Known-Plaintext-Angriffe auf historische Chiffren unter Verwendung von neuronalen Netzwerken

Nino Fürthauer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Sichere Informationssysteme

in Hagenberg

im Mai 2022

Betreuung:

FH-Prof. DI Dr. Eckehard Hermann

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt. Die vorliegende, gedruckte Arbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Hagenberg, am 31. Mai 2022

Nino Fürthauer

Inhaltsverzeichnis

Danksagung

An dieser Stelle möchte ich mich bei den Personen bedanken, die entscheidend zum Gelingen dieser Arbeit beigetragen haben.

Zuallererst gebührt mein Dank Dr. Vasily Mikhalev und Dr. Nils Kopal von der Universität Siegen. Die beiden führten mich in zahlreichen Meetings zielsicher durch die Welt der historischen Kryptografie. Ganz egal, ob es um die Zielsetzung der Arbeit oder um die praktische Einordnung der Ergebnisse ging, Vasily und Nils standen mir stets mit Rat zur Seite. Danke dafür!

Ebenfalls bedanken möchte ich mich bei Prof. Bernhard Esslinger (Universität Siegen), der vor allem bei der Erstellung des zu dieser Arbeit gehörenden Papers *Evaluating Deep Learning Techniques for Known-Plaintext Attacks on the Complete Columnar Transposition Cipher* eine wertvolle Hilfe war.

Großer Dank gilt meinen beiden Professoren FH-Prof. Dr. Harald Lampesberger und FH-Prof. DI Dr. Eckehard Hermann. Als Leiter des *Smart Security Labs (SSL)* am Department Sichere Informationssysteme ermöglichten sie mir diese Kooperation mit den Kollegen der Universität Siegen. Vor allem die Publikation der Ergebnisse wäre ohne ihren Einsatz so wohl nicht möglich gewesen. Zusätzlich bedanken möchte ich mich bei Eckehard als offiziellen Betreuer der Masterarbeit für sein detailliertes Feedback zu den einzelnen Kapiteln, das vor allem die sprachliche Klarheit der Arbeit entscheidend verbessert hat. Mein herzlichster Dank gebührt Harald und Eckehard jedoch für die projektbegleitenden Meetings und Diskussionen. Obwohl nach rund zweijähriger Pandemie im gesamten Hochschulbereich eine „Online-Meeting-Müdigkeit“ festgestellt werden konnte, waren unsere Diskussionen nicht nur stets fachlich wertvoll und hilfreich, sondern vor allem auch immer unterhaltsam und motivierend. Wenngleich ich zu Beginn des Masterstudiums eher durch Zufall im SSL gelandet bin, hätte ich mir rückblickend kein besseres Lab vorstellen können.

Abschließend verdienen auch meine Eltern Christian und Sabine Erwähnung. Wie in Forschungsprojekten üblich, gab es auch bei dieser Arbeit regelmäßig Rückschläge und fehlgeschlagene Experimente. Meine Eltern ertrugen dabei stets mit Fassung meine daraus folgenden frustrierten Monologe. Da dies alles andere als selbstverständlich ist, möchte ich mich an dieser Stelle herzlich für ihre Unterstützung bedanken.

Kurzfassung

Ein Known-Plaintext-Angriff (KPA) ist eine Angriffsart der Kryptoanalyse, bei der aus einem Ciphertext und dem dazugehörigen Plaintext der verwendete Schlüssel extrahiert wird. In dieser Arbeit werden solche Angriffe mit neuronalen Netzwerken auf verschiedene historische Chiffren durchgeführt, was im Folgenden als neuronale KPA bezeichnet wird. Zur Auswahl einer geeigneten Architektur wird eine umfangreiche Evaluierung verschiedener Varianten rekurrenter neuronaler Netzwerke (RNN), Encoder-Decoder und Attention-basierter Architekturen durchgeführt. Mit der so gewählten Architektur, einem GRU-LSTM-RNN mit 8192 Units, wird zusätzlich gezeigt, dass RNN in der Lage sind, die Entschlüsselungsfunktion der Caesar-Chiffre, der MASC, der Vigenère- und der Autokey-Vigenère-Chiffre, der Enigma, der Spaltentransposition und der ADFGVX-Chiffre zu erlernen. Bei den neuronalen KPA zeigt sich, dass das Erlernen dieser bei weniger Chiffren erfolgreich ist als das Erlernen der Entschlüsselungsfunktionen. Bei den betrachteten Chiffren (Caesar, MASC, Vigenère, Autokey-Vigenère, Playfair, Hill, Spaltentransposition) ist es nur bei der Caesar-, der Vigenère- und der Autokey-Vigenère-Chiffre möglich, das GRU-LSTM-RNN End-to-End erfolgreich mit einer Accuracy von ungefähr 95% zu trainieren. Aus diesem Grund wird für die Spaltentransposition, der einzigen betrachteten reinen Transpositionschiffre dieser Arbeit, ein neuer Algorithmus für neuronale KPA vorgestellt. Dieser Algorithmus kombiniert mehrere GRU-LSTM-RNN-Modelle mit Pre- und Post-Processing-Schritten. Im Gegensatz zu den End-to-End-Modellen, die auf maximale Schlüssellängen von 6 Zeichen limitiert sind, unterstützt der Algorithmus Schlüssellängen von $2 - 20$ Zeichen. Durch seine modulare Architektur kann er auch in Zukunft mit zusätzlichen Modellen erweitert werden, sodass noch längere Schlüssel verarbeitet werden können. In einer umfangreichen Evaluierung mit verschiedenen Plain-/Ciphertextlängen zeigt sich, dass der Algorithmus bei ausreichend langen Plain-/Ciphertexten auch bei Schlüssellängen von 20 Zeichen in mindestens 96% der Fälle den richtigen Schlüssel ausgibt. Dieses Ergebnis entspricht nach bestem Wissen einem neuem Stand der Technik im Bereich Deep-Learning-basierter Known-Plaintext-Angriffe auf die (komplette) Spaltentransposition.

Abstract

A known-plaintext attack (KPA) is a special kind of attack where the used key is calculated based on a given ciphertext and a corresponding plaintext. This thesis examines such attacks on historic ciphers using neural networks, which is called neural KPA. At first, several architectures including different variants of traditional recurrent neural networks (RNN), encoder-decoders and attention-based architectures like Transformers are evaluated. As a result of this evaluation, a GRU-LSTM-RNN with 8192 units is chosen. The thesis shows that this architecture can learn a representation of the decryption function of the following ciphers: Caesar, MASC, Vigenère, Autokey-Vigenère, Enigma, columnar transposition and ADFGVX. When it comes to neural KPA, trainings are less successful: Only attacks on Caesar, Vigenère and Autokey-Vigenère can be learned end-to-end with a sufficient accuracy ($\approx 95\%$), for other ciphers (MASC, Playfair, Hill, columnar transposition) this is not possible. As a result, a new algorithm for neural KPA on (complete) columnar transposition is proposed in this thesis. The algorithm combines several GRU-LSTM-RNN with additional pre- and post-processing-steps to ultimately support key lengths of $2 - 20$ characters. This is a significant increase in comparison to the end-to-end-models, that only support keys up to 6 characters. Due to a modular architecture, the algorithm can be extended easily, so that it maybe will be able to support even longer keys in the future. Each currently supported key length has been empirically evaluated with plain-/ciphertext-pairs of different lengths. For plain- and ciphertexts with a length of five times the key length, the algorithm achieves a success rate of at least 96% which is a new state of the art on deep-learning-based known-plaintext attacks against (complete) columnar transposition.

Kapitel 1

Einleitung

Schon seit Jahrhunderten wird versucht, mit verschiedenen Methoden Botschaften vor unbefugter Einsicht zu schützen. Einer der ältesten bekannten verschlüsselten Texte findet sich auf einer Tontafel eines mesopotamischen Töpfers, der ca. um 1500 v. Chr. übliche Keilschriftbuchstaben in einer ungewöhnlichen Weise verwendete, sodass sie für Uneingeweihte unverständlich war [1, S. 10]. Später wurden auch gezielt Werkzeuge zur Verschlüsselung eingesetzt, ein sehr frühes und bekanntes Beispiel dafür ist die Skytale, die möglicherweise von den Spartanern verwendet wurde. Bei der Skytale handelt es sich um einen einfachen Holzstab, um den ein Lederstreifen gewickelt wurde, auf welchen anschließend die zu übermittelnde Nachricht geschrieben wurde. Nur wenn bei der Entschlüsselung ein Stab des richtigen Durchmessers verwendet wurde, konnte die originale Nachricht wiederhergestellt werden – der Durchmesser war also der Schlüssel des Verfahrens [1, S. 10]. In der Forschung ist mittlerweile jedoch umstritten, ob die Skytale tatsächlich ein Verschlüsselungsverfahren war oder ob es in Wahrheit für andere Zwecke verwendet wurde [2]. Dass Verschlüsselungstechniken nicht zuletzt auch militärisch relevant waren, zeigt die sogenannte Caesar-Chiffre, die laut Sueton von ihrem Namensgeber, Gaius Julius Caesar, verwendet wurde [1, S. 10f]. Bei der Caesar-Chiffre handelt es sich dabei um eine einfache Substitutionschiffre, bei der jeder Buchstabe des Klartexts um eine bestimmte Distanz im Alphabet verschoben wird. So wird beispielsweise aus einem „N“ ein „Q“, wenn als Schlüssel 3 gewählt wurde.

Aufgrund der Brisanz vieler verschlüsselter Nachrichten bestand auch ein großes Interesse daran, die Verschlüsselung zu brechen und somit den Klartext einsehen zu können. Im Jahr 855 veröffentlichte etwa der arabische Universalgelehrte al-Kindī ein erstes Buch über Kryptologie, in dem er ein grundlegendes Verfahren, das später als Häufigkeitsanalyse bekannt wurde, beschreibt [77]. Solche Methoden werden dabei unter dem Begriff *Kryptoanalyse* zusammengefasst, welcher somit das Gegenstück zur *Kryptografie* beschreibt. Kryptografie und Kryptoanalyse bilden zusammen den Oberbegriff *Kryptologie*. Der wohl – nicht zuletzt wegen dem Film *The Imitation Game* – in der Öffentlichkeit bekannteste Erfolg der Kryptoanalyse ist die erfolgreiche Entschlüsselung von Enigma-Chiffren durch Kryptoanalytiker rund um Alan Turing in Bletchley Park während des Zweiten Weltkriegs.

Wenn auch die Entschlüsselung solcher historischen Chiffren heute keine militärischen Vorteile mehr bringt, so kann sie immer noch ein Gewinn für die Geschichtsforschung sein. Tausende verschlüsselte Schriftstücke wie beispielsweise militärische,

nachrichtendienstliche oder diplomatische Dokumente finden sich in Archiven, aber alleine die schiere Anzahl von ihnen macht eine umfassende Entschlüsselung schwierig. Abhilfe soll das vom Swedish Research Council geförderte DECRYPT Project [3] schaffen: Mithilfe eines breiten, interdisziplinären Ansatzes, der unter anderem die Linguistik, die Informatik und die Kryptologie umfasst, sollen Ressourcen und Tools für eine (semi-)automatische Entschlüsselung bereitgestellt werden. Eine mögliche Herangehensweise für solche Tools ist der Einsatz von Methoden des *Deep Learnings*, einem Teilbereich der Künstlichen Intelligenz (KI), der in den letzten Jahren beeindruckende Ergebnisse in verschiedenen Anwendungsgebieten wie etwa der maschinellen Übersetzung erzielt hat.

1.1 Problemstellung

Das sogenannte *Universal Approximation Theorem (UAT)* besagt, dass für jede beliebige Funktion in der Theorie ein ausreichend großes Netzwerk existiert, das die Funktion repräsentieren kann[1] [4, S. 193]. Ein (symmetrisches) Verschlüsselungsverfahren besteht aus einer Verschlüsselungsfunktion E und einer Entschlüsselungsfunktion D. Sei m ein beliebiger Klartext (*Plaintext*), k ein beliebiger, für das jeweilige Verfahren gültiger Schlüssel und c der resultierende Geheimtext (*Ciphertext*), so ergeben sich die beiden Funktionen $E(m, k) = c$ und $D(c, k) = m$. Aus dem UAT ergibt sich somit, dass neuronale Netzwerke sowohl E als auch D in der Theorie approximieren können müssten. Dass dies zumindest für D bei den historischen Substitutionschiffren Vigenère, Vigenère-Autokey und Enigma möglich ist, zeigt Greydanus in [5], wo mithilfe von einfachen LSTM-basierten Recurrent Neural Networks (RNN) die entsprechenden Funktionen erlernt wurden. Auf Basis dieser Arbeit stellt sich die Frage, ob diese Ergebnisse auf ähnliche Problemstellungen übertragen werden können, d. h. ob auch andere Chiffren derart erlernt werden können. Da von Greydanus lediglich Substitutionschiffren betrachtet wurden, wäre insbesondere interessant, ob der Ansatz bei Transpositionschiffren genauso verwendet werden kann. Problematisch ist jedoch, dass in [5] unabhängig von der behandelten Chiffre mindestens 1 Mio. Trainingssamples benötigt werden. Diese Datenmenge müsste mit einem komplexeren Netzwerk verringert werden können.

Für die Kryptoanalyse interessanter als das Erlernen von D ist ein sogenannter *Known-Plaintext-Angriff (KPA)*. Um einen KPA durchführen zu können, muss zu einem Ciphertext auch der Plaintext (in diesem Szenario auch *Crib* genannt) bekannt sein. Das Ziel solch eines Angriffes besteht dabei darin, den verwendeten Schlüssel k oder zumindest Teile davon zu rekonstruieren. Dieser kann wiederum nützlich sein, wenn beispielsweise der Verdacht besteht, dass k für die Erstellung eines anderen Ciphertexts, zu dem kein Plaintext vorliegt, verwendet wurde. Ein Known-Plaintext-Angriff lässt sich formal als Funktion $f(c, m) = k$ beschreiben. Ein neuronales Netzwerk müsste also auch einen KPA erlernen können. Dass dies in der Praxis zumindest für Vigenère und Vigenère-Autokey funktioniert, hat Greydanus ebenfalls in [5] gezeigt. Auch hier stellt sich jedoch die Frage nach der Verallgemeinerung der Resultate, insbesondere da Vigenère und Vigenère-Autokey sehr ähnliche Chiffren sind, bei denen lediglich der Schlüssel

[1] Dies bedeutet nicht, dass die Funktion auch in der Praxis erlernt werden kann. Es ist beispielsweise möglich, dass das Netzwerk durch Overfitting eine andere Funktion erlernt oder dass der Lernalgorithmus nicht die richtigen Parameter findet [4, S. 193]

leicht unterschiedlich konstruiert wird. Für Transpositionschiffren liegen generell bisher keine Publikationen vor, die einen erfolgreichen KPA mittels neuronaler Netzwerke (im Folgenden kurz *neuronaler KPA* genannt) demonstrieren.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, die Arbeit von Greydanus zu verbessern und auf weitere Chiffren auszudehnen. Dazu soll zuerst im Zuge einer empirischen Evaluation eine Architektur gesucht werden, die für das Erlernen der Entschlüsselungsfunktion möglichst wenige, aber mindestens unter einer Million, Trainingssamples benötigt. Als Referenzchiffre dient hierbei Vigenère, als Qualitätsmetrik soll eine genauso hohe Accuracy (99%) im Training erreicht werden. Anschließend soll das Erlernen der Entschlüsselungsfunktion auf folgende Chiffren ausgeweitet werden: Caesar, MASC, Playfair, Hill, Spaltentransposition (eine Transpositionschiffre) und ADFGVX (eine Kombinationschiffre).

Die von Greydanus präsentierten Known-Plaintext-Angriffe sollen ebenfalls erweitert werden. Dazu soll zunächst das verwendete Konzept auf weitere Substitutionschiffren (Caesar, MASC, Playfair, Hill) angewandt werden. Die Auswahl dieser Substitutionschiffren basiert dabei auf einer Schwierigkeitseinschätzung von Nils Kopal von der Universität Siegen. Die Caesar-Chiffre soll dabei die am einfachsten anzugreifende Chiffre sein, die Hill-Chiffre hingegen die schwerste. Im nächsten Schritt soll evaluiert werden, ob neuronale KPA auch auf die Klasse der Transpositionschiffren durchgeführt werden können. Die Spaltentransposition wird dabei exemplarisch als Testchiffre verwendet.

1.3 These

Die simple LSTM-RNN-Architektur von Greydanus kann durch eine bessere ersetzt werden, was sich beispielhaft daran zeigt, dass die Entschlüsselungsfunktion der Vigenère-Chiffre mit weniger als einer Million Trainingssamples bei gleichbleibender Accuracy erlernt werden kann. Neuronale KPA sind prinzipiell auch bei anderen Substitutionschiffren und bei Vertretern der Klasse der Transpositionschiffren durchführbar.

1.4 Forschungsfragen

- Kann mithilfe einer verbesserten Architektur der in [5] präsentierte Ansatz zum Lernen der Entschlüsselungsfunktionen verschiedener Chiffren weiter verbessert werden?
  - Wie kann der Ansatz von Greydanus so weit verbessert werden, dass der Ansatz auch in einem Szenario, in dem weniger Trainingssamples als in [5] vorhanden sind, einsetzbar wird?
  - Kann der Ansatz auch bei weiteren Substitutions- (Caesar, MASC, Playfair, Hill), Transpositions- (Spaltentransposition) und Kombinationschiffren (ADFGVX) eingesetzt werden?
- Kann mittels neuronaler Netze ein Known-Plaintext-Angriff auf weitere Chiffren durchgeführt werden?

  - Kann ein neuronaler KPA auf weitere Substitutionschiffren, namentlich Caesar, MASC, Playfair und Hill, durchgeführt werden?
  - Funktioniert dieser Ansatz auch bei Transpositionschiffren bzw. als Vertreter davon bei der Spaltentransposition?
- Wie gut eignen sich die erstellten Modelle tatsächlich für neuronale KPA auf Transpositionschiffren in der Praxis?

1.5 Methodik

Im Zuge dieser Arbeit wird die Design-Science-Methode angewendet. Dazu wird zunächst die Arbeit von Sam Greydanus reproduziert. Anschließend wird mit einer empirischen Evaluierung auf Basis der Entschlüsselungsfunktion der Vigenère-Chiffre nach einer stärken Architektur, d. h. einer Architektur, die weniger Trainingssamples benötigt, gesucht. Diese Architektur wird für das Erlernen weiterer Entschlüsselungsfunktionen verwendet. Da Greydanus ebenfalls dieselbe Architektur für das Erlernen der Entschlüsselungsfunktion und für den KPA verwendet hat, wird angenommen, dass eine Architektur, welche die Entschlüsselungsfunktion erlernen kann, auch einen KPA auf dieselbe Chiffre erlernen kann. Aus diesem Grund wird die ausgewählte Architektur auch für den KPA-Teil dieser Arbeit herangezogen. Zusätzlich wird dieser Ansatz auch bei der Spaltentransposition als Vertreter der Transpositionschiffren getestet und so erweitert, dass auch auf diese Verschlüsselung neuronale KPA möglich sind. Abschließend wird eine Evaluierung des neuronalen KPA auf die Spaltentransposition durchgeführt, um Praxistauglichkeit und Einschränkungen feststellen zu können.

1.6 Gliederung

Die weitere Arbeit ist folgendermaßen aufgebaut: Kapitel 2 gibt einen Überblick über die wichtigsten Grundlagen zum Deep Learning und allen getesteten Architekturen. Bei Kapitel 3 handelt es sich ebenfalls um ein Grundlagenkapitel, diesmal zum Bereich Kryptologie. Ein besonderer Schwerpunkt wird dabei auf die verwendeten historischen Chiffren gelegt, diese werden dabei mit Beispielen möglichst anschaulich erklärt. Kapitel 4 beschreibt den aktuellen Stand der Technik, während Kapitel 5 die empirische Architekturevaluierung und das Erlernen der Entschlüsselungsfunktionen beschreibt. Das sechste Kapitel betrachtet die eigentlichen neuronalen KPA, inklusive der Vorstellung eines neuen Algorithmus für neuronale KPA auf die Spaltentranspositionschiffre. Das letzte Kapitel fasst abschließend die Ergebnisse nochmals zusammen und gibt einen Ausblick auf weitere mögliche Verbesserungen des präsentierten Algorithmus.

Kapitel 2

Einführung Deep Learning

In diesem Kapitel sollen die wichtigsten Grundlagen des Deep Learnings zusammengefasst werden. Dabei wird zunächst der kleinste Baustein eines neuronalen Netzwerks, das Neuron, betrachtet und es wird erklärt, wie Training und Inferenz (also die Anwendung eines trainierten Modells[1] auf unbekannte Daten) funktionieren. Mehrere solcher Neuronen werden anschließend im Verbund eines simplen *Feedforward Neural Networks (FFNN)* beschrieben, was der einfachsten Variante eines neuronalen Netzwerks entspricht. Abschließend wird ein Überblick über alle in dieser Arbeit verwendeten Architekturen gegeben.

2.1 Grundlagen des Machine Learnings

Machine Learning (ML) ist ein Teilbereich der Künstlichen Intelligenz (KI). Die Disziplin zielt darauf ab, Systeme zu bauen, die selbstständig über gemachte Erfahrungen besser in vorher bestimmten Aufgaben werden [6]. Dieser Leistungsgewinn wird über eine festgelegte Metrik bestimmt, die Erfahrung wird bei einem sogenannten *Training* gemacht. Ein Beispiel dafür ist ein System, das erkennen sollte, ob auf einem Bild eine Katze sichtbar ist. Um Erfahrung zu ermöglichen, wird dem System beim Training eine große Menge an verschiedenen Bildern präsentiert. Auf manchen davon befindet sich eine Katze, auf anderen hingegen nicht. Diese Sammlung von Bildern nennt sich *Trainingsdatensatz*, ein einzelnes Bild *Trainingsdatenpunkt* [7, S. 4]. Über ein *Label* $y^{(i)}$, das den gewünschten Systemoutput für einen bestimmten Trainingsdatenpunkt $x^{(i)}$ darstellt, wird dem System mitgeteilt, ob es sich bei einem Bild um ein Katzenbild handelt ($y^{(i)} = 1$) oder nicht ($y^{(i)} = 0$)[2]. Wenn ein passendes ML-Verfahren gewählt wurde und genügend Daten vorliegen, wird das System im Laufe des Trainings selbst erlernen, welche Eigenschaften eines Bildes auf eine Katze hindeuten. Schlussendlich wird es fähig sein, auch Bilder, die nicht im Trainingsdatensatz vorhanden sind, richtig einzuordnen. Ein Beispiel für eine mögliche Metrik ist die *Accuracy*, die angibt, wie viele Bilder der binäre Klassifikator richtig klassifiziert hat.

[1] Unter einem Modell wird das Ergebnis eines Trainings mit einem Machine-Learning-Algorithmus verstanden [78].

[2] i bezeichnet den Index eines Datenpunkts/Labels innerhalb des Trainingsdatensatzes. Es gilt $i \in \mathbb{N}$. $x^{(5)}$ beschreibt also den fünften Datenpunkt, $y^{(5)}$ das dazugehörige fünfte Label.

Dieses selbstständige Lernen stellt den Hauptunterschied von Machine Learning zu „traditioneller“ Programmierung dar. Bei Letzterem wird eine Funktion oder ein Algorithmus, hier ϕ genannt, manuell festgelegt. Anschließend kann dieses Programm auf Inputdaten X angewandt werden, um die gewünschten Outputdaten Y zu erhalten. Damit dies möglich ist, muss $\phi : X \mapsto Y$ aber bekannt sein. Bei Machine-Learning-Verfahren muss die Abbildung $X \mapsto Y$ hingegen nicht bekannt sein. Stattdessen werden dem ausgewählten Verfahren X und Y als Trainingsdaten präsentiert, wodurch das System die Abbildung selbst erlernt. Um ein Modell trainieren zu können, das ausreichend *generalisiert*, d.h. die auf dem Trainingsdatensatz gemachten Erfahrungen können auch mit ausreichend guter Leistung auf neue, unbekannte Datenpunkte angewendet werden (siehe Kapitel 2.6), ist neben dem ML-Algorithmus der Trainingsdatensatz von entscheidender Bedeutung. Häufige Fehler dabei können sein [7, S. 24ff]:

- Unzureichende Datenmenge: Die meisten ML-Algorithmen brauchen je nach Komplexität der Aufgabe sehr viele Datenpunkte. Liegt ein zu kleiner Trainingsdatensatz vor, kann ϕ nicht erlernt werden.
- Nicht repräsentative Daten: Damit das erstellte Modell auf unbekannte Datenpunkte angewendet werden kann, müssen die verwendeten Trainingsdaten so repräsentativ wie möglich sein. Stammen beispielsweise die Datenpunkte im Produktivbetrieb aus einer komplett anderen Quelle als die Trainingsdaten, so wird mit hoher Wahrscheinlichkeit die Erkennungs- bzw. Vorhersageleistung des Modells stark abfallen. Die zur Beurteilung der Modellqualität verwendete Metrik (z.B. die Accuracy) wird aufgrund der unterschiedlichen Datenquellen einen niedrigeren Wert ergeben. Ein Beispielszenario ist ein System, das mit RGB-Bildern trainiert wurde, aber im Produktivbetrieb mit 2D-gemappten ToF-Bildern arbeiten muss.
- Irrelevante Merkmale: Merkmale, auch *Features* genannt, beschreiben Eigenschaften eines Datenpunktes. Auf Basis dieser Merkmale wird versucht, ϕ zu erlernen. Handelt es sich bei den verwendeten Merkmalen aber hauptsächlich um Merkmale, die Y nicht erklären können (die also in keinem bzw. geringem Zusammenhang mit dem gewünschten Output stehen), so wird der ML-Algorithmus ϕ nicht ausreichend erlernen können. Dieses Problem ist auf den ersten Blick dem folgenden Punkt „Minderwertige Daten“ ähnlich. Der Unterschied ist jedoch, dass die Merkmale bei minderwertige Daten Y unter Umständen schon erklären können, wenn sie in ausreichender Qualität vorliegen. Bei irrelevanten Merkmale ist dies nicht der Fall, auch wenn die Daten selbst von ausreichender Qualität sind.
- Minderwertige Daten: Bei fehlerhaften Daten fällt es dem ML-Algorithmus schwerer, den Zusammenhang zwischen X und Y zu erkennen. Fehlerhafte Daten entstehen unter anderem, wenn für viele Trainingsdatenpunkte kein Wert für gewisse Merkmale zur Verfügung steht. Solche fehlenden Werte müssen dann mit Platzhaltern versehen werden. Ein Beispiel ist ein Datensatz, der tägliche Wetterdaten über ein ganzes Jahr enthält. Datenpunkte für Tage, an denen die Temperatur nicht verfügbar ist, sind fehlerhaft und müssen nachbearbeitet werden. Enthält der Datensatz insgesamt zu viele solcher fehlerhaften Datenpunkte, wird es für das System schwierig, den Zusammenhang zwischen X und Y zu erkennen.

Nicht zuletzt aufgrund dieser Anforderungen an den Trainingsdatensatz ist es nicht

sinnvoll, Machine Learning für jedes Problem einzusetzen. Nach Mitchell [8] bietet sich die Verwendung von Machine Learning insbesondere in folgenden zwei Szenarien an:

1. Bei Anwendungen, bei denen ϕ unbekannt oder zu komplex für einen Algorithmus ist.
2. Es ist erforderlich, dass die Anwendung im Produktivbetrieb ständig an die Umgebung angepasst wird, also weiter lernt. Ein Beispiel ist ein Spracherkennungssystem, das sich an den Benutzer oder die Benutzerin anpasst.

Die bisher betrachtete Aufgabenstellung des Erlernens von $\phi : X \mapsto Y$ unter Verwendung eines vollständig annotierten Trainingsdatensatzes X, bei dem für jeden Trainingsdatenpunkt $x^{(i)} \in X$ ein Label $y^{(i)} \in Y$ existiert, wird als *Supervised Learning* bezeichnet. Beim Supervised Learning wird dem System während dem Training mitgeteilt, was der richtige Output ist. Ziel des Trainings ist somit, dass das System ein Modell erstellt, dessen Vorhersage $\hat{y}^{(i)}$ für einen Datenpunkt $x^{(i)}$ möglichst genau $y^{(i)}$ entspricht. Es existiert aber auch die Kategorie des *Unsupervised Learnings*, bei welcher keine Labels zur Verfügung stehen. Anstatt eine Abbildung ϕ zu erlernen, ist das Ziel eines solchen Systems, gemeinsame Muster in den Trainingsdaten zu erkennen und diese entsprechend zu strukturieren. Ein bekannter Algorithmus dafür ist das sogenannte *k-Means Clustering*, bei dem ein Datensatz in k Kategorien aufgeteilt wird [9, S. 24]. Eine andere Verwendungsmöglichkeit für Unsupervised Learning ist die Anomalieerkennung. Eine Mischform von Supervised und Unsupervised Learning wird als *Semi-Supervised Learning* bezeichnet [7, S. 14]. Semi-Supervised Learning hat den Vorteil, dass nicht der gesamte Datensatz gelabelt sein muss, was üblicherweise teuer und aufwendig ist. Ein komplett anderer Ansatz ist das sogenannte *Reinforcement Learning*. Dabei lernt ein System über Konsequenzen (Belohnung und Bestrafung) für durchgeführte Aktionen, wie es sich bei der Interaktion mit einer Umgebung zu verhalten hat. Es wird somit eine sogenannte *Policy* gelernt, auf Basis der das System entscheidet, welche Aktionen es setzen soll. Ziel des Systems ist dabei, so zu handeln, dass es möglichst viele Belohnungen bekommt. Es kann beispielsweise ein Spiel erlernt werden, ohne dass das zugrundeliegende Regelwerk explizit programmiert werden muss [9, S. 199]. Da es sich sowohl beim Erlernen einer Entschlüsselungsfunktion als auch beim Erlernen eines KPA um Probleme des Supervised Learnings handelt, sollen die übrigen Kategorien an dieser Stelle nicht weiter behandelt werden.

2.2 Begriffliche Einordnung des Deep Learnings

Deep Learning ist ein Teilbereich des maschinellen Lernens, genauer gesagt des *Representation Learnings* [4, S. 4f]. Wie bereits in Kapitel 2.1 erwähnt, ist es wichtig, relevante Features für das Lernen auszuwählen. Dieses *Feature Engineering* ist jedoch oft schwierig, da nicht klar ist, welche Features für die Vorhersage wichtig sind. Ein Beispiel dafür ist das bereits zuvor erwähnte System zur Erkennung von Katzen auf Bildern. Für Menschen ist es relativ simpel, eine Katze anhand ihrer Eigenschaften – Ohren, Schwanz, vier Beine – zu erkennen. Es stellt sich jedoch die Frage, wie diese Eigenschaften als Features verwendet werden können, da ein Bild für einen Computer nichts anderes als eine Folge von Pixelwerten ist. Welche Kombination von Zahlen stellt also beispielsweise ein Katzenohr dar?

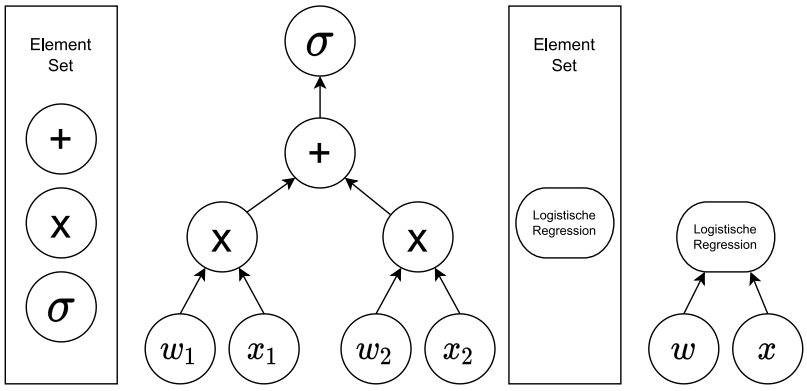

Abbildung 2.1: Diese Abbildung zeigt zwei alternative Berechnungsgraphen für eine logistische Regression ($\sigma(w^T x)$) mit zwei Features. Links wurde eine einzelne Rechenoperation als kleinste Einheit gewählt, wodurch eine Tiefe von 3 entsteht. Rechts wurde die logistische Regression selbst als kleinste Einheit definiert, weshalb die Tiefe lediglich 1 ist. Im Zuge dieser Arbeit wird letztere Definition verwendet. (Bildquelle: [4, S. 7])

Representation Learning schafft für dieses Problem Abhilfe, da es zusätzlich zu der Abbildung $X \mapsto Y$ auch die zu verwendenden Features lernt [4, S. 4]. Ein Rohdatenpunkt x muss also nicht zuerst vorverarbeitet werden, sodass seine numerische Darstellung lediglich die aussagekräftigsten Features enthält, sondern x kann direkt an das System übergeben werden. Je nach Komplexität dieser Features kann aber auch dieser Lernvorgang selbst schwierig sein. Ein Beispiel für ein komplexes Feature ist das bereits angesprochene Katzenohr. Aus vielen einzelnen in Zusammenhang stehenden Pixelwerten ergibt sich erst in der Gesamtheit das Ohr. Das System muss also zuerst diesen Zusammenhang aus Einzelwerten erkennen, bevor es das Katzenohr als einzelnes Feature auffassen kann. Mit einer zunehmenden Anzahl aus Einzelwerten wird das Lernen solcher Features aber zunehmend schwierig für das System, d.h. es braucht in der Regel zunehmend mehr Trainingsdatenpunkte bis es den Zusammenhang erkennt.

