktisches Beispiel für ein einseitig trainiertes Netz könnte man einen Klassifikator nehmen, der nur mithilfe von Katzen-Bildern trainiert wird und im Betrieb dann für ein Bild aussagen soll, ob es sich um eine Katze handelt oder nicht. Da die in diesen Versuchen verwendeten Daten sehr generisch sind, können die Ergebnisse unter Verwendung von realen Daten theoretisch stark abweichen.

6.1.1 Aufbau

Es wurden zwei Klassen definiert. Eine Klasse erhält Daten aus einer Gaußschen Glockenkurve mit wechselnder Standardabweichung, die andere Klasse erhält Zufalls-Daten aus dem Wertebereich 0 bis 1 mithilfe der Python3.5 random() Funktion. Wie weiter unten bei den Testdaten zu sehen ist, sind die Daten der Gaußschen Glockenkurve symmetrisch um das Maximum angeordnet. Aufbau des Netzes, Hyperparameter und Trainingszeiten werden in Tabelle 3 beschrieben.

Tabelle 3: Aufbau des Netzes, Hyperparameter und Trainingskonfiguration

Netz	
Typ	Feed-forward fully connected
Struktur / Neuronen pro Layer	Input-Hidden1-Hidden2-Output / 5-10-5-1
Kostenfunktion	Mean Squared Error
Optimierungsverfahren	Adam
Aktivierungsfunktion	Tangens Hyperbolicus
Menge Testdaten	200.000 Zeilen zu je 5 Zahlen (siehe Größe Input Layer)
Batchgröße	150
Epochen	6
Durchmischung des Trainingssets pro Epoche	True
Computer	
RAM	8 GB
CPU	2 Kerne
GPU	Keine
Trainingszeit	~10 Sekunden
Durlaufzeit gesamt	~20 Sekunden

6.1.2 Versuch 1

Bei diesem Versuch wurde ein Classifier auf übliche Weise trainiert. Das Netz wurde darauf trainiert, bei Daten aus der Glockenkurve den Output Wert 1 zu generieren, bei Daten aus dem Zufallsgenerator soll es den Output Wert 0 generieren. Durch diesen Versuch soll sich zeigen, ob die Daten generell unterscheidbar und somit

klassifizierbar sind. Dies dient als Grundvoraussetzung für weitere Versuche, denn sollten die Daten nicht mit einem herkömmlichen Classifier klassifizierbar sein, können sie auch durch das in dieser Arbeit beschriebene System nicht klassifiziert werden. Die Testergebnisse bewiesen eindeutig, dass das Netz die Daten sehr gut erkennt, so wurde für die Gaußschen Daten der Wert 0.97 erzeugt, für die Zufallszahlen wurde 0.009 erzeugt.

Test mit Daten aus Gaußkurve
Test-Vektor: [[0.07964133, 0.13833559, 0.14822053, 0.13833559, 0.07964133]]
Ergebnis: [[0.97573]]

Test mit zufälligen Werten
Test-Vektor: [[0.72174441, 0.24513882, 0.74171, 0.94385448, 0.58883685]]
Ergebnis: [[0.00970059]]

6.1.3 Versuch 2

Nach dem positiven Ergebnis aus Versuch 1 wird in Versuch 2 nun die eigentliche Problemstellung der Versuchsreihe bearbeitet. Das Netz wird darauf trainiert, bei Input Daten aus der Gaußschen Kurve im einzigen Output Neuron den Wert 1 zu generieren. Zufallsdaten werden für das Training nicht verwendet. Bei Tests zeigt sich, dass Daten aus der Glockenkurve richtig als solche erkannt werden und ein Wert nahe an 1 produziert wird. Allerdings wird wie erwartet ein sehr ähnlicher Wert auch für Daten aus der Zufallsverteilung generiert. Dies unterstützt die Vermutung, dass einseitiges Training nicht funktioniert, da der Classifier alle Daten der gleichen Kategorie zuordnet.

Test mit Daten aus Gaußkurve
Test-Vektor: [[0.05882338 0.18503906 0.21353919 0.18503906 0.05882338]]
Ergebnis: [[0.9991613]]

Test mit zufälligen Werten
Test-Vektor: [[0.83121414 0.86944438 0.90689918 0.12762873 0.6918754]]
Ergebnis: [[0.9988872]]

6.1.4 Source-Code

Für diesen Versuch wurde Keras mit Tensorflow als Backend verwendet. Der Code ist relativ einfach gehalten und es sind keine Möglichkeiten zum Laden oder Speichern von Modellen vorhanden, da er ausschließlich für den Versuch dienen soll. Der Code befindet sich auf der beigelegten CD unter dem Namen „VV1_einseitiges_Training_und_VV2_inverses_Training.py“.

6.1.5 Fazit

Versuch 1 stellt als Grundvoraussetzung fest, dass die verwendeten Daten eindeutig mit einem Classifier unterscheidbar sind. Auf diesem Ergebnis aufbauend zeigt Versuch 2 eindeutig, dass ein neuronales Netz unter normalen Bedingungen nicht als Klassifikator genutzt werden kann, wenn es mit einseitigem Training nur auf eine Klasse trainiert wird. Nach dem Training erzeugt das Netz für nahezu jeden Input einen hohen Output Wert, die Input Daten werden also kaum noch berücksichtigt.

6.2 Versuchsreihe 2

Diese Versuchsreihe beschäftigt sich mit inversem Training. Der Versuch von dem Verfahren inversem Training ist, ein Netz trotz einseitigem Training so zu trainieren, dass das Netz sinnvolle Klassifizierung durchführen kann. Dadurch soll das in Kapitel 6.1 beschrieben Problem gelöst und einseitiges Training ermöglicht werden. Siehe zu Details Kapitel 3.6.1. Da die in diesen Versuchen verwendeten Daten sehr generisch sind, können die Ergebnisse unter Verwendung von realen Daten theoretisch stark abweichen.

6.2.1 Aufbau

Für diesen Versuch wurde dasselbe Netz wie in Tabelle 3 verwendet. Aus den Trainingsdaten A der Gaußschen Kurve wurde ein zweites Trainingsset B gebildet, bei dem für jeden Wert aus dem Gaußschen Trainingsset A das additive Invers enthalten ist. Wenn im Trainingsset A also der Wert 0,342 vorkommt, gibt es im Trainingsset B den Wert -0,342. Für das Trainingsset A war das Trainingsziel der Wert 1, während es für das Trainingsset B der Wert 0 war. Die beiden Trainingssets wurden verknüpft, durchmischt und das Netz damit trainiert. Das Ziel war, dass beim Test Daten aus der Gaußschen Kurve noch immer den gewünschten hohen Wert erzielen, während Zufallsdaten einen niedrigen Wert erzielen. Das Netz soll also weiterhin den Input verarbeiten und nicht alle Daten der gleichen Kategorie zuordnen.