Deep Learning setzt an diesem Punkt an, indem es zu erlernende Features in einfachere Teile zerlegt, die erst in Kombination das gesamte Feature ergeben [4, S. 5]. So werden beispielsweise bei der Bilderkennung zunächst Ecken und Kanten erkannt. Diese Elemente werden im nächsten Schritt wiederum zu einfachen geometrischen Figuren kombiniert und dieser Ablauf wiederholt sich, bis am Ende komplexe Features wie das erwähnte Katzenohr erlernt wurden. Nach Goodfellow et al. wird somit eine komplexe Funktion durch die Abfolge mehrerer einfacher Funktionen gebildet [4, S. 5]. Die Anzahl der für diese komplexe Funktion auszuführenden Operationen wird als *Tiefe* eines Netzwerks bezeichnet. Da aber nicht einheitlich geregelt ist, was die kleinste Einheit einer solchen Sequenz ist, und zusätzlich auch alternative Definitionen existieren, ist die Tiefe nicht eindeutig festlegbar [4, S. 7]. Abbildung 2.1 zeigt anhand zweier Berechnungsgraphen diese Uneindeutigkeit. In der Praxis wird für neuronale Netzwerke (siehe Kapitel 2.5) oft eine einfache Definition verwendet: So besteht ein Netzwerk der Tiefe l aus l Schichten (auch *Layer* genannt), wobei der Input-Layer aus dieser Zählweise ausgenommen wird. Für den Schichtindex l gilt üblicherweise $l \in \mathbb{N}$. Wie tief ein Modell sein muss, um als „deep“ zu gelten, ist ebenfalls nicht eindeutig geregelt [4, S. 8]. Zusam-

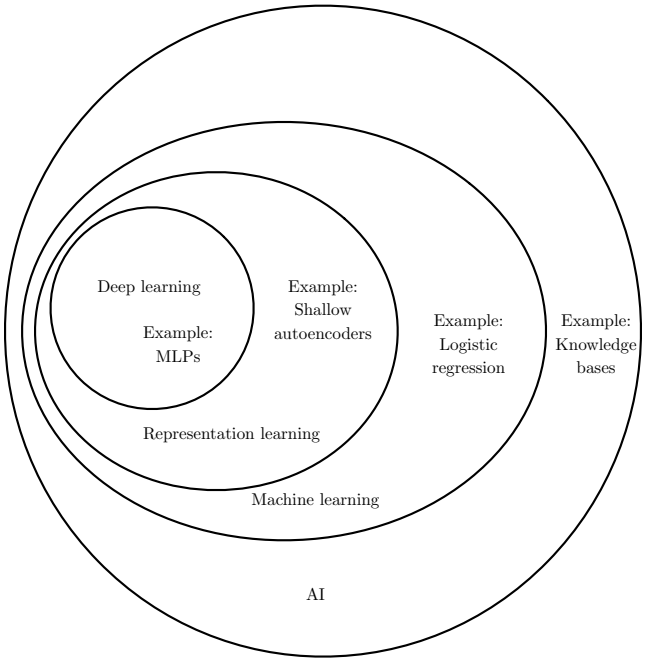

Abbildung 2.2: In diesem Venn-Diagramm wird die Beziehung der bisher besprochenen Konzepte *Machine Learning*, *Representation Learning* und *Deep Learning* veranschaulicht.(Bildquelle: [4, S. 9])

menfassend für Kapitel 2.1 und Kapitel 2.2 zeigt Abbildung 2.2 den Zusammenhang der bisher besprochenen Konzepte. Deep Learning ist also ein Teil des Representation Learnings, was wiederum Teil des Machine Learnings ist. KI (oder englisch AI) ist schlussendlich der Überbegriff für all diese Ansätze.

2.3 Historische Entwicklung des Deep Learnings

Deep Learning hat über die letzten Jahre in zahlreichen Anwendungsgebieten wie Spracherkennung, Computer Vision oder auch in der Bioinformatik neue Standards gesetzt und es wird davon ausgegangen, dass dieser Trend zukünftig sogar noch beschleunigt wird [10]. Die Geschichte hinter Deep Learning und der dabei am häufigsten anzutreffenden Architektur, den neuronalen Netzen, reicht jedoch Jahrzehnte zurück, weshalb in diesem Kapitel eine Übersicht über die Entwicklung des Deep Learnings gegeben werden soll. Aufgrund der großen Anzahl an mitwirkenden Disziplinen und unterschiedlichen Ansätzen erhebt diese historische Zusammenfassung jedoch keinerlei Anspruch auf Vollständigkeit, sondern gibt lediglich einen Überblick über die wichtigsten Meilensteine.

Ausgangspunkt für die Erforschung von Deep Learning und neuronalen Netzen war das Bestreben, ein System zu bauen, welches das menschliche Gehirn simuliert [11, S. 5]. Bereits Aristoteles befasste sich im 4. Jahrhundert vor Christus mit dem rationalen Teil des menschlichen Verstandes und entwickelte eine Menge von Gesetzen, die korrektes logisches Schließen erlauben sollte [12, S. 27]. Über die Jahrhunderte entwickelte sich

die Erkenntnis, dass das Gehirn aus einzelnen Neuronen besteht, die über Synapsen miteinander verknüpft sind. 1949 prägte der kanadische Psychologe Donald O. Hebb die Regel „What fires together, wires together“, was bedeutet, dass die Verbindung zwischen Neuronen umso mehr verstärkt wird, je häufiger sie miteinander aktiv sind [11, S. 7]. Diese Lernregel, die mathematisch als $\Delta w_i = \eta x_i y$[3] beschrieben werden kann, ist die älteste Lernregel für künstliche neuronale Netze [13, S. 268]. Aus diesem Grund wird Hebb auch gelegentlich als „Vater der neuronalen Netzwerke“ bezeichnet [14, S. 81].

Bereits sechs Jahre zuvor wurde von Warren McCulloch und Walter Pitts mit dem sogenannten McCulloch-Pitts-Neuron (kurz MCP-Neuron) ein einfaches Neuronenmodell vorgestellt, das über erregende und hemmende Inputs einen binäre Output liefern konnte [15]. Beim MCP-Neuron wird nur 1 ausgegeben, wenn die gewichtete Summe der erregenden Inputs größer als ein festgelegter Schwellwert ist und zusätzlich kein hemmender Input anliegt. Dieses Modell war jedoch noch nicht lernfähig, erst spätere Ansätze konnten ihre Gewichte über ein Training anpassen [16, S. 9]. 1958 veröffentlichte Rosenblatt mit seinem Perceptron ein lernfähiges Neuronenmodell, das einen visuellen Stimulus über einen Sensor aufnehmen und mittels einer Association Area mit einem Output verbinden konnte [17]. Das Perceptron ist den Neuronen in neuronalen Netzen sehr ähnlich, ein wichtiger Unterschied ist jedoch, dass moderne Neuronen auf die gewichtete Summe der Inputs eine nichtlineare Aktivierungsfunktion anwenden (siehe Kapitel 2.4). Die Arbeiten von McCulloch und Pitts, Hebb und Rosenblatt werden heute der „ersten“ Welle des Deep Learnings zugerechnet, die als Teil der Kybernetik verstanden wird [4, S. 12f]. 1969 wurde von Minsky und Papert jedoch nachgewiesen, dass Perceptronen stark eingeschränkt sind und ein einzelnes einfaches Perceptron mit zwei Eingängen beispielsweise nicht erkennen kann, ob sich seine beiden Eingänge voneinander unterscheiden (XOR-Problem) [18]. Für mehrschichtige Perceptron-Netze, sogenannte *Multilayer Perceptrons (MLP)*, galt diese Erkenntnis zwar nicht, trotzdem folgte ein starker Rückgang der Forschung an neuronalen Netzen [12, S. 45]. Diese Phase ist auch als „the AI Winter“ bekannt [13, S. 270].

In den 1980er-Jahren startete unter dem Namen Konnektionismus die zweite Welle des Deep Learnings [4, S. 12]. Die zentrale Idee des Konnektionismus ist, dass eine große Anzahl einzelner Einheiten intelligentes Verhalten erreichen kann, wenn sie in einem Netzwerk verbunden wird [4, S. 16]. Mehrere Forschungsgruppen wandten sich dem Backpropagation-Algorithmus (siehe Kapitel 2.6), der auch heute noch die vorherrschende Basis des Trainings von neuronalen Netzen darstellt, zu [12, S. 48]. Die erste Anwendung von Backpropagation im Kontext von mehrschichtigen neuronalen Netzen beschrieb Werbos 1982 [19], nachdem er die Idee schon 1974 im Zuge seiner Dissertation vorgestellt hat [20]. Einige Jahre später verstärkte ein Paper von Rumelhart, Hinton und Williams [21], das zeigte, dass *Hidden Layers*, also Schichten zwischen Input- und Output-Layer, interne Repräsentationen erlernen können, die Verbreitung des Backpropagation-Algorithmus [16]. Die Autoren nahmen dabei direkt auf das XOR-Problem von Minsky und Papert Bezug, indem sie erklärten zu glauben, dass sie einen Lernalgorithmus gefunden haben, der dieses Problem lösen kann [21]. 1980 veröffentlichte Fukushima mit seinem Neocognitron [22] eine Architektur, die als Wegbereiter für Convolutional Neural Networks[4] gilt [13, S. 271]. Wenn auch das Neocognitron nicht

[3] η bezeichnet die sogenannte *Lernrate*, die in Kapitel 2.6 genauer betrachtet wird.

[4] Convolutional Neural Networks (CNN) sind eine Architektur, die vor allem im Bereich Computer

mittels Backpropagation trainiert wurde, so ist es nach Schmidhuber das erste neuronale Netzwerk, das das Attribut „deep“ verdient [16]. In den 90ern wurden schließlich signifikante Fortschritte in dem für diese Arbeit wichtigen Bereich der Sequenzverarbeitung erzielt. So wurden die mathematischen Schwierigkeiten (*Vanishing und Exploding Gradients*) beim Modellieren von langen Sequenzen unter anderem von Hochreiter in [23] beschrieben. 1997 stellten Hochreiter und Schmidhuber schließlich das sogenannte *Long Short-Term Memory (LSTM)* vor, welches das Problem des Vanishing Gradients verringert [24]. LSTM wurde seither zu einem Standard bei der Sequenzmodellierung und auch in dieser Arbeit wird diese Variante der RNN eingesetzt. Zu diesem Zeitpunkt ließ die zweite Welle der neuronalen Netzwerke jedoch schon wieder nach. Während andere KI-Ansätze wie Kernel Machines vielversprechende Fortschritte zeigten, scheiterten viele Unternehmen, die auf neuronale Netzwerke setzten, an unrealistischen Erwartungen, was zu einem Rückgang von Investitionen führte [4, S. 17].

2006 stellten Hinton, Osindero und Teh eine effiziente Methode zum Training von mehrschichtigen Netzwerken vor [25], was einen erneuten Forschungsaufschwung im Bereich der neuronalen Netze auslöste. Diese dritte und bis ins heutige Jahr 2022 immer noch andauernde Welle prägte schlussendlich den Begriff „Deep Learning“, da erstmals wirklich tiefe Netze trainiert werden konnten [4, S. 17]. Entscheidende Faktoren, die diese Entwicklung begünstigen, sind die zunehmende Verfügbarkeit von Daten und die immer höher werdende Rechenleistung, wodurch immer größere und mächtigere Netzwerke trainiert werden können. Als Resultat all dieser Entwicklungen stellen neuronale Netzwerke heute den leistungsfähigsten KI-Ansatz dar [4, S. 18].

2.4 Das Neuron

Das künstliche Neuron ist der Grundbaustein von neuronalen Netzwerken, die aus mehreren Schichten miteinander verbundener Neuronen bestehen. Wie beim McCulloch-Pitts-Neuron sind auch heute verwendete Neuronenmodelle in ihrer Grundidee natürlichen Neuronen sehr ähnlich. So werden bei einem natürlichen Neuron chemische Signale über sogenannte Dendriten aufgenommen und über den Zellkörper weitergeleitet. Wird ein gewisser Reizschwellwert überschritten, wird über das Axon, ein langer Zellfortsatz mit einer baumartigen Endstruktur, ein Signal an die folgenden Neuronen abgegeben [13, S. 32f]. Innerhalb eines Neurons werden Signale elektrisch übertragen, für den Übersprung über den synaptischen Spalt zwischen zwei Neuronen werden diese elektrischen Signale mittels Neurotransmitter in chemische Signale übertragen [13, S. 256f]. Abbildung 2.3 zeigt den schematischen Aufbau eines solchen natürlichen Neurons.

Ein künstliches Neuron besitzt ebenfalls eingehende Werte (ähnlich den Dendriten) und ausgehende Werte (ähnlich der Weiterleitung über das Axon). Der Input entspricht dabei einem n_x-dimensionalen Vektor, der die n Features eines Datenpunktes x darstellt. Für jeden Input gilt also $x \in \mathbb{R}^{n_x}$. Die Dimensionalität des finalen Outputs eines Neurons bzw. eines neuronalen Netzwerks wird als n_y bezeichnet. Im Falle eines einzelnen Neurons gilt dabei $n_y = 1$, bei neuronalen Netzwerken zur Klassifikation mehrerer Klassen entspricht n_y hingegen in der Regel der Anzahl der Klassen. Ein künstliches Neuron ist schlussendlich eine Funktion, die Anhand eines Inputs x einen Output $\hat{y} \in \mathbb{R}^{n_y}$ be-

Vision Anwendung findet.

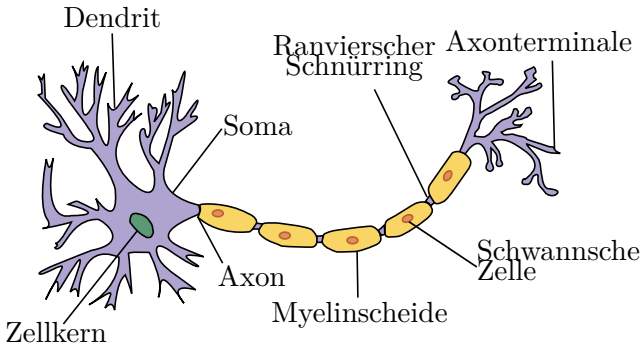

Abbildung 2.3: Vereinfachter Aufbau eines natürlichen Neurons. (Bildquelle: [79])

rechnet. $\hat{y}$ ist nicht mit den Labels $y \in Y$ zu verwechseln, da es sich bei $\hat{y}$ um einen *Prädiktionswert* handelt, den das Neuron bzw. das neuronale Netzwerk ausgibt [9, S. 178]. Im Training weicht dieser Prädiktionswert üblicherweise von dem jeweiligen Label ab, wobei versucht wird, diesen Fehler zu reduzieren. Bei der Inferenz gibt es keine Labels mehr, weshalb in dieser Phase auch weniger Verwechslungsgefahr besteht. Konkret gelten für ein Neuron folgende Gleichungen:

$$z = w^\mathsf{T} x + b \tag{2.1}$$

$$\hat{y} = a = g(z). \tag{2.2}$$

Gleichung 2.1 beschreibt die gewichtete Inputsumme $z \in \mathbb{R}$ eines Neurons. Dabei wird zunächst das Skalarprodukt von x und einem ebenfalls n_x-dimensionalen *Gewichtsvektor* $w \in \mathbb{R}^{n_x}$ gebildet und anschließend ein sogenannter *Bias* $b \in \mathbb{R}$ addiert. Da es sich sowohl bei w als auch bei x normalerweise um Spaltenvektoren handelt, muss w transponiert werden [7, S. 117]. Das Skalarprodukt stellt eine gewichtete Summe aller Inputfeatures x_i, wobei $i \in \mathbb{N}_0$ das i-te Feature des Inputvektors beschreibt, und der jeweiligen Gewichte w_i dar. Es gilt also $z = \sum_{i=1}^{n_x} w_i x_i + b$. w und b sind dabei Parameter, die im Zuge eines Trainings so angepasst werden, dass der Vorhersagefehler des Neurons minimiert wird (siehe Kapitel 2.6). Bis zu diesem Punkt handelt es sich bei dem Neuron um eine *lineare Regression* [4, S. 106].

Damit ein neuronales Netzwerk komplexere, nichtlineare Funktionen darstellen kann, braucht es in den Neuronen zusätzlich einen nichtlinearen Teil. Um dies zu veranschaulichen, werden in Gleichung 2.3 zwei hintereinander folgende Neuronen mit lediglich linearer Funktionalität betrachtet. $a^{[1]}$ ist dabei der Output des ersten Neurons, $a^{[2]}$ die des zweiten. Der Index in den eckigen Klammern beschreibt die Schicht des neuronalen Netzwerks. Aus Gründen der Übersichtlichkeit wird angenommen, dass $n_x = 1$, womit beide Gewichte $w^{[1]}$ und $w^{[2]}$ einer einzelnen reellen Zahl entsprechen.

$$a^{[2]} = w^{[2]}a^{[1]} + b^{[2]} = w^{[2]}(w^{[1]}x + b^{[1]}) + b^{[2]} = w^{[2]}w^{[1]}x + (w^{[2]}b^{[1]} + b^{[2]}) = w'x + b' \tag{2.3}$$

Es wird ersichtlich, dass eine Verkettung von Neuronen ohne nichtlinearer Funktion zu einer weiteren linearen Funktion führt, wodurch tiefere Netzwerke sinnlos werden würden. Die deshalb benötigte nichtlineare Funktion, die sogenannte *Aktivierungsfunktion*,

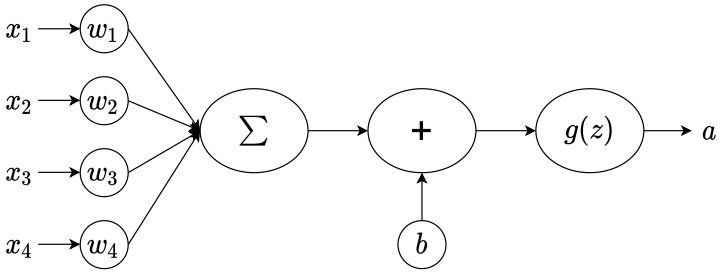

Abbildung 2.4: Schematische Darstellung eines einzelnen künstlichen Neurons.

wird in Gleichung 2.2 dargestellt und als $g(z)$ bezeichnet[5]. Beispiele für beliebte Aktivierungsfunktionen sind die *logistische Sigmoid-Funktion*, *Tangens Hyperbolicus (tanh)* und *ReLU* [7, S. 293]. Da aktuelle Trainingsverfahren auf dem Gradientenabstiegverfahren basieren (siehe Kapitel 2.6) müssen Aktivierungsfunktionen differenzierbar sein. Auch Funktionen wie die Heaviside-Funktion, die an allen differenzierbaren Stellen eine Steigung von 0 haben, können nicht verwendet werden. Die tanh- und die ReLU-Funktionen sind Standardaktivierungsfunktionen für Neuronen in Hidden Layers, weshalb erst in Kapitel 2.5 gesondert auf diese eingegangen wird. Anzumerken ist an dieser Stelle, dass bei Gleichung 2.2 $a = \hat{y}$ gilt. Der Prädiktorwert $\hat{y}$ ist der Output eines neuronalen Netzwerkes, da das einzelne Neuron in diesem Kapitel aber keine nachfolgenden Neuronen besitzt, ist a gleich dem endgültigen Ergebnis. Bei neuronalen Netzwerken mit mehreren Schichten entspricht a jedoch nur dem Output der aktuellen Schicht, der wiederum als Input der nächsten Schicht dient. Abbildung 2.4 zeigt zusammenfassend den gesamten Aufbau eines künstlichen Neurons.

Zum Abschluss der Betrachtung des einzelnen Neurons soll noch auf einen weiteren Vorteil von nichtlinearen Aktivierungsfunktionen eingegangen werden. Wird eine logistische Sigmoid-Funktion als Aktivierungsfunktion verwendet, kann der Output $\hat{y}$ aus probabilistischer Perspektive bei einer binären Klassifikationsaufgabe als Wahrscheinlichkeit für eine Klassenzugehörigkeit interpretiert werden [4, S. 137]. Die in Abbildung 2.5 dargestellte logistische Funktion ist dabei folgendermaßen definiert:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{2.4}$$

$\sigma(z)$ kann Werte zwischen 0 und 1 annehmen, weshalb $\hat{y}$ direkt als Wahrscheinlichkeit $\hat{p}(y = 1|x)$ interpretiert werden kann. Mit anderen Worten: $\hat{y}$ gibt an, wie wahrscheinlich es ist, dass ein Datenpunkt x der positiven Kategorie zuzurechnen ist. Es gilt Gleichung 2.5:

$$\hat{y} = \begin{cases} 0 & \text{wenn } \hat{p} < 0.5 \\ 1 & \text{wenn } \hat{p} \geq 0.5. \end{cases} \tag{2.5}$$

[5]Wenn bei Regressionsaufgaben auch negative Zahlen ausgegeben werden können, wird auf eine Aktivierungsfunktion im Output-Layer verzichtet. Sollten nur positive Zahlen vorhergesagt werden, kann hingegen beispielsweise eine ReLU verwendet werden. Der Wegfall von Aktivierungsfunktionen ist also nur in bestimmten Situation sinnvoll. [7, S. 294]

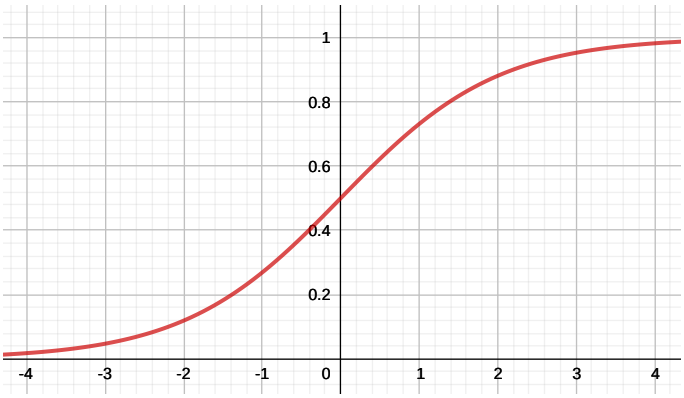

Abbildung 2.5: Darstellung eines Ausschnitts der logistischen Funktion.

Ein einzelnes Neuron mit der logistischen Funktion als Aktivierungsfunktion wird auch als *logistische Regression* bezeichnet. Diese Art der Regression zeigt, dass sich einige Regressionsalgorithmen auch zur Klassifikation einsetzen lassen (und umgekehrt) [7, S. 144]. Bei dem Erlernen der Entschlüsselungsfunktion und dem neuronalen KPA selbst handelt es sich ebenfalls um Klassifikationsprobleme. Da bei diesen aber mehr als zwei Klassen vorhanden sind (jeder Buchstabe entspricht einer eigenen Klasse), braucht es eine verallgemeinerte Variante der logistischen Funktion, die sogenannte *Softmax-Funktion*. Für solch eine Klassifikation braucht es jedoch mehrere Neuronen, weshalb erst in Kapitel 2.5 näher darauf eingegangen wird.

2.5 Feedforward Neural Networks (FFNN)

Nachdem im letzten Kapitel ein einzelnes Neuron isoliert betrachtet wurde, soll nun ein gesamtes neuronales Netzwerk in einer einfachen Feedforward-Architektur beschrieben werden. Feedforward Neural Networks gelten als Fundament des Deep Learnings, da sie nicht nur selbst für viele Probleme eingesetzt werden können, sondern auch den Ausgangspunkt für spezialisiertere Netzwerkarchitekturen wie CNN darstellen [4, S. 163]. FFNN erhalten ihren Namen aufgrund der Tatsache, dass der Informationsfluss im Gegensatz zu beispielsweise RNN lediglich vorwärts gerichtet ist [4, S. 163]. Der Ablauf startet dabei bei dem Input x und gelangt über die Hidden Layers zum Output $\hat{y}$. Zu keinem Zeitpunkt wird dabei wie bei einem RNN ein Output wieder zurück in die gleiche oder eine frühere Schicht geleitet. Ein weiterer in diesem Kontext häufig auftretender Begriff ist *Multilayer Perceptron (MLP)*. Ein MLP beschreibt dabei ursprünglich eine Architektur von mehreren aufeinander folgenden Perceptronen, mit der unter anderem das XOR-Problem von Minsky und Papert gelöst werden kann [26, S. 421]. Die heutige Verwendung dieses Begriffs ist jedoch nicht ganz eindeutig: Während Goodfellow et al. [4, S. 163] und Murphy [26, S. 419] MLP als Synonym für FFNN sehen, beschreibt Géron ein MLP als eine bestimmte Variante von FFNN [7, S. 290]. In dieser Arbeit wird die Sichtweise von Goodfellow et al. und von Murphy übernommen und beide Begriffe werden als äquivalent betrachtet.

FFNN besitzen einen schichtbasierten Aufbau. Ein Netzwerk mit L Schichten besitzt

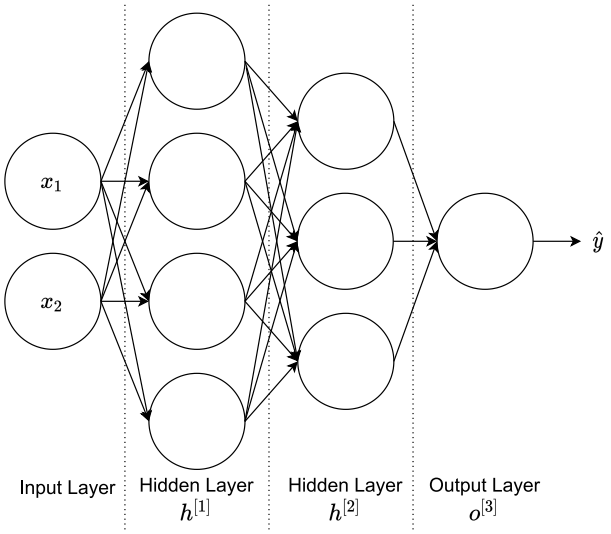

Abbildung 2.6: Ein neurales Netzwerk mit insgesamt drei Schichten, wovon zwei Hidden Layers sind.

dabei $L-1$ Hidden Layer und einen Output-Layer. Im Zuge dieser Arbeit wird die l-te Schicht eines Netzwerks mittels eines hochgestellten Schichtindex in eckigen Klammern beschrieben (der Input-Layer wird dabei üblicherweise nicht mitgezählt). Beispielsweise wird die Gewichtsmatrix, die alle Gewichte für Verbindungen der Schicht $l-1$ zu Schicht l zusammenfasst, als $W^{[l]}$ bezeichnet. Mit dem Schichtindex $l \in \mathbb{N}$ werden somit die einzelnen Schichten eines Netzwerks durchnummeriert. Die Dimension $n_h^{[l]} \in \mathbb{N}$ beschreibt die Anzahl der Neuronen (auch *Hidden Units* genannt) eines Hidden Layers l. Jedes Neuron einer Schicht ist mit jedem Neuron der folgenden Schicht verbunden, was als *Fully Connected Layer*, kurz *FC Layer*, bezeichnet wird [7, S. 290]. Dadurch gilt für eine Gewichtsmatrix $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$. Im Zusammenhang mit einem Biasvektor $b^{[l]} \in \mathbb{R}^{n^{[l]}}$ ergibt sich somit für den Output $a^{[l]}$ der Schicht l analog zu der Berechnung des Outputs eines einzelnen Neurons

$$a^{[l]} = g^{[l]}(W^{[l]}a^{[l-1]} + b^{[l]}). \tag{2.6}$$

Abbildung 2.6 zeigt ein FFNN mit zwei Hidden Layers und einem Output-Layer. Jedes Neuron ist dabei mit allen Neuronen der folgenden Schicht verbunden, es ist also jede Schicht ein FC Layer.

Als Aktivierungsfunktionen für Neuronen in FFNN werden üblicherweise die bereits in Kapitel 2.4 angesprochene ReLU und die tanh-Funktion verwendet. Tanh ist eine sigmoide Funktion, die der logistischen Funktion sehr ähnlich ist. Durch ihre Definition als

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1 \tag{2.7}$$

wird ersichtlich, dass es sich bei der tanh-Funktion um eine skalierte logistische Funktion handelt [80]. Tanh löste ab Mitte der 1990er die logistische Funktion als Standardak-

tivierungsfunktion ab und wurde als solche auch bis in die 2010er-Jahre beibehalten [81]. Da es sich jedoch um eine gesättigte Funktion handelt, die sich -1 und 1 für kleine/-große x annähert, kann beim Training das Problem des Vanishing Gradients, also des verschwindenden Gradienten, auftreten [26, S. 442f]. Wie in Abbildung 2.7 ersichtlich, wird der Gradient bei der Annäherung an -1 und 1 minimal. Über mehrere Schichten kann es durch den im folgenden Kapitel erklärten Backpropagation-Algorithmus passieren, dass in frühen Schichten Parameter kaum mehr merkbar aktualisiert werden und somit kaum mehr Lernfortschritt besteht. Dies ist vor allem beim Training tiefer Netzwerke ein Problem, da der Gradient dabei über zunehmend viele Schichten rückgereicht werden muss.

Die Nutzung einer nicht-sättigenden Funktion wie der ReLU, kurz für Rectified Linear Unit, verringert dieses Problem [26, S. 443]. Die Funktion ist definiert als

$$ReLU(z) = max(0, z). \tag{2.8}$$

Werte kleiner 0 werden auf 0 abgebildet, für Werte größer 0 entspricht die Funktion einer linearen Funktion. Dadurch dass der Gradient im positiven Wertebereich immer 1 ist (lediglich beim Nullpunkt ist die erste Ableitung nicht definiert), kann er nicht verschwindend klein werden. Zusätzlich ist die ReLU auch noch sehr schnell berechenbar, was bei großen Netzwerken mit vielen Neuronen ein weiterer Vorteil ist [7, S. 293]. Der Gradient kann zwar bei der ReLU im positiven Wertebereich nicht mehr verschwinden, im negativen Bereich ist er jedoch 0, weshalb ein anderes Problem auftreten kann: das sogenannte *Dying-ReLU-Problem* [27]. Dieses Problem, bei dem ein Neuron „stirbt“ und keinerlei Output außer 0 mehr liefern kann, tritt auf, wenn die Gewichte des Neurons so angepasst werden, dass die gewichtete Summe für jede Instanz des Trainingsdatensatzes negativ wird [7, S. 337]. Auch für dieses Problem gibt es in Form der *Leaky ReLU*, die als

$$LReLU(z, \alpha) = max(\alpha z, z) \tag{2.9}$$

definiert ist, eine Lösung [28]. Dadurch dass im negativen Teil der Funktion nun die Steigung $\alpha \in \mathbb{R}$ existiert, liefert das Neuron auch bei einer negativen gewichteten Summe einen Output und der Gradient ist ebenfalls nicht 0, wodurch auch ein weiterer Lernfortschritt erzielt werden kann.

Die ReLU wird heute in der Literatur häufig als Standardaktivierungsfunktion empfohlen (z.B. [26, S. 424] oder [4, S. 169]), es gibt jedoch mittlerweile auch verbesserte Funktionen wie ELU [29] und SELU [30], die in bestimmten Szenarien besser funktionieren. Bei RNN ist die Tangens hyperbolicus hingegen noch weitverbreitet, in TensorFlow wird sie beispielsweise standardmäßig für LSTM [31] und GRU [32] eingesetzt.

Bisher war bei einem einzelnen Neuron lediglich eine binäre Klassifikation möglich. Mittels eines Output-Layers mit $k \in \mathbb{N}$ Neuronen und einer *Softmax*-Funktion können jedoch k-Klassen unterschieden werden, wodurch eine sogenannte *Multiclass-Klassifikation* ermöglicht wird [7, S. 295]. Bei einer Multiclass-Klassifikation handelt es sich um eine Klassifikation mit mehr als zwei möglichen Klassen, wobei ein Datenpunkt immer nur einer Klasse angehören kann. Die Softmax-Funktion kann dabei als Generalisierung der für die binäre Klassifikation genutzten logistischen Sigmoid-Funktion (siehe Kapitel 2.4) gesehen werden und gibt eine Wahrscheinlichkeitsverteilung über

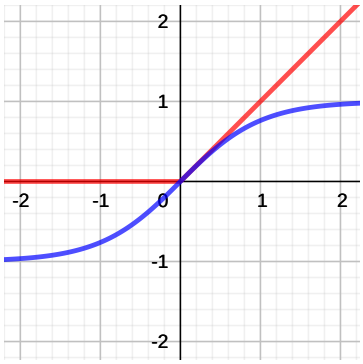

Abbildung 2.7: Darstellung einer ReLU (rot) und der Tangens hyperbolicus (blau).

alle Klassen aus [4, S. 178]. Jeder Datenpunkt kann dabei jedoch nur einer einzigen Klasse angehören, weshalb sich die Klassen gegenseitig ausschließen [7, S. 150f]. Formal betrachtet gilt für die Softmax

$$softmax(z)_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.10}$$

wobei die jeweiligen z durch eine lineare Schicht wie in Gleichung 2.1 beschrieben berechnet werden [4, S. 179].

2.6 Training und Inferenz

Wie bereits beschrieben, braucht es für ein Modell passende Gewichte und Bias-Werte, um einen möglichst kleinen Vorhersagefehler zu erreichen. Diese werden während dem Training auf Basis des Trainingsdatensatzes erlernt. Die Grundidee dabei ist, dass mit initialisierten Parametern Vorhersagen für die Trainingsdatenpunkte getroffen werden, welche anschließend mit den entsprechenden Labels verglichen werden. Im nächsten Schritt werden die bestehenden Parameter so verändert, dass der Unterschied zwischen den Outputs $\hat{y}$ und den tatsächlichen Werten y bei der nächsten Vorhersage geringer wird. Es handelt sich also um eine iterative Annäherung der in Gleichung 2.1 angeführten Parameter w und b an ihren optimalen Wert.

Zunächst braucht es eine Metrik, um beurteilen zu können, wie gut die aktuell gewählten Parameter sind. Dazu wird eine sogenannte *Verlustfunktion (engl. Loss Function)* verwendet. In der Literatur finden sich verschiedene alternative Bezeichnungen für diese Funktion wie beispielsweise *Cost Function*, *Error Function* oder *Objective Function* [4, S. 79]. Teilweise werden diese Begriffe synonym verwendet, wie beispielsweise in dem Standardwerk *Deep Learning* von Goodfellow et al. [4, S. 79], teilweise werden sie aber auch mit unterschiedlicher Bedeutung belegt. Für eine präzisere Unterscheidung wird in dieser Arbeit die Verlustfunktion $\mathcal{L}(\hat{y}, y)$ in Hinblick auf ein einzelnes Trainingspaar $(x^{(i)}, y^{(i)})$ verwendet, die Kostenfunktion J hingegen für einen *Batch* aus $m \in \mathbb{N}$ Trainingspaaren. Ein Batch ist eine Menge von Datenpunkten, die in einem (Trainings-)Schritt miteinander verarbeitet werden. J entspricht dabei dem gemittelten Verlust

über alle m Paare:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}). \tag{2.11}$$

Je nach Aufgabenstellung eignen sich verschiedene Kostenfunktionen unterschiedlich gut. Bei dem in dieser Arbeit vorliegenden Problem der Klassifizierung mit mehreren Klassen bietet sich die *Kreuzentropie* als zu minimierende Funktion an [82]. Gleichung 2.12 beschreibt die Kreuzentropie als Kostenfunktion, wobei $K \in \mathbb{N}$ der Anzahl der möglichen Klassen entspricht.

$$J = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} log(\hat{y}_k^{(i)}) \tag{2.12}$$

Die Kreuzentropie ist ursprünglich ein Konzept der Informationstheorie und wird über zwei diskrete Wahrscheinlichkeitsverteilungen berechnet [7, S. 151][6]. $y_k^{(i)}$ entspricht dabei der tatsächlichen Wahrscheinlichkeit, dass $y^{(i)}$ der Klasse k entspricht, während $\hat{y}_k^{(i)}$ der vom Modell ausgegebenen Wahrscheinlichkeit entspricht. Es wird also die Kreuzentropie über eine für den Trainingsdatensatz gegebene Wahrscheinlichkeitsverteilung und eine von dem Modell gelernte Wahrscheinlichkeitsverteilung berechnet. Je ähnlicher dabei diese beiden Verteilungen durch das Training werden (das Modell strebt nach einer Annäherung an die tatsächliche Verteilung), desto kleiner wird die Kreuzentropie und desto näher liegen die vom Modell ausgegebenen Wahrscheinlichkeiten an der tatsächlichen Verteilung [9, S. 70].

Um dieses Minimierungsproblem zu lösen, basiert der beim Training von neuronalen Netzwerken eingesetzte *Backpropagation-Algorithmus* auf dem *Gradientenverfahren* [13, S. 162]. Dieses Verfahren versucht ein Optimierungsproblem zu lösen, indem es Parameter schrittweise anpasst, um die optimale Lösung zu finden. Wie später noch genauer beschrieben, garantiert das Gradientenverfahren jedoch nicht, dass diese optimale Lösung auch tatsächlich gefunden wird. Abbildung 2.8 zeigt dieses Prinzip anhand der einfachen Beispielsfunktion $f(x) = x^2$. Die Richtung und (teilweise) die Weite der blau eingezeichneten Schritte wird dabei über die Steigung der Funktion beim jeweiligen Ausgangspunkt bestimmt, welche über die erste Ableitung der Funktion bestimmt werden kann. Für das vereinfachte Beispiel aus Abbildung 2.8 gilt deshalb $f'(x) = \frac{dy}{dx} = 2x$. Ist die Steigung negativ, nähert sich ein zunehmendes x dem Minimum an (da bei zunehmenden Wert x $f(x)$ kleiner wird), ist die Steigung hingegen positiv, entfernt sich ein zunehmendes x.

Um die Schrittweite zu regulieren, wird die sogenannte *Lernrate (engl. Learning Rate)* $\eta \in \mathbb{R}^+$ eingesetzt. Ein kleines η bedeutet, dass der Algorithmus möglicherweise viele Schritte durchführen muss, bevor er konvergiert, dafür ist die Gefahr verringert, das Minimum zu „überspringen“. Bei einer großen Lernrate kann der Algorithmus um das Minimum oszillieren und so niemals die optimale Lösung erreichen [7, S. 122].

In dem bisherigen Beispiel hängt der Funktionswert lediglich von x ab, was zwar eine anschauliche Darstellung ermöglicht, aber in der Regel nicht den realen Begebenheiten bei der Verwendung eines neuronalen Netzwerks entspricht. Tatsächlich hängt

[6] Es gibt auch eine Kreuzentropie über zwei stetige Wahrscheinlichkeitsverteilungen, diese ist aber für den vorliegenden Anwendungsfall nicht relevant, weshalb nicht weiter darauf eingegangen wird.

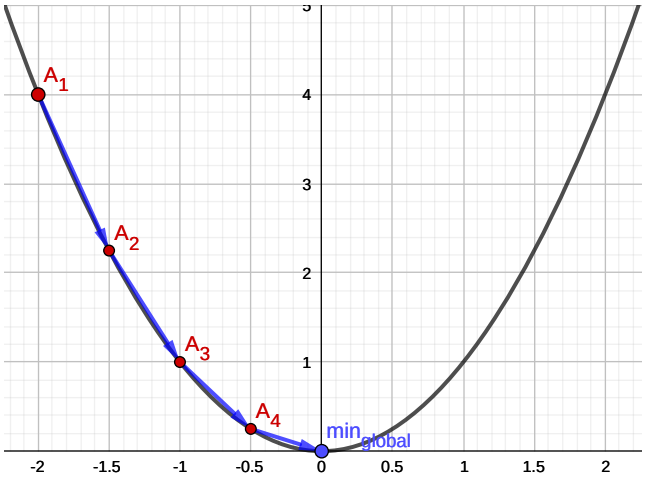

Abbildung 2.8: Anhand der Funktion $f(x) = x^2$ und einem Initialisierungspunkt $A_1 = (-2, 4)$ verdeutlicht die Abbildung die Grundidee des Gradientenabstiegs. Im Zuge von vier Iterationen wird das globale Minimum $(0, 0)$ erreicht, die Lösung für das Minimierungsproblem ist also $x = 0$. (Anmerkung: Die eingezeichneten Schritte dienen lediglich der Illustration und wurden nicht mittels des Gradientenabstiegverfahrens berechnet.)

das Ergebnis der Kostenfunktion von den Gewichten und Bias-Werten ab, wobei es je nach Anzahl der Schichten und der Neuronen pro Schicht sehr viele dieser geben kann. Um den Einfluss eines einzigen Parameters auf die Gesamtkosten berechnen zu können, wird deshalb eine partielle Ableitung benötigt. Um beispielsweise den Anteil eines spezifischen Gewichts $w_{i,j}^{[l]}$ [7] an den Gesamtkosten J zu erhalten, muss J nach $w_{i,j}^{[l]}$ abgeleitet werden: $\frac{\partial J}{\partial w_{i,j}^{[l]}}$. Somit kann der neue Wert von $w_{i,j}^{[l]}$ berechnet werden:

$$w_{i,j}^{[l]} = w_{i,j}^{[l]} - \eta \frac{\partial J}{\partial w_{i,j}^{[l]}}. \tag{2.13}$$

Die Bias-Werte werden ebenso nach demselben Prinzip angepasst. Diese auf der Idee des Gradientenabstiegs basierende „Fehlerrückführung“ auf die Parameter aller Schichten wird als die bereits oben angesprochene *Backpropagation* bezeichnet [12, S. 852].

Das Gradientenverfahren kann mittels verschiedener Ansätze, die sich über die Anzahl der für die Berechnung der Gradienten verwendeten Trainingsdatenpunkte unterscheiden, implementiert werden. Beim *Batch-Gradientenverfahren* wird dabei der gesamte Trainingsdatensatz verwendet. Dies hat zwar den Vorteil, dass das Verfahren bei einer konvexen[8] Kostenfunktion immer konvergiert, allerdings ist es insbesondere für sehr große Datensätze langsam [7, S. 126f]. Sobald das zu trainierende ML-Verfahren alle

[7] $w_{i,j}^{[l]}$ bezeichnet das Gewicht für den j-ten Input von Neuron i der l-ten Schicht in einem neuronalen Netzwerk. Es gilt für die beiden Indices $i, j \in \mathbb{N}_0$.

[8] Werden zwei beliebige Funktionswerte A und B einer konvexen Funktionen f mit einer Geraden g verbunden, liegen alle Funktionswerte zwischen A und B unter dieser Geraden. Da f g somit nicht schneidet, gibt es nur ein globales Minimum und keine lokalen Minima.

Datenpunkte eines Trainingsdatensates verarbeitet hat, gilt eine sogenannte *Epoche* als abgeschlossen. Da beim Batch-Gradientenverfahren immer der gesamte Datensatz verwendet wird, entspricht eine Algorithmusiteration einer Epoche. Der dazu gegensätzliche Ansatz ist das *stochastische Gradientenverfahren (SGD)*, bei dem statt dem gesamten Datensatz nur ein einziger zufälliger Datenpunkt verwendet wird [7, S. 127]. Aufgrund dieser zufallsbasierten Parameteränderungen entwickeln sich die Parameter weit unregelmäßiger als beim Batch-Gradientenverfahren, dafür ist SGD schneller und skaliert besser bei großen Datensätzen. Diese Unregelmäßigkeit sorgt jedoch auch dazu, dass das Minimum nicht genau erreicht wird, sondern eine Oszillation rund um diesen Extrempunkt auftritt [4, S. 286f]. Um dem entgegen zu wirken, kann η im Laufe der Algorithmusiterationen verringert werden, wodurch die Schrittweite und somit die Schwankung um das Minimum abnimmt [4, S. 286f]. Eine dritte Möglichkeit ist ein Kompromiss zwischen den beiden bisher vorgestellten Verfahren: das *Mini-Batch-Gradientenverfahren*. Dabei wird der Trainingsdatensatz in mehrere Mini-Batches aufgeteilt und es wird für jeden Trainingsschritt einer dieser Teildatensätze verwendet. Der Vorteil dabei ist, dass Berechnungen durch Mini-Batches effizienter als bei SGD sind (mehrere Datenpunkte können bei der Verwendung von GPUs über Matrizenoperationen parallel bearbeitet werden [7, S. 130]) und gleichzeitig nicht wie beim Batch-Gradientenverfahren alle Datenpunkte in den Speicher geladen werden müssen [83]. Eine Gemeinsamkeit von SGD und Mini-Batch ist, dass bei beiden pro Epoche mehrere Parameterupdates durchgeführt werden.

Aufbauend auf diesen Grundvarianten des Gradientenverfahren entstanden weitere Optimierungsalgorithmen, die ein schnelleres Training erreichen sollen. Beispiele dafür sind Momentum [33], AdaGrad [34] und RMSProp [35]. Eine Kombination von Adagrad und RMSProp stellten Kingma und Ba 2014 unter dem Namen Adam vor [36]. Im Gegensatz zu Gradientenverfahren mit fixer Lernrate verwendet Adam eine adaptive Lernrate, d.h. es werden individuelle Lernraten für verschiedene Parameter berechnet [36]. Aufgrund seiner Performance wird das Verfahren oftmals als Standardoptimizer für Deep Learning empfohlen (z.B. [37]), weshalb es auch im Zuge dieser Arbeit eingesetzt wird.

Bisher wurden nur konvexe Kostenfunktionen, bei denen das Minimum immer global ist, betrachtet. Es gibt aber ebenfalls Funktionen, bei denen Sattelpunkte und lokale Minima auftreten. Abbildung 2.9 zeigt beispielhaft eine solche Funktion (es handelt sich dabei aber nicht um eine tatsächliche Kostenfunktion). Theoretisch ist es deshalb möglich, dass das Gradientenverfahren nicht bei einem globalen Minimum konvergiert, sondern bei einem lokalen, was nicht der optimalen Lösung entspricht. Aus empirischen und auch theoretischen Arbeiten hat sich jedoch gezeigt, dass diese Gefahr in der Praxis bei großen neuronalen Netzwerken zu vernachlässigen ist, da Trainings fast immer zu Ergebnissen ähnlich guter Qualität führen [10]. Tatsächlich führen auch lokale Minima oft zu guten Ergebnissen, bei Konvergenz beim globale Minimum kann hingegen sogar *Overfitting* (siehe unten) entstehen [38]. SGD und zum Teil auch das Mini-Batch-Gradientenverfahren haben durch ihre im Vergleich zum Batch-Gradientenverfahren stärker oszillierende Parameterentwicklung jedoch eine höhere Chance, lokale Minima wieder zu verlassen [83].

Zusätzlich zu dem Optimierungsverfahren selbst ist auch die richtige Initialisierung der in Gleichung 2.1 angeführten Parameter w und b wichtig. Da die Kostenfunktion von tiefen neuronalen Netzwerken nicht konvex ist, hat die Initialisierung einen entscheiden-

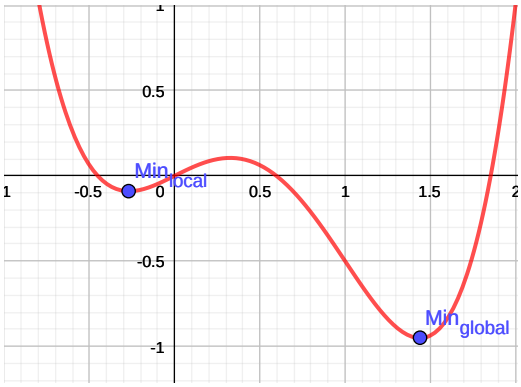

Abbildung 2.9: Eine Funktion mit einem lokalen und einem globalen Minimum. Theoretisch wäre es denkbar, dass das Gradientenverfahren in solch einem lokalen Minimum hängen bleibt.

den Einfluss auf das Training und das Ergebnis [26, S. 446]. Aktuelle Initialisierungsstrategien sind insgesamt sehr einfache, heuristische Verfahren, da die Optimierung von neuronalen Netzwerken und der genaue Einfluss von Parametern darauf noch nicht ausreichend verstanden wird [4, S. 293]. Das einzig wirklich sichere Wissen in diesem Bereich ist, dass Initialparameter bei Neuronen in einer Schicht mit denselben Inputs und der gleichen Aktivierungsfunktion nicht gleich sein dürfen, da dies zu einer „Symmetrie“ der Neuronen führen würde [4, S. 293]. Der Lernalgorithmus würde jedes betroffene Neuron exakt gleich aktualisieren, was zu Folge hätte, dass alle Neuronen dasselbe lernen, was natürlich der Grundidee von neuronalen Netzwerken mit mehreren Neuronen widerspricht. Eine einfache Initialisierung der Gewichte[9] mit 0 oder einer anderen Konstante ist deshalb nicht sinnvoll. Eine beliebte Initialisierungsstrategie war lange Zeit, die Gewichte zufällig aus einer Normalverteilung $\mathcal{N}(0;1)$ zu wählen[7, S. 334]. Glorot und Bengio konnten jedoch 2010 nachweisen, dass Verteilungen mit fixer Varianz unter Umständen (v.a. bei logistischen Aktivierungsfunktionen) zu einem verschwindenden oder explodierenden[10] Gradienten führen kann [39]. Um diese Problematik zu verbessern, schlagen die Autoren eine Initialisierung der Gewichte einer Schicht auf Basis folgender Gleichverteilung vor [39]:

$$U[-\frac{\sqrt{6}}{\sqrt{n^{[l-1]}+n^{[l]}}},\frac{\sqrt{6}}{\sqrt{n^{[l-1]}+n^{[l]}}}]. \tag{2.14}$$

$n^{[l-1]} \in \mathbb{N}$ entspricht dabei der Dimension des Inputs, $n^{[l]} \in \mathbb{N}$ der des Outputs einer Schicht l. Diese mittlerweile *Glorot-* oder *Xavier-Initialisierung* genannte Methode wird von dem für diese Arbeit verwendeten Framework TensorFlow[11] in der Version 2.8.0 standardmäßig für LSTM [31], GRU [32] und Dense Layer [40] verwendet. Für Neuro-

[9] Anmerkung: Dies gilt nur für die Gewichte, bei dem Bias ist es üblich, diesen mit 0 zu initialisieren. Solange die Gewichte zufällig initialisiert werden, ist dies kein Problem.[84]

[10] Der explodierende Gradient (engl. *Exploding Gradient*) ist das Gegenteil des verschwindenden Gradienten. Dabei werden Gewichtsaktualisierungen zu groß, sodass Divergenz auftreten kann.

[11] https://www.tensorflow.org/

nen mit ReLU als Aktivierungsfunktion gibt es ebenfalls eine spezielle Initialisierung (*Initialisierung nach He* [41]), da aber im Zuge der Arbeit lediglich die tanh- und die Softmax-Funktion verwendet werden, wird der TensorFlow-Standard eingesetzt.

Während des Trainings, bei dem ein Modell auf Basis der Trainingsdaten erstellt werden soll, das auch bei unbekannten Daten eine der Aufgabe entsprechende hohe Leistung erbringt, können zwei grundsätzliche Probleme auftreten: *Underfitting* und *Overfitting*. Beim Underfitting ist das Modell zu einfach, um die den Daten zugrundeliegende Struktur zu erlernen. Ein Beispiel, das zu Underfitting führen würde, wäre der Versuch, einen komplexen, nichtlinearen Datensatz mit einer einfachen linearen Regression zu erlernen. Im Wesentlichen bieten sich drei Ansätze an, um das Problem des Underfittings zu verringern[7, S. 30]:

- Einsatz von Verfahren mit mehr Parametern: Mehr Parameter bedeuten mehr Freiheitsgrade, weshalb sich das Modell besser an den Datensatz anpassen kann.
- Verwendung von besseren Features
- Verringerung der Regularisierung: Der Begriff Regularisierung beschreibt Methoden, die die Freiheitsgrade eines Modells verringern. Bei den meisten Verfahren wird der Grad der Regularisierung über sogenannte *Regularisierungsparameter* gesteuert. Wird die Regularisierung reduziert, hat das betroffene Modell mehr Freiheitsgrade, weshalb es sich besser an einen Datensatz anpassen kann.

Beim Overfitting spezialisiert sich das Modell zu stark auf den Trainingsdatensatz und kann deshalb bei unbekannten Daten keine ausreichende Leistung mehr erbringen. Das Modell ist nicht mehr fähig, zu *generalisieren*, da Muster im Trainingsdatensatz erlernt werden, die so nicht bei unbekannten Daten vorkommen. Insbesondere bei verrauschten Trainingsdaten neigen Modelle zum Overfitting [42, S. 122]. Entsprechende Gegenmaßnahmen beim Auftreten von Overfitting sind [7, S. 30]:

- Einsatz von Verfahren mit weniger Parametern
- Training mit mehr Daten
- Verringerung des Rauschens der Daten: Der Begriff *Rauschen* fasst alle Datenfehler zusammen, die es einem ML-Verfahren erschweren, Muster in den Daten zu erkennen. Rauschen kann verringert werden, wenn beispielsweise Datenpunkte, deren Werte sehr weit von den übrigen abweichen, aus dem Trainingsdatensatz entfernt werden.
- Erhöhung der Regularisierung: Über eine Erhöhung der Regularisierungsparameter können dem Modell Freiheitsgrade entzogen werden.

Eine einfache Art der Regularisierung, die auch im Zuge dieser Arbeit verwendet wird, ist die sogenannte *Dropout-Regularisierung* [43]. Dabei wird in jedem Trainingsschritt ein Neuron mit der Wahrscheinlichkeit $p \in \mathbb{R}^+$ ausgelassen, wodurch es temporär nicht an den Berechnungen dieser Iteration teilnimmt. Der Regularisierungseffekt kommt dabei durch die Verringerung der sogenannten *Co-Adaptation* der einzelnen Neuronen zustande [43]. Mit anderen Worten bedeutet dies, dass Gewichte zu bestimmten Vorgängerneuronen nicht übermäßig stark ausgeprägt werden können, da die Vorgängerneuronen in jedem Trainingsschritt unterschiedlich sind. Neuronen müssen also lernen, unabhängig von anderen sinnvolle Features zu erlernen, wodurch sie kontextunabhängiger werden und weniger zu Overfitting neigen [43].

Um Overfitting erkennen zu können, wird vor dem Training üblicherweise ein Teil des vorhandenen Datensatzes für einen *Testdatensatz* zurückgehalten. Diese Daten werden nicht für das Training verwendet, sie sind für das Modell also nach dem Training unbekannt. Ist bei einer anschließenden Evaluierung des Modells mit dem Testdatensatz die Korrektheit der Modellaussagen, die z.B. durch die Accuracy ausgedrückt werden kann, viel geringer als im Training, so liegt Overfitting vor. Erreicht das Modell hingegen im Training schon keine ausreichend korrekte Vorhersageleistung, handelt es sich um Underfitting. Damit bei der Auswahl der Architektur und der *Hyperparameter* kein unbewusstes Overfitting auf den Testdatensatz erfolgt, kann ein zusätzlicher Datensatz, ein sogenannter *Validierungsdatensatz*, verwendet werden. Bei Hyperparameter handelt es sich um Konfigurationsparameter, die nicht während dem Training erlernt werden. Ein Beispiel dafür ist die bereits erwähnte Lernrate. Es können somit verschiedene Modelle mit unterschiedlichen Hyperparameter erstellt werden, die anschließend in Hinblick auf ihre Leistung beim Validierungsdatensatz evaluiert werden. Das Modell mit der besten Leistung bei Verwendung des Validierungsdatensatzes wird ausgewählt und abschließend mit dem Testdatensatz auf die Tauglichkeit für den angedachten Einsatzzweck getestet. Erfüllt das Modell bei dieser finale Evaluierung die gestellten Leistungsanforderungen, kann es zur *Inferenz* in der Produktivumgebung eingesetzt werden. Im Kontext von Deep Learning bedeutet Inferenz, dass ein trainiertes Modell mit unbekannten Daten verwendet wird und auf Basis dieser Vorhersagen trifft. [85].

2.7 Recurrent Neural Networks (RNN)

Recurrent Neural Networks sind der für diese Arbeit wichtigste Netzwerktyp, da sowohl das Lernen der Entschlüsselungsfunktionen als auch die neuronalen KPA mithilfe von RNN realisiert werden. RNN eignen sich insbesondere für sequenzielle Daten wie beispielsweise Sprach- oder Textdaten [10]. Der Begriff „sequenzielle Daten“ beschreibt dabei Daten, die aus mehreren Datenelementen bestehen, wobei jedes Element von anderen Elementen abhängt. Ein Beispiel ist ein einfacher Satz, bei dem jedes Wort semantisch mit den umliegenden Wörtern in Zusammenhang steht. Eine solche Sequenz besteht aus $T \in \mathbb{N}$ diskreten Schritten, den sogenannten *Time-Steps*.

Im Gegensatz zu den bisher betrachteten FFNN fließt bei RNN Information nicht nur von den Inputs Richtung Output, sondern es gibt auch rückgekoppelte Verbindungen, womit ein Output eines Time-Steps $t \in \mathbb{N}$ innerhalb einer Schicht zum Input eines Time-Steps $t+1$ wird. Dieser zweite Time-Step besitzt somit zwei Inputs: Das zweite Element der Inputsequenz $x_{t_2} \in \mathbb{R}^{n_x}$ und den Output des vorherigen Time-Steps. Welcher Output von t in $t+1$ verwendet wird, hängt von der genauen Architektur des Netzwerkes ab, normalerweise handelt es sich dabei aber um den *Hidden State* $h_t \in \mathbb{R}^{n_h}$, der dem Output des Hidden Layers entspricht. $n_h \in \mathbb{N}$ enstpricht bei RNN der sogenannten *Unit Size*, die angibt, wieviele Neuronen innerhalb einer Zelle verwendet werden. Es ist auch möglich, den endgültigen Output $\hat{y}_t$ zu verwenden, für den unter Umständen noch weitere Operationen auf h_t angewendet werden müssen, solche RNN sind jedoch in der Regel schwächer und werden deshalb seltener verwendet [4, S. 370]. Abbildung 2.10 zeigt eine einfache RNN-Zelle in einer kompakten Notation, bei der die Rückverbindung klar ersichtlich ist. Außerdem wird das Netzwerk entlang der Zeitachse aufgerollt dargestellt, womit der Zusammenhang der einzelnen Time-Steps besser illustriert wird.

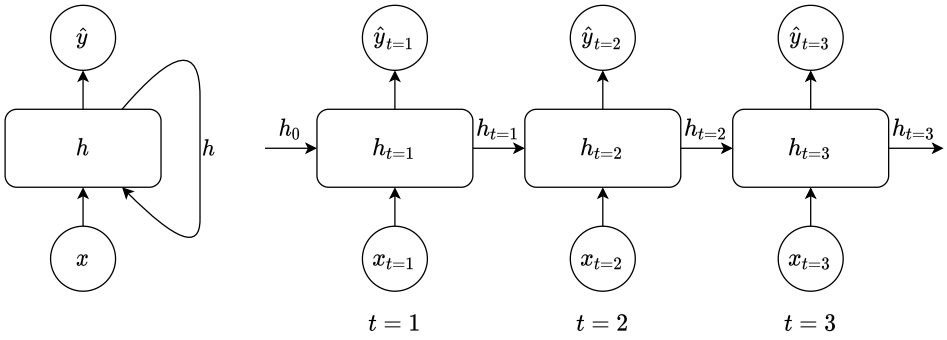

Abbildung 2.10: Darstellung einer einfachen RNN-Zelle. Links ist die kompakte Notation sichtbar, rechts wird diese entlang der Zeitachse aufgerollt.

Da über den Hidden State Informationen von vorherigen Time-Steps weitergereicht werden, besitzt ein RNN gewissermaßen ein Gedächtnis, wobei dieses zumindest in der Grundversion ziemlich kurzzeitig ist [7, S. 504]. h_t wird in dieser Grundversion folgendermaßen berechnet:

$$z_t = W_x x_t + W_h h_{t-1} + b \tag{2.15}$$

$$h_t = g(z_t). \tag{2.16}$$

Wie in einer „herkömmlichen“ FFNN-Schicht wird auch bei einer rekurrenten Schicht zunächst eine gewichtete Summe z_t über die Inputs und einen Bias b berechnet. Wie Gleichung 2.15 zeigt, gibt es jedoch zwei Inputs, x_t und h_{t-1}, weshalb auch zwei Gewichtsmatrizen W_x und W_h benötigt werden (die Größen dieser Matrizen lassen sich genau wie bei den FFNN-Schichten bestimmen). Da die Parameter für jeden Time-Step gleich sind, ist es möglich, das RNN mit Sequenzen variabler Länge zur verwenden [4, S. 363]. Anschließend wird in Gleichung 2.16 wieder eine Aktivierungsfunktion g auf z_t angewandt, wobei dies, wie bereits angeführt, bei RNN normalerweise tanh ist. Bei den einfachen RNN-Zellen entspricht dieses h_t bereits dem Output $\hat{y}_t$ eines Time-Steps.

RNN können auf verschiedene Arten eingesetzt werden:

- Aligned Sequence-to-Sequence (Seq2Seq): Es wird auf Basis einer Inputsequenz eine ebenso lange Outputsequenz erzeugt [26, S. 501f]. Beispiel: Erlernen der Entschlüsselungsfunktion, wobei der ausgegebene Plaintext genauso lang wie der eingegebene Ciphertext ist (z.B. bei Vigenère).
- Unaligned Sequence-to-Sequence (Seq2Seq mit Encoder-Decoder): Es wird auf Basis einer Inputsequenz eine Outputsequenz anderer Länge erzeugt [26, S. 502]. Beispiel: Maschinelle Übersetzung, bei der ein Satz in der Zielsprache eine andere Wortanzahl als in der Ausgangssprache besitzt.
- Vector-to-Sequence (Vec2Seq): Auf Basis eines einzelnen Input-Time-Steps wird eine Sequenz erzeugt [7, S. 505]. Beispiel: Ein generatives Modell, das auf Basis einer eingegebenen Note ein passendes Musikstück erschafft.
- Sequence-to-Vector (Seq2Vec): Auf Basis einer Inputsequenz wird ein einzelner Vektor ausgegeben [7, S. 505]. Beispiel: Eine Sentiment-Analyse, bei der eine

schriftliche Kundenbewertung auf eine Zufriedenheitsskala von 1 bis 5 gemappt wird.

In dieser Arbeit entsprechen die eingesetzten Modelle dem aligned Seq2Seq-Ansatz. Bei der Spaltentransposition, der Playfair- und der Hill-Chiffre wird auf ein *bidirektionales RNN (BRNN)* [44] als Basis für das aligned Seq2Seq-Modell zurückgegriffen, wodurch der Informationsfluss nicht nur vom ersten zum letzten Time-Step gerichtet ist, sondern gleichzeitig auch vom letzten zum ersten.

Der Backpropagation-Algorithmus wird bei RNN als *Backpropagation Through Time* bezeichnet, da die herkömmliche Backpropagation rückwärts über alle Time-Steps durchgeführt wird [7, S. 506]. Ein daraus folgendes, zunehmendes Problem beim Training mit langen Sequenzen ist jedoch das Vanishing Gradient Problem, wie Hochreiter 1991 gezeigt hat [23]. Aus diesem Grund und auch aufgrund der „Vergesslichkeit“ von Standard-RNN-Zellen wurden später mächtigere RNN-Zellen wie LSTM und GRU vorgestellt.

Long Short-Term Memory (LSTM) wurde, wie schon in Kapitel 2.3 erwähnt, 1997 von Schmidhuber und Hochreiter vorgestellt [24]. Im Gegensatz zu der oben betrachteten Standardzelle besitzt eine LSTM-Zelle zusätzlich einen *Cell-State* $c \in \mathbb{R}^{n_h}$ und drei *Control Gates* (Input-, Forget- und Output-Gate). Abbildung 2.11 gibt einen schematischen Überblick über die Struktur einer solchen LSTM-Zelle. In der Literatur finden sich verschiedene Notationen, aufgrund ihrer Einfachheit und Konsistenz mit bisherigen Notationen wird in der Abbildung und den folgenden Gleichungen die Notation an Murphy angelehnt [26, S. 507]. Folgende Gleichungen beschreiben die Operationen einer LSTM-Zelle[12]:

$$F_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{2.17}$$

$$I_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{2.18}$$

$$O_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{2.19}$$

$$\tilde{c_t} = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.20}$$

$$c_t = c_{t-1} \odot F_t + \tilde{c_t} \odot I_t \tag{2.21}$$

$$\hat{y}_t = h_t = tanh(c_t) \odot O_t \tag{2.22}$$

Die Gleichungen 2.17 bis 2.19 Beschreiben das Forget-Gate $F_t \in \mathbb{R}^{n_h}$, das Input-Gate $I_t \in \mathbb{R}^{n_h}$ und das Output-Gate $O_t \in \mathbb{R}^{n_h}$. Bei allen dreien handelt es sich um herkömmliche FC Layer mit einer logistischen Aktivierungsfunktion σ. Jedes Gate besitzt eigene Gewichtsmatrizen $W_x \in \mathbb{R}^{n_h \times n_x}$ und $W_h \in \mathbb{R}^{n_h \times n_h}$ und einen eigenen Bias b (zur besseren Unterscheidung haben diese eine tiefgestellte Markierung). Da die Ausgabewerte dieser Funktion im Intervall $(0, 1)$ liegen, können die Ergebnisse prozentuell interpretiert werden. Wie in Gleichung 2.21 ersichtlich, bestimmt F_t, welcher Anteil des

[12] $\odot$ entspricht der komponentenweisen Multiplikation

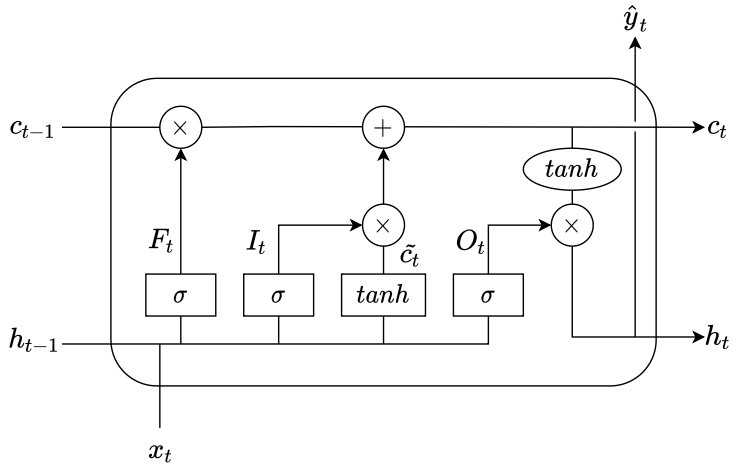

Abbildung 2.11: Schematischer Aufbau einer LSTM-Zelle. Rechteckige Elemente stellen dabei FC Layer mit der entsprechenden Aktivierungsfunktion dar, runde Elemente sind komponentenweise Operationen. (Bildquelle [86])

bisherigen Cell States für den neuen weiterverwendet wird und I_t legt den dazu addierten Anteil des Candidate Cell States $\tilde{c}_t \in \mathbb{R}^{n_h}$ fest. $\tilde{c}_t$ (Gleichung 2.20) legt dabei den Grundbeitrag des aktuellen Time-Steps zu einem neuen Cell State c_t fest. Auch dabei handelt es sich um einen FC Layer mit einer Aktivierungsfunktion, üblicherweise wird dafür tanh eingesetzt. Für den Output des aktuellen Time-Steps t wird ebenfalls tanh auf c_t angewandt (Gleichung 2.22). Das Output-Gate O_t regelt im letzten Schritt welcher Anteil des Ergebnisses der tanh als Hidden State h_t an den nächsten Time-Step weitergegeben wird. Je nach Architektur dient h_t zusätzlich als Output $\hat{y}_t$ und wird an die nächste Schicht im Netzwerk weitergegeben (beispielsweise bei RNN mit zwei aufeinanderfolgenden LSTM-Zellen).

Die Stärke von LSTM liegt in der Verwendung des Cell States und der steuernden Gates. Durch das Input-Gate wird der Cell State vor irrelevanten Inputs geschützt und das Output-Gate verhindert, dass der nächste Time-Step durch derzeit irrelevante gespeicherte Informationen negativ beeinflusst wird [24]. Das Forget Gate wurde 2000 von Gers, Schmidhuber und Cummins ergänzt, da beim Training mit sehr langen Sequenzen bei der ursprünglichen LSTM-Architektur von Hochreiter und Schmidhuber Probleme durch fehlende Reset-Möglichkeiten auftreten können [45]. Durch dieses zusätzliche Gate lernt das LSTM zu vergessen, weshalb nicht mehr relevante Information wieder aus dem Cell State entfernt werden können.

Einen anderen, ebenfalls Gate-basierten Zelltyp stellten Cho et al. 2014 vor [46]. Die *Gated Reccurent Unit (GRU)* ist von LSTM inspiriert, jedoch einfacher zu berechnen und zu implementieren. Abbildung 2.12 zeigt den Aufbau einer GRU-Zelle, wobei sofort ersichtlich wird, dass im Gegensatz zu LSTM lediglich zwei Gates, das Reset-Gate $R_t \in \mathbb{R}^{n_h}$ und das Update-Gate $Z_t \in \mathbb{R}^{n_h}$, eingesetzt werden.