6.2.2 Versuch 1

Leider konnte diese Vermutung in dieser Form nicht bestätigt werden, ganz im Gegenteil führte es zum umgekehrten Effekt. So führen Daten der Gaußschen Kurve zu einem Output rund um 0.5, die Zufallsdaten führen zu einem Output nahe an 1. Der Effekt scheint jedoch stabil genug um ihn nutzen zu können. In den unten angeführten Trainingsdaten sieht man gut, dass die 5 Werte einer Gaußkurve symmetrisch um den dritten Wert angelegt sind, während die Zufallsdaten unsystematisch wirken. Dieses Ergebnis wirkt durchaus vielversprechend und soll noch eingehender überprüft werden. Der Versuch wurde auch mit anderen Aktivierungsfunktionen und anderen Kostenfunktionen durchgeführt, dies führte jedoch durchgehend zu schlechteren Ergebnissen. Folgende Hyperparameter wurden getestet:

- Struktur: 10-5-1, 25-15-1, 35-20-1
- Input Size: 5
- Datenmenge: 300.000
- Batch Size: 150
- Epochen: 6, 8
- Aktivierungsfunktion: Tangens Hyperbolicus, Sigmoid
- Bias Units: True, False

Test mit Daten aus Gaußkurve
Test-Vektor: [[0.07637295, 0.15160146, 0.16516723, 0.15160146, 0.07637295]]
Ergebnis: [[0.51752734]]

Test mit zufälligen Werten
Test-Vektor: [[0.72344703, 0.13133322, 0.71790152, 0.3656511, 0.3596931]]
Ergebnis: [[0.98076123]]

6.2.3 Versuch 2

Um die Ergebnisse aus Versuch 1 nochmals kritisch zu prüfen, wurde die Dichtefunktion der logistischen Verteilung implementiert. Auch hier wurde zuerst ein Netz auf logistische Daten und Gaußsche Daten trainiert, um sichergehen zu können, dass die Daten grundsätzlich unterscheidbar und somit klassifizierbar sind. Dabei zeigten sich deutlich schlechtere Ergebnisse als in Kapitel 6.1.2, was an der Ähnlichkeit der Verteilungen liegen könnte, jedoch sind die Daten noch eindeutig unterscheidbar.

Test mit Daten aus Gaußkurve
Test-Vektor: [[0.06230742 0.17990386 0.19870666 0.19870666 0.17990386 0.06230742]]
Ergebnis: [[0.80117524]]

Test mit Daten aus logistischer Verteilung
Test-Vektor: [[0.06692858 0.0842059 0.08611729 0.08611729 0.0842059 0.06692858]]
Ergebnis: [[0.11609659]]

Darauf aufbauend wurde dann der Versuch mit inversem Training wiederholt, das Netz wieder auf Gaußsche Daten und deren additivem Invers trainiert. Diesmal wurde das Netz im Zuge der Validierung mit Daten der Gaußschen Kurve und der logistischen Verteilung getestet. Hier hebt sich die logistische Verteilung zwar um einen geringen Wert von circa 0.08 von der Gaußkurve ab, aber der Unterschied ist leider sehr instabil und bei weitem nicht ausreichend.

Test mit Daten der Gaußkurve
Test-Vektor: [[0.03812739 0.21061083 0.24721 0.24721 0.21061083 0.03812739]]
Ergebnis: [[0.52316606]]

Test mit Daten der logistischen Verteilung
Test-Vektor: [[0.07447177 0.12563193 0.13258602 0.13258602 0.12563193 0.07447177]]
Ergebnis: [[0.44487783]]

Unter Berücksichtigung der Ähnlichkeit von Gaußkurve und logistischer Verteilung ist es verwunderlich, dass das Netz die Daten unterscheiden kann. Da es sich hier aber um perfekte Testbedingungen handelt müsste der Unterschied größer sein um den Effekt praktisch nutzen zu können. Die Ähnlichkeit der beiden Verteilungen wird auch in Abbildung 29 gezeigt.

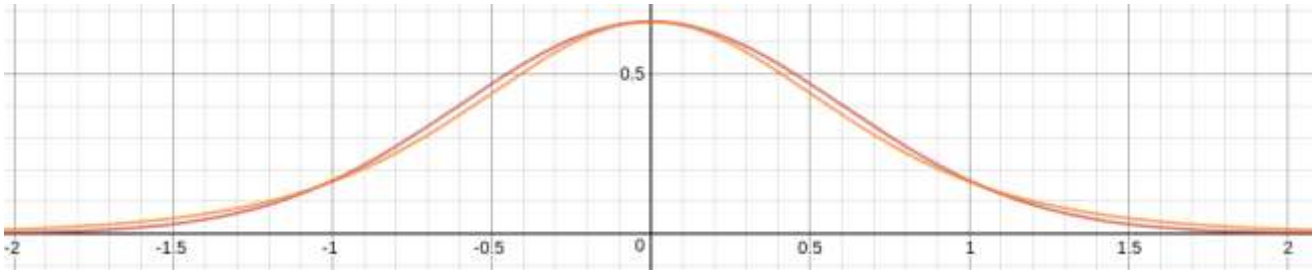

Abbildung 29: Orange ist die logistische Verteilung, rot die Gaußsche Verteilung. Erstellt mit www.desmos.com

6.2.4 Source Code

Es wurde der Source Code aus Kapitel 6.1 erweitert. Für den Versuch 1 wurde der Source Code um eine Funktion ergänzt, die für jeden Wert der Gaußschen Trainingsdaten auch das additive Invers des Werts den Trainingsdaten hinzufügt. Für den Versuch 2 wurde außerdem die Dichtefunktion der logistischen Verteilung implementiert. Der Code befindet sich auf der beigelegten CD unter dem Namen „VV1_einseitiges_Training_und_VV2_inverses_Training.py“.

6.2.5 Fazit

Durch die durchgeführten Versuche konnte gezeigt werden, dass einseitiges Training mithilfe von inversen Trainingsdaten prinzipiell möglich ist. Um das zu verdeutlichen sei nochmals darauf hingewiesen, dass für die Versuche mit inversem Training das Netz nie mit den Zufallsdaten oder den logistischen Daten trainiert wurde, das Netz diese Verteilungen also nicht kennt. Trotzdem erkennt das Netz, wenn Daten definitiv nicht (Versuch 1) oder zumindest eher nicht (Versuch 2) zu der trainierten Kategorie gehören. Mit diesem Vorgehen ist es daher

prinzipiell möglich einseitiges Training durchzuführen, allerdings ist das Ergebnis zu instabil und der Effekt nicht stark genug, um dieses Verfahren weiterhin zu nutzen. Es wurde versucht ein stabileres Ergebnis und eine stärkere Unterscheidung durch eine Vielzahl an verschiedenen Hyperparametern zu erzielen, allerdings waren die Ergebnisse durchgängig schlechter. Um den Effekt nutzen zu können müsste der Unterschied größer ausfallen, außerdem garantiert der Erfolg dieses Versuchs noch nicht, dass dies ebenso mit den reellen Daten möglich wäre.

6.3 Versuchsreihe 3

Diese Versuchsreihe beschäftigt sich mit Rekonstruktionsnetzen. Dieses Verfahren soll helfen das in Kapitel 6.1 beschriebene Problem zu lösen und einseitiges Training zu ermöglichen. Siehe zu Details Kapitel 3.6.1. Da die in diesen Versuchen verwendeten Daten sehr generisch sind, können die Ergebnisse unter Verwendung von realen Daten theoretisch stark abweichen.