Dieser Zelltyp ist folgendermaßen definiert:

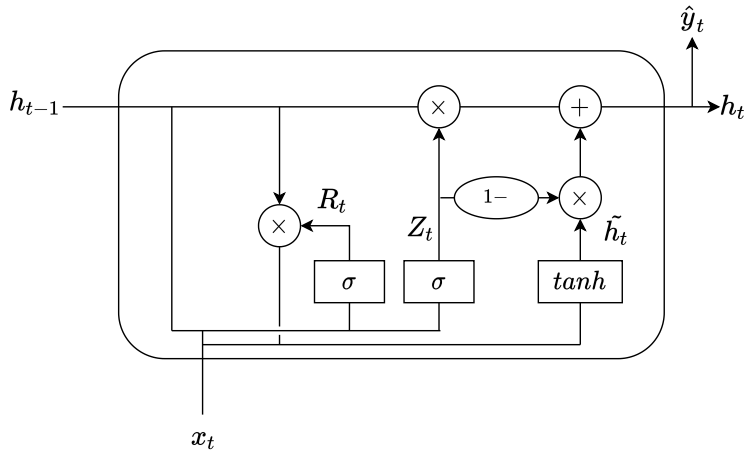

Abbildung 2.12: Schematischer Aufbau einer GRU-Zelle. (Bildquelle [26, S. 506])

$$R_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \tag{2.23}$$

$$Z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \tag{2.24}$$

$$\tilde{h_t} = tanh(W_{xh}x_t + W_{hh}(R_t \odot h_{t-1}) + b_h) \tag{2.25}$$

$$h_t = Z_t \odot h_{t-1} + (1 - Z_t) \odot \tilde{h_t} \tag{2.26}$$

Die Gleichungen 2.23 und 2.24 zeigen, dass sich die Berechnung der Gates nicht von den LSTM-Gates unterscheidet. $R_t \in \mathbb{R}^{n_h}$ bestimmt dabei, wie viel Information der vorhergehenden Time-Steps für die Berechnung des aktuellen Candidate Hidden States $\tilde{h_t} \in \mathbb{R}^{n_h}$ verwendet werden soll. Bei einem R_t nahe 0 würde $\tilde{h_t}$ (Gleichung 2.25) fast ausschließlich vom Input x_t abhängen. Das Update-Gate $Z_t \in \mathbb{R}^{n_h}$ bestimmt hingegen, das Verhältnis von h_{t-1} und $\tilde{h_t}$ für das Update des Hidden States (Gleichung 2.26). Laut Cho et al. ermöglicht gerade dieses Update-Gate das Lernen von Abhängigkeiten über lange Sequenzen und ist damit dem LSTM-Cell-State c_t ähnlich [46].

Insgesamt lässt sich sagen, dass sowohl LSTM als auch GRU große Verbesserungen gegenüber Standard-RNN-Zellen darstellen. Welche der beiden Gate-basierten Zellen verwendet wird, macht meist keinen signifikanten Unterschied wie Greff et al. zeigen [47]. In der Praxis ist es deshalb oft empfehlenswert, beide Varianten zu testen.

2.8 Encoder-Decoder

Wie schon im vorherigen Kapitel angemerkt, basieren Encoder-Decoder-Architekturen auf herkömmlichen RNN. Während diese eine Sequenz der Länge $T \in \mathbb{N}$ auf eine ebenso lange Zielsequenz abbilden können, scheitern sie bei der Aufgabe, eine Sequenz der Länge T auf eine Sequenz der Länge $T' \neq T$ abzubilden. Gerade im Bereich der maschinellen

Übersetzung (*Neural Machine Translation*) ist dies jedoch essenziell, da Sätze in einer Zielsprache häufig länger oder kürzer als in der Ausgangssprache sind.

Um diese Einschränkung zu umgehen, schlugen Cho et al. 2014 eine Encoder-Decoder-Architektur vor, die aus zwei einzelnen RNN besteht: einem Encoder-RNN, das die Inputsequenz codiert, und einem Decoder-RNN, das auf Basis der codierten Information eine Outputsequenz erstellt [46]. Der Encoder-Teil funktioniert dabei grundsätzlich wie die oben beschriebenen Standard-RNN mit der Inputsequenz als Input, der einzige Unterschied ist, dass die Outputs $\hat{y}_t$ irrelevant sind. Von Relevanz ist hingegen der letzte Hidden State h_t, da dieser die Codierung der Inputsequenz darstellt und an den Decoder übergeben wird (in diesem Kontext wird h_t oft auch als Kontextvektor c bezeichnet) [26, S 502]. c entspricht somit h_0 des Decoder-RNN. Als Input für den Decoder dient zunächst ein einzelner <SOS>-Token (Start-of-Sequenz), der dem Netzwerk signalisiert, dass dies der Start der Outputsequenz ist [7, S. 547f]. Aufbauend auf diese Inputs x_0 und c soll der Decoder das erste Element der Outputsequenz erzeugen, was anschließend wiederum als Input für den nächsten Rechenschritt dient. Dies wird fortgesetzt bis der Decoder einen <EOS>-Token (End-of-Sequence) ausgibt, der das Ende der Outputsequenz symbolisiert [7, S. 547f].

Wird bereits zum Trainingszeitpunkt dieser Ansatz mit $x_t = \hat{y}_{t-1}$ gewählt, kann das zu einer langsamen und instabilen Konvergenz führen [87]. Ist beispielsweise bereits der erste Output $\hat{y}_1$ falsch, kann sich dieser Fehler durch die gesamte Outputsequenz ziehen. Um dies zu verhindern, kann eine Technik namens *Teacher Forcing* eingesetzt werden, bei denen die Labels (also die *Ground Truth*) als Inputs verwendet werden: $x_t = y_{t-1}$ [26, S. 503]. Allerdings schafft auch Teacher Forcing neue Probleme, da das damit erstellte Modell möglicherweise zu sehr auf die Trainingssequenzen zugeschnitten ist, wodurch es bei der Inferenz fehleranfälliger sein kann[13] [87]. Eine Weiterentwicklung von Teacher Forcing, das sogenannte *Curriculum Learning*, schafft bei diesem Problem Abhilfe, indem es auf Teacher Forcing basierende Inputs und Modellausgaben-basierte Inputs abwechselt [48]. Das Verhältnis zwischen diesen zwei Input-Arten verändert sich dabei im Laufe des Trainings zugunsten der Modellausgaben-basierte Inputs. Das Modell trainiert somit zunehmend mit Inputs, wie sie auch zur Inferenz vorkommen.

Abbildung 2.13 zeigt den schematischen Aufbau eines Encoder-Decoder-Netzwerkes, das aus einer RNN-Schicht besteht. Eine ähnliche Architektur wird bei der Architekturevaluierung für das Erlernen der Entschlüsselungsfunktion und den neuronalen KPA eingesetzt (dabei werden aber eine GRU- und eine LSTM-Schicht verwendet). Da zu jedem Time-Step ein Buchstabe bzw. <EOS> ausgegeben werden muss, gibt es insgesamt 27 Klassen, wobei diese Klassifikation wie üblich über einen nachgelagerten, zeitverteilten[14] Softmax-Layer durchgeführt wird.

[13] Schlussendlich ist mit Teacher Forcing der Verarbeitungsmodus des Decoders bei der Inferenz anders als im Training. Durch einen möglicherweise hohen kumulativen Fehler, der bei Teacher Forcing nicht auftreten kann, kann das Modell zur Inferenz in einen Zustand kommen, den es nicht vom Training kennt [48].

[14] Es existiert also bei jedem Time-Step ein Softmax-Layer.

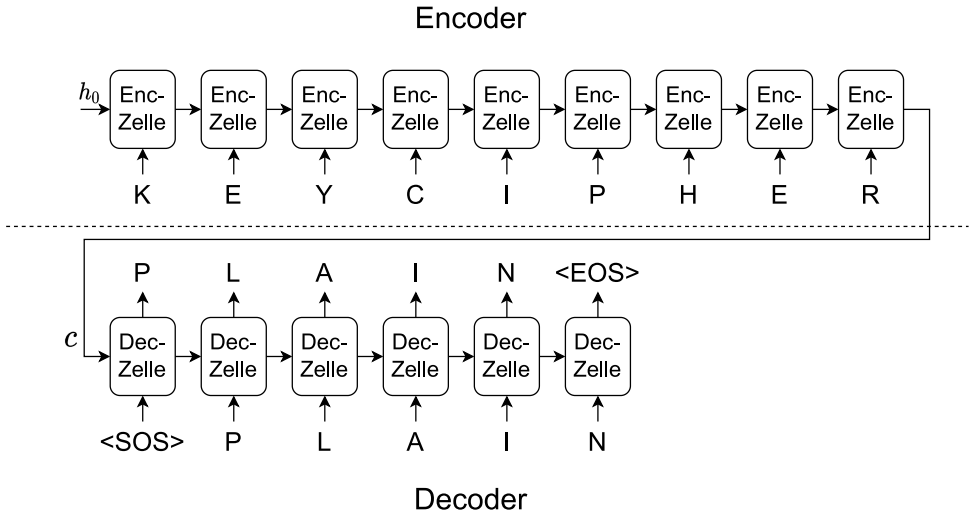

Abbildung 2.13: Verwendung eines einfachen Encoder-Decoders zum Erlernen der Entschlüsselungsfunktion. (Enc-Zelle = Encoder-Zelle, Dec-Zelle = Decoder-Zelle)

2.9 Attention-basierte RNN

Auch wenn Encoder-Decoder-Architekturen die Einsatzmöglichkeiten von RNN stark erweitern, kann es dennoch ein Problem sein, die gesamte Inputsequenz mittels eines einzigen Vektors fixer Länge zu codieren. Cho et al. zeigten in [49] am Beispiel von Neural Machine Translation, dass bei zunehmender Sequenzlänge die Leistung des Encoder-Decoders schnell abnimmt. Aus diesem Grund erweiterten Bahdanau et al. den Encoder-Decoder-Ansatz um ein Attention-Verfahren, das dem Netzwerk ermöglicht, sich für jeden Output auf die wichtigsten Stellen in der Inputsequenz zu fokussieren [50].

In der ursprünglichen Implementierung von Bahdanau et al. wird als Encoder ein BRNN verwendet, das für jeden Time-Step einen konkatenierten Output aus dem Forward und dem Backward Hidden State erstellt [50]. Da RNN zu einem Time-Step t vor allem kürzliche Inputs repräsentieren, wird durch das BRNN erreicht, dass h_t einen Fokus auf Inputs vor und nach dem aktuellen Time-Step legt. Im Gegensatz zu einem einfachen Encoder-Decoder, bei dem die Outputs des Encoders verworfen werden, sind sie für die *Bahdanau-Attention* essenziell, da der Decoder diese in einem *Alignment-Modell*[15] verwendet [50]. Um die Hidden States des Encoders und des Decoders unterscheiden zu können, bezeichnet im Folgenden h_t den Hidden State zum Time-Step t des Encoders und $s_i \in \mathbb{R}^{n_h}$ den Hidden State zum Time-Step $i \in \mathbb{N}$ des Decoders.

Das Alignment-Modell ist nichts anderes als ein einfaches neuronales Netzwerk bei dem zu einem Decoder-Time-Step i unter Zuhilfenahme von s_{i-1} für alle Encoder-Outputs ein eigener Score berechnet wird. Der Score $e_{it} \in \mathbb{R}$ eines Encoder-Outputs zum Time-Step t für den Decode-Time-Step i wird mit folgender Gleichung ermittelt [50]:

$$e_{it} = v^T tanh(W_a s_{i-1} + U_a h_t). \tag{2.27}$$

[15] Manchmal wird das Alignment-Modell auch *Attention-Schicht* genannt [7, S. 554].

$v \in \mathbb{R}^{n_h}$, $W_a \in \mathbb{R}^{n_h \times n_h}$ und $U_a \in \mathbb{R}^{n_h \times 2n_h}$ sind dabei Gewichte, die im Training erlernt werden. Insgesamt handelt es sich bei Gleichung 2.27 um einen einfachen Dense Layer, dessen Ergebnis ausdrückt, wie gut s_{i-1} zum Time-Step t passt. Aus diesen Scores werden über einen anschließenden Softmax-Layer für jeden Encoder-Output Gewichte $\alpha_{it} \in \mathbb{R}$ berechnet [50]:

$$\alpha_{it} = \frac{e^{e_{it}}}{\sum_{j=1}^{T} e^{e_{ij}}}. \tag{2.28}$$

Die Summe dieser Werte α ergibt, wie bei Softmax-Outputs üblich, 1. Jedem Element der Inputsequenz wird somit ein Gewicht zugewiesen, das ausdrückt, wie wichtig es für den Output des aktuellen Time-Steps s_i des Decoders ist. Auf Basis dieser Gewichte wird anschließend ein Kontextvektor $c_i \in \mathbb{R}^{2n_h}$ erstellt [50]:

$$c_i = \sum_{j=1}^{T} \alpha_{ij} h_j. \tag{2.29}$$

Zusammen mit dem vorherigen Hidden State s_{i-1} und dem vorherigen Output $\hat{y}_{i-1}$ wird dann abschließend mit c_i der Output $\hat{y}_i$ für den aktuellen Time-Step berechnet.

Schon kurze Zeit nach der Veröffentlichung des Papers von Bahdanau et al. wurden bereits Verbesserungen des Attention-Mechanismus publiziert, u.a. die sogenannte *Luong-Attention* [51]. Die Luong-Attention erzielt teilweise bessere Ergebnisse als die Bahdanau-Attention und wird deshalb mittlerweile häufiger eingesetzt [7, S. 555f]. Da sich die beiden in vielen Punkten aber ähnlich sind und für diese Arbeit die Bahdanau-Variante verwendet wird, soll an dieser Stelle die Luong-Attention nicht weiter behandelt werden.

2.10 Transformer

Die letzte betrachtete Architektur in diesem Einführungskapitel ist ebenfalls Attention-basiert und ebenfalls eine Seq2Seq-Architektur. Im Gegensatz zu den bisherigen Ansätzen zur Sequenzverarbeitung besitzen Transformer jedoch keine rekurrenten Schichten, sondern verwenden ausschließlich Attention-Mechanismen, weswegen das entsprechende Paper von Vaswani et al. auch den treffenden Namen „Attention Is All You Need“ trägt [52]. Ein großer Nachteil von rekurrenten Schichten ist die inhärente sequenzielle Abarbeitung von Time-Steps, was ein Parallelisieren unmöglich macht. Die Autoren von [52] konnten zeigen, dass Transformer durch ihre rein Attention-basierte Architektur schneller trainiert werden können als RNN und gleichzeitig bessere Ergebnisse bei der maschinellen Übersetzung erzielen. Abbildung 2.14 zeigt die Architektur eines Transformers nach Vaswani et al., die im Folgenden überblicksartig erklärt wird.

Prinzipiell besteht auch ein Transformer aus einem Encoder (Stack auf der linken Seite) und einem Decoder (rechte Seite). Beide bestehen dabei aus $N\times$ den in Abbildung 2.14 gezeigten Blöcken, wobei im Paper $N = 6$ verwendet wird [52]. Der finale Output des Encoders wird dabei jedem der N Decoder-Blöcke zur Verfügung gestellt. Die Inputs des Encoders und des Decoders sind ähnlich der eines RNN, wobei beim

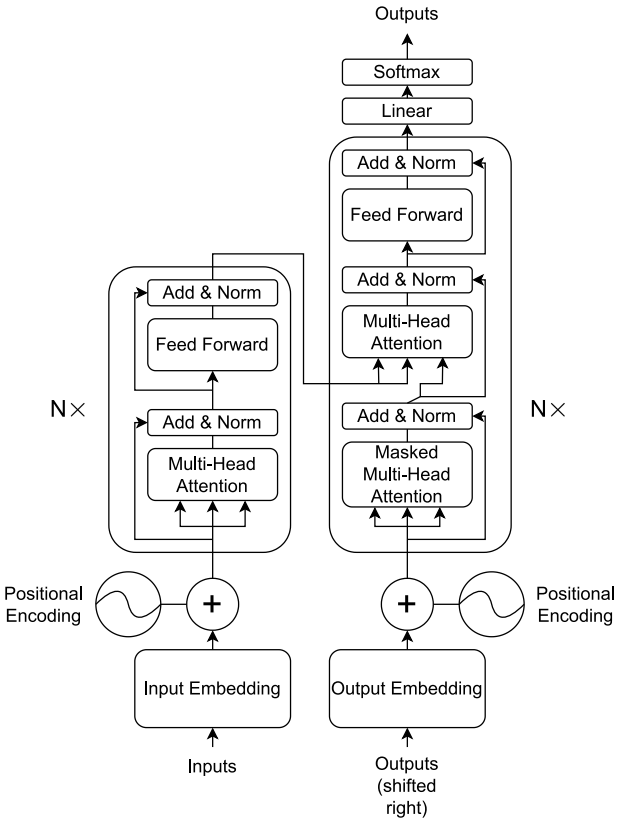

Abbildung 2.14: Darstellung der Architektur eines Transformer nach Vaswani et al.. (Bildquelle: [52])

Training der Decoder-Input wieder der um ein Element verschobene Output ist[16]. Bei der Inferenz muss der Decoder mehrmals ausgeführt werden, da bei jedem Durchgang ein Element ausgegeben wird, das zusammen mit der bisherigen Sequenz als Input für den nächsten Decoder-Durchlauf dient [7, S. 559]. Die beiden Embedding-Layers werden dazu verwendet, Input-Token in Vektoren zu überführen[17], die anschließend von den folgenden Schichten verarbeitet werden können. Da es bei Transformer jedoch keine sequenzielle Verarbeitung wie bei RNN (und auch keine Convolutional-Layer) gibt, müssen Tokenpositionen innerhalb einer Sequenz auf anderem Weg bekannt gegeben werden. In [52] werden dazu *Positional Encodings* mittels Sinus- und Cosinusfunktionen erzeugt:

$$PE_{(pos,2i)} = sin(\frac{pos}{10\,000^{\frac{2i}{d_{model}}}}) \tag{2.30}$$

[16] Auch hier wird zu Beginn ein <SOS>-Token angehängt [7, S. 559]

[17] Textsequenzen und auch einzelne Elemente wie Wörter oder Buchstaben können nicht direkt von neuronalen Netzwerken verarbeitet werden. Um dies zu ermöglichen, müssen sie zuerst in eine numerische Form überführt werden. Bei einem Embedding wird ein Input-Token (also ein einzelner Teil der Inputsequenz) in einen Dense-Vektor umgewandelt [7, S. 437f]. Vaswani et al. verwenden dabei einen Vektor der Dimension $d_{model} = 512$ [52].

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10\,000^{\frac{2i}{d_{model}}}}). \tag{2.31}$$

$pos \in \mathbb{N}$ entspricht der Tokenposition innerhalb der Sequenz, $i \in \mathbb{N}_0$ beschreibt die Komponente des entstehenden eindeutigen Encoding-Vektors, der ebenfalls die Dimension $d_{model} \in \mathbb{N}$ besitzt und somit zu dem Embedding-Vektor addiert werden kann. Vaswani et al. führen in [52] an, dass die Positional Encodings auch erlernt werden könnten, jedoch nehmen sie an, dass die fixen Encodings zur Inferenzzeit besser mit Sequenzen, die länger als die Trainingsdaten sind, funktionieren als die erlernten.

Das zentrale Konzept von Transformer, das in den *Multi-Head-Attention-Layern* realisiert wird, ist die sogenannte *Self-Attention*. Self-Attention beschreibt, dass das Modell die Beziehung jedes einzelnen Elements mit den anderen Elementen der Sequenz codiert - die Inputsequenz achtet also auf sich selbst [7, S. 559]. Diese Self-Attention wird über die *Scaled Dot-Product Attention* realisiert, welche folgendermaßen definiert ist [52]:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.32}$$

Q (Queries), K (Keys) und V (Values) sind drei Matrizen, die durch drei unabhängige Dense Layer aus dem Input des Multi-Head-Attention-Layers erzeugt werden. Einfach beschrieben wird bei der Scaled Dot-Product Attention eine Art Dictionary-Lookup durchgeführt, bei der auf eine Frage (= Query) auf Basis eines passenden Keys ein Wert (Value) zurückgegeben wird [7, S. 562]. Um den passenden Key für ein Query zu finden, wird ein Skalarprodukt als Ähnlichkeitsmaß verwendet (wie bei der Luong-Attention [51]). Um eine Sättigung der Softmax-Funktion und die damit einhergehenden kleinen Gradienten zu vermeiden, wird das Skalarprodukt mit der Wurzel von $d_k \in \mathbb{N}$, der Dimensionalität der Keys und Queries, skaliert [52]. Über die Softmax-Funktion werden wieder Gewichte für die anschließende gewichtete Summe der Values erzeugt. Da in der Praxis mehrere solcher Vorgänge parallel ausgeführt werden, werden bei Gleichung 2.32 Matrizen statt Vektoren eingesetzt [52]. Ein einfaches Beispiel zur Veranschaulichung: Auf den Satz „Die Katze sitzt auf dem Tisch“ wird die Scaled Dot-Product Attention angewandt. Bei der Codierung des Wortes „sitzt“ kann ein Query (im übertragenen Sinne) „Wo sitzt die Katze?“ sein. Auf diese Frage wird der Key von „Tisch“ und eventuell noch von „auf“ einen großen Ähnlichkeitswert mit dem Query ergeben, weshalb auf diese beiden Wörter große Aufmerksamkeit gelegt wird. Da ein Multi-Head-Attention-Layer aus mehreren, parallelen Scaled Dot-Product Attentions besteht (in [52] werden 8 verwendet), können mehrere solcher Fragen zu einem Wort gestellt werden. Die einzelnen Outputs zu einem Wort werden abschließend konkateniert und über einen weiteren Dense Layer zurück in den ursprünglichen Raum projiziert (die einzelnen Heads verwenden zur schnelleren Berechnung Vektoren kleinerer Dimensionalität) [52].

Im Decoder finden sich zwei verschiedene Multi-Head-Attention-Layer: Der zweite entspricht dem Layer des Encoders (nur dass zusätzlich zum Decoder-Input auch der Encoder-Output miteinbezogen wird), bei dem ersten handelt es sich um eine geringfügig angepasste Variante. Bei der *Masked Multi Head-Attention* wird ein Teil des Decoder-Inputs so maskiert, dass bei jedem bearbeiteten Element nur auf vorherige Elemente in Sequenz geachtet werden kann [52]. Dies wird gemacht, da das Modell bei der Inferenz

die nachfolgenden Elemente noch nicht kennt und so im Training vermieden werden kann, dass das Modell auf Basis von Elementen Vorhersagen trifft, die eigentlich zu diesem Zeitpunkt noch unbekannt sein sollten.

Kapitel 3

Einführung Kryptologie

Nachdem das vorherige Kapitel die wichtigsten Konzepte des Deep Learnings zusammengefasst hat, gibt dieses Kapitel eine Einführung in den zweiten wichtigen Teilbereich dieser Arbeit – die klassische Kryptologie. Wie schon in Kapitel 1 angeführt, fasst der Begriff Kryptologie die Kryptoanalyse und die Kryptografie zusammen. Das US-amerikanische National Institute of Standards and Technology (NIST) definiert Kryptoanalyse in der Special Publication (SP) 800-57 folgendermaßen [53, S. 8]:

> „*The study of mathematical techniques for attempting to defeat cryptographic techniques and information-system security. This includes the process of looking for errors or weaknesses in the implementation of an algorithm or in the algorithm itself.*“

Bei klassischer Kryptoanalyse ist der alleinige Fokus auf mathematische Techniken jedoch in manchen Fällen zu eng. Ein Beispiel dafür ist ein Angriff auf die Caesar-Chiffre, bei der eine einfache Brute-Force-Suche über alle 25 möglichen Schlüssel (bei einem lateinischen Alphabet) auch manuell schnell zu einer erfolgreichen Entschlüsselung führt. Eine Definition des der Kryptoanalyse entgegengesetzten Begriffs der Kryptografie findet sich hingegen in SP 800-59 [54, S. 14]:

> „*The discipline that embodies the principles, means, and methods for the transformation of data in order to hide their semantic content, prevent their unauthorized use, or prevent their undetected modification.*“

Die Kryptografie und die Kryptoanalyse verfolgen somit gegensätzliche Ziele: Während die Kryptografie auf die Sicherstellung der Schutzziele Vertraulichkeit, Integrität und Authentizität[1] abzielt, beabsichtigt die Kryptoanalyse ebendiese zu verletzen. Es ist jedoch anzumerken, dass die im folgenden vorgestellten klassischen Verfahren ausschließlich die Vertraulichkeit schützen. Der Schutz der Authentizität („*[...] prevent their unauthorized use [...]*“) und der Integrität („*[...] prevent their undetected modification.*“) sind somit keine Schutzziele, die durch die genannten klassischen Verfahren sichergestellt werden.

[1] Ergänzend zu der Definition in SP 800-59 kann auch die Nicht-Abstreitbarkeit z.B. durch digitale Signaturen genannt werden.

3.1 Klassische und moderne Kryptologie

In der Literatur finden sich kaum genaue bzw. eindeutige Definitionen der Begriffe „klassische“ und „moderne Kryptologie“. Eine verwendbare Unterscheidung liefert jedoch Nils Kopal vom CrypTool-Projekt in seiner Video-Reihe „Basics of Cryptology“ [76]. So werden klassische Verfahren meist manuell oder mit Hilfe von einfachen Maschinen durchgeführt, während moderne Kryptologie erst durch die Verwendung von Computern ermöglicht wird [76]. Ein weiteres bei den folgenden Verfahren sichtbares Unterscheidungsmerkmal ist, dass bei klassischen Verfahren grundsätzlich einzelne Zeichen (z.B. Buchstaben) als kleinste Einheit dienen, während moderne Verfahren wie RSA oder AES auf Bitebene arbeiten. Klassische Kryptoverfahren können in die zwei Hauptgruppen Substitutions- und Transpositionschiffren unterteilt werden. Wie bei ADFG(V)X in Kapitel 3.3.2 ersichtlich wird, gibt es aber auch Kombinationschiffren, die Merkmale sowohl von Substitutions- als auch von Transpositionschiffren aufweisen. Die im Folgenden beschriebenen Chiffren wurden auf Anraten von Nils Kopal und Vasily Mikhalev vom CrypTool-Projekt aufgrund ihrer zunehmenden Komplexität ausgewählt. Die Reihenfolge der Chiffren in den folgenden Kapiteln bestimmt ihre Stärke. Gestartet wird in jedem Kapitel mit der schwächsten Chiffre.

3.2 Klassische Substitutionschiffren

Substitutionschiffren erhalten ihren Namen von durchgeführten Substitutionen, bei denen im Zuge der Verschlüsselung Zeichen des Plaintexts durch Zeichen eines Ciphertextalphabets bzw. mehrerer Ciphertextalphabete ersetzt werden. Es wird zwischen der einfachen (monoalphabetischen) und der polyalphabetischen Substitution unterschieden.

Bei der monoalphabetischen Substitution werden Zeichen des Plaintexts Zeichen des Ciphertextalphabets fix zugeordnet. Diese Zuordnung bleibt bei der gesamten Verschlüsselung unverändert [55, S. 32]. Es handelt sich im einfachsten Fall wie beispielsweise bei der Caesar-Chiffre um eine bijektive Abbildung eines Plaintextalphabets $\mathcal{P}$ auf ein Ciphertextalphabet $\mathcal{C}$, wobei die Abbildung f den verwendeten Schlüssel darstellt [56, S. 17]:

$$f : \mathcal{P} \mapsto \mathcal{C}. \tag{3.1}$$

Ein Problem bei solchen bijektiven Abbildung ist, dass ein so entstandener Ciphertext über eine einfache Häufigkeitsanalyse, die in Kapitel 3.2.1 genauer vorgestellt wird, entschlüsselt werden kann. Bei dieser Art von Angriff wird die Häufigkeit der einzelnen Zeichen im Ciphertext bestimmt und mit der Häufigkeitsverteilung von Zeichen einer angenommen Ausgangssprache verglichen. Durch den Vergleich der Häufigkeiten der einzelnen Zeichen kann auf die Abbildung f geschlossen werden. Um dies zu erschweren, wurden ausgehend von der bijektiven monoalphabetischen Substitution komplexere Verfahren entwickelt.

Ein Beispiel einer solchen Weiterentwicklung ist die homophone Verschlüsselung, die als Sonderform der monoalphabetischen Substitution gilt [55, S. 36]. Bei diesem Verfahren werden einem Plaintextzeichen mehrere Ciphertextzeichen zugeordnet, wodurch

die Abbildung nicht mehr bijektiv ist. Die Anzahl der für ein bestimmtes Plaintextzeichen vorgesehenen Ciphertextzeichen wird üblicherweise von der Häufigkeit des zu verschlüsselnden Zeichens bestimmt [55, S. 36].

Ein anderer Ansatz sind polygraphische Substitutionsverfahren, bei denen nicht individuelle Zeichen, sondern ganze Buchstabengruppen verschlüsselt werden [55, S. 36]. Diese Zeichenkombinationen können Digramme, Trigramme etc. sein. Ein Vertreter dieser Gruppe ist die in Kapitel 3.2.3 beschriebene Playfair-Chiffre, bei der standardmäßig Digramme verschlüsselt werden. Der Vorteil von polygraphischen Verfahren ist, dass Häufigkeitsverteilungen auf monographischer Basis nicht mehr verwendbar sind, da die Verschlüsselung eines Zeichens in Abhängigkeit eines Nachbarzeichens/mehrerer Nachbarzeichen durchgeführt wird. Ein Beispiel: Bei einer digraphischen Verschlüsselung (also einer Verschlüsselung auf Basis von Digrammen) gibt es bei der Verwendung der Großbuchstaben des lateinischen Alphabets statt 26 Einzelbuchstaben $26 \cdot 26 = 676$ Digramme. Somit ist eine Häufigkeitsanalyse zwar nicht unmöglich, es muss jedoch statt der Verteilung von einzelnen Buchstaben eine Verteilung über alle möglichen Digramme verwendet werden, was durch die höhere Anzahl der Möglichkeiten aufwendiger ist.

Eine der wichtigsten Verbesserungen in der klassischen Kryptologie ist die Entwicklung von polyalphabetischen Verschlüsselungen im Zeitalter der Renaissance [1, S. 18]. Im Gegensatz zur monoalphabetischen Substitution gibt es bei der polyalphabetischen Variante mehrere Ciphertextalphabete, die bei der Verschlüsselung abgewechselt werden. Die Zuordnung zwischen Plaintextzeichen und Ciphertextzeichen ist deshalb nicht fix, sondern variabel. Welches der Ciphertextalphabete konkret für ein bestimmtes Plaintextzeichen verwendet wird, wird üblicherweise von dem Schlüssel bestimmt [55, S. 38]. Durch die verschiedenen Substitutionen wird die Häufigkeitsverteilung von Zeichen verschleiert, womit eine Häufigkeitsanalyse je nach Chiffre zwar nicht unmöglich, jedoch sehr aufwendig wird [88]. Eine der bekanntesten polyalphabetischen Verschlüsselungen ist die Vigenère-Chiffre (Kapitel 3.2.2), aber auch bei der viel später erfundenen und viel komplexeren Enigma (Kapitel 3.2.5) handelt es sich um eine Verschlüsselung auf Basis polyalphabetischer Substitution.

3.2.1 Caesar-Chiffre und monoalphabetische Substitution (MASC)

Die Caesar-Chiffre ist die einfachste in dieser Arbeit betrachtete Verschlüsselung. Es handelt sich dabei um eine Verschiebe-Chiffre, bei der das Ciphertextalphabet durch eine bestimmte Verschiebung des Plaintextalphabets entsteht. Laut den Überlieferungen von Sueton verwendete Caesar eine Verschiebung von 3, weshalb beispielsweise ein „A“ zu einem „D“ wurde [1, S. 11]. Allgemein wird der Begriff heute aber für Verschiebungen beliebiger Distanz verwendet. Formal handelt es sich bei der Caesar-Chiffre um eine monographische monopartite monoalphabetische Substitution. Dies bedeutet, dass jedem einzelnen Plaintextzeichen (monographisch) genau ein Ciphertextzeichen (monopartit) zugeordnet wird.

Zur Veranschaulichung wird eine Caesar-Chiffre mit einer Verschiebung von 10 betrachtet (oben das Plaintextalphabet, unten das Ciphertextalphabet):

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
K L M N O P Q R S T U V W X Y Z A B C D E F G H I J

Der Plaintext *HAGENBERG* ergibt verschlüsselt somit *RKQOXLOBQ*. Die Caesar-Chiffre kann auch mathematisch dargestellt werden, indem die einzelnen Buchstaben aufsteigend durchnummeriert werden, angefangen bei $A = 0$ hin zu $Z = 25$. Ein Plaintextzeichen $x \in \mathbb{N}_0$ wird somit bei einer Verschiebung von $s \in \mathbb{N}$ und einer Alphabetlänge von $n \in \mathbb{N}$ folgendermaßen verschlüsselt:

$$Encrypt(x) = x + s \ mod \ n \tag{3.2}$$

Aufgrund der sehr eingeschränkten Anzahl von möglichen Schlüsseln, die $n - 1$ entspricht, bietet die Caesar-Chiffre selbst aus der Perspektive von klassischen Chiffren einen sehr schwachen Schutz. Um erfolgreich einen Angriff auf diese Chiffre durchführen zu können, müssen lediglich alle verfügbaren Schlüssel probiert werden, was selbst manuell ohne großen Zeitaufwand machbar ist.

Mit dem Begriff MASC, kurz für *Monoalphabetische Substitutionschiffre*, wird in dieser Arbeit eine einfache allgemeine monoalphabetische Substitution bezeichnet. Diese ebenfalls monographische und monopartite Verschlüsselung ist nicht wie die Caesar-Chiffre auf eine Verschiebungsoperation bei der Erstellung des Ciphertextalphabets beschränkt. Eine Möglichkeit, das Ciphertextalphabet zu definieren, ist beispielsweise mit einem Schlüsselwort zu arbeiten. Dabei werden an die einzigartigen Buchstaben des Worts die fehlenden Zeichen des Alphabets angehängt. Das Schlüsselwort *SECRET* führt im einfachsten Fall zu folgender Abbildung, wodurch der Plaintext *HAGENBERG* in *DSBTKETOB* umgewandelt wird (oben wieder das Plaintextalphabet, unten das Ciphertextalphabet):

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
S E C R T A B D F G H I J K L M N O P Q U V W X Y Z

Es ist ersichtlich, dass die letzten sechs Zeichen des Alphabets nicht verschlüsselt werden, weshalb bei diesem Schlüsselwort ein anderer Konstruktionsweg verwendet werden sollte (z.B. die fehlenden Zeichen werden statt in alphabetischer Reihenfolge rückwärts an das Schlüsselwort angehängt). Sender und Empfänger der verschlüsselten Nachricht müssen somit das Schlüsselwort und den Konstruktionsmechanismus des Ciphertextalphabets kennen. Das Ciphertextalphabet kann auch zufällig generiert werden, allerdings muss der Empfänger somit das ganze Alphabet kennen, wodurch es effektiv zum Schlüssel wird. MASC bietet eine im Vergleich zu der Caesar-Chiffre, die als Sonderfall von MASC betrachtet werden kann, höhere Sicherheit. Bei der Verwendung des lateinischen Alphabets mit 26 Buchstaben sind 26! verschiedene Schlüssel möglich (allgemein $n!$). Dadurch ist ein Brute-Force-Angriff nicht mehr durchführbar. Dennoch kann die einfache monoalphabetische Substitution sehr leicht entschlüsselt werden, da sie aufgrund der fixen Zuordnung von Plaintext- und Ciphertextzeichen anfällig für Häufigkeitsanalysen ist.