6.3.1 Aufbau

Für dieses Netz wird ein fully connected Feed-forward Neural Network verwendet, das aus einem oder mehreren Hidden Layers besteht. Input Layer und Output Layer müssen gleich groß sein. Die Grundidee besteht darin, dass das Netz durch unsupervised Training darauf trainiert wird, den Input einer Kategorie möglichst gut zu rekonstruieren, wodurch es Muster dieser Daten erlernen muss. Die Vermutung ist, dass dieses Netz folglich Daten aus der trainierten Kategorie besser rekonstruieren kann, als Daten einer nicht trainierten Kategorie. Dadurch würde einseitiges Training ermöglicht werden.

6.3.2 Versuch 1

Für Versuch 1 wurden viele Teile aus den Versuchen zu inversem Training übernommen, so werden auch hier die Daten der Gaußschen Kurve und der logistischen Verteilung verwendet. In diesem ersten Versuch wird das Netz darauf trainiert, die Daten der Gaußschen Kurve möglichst gut zu rekonstruieren. Im Anschluss daran wird das Netz mit den Daten der verschiedenen Verteilungen getestet und geprüft, wie gut es den jeweiligen Input rekonstruieren kann. Die Ergebnisse sprechen eindeutig für die Vermutung, es lassen sich gemessen an den Kosten Daten der Gaußschen Verteilung circa 16 Mal besser rekonstruieren als Daten der logistischen Verteilung oder Zufallsdaten. Im Grunde ist die Vermutung damit bestätigt, sie soll aber durch weitere Versuche geprüft werden. Da die Hyperparameter im folgenden Versuch weiter verbessert werden sollen, werden sie hier nicht beschrieben.

Test mit Gaußschen Daten:
Input-Vektor: [[0.10262109 0.11884977 0.1232931 0.1232931 0.11884977 0.10262109]]
Output-Vektor: [[0.10267478 0.1187509 0.12326355 0.12316258 0.11886863 0.10277267]]
Durchschnittliche Kosten: 0.06883316909311857

Test mit logistischen Daten:
Input-Vektor: [[0.08237054 0.09146414 0.09394441 0.09394441 0.09146414 0.08237054]]
Output-Vektor: [[0.08230237 0.09181388 0.09430274 0.09447139 0.09173551 0.0821531]]
Durchschnittliche Kosten: 1.032513121135073

6.3.3 Versuch 2

Dieser Versuch beschäftigt sich mit der Umsetzung der Rekonstruktions-Netze in dem in Kapitel 3.5 beschriebenen System, in dem zwei Classifier denselben Input erhalten und jeweils auf eine andere Kategorie trainiert werden. Durch Vergleich der Rekonstruktions-Kosten der beiden Netze soll festgestellt werden, welcher

Kategorie der Input zuzuordnen ist. Für diesen Versuch wurden die grundlegenden Elemente aus Versuch 1 weiterverwendet. Erste Testergebnisse zeigen, dass das System sehr zuverlässig arbeitet.

Testergebnis eines Durchlaufs

Input: GAUSS
Total Cost GAUSS NET: 0.00015969639083635023
Total Cost LOGISTIC NET: 0.006278321793286912
Entscheidung: GAUSS

Input: LOGISTIC
Total Cost GAUSS NET: 0.018399918250033127
Total Cost LOGISTIC NET: 0.0037654493128961783
Entscheidung: LOGISTIC

Insgesamt wurden von 10000 Gaußschen Daten und 10000 logistischen Daten beispielsweise 8225 Gaußsche Daten und 8255 logistische Daten korrekt klassifiziert:

- Gaußsche Daten korrekterweise als Gaußsche Daten erkannt: 8225 (82,25%)
- Gaußsche Daten falscherweise als logistische Daten erkannt: 1775 (17,75%)
- Logistische Daten korrekterweise als logistische Daten erkannt: 8255 (82,55%)
- Logistische Daten falscherweise als Gaußsche Daten erkannt: 1745 (17,45%)

6.3.4 Versuch 3

In diesem Versuch soll das System noch um die in Kapitel 3.5 geplanten Faktoren erweitert werden, um das Ergebnis tendenziell gezielt in die eine oder andere Richtung lenken zu können. Dafür werden die Kosten jedes Netzes mit einem Faktor multipliziert und dann dieses Ergebnis verglichen. Der Faktor ist standardmäßig 1. Wenn beispielsweise der Faktor für das Gaußsche Netz auf 0.8 geändert wird, sollten mehr Daten als Gaußsche Daten klassifiziert werden, sowohl korrekterweise als auch falscherweise. Anhand der Ergebnisse in Versuch 2 sieht man circa, wie die Erkennung bei einem Faktor von 1 verteilt ist, jedes Netz erkennt dabei circa 82% richtig und 18% falsch. Durch eine Änderung des Faktors des Gaußschen Netzes auf 0.8 wurden beispielsweise folgende Ergebnisse erzielt:

- Gaußsche Daten korrekterweise als Gaußsche Daten erkannt: 8702 (87,02%)
- Gaußsche Daten falscherweise als logistische Daten erkannt: 1298 (12,98%)
- Logistische Daten korrekterweise als logistische Daten erkannt: 7355 (73,55%)
- Logistische Daten falscherweise als Gaußsche Daten erkannt: 2645 (26,45%)

Dies zeigt, dass durch die Änderung des Faktors einerseits mehr Gaußsche Daten korrekt erkannt wurden, allerdings auch mehr logistische Daten falscherweise als Gaußsche Daten klassifiziert wurden. Umgelegt auf Malware entspricht das circa dem Kompromiss, dass eine niedrigere False Positive Rate für den Nachteil einer niedrigeren Erkennungsrate erzielt werden kann. Um die korrekte Funktion sicherzustellen, wurde der Test auch umgekehrt wiederholt und der logistische Faktor auf 0.8 gesetzt, während der Gaußsche Faktor 1 beträgt. Dies führte erwartungsgemäß zum gegenteiligen Effekt, wie die folgenden Ergebnisse zeigen:

- Gaußsche Daten korrekterweise als Gaußsche Daten erkannt: 6830 (68,30%)
- Gaußsche Daten falscherweise als logistische Daten erkannt: 3170 (31,70%)
- Logistische Daten korrekterweise als logistische Daten erkannt: 8924 (89,24%)

- Logistische Daten falscherweise als Gaußsche Daten erkannt: 1076 (10,76%)

6.3.5 Versuch 4

Um den Effekt noch weiter zu verstärken und auszutesten wurde das Modell aus Versuch 1 systematisch mit den nachstehenden Hyperparametern getestet. Dabei wurde das Modell automatisiert mit jeder möglichen Kombination dieser Hyperparameter gebaut, trainiert und getestet, insgesamt über 2.000 verschiedene Konfigurationen. Ziel soll vor allem sein, die Konfiguration mit den niedrigsten Kosten zu finden, also das Netz, welches den Input am besten rekonstruieren konnte. Außerdem soll auch die Konfiguration mit dem größten Kosten-Unterschied zwischen den trainierten Daten und den untrainierten Daten zu finden, da diese Konfiguration die unterschiedlichen Datentypen am besten abgrenzen kann. Mit der in Tabelle 3 beschriebenen Konfiguration betrug die Durchlaufzeit allerdings circa 5 Tage.