Die Häufigkeitsanalyse ist eine statistische Methode der Kryptoanalyse, mit der die Buchstabenhäufigkeit einer Sprache mit der Häufigkeit der Zeichen in einem Ciphertext verglichen wird. Abbildung 3.1 zeigt eine solche Häufigkeitsverteilung für die deutsche Sprache. In dem Diagramm wird klar ersichtlich, dass einige wenige Buchstaben die Mehrheit aller in einem Text vorkommenden Buchstaben darstellen. Vor allem „E“ und „N“ sind im Deutschen sehr häufig anzutreffen. Bei einer einfachen monoalphabetischen Substitution wird zwar ein Buchstabe durch einen anderen ersetzt, die

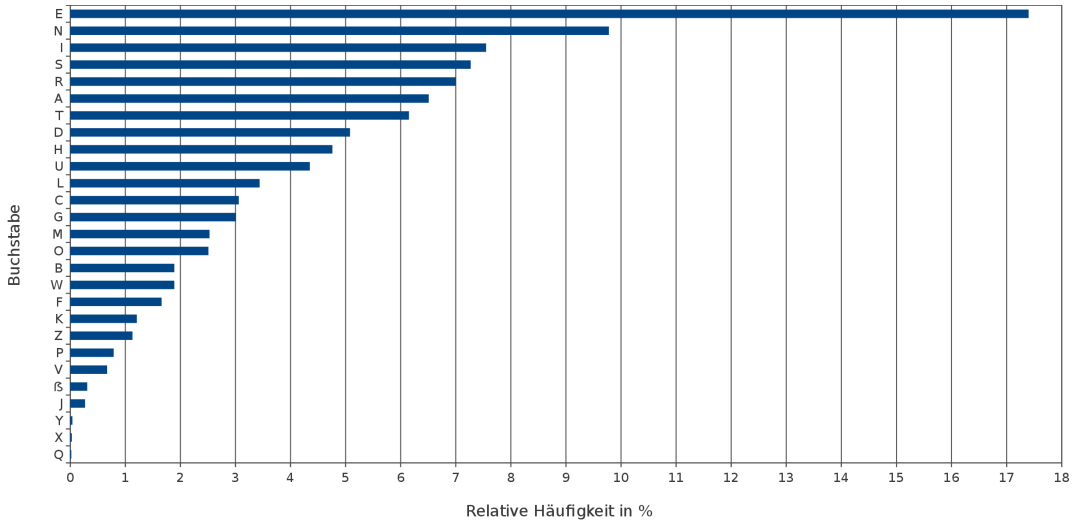

Abbildung 3.1: Buchstabenhäufigkeit der deutschen Sprache. (Bildquelle: [89])

Häufigkeitsverteilung selbst bleibt aber gleich. Im oben genannten Ciphertextalphabet, das auf dem Schlüsselwort *SECRET* basiert, wird „E“ mit „T“ verschlüsselt. „T“ wird nun statistisch mit einer relativen Häufigkeit von ungefähr 17% der häufigste Buchstabe in einem verschlüsselten deutschen Text sein. Bei der Entschlüsselung kann somit angenommen werden, dass „T“ „E“ entspricht, womit Schritt für Schritt das Ciphertextalphabet rekonstruiert werden kann. Je länger der vorliegende Ciphertext ist, desto besser funktioniert dieser Angriff, da sich die Buchstabenhäufigkeiten des Ciphertexts immer mehr an die tatsächlichen Häufigkeiten der verwendeten Sprache annähern. Bei großen Abweichungen der Häufigkeiten können weitere Verteilungen, wie etwa die Di- oder die Trigrammverteilung, ergänzend verwendet werden [56, S. 19]. Auch einfaches Raten kann weiterhelfen. Ein solcher Angriff, bei dem nur der Ciphertext für eine Entschlüsselung benötigt wird, wird als *Ciphertext-Only Angriff* bezeichnet [56, S. 2].

3.2.2 Vigenère und Autokey-Vigenère

Die Vigenère-Chiffre, benannt nach dem französischem Gelehrten Blaise de Vigenère, ist im Gegensatz zu der Caesar-Chiffre und der MASC eine polyalphabetische Chiffre. Entgegen ihrem Name wurde sie nicht von Vigenère selbst erfunden, sondern mit kleinen Abweichungen bereits 1553 von Giovan Battista Bellasso in seinem Werk *La cifra del Sig. Giovan Battista Bellaso* beschrieben [1, S. 22f.]. Bei Bellasos Variante der Chiffre wird für zwei Buchstaben je ein Ciphertextalphabet verwendet (also ein Alphabet für „AB“, eines für „CD“ etc.) [1, S. 22f.]. Insgesamt stehen somit 11 Ciphertextalphabete zur Verfügung (die Buchstaben J, K, U und W wurden von Bellasso nicht verwendet). Welche Tabelle zur Verschlüsselung verwendet wird, wird durch ein Schlüsselwort bestimmt. Bei dem Schlüssel *SECRET* wird beispielsweise der erste Buchstabe mit dem dem Buchstaben S zugewiesenen Alphabet verschlüsselt, der zweite mit dem Alphabet für E und so weiter. Beim Ende des Schlüsselworts wird solange wieder von vorne begonnen, bis der gesamte Plaintext verschlüsselt ist.

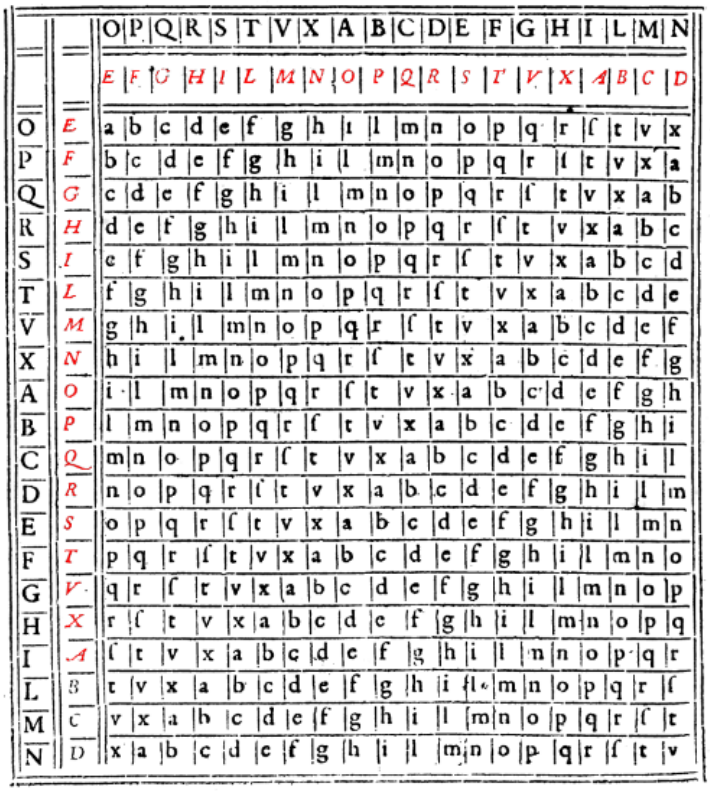

Abbildung 3.2: Tabula Recta in Vigenères *Traicté des chiffres* [58]. (Bildquelle: [57])

Die heute bekannte Variante der Vigenère-Chiffre entstand 1586, als Blaise de Vigenère die Ideen von Bellaso weiterentwickelte [57]. Dabei verwendete er in Anlehnung an eine Idee von Johannes Trithemius aus 1508 eine sogenannte *Tabula Recta* (siehe Abbildung 3.2) [57]. Die verwendete Zeile der Tabula Recta wird dabei von von einem Schlüsselwort bestimmt. Ein Buchstabe eines Wortes wird somit verschlüsselt, indem der entsprechende Buchstabe des Schlüssels die Zeile und der zu verschlüsselnde Buchstabe selbst die Spalte bestimmt. Bei der Vigenère-Chiffre handelt es sich somit um eine monographische monopartite polyalphabetische Verschlüsselung. Bei Verwendung des heutigen Alphabets funktioniert die Tabula Recta genauso, nur ist die Tabula Recta üblicherweise alphabetisch beginnend bei „A“ angeordnet. Anstatt mit einer Ersetzungstabelle kann die Verschlüsselung auch mathematisch erfolgen. Dazu wird wieder jeder Buchstabe beginnend bei $A = 0$ durchnummeriert. Gleichung 3.3 definiert die Vigenère-Verschlüsselung für ein Plaintextzeichen $x \in \mathbb{N}_0$ und ein entsprechendes Schlüsselzeichen $k \in \mathbb{N}_0$ bei einem $n \in \mathbb{N}$ Zeichen langem Alphabet.

$$Encrypt(x) = x + k \ mod \ n \tag{3.3}$$

Der Plaintext *HAGENBERG* wird verschlüsselt mit dem Schlüssel *SECRET* somit zu *ZEIVRUWVI*:

S E C R E T S E C
H A G E N B E R G
Z E I V R U W V I

Es wird ersichtlich, dass die Vigenère-Chiffre schlussendlich aus mehreren Caesar-Chiffren besteht. Jeder einzigartige Buchstabe im Schlüssel entspricht einer zusätzlichen Caesar-Chiffre, wodurch ein Schlüssel mit nur einem Zeichen äquivalent zu einer einfachen Caesar-Chiffre ist. Durch diesen polyalphabetischen Ansatz wird eine Häufigkeitsanalyse erschwert. Eine Schwäche ist jedoch die Wiederholung des Schlüssels. Gelingt eine Bestimmung der Schlüssellänge $l \in \mathbb{N}$, so kann anschließend eine Häufigkeitsanalyse für jede Stelle des Schlüssels durchgeführt werden. Dies resultiert in l Häufigkeitsverteilungen, die wiederum zu der Rekonstruktion der einzelnen Verschiebungen verwendet werden [88]. Da für die Bestimmung der Schlüssellänge nach Wiederholungen im Ciphertext gesucht wird, ist die Schlüssellänge entscheidend für die Sicherheit des Verfahrens. Ein langer Schlüssel führt zu weniger Wiederholungen und ist somit sicherer als ein kürzerer [57]. Diese Methode, wurde erst im 19. Jahrhundert zuerst von Charles Babbage und anschließend von Friedrich Wilhelm Kasiski (deshalb auch der Name *Kasiski-Test*) beschrieben [1, S. 33]. Die Vigenère-Chiffre galt somit für ca. 300 Jahre als sicher.

Laut Schmeh hat Vigenère selbst eine Verbesserung der herkömmlichen Vigenère-Chiffre in [58][2] vorgeschlagen [1, S. 23]. In der heute Autokey-Vigenère genannten Variante wird das Schlüsselwort bei Plaintexten, die länger als der Schlüssel sind, nicht wiederholt, sondern es werden stattdessen die Zeichen des Plaintexts an den Schlüssel angehängt:

S E C R E T H A G
H A G E N B E R G
Z E I V R U L R M

Dies hat zur Folge, dass Angriffe wie der Kasiski-Test nicht mehr durchführbar sind, jedoch werden andere Schwächen in die Chiffre eingeführt [59, S. 288]. So ist beispielsweise der verwendete Schlüssel besonders unsicher, wenn Teile des dafür verwendeten Plaintexts bekannt sind [90].

3.2.3 Playfair

Bei der Playfair-Chiffre handelt es sich im Gegensatz zu den bisher vorgestellten Chiffren um eine bigraphische bipartite monoalphabetische Verschlüsselung. Es werden somit gleichzeitig zwei Plaintextzeichen (bigraphisch) in zwei Ciphertextzeichen (bipartit) verschlüsselt. Obwohl die Chiffre bereits 1854 erfunden wurde, wurde sie von den Briten noch im ersten Weltkrieg eingesetzt [56, S. 32]. Trotz ihres Namens wurde sie von Charles Wheatstone erdacht, sein Freund Lord Lyon Playfair machte sie jedoch in militärischen und diplomatischen Kreisen bekannt, weshalb die Chiffre schlussendlich seinen Namen bekam [77].

Zur Verschlüsselung wird eine 5×5-Matrix mit dem Alphabet befüllt (mit Ausnahme von „J“). Diese Befüllung kann komplett zufällig erfolgen oder es wird ein Schlüsselwort verwendet, wodurch der Schlüssel leichter merkbar ist. In letzterem Fall werden

[2] Die Originalquelle [58] lag bei der Erstellung dieser Arbeit nicht vor.

S	E	C	R	T
A	B	D	F	G
H	I	K	L	M
N	O	P	Q	U
V	W	X	Y	Z

Abbildung 3.3: 5×5-Matrix für eine Playfair-Verschlüsselung mit dem Schlüsselwort *SECRET* und einem Beispieldigramm.

die einzigartigen Zeichen des Schlüsselworts in die Matrix geschrieben und diese wird mit dem restlichen Alphabet aufgefüllt. Anschließend wird der Plaintext in Digramme unterteilt. Diese Digramme werden nach folgenden Regeln mithilfe der erstellten Matrix verschlüsselt [55, S. 37]:

1. Sind beide Plaintextbuchstaben in derselben Spalte, so werden die Buchstaben direkt unter den Plaintextbuchstaben für die Substitution verwendet.
2. Befinden sich beide Plaintextbuchstaben in derselben Zeile, werden die Buchstaben rechts davon verwendet.
3. Bei zwei Buchstaben, die nicht in derselben Spalte oder Zeile sind, wird ein Buchstaben mit dem substituiert, der dieselbe Zeilennummer wie dieser hat, aber in der Spalte des zweiten Buchstaben ist.
4. Besteht ein Digramm aus zwei gleichen Buchstaben, wird ein Füller wie beispielsweise ein X eingefügt.

Als Beispiel wird wieder der Plaintext *HAGENBERG* mit dem Schlüsselwort *SECRET* betrachtet. Zunächst wird der Plaintext in Digramme unterteilt: HA GE NB ER GX (auch hier wird ein Füller verwendet). Anschließend wird eine Matrix mit dem Schlüssel erstellt und die Digramme werden nach den oben genannten Regeln verschlüsselt. Die Verschlüsselung des letzten Digramms GX in DZ wird in Abbildung 3.3 dargestellt. Die orangenen Buchstaben sind die Plaintextbuchstaben, die hellblauen die Ciphertextbuchstaben. Insgesamt ergibt sich der Ciphertext *NHBTOACTDZ*.

3.2.4 Hill

Die Hill-Chiffre wurde 1929 von Lester Hill vorgestellt [60]. Die polygraphische polypartite polyalphabetische Verschlüsselung basiert auf linearer Algebra und bildet in einem Durchgang $d \in \mathbb{N}$ Plaintextbuchstaben auf d Ciphertextbuchstaben ab [56, S. 33]. Zunächst muss vorbereitend das gesamte Alphabet in Zahlen codiert werden, wobei alle Zahlen modulo n gerechnet werden (n entspricht wie bereits oben der Alphabetlänge). Im einfachsten Fall wird das Alphabet beginnend bei $A = 0$ bis zu $Z = 25$ durchnummeriert. Als Schlüssel wird eine $d \times d$-Matrix K benötigt, zu der eine inverse Matrix K^{-1} vorhanden ist. Zur Verschlüsselung werden nun d Zeichen des Plaintext in einen d-dimensionalen Vektor $\vec{m}$ überführt, der anschließend mit K multipliziert wird (Gleichung 3.4). Die Entschlüsselung erfolgt analog, indem der Ciphertext mit der Inversen K^{-1} multipliziert wird.

$$Encrypt(\vec{m}) = K \cdot \vec{m} \ mod \ n \tag{3.4}$$

Am Beispiel des Plaintexts *HAGENBERG* wird die Hill-Chiffre mit einem Wert $d = 2$ demonstriert. Als Schlüssel wird folgende Matrix K verwendet:

$$K = \begin{pmatrix} 3 & 2 \\ 3 & 5 \end{pmatrix}. \tag{3.5}$$

Als Inverse K^{-1} ergibt sich modulo 26 somit:

$$K^{-1} = \begin{pmatrix} 15 & 20 \\ 17 & 9 \end{pmatrix} \tag{3.6}$$

Zu dem Plaintext muss zunächst ein Padding angehängt werden, da die Textlänge kein Vielfaches von d ist. Für dieses Beispiel wird deshalb ein „A“ angehängt. In Zahlen codiert ergibt der Plaintext somit $(7, 0, 6, 4, 13, 1, 4, 17, 6, 0)$. Aus dieser Sequenz wird für die ersten 2 Zeichen der Vektor $\vec{m}$ gebildet:

$$\vec{m} = \begin{pmatrix} 7 \\ 0 \end{pmatrix}. \tag{3.7}$$

Die Verschlüsselung von „HA“ erfolgt anschließend nach Gleichung 3.8:

$$Encrypt(\vec{m}) = \begin{pmatrix} 3 & 2 \\ 3 & 5 \end{pmatrix} \cdot \begin{pmatrix} 7 \\ 0 \end{pmatrix} = \begin{pmatrix} 21 \\ 21 \end{pmatrix}. \tag{3.8}$$

Das Ergebnis der Verschlüsselungsoperation $\begin{pmatrix} 21 & 21 \end{pmatrix}^T$ kann wieder in eine alphabetische Form konvertiert werden, wodurch „HA“ zu „VV“ verschlüsselt wurde. Der gesamte Ciphertext lautet *VVAMPSUTSS*.

3.2.5 Enigma

Bei der wohl bekanntesten Verschlüsselungsmaschine, der Enigma, handelt es sich um eine monographische monopartite polyalphabetische Substitution. Obwohl im Zweiten Weltkrieg mehrere sogenannte Rotor-Verschlüsselungsmaschinen eingesetzt wurden, erreichte die Enigma aufgrund ihrer erfolgreichen Entschlüsselung in Bletchley Park (UK) einen besonderen Bekanntheitsgrad, der unter anderem zu zwei Filme – Enigma (2001) und The Imitation Game (2014) – führte. Im Gegensatz zu den bisherigen Chiffren wird die Ver- bzw. Entschlüsselung bei der Enigma nicht manuell durchgeführt, sondern mit einer speziellen Maschine.

Die Enigma wurde von dem Deutschen Arthur Scherbius erfunden und 1918 zur Patentierung eingereicht [1, S. 282]. In Bezug auf Größe und Form ähnelt sie einer Schreibmaschine. Der Begriff Rotor-Verschlüsselungsmaschine leitet sich von den zur Erstellung der Ciphertextalphabete verwendeten Rotoren ab, von denen bei der gebräuchlichsten Variante der Enigma drei Stück eingesetzt werden. Eine spätere Version, die von der deutschen Kriegsmarine eingesetzte Enigma M4, verwendet vier Rotoren, wodurch die Sicherheit nochmals erhöht wird [1, S. 290]. Jeder Rotor besitzt 26 Signalein- und Ausgänge, die jeweils unterschiedlich miteinander verbunden sind. Ein vierter bzw. fünfter fixer Rotor wird als Reflektor bezeichnet [61]. Im Gegensatz zu den übrigen Rotoren, die Kontaktstellen auf beiden Seiten haben und somit das Signal weiterleiten können,

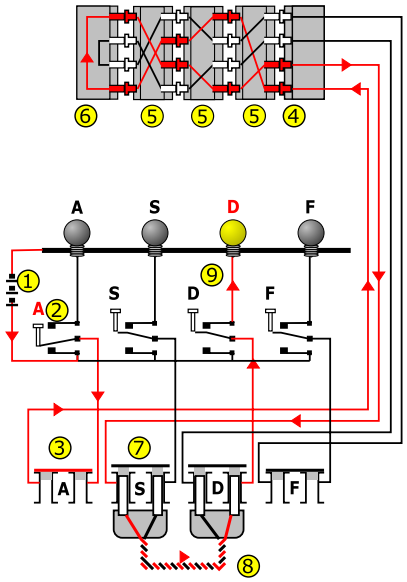

Abbildung 3.4: Vereinfachte Darstellung des Signalflusses der Enigma. (Bildquelle: [91])

hat der Reflektor nur auf einer Seite Kontakte, die nach einem bestimmten Muster miteinander verbunden sind [61]. Somit wird über den Reflektor ein Buchstabe auf einen anderen abgebildet und das Signal wieder zu den vorherigen Rotoren zurückgeleitet. Ab Ende 1938 verwendeten die Deutschen zwei zusätzliche Rotoren, womit aus fünf Rotoren drei bzw. vier ausgewählt wurden und die Sicherheit nochmals verstärkt wurde [1, S. 286]. Nach jedem Signaldurchlauf dreht sich der erste Rotor von rechts um eine Stelle weiter, wodurch sich das Ciphertextalphabet nach jedem Buchstaben ändert. Wurden alle Positionen des ersten Rotors durchlaufen, dreht sich der zweite Rotor eine Stelle weiter (analog bewegt sich der dritte Rotor erst nach allen Positionen des zweiten) [56, S. 35]. Mit der sogenannten Ringstellung wird außerdem eine Verschiebung der Verbindungen relativ zum Alphabet erreicht. Eine Ringstellung von 4 verschiebt beispielsweise die von „A“ ausgehende Verbindung innerhalb eines Rotors zu „D“. Eine zusätzliche monoalphabetische Substitution kann über das Steckbrett erreicht werden, wobei einzelne Buchstaben über Stecker auf andere Buchstaben abgebildet werden können (beispielsweise kann ein „N“ auf ein „F“ geändert werden). Die Konfiguration der Enigma, also die Kombination der verwendeten Rotoren (nummeriert mit I bis V), die verwendeten Ringstellungen und die Steckerverbindungen, ist Teil des Schlüssels. Das deutsche Militär wechselte diese Konfiguration täglich (Tagesschlüssel) [1, S. 285]. Zusätzlich wurde für die Verschlüsselung einer Nachricht eine Startposition der Rotoren verwendet. Diese Position wurde eingestellt, indem die Rotoren so eingesetzt wurden, dass für jeden Rotor in einem kleinen Fenster der richtige Buchstabe sichtbar wurde. Ein Beispiel wäre die Startposition „KEY“, was bedeutet, dass der rechte Rotor bei „Y“ startet, der mittlere bei „E“ und der letzte bei „K“.

Abbildung 3.4 zeigt ein vereinfachtes Schema des Signalflusses der Enigma mit drei Rotoren. Ausgehend von einer Batterie (1) fließt bei einem Drücken der Taste „A“ Strom entlang der roten Linie. Da für „A“ kein Stecker verwendet wird, gelangt das

Signal ungehindert zu den drei Rotoren. Bei dem Reflektor (6) wird das Signal wieder zurück geleitet und gelangt so zu „S“ (7). Durch die Steckerverbindung wird „S“ mit „D“ substituiert (8), woraufhin eine Lampe für „D“ aufleuchtet (9) und das Ergebnis der Verschlüsselung anzeigt.

3.3 Klassische Transpositions- und Kombinationschiffren

Im Gegensatz zu Substitutionschiffren findet bei Transpositionschiffren keine Ersetzung von Zeichen statt, stattdessen wird die Reihenfolge der Zeichen des Plaintexts verändert. Es handelt sich somit um eine Transposition des Plaintextes. Ein einfacher aber wichtiger Vertreter dieser Kategorie von Chiffren ist die sogenannte *Spaltentransposition*. Weitere Chiffren, die Teil dieser Chiffrenklasse sind, sind die in Kapitel 1 erwähnte Skytale und die doppelte Spaltentransposition, bei der zwei einfache Spaltentranspositionen hintereinander durchgeführt werden [55, S. 27ff.].

Als Kombinationschiffren werden im Zuge dieser Arbeit Chiffren bezeichnet, die die Prinzipien Substitution und Transposition vereinigen. Es hat sich historisch gezeigt, dass diese Kombination im Durchschnitt sicherer ist als Chiffren, die auf Substitution oder Transposition alleine setzen [55, S. 42]. Ein Beispiel für solch eine Kombinationschiffre ist ADFGX bzw. die Erweiterung ADFGVX.

3.3.1 Spaltentransposition

Die Spaltentransposition, manchmal auch eindeutiger einfache Spaltentransposition genannt, ist eine Transpositionschiffre, die sowohl eigenständig als auch als Komponente in komplexeren Chiffren verwendet wurde. Beispiele für letzteren Anwendungsfall sind die bereits erwähnten Chiffren ADFG(V)X und die doppelte Spaltentransposition.

Aufgrund ihrer Einfachheit ist die Spaltentransposition historisch eine der meistverwendetsten Transpositionschiffren [62]. Die Irish Republican Army (IRA) nutzte beispielsweise in den 1920er-Jahren die Spaltentransposition ausgiebig, wie Tom Mahon und James Gillogly in ihrem Buch *Decoding the IRA* beschreiben [63]. Um die Sicherheit der einfachen Spaltentransposition zu erhöhen, ergänzte die IRA die Chiffre um verschiedene Modifikationen wie etwa der Verwendung von zusätzlichen Spalten, die bedeutungslose Buchstaben zur zusätzlichen Verschleierung der Nachricht enthielten [63, S. 28f.].

Das der Spaltentransposition zugrundeliegende Prinzip beschränkt sich im Wesentlichen darauf, einen Plaintext in eine Matrix zu schreiben und die Spalten von dieser neu anzuordnen. Dazu wird zunächst ein Schlüssel benötigt, der angibt, in welcher Reihenfolge die Spalten angeordnet werden sollten. Es handelt sich bei dem Schlüssel also um eine Reihe von Zahlen. Da diese jedoch vergleichsweise schwer zu merken ist, kann auch ein Schlüsselwort oder eine kurze Phrase eingesetzt werden, wovon anschließend die Zahlenreihe abgeleitet wird [62]. Dabei wird jedem Buchstaben des Schlüsselworts bzw. der Phrase eine Zahl aufbauend auf seiner relativen Position im Alphabet zugewiesen. Kommt ein einzelner Buchstabe mehrmals vor, wird dessen Anfangswert aufsteigend inkrementiert. Aus dem Schlüsselwort *SECRET* wird somit der Schlüssel $[5, 2, 1, 4, 3, 6]$. Sobald der zu verwendende Schlüssel bekannt ist, kann die Matrix erstellt werden. Die Breite dieser wird durch die Länge des Schlüssels bestimmt. Der Plaintext wird an-

B	L	U	E
M	Y	S	E
C	R	E	T
T	E	X	T

→

B	E	L	U
M	E	Y	S
C	T	R	E
T	T	E	X

Abbildung 3.5: Matrix einer kompletten Spaltentransposition vor und nach der Umordnung der Spalten.

schließend Zeile für Zeile in die Matrix geschrieben und die Spalten werden gemäß dem Schlüssel neu angeordnet. Abschließend wird der Text der so neu geordneten Matrix spaltenweise ausgelesen, um den finalen Ciphertext zu erhalten. Anzumerken ist, dass die letzte Zeile der Matrix in der Regel unvollständig ist, was bedeutet, dass die Anzahl der Buchstaben in dieser kleiner ist als die Breite der Matrix. Lasry et al. nennen solch eine Spaltentransposition eine inkomplette Spaltentransposition [62]. Sind hingegen alle Zeilen gleich lang, handelt es sich um eine komplette Spaltentransposition [62]. Abbildung 3.5 veranschaulicht die Verschlüsselungsoperation der Spaltentransposition anhand eines Beispiels für die komplette Spaltentransposition. Als Plaintext dient *MYSECRETTEXT* und der Schlüssel ist *BLUE* ($[1, 3, 4, 2]$). Der resultierende Ciphertext lautet *MCTETTYRESEX*. Um den Ciphertext wieder zu entschlüsseln, muss diese Vorgehensweise rückwärts durchgeführt werden.

Für die Sicherheit der Spaltentransposition ist die Schlüssellänge $l \in \mathbb{N}$ entscheidend. Da es für eine Länge l $l!$ mögliche Schlüssel gibt, wird der Schlüsselraum zunehmend größer, wodurch manuelle und Brute-Force-basierte Methoden schwieriger werden. Die Grenze, bis zu welcher Brute-Force-Angriffe auf die Spaltentransposition möglich sind, liegt laut Lasry et al. bei 10 bis 15 Schlüsselzeichen [62].

3.3.2 ADFG(V)X

ADFGX ist eine Chiffre, die von der deutschen Armee im Ersten Weltkrieg 1918 eingeführt wurde [1, S. 43]. Basierend auf dem Namen, werden bei dieser Chiffre ausschließlich die Buchstaben „A“, „D“, „F“, „G“ und „X“ für den Ciphertext verwendet. Diese Zeichen wurden gewählt, da sich ihre Codierungen im Morse-Alphabet größtmöglich unterscheiden, wodurch Übertragungsfehler minimiert werden konnten [1, S. 43]. Die Verschlüsselung mittels ADFGX erfolgt in zwei Schritten: Zunächst wird eine monographische bipartite monoalphabetische Substitution durchgeführt. Der so entstehende Ciphertext wird anschließend nochmals mit einer Spaltentransposition verschlüsselt [64]. Dieser zweiter Verschlüsselungsschritt hat den Vorteil, dass die zuvor entstandenen Digramme mit hoher Wahrscheinlichkeit getrennt werden, was die Sicherheit der Chiffre erhöht [64].

Die Substitution erfolgt über ein sogenanntes *Polybius-Quadrat* [64]. Dabei handelt es sich bei ADFGX um eine 5×5-Matrix, die mit dem gesamten Alphabet mit Ausnahme von „J“ befüllt wird. Die Zeilen und Spalten werden anschließend mit „A“, „D“, „F“, „G“ und „X“ beschriftet, sodass eine Zeile mit dem Index $i \in \mathbb{N}_0$ den gleichen Namen wie Spalte mit dem Index i trägt. Um einen einzelnen Buchstaben des Plaintexts zu

×	**A**	**D**	**F**	**G**	**X**
A	S	E	C	R	T
D	A	B	D	F	G
F	H	I	K	L	M
G	N	O	P	Q	U
X	V	W	X	Y	Z

Abbildung 3.6: Polybius-Quadrat für ADFGX basierend auf dem Schlüsselwort *SECRET*.

verschlüsseln, wird der Zeilen- und der Spaltenname des jeweiligen Buchstabens in der Matrix verwendet. Zur Veranschaulichung wird wieder auf den Plaintext *HAGENBERG* und den Schlüssel *SECRET* zurückgegriffen. Um das Polybius-Quadrat zu befüllen, werden die einzigartigen Buchstaben des Schlüssels zu Beginn eingefügt, anschließend wird die Matrix mit dem übrigen Alphabet aufgefüllt. Dies ist nur eine denkbare Möglichkeit und es können auch andere Vorgehensweisen, wie zum Beispiel eine komplette Zufallsbefüllung, verwendet werden. Abbildung 3.6 zeigt dieses Polybius-Quadrat. *HAGENBERG* wird dementsprechend in *FA DA DX AD GA DD AD AG DX* verschlüsselt. Nach der folgenden Spaltentransposition mit dem Schlüssel *GEHEIM* ergibt sich der finale Ciphertext *ADDAAGFAADGADDDXDX*.

Ein Nachteil von ADFGX ist, dass nur die Buchstaben des Alphabets verschlüsselt werden können und Ziffern ausgeschrieben werden müssen, wodurch sich Nachrichten mit Ziffern verlängern. Um dies zu umgehen, wurde ADFGVX eingeführt, das statt einer 5×5-Matrix eine 6×6-Matrix verwendet, wodurch auch die Ziffern 0-9 in die Matrix aufgenommen werden können [64]. Die restliche Methode zur Ver- und Entschlüsselung (bei der einfach alle Schritte rückwärts ausgeführt werden müssen) bleibt gleich wie bei ADFGX. Der Plaintext *HAGENBERG* mit dem Substitutionsschlüssel *SECRET* und dem Transpositionsschlüssel *GEHEIM* wird somit zu *VDDXVGDAAAFADDDGAG* verschlüsselt. Bei ADFGVX gibt es insgesamt 36! Möglichkeiten, das Polybius-Quadrat anzuordnen, was 36! möglichen Schlüssel entspricht. Für den Transpositionsschlüssel wurden historisch Schlüssellängen von 16 bis 23 Zeichen verwendet [64]. Insgesamt ergeben sich somit

$$36! \cdot \sum_{l=16}^{23} l! \approx 10^{64} \tag{3.9}$$

mögliche Schlüssel für ADFGVX.

Kapitel 4

Stand des Wissens

Recurrent Neural Networks (RNN) finden in vielfältigen Aufgabenfeldern Anwendung, bei denen sequentielle Daten verarbeitet werden müssen. Beispiele dafür sind die Spracherkennung [65], die maschinelle Übersetzung (NMT) (z.B. [46], [66]) oder die Generierung von Bildbeschreibungen [67]. Mit der Einführung von Long Short-Term Memory (LSTM) in [24] werden herkömmliche RNN-Zellen verbessert, sodass der Trainingsprozess schneller durchgeführt werden kann und längere Sequenzen verarbeitet werden können. [46] schlägt Gated Recurrent Units (GRU) vor, die den LSTM-Zellen ähnlich sind, jedoch weniger Parameter benötigen und deshalb leichter zu berechnen sind. Ebenfalls im gleichen Paper [46] wird eine Encoder-Decoder-Architektur vorgestellt, die aus zwei miteinander verbundenen RNN besteht. Dies ermöglicht eine Verarbeitung von sequentiellen Daten, bei denen der Output anderer Länge als der Input ist. Solche Encoder-Decoder-Architekturen können über einen Attention-Mechanismus, bei dem der Fokus bei der Ausgabe eines Tokens auf bestimmte relevante Teile des Inputs gelegt wird, noch weiter verbessert werden, wie beispielsweise [50] und [51] demonstrieren. State-of-the-Art-Performance wird mittlerweile von Transformern erreicht, die nur mehr auf Attention basieren und keinen rekurrenten Netzanteil mehr besitzen [52]. Trotz dieser Neuerungen werden auch herkömmliche LSTM- und GRU-basierte Architekturen noch eingesetzt, wie unter anderem [5] zeigt. Ferner hat sich gezeigt, dass in bestimmten Anwendungsfällen Kombinationen von LSTM und GRU bessere Ergebnisse liefern als Standalone-LSTM bzw. Standalone-GRU. [68] verwendet eine solche hybride Architektur beispielsweise erfolgreich für eine Vorhersage von Währungskursen.