Hyperparameter

Es wurden alle möglichen Kombinationen der folgenden Hyperparamter getestet:

- Struktur: 10-5-5-10, 10-15-10-10, 10-10-10-10, 10-20-15-10, 10-30-30-10
- Aktivierungsfunktion: Tangens Hyperbolicus, Sigmoid, RELU
- Optimierungsalgorithmen: ADAM, SGD
- Datenmenge: 100.000, 500.000, 1.000.000
- Bias- Units: True, False
- Dropout: 0.0, 0.2, 0.5
- Kostenfunktion: Mean Squared Error, Mean Absolute Error
- Epochen: 5, 10
- Batch Größe: 150

Die Hyperparameter in der nachstehenden Tabelle 4 erzielten dabei die niedrigsten Kosten. Das Netz erzielte dabei für die Verarbeitung von 500 Gaußschen Testdaten Kosten von circa 0,008. Außerdem konnte das Netz die Gaußschen Daten circa 1,38 Mal besser rekonstruieren als die logistischen Daten. Die extrem niedrigen Kosten liegen vermutlich daran, dass das Netz nicht alle Input Daten verarbeiten und abspeichern kann, da die beiden Hidden Layers kleiner als der Input und die Output Layer sind. Weiter ist das Optimierungsverfahren SGD dafür bekannt, nur langsam ein globales Minimum zu finden. Zusammen mit der Tatsache, dass 100.000 Testdaten sehr wenig sind, hat das Netz vermutlich noch kein Minimum gefunden und kann daher gut generalisieren, was zu geringen Kosten führt. Jedoch sind niedrige Kosten alleine nicht die wichtigste Eigenschaft der Netze in diesem System, wie bereits weiter oben beschrieben wurde.

Tabelle 4: Hyperparameter für die niedrigsten Kosten im Zuge der Versuche.

Netz	
Typ	Feed-forward Neural Network
Struktur	10-5-5-10
Initialization	Xavier
Kostenfunktion	Mean Absolute Error
Optimierungsverfahren	SGD
Aktivierungsfunktion	Tangens Hyperbolicus
Menge Testdaten	100000
Batchgröße	150
Epochen	5
Durchmischung des Trainingssets pro Epoche	True

Die in der Tabelle 5 beschriebenen Hyperparameter erzielten laut den Testergebnissen das beste Verhältnis zwischen den beiden verwendeten Datentypen, den Gaußschen Daten und den logistischen Daten. Das Netz

erzielte für die Verarbeitung von 500 Gaußschen Testdaten Kosten von circa 0,277 und konnte die Gaußschen Daten beeindruckender Weise circa 98,3 Mal besser rekonstruieren als logistische Daten. Hier wird sich im Vergleich zum Netz mit niedrigen Kosten genau der gegenteilige Effekt eingestellt haben, so besitzt das Netz Hidden Layers mit mehr Units als die Input Layer hat und kann dadurch sehr viele Informationen abspeichern. Das Optimierungsverfahren Adam ist außerdem ein sehr gutes und modernes Optimierungsverfahren und führt schnell zum globalen Minimum. Kombiniert mit der Tatsache, dass die Konfiguration mit 1.000.000 Testdaten und 10 Epochen die jeweils höchsten Werte dieser beiden Hyperparameter verwendet, liegt die Vermutung nahe, dass sich das Netz sehr stark spezialisiert hat und vermutlich Overfitting vorliegt. Das Netz ist perfekt auf die Testdaten trainiert, kann daher auch neue Daten derselben Kategorie nur unter höheren Kosten als vergleichbare Konfigurationen erkennen. Da jedoch wie bereits erwähnt vor allem wichtig ist, dass das Netz die Kategorie, auf die es trainiert wurde, wesentlich besser rekonstruieren kann als alle anderen Kategorien, ist diese Konfiguration die für den Anwendungsfall geeignetste. Sollte sich bei weiteren Versuchen herausstellen, dass das Overfitting zu stark auftritt und alle neuen Daten gleich schlecht rekonstruiert werden, kann man auf eine Konfiguration mit einem guten Verhältnis zwischen niedrigen Kosten und gutem Unterschied zurückgreifen.

Tabelle 5: Hyperparameter für den größten Kosten-Unterschied zwischen den Datentypen im Zuge der Versuche.

Netz	
Typ	Feed-forward Neural Network
Struktur	10-20-15-10
Initialization	Xavier
Kostenfunktion	Mean Absolute Error
Optimierungsverfahren	ADAM
Aktivierungsfunktion	Sigmoid
Menge Testdaten	1000000
Batchgröße	150
Epochen	10
Durchmischung des Trainingssets pro Epoche	True

6.3.6 Versuch 5

Für diesen Versuch wird das vollständige geplante System verwendet, bestehenden aus einem RNN zur Vorverarbeitung und zwei parallelen Classifiern zur Entscheidung. Als Daten werden abermals Daten aus der Gaußschen Kurve und der logistischen Verteilung verwendet. Die Hyperparameter wurden angelehnt an das Ergebnis von Versuch 4, auch wenn die dort Hyperparameter direkt für die mathematischen Daten optimiert sind und nicht für eine vorverarbeitete Form. In ersten Durchläufen zeigte sich jedoch, dass das Netz deutlich größer ausgelegt werden muss, da die Daten sehr homogen sind. Durch die Verkettung als String sind sie nur noch ein schwer unterscheidbarer Text-Strom und es gibt kaum Steuerzeichen oder Muster in dem Datenfluss, die einzelne Wörter voneinander trennen. Das RNN hat daher 300 Hidden Units und 280 Output Units. Die Classifier besitzen jeweils 280 Input Units, einen Hidden Layer mit 300 Units und einen Hidden Layer mit 280 Units. Aufgrund der Trainingsart muss der Output Layer dieselbe Größe wie der Input Layer, also 280 Units, haben. Es werden immer 10 Messpunkte berechnet und als Text aneinandergehängt, getrennt durch Leerzeichen. Dieser String wird danach mit One-Hot Encoding kodiert und als Input für das RNN verwendet. Das Ergebnis des RNNs dient dann als Input für die beiden Classifier, die daraus erkennen sollen, ob es sich um Daten der Gaußschen Kurve oder der logistischen Verteilung handelt. Für das Training der Classifier müssen die Daten ebenfalls durch das RNN vorverarbeitet werden. Es wurden insgesamt 3 Millionen Testdaten über 30 Epochen trainiert. Das Ergebnis lag in einem sinnvollen Bereich, circa 85% aller Daten wurden richtig erkannt. Die Komplexität und Durchlaufzeit sind jedoch wesentlich höher als bei allen bisherigen Netzen, so dauerte ein Training circa 25 Stunden. One-Hot Encoding auf Basis der Zahlen statt der Ziffern würde theoretisch noch bessere Ergebnisse liefern, da die Zahlen aber mit 10 Nachkommastellen generiert werden kann es theoretisch

10 Milliarden verschiedene Zahlen geben, der One-Hot Vektor für einen Datensatz wäre also 10 Mal 10 Milliarden groß, was zu einem sehr hohen Rechenaufwand führt und daher unpraktikabel ist.

Testergebnis

- Gaußsche Daten korrekterweise als Gaußsche Daten erkannt: 4562 (91,24%)
- Gaußsche Daten falscherweise als logistische Daten erkannt: 438 (8,76%)
- Logistische Daten korrekterweise als logistische Daten erkannt: 4298 (85,96%)
- Logistische Daten falscherweise als Gaußsche Daten erkannt: 702 (14,04%)

6.3.7 Source Code

Für diese Versuche wurde Keras mit Tensorflow als Backend verwendet. Versuch 1 ist in einem eigenen Programm umgesetzt, der Code ist sehr kurzgehalten. Versuch 2 und Versuch 3 bauen aufeinander auf und sind ebenfalls in einem eigenen Programm umgesetzt, das auch Funktionen wie Speichern und Laden von Modellen unterstützt und vollständig über CLI Argumente gesteuert werden kann. Das Programm gibt auch umfassende Metriken zur Performance des Netzes aus und kommt dem endgültigen System sehr nahe. Versuch 4 ist in einem sehr stark reduzierten Programm umgesetzt, da hier der Fokus ausschließlich darauf liegt, möglichst schnell viele Modelle zu trainieren. Der Code dieser Programme befindet sich auf der beigelegten CD im Ordner VV3.

6.3.8 Fazit

In Versuch 1 wurde gezeigt, dass ein Netz Daten einer Kategorie mit niedrigeren Kosten erkennen kann, wenn es auf diese Kategorie trainiert wurde, als wenn es nicht auf diese Kategorie trainiert wurde. Durch diesen erfolgreichen Versuch kann der Ansatz für das Classifier-System verwendet werden. Danach wurde der Ansatz auf das Zwei-Classifier System angewendet und gezeigt, dass sich mit diesem System hervorragend zwei Kategorien automatisiert erkennen lassen. In Versuch 3 wurde dieses System um Faktoren ergänzt, mit denen sich Erkennungsraten steuern lassen. Um den Effekt dieses Verfahrens noch weiter zu verstärken und zuverlässiger zu machen, wurde in Versuch 4 das Ziel verfolgt, optimale Hyperparameter zu finden, was durch umfassende und aufwendige Tests versucht wurde. Damit wurde sichergestellt, dass der Effekt sehr ausgeprägt und stabil ist und für weitere Netze in dieser Arbeit verwendet werden kann. In Versuch 5 wurde versucht, das vollständige geplante System inklusive RNN zur Vorverarbeitung der mathematischen Daten zu verwenden. Es wurde gezeigt, dass der Rechenaufwand stark ansteigt, das System aber durchaus gute Ergebnisse liefert. Dieser Versuch ist eine gute Grundlage um die Komplexität und den Rechenaufwand des Hauptprogramms zu schätzen.

6.4 Versuchsreihe 4

Diese Versuchsreihe beschäftigt sich mit Deep Belief Networks in Keras. Da es kein vorgefertigtes Modul für DBNs in Keras gibt, diese aber theoretisch gute Ergebnisse versprechen, wird ein DBN programmiert und getestet um vor allem im Gegensatz zu einem normalen Feed-forward Neural Network eventuelle Vorteile festzustellen. Nähere Überlegungen dazu in Kapitel 3.6.1. Da die in diesen Versuchen verwendeten Daten sehr generisch sind, können die Ergebnisse unter Verwendung von realen Daten theoretisch stark abweichen.

6.4.1 Aufbau

Zum Trainieren und Testen werden, wie in anderen Versuchen, Daten der logistischen Verteilung und der Gaußschen Kurve verwendet. Da es wie bereits erwähnt kein fertiges Modul dafür in Keras gibt, werden in diesen Versuchen für ein DBN mit 4 Layers zuerst drei fully connected Feed-forward Neural Networks mit jeweils zwei Layers gebaut. Nachdem diese auf dieselbe Art trainiert wurden, wie das Pre-Training bei DBNs passiert,

werden diese drei Netze zu einem einzigen Netz zusammenfügt. Es entspricht dann einem DBN nach dem Pre-Training. Die drei Netze bekommen jeweils in der Weise Input, dass sie quasi seriell hintereinander trainiert werden, wie in Tabelle 6 dargestellt wird. Das Training ist unsupervised, diese Trainingsart basiert meistens darauf, die Inputdaten möglichst gut im Output zu rekonstruieren. In diesem Fall ist dies jedoch anders, Input und gewünschter Output sind nur im ersten Netz ident. Für die anderen Netze besteht der Input jeweils aus einem bereits durch die vorhergehenden, fertig trainierten Netze, vorverarbeiten Ergebnis von train_data. Der gewünschte Output ist jedoch für alle Netze train_data. Durch dieses Vorgehen läuft das Training wie in einem zusammenhängenden DBN ab. Eine allgemeine Beschreibung zum Pre-Training in DBNs befindet sich in Kapitel 2.3.1.2.1.

Tabelle 6: Inputdaten, Trainingsziel und Outputdaten der einzelnen Netze

Netz 1	Input	train_data
	Gewünschter Output	train_data
	Process train_data → train_data_step1	
Netz 2	Input	train_data_step1
	Gewünschter Output	train_data
	Process train_data_step1 → train_data_step2	
Netz 3	Input	train_data_step2
	Gewünschter Output	train_data
	Predict(train_data)	

Nachdem das Training für alle Layers abgeschlossen ist, wird ein fully connected Feed-forward Neural Network mit 4 Schichten erstellt und die jeweiligen Gewichtsmatrizen aus den einzelnen Netzen übertragen. Für die Gewichtsmatrix Input Layer-Hidden Layer1 werden die Gewichtungen aus Netz 1 genommen, für die Gewichtsmatrix Hidden Layer1 – Hidden Layer2 werden die Gewichtungen aus Netz 2 genommen und für Hidden Layer 2 – Output Layer werden die Gewichtungen aus Netz 3 genommen. Das Vorgehen soll anhand der Tabelle 7 leichter verständlich sein.

Tabelle 7: Zusammenhang zwischen vortrainierten Netzen, Gewichtsmatrizen und FFNN Layers

Vortrainierte Netze	Gewichtsmatrizen	FNN Layers
Netz 1	Matrix 1	Input Layer
		Hidden Layer 1
Netz 2	Matrix 2	
		Hidden Layer 2
Netz 3	Matrix 3	
		Output Layer

Am Ende wurde die korrekte Übertragung der Gewichtungen dadurch überprüft, dass derselbe Datensatz von allen drei Netzen seriell verarbeitet wurde und vom DBN verarbeitet wurde, das Ergebnis war ident.

6.4.2 Versuch 1

Da normalerweise dieses spezielle Trainingsverfahren bei DBNs als unsupervised Pre-Training für ein danach folgendes supervised Training verwendet wird, wird dieses Vorgehen hier abgewandelt, da kein supervised Training notwendig beziehungsweise möglich ist. Es ergeben sich zwei Möglichkeiten: Das Netz direkt nach dem Pre-Training verwenden oder das ganze Netz nochmals mit unsupervised Training weitertrainieren. Dieser Versuch dient dazu herauszufinden, welches Vorgehen zu besseren Ergebnissen führt. Die verwendeten Hyperparameter werden in Tabelle 8 dargestellt.