In den letzten Jahren wurde Deep Learning für verschiedene Aufgabenstellungen im Zusammenhang mit historischer Kryptografie eingesetzt. [5] zeigt, dass einfache RNN mit einer einzelnen LSTM-Zelle die Entschlüsselungsfunktion der drei polyalphabetischen Chiffren Vigenère, Autokey-Vigenère und Enigma mit hoher Accuracy ($\approx 99\%$) erlernen können. Einschränkungen bei dieser Arbeit sind jedoch die kurze Ciphertextlänge von 14 Zeichen und bei Vigenère und Autokey-Vigenère der kurze Schlüssel mit maximal 6 Zeichen. Zusätzlich demonstriert [5] auch, dass einfache neuronale KPA auf Vigenère und Autokey-Vigenère möglich sind. Die LSTM-RNN erreichen dabei rund 99% Accuracy bei Vigenère und rund 95% Accuracy bei Autokey-Vigenère. Auch bei dieser Aufgabenstellung ist die Schlüssellänge jedoch auf 1 bis 6 Zeichen limitiert. Eine weitere Einschränkung der Arbeit ist der Umstand, dass das verwendete Modell sehr

ineffizient in Hinblick auf die Trainingsdatenmenge ist, weshalb für jedes Training mindestens eine Million Trainingsdatenpunkte benötigt werden [5].

Dass auch Ciphertext-Only-Angriffe prinzipiell mit der Unterstützung von neuronalen Netzwerken möglich sind, zeigt [69]. Die Autoren verwenden ein sehr flaches Netzwerk mit einem Hidden Layer, das verwendet wird, um die Kryptoanalyse mit bereits bekannten Schwachstellen einer Chiffre zu automatisieren. Dieser Ansatz wird für die Caesar-Chiffre und Vigenère demonstriert, allerdings wird dem Netzwerk nicht der Ciphertext selbst, sondern eine vorverarbeitete Version übergeben. So ist der Input bei der Caesar-Chiffre die Häufigkeitsverteilung der Buchstaben des Ciphertexts, womit das neuronale Netzwerk eine Häufigkeitsanalyse automatisiert. [69]

Einen anderen Ansatz wählt [70]. Die Autoren Aldarrab und May verwenden einen Transformer auf Zeichenebene[1], um Ciphertext-Only Angriffe auf einfache monographische monopartite Substitutionschiffren durchzuführen. Im Gegensatz zu [69] wird bei dem Modell von [70] nicht der verwendete Schlüssel, sondern ausgehend von einem eingegebenen Ciphertext direkt der entsprechende Plaintext ausgegeben. Die Arbeit von Aldarrab und May unterstützt 14 Zielsprachen, weshalb keine Sprachidentifizierung als Vorverarbeitungsschritt benötigt wird.

Eine weitere Möglichkeit, Deep Learning im Kontext von historischer Kryptografie einzusetzen, ist die Identifizierung von Chiffren. [71] verwendet einen Ansatz auf Basis von Feature-Engineering zur Erkennung und Unterscheidung von 56 unterschiedlichen historischen Chiffren, während [72] zeigt, dass dies mit LSTM-RNN und Transformer auch auf Basis von Feature Learning möglich ist.

Im Kontext von herkömmlichen, automatisierten Angriffen auf die (einfache) Spaltentransposition präsentiert [62] eine Methode basierend auf zweistufigem Hill Climbing, die bei Ciphertext-Only-Angriffen Schlüssel bis zu einer Länge von 1 000 Zeichen rekonstruieren kann.

[1] Ein Token entspricht somit einem Buchstaben.

Kapitel 5

Erlernen der Entschlüsselungsfunktion

In diesem Kapitel wird die erste Aufgabenstellung, das Erlernen der Entschlüsselungsfunktion, behandelt. Ausgehend von den Ergebnissen von [5] wird nach einer Architektur gesucht, die bei gleicher Accuracy weniger Trainingsdatenpunkte benötigt als Greydanus' Architektur. Abschließend wird diese neue Architektur zusätzlich zu den bereits in [5] verwendeten Chiffren Vigenère, Autokey-Vigenère und Enigma auf die Caesar-Chiffre, die MASC, die Playfair-Chiffre, die Hill-Chiffre, die (einfache) Spaltentransposition und ADFGVX angewandt.

5.1 Ausgangssituation

Die von dem neuronalen Netzwerk zu approximierende Zielfunktion ist als die Entschlüsselungsfunktion D einer Chiffre definiert. Es gilt dabei

$$D(c, k) = m. \tag{5.1}$$

Der Plaintext m, der Ciphertext c und der Schlüssel k sind Sequenzen von Zeichen eines Alphabets A. Im Rahmen dieser Arbeit wird für A grundsätzlich das lateinische Alphabet bestehend aus 26 Großbuchstaben gewählt. Für manche Chiffren wird A jedoch angepasst:

- Caesar-Chiffre: Der Schlüssel k ist bei dieser Chiffre ein Verschiebungswert, weshalb k eine Sequenz aus maximal zwei Zeichen des Alphabets B ist, das aus den Ziffern $0 - 9$ besteht.
- ADFGVX: Für ein vollständiges Polybius-Quadrat werden auch die Ziffern $0 - 9$ benötigt, weshalb diese mit A zu einem Alphabet C kombiniert werden. Die verwendeten m enthalten bei dieser Chiffre somit auch Ziffern.

Das Erlernen der Entschlüsselungsfunktion ist die primäre Problemstellung in Greydanus' Arbeit. Ziel von [5] ist zu zeigen, dass RNN den Entschlüsselungsalgorithmus von ausgewählten polyalphabetischen Chiffren, v.a. der Enigma, erlernen können. Wie in [5] angeführt, ist es explizit nicht das Ziel, mit dieser erlernten Algorithmusrepräsentation Angriffe durchzuführen (*„[...] our work is not aimed at 'cracking' the Enigma cipher.“* [5]). Um sichergehen zu können, dass tatsächlich nur die Entschlüsselungsfunktion erlernt wird, werden in [5] nur zufällige m und k verwendet. Um diese zu generieren,

werden die notwendigen Zeichen zufällig gleichverteilt aus A gezogen. Werden stattdessen m verwendet, die auf natürlicher Sprache basieren, besteht die Gefahr, dass das Modell im Hintergrund eine Art Sprachmodell erlernt. Dies bedeutet, dass das Modell die Buchstabenverteilungen der Sprache lernt und somit Outputs nicht mehr nur auf Basis der erlernten Entschlüsselungsfunktion ausgibt. Denkbar ist beispielsweise bei einem englischen Sprachmodell, dass nach dem Buchstaben „t“ „h“ ausgegeben wird, obwohl der entsprechende Ciphertextbuchstabe gar nicht entschlüsselt werden konnte. Die Gesamtlänge der Trainingssequenzen in [5] beträgt 20, wobei davon 14 Zeichen auf den Plain-/Ciphertext entfallen und die restlichen 6 Zeichen für den Schlüssel verwendet werden (zumindest bei Vigenère und Autokey-Vigenère). Bei der Enigma besteht der Schlüssel nur aus 3 Zeichen, sodass die restlichen Schlüsselstellen mit dem „-“-Symbol gepadded werden. Der Schlüssel wird vor den Cipher- bzw. Plaintext gestellt. Insgesamt ergibt sich somit folgende Sequenzstruktur (Beispiel zeigt einen Vigenère-Datenpunkt mit dem Schlüssel SPWPZN):

Plaintext: S P W P Z N T C G B P M R O G R A P Q P
Ciphertext: S P W P Z N L R C Q O Z J D C G Z C I E

Beim Erlernen der Entschlüsselungsfunktion handelt es sich um ein Problem des Supervised Learnings. Der Ciphertext inklusive des angehängten Schlüssels dient als Trainingsdatenpunkt, der entsprechende Plaintext (inklusive des Schlüssels) ist das dazugehörige Label. Greydanus generiert die Trainingsdaten on-the-fly zum Trainingszeitpunkt, wodurch kein großer Datensatz permanent gespeichert werden muss. In [5] wird auch angeführt, dass dieser Ansatz die Wahrscheinlichkeit von Overfitting reduziert, einen Beweis dieser Aussage liefert die Arbeit jedoch nicht.

Das in [5] eingesetzte rekurrente neuronale Netzwerk basiert auf einer einfachen Architektur mit einer einzelnen LSTM-Zelle mit maximal 512 Units. Lediglich bei der Enigma wird eine LSTM-Zelle mit 3000 Units verwendet. Auf die LSTM-Schicht erfolgt eine Dropout-Regularisierung. Im Source Code von Greydanus wird jedoch standardmäßig eine Dropout-Rate von 0 verwendet, weshalb praktisch keine Regularisierung stattfindet. Nach der Regularisierungsschicht folgt ein Softmax-Layer, über den für jeden Time-Step ein Zeichen ausgegeben wird. Es findet somit eine Klassifikation über 27 Klassen statt, die sich aus den 26 Buchstaben des lateinischen Alphabets und dem Padding-Zeichen „-“ ergeben. Die Gewichte des Netzwerks werden mit der Initialisierungsmethode nach Glorot initialisiert [5]. Als Optimizer verwendet Greydanus Adam mit einer Lernrate $\eta = 5 \cdot 10^{-4}$. Trainiert wird mit dem Mini-Batch-Gradientenverfahren und einer Batch-Größe von 50 Trainingsdatenpunkten. Mit diesen Konfigurationen erreicht [5] bei Vigenère, Autokey-Vigenère und Enigma 99% Accuracy im Training. Eine Limitierung ist jedoch, dass das Training dieser Architektur sehr datenineffizient ist. Es werden schon bei der Vigenère-Chiffre, der einfachsten in [5] verwendeten Chiffre, mindestens eine Million Trainingsdatenpunkte benötigt, um diese hohe Accuracy zu erreichen.

5.2 Optimierung der Architektur

Greydanus' Arbeit erschien 2017. In demselben Jahr wurde ebenfalls die Transformer-Architektur von Vaswani et al. in [52] präsentiert, welche den neuen State of the Art

bei der Verarbeitung von natürlicher Sprache darstellte. Zusätzlich entwickelt sich der Deep-Learning-Bereich und die dafür verfügbare Hardware schnell weiter, weshalb anzunehmen ist, dass mittlerweile bessere Architekturen für die Problemstellung verfügbar sind. Bevor somit das Erlernen der Entschlüsselungsfunktionen von weiteren Chiffren getestet wird, soll zunächst eine stärkere Netzwerkarchitektur gefunden werden. Aufgrund der bereits hohen Accuracy von Greydanus' einfachem LSTM-Modell wird dieses „stärker" im Sinne eines dateneffizienteren Trainings bei gleichbleibender Accuracy definiert. Ziel der Architekturverbesserung ist es somit, eine Architektur zu erreichen, die weniger Trainingsdatenpunkte für das Erreichen einer Trainingsaccuracy von 99% benötigt. Als Referenzchiffre dient dabei die Vigenère-Chiffre.

Der Trainingsdatensatz für die Vigenère-Chiffre wird analog zu [5] erstellt, womit sichergestellt wird, dass auch die verbesserte Architektur nicht zu einem impliziten Sprachmodell führt. Ein Unterschied ist jedoch, dass k nicht auf exakt 6 Zeichen festgelegt wird, sondern Schlüssel der Länge $1 - 6$ Zeichen verwendet werden. Aufgrund der sehr hohen beobachteten Accuracy in Greydanus' Arbeit ist anzunehmen, dass auch diese leicht schwerere Aufgabe mit hoher Accuracy erlernt werden kann. Ein Vorteil von verschiedenen Schlüssellängen ist, dass der Datensatz realistischer wird, auch wenn natürlich dennoch die künstliche Limitierung auf maximal 6 Zeichen bestehen bleibt. Die Länge eines Plaintexts m bleibt bei 14 Zeichen. Da auf dem verwendeten Nvidia DGX A100 Server Speicherplatz kein limitierender Faktor ist, wird der gesamte Datensatz mit 2 000 000 Datenpunkten im Voraus erstellt, anstatt diese on-the-fly während des Trainings zu erstellen. Aufgrund der Vielzahl an anfallenden Trainings, reduziert dieser Ansatz die insgesamt anfallende Trainingszeit. Die Gesamtgröße des Trainingsdatensatzes benötigt 86 MB Speicherplatz, der ebenfalls gleich generierte Evaluierungsdatensatz mit 1500 Datenpunkten beschränkt sich auf 71,3 KB.

Trainiert wird mit zwei Nvidia A100 40 GB GPUs der Nvidia DGX A100. Die Hyperparameter Batch-Größe, Lernrate und Optimizer (inkl. dessen Hyperparameter) bleiben bei den Architekturevaluierungen zunächst unverändert. Lediglich die Dropout-Rate wurde auf 0.5 erhöht, um ein gewisses Maß an Regularisierung zu erreichen. Dies kann in Anbetracht der sehr hohen Accuracy vermeiden, dass Overfitting auftritt. Wie bei Greydanus entspricht eine Epoche einem Trainingsschritt, also einem Mini-Batch mit 50 Trainingsdatenpunkten. Dies entspricht zwar nicht der zuvor gegebenen Definition einer Epoche, jedoch ermöglicht es einen sofortigen Abbruch des Trainings, sobald die angestrebte Accuracy nach einem Trainingsschritt erreicht wurde. Dazu wird ein einfacher TensorFlow-Callback[1] verwendet, der am Ende einer Epoche die aktuelle Accuracy des Modells überprüft und gegebenenfalls das Training abbricht. Da ein Callback eine Funktion ist, die zu einem bestimmten Zeitpunkt von TensorFlow aufgerufen wird, wäre dies prinzipiell auch am Ende jedes Trainingsschritts innerhalb einer Epoche möglich. Es ist jedoch die Auswertung der benötigten Trainingsdaten einfacher, wenn eine Epoche einem Mini-Batch entspricht. So muss am Ende eines Trainings lediglich die Anzahl der benötigten Epochen mit der Batch-Größe multipliziert werden. Werden hingegen in einer Epoche wie üblich mehrere Trainingsschritte durchgeführt, muss am Ende des Trainings evaluiert werden, wie viele Batches in der letzten Epoche verwendet wurden. Diese müssen dann zu dem Produkt aus Epochen mal Trainingsschritten pro Epoche

[1] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/Callback

addiert werden. Das Ergebnis muss anschließend nochmals mit der Batch-Größe multipliziert werden, erst dann ist die Anzahl der verwendeten Trainingsdaten bekannt. Es ist also ersichtlich, dass eine Abweichung der üblichen Epochen-Definition in diesem konkreten Fall einen Vorteil bietet.

Bevor die alphabetischen Sequenzen als Trainingsdaten verwendet werden können, müssen diese in eine für das System verarbeitbare Form überführt werden. Wie in [5] wird dafür *One-Hot Encoding* verwendet. Dabei wird jedes Zeichen in einen Vektor der Dimension $k \in \mathbb{N}$ umgewandelt, wobei k der Anzahl der verfügbaren Klassen, also in diesem Fall 27, entspricht. Alle Komponenten des Vektors besitzen dabei den Wert 0, lediglich die Komponente an der Stelle des Zeichenindex $i \in \mathbb{N}_0$ hat den Wert 1. Hat der Buchstabe „A“ beispielsweise den Index 0, besitzt die zugehörige Vektordarstellung nur bei dieser Komponente den Wert 1.

Bei den ersten Trainings wird schnell ersichtlich, dass die Accuracy von 99% in vielen Fällen (z.B. beim Einsatz einzelner LSTM-Zellen ab 512 Units) leicht erreicht wird. Das Trainingsziel wird deshalb von den 99% von Greydanus auf $99,9\%$ erhöht. Um Greydanus' Architektur in Hinblick auf diese zusätzliche Änderung – neben der Änderung der Schlüssellänge – bewerten zu können, wird zu Beginn eine erneute Evaluierung seiner Architektur mit den angepassten Rahmenbedingungen durchgeführt. Das LSTM-RNN wird dabei zehn Mal trainiert und anschließend werden die durchschnittlich benötigten Trainingsdatenpunkte ermittelt. Dieses wiederholte Trainieren der Architektur ist hilfreich, da in einem neuronalen Netzwerk mehrere Zufallsfaktoren (z.B. die Parameterinitialisierung) vorhanden sind, die aber alle Einfluss auf die Leistung des resultierenden Modells haben können. Ein einzelnes Trainingsergebnis ist deshalb in Hinblick auf die benötigte Datenmenge nicht aussagekräftig, da es mehr das Resultat einer zufälligerweise guten Initialisierung als wirklich einer Architekturverbesserung sein kann. Die erneute Evaluierung von Greydanus LSTM-RNN ergibt, dass dieses ungefähr $1,3$ Millionen[2] Trainingsdatenpunkte für das angepasste Trainingsziel benötigt.

Der einfachste Weg, bei ausreichend vorhandener Rechenleistung die Lernfähigkeit der Architektur zu steigern, ist die Anzahl der Neuronen zu erhöhen. Bei RNN entspricht dies einer Erhöhung der Units. Für LSTM-RNN mit 1024, 2048, 4096 und 5000 Units werden deshalb jeweils fünf Trainings durchgeführt. Anschließend wird wieder der Durchschnitt der benötigten Trainingsdatenpunkte bestimmt. 5000 Units ist ungefähr die Grenze, ab der eine weitere Erhöhung problematisch in Hinblick auf die Trainingszeit und den benötigten VRAM wird. Wie Tabelle 5.1 mit den Durchschnittsergebnissen zeigt, führt eine weitere Steigung der Unit-Anzahl auch zu keiner Verbesserung mehr. Es ist jedoch klar ersichtlich, dass bereits eine Erhöhung der Units von 512 auf 4096 eine Reduktion der notwendigen Trainingsdatenpunkte um $1\,361\,525 - 480\,480 = 881\,045$ erreicht.

Eine weitere Möglichkeit, LSTM-RNN zu verbessern, besteht darin, mehrere LSTM-Zellen hintereinander zu verwenden (*Stacked LSTM*). Da ein solches Stacking die Parameteranzahl und somit die Trainingsdauer stark erhöht, dauern Durchschnittsermittlungen mit fünf oder mehr Trainings sehr lange[3]. Als erste Abschätzung werden deshalb

[2] Genau handelt es sich um 1 361 525 Datenpunkte. Aufgrund der angesprochenen Schwankungen ist eine zu feingranulare Betrachtung jedoch sinnlos.

[3] Auch die Erhöhung der Units verlängert die Trainingsdauer und erhöht die Parameteranzahl. Bei einem LSTM mit drei Zellen und 4096 Units besitzt die Architektur allerdings nochmals ca. 66 Millionen

Units	Trainingsdatenpunkte (⌀)
1024	591 640
2048	596 960
4096	480 480
5000	503 350

Tabelle 5.1: Durchschnittsergebnisse von einem LSTM-RNN mit verschiedener Unit-Anzahl. Die Ergebnisse basieren auf jeweils fünf Trainings.

Zellen	Trainingsdatenpunkte (⌀)
1	571 925
2	1 087 925
3	439 825

Tabelle 5.2: Durchschnittsergebnisse von einem LSTM-RNN bei verschiedener Zellen-Anzahl. Die Ergebnisse basieren auf jeweils zwei Trainings.

lediglich zwei Trainings mit LSTM-RNN mit einer, zwei und drei LSTM-Zellen und jeweils 4096 Units durchgeführt. Tabelle 5.2 zeigt die Ergebnisse dieser Evaluierung. Auffallend ist, dass RNN mit drei LSTM-Zellen anscheinend eine bessere Leistung in Hinblick auf die Trainingsdaten erbringen als RNN mit lediglich einer LSTM-Zelle. Aufgrund der wenigen durchgeführten Trainings wird deshalb nochmals eine separate Evaluierung dieser Architektur mit drei Trainings durchgeführt. Dabei zeigt sich, dass in einem Training tatsächlich wieder eine bessere Leistung als bei einer einzigen LSTM-Zelle erbracht wurde (518 050 benötigte Datenpunkte). In den anderen beiden Trainings konnte das Netzwerk das geforderte Trainingsziel von $99,9\%$ jedoch innerhalb von 30 000 Epochen (entspricht $1,5$ Millionen Datenpunkten) gar nicht erreichen. Dies deutet daraufhin, dass diese Architektur zwar sehr gute Ergebnisse liefern kann, dabei aber sehr inkonsistent ist. Aus diesem Grund wird im weiteren Verlauf dem RNN mit der einzelnen LSTM-Zelle der Vorzug gegeben. Insgesamt decken sich diese Ergebnisse mit [5], in dem auch RNN mit mehreren Zellen schlechtere Leistungen zeigen.

Die Wahl des Optimizers kann die Trainingsleistung entscheidend beeinflussen. Es werden deshalb die Optimizer Adam, Nadam [73], RMSProp und Adadelta[4] [74] mit dem LSTM-RNN mit 4096 Units getestet. Es stellt sich jedoch heraus, dass im Durchschnitt die Wahl des Optimizers bei dieser Aufgabe keine Rolle spielt.

Da GRU-Zellen wie bereits beschrieben einfacher zu berechnen sind, können diese möglicherweise mit mehr Units verwendet werden. Dies kann zu einer besseren Leistung führen, auch wenn bei LSTM ab 4096 Units kein Leistungszugewinn mehr erkennbar war. Da RNN mit sehr vielen Units lange für das Training benötigen, werden bei GRU anstatt fünf nur mehr drei Trainings für die Durchschnittsberechnung herangezogen. Treten dabei Ergebnisse auf, die durch die vorkommenden Ergebnisschwankungen erklärt werden können, werden stattdessen nochmals spezifischere Evaluierungen durchgeführt. Tabelle 5.3 zeigt die Ergebnisse der Evaluierungen eines RNN mit einer einzelnen

Parameter mehr als ein LSTM mit einer Zelle und 8192 Units (336 146 459 : 269 574 171 Parameter).

[4] Adadelta ist eine Erweiterung von dem in Kapitel 2.6 besprochenen Adagrad.

Units	Trainingsdatenpunkte (⌀)
1024	1 500 000
2048	1 121 066
4096	592 483
8192	358 600
10 000	356 416

Tabelle 5.3: Durchschnittsergebnisse von einem GRU-RNN mit verschiedener Unit-Anzahl. Die Ergebnisse basieren auf jeweils drei Trainings.

Zellen	Trainingsdatenpunkte (⌀)
1	592 483
2	435 233
3	1 267 766

Tabelle 5.4: Durchschnittsergebnisse von einem GRU-RNN bei verschiedener Zellen-Anzahl. Die Ergebnisse basieren auf jeweils drei Trainings.

GRU-Zelle mit 1024, 2048, 4096, 8192 und 10 000 Units. Es zeigt sich, dass GRU-Zellen bei dieser Aufgabenstellung bei gleicher Unit-Anzahl weniger leistungsfähig als LSTM-Zellen sind. Bei 1024 Units ist das GRU-RNN nicht in der Lage, das Trainingsziel innerhalb von 30 000 Epochen zu erreichen. Auch mit 4096 Units benötigt das GRU-RNN mehr Datenpunkte als das LSTM-RNN mit der entsprechend Unit-Anzahl. Beim Einsatz von GRU-Zellen kann die verwendete Hardware jedoch auch Netzwerke mit 8192 und 10 000 Units in annehmbarer Zeit trainieren, die wiederum dateneffizienter als das beste LSTM-RNN sind. Da der Unterschied zwischen 8192 und 10 000 Units minimal ist, gleichzeitig aber der Speicherbedarf und die Trainingsdauer bei der höheren Unit-Anzahl steigt, wird der Architektur mit 8192 Units im weiteren Verlauf der Vorzug gegeben.

Auch bei der GRU-Evaluierung werden wieder RNN mit mehreren Zellen getestet. Auch in diesem Fall werden wie bei den Einzelzellen-GRU-RNN drei Trainings durchgeführt. Da 8192 und 10 000 Units in der Triple-Stacked-Variante allerdings zu viel Arbeitsspeicher benötigen, werden bei den Stacked-Architekturen 4096 Units eingesetzt. Tabelle 5.4 zeigt die Resultate der Evaluierung. Es zeigt sich dabei, dass eine Architektur mit zwei Zellen prinzipiell eine gute Leistung erbringt, jedoch ist ein GRU-RNN mit nur einer Zelle und 8192 bzw. 10 000 Units noch dateneffizienter. Letztere Architektur zeigt außerdem eine bessere Leistung als die beste LSTM-Architektur (einzelne LSTM-Zelle, 4096 Units).

Im nächsten Schritt wird getestet, ob eine Kombination von LSTM und GRU nochmals bessere Ergebnisse hervorbringt. Zunächst wird eine LSTM-GRU-Architektur getestet, bei der eine GRU-Zelle auf eine LSTM-Zelle folgt. Es werden dabei fünf Trainings mit 1024, 2048, 4096 und 8192 Units durchgeführt, wobei die LSTM- und die GRU-Zelle gleich viele Units verwenden. Tabelle 5.5 fasst die Ergebnisse der Trainings zusammen. Es wird dabei ersichtlich, dass bis inklusive 2048 Units das LSTM-GRU-RNN das Trainingsziel nicht innerhalb von 30 000 Epochen erreichen kann, ab 4096

Units	Trainingsdatenpunkte (⌀)
1024	1 500 000
2048	1 500 000
4096	333 150
8192	300 950

Tabelle 5.5: Durchschnittsergebnisse von einem LSTM-GRU-RNN mit verschiedener Unit-Anzahl. Die Ergebnisse basieren auf jeweils fünf Trainings.

Units	Trainingsdatenpunkte (⌀)
1024	1 500 000
2048	904 110
4096	401 440
8192	272 100

Tabelle 5.6: Durchschnittsergebnisse von einem GRU-LSTM-RNN mit verschiedener Unit-Anzahl. Die Ergebnisse basieren auf jeweils fünf Trainings.

Units erzielt es aber bessere Ergebnisse als die bisher beste Architektur, das GRU-RNN mit 10 000 Units. Ein weiterer Test hat jedoch gezeigt, dass eine weitere Steigerung der Unit-Anzahl zu Speicherproblemen führt, weshalb bei der verwendeten Hardware mit 40 GB VRAM 8192 Units das Maximum darstellt.

Die gleiche Evaluierung wird nochmals durchgeführt, diesmal allerdings mit einem GRU-LSTM-RNN. Die Reihenfolge der zwei Zellen wird also getauscht. Tabelle 5.6 zeigt wieder die Ergebnisse. Im Vergleich zu dem LSTM-GRU-RNN ist die GRU-LSTM-Variante teilweise besser (2048 Units) und teilweise schlechter (4096 Units). Bei der höchsten Unit-Anzahl (8192) braucht sie jedoch ca. 30 000 Trainingsdatenpunkte weniger als das LSTM-GRU-RNN, was das beste bisherige Ergebnis darstellt.

Nach den bisher getesteten traditionellen RNN-Architekturen werden auch modernere Architekturen evaluiert. Begonnen wird mit einer einfachen Encoder-Decoder-Architektur, bei der sowohl der Encoder als auch der Decoder aus einem RNN mit einer einzelnen LSTM-Zelle mit 2048 Units besteht. Da wie in Kapitel 2.8 beschrieben ein Decoder <SOS>- und <EOS>-Token benötigt, werden diese an die Labels an die entsprechende Stelle eingefügt. Da es somit zwei zusätzliche Zeichen gibt, muss der Softmax-Layer des Decoders von 27 auf 29 Neuronen vergrößert werden. Alle übrigen Hyperparameter bleiben gleich wie bei den herkömmlichen RNN. Bei dem Training des Encoder-Decoder zeigt sich, dass diese Architektur nicht für das Erlernen der Entschlüsselungsfunktion geeignet ist, da die Accuracy ungefähr bei Epoche 5000 auf den Wert 35% steigt, anschließend aber auf diesem Plateau bleibt. Es ist somit kein Lernfortschritt festzustellen. Andere Lernraten wie $\eta = 1 \cdot 10^{-3}$ oder $\eta = 1 \cdot 10^{-4}$ verbessern das Training nicht. Auch wenn der Encoder-Decoder mit einer Bahdanau-Attention-Schicht zu einer Attention-Architektur erweitert wird, bleibt die Trainingsleistung konstant niedrig auf $\approx 35\%$ Accuracy. Die gleiche Accuracy erreicht auch eine Transformer-Architektur nach den Spezifikationen aus [52]. Insgesamt lässt sich also sagen, dass Architekturen, die

üblicherweise bei NLP[5]-Aufgaben wie der maschinellen Übersetzung bessere Ergebnisse liefern als einfache LSTM/GRU-RNN, diesen bei dem Erlernen der Entschlüsselungsfunktion der Vigenère-Chiffre unterlegen sind.

Im Zuge der Evaluierung verschiedener Architekturen konnte somit festgestellt werden, dass die für diese konkrete Aufgabenstellung beste Architektur ein GRU-LSTM-RNN mit 8192 Units ist. Im Vergleich zu der von Greydanus für das Erlernen der Vigenère-Entschlüsselungsfunktion verwendeten Architektur braucht das GRU-LSTM-RNN $1\,361\,525 - 272\,100 = 1\,089\,425$ Trainingsdatenpunkte weniger, um im Training $99,9\%$ Accuracy zu erreichen. Es wird somit diese Architektur in den folgenden Kapiteln weiterverwendet.

5.3 Anwendung der Architektur auf weitere Chiffren

Zur Evaluierung, inwiefern die Ergebnisse von [5] auf weitere Chiffren übertragbar sind, wird ein GRU-LSTM-RNN mit weiteren Entschlüsselungsfunktionen trainiert. Bei diesen weiteren Chiffren handelt es sich um die Substitutionschiffren Caesar, MASC, Playfair und Hill, um die Spaltentransposition und um die Kombinationschiffre ADFGVX. Um überprüfen zu können, ob die neue Architektur auch die Chiffren Autokey-Vigenère und Enigma ähnlich gut erlernen kann wie das LSTM-RNN in [5], sollen auch diese beiden Chiffren erlernt werden.

Die Datensätze all dieser Chiffren werden nach der in Kapitel 5.1 beschriebenen Methode zufällig generiert. Trainingsdatensätze enthalten dabei wieder 2 000 000 Datenpunkte, die Evaluierungsdatensätze 1500. Es wird für jede Chiffre ein Modell trainiert, anschließend wird dieses mit dem jeweiligen Evaluierungsdatensatz evaluiert. Als Metrik dient wieder die Accuracy. Es ist anzumerken, dass durch diesen Evaluierungsansatz die Ergebnisse nicht direkt mit denen von [5] verglichen werden können. Greydanus führt in [5] keine Evaluierung mit einem Evaluierungsdatensatz durch, sondern testet für jedes Modell 20 zufällige Sequenzen, die mit dem Schlüssel „KEY“ verschlüsselt werden. Der Schlüssel „KEY“ wird dabei während dem Training nicht verwendet. Die sehr hohe Accuracy von 99% ist in [5] die Leistung des Modells zum Ende des Trainings. Eine Evaluierung mit einem eigenen Evaluierungsdatensatz ist jedoch aufgrund des größeren Umfangs an Evaluierungsdatenpunkten aussagekräftiger (v.a. im Hinblick auf Overfitting), weshalb nicht auf die Methode von Greydanus zurückgegriffen wird.

Die Datenpunkte aller Datensätze werden grundsätzlich zufällig auf Basis des lateinischen Großbuchstabenalphabets A erzeugt. Der Schlüssel jedes Datenpunkts besitzt eine Länge von $1 - 6$ Zeichen, der Plaintext besteht aus 14 Zeichen. Da die verwendete Bibliothek *tf.keras*[6] in einem Trainingsbatch nur Sequenzen gleicher Länge akzeptiert, werden Schlüssel variabler Länge immer mit dem Zeichen „-“ auf eine Länge von 6 gepadded. Im Gegensatz zu [5] findet bei Chiffren mit fixer Schlüssellänge (z.B. Enigma) kein solches Padding statt, da dies die entstehende Sequenz unnötig verlängern würde.

Bei manchen Chiffren muss von dieser Vorgehensweise abgewichen werden. Für die einzelnen Chiffren gelten folgende Abweichungen, Konfigurationen und Anmerkungen:

- Caesar-Chiffre: Der Schlüssel der Caesar-Chiffre besteht aus einer Zahl des Inter-

[5] Natural Language Processing

[6] https://www.tensorflow.org/api_docs/python/tf/keras

valls $[1, 25]$, die die Verschiebung des Alphabets angibt. Zur Darstellung dieser Zahl werden die Ziffern $0 - 9$ benötigt, weshalb als Basis für den Schlüssel k ein Alphabet B verwendet wird, das nur aus diesen Ziffern besteht. m wird weiterhin aus A generiert. Da die maximale Verschiebung 25 ist, beschränkt sich die Schlüssellänge auf zwei Zeichen.