Tabelle 8: Aufbau und Hyperparameter des DBNs

Netz	
Typ	DBN (3 zusammengefügte FFNNs)
Struktur	5-5-5-5
Initialization	Xavier
Kostenfunktion	Mean Squared Error
Optimierungsverfahren	ADAM
Aktivierungsfunktion	Tangens Hyperbolicus
Menge Testdaten	300000
Batchgröße	150
Epochen	7
Durchmischung des Trainingssets pro Epoche	True

Die Ergebnisse zeigen, dass es kaum einen Unterschied zwischen den beiden Vorgehen gibt. Es wurden beide Netze mit Gaußschen Daten trainiert und mit jeweils 1000 Gaußschen Daten und 1000 logistischen Daten getestet. Mit den Gaußschen Daten soll die Rekonstruktionsfähigkeit überprüft werden, während durch die logistischen Daten sichergestellt wird, dass das Netz nicht zu stark generalisiert. Beide Netze erreichen dabei ein ähnliches gutes Ergebnis, in beiden Fällen erreichen auch wie gewünscht die logistischen Daten deutlich niedrigere Werte. Als Beispiel für die Rekonstruktionsfähigkeit wurde dem Ergebnis pro Netz und pro Daten Art ein Input Vektor und das erzeugte Ergebnis hinzugefügt. Da die Performance durch weiterführendes unsupervised Training kaum besser wird, aber Aufwand und Gefahr von Overfitting steigen, wird das DBN direkt nach dem Pre-Training verwendet.

Ergebnis

Pretraining only Model:

Test mit Gaußschen Daten:
Input Vektor: [[0.07679624 0.15028589 0.16344268 0.15028589 0.07679624]]
Ergebnis: [[0.07688743 0.1502006 0.16383462 0.15013139 0.07682167]]
Durchschnittliche Kosten: 0.5639401343336149

Test mit logistischen Daten:
Input Vektor: [[0.07201432 0.09931373 0.10364241 0.09931373 0.07201432]]
Ergebnis: [[0.07198341 0.09964842 0.10360822 0.09963159 0.07198457]]
Durchschnittliche Kosten: 4.688453937368583

Additional Training Model:

Test mit Gaußschen Daten:
Input Vektor: [[0.07528963 0.1547045 0.16927738 0.1547045 0.07528963]]
Ergebnis: [[0.07542691 0.15458909 0.16939388 0.15454891 0.07536919]]
Durchschnittliche Kosten: 0.5404262925417627

Test mit logistischen Daten:
Input Vektor: [[0.07462036 0.12125089 0.12957831 0.12125089 0.07462036]]
Ergebnis: [[0.0746827 0.12172858 0.12972571 0.12168805 0.0745974]]
Durchschnittliche Kosten: 4.171802531272271

6.4.3 Versuch 2

Da im ersten Ansatz ein FFNN implementiert wurde und dies eventuell durch ein DBN ersetzt werden soll, geht es vor allem um einen Vergleich der Performance beider Netze. Zur Evaluierung ob das DBN oder ein Feed-forward Neural Network besser abschneidet werden beide Netze mit denselben Hyperparametern ausgestattet und erhalten denselben Input. Sie werden also unter anderem mit gleich vielen Testdaten über gleich viele Zyklen trainiert. Durch Vergleich der Ergebnisse soll dann festgestellt werden, welches Netz bessere Ergebnisse liefert. Ein Unterschied, der sich auf jeden Fall beobachten lässt, ist die weitaus längere Trainingszeit des DBNs. Dadurch, dass dafür drei einzelne Netze trainiert werden ist das Verfahren ineffizienter und dauert circa 6 Sekunden pro Zyklus laut Keras Angabe, statt 3 Sekunden wie das FFNN. Da sich auch nach mehreren Durchläufen kaum stabile Ergebnisse eingestellt haben wurde die Anzahl an Test-Daten erhöht und jedes Netz mit 10000 Gaußschen Daten und 10000 logistischen Daten getestet. Die daraus resultierenden Ergebnisse sprechen knapp aber doch für das FFNN, denn während die Kosten für die trainierten Gaußschen Daten gering höher sind als beim DBN, sind die Kosten für die logistischen Daten deutlich höher als beim DBN, was wichtig ist damit das Netz zwischen den beiden Datentypen unterscheiden kann. Denn im Grunde geht es nicht darum, die trainierte Kategorie möglichst gut zu erkennen, sondern sie immer wesentlich besser zu erkennen als jede andere Kategorie. Deswegen und aufgrund der einfacheren Erstellung von FFNNs werden in dieser Arbeit weiterhin FFNNs statt DBNs verwendet werden.

Ergebnis
Testing FFNN:
Test mit Gaußschen Daten:
Input Vektor: [[0.07096476 0.16478657 0.18308666 0.16478657 0.07096476]]
Ergebnis: [[0.07088961 0.16448341 0.18316524 0.16493525 0.07096101]]
Durchschnittliche Kosten: 0.6409545827916464

Test mit logistischen Daten:
Input Vektor: [[0.07139889 0.09696312 0.1009714 0.09696312 0.07139889]]
Ergebnis: [[0.07137123 0.09648073 0.10176937 0.09761301 0.07127587]]
Durchschnittliche Kosten: 5.410763651248699

Testing DBN:
Test mit Gaußschen Daten:
Input Vektor: [[0.08058169 0.12237803 0.12893975 0.12237803 0.08058169]]
Ergebnis: [[0.08056125 0.12236252 0.12906158 0.12240242 0.08054851]]
Durchschnittliche Kosten: 0.5728804802609228

Test mit logistischen Daten:
Input Vektor: [[0.05924348 0.17446886 0.20570247 0.17446886 0.05924348]]
Ergebnis: [[0.05935863 0.17322671 0.2020512 0.17835702 0.05623131]]
Durchschnittliche Kosten: 4.02799151670934

6.4.4 Source Code

Für diese Versuche wurde Keras mit Tensorflow als Backend verwendet. Beide Versuche sind im selben Programm umgesetzt, das neben der Erstellung, dem Training und der Vereinigung der einzelnen Netze auch Test-Funktionen beinhaltet. Der Code befindet sich auf der beigelegten CD unter dem Namen „VV4_dbn.py“.

6.4.5 Fazit

In Versuch 1 wurde gezeigt, dass das erstellte DBN direkt nach dem Pre-Training verwendet werden kann, ohne ein weiteres Training durchlaufen zu müssen. Dies erspart Zeit und senkt außerdem die Gefahr von Overfitting. Versuch 2 führt allerdings zu dem Ergebnis, dass DBNs zumindest in der für diese Arbeit programmierten Form knapp schlechter sind als FFNNs, abgesehen von längerer Trainingszeit und komplexerem Aufbau. Das Ergebnis ist deutlich genug um zu entscheiden, dass daher auch künftig FFNNs verwendet werden.

7 Glossar

ADAM	-	Adaptive Momentum Estimation, Optimierungsverfahren
AI	-	Artificial Intelligence, Englisch für künstliche Intelligenz
ANN	-	Artificial Neural Network, Bezeichnung für neuronales Netz
APAC	-	Asia Pacific, Wirtschaftsraum
API	-	Application Programming Interface, Programm-Schnittstelle
ART	-	Adaptive Resonanz Theorie, Typ von neuronalem Netz
BPTT	-	Backpropagation Through Time, Abwandlung von Backpropagation für RNNs
CEC	-	Constant Error Carrousel, Verbesserungsalgorithmus für LSTMs
CNN	-	Convolutional Neural Network, Typ von neuronalem Netz
CNTK	-	Microsoft Cognitive Toolkit, Framework für neuronale Netze
CPU	-	Core Processing Unit, Recheneinheit im Computer
CSN	-	Constraint Satisfaction Network, Typ von neuronalem Netz
DBN	-	Deep Belief Network, Typ von neuronalem Netz
DL4J	-	Deeplearning4J, Framework für neuronale Netze
DNN	-	Deep Neural Network, Bezeichnung für neuronales Netz mit mehr als zwei Layers
EMEA	-	Europe Middle-East Africa, Wirtschaftsraum
ESN	-	Echo State Network, Typ von neuronalem Netz
FFNN	-	Feed-forward Neural Network, Typ von neuronalem0 Netz
GPU	-	Graphics Processing Unit, Grafik-Recheneinheit im Computer
GRU	-	Gated Recurrent Unit, Typ von neuronalem Netz
HIDS	-	Host Intrusion Detection System, zur Erkennung von Angriffen auf Endgeräten
KI	-	Künstliche Intelligenz, Bezeichnung für eine Art von modernen Algorithmen
LSTM	-	Long Short-Term Memory, Typ von neuronalem Netz
MCDNN	-	Multi Column Deep Neural Network, Typ von neuronalem Netz
MLP	-	Multi-Layer Perceptron, Typ von neuronalem Netz
NN	-	Neural Network, Englisch für neuronales Netz
OC	-	Optical Character Recognition (optische Zeichenerkennung)
ONNX	-	Open Neural Network Exchange, Format für neuronale Netze
OSC	-	One Side Class, Algorithmus für neuronale Netze
OSP	-	One Side Perceptron, Typ von neuronalem Netz
RBM	-	Restricted Boltzmann Machine, Typ von neuronalem Netz
RNN	-	Recurrent Neural Network, Typ von neuronalem Netz
RTRL	-	Real-Time Recurrent Learning, Verbesserungsalgorithmus für RNNs
SDG	-	Stochastic Gradient Descent, Verbesserungsalgorithmus für NNs
SRN	-	Simple Recurrent Network, Typ von neuronalem Netz
TBPTT	-	Truncated BPTT, Abwandlung von BPTT
TPU	-	Tensor Processing Unit, spezielle Recheneinheit für Tensors
VM	-	Virtual Machine, Englisch für virtuelle Maschine

8 Literaturverzeichnis

[1] Mandiant, “M-Trends 2018 Special Report.” [Online]. Available: https://www.fireeye.com/content/dam/collateral/en/mtrends-2018.pdf. [Accessed: 05-May-2018].

[2] W. Lee, S. J. Stolfo, and K. W. Mok, “A data mining framework for building intrusion detection models,” in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, 1999, pp. 120–132.

[3] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, “Malware Detection by Eating a Whole EXE,” *ArXiv Prepr. ArXiv171009435*, 2017.

[4] DeepInstinct, “Malware classification whitepaper.pdf.” [Online]. Available: http://blog.deepinstinct.com/2018/01/08/classifying-malware-using-deep-learning/. [Accessed: 05-Feb-2018].

[5] Invincea, “X_by_Invincea_Datasheet.pdf,” 2018. [Online]. Available: https://www.invincea.com/datasheets/x-by-invincea-datasheet/. [Accessed: 05-Feb-2018].

[6] W. Huang and J. W. Stokes, “MtNet: a multi-task neural network for dynamic malware classification,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016, pp. 399–418.

[7] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, “Why Does Unsupervised Pre-training Help Deep Learning?,” *J Mach Learn Res*, vol. 11, pp. 625–660, Mar. 2010.

[8] K. Rieck, P. Trinius, C. Willems, and T. Holz, “Automatic analysis of malware behavior using machine learning,” *J. Comput. Secur.*, vol. 19, no. 4, pp. 639–668, 2011.

[9] D. C. Ciresan, U. Meier, J. Masci, L. Maria Gambardella, and J. Schmidhuber, “Flexible, high performance convolutional neural networks for image classification,” in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, 2011, vol. 22, p. 1237.

[10] Ah-Hwee Tan, Ning Lu, and Dan Xiao, “Integrating Temporal Difference Methods and Self-Organizing Neural Networks for Reinforcement Learning With Delayed Evaluative Feedback,” *IEEE Trans. Neural Netw.*, vol. 19, no. 2, pp. 230–244, Feb. 2008.

[11] R. Benchea and D. T. Gavriluţ, “Combining Restricted Boltzmann Machine and One Side Perceptron for Malware Detection,” in *Graph-Based Representation and Reasoning*, vol. 8577, N. Hernandez, R. Jäschke, and M. Croitoru, Eds. Cham: Springer International Publishing, 2014, pp. 93–103.

[12] C. Vatamanu, D. Cosovan, D. Gavrilut, and H. Luchian, “A comparative study of malware detection techniques using machine learning methods,” *World Acad. Sci. Eng. Technol. Int. J. Comput. Inf. Eng.*, vol. 2, no. 5, 2015.

[13] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, “Malware classification with recurrent networks,” in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, 2015, pp. 1916–1920.

[14] R. Moskovitch *et al.*, “Detection of unknown computer worms activity based on computer behavior using data mining,” in *Computational Intelligence in Security and Defense Applications, 2007. CISDA 2007. IEEE Symposium on*, 2007, pp. 169–177.

[15] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, “Data mining methods for detection of new malicious executables,” in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, 2001, pp. 38–49.

[16] M. Wastian, “Abnormal event detection by using data mining and machine learning methods,” *Wien, Techn. Univ., Dipl.-Arb., 2014*, Wien, p. 125, 2013.

[17] M. Cogswell, F. Ahmed, R. Girshick, L. Zitnick, and D. Batra, “Reducing overfitting in deep networks by decorrelating representations,” *ArXiv Prepr. ArXiv151106068*, 2015.

[18] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, “Behavior-based Spyware Detection.,” in *Usenix Security Symposium*, 2006, p. 694.

[19] G. D. Rey and K. F. Wender, *Neuronale Netze: eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*. H. Huber, 2008.

[20] T. Kaffka, *Neuronale Netze. Grundlagen: Mit Beispielprogrammen in Java*. MITP-Verlags GmbH & Co. KG, 2017.

[21] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, Dec. 1943.