- MASC: Bei der einfachen monoalphabetischen Substitution wird der Schlüssel durch ein Schlüsselwort der variablen Länge von $1 - 6$ Zeichen dargestellt. Das Ciphertextalphabet ergibt sich, indem an die einzigartigen Buchstaben des Schlüsselworts die übrigen fehlenden Zeichen des lateinischen Alphabets angehängt werden.
- Playfair-Chiffre: Wie bei der MASC wird die Verschlüsselungsmatrix mit den einzigartigen Zeichen eines Schlüsselworts der variablen Länge von $1 - 6$ Zeichen befüllt. Der Buchstabe „J“ wird dabei wie in Kapitel 3.2.3 angeführt ausgelassen, stattdessen wird bei einem Vorkommen in dem Schlüsselwort „I“ verwendet. Die restliche Matrix wird mit den übrigen Buchstaben des lateinischen Alphabets aufgefüllt.
- Hill-Chiffre: Für die Hill-Chiffre wird eine Matrix-Dimension $d = 3$ verwendet. Da somit eine 3×3-Matrix entsteht, muss der Schlüssel aus genau 9 Zeichen bestehen. Um die Matrix in einem String darstellen zu können, wird sie von einem Schlüsselwort abgeleitet. Dazu wird jeder Buchstabe in eine Zahl nach dem Schema $A = 0$ bis $Z = 25$ umgewandelt. Anschließend wird diese Zahlensequenz zeilenweise in die Matrix geschrieben. Damit kein Padding verwendet werden muss, besteht jeder Plaintext aus 15 Zeichen, wodurch fünf Substitutionsschritte mit jeweils 3 Zeichen durchgeführt werden.
- Enigma: Zur Generierung der Datensätze für die Enigma wird das Python-Modul *py-enigma*[7] in der Version 0.1 verwendet. Bei der Initialisierung der damit erstellten virtuellen Enigma müssen die in Kapitel 3.2.5 beschriebenen Konfigurationen angegeben werden. Sowohl für den Trainings- als auch den Evaluierungsdatensatz wird die virtuelle Enigma mit den drei Rotoren „I“, „II“ und „III“ in dieser Reihenfolge verwendet. Als Reflektor wird Typ „B“ eingesetzt und die Ringstellungen werden wie bei [5] auf 2,14,8 gestellt. Auf Steckerverbindungen wird verzichtet. Diese Konfigurationen sind für alle Datenpunkte gleich und werden nicht verändert. Als Schlüssel k dient die Startposition der drei Rotoren, wodurch k aus drei zufälligen Großbuchstaben besteht.
- Spaltentransposition: Der Schlüssel der Spaltentransposition wird in alphabetischer Form angegeben, wodurch das neuronale Netzwerk selbst die Umwandlung in die numerische Form erlernen muss. Durch die variable Schlüssellänge wird sowohl die inkomplette als auch die komplette Spaltentransposition von den Datensätzen repräsentiert.
- ADFGVX: Bei ADFGVX werden zwei Alphabete verwendet: A für die Generierung des Transpositionsschlüssel k_t und C für den Substitutionsschlüssel k_s und den Plaintext. C besteht dabei aus den Großbuchstaben des lateinischen Alphabets und den Ziffern $0 - 9$. Dieses Alphabet fasst somit die Alphabete A und B zusammen. k_t ist wie bei der Spaltentransposition von variabler Länge, k_s hat hin-

[7] https://pypi.org/project/py-enigma/

gegen eine fixe Länge von 6 Zeichen. Das Polybios-Quadrat für die Substitution wird mit den einzigartigen Zeichen von k_s zeilenweise gefüllt. Die übrigen Zeichen von C vervollständigen anschließend das Quadrat. k_t und k_s ergeben konkateniert den Gesamtschlüssel, der wie in Kapitel 5.1 mit einem Beispiel illustriert an den Plain- bzw. Ciphertext angehängt wird.

Für alle Chiffren wird prinzipiell das GRU-LSTM-RNN mit 8192 Units aus Kapitel 5.2 eingesetzt. Für die Spaltentransposition und polypartite Substitutionen (Playfair, Hill, ADFGVX) wird jedoch die bidirektionale Variante dieser Architektur eingesetzt. Da bei solchen BRNN zwei RNN parallel betrieben werden, wird die Unit-Anzahl auf 4096 reduziert, wodurch Speicherprobleme durch zu viele Parameter vermieden werden. Ansonsten bleiben alle Hyperparameter gleich. Aus folgenden Gründen wird bei diesen Chiffren ein BRNN benötigt:

- Spaltentransposition: Es kann passieren, dass ein Buchstabe, der relativ am Anfang des Plaintexts vorkommt, im Ciphertext am Ende der Sequenz steht. Da für die Ausgabe bei dem entsprechenden Time-Step am Anfang jedoch schon die Information von diesem Ciphertextbuchstaben benötigt wird, muss es eine Möglichkeit für das RNN geben, Information vom Ende der Sequenz zum Anfang der Sequenz zu transportieren.
- Polypartite Substitutionschiffren: Wird ein Plaintextzeichen durch mehrere Ciphertextzeichen repräsentiert, muss für die Ausgabe dieses Plaintextzeichens zum Time-Step t_1 auch Information aus den folgenden Time-Steps vorhanden sein. Bei Playfair werden beispielsweise Einheiten aus zwei Plaintextzeichen durch zwei Ciphertextzeichen repräsentiert (= bigraphisch bipartit). Zum Zeitpunkt t_1 muss somit das erste Plaintextzeichen ausgegeben werden, worauf aber auch das zweite Ciphertextzeichen zum Zeitpunkt t_2 Einfluss hat. Aus diesem Grund muss wie bei der Spaltentransposition Information in beide Richtungen fließen können.

In Tabelle 5.7 werden die Ergebnisse der Evaluierungen der nach diesen Grundsätzen trainierten Modelle aufgelistet. Auffallend ist, dass viele Chiffren, auch komplexe wie ADFGVX und Enigma, in einem ähnlich hohen Accuracy-Bereich liegen wie Vigenère. Bei der Enigma ist anzumerken, dass es sich genau genommen um eine vereinfachte Enigma handelt, da die oben angeführten, fixen Konfigurationen verwendet werden. Überraschend ist die sehr hohe Accuracy bei ADFGVX, bei der nahezu fehlerfrei von dem neuronalen Netzwerk entschlüsselt werden kann. Eine mögliche Erklärung dafür ist, dass der Schlüsselanteil an der Ausgabesequenz durch die zwei verwendeten Schlüssel vergleichsweise groß ist. Da diese vorgestellten Schlüssel sowohl beim Ciphertext als auch beim auszugebenden Plaintext gleich sind, muss das Netzwerk lediglich die Eingabe an diesen Time-Steps wieder ausgeben. Zusätzlich ist ein Ciphertext von ADFGVX durch die bipartite Substitution länger als der Plaintext. Da es sich bei dem verwendeten GRU-LSTM-BRNN um eine aligned Seq2Seq-Architektur handelt, muss dem Plaintext (also dem Label) ein Padding angehängt werden. Es handelt sich dabei um eine Folge von „-“-Zeichen, weshalb das Netzwerk eventuell lernt, dass ab dem ersten „-“ zum Ende der Ausgabesequenz nur mehr „-“ ausgegeben wird. Insgesamt ist es also möglich, dass die Accuracy bei Betrachtung des ausgegebenen Plaintexts allein um einige Prozentpunkte niedriger ausfällt. Versuche mit Probebeispielen wie dem Input *SECRETKEYABCGGDAADAFFDAAFAVGVXDAFGFGGGGG* und dem

Chiffre	Accuracy
Caesar	$99,99\%$
MASC	$98,77\%$
Vigenère	$97,40\%$
Autokey-Vigenère	$89,59\%$
Playfair	$73,47\%$
Hill	-
Enigma	$95,37\%$
Spaltentransposition	$98,90\%$
ADFGVX	$99,73\%$

Tabelle 5.7: Ergebnisse der Evaluierungen der trainierten Modelle.

richtig vorhergesagten Output *SECRETKEYABCEVALUATIONCASE* deuten jedoch darauf hin, dass das erstellte Modell sehr wohl in der Lage ist, richtig zu entschlüsseln. Ebenfalls auffällig ist die im Vergleich zu den übrigen Chiffren geringe Accuracy bei Autokey-Vigenère. Obwohl im Training das Modell immer die angestrebten $99,9\%$ erreicht, gehen bei der Evaluierung rund 10 Prozentpunkte verloren. Das Training wurde mehrmals mit unterschiedlichen Trainings- und Evaluierungsdatensätze wiederholt, allerdings tritt immer solch ein Verhalten auf. Es kann nicht erklärt werden, warum dies so ist, weshalb weitere Arbeiten zur genaueren Ursachenforschung notwendig sind. Weitere Untersuchungen sind außerdem bei der Playfair-Chiffre und der Hill-Chiffre notwendig. Beim Training mit der Playfair-Chiffre erreicht das GRU-LSTM-BRNN ca. 96% Accuracy, jedoch fällt die Leistung unerklärlicherweise bei der Evaluierung auf $73,47\%$ Accuracy ab. Auch bei dieser Chiffre wurden mehrmals verschiedene Datensätze zum Training verwendet, jedoch veränderte dies das Ergebnis nicht. Die Hill-Chiffre kann überhaupt nicht erlernt werden. Beim Training erreicht das Modell nicht mehr als 40% Accuracy, weshalb auf eine Evaluierung verzichtet wird. Insgesamt ergibt sich der nicht überprüfte Eindruck, dass ein GRU-LSTM-(B)RNN die Entschlüsselungsfunktion von verschiedenen Chiffren mit hoher Accuracy erlernen kann, jedoch polygraphisch polypartite Chiffren eine Ausnahme davon darstellen. Da aber bei der monographischen bipartiten ADFGVX-Chiffre eine, unter Einschränkungen, hohe Accuracy erreicht wird, ist es möglich, dass hauptsächlich polygraphische Chiffren schwer zu erlernen sind. Dies ist eine Hypothese, die in zukünftigen Untersuchungen genauer betrachtet werden muss.

Kapitel 6

Known-Plaintext-Angriffe mittels neuronaler Netze

Dieses Kapitel widmet sich der zweiten Aufgabenstellung dieser Arbeit – den neuronalen Known-Plaintext-Angriffen. Wieder ausgehend von [5] werden zunächst neuronale KPA auf Vigenère und Autokey-Vigenère durchgeführt. Anschließend werden Angriffe auf die zusätzlichen Chiffren Caesar, MASC, Playfair, Hill und die Spaltentransposition betrachtet. Da zum Zeitpunkt der Erstellung dieser Arbeit keine Literatur zu neuronalen KPA auf die Spaltentransposition bzw. allgemein auf Transpositionschiffren vorhanden ist, wird auf diese Chiffre ein besonderer Fokus gelegt.

6.1 Ausgangssituation

Ein Known-Plaintext-Angriff kann als Funktion KPA definiert werden, die für einen Ciphertext c und einen dazugehörigen Plaintext m den verwendeten Schlüssel k ausgibt. Die von dem neuronalen Netzwerk zu approximierende Zielfunktion ist somit

$$KPA(c, m) = k. \tag{6.1}$$

Wie schon zuvor handelt es sich bei m, c und k um Sequenzen von zufälligen Zeichen aus dem lateinischen Großbuchstabenalphabet A. Die in den Kapiteln 5.1 und 5.3 genannten Anmerkungen und Abweichungen bei gewissen Chiffren gelten auch für dieses Kapitel. Die verwendeten Chiffren bleiben grundsätzlich die gleichen, lediglich die Enigma und ADFGVX werden bei den neuronalen KPA aufgrund ihrer großen Komplexität nicht betrachtet.

Wie schon beim Erlernen der Entschlüsselungsfunktion handelt es sich bei einem neuronalen KPA um ein Problem des Supervised Learnings. Als Input wird wie in [5] eine Konkatenation von Plaintext und Ciphertext verwendet. Dabei wird die One-Hot-Repräsentation eines Ciphertextzeichens mit der Repräsentation des entsprechenden Plaintextzeichens konkateniert. Es entsteht somit für jeden Time-Step ein Vektor, dessen Dimensionalität der doppelten Anzahl an Zeichen des dem Plain-/Ciphertext zugrundeliegenden Alphabets entspricht. Als Label wird der entsprechende Schlüssel k verwendet. Da aber die Architektur aus Kapitel 5.2 weiterverwendet wird und es sich dabei um eine aligned Seq2Seq-Architektur handelt, muss der Schlüssel mit einem Padding

auf die Länge des Plain-/Ciphertexts erweitert werden. Dies erfolgt mit dem Zeichen „-“. Grundsätzlich wird erneut eine variable Schlüssellänge von $1-6$ Zeichen verwendet, die Länge des Plain- bzw. des Ciphertexts entspricht wieder 14 Zeichen. Durch diese Diskrepanz entspricht das Padding einem relativ großen Anteil der Outputsequenz. Da ein Modell einfach eine Sequenz von „-“-Zeichen ausgeben kann, bei der ein großer Teil tatsächlich mit dem entsprechenden Label übereinstimmt, gibt es bei der Accuracy eine gewisse Verzerrung. Bei einem Output, der nur aus „-“-Zeichen besteht, ist dementsprechend im Durchschnitt mit einer Accuracy von 75% zu rechnen. Erst Accuracy-Werte über diesen 75% deuten auf einen tatsächlichen Lernfortschritt hin. Unaligned Seq2Seq-Architekturen würden dieses Problem lösen, allerdings zeigen Tests, dass bei diesen auch bei den neuronalen KPA dasselbe Problem in Hinblick auf mangelnde Lernleistung auftritt wie in Kapitel 5.2. Folgendes Beispiel eines Trainingsdatenpunkts für die Vigenère-Chiffre verdeutlicht die Sequenzstruktur. Der Input besteht dabei aus einem zufälligen Plaintext und dem dazugehörigen Ciphertext, wobei beide auf Buchstabenebene konkateniert werden. Der Input des ersten Time-Steps ist somit ein Vektor, der das „F“ des Plaintexts und das „C“ des Ciphertexts darstellt. Der erwartete Output (also das Label) ist der verwendete Schlüssel inklusive dem notwendigen Padding.

Input (Plaintext):	F H U S R E T H Y P D M A Q
Input (Ciphertext):	C U Z H O R Y W V C I B X D
Label (Schlüssel):	- - - - - - - - - - X N F P

6.2 Neuronale KPA bei Substitutionschiffren

Für die Trainingsdatensätze werden 2 000 000 Datenpunkte verwendet, Evaluierungsdatensätze umfassen 1500 Datenpunkte. Bei der Generierung der Datensätze gelten für die entsprechenden Chiffren die Anmerkungen aus Kapitel 5.3. Für jede Chiffre wird ein eigenes Modell trainiert, das anschließend mit dem entsprechenden Evaluierungsdatensatz evaluiert wird. Als Metrik dient erneut die Accuracy. Für Vigenère, Autokey-Vigenère, Caesar und MASC wird ein GRU-LSTM-RNN mit 8192 Units verwendet, bei den polygraphisch polypartiten Chiffren Playfair und Hill ein GRU-LSTM-BRNN mit 4096 Units. Bei dem neuronalen KPA auf die Vigenère-Chiffre wird ein zusätzliches zweites Modell, das auf der Architektur von Greydanus aufbaut, trainiert. Dies ermöglicht einen Vergleich, ob das GRU-LSTM-RNN mit 8192 Units auch bei dem neuronalen KPA dem LSTM-RNN mit 512 in Hinblick auf Dateneffizienz überlegen ist. Tabelle 6.1 zeigt die Ergebnisse der Evaluierungen.

Es zeigt sich, dass das GRU-LSTM-RNN bei der Vigenère- und der Autokey-Vigenère-Chiffre wie schon in [5] eine hohe Accuracy erreicht. Bei Greydanus wird im Training eine Accuracy von 99% bei Vigenère und 95% bei Autokey-Vigenère erreicht [5]. In der Arbeit wird allerdings wie schon beim Erlernen der Entschlüsselungsfunktion keine Evaluierung mit einem spezifischen Evaluierungsdatensatz durchgeführt. Da Tabelle 6.1 jedoch Evaluierungsergebnisse zeigt und die entsprechenden Trainings bei 99% abgebrochen wurden, entspricht die Leistung des GRU-LSTM-RNN dem LSTM-RNN (bei Vigenère) bzw. übertrifft dieses sogar (bei Autokey-Vigenère). Bei Vigenère benötigt das GRU-LSTM-RNN 458 500 Trainingsdatenpunkte, während das LSTM-RNN von Greydanus mehr als 30 000 Epochen und somit mehr als $1,5$ Millionen Trainingsdatenpunkte

Chiffre	Accuracy
Caesar	$99,83\%$
MASC	$83,08\%$
Vigenère	$96,57\%$
Autokey-Vigenère	$93,55\%$
Playfair	$78,85\%$
Hill	-

Tabelle 6.1: Ergebnisse der Evaluierungen der für neuronale KPA auf Substitutionschiffren trainierten Modelle.

für das Erlernen des neuronalen KPA benötigt. Am Ende dieser 30 000 Epochen wird eine Accuracy von lediglich $\approx 84\%$ erreicht. Da das Training des LSTM-RNN sehr lange dauert und zu diesem Zeitpunkt bereits klar ist, dass das GRU-LSTM-RNN dateneffizienter ist, wird das Training nach den 30 000 Epochen nicht mehr fortgeführt.

Bei der Caesar-Chiffre wird mit $99,83\%$ ebenfalls eine sehr hohe Accuracy erreicht. Weil bei Vigenère mehrere Caesar-Chiffren zum Einsatz kommen (siehe Kapitel 3.2.2) ist dieses Ergebnis nach der hohen Accuracy bei Vigenère erwartbar. Bei der allgemeinen monoalphabetischen Substitution (MASC) ist das Netzwerk allerdings nicht in der Lage, den neuronalen KPA zu erlernen. Obwohl es sich um eine vergleichsweise simple Chiffre handelt, entsprechen die $83,08\%$ Accuracy großteils der zu Beginn des Kapitels genannten Verzerrung aufgrund des eingesetzten Paddings. Ausgehend von der durchschnittlich anzunehmenden Accuracy des Paddings (75%), entfallen bei der MASC $8,08\%$ auf Zeichen des Schlüssels. Bei Sequenzen aus 14 Zeichen entspricht ein korrektes Zeichen einer Accuracy von $\frac{1}{14} \cdot 100 = 7,14\%$, was bedeutet, dass ungefähr ein richtiges Zeichen eines Schlüssel bei der MASC ausgegeben wird. Es kann somit kein Schlüssel aus einem MASC-Ciphertext und einem entsprechenden Plaintext extrahiert werden. Auch bei Playfair kann der neuronale KPA nicht erlernt werden. Bei der Hill-Chiffre erreicht das Netzwerk wie schon beim Erlernen der Entschlüsselungsfunktion ein Plateau bei 40% Accuracy, bei dem es keinen Trainingsfortschritt mehr gibt.

Insgesamt lässt sich sagen, dass neuronale KPA mit dem in dieser Arbeit und in [5] verwendeten Ansatz nicht auf beliebige Substitutionschiffren durchführbar sind. Es besteht daher die Annahme, dass neuronale Netzwerke die Verschiebungen, die bei Chiffren wie Vigenère oder Caesar durchgeführt werden, besser erlernen können als andere grundlegende Verschlüsselungsoperationen. Substitutionen die über (teilweise) zufällige Ciphertextalphabete wie bei MASC oder Playfair durchgeführt werden, sind nach aktuellem Erkenntnisstand für neuronale Netzwerke schwierig zu erlernen, da sich diese Ciphertextalphabete bei jedem Trainingsdatenpunkt ändern. Bei Vigenère bzw. Autokey-Vigenère bleiben sie im Vergleich bei allen Trainingsdatenpunkten konstant, es ändert sich lediglich die Auswahl der verwendeten Alphabete. Ob dies tatsächlich generell gilt, muss in zukünftigen Untersuchungen evaluiert werden.

6.3 Neuronale KPA bei der Spaltentransposition

Bei dem neuronalen KPA auf die Spaltentransposition wird prinzipiell gleich vorgegangen wie bei den neuronalen KPA auf die Substitutionschiffren. Es wird ein GRU-LSTM-BRNN mit 4096 Units und den gleichen Hyperparametern verwendet. Änderungen ergeben sich jedoch bei dem Trainings- und dem Evaluierungsdatensatz. Da es für einen numerischen Schlüssel viele passende alphabetische gibt, ist es für ein neuronales Netzwerk bei dem KPA unmöglich zu bestimmen, welcher alphabetische Schlüssel tatsächlich verwendet wurde. Tatsächlich ist dies auch irrelevant, für die Ver- bzw. Entschlüsselung ist nur der numerische Schlüssel entscheidend. Im Gegensatz zu den Datensätzen für das Erlernen der Entschlüsselungsfunktion der Spaltentransposition werden bei den neuronalen KPA deshalb Datensätze mit numerischen Schlüssel verwendet. Wie in Kapitel 3.3.2 angeführt, wurden bei der historischen Anwendung der Spaltentransposition üblicherweise Schlüssellängen von 16 bis 23 Zeichen verwendet. Es wird deshalb versucht, ein Modell für diese Schlüssellängen zu trainieren, anstatt wie bei den Substitutionschiffren die variable Länge von $1-6$ Zeichen weiterzuverwenden. Problematisch bei solchen Schlüssellängen ist, dass die numerische Schlüsselsequenz bei Schlüssellängen größer 9 Zeichen Zahlen mit zwei Ziffern inkludiert. Zu einem einzelnen Time-Step kann jedoch nur eine Ziffer ausgegeben werden. Um dieses Problem zu umgehen, werden Zahlen ≥ 10 mit Buchstaben kodiert (a entspricht somit 10, $b = 11$ usw.). Zusätzlich wird die Länge eines Plain- bzw. Ciphertext erhöht, da dies das Finden des richtigen Schlüssels erleichtert.

In einem ersten Schritt wird ein Modell für eine Plaintextlänge von 20 Zeichen und einer Schlüssellänge von $2-6$ Zeichen trainiert. Dieses Modell erreicht eine Accuracy von $\approx 98\%$ bei der Evaluierung, weshalb die maximale Schlüssellänge auf 12 Zeichen verdoppelt und die Plaintextlänge auf 24 Zeichen angehoben wird. Das mit diesem Datensatz trainierte Modell erreicht in der Evaluierung nur mehr $\approx 54\%$ Accuracy, was darauf hindeutet, dass mit zunehmender Schlüssel- und Plaintextlänge die korrekte Vorhersage eines Schlüssels zunehmend schwieriger wird. Es ist also nicht möglich, mit dem GRU-LSTM-BRNN mit 4096 Units neuronale KPA auf realistische Schlüssellängen durchzuführen, wenn das Modell End-to-End trainiert wird. End-to-End bedeutet in diesem Kontext, dass dem Modell Plain- und Ciphertext als Input gegeben werden und der Output direkt dem gesuchten Schlüssel entspricht. Um dennoch das Ziel von $16-23$ Zeichen langen Schlüssel zu erreichen, wird ein alternatives Verfahren verwendet, bei dem mehrere GRU-LSTM-BRNN mit Pre- und Post-Processing-Schritten ergänzt werden. Abbildung 6.1 gibt einen Überblick über diesen Algorithmus, die einzelnen Verarbeitungsschritte werden im Folgenden im Detail beschrieben. Anzumerken ist bereits vorab, dass aufgrund der Art der Datenvorverarbeitung (*Data Preparation*) nur die komplette Spaltentransposition unterstützt wird.

6.3.1 Data Preparation

Wie oben beschrieben, nimmt die Accuracy bei zunehmender Länge der Plain- und Ciphertextinputs ab. Um dieses Problem zu begrenzen, wird ein Plain- bzw. ein entsprechender Ciphertext in Blöcke der Länge $l \in \mathbb{N}$ zerlegt. l entspricht der Länge des Schlüssel, der für die Erstellung des jeweiligen Ciphertexts genutzt wurde. Eine Zeile

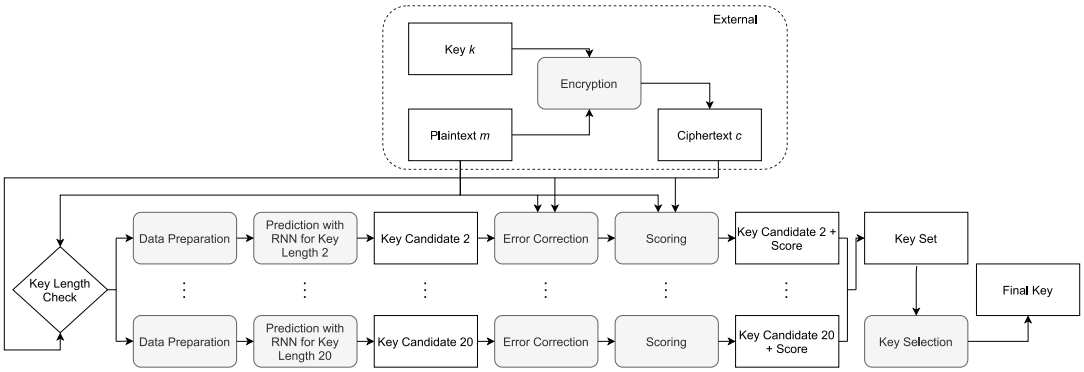

Abbildung 6.1: Flowchart des Algorithmus für neuronale KPA auf die komplette Spaltentransposition.

B	L	U	E		B	E	L	U
M	Y	S	E	→	M	E	Y	S
C	R	E	T		C	T	R	E
T	E	X	T		T	T	E	X

Abbildung 6.2: Die Abbildung zeigt dasselbe Beispiel wie in Abbildung 3.5, allerdings ist das Daten-Paar des ersten Blocks zur besseren Sichtbarkeit eingefärbt.

$x \in \mathbb{N}$ einer bereits neu angeordneten Verschlüsselungsmatrix entspricht einer Permutation des x-ten l Zeichen langen Substrings des Plaintexts m. Abbildung 6.2 zeigt ein solches Paar bestehend aus einem Plaintext-Substring (cyan) und der entsprechenden Zeile der neu geordneten Matrix (orange). Insgesamt ergeben sich aus einem Plaintext m und einem Ciphertext c bei einer Schlüssellänge l $\frac{len(m)}{l}$ Daten-Paare. Die Permutation jedes Paares basiert auf demselben Schlüssel, weshalb die Schlüsselvorhersage eines Modells anstatt auf der gesamten Konkatenation des Plain- und Ciphertexts auch auf einem einzelnen Daten-Paar durchgeführt werden kann. Zur Veranschaulichung wird wieder das Beispiel m = MYSECRETTEXT und c = MCTETTYRESEX betrachtet. Der verwendete Schlüssel ist k = BLUE ($[1, 3, 4, 2]$). Werden dieser Plaintext/Ciphertext nach der beschriebenen Methode in einzelne Daten-Paare unterteilt, entstehen die drei Paare [MYSE, MEYS], [CRET, CTRE] und [TEXT, TTEX]. Bevor ein Paar einem Modell übergeben werden kann, wird auf Buchstabenebene wieder ein One-Hot-Encoding durchgeführt. Abschließend werden der Plaintext-Substring und die entsprechende Matrix-Zeile wie in Kapitel 6.1 beschrieben konkateniert.

6.3.2 Prediction

Als Architektur wird wieder das GRU-LSTM-BRNN verwendet. In Evaluierungen zeigt sich, dass Overfitting bei dieser Aufgabenstellung und den verwendeten Datensätzen kein Problem darstellt, weshalb auf die Dropout-Regularisierung verzichtet wird. Es wird für jede unterstützte Schlüssellänge l ein spezifisches Modell trainiert. Insgesamt

Modell	Accuracy	Modell	Accuracy
2	92.83%	12	82.58%
3	96.07%	13	80.27%
4	90.40%	14	79.71%
5	91.40%	15	78.01%
6	90.03%	16	76.20%
7	89.55%	17	74.68%
8	87.67%	18	74.76%
9	86.40%	19	72.32%
10	84.60%	20	70.53%
11	82.88%		

Tabelle 6.2: Evaluierungsergebnisse der einzelnen Modelle.

werden Schlüssel von 2 bis 20 Zeichen unterstützt, weshalb 19 Modelle im Algorithmus eingesetzt werden. Bei zunehmender Schlüssellänge wird es für das Modell schwieriger, eine hohe Accuracy zu erreichen. Gleichzeitig sind Modelle bei sehr kurzen Schlüssel sehr schnell trainierbar, weshalb diese nicht so viele Units benötigen wie Modelle für längere Schlüssel. Insgesamt wird deshalb versucht, die kleinstmögliche Anzahl an Units zu verwenden. Dies hat neben schnelleren Trainings bei kleineren l den Vorteil, dass die gespeicherten Modelle weniger Speicherplatz[1] und auch weniger Arbeitsspeicher benötigen. Werden jedoch zu wenige Units gewählt, konvergiert das Modell nicht. Bei Evaluierungen hat sich gezeigt, dass sich in Abhängigkeit von l folgende Unit-Zahlen für ein speichereffizientes Training eignen:

- $2 \leq l < 4$: 512 Units
- $4 \leq l < 6$: 1024 Units
- $6 \leq l < 10$: 2048 Units
- $l \geq 10$: 4096 Units

Für das Training wird das Mini-Batch-Gradientenverfahren mit einer Batch-Größe von 100 verwendet. Als Optimizer dient Adam mit Standard-Hyperparametern und einer Lernrate von $\eta = 5 \cdot 10^{-3}$. Als Kostenfunktion wird wie schon bei allen vorherigen Kapiteln die Kreuzentropie gewählt. Trainiert wird jedes Modell mit einem für die Schlüssellänge spezifischen Trainingsdatensatz, der 2 Millionen Datenpunkte und Labels der jeweiligen Länge l enthält. Aus bereits genannten Gründen werden die Datenpunkte wieder zufallsgeneriert. Die Evaluierungsdatensätze enthalten jeweils 1500 Datenpunkte und Labels. Tabelle 6.2 zeigt die Evaluierungsergebnisse der einzelnen Modelle. Dabei wird ersichtlich, dass auch bei diesem Ansatz die Accuracy bei zunehmender Schlüssellänge abnimmt. Wie in Kapitel 6.3.6 gezeigt wird, gibt der gesamte Algorithmus mithilfe der der Prediction folgenden Post-Processing-Schritte dennoch mit hoher Wahrscheinlichkeit den richtigen Schlüssel aus.

Neben der höheren Accuracy bietet ein Ansatz mit mehreren spezifischen Modellen noch einen weiteren entscheidenden Vorteil: Der gesamte Algorithmus wird modula-

[1] 512 Units benötigen 96,6 MB, 1024 Units 382 MB, 2048 Units 1,52 GB und 4096 Units 6,06 GB auf der Festplatte, um die Gewichte zu speichern.

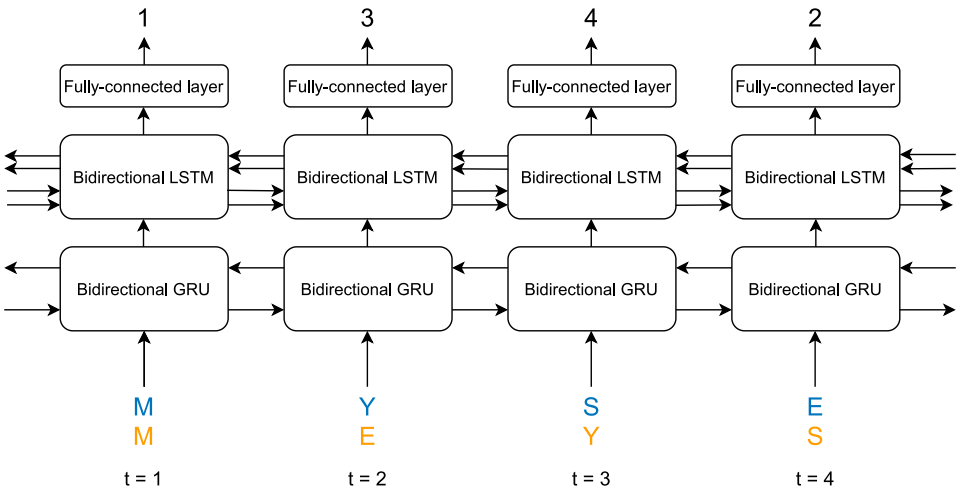

Abbildung 6.3: Architektur des GRU-LSTM-RNN inklusive der durch Pfeile dargestellten Informationsflüsse. Der Input und der Output in dem Diagramm entsprechen dem Beispiel aus Abbildung 6.2, was auch durch die Farben verdeutlicht wird.

rer, weshalb er leichter erweitert werden kann. Sollte er beispielsweise längere Schlüssel unterstützen, können zusätzliche Modelle trainiert und dem Algorithmus hinzugefügt werden. Bei einem universalen Modell, das alle Schlüssellängen abdeckt, muss hingegen das gesamte Modell komplett neu trainiert werden. Dafür werden zunehmend größere Datensätze, die alle Schlüssellängen gleichmäßig präsentieren, benötigt, wodurch auch die Trainingsdauer steigt. Bei dem modularen Ansatz ist die einzige Grenze in Hinblick auf die Anzahl der Modelle der verfügbare Festplatten- und Arbeitsspeicher. Bei den aktuell unterstützten Schlüssellängen benötigen die 19 Modelle ungefähr 74 GB Festplattenspeicher zur Speicherung der Parameter und auch die Arbeitsspeicher-Anforderungen entsprechen dieser Größenordnung. Es ist aber denkbar, dass mit spezialisierten Bibliotheken wie *TensorFlow Lite*[2] der Speicherbedarf reduziert werden kann. Der Nvidia DGX A100 Server bietet genügend Hardware-Ressourcen für den Betrieb aller Modelle, weshalb diese Optimierung nicht weiter in dieser Arbeit verfolgt wird.

Problematisch ist selbst auf dem Nvidia DGX A100 Server der Inferenzbetrieb aller Modelle auf GPU-Basis. Aufgrund der hohen Speicheranforderungen, der VRAM-Begrenzung von 40 GB bei den Nvidia A100 GPUs und der Speichermanagement-Strategie[3] von TensorFlow ist es aufwendig und ressourcenintensiv, alle 19 Modelle gleichzeitig auf GPUs zu betreiben. Der rechenintensivste Teil ist jedoch das Training, weshalb zur Inferenz die Modelle auch auf der CPU betrieben werden können, solange genügend Arbeitsspeicher zur Verfügung steht.

Insgesamt werden bei der Inferenz für einen Plaintext m (und den dazugehörigen Ciphertext c) $\frac{len(m)}{l}$ Schlüssel für eine Schlüssellänge l vorhergesagt. Diese müssen für

[2] https://www.tensorflow.org/lite

[3] Selbst beim Einsatz von mehreren GPUs ist standardmäßig nicht mehr VRAM als bei einer GPU verfügbar. Es ist prinzipiell denkbar, die Modelle mit nicht-standardmäßigen Verteilungsstrategien auf mehrere GPUs aufzuteilen, allerdings sind die dafür notwendigen Konfigurationen im Vergleich zu einem Betrieb auf der CPU viel komplizierter.

den restlichen Algorithmus auf einen einzelnen Key Candidate zusammengefasst werden. Das Beispiel aus Abbildung 6.2 kann beispielsweise zu den Vorhersagen [[1,3,4,2], [1,3,4,2], [2,3,4,1]] führen, die anschließend zu einem Key Candidate weiterverarbeitet werden.

6.3.3 Key Candidate

Um einen einzelnen Key Candidate zu erhalten, wird über alle Vorhersagen eine spaltenweise Mehrheitsentscheidung durchgeführt. Aus den oben genannten Beispielvorhersagen wird somit der Key Candidate [1,3,4,2]. Die Mehrheitsentscheidung hat den Vorteil, dass unsichere Vorhersagen, die auf uneindeutigen Inputs basieren, von anderen Vorhersagen korrigiert werden. Für das Daten-Paar [TEXT, TTEX] passen beispielsweise die beiden Vorhersagen [1,3,4,2] und [2,3,4,1]. Da aber die anderen beiden Paare eindeutig zu der Vorhersage [1,3,4,2] führen, wird auch der Key Candidate dieser entsprechen. Wie in Kapitel 6.3.6 ersichtlich wird, hat die Mehrheitsentscheidung einen großen Einfluss auf den gesamten Algorithmus. Je länger der vorhandene Plain-/Ciphertext ist, desto mehr Daten-Paare stehen zur Verfügung, was die Mehrheitsentscheidung aussagekräftiger macht und die Wahrscheinlichkeit für einen korrekt vorhergesagten Schlüssel erhöht. Eine Ausnahme tritt bei diesem Verarbeitungsschritt auf, wenn $len(m) = l$ gilt. In solch einem Fall gibt es nur ein Daten-Paar, wodurch auch keine Mehrheitsentscheidung durchgeführt werden kann. Dementsprechend wird dieser Schritt übersprungen.

6.3.4 Error Correction und Scoring

Durch die Mehrheitsentscheidung und theoretisch auch durch die Vorhersagen des RNN kann es zu ungültigen Key Candidates kommen. Bei einem solchen ungültigen Schlüssel treten einzelne Zahlen mehrmals auf, wodurch andere in der Schlüsselsequenz fehlen. Dies hat zur Folge, dass die Transposition nicht eindeutig durchführbar ist. Um solche Fehler bis zu einem gewissen Grad ausbessern zu können, wird ein Error-Correction-Mechanismus (ECM) eingeführt. Dieser funktioniert folgendermaßen:

1. Es wird eine interne Liste mit allen fehlenden Zahlen und allen mehrfach vorkommenden Zahlen angelegt.
2. Die Elemente dieser Liste werden auf alle Positionen verteilt, an denen eine Zahl mehrfach vorkommt. Das Ergebnis dieser Operation ist ein gültiger Schlüssel.
3. Es wird ein Score berechnet, der die Qualität des so generierten Schlüssels bewertet. Dazu wird eine neue Metrik, der sogenannten *Ciphertext Similarity Score (CSS)*, eingeführt. Die Berechnung des CSS wird weiter unten im Detail erklärt. Ergibt sich ein CSS von 1.0, wurde der richtige Schlüssel gefunden und der ECM war erfolgreich. Ist der CSS niedriger, wird mit einer Permutation der Liste aus Schritt 1 bei Schritt 2 fortgefahren.

Wird kein Schlüssel mit einem CSS von 1.0 gefunden, gibt der ECM den Schlüssel mit dem höchsten Score zurück.

Insgesamt können über diesen Mechanismus bis zu ε Fehler (fehlende Zahlen) korrigiert werden. Besitzt ein Key Candidate mehr Fehler, wird er als falsche Vorhersage gewertet. ε bestimmt, wie viele Permutationen bei dem Brute-Force-Ansatz des ECM getestet werden, weshalb die Wahl des Parameters einen großen Einfluss auf die Lauf-

zeit des Algorithmus besitzt. Im schlimmsten Fall ergibt sich eine Laufzeit von $O(\varepsilon!)$, wodurch die Vorhersage eines einzelnen Schlüssel sehr lange dauern kann, wenn ε groß ist. Andererseits erhöht ein größeres ε die Chance, einen Schlüssel mit einem Score von 1.0 vorherzusagen. In Tests zeigt sich, dass $\varepsilon = 5$ einen guten Kompromiss zwischen Geschwindigkeit und Vorhersageleistung darstellt.

Der Ciphertext Similarity Score (CSS) wird nicht nur bei dem ECM eingesetzt, sondern auch am Ende des Algorithmus bei der Auswahl des finalen Schlüssels. Die Grundidee des CSS ist, dass mit einem Schlüsselvorschlag $\hat{k}$ der Plaintext m verschlüsselt wird (Gleichung 6.2) und der daraus folgende Ciphertext $\hat{c}$ mit dem tatsächlichen Ciphertext c stellenweise verglichen wird (Gleichungen 6.3 und 6.4). Je mehr Stellen übereinstimmen, desto näher muss $\hat{k}$ an dem tatsächlich verwendeten Schlüssel k liegen. Ein CSS von 1.0 bedeutet, dass $\hat{c} = c$ gilt, wodurch ein korrekter Schlüssel gefunden wurde. Für Kryptoanalysten kann jedoch auch ein Schlüssel hilfreich sein, der einen geringeren Score als 1.0 besitzt, da ein solcher möglicherweise mit wenig manueller Nachbearbeitung zu dem korrekten Schlüssel führen kann. Formal lässt sich der CSS mit folgenden Gleichungen berechnen:

$$\hat{c} = Enc(m, \hat{k}) \tag{6.2}$$

$$match(c_i, \hat{c}_i) = \begin{cases} 1 & \text{if } c_i = \hat{c}_i \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

$$CSS(c, \hat{c}) = \frac{\sum_{i=1}^{len(c)} match(c_i, \hat{c}_i)}{len(c)}. \tag{6.4}$$

Das Konzept des CSS ist mit der Hamming-Distanz [75] verwandt, die als

$$Hamming(c, \hat{c}) = \sum_{i=1}^{len(c)} (1 - match(c_i, \hat{c}_i)). \tag{6.5}$$

definiert werden kann. Je mehr Zeichen bei $\hat{c}$ von c abweichen, desto größer wird die Hamming-Distanz. Eine Hamming-Distanz von 0 entspricht somit einem CSS von 1.0. Der Vorteil des CSS ist, dass er besser interpretierbar ist, da er als Prozentzahl gedeutet werden kann. Nichtsdestotrotz kann der CSS mit folgender Formel in die Hamming-Distanz umgerechnet werden:

$$Hamming(c, \hat{c}) = len(c) - len(c) \cdot CSS. \tag{6.6}$$

6.3.5 Umgang mit unbekannten Schlüssellängen

Bei dem ersten Schritt des Algorithmus – der Data Preparation – wird die Schlüssellänge l benötigt, um den Plain- und Ciphertext in Blöcke aufzuteilen. Diese ist jedoch bei einem KPA unbekannt. Versuche mit einem zusätzlichen neuronalen Netzwerk[4], das in einem ersten Schritt die Schlüssellänge vorhersagt, zeigen, dass ein solcher zweistufiger Ansatz nicht funktioniert. Die Idee dieses Ansatzes ist, dass mit dem ersten neuronalen

[4] Gleiche GRU-LSTM-BRNN-Architektur und Hyperparameter wie das Hauptmodell

Netzwerk die Schlüssellänge vorhergesagt wird, welche anschließend für die Data Preparation und das Laden des entsprechenden KPA-Modells verwendet wird. Bei einem Datensatz mit einer maximalen Schlüssellänge von 12 Zeichen werden lediglich $\approx 60\%$ Accuracy erreicht, was unzureichend ist, da der gesamte restliche Algorithmus von dieser ersten Vorhersage abhängig ist. Bei einer falsch vorhergesagten Schlüssellänge ist er nicht mehr in der Lage, einen richtigen Schlüssel aus einem Plain- und dem dazugehörigen Ciphertext zu rekonstruieren. Bei den $\approx 60\%$ Accuracy der ersten Stufe funktioniert der Algorithmus somit unabhängig von den folgenden Schritten in etwas weniger als der Hälfte der Fälle nicht. Aus diesem Grund werden alle möglichen Schlüssellängen von dem Algorithmus mit jeweils einer Algorithmusiteration getestet. Da momentan lediglich die komplette Spaltentransposition unterstützt wird, müssen nur l getestet werden, die ein echter Teiler von $len(m)$ sind. Dies beschleunigt die Algorithmusausführung.

Zusammengefasst lässt sich sagen, dass eine Konkatenation von einem Plaintext m und einem Ciphertext c als Input für den vorgeschlagenen Algorithmus dient. In Abhängigkeit von $len(m)$ werden die Schritte der Kapitel 6.3.1 bis 6.3.4 für alle möglichen l durchgeführt. Der Output jeder dieser Algorithmusiterationen ist jeweils ein Key Candidate $\hat{k}_l$ mit einem dazugehörigen CSS. Aus diesen Kandidaten wird der Schlüssel mit dem höchsten CSS als finaler Output des Algorithmus ausgewählt.

6.3.6 Evaluierung des Algorithmus

Für jede unterstützte Schlüssellänge (2 bis 20 Zeichen) werden mit dem gesamten Algorithmus fünf Evaluierungen mit unterschiedlichen Plain-/Ciphertextlängen durchgeführt. In jeder Evaluierungsiteration wird die Plain-/Ciphertextlänge um die Schlüssellänge l erhöht, was in Plain-/Ciphertextlängen von l bis $5 \cdot l$ resultiert. Jede Längenerhöhung um l Zeichen stellt ein zusätzliches Daten-Paar für die Mehrheitsentscheidung dar, wodurch effektiv die Leistung des Algorithmus für Inputs von 1 bis 5 Daten-Paare evaluiert wird. Jeder Evaluierungsdatensatz besteht aus 100 Datenpunkten mit zufallsgenerierten m und k, die entsprechenden c werden wieder durch die Verschlüsselung der m mit den jeweiligen k erstellt. Der Vorteil beim Einsatz von für jede Schlüssellänge individuellen Datensätzen liegt in dem detaillierten Einblick in die Performance des Algorithmus. Bei einem einzigen großen Datensatz, der alle Schlüssellängen kombiniert, wird hingegen nicht direkt ersichtlich, welchen Einfluss die Schlüssellänge auf die Leistung des Algorithmus hat.

Als Metrik dient bei den Evaluierungen wieder der CSS. Wie oben bereits angesprochen, kann für Kryptoanalysten jedoch auch ein Schlüssel mit einem CSS < 1.0 nützlich sein. Aus diesem Grund werden drei Qualitätsklassen eingeführt:

- Perfect: Score $= 1.0$
- Good: Score ≥ 0.9
- Acceptable: Score ≥ 0.8

Tabelle 6.3 zeigt die Ergebnisse der Evaluierung. Es ist klar ersichtlich, dass bei zunehmender Schlüssellänge zwei Daten-Paare (ein Daten-Paar entspricht einer „row“) zu wenig werden, um zuverlässig einen korrekten („perfect“) Schlüssel vorherzusagen. Dies ist jedoch wenig überraschend, da bei lediglich zwei Daten-Paaren keine wirkliche Mehrheitsentscheidung durchgeführt werden kann. Wird dem Algorithmus ein einziges Daten-Paar mehr übergeben, steigt die Anzahl der vorhergesagten Schlüssel mit

Keylength	1 row			2 rows			3 rows			4 rows			5 rows		
	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.
2	96%	96%	96%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
3	70%	70%	70%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
4	94%	94%	94%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
5	98%	98%	98%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
6	100%	100%	100%	100%	100%	100%	100%	100%	100%	99%	99%	100%	100%	100%	100%
7	100%	100%	100%	100%	100%	100%	99%	100%	100%	99%	99%	100%	100%	100%	100%
8	100%	100%	100%	99%	99%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
9	100%	100%	100%	99%	99%	100%	97%	99%	100%	97%	97%	100%	100%	100%	100%
10	100%	100%	100%	95%	98%	100%	96%	97%	100%	100%	100%	100%	100%	100%	100%
11	100%	100%	100%	65%	95%	98%	94%	96%	100%	99%	99%	100%	99%	99%	100%
12	100%	100%	100%	62%	93%	99%	91%	94%	100%	99%	100%	100%	99%	100%	100%
13	100%	100%	100%	68%	93%	99%	88%	91%	100%	97%	100%	100%	100%	100%	100%
14	99%	99%	100%	44%	86%	99%	90%	99%	100%	97%	99%	100%	100%	100%	100%
15	100%	100%	100%	51%	90%	99%	89%	100%	100%	96%	100%	100%	100%	100%	100%
16	100%	100%	100%	40%	87%	98%	89%	99%	100%	94%	99%	100%	100%	100%	100%
17	96%	96%	96%	52%	87%	97%	77%	99%	100%	92%	100%	100%	98%	100%	100%
18	98%	98%	98%	31%	74%	97%	74%	99%	100%	89%	98%	100%	97%	100%	100%
19	95%	95%	95%	29%	61%	95%	75%	95%	99%	85%	100%	100%	97%	100%	100%
20	83%	87%	87%	23%	71%	85%	75%	95%	98%	90%	99%	100%	96%	100%	100%

Tabelle 6.3: Ergebnisse der Evaluierungen mittels Ciphertext Similarity Score (CSS).

Keylength	1 row			2 rows			3 rows			4 rows			5 rows		
	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.	Perfect	Good	Accept.
2	95%	95%	95%	97%	97%	97%	100%	100%	100%	98%	98%	98%	100%	100%	100%
3	65%	65%	65%	63%	63%	63%	69%	69%	69%	66%	66%	66%	66%	66%	66%
4	79%	79%	79%	95%	95%	95%	99%	99%	99%	100%	100%	100%	100%	100%	100%
5	82%	82%	82%	90%	90%	90%	100%	100%	100%	100%	100%	100%	100%	100%	100%
6	67%	67%	67%	85%	85%	85%	99%	99%	99%	99%	99%	99%	100%	100%	100%
7	59%	59%	59%	85%	85%	85%	99%	99%	99%	99%	99%	99%	100%	100%	100%
8	56%	56%	56%	82%	82%	82%	100%	100%	100%	100%	100%	100%	100%	100%	100%
9	57%	57%	57%	69%	69%	69%	97%	97%	97%	97%	97%	97%	100%	100%	100%
10	41%	41%	83%	70%	70%	78%	96%	96%	99%	100%	100%	100%	100%	100%	100%
11	33%	33%	71%	64%	64%	95%	94%	94%	100%	99%	99%	100%	99%	99%	100%
12	30%	30%	75%	58%	58%	93%	91%	91%	100%	99%	99%	100%	99%	99%	100%
13	20%	20%	63%	62%	62%	92%	88%	88%	99%	97%	97%	100%	100%	100%	100%
14	22%	22%	51%	42%	42%	83%	90%	90%	99%	97%	97%	100%	100%	100%	100%
15	8%	8%	41%	48%	48%	88%	89%	89%	100%	96%	96%	100%	100%	100%	100%
16	10%	10%	35%	37%	37%	83%	88%	88%	99%	94%	94%	99%	100%	100%	100%
17	7%	7%	26%	44%	44%	84%	77%	77%	99%	92%	92%	100%	98%	98%	100%
18	6%	6%	30%	26%	26%	72%	74%	74%	99%	89%	89%	98%	97%	97%	100%
19	1%	1%	21%	27%	27%	57%	75%	75%	95%	85%	85%	100%	97%	97%	100%
20	2%	11%	28%	20%	48%	74%	75%	95%	98%	90%	99%	100%	96%	100%	100%

Tabelle 6.4: Ergebnisse der Evaluierungen mittels Key Similarity Score (KSS).

CSS 1.0 signifikant. Generell zeigt sich, dass der Algorithmus mit zunehmender Plain-/Ciphertextlänge besser funktioniert. Werden beispielsweise fünf Daten-Paare zur Verfügung gestellt, liegt die Wahrscheinlichkeit für eine Vorhersage mit CSS 1.0 bei mindestens 96%, selbst bei Schlüsseln mit einer Länge von 20 Zeichen.

Unerwartet sind jedoch die Ergebnisse, wenn lediglich ein Daten-Paar als Input dient (Spalte „1 row“). Teilweise (v.a. bei kurzen Schlüsseln) tritt dabei eine höhere Anzahl an perfekten Vorhersagen auf als bei der vierfachen Plain-/Ciphertextlänge. Um die Hypothese zu prüfen, dass bei solch kurzen Inputs mehrere Schlüssel passen, werden die Evaluierungen erneut mit einer anderen Metrik durchgeführt. Diese Metrik namens Key Similarity Score (KSS) ist ähnlich wie der CSS, der einzige Unterschied ist, dass anstatt

c und $\hat{c}$ der tatsächliche Schlüssel k und der vorhergesagte Schlüssel $\hat{k}$ stellenweise verglichen werden. Der KSS vergleicht somit, wie ähnlich $\hat{k}$ dem Schlüssel, der tatsächlich für die Erstellung des Ciphertexts c verwendet wurde, ist. Um den Unterschied zwischen CSS und KSS näher zu beschreiben, wird als Beispiel m = HELL und c = EHLL betrachtet. Der originale Schlüssel ist k = [2,1,4,3]. Wenn der Algorithmus den Schlüssel $\hat{k}$ = [2,1,3,4] vorhersagt, ist der CSS 1.0, obwohl $\hat{k}$ nicht k entspricht. Der KSS ist hingegen 0.5, da nur die Hälfte der Zahlen gleich ist. Wird somit die gleiche Evaluierung nochmals durchgeführt und die Ergebnisse der KSS-Evaluierung sind niedriger als bei der CSS-Evaluierung, bestätigt dies, dass mehrere passende Schlüssel auftreten. Aus Effizienzgründen werden die drei Kategorien „perfect“, „good“ und „acceptable“ beibehalten, zur Beurteilung der Hypothese ist jedoch hauptsächlich die „perfect“-Kategorie relevant. Tabelle 6.4 zeigt die Ergebnisse der KSS-Evaluierung. Bei einem Vergleich der Tabellen 6.3 und 6.4 fällt auf, dass tatsächlich bei zunehmender Schlüssellänge lediglich ein kleiner werdender Anteil der perfekten CSS-Vorhersagen den tatsächlich verwendeten Schlüsseln entspricht. Ab drei Input-Paaren (also ab Spalte „3 rows“) tritt dieser Effekt kaum mehr auf. Eine Ausnahme davon ist jedoch die Schlüssellänge 3, bei der auch bei fünf Input-Paaren nur in 66% der Fälle der tatsächliche Schlüssel ausgegeben wird. Warum genau diese Schlüssellänge von dem allgemeinen Verhalten der anderen Längen abweicht, wird in der vorliegenden Arbeit nicht weiter erklärt.

Kapitel 7

Fazit und Ausblick

In diesem abschließenden Kapitel werden zunächst die Ergebnisse dieser Arbeit in einem Fazit zusammengefasst. Anschließend gibt ein Ausblick eine Übersicht über noch notwendige zukünftige Verbesserungen des vorgestellten Algorithmus für neuronale KPA auf die Spaltentransposition.

7.1 Fazit

In dieser Arbeit wird die Arbeit von Greydanus ([5]) verbessert und auf zusätzliche Chiffren erweitert. Mit einer empirischen Evaluierung konnte eine Architektur gefunden werden, die beim Erlernen der Entschlüsselungsfunktion und bei neuronalen KPA weniger Trainingsdaten benötigt als die Architektur in [5], gleichzeitig aber eine vergleichbare Accuracy erreicht. Diese Architektur, ein GRU-LSTM-RNN mit 8192 Units, benötigt beim Erlernen der Vigenère-Entschlüsselungsfunktion für eine Accuracy im Training von $99,9\%$ 1 089 425 Datenpunkte weniger als das LSTM-RNN mit 512 Units von Greydanus. Auch bei einem neuronalen KPA auf die Vigenère-Chiffre benötigt dieses GRU-LSTM-RNN über eine Million Trainingsdatenpunkte weniger.

Es konnte gezeigt werden, dass das Lernen der Entschlüsselungsfunktion auch bei anderen Chiffren möglich ist. Zusätzlich zu den in [5] verwendeten Chiffren Vigenère, Autokey-Vigenère und Enigma konnten die Entschlüsselungsfunktionen der Caesar-Chiffre, der MASC, der Spaltentransposition und der vergleichsweise starken Chiffre ADFGVX mit ähnlicher Accuracy ($> 95\%$) erlernt werden. Lediglich die Playfair- und die Hill-Chiffre konnten nicht erlernt werden. Beide Chiffren sind polygraphisch und polypartit. Da die monographische bipartite ADFGVX-Chiffre allerdings erlernt werden konnte, liegt der Grund des mangelhaften Trainingserfolgs bei Playfair und Hill möglicherweise in der Polygraphie der Chiffren. Diese Hypothese kann in zukünftigen Untersuchungen weiter analysiert werden, indem weitere polygraphische und monographische Chiffren betrachtet werden. Können die Entschlüsselungsfunktionen aller polygraphischen Chiffren im Gegensatz zu den monographischen nicht erlernt werden, ist das ein Indikator, dass tatsächlich die Polygraphie einen Trainingsfortschritt erschwert bzw. verhindert. Insgesamt konnte die erste Forschungsfrage „Kann mithilfe einer verbesserten Architektur der in [5] präsentierte Ansatz zum Lernen der Entschlüsselungsfunktionen verschiedener Chiffren weiter verbessert werden?“ mit der GRU-LSTM-RNN-

Architektur und der folgenden Evaluierung der zusätzlichen Chiffren beantwortet werden.

Im Gegensatz zu dem Erlernen der Entschlüsselungsfunktion konnten die neuronalen KPA nicht mit der bestehenden Architektur einfach auf weitere Chiffren angewandt werden. Neben den in [5] verwendeten Chiffren Vigenère und Autokey-Vigenère konnte nur bei der sehr simplen Caesar-Chiffre erfolgreich ein GRU-LSTM-RNN für neuronale KPA (End-to-End) trainiert werden. Bei MASC, Playfair, Hill war dies nicht möglich. Für die bei den Substitutionschiffren verwendeten Schlüssellängen von $1-6$ Zeichen war ein neuronaler KPA mit einem End-to-End-Modell bei der Spaltentransposition grundsätzlich möglich. Da auf diese Chiffre aufgrund der mangelnden Literatur in Bezug auf neuronale KPA auf Transpositionschiffren ein besonderer Fokus gelegt wurde, wurde jedoch versucht, realistische Schlüssellängen von $16-23$ Zeichen zu erreichen. Dies war End-to-End mit einem GRU-LSTM-BRNN mit 4092 Units nicht möglich, da mit zunehmender Schlüssellänge und zunehmender Plain-/Ciphertextlänge die Accuracy des RNN rasch signifikant abfiel. Wurde bei einer Plain-/Ciphertextlänge von 20 Zeichen und einer Schlüssellänge von $2-6$ Zeichen noch eine Accuracy von $\approx 98\%$ erreicht, fiel diese bereits bei Plain-/Ciphertextlängen von 24 Zeichen und einer Schlüssellänge von $2-12$ Zeichen auf $\approx 54\%$ ab. Es wäre interessant, in zukünftigen Arbeiten den Einfluss der Länge des Schlüssels und des Plain-/Ciphertexts auch bei den Substitutionschiffren zu analysieren. Möglicherweise fällt bei den anderen Chiffren die Modellleistung ebenfalls stark ab. Greydanus hat in [5] erste derartige Versuche durchgeführt. Diese zeigen, dass zumindest bei Vigenère und Autokey-Vigenère Modelle, die mit 20 Zeichen langen Nachrichten trainiert wurden, auf Nachrichten mit über 100 Zeichen generalisieren. Ein Modell für die Enigma generalisiert hingegen kaum über die ursprünglichen 20 Zeichen hinaus, was ein Hinweis ist, dass die Komplexität der Zielchiffre einen entscheidenden Einfluss auf die Generalisierungsfähigkeit der Modelle zu haben scheint. Diese Evaluierung wurde jedoch für das Erlernen der Entschlüsselungsfunktion durchgeführt, weshalb es fraglich ist, wie sehr sich diese Beobachtung auf Modelle für neuronale KPA übertragen lassen.

Neben der geringen Accuracy bei den immer noch zu kurzen Schlüsseln ist bei einem End-to-End-Ansatz die fixe Plain-/Ciphertextlänge bei der Spaltentransposition problematisch. Wie oben beschrieben, besitzen Modelle abhängig von der Chiffre zwar einen gewissen Toleranzbereich, in der Praxis können Texte jedoch Längen weit über diesen Bereich hinaus aufweisen. Aufgrund dieser zwei Schwächen des End-to-End-Ansatzes, wird in dieser Arbeit ein neuer Algorithmus für neuronale KPA auf die Spaltentransposition vorgestellt. In diesem Algorithmus werden 19 GRU-LSTM-BRNN (pro Schlüssellänge ein spezifisches Modell) mit mehreren Pre- und Post-Processing-Schritten kombiniert. Durch eine Zerlegung des Plain-/Ciphertexts in einzelne Blöcke, deren Längen der trainierten Schlüssellänge des verwendeten Modells entsprechen, wird das Problem der limitierten Textlängen gelöst und es können beliebig lange Inputs verarbeitet werden. Mittels einer eingeführten Error Correction werden etwaige Vorhersagefehler der RNN bis zu einem gewissen Grad korrigiert, was die Gesamtleistung des Algorithmus erhöht. Bei unterstützten Schlüssellängen von $2-20$ Zeichen gibt der Algorithmus laut durchgeführten Evaluierungen in mindestens 96% der Fälle den richtigen Schlüssel aus, vorausgesetzt die Plain/-Ciphertextlänge entspricht der fünffachen Schlüssellänge l. Die Mindestlänge, die für die im Algorithmus verwendete Mehrheitsentscheidung notwendig

ist, ist $3 \cdot l$. Diese Mindestlänge führt in mindestens 74% der Fälle zu korrekten Ausgaben. Der Algorithmus kann auch mit kürzeren Inputs arbeiten, allerdings kann die Mehrheitsentscheidung in solchen Fällen nicht erfolgen, was die Leistung des Algorithmus signifikant verringert. Bei sehr kurzen Inputs, die l entsprechen, ist die Leistung des Algorithmus zwar hoch, allerdings sind bei solchen Inputs die verwendeten Schlüssel nicht eindeutig, was mit einer entsprechenden Evaluierung gezeigt werden konnte.

Durch die Evaluierung von neuronalen KPA auf die Caesar-Chiffre, die MASC, die Playfair- und die Hill-Chiffre und die Erstellung des Algorithmus für neuronale KPA auf die (komplette) Spaltentransposition konnte auch die zweite Forschungsfrage („Kann mittels neuronaler Netze ein Known-Plaintext-Angriff auf weitere Chiffren durchgeführt werden?“) erfolgreich beantwortet werden.

7.2 Ausblick

Obwohl der vorgeschlagene Algorithmus nach bestem Wissen einem neuen Stand der Technik für neuronale KPA auf die Spaltentransposition entspricht, besitzt er dennoch Einschränkungen, die in weiteren Arbeiten verbessert werden sollten:

- Momentan wird nur die komplette Spaltentransposition unterstützt. Dies liegt an dem Data-Preparation-Schritt (Kapitel 6.3.1). Bei der inkompletten Spaltentransposition können die für einen Block notwendigen Buchstaben nicht einfach extrahiert werden, da die Zeilenanzahl nicht als Schrittweite über den Ciphertext verwendet werden kann. Bei dem Beispiel aus Abbildung 6.2 wird die orangene Zeile der Verschlüsselungsmatrix erstellt, indem ausgehend von einem Index $i = 0$ alle Buchstaben des Ciphertexts mit einem Index $i = j \cdot c$ extrahiert werden. $c \in \mathbb{N}$ entspricht dabei der Anzahl an Zeilen, für j gilt in Abbildung 6.2 $j \in \{0, 1, 2, 3\}$. Bei der inkompletten Spaltentransposition kann dieser Ansatz nicht direkt durchgeführt werden, da nicht bekannt ist, wie lange die letzte Zeile ist. Ist sie kürzer als die übrigen (also liegt eine tatsächliche inkomplette Spaltentransposition vor), führt der beschriebene Ansatz nicht zu den korrekten Zeilen der Verschlüsselungsmatrix. Möglicherweise kann der Algorithmus für die inkomplette Spaltentransposition erweitert werden, indem die letzte, inkomplette Zeile der Matrix bei der Data Preparation weggelassen wird. Für eine praktische Einsetzbarkeit des Algorithmus ist es von großer Wichtigkeit, dass eine Anpassung an die inkomplette Spaltentransposition in zukünftigen Arbeiten erfolgt. Die komplette Spaltentransposition ist lediglich ein Sonderfall der allgemeinen (inkompletten) Spaltentransposition, weshalb solch eine Erweiterung die praktischen Einsatzmöglichkeiten des Algorithmus signifikant erhöhen.
- Die maximale, momentan unterstützte Schlüssellänge entspricht 20 Zeichen. Dies entspricht zwar schon einem Teil der angestrebten Länge von $16 - 23$ Zeichen, jedoch wird das Ziel noch nicht komplett erreicht. Es wurde ein Modell für die Schlüssellänge 23 trainiert, allerdings wurde in diesem Training lediglich eine Accuracy von $\approx 50\%$ erreicht. In zukünftigen Arbeiten kann deshalb versucht werden, die maximale Schlüssellänge anzuheben. Ein großer Vorteil der Algorithmus-Architektur ist der modulare Aufbau, wodurch Modelle, die längere Schlüssel unterstützen, einfach zusätzlich geladen und integriert werden können. Die verwen-

deten Modell müssen nicht einmal die gleiche Architektur besitzen, sondern lediglich mit der tf.keras.Model-API[1] kompatibel sein. Bei ausreichenden Hardware-Ressourcen ist der einzige die Schlüssellänge limitierende Faktor somit die Leistungsfähigkeit der momentan verfügbaren Architekturen. Im Bereich des Deep-Learnings gab es die letzten Jahre zahlreiche Entwicklungen, die leistungsfähigere Architekturen hervorbrachten, weshalb davon auszugehen ist, dass in ein paar Jahren auch bessere Architekturen für den neuronalen KPA verfügbar sein werden. Der einzige Nachteil, der beim Laden von weiteren Modellen entsteht, ist der zunehmend große Arbeitsspeicherverbrauch. Eine Alternative ist deshalb die Weiterentwicklung des Algorithmus mit einem Modell zur Schlüssellängenerkennung. Wie beschrieben, liefern bisherige Versuche eine zu geringe Erkennungsleistung. Kann dieses vorgelagerte Modell jedoch so weit verbessert werden, dass eine fast perfekte Erkennungsleistung erreicht wird, so können die benötigten Modelle zur Schlüsselerkennung dynamisch geladen werden. Ein weiteres Problem bei solch einem Ansatz ist aber die lange Ladedauer von großen Modellen bei TensorFlow. Damit dieser Ansatz in akzeptabler Zeit möglich ist, muss deshalb eine Optimierung des Modell-Ladeprozesses erreicht werden.

- In dieser Arbeit wurde aus genannten Gründen exklusiv mit zufallsgenerierten Sequenzen gearbeitet. Natürliche Sprache besitzt im Gegenteil zu der diesen Sequenzen zugrundeliegenden Gleichverteilung eine stark verzerrte Häufigkeitsverteilung der einzelnen Zeichen. In zukünftigen Arbeiten wäre es deshalb interessant zu evaluieren, wie sich die Leistung des Algorithmus verändert, wenn die Modelle mit natürlicher Sprache trainiert werden. Da bei solchen Trainings auch Sprachmodelle erlernt werden können, müsste die Leistung des Algorithmus besser werden. Möglicherweise könnte auch so eine Erhöhung der maximalen Schlüssellänge erreicht werden. Ein Vorteil, den das Training mit zufälligen Sequenzen hingegen mit sich bringt, ist die Unterstützung von mehreren Sprachen. Dadurch dass kein Sprachmodell erlernt wird, funktionieren die Modelle bei jeder Sprache, die aus den Zeichen des den Zufallssequenzen zugrundeliegenden Alphabets besteht.

Zusammenfassend lässt sich sagen, dass mit dem in dieser Arbeit vorgestellten Algorithmus ein erster Schritt in Richtung praktisch relevanter neuronaler KPA auf die Spaltentransposition getan wurde. Aus den oben genannten Gründen ist der Algorithmus in seiner derzeitigen Form nur sehr limitiert in der Praxis einsetzbar, jedoch bildet er einen Ausgangspunkt für weitere Forschung. Generell ist ein Known-Plaintext-Angriff auf die Spaltentransposition praktisch nur in speziellen Szenarien wirklich nützlich. Ein solches Szenario wäre beispielsweise, wenn zwei Ciphertexte vorliegen, bei denen der Verdacht besteht, dass derselbe Schlüssel zur Verschlüsselung verwendet wurde. Ist zu einem Ciphertext der zugehörige Plaintext bekannt, kann ein KPA zu dem Schlüssel und somit zu einer Entschlüsselung auch des zweiten Ciphertexts führen. In der Praxis sind die mächtigeren Ciphertext-Only-Angriffe, bei denen nur der Ciphertext zur Entschlüsselung benötigt wird, für Kryptoanalysten relevanter. Die Leistung von Ansätzen wie das in [62] vorgestellte zweistufige Hill Climbing, das Schlüssel bis zu einer Länge von 1000 Zeichen rekonstruieren kann, erreichen neuronale KPA derzeit noch bei Weitem nicht. Bei der derzeitigen rasanten Weiterentwicklung des Deep Learnings

[1] https://www.tensorflow.org/api_docs/python/tf/keras/Model

und immer leistungsfähigerer Hardware ist jedoch nicht auszuschließen, dass in einigen Jahren neuronale KPA zu anderen, nicht-neuronalen Ansätzen aufschließen bzw. diese vielleicht sogar übertreffen. Dieser abschließende Vergleich beantwortet in Kombination mit den genannten Einschränkungen und der Evaluierung des Algorithmus die dritte und letzte Forschungsfrage („Wie gut eignen sich die erstellten Modelle tatsächlich für neuronale KPA auf Transpositionschiffren in der Praxis?“). Insgesamt konnten somit alle vorhandenen Forschungsfragen detailliert beantwortet werden, wodurch die Ziele der Arbeit als erreicht angesehen werden.

Quellenverzeichnis

Literatur

[1] Klaus Schmeh. *Codeknacker gegen Codemacher: Die faszinierende Geschichte der Verschlüsselung*. 4. Aufl. Wiesbaden: Springer Fachmedien Wiesbaden, 2022.

[2] Thomas Kelly. „The Myth of the Skytale“. *Cryptologia* 22.3 (1998), S. 244–260.

[3] Beáta Megyesi u. a. „Decryption of Historical Manuscripts: The DECRYPT Project“. *Cryptologia* 44.6 (2020), S. 545–559.

[4] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. 1. Aufl. Cambridge, MA: MIT press, 2016.

[5] Sam Greydanus. „Learning the Enigma with Recurrent Neural Networks“. *arXiv preprint arXiv:1708.07576* (2017).

[6] Michael I. Jordan