[22] N. Weidtmann and D. Evers, *Konstruktion und Wirklichkeit: Virtualität, Vergessen, künstliches Bewusstsein, autobiographische Erinnerung, Emotionen*. LIT Verlag Münster, 2011.
[23] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.,” *Psychol. Rev.*, vol. 65, no. 6, p. 386, 1958.
[24] B. Widrow and M. E. Hoff, “Adaptive switching circuits,” STANFORD UNIV CA STANFORD ELECTRONICS LABS, 1960.
[25] R. Libbey, *Signal And Image Processing Sourcebook*. Springer US, 1994.
[26] M. L. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Mit Press, 1972.
[27] D. E. Rumelhart and J. L. McClelland, *Parallel distributed processing: explorations in the microstructure of cognition*. Cambridge, Mass: MIT Press, 1986.
[28] U. Eberl, “Deep Learning: Wo künstliche Intelligenz den Menschen schon übertrifft,” *Die Zeit*, Hamburg, 11-Oct-2016.
[29] “Ramp function,” *Wikipedia*, 08-Feb-2018. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/c/c9/Ramp_function.svg/600px-Ramp_function.svg.png. [Accessed: 06-Mar-2018].
[30] Google, “Hyperparameter-Abstimmung – Übersicht | Cloud Machine Learning Engine (Cloud ML Engine),” *Google Cloud Platform*. [Online]. Available: https://cloud.google.com/ml-engine/docs/hyperparameter-tuning-overview. [Accessed: 06-Mar-2018].
[31] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” *ArXiv150201852 Cs*, Feb. 2015.
[32] Keras, “Initializers - Keras Documentation.” [Online]. Available: https://keras.io/initializers/. [Accessed: 06-Mar-2018].
[33] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 2010, pp. 249–256.
[34] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, “Greedy Layer-wise Training of Deep Networks,” in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, Cambridge, MA, USA, 2006, pp. 153–160.
[35] G. E. Hinton, “Deep belief networks,” *Scholarpedia*, vol. 4, no. 5, p. 5947, May 2009.
[36] R. Socher *et al.*, “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank,” p. 12.
[37] N. E. Mastorakis, WSEAS (Organization), and WSEAS International Conference on Neural Networks, Eds., *Recent advances in neural networks: proceedings of the 10th WSEAS International Conference on Neural Networks (NN ’09), Prague, Czech Republic, March 23-25, 2009*. Prague: WSEAS, 2009.
[38] H. T. Siegelmann, “Computation Beyond the Turing Limit,” *Science*, vol. 268, no. 5210, pp. 545–548, Apr. 1995.
[39] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.
[40] HealthResearchFunding, “Adaptive Resonance Theory Explained,” *HRFnd*, 18-Sep-2017. [Online]. Available: https://healthresearchfunding.org/adaptive-resonance-theory-explained/. [Accessed: 06-Mar-2018].
[41] S. Vasilic and M. Kezunovic, “Fuzzy ART Neural Network Algorithm for Classifying the Power System Faults,” *IEEE Trans. Power Deliv.*, vol. 20, no. 2, pp. 1306–1314, Apr. 2005.
[42] G. A. Carpenter, S. Grossberg, and D. B. Rosen, “Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system,” *Neural Netw.*, vol. 4, no. 6, pp. 759–771, 1991.
[43] G. A. Carpenter, “Distributed Learning, Recognition, and Prediction by ART and ARTMAP Neural Networks,” *Neural Netw. Off. J. Int. Neural Netw. Soc.*, vol. 10, no. 8, pp. 1473–1494, Nov. 1997.
[44] G. A. Carpenter, S. Grossberg, and J. H. Reynolds, “ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network,” *Neural Netw.*, vol. 4, no. 5, pp. 565–588, 1991.
[45] G. A. Carpenter, S. Grossberg, N. Markuzon, J. H. Reynolds, and D. B. Rosen, “Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps,” *IEEE Trans. Neural Netw.*, vol. 3, no. 5, pp. 698–713, 1992.
[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” *ArXiv13013781 Cs*, Jan. 2013.

[47] D. Ciregan, U. Meier, and J. Schmidhuber, “Multi-column deep neural networks for image classification,” in *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on*, 2012, pp. 3642–3649.
[48] G. E. Hinton, S. Osindero, and Y.-W. Teh, “A fast learning algorithm for deep belief nets,” *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006.
[49] G. L. Shaw, “Donald Hebb: The Organization of Behavior,” in *Brain Theory*, Springer, Berlin, Heidelberg, 1986, pp. 231–233.
[50] “Styles of Truncated Backpropagation - R2RT.” [Online]. Available: https://r2rt.com/styles-of-truncated-backpropagation.html. [Accessed: 07-Mar-2018].
[51] I. Sutskever, “Training recurrent neural networks,” *Univ. Tor. Tor. Ont Can.*, 2013.
[52] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *J. Mach. Learn. Res.*, vol. 12, no. Jul, pp. 2121–2159, 2011.
[53] M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method,” *ArXiv12125701 Cs*, Dec. 2012.
[54] Google Cloud Platform, “TensorFlow and Deep Learning without a PhD, Part 2 (Google Cloud Next ’17).” [Online]. Available: https://www.youtube.com/watch?v=fTUwdXUFfI8. [Accessed: 06-Mar-2018].
[55] Joyce Wang, “YOW! Data 2017 - Covariate Shift - Challenges and Good Practice.” [Online]. Available: https://www.youtube.com/watch?v=YzvlBz_nV2k. [Accessed: 08-Mar-2018].
[56] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *ArXiv150203167 Cs*, Feb. 2015.
[57] Google Developers, “Effective TensorFlow for Non-Experts (Google I/O ’17).” [Online]. Available: https://www.youtube.com/watch?v=5DknTFbcGVM. [Accessed: 09-Mar-2018].
[58] Coding Tech, “TensorFlow 101.” [Online]. Available: https://www.youtube.com/watch?v=oxf3o8IbCk4. [Accessed: 06-Mar-2018].
[59] Tensorflow, “Using AOT compilation,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/performance/xla/tfcompile. [Accessed: 06-Mar-2018].
[60] Google Developers, “Keynote (TensorFlow Dev Summit 2017).” [Online]. Available: https://www.youtube.com/watch?v=4n1AHvDvVvw. [Accessed: 06-Mar-2018].
[61] Caffe, “Caffe | Deep Learning Framework.” [Online]. Available: http://caffe.berkeleyvision.org/. [Accessed: 06-Mar-2018].
[62] Berkeley AI Research, “DIY Deep Learning for Vision: a Hands-On Tutorial with Caffe,” *Google Docs*. [Online]. Available: https://docs.google.com/presentation/d/1UeKXVgRvvxg9OUdh_UiC5G71UMscNPlvArsWER41PsU. [Accessed: 06-Mar-2018].
[63] Theano, “Theano 1.0.0 documentation.” [Online]. Available: http://deeplearning.net/software/theano/. [Accessed: 06-Mar-2018].
[64] PyTorch, “PyTorch.” [Online]. Available: http://pytorch.org/. [Accessed: 06-Mar-2018].
[65] Github, “gluon-api: A clear, concise, simple yet powerful and efficient API for deep learning,” 05-Mar-2018. [Online]. Available: https://github.com/gluon-api/gluon-api. [Accessed: 06-Mar-2018].
[66] Microsoft, “AWS and Microsoft announce Gluon, making deep learning accessible to all developers,” *Stories*, 12-Oct-2017. [Online]. Available: https://news.microsoft.com/2017/10/12/aws-and-microsoft-announce-gluon-making-deep-learning-accessible-to-all-developers/. [Accessed: 06-Mar-2018].
[67] Microsoft Cognitive Toolkit, “Intro to Microsoft Cognitive Toolkit.” [Online]. Available: https://www.youtube.com/watch?v=9gDDO5ldT-4. [Accessed: 06-Mar-2018].
[68] “Installing TensorFlow from Sources,” *TensorFlow*. [Online]. Available: https://www.tensorflow.org/install/install_sources. [Accessed: 04-Apr-2018].
[69] R. Luh, S. Schrittwieser, and S. Marschalek, “LLR-Based Sentiment Analysis for Kernel Event Sequences,” 2017, pp. 764–771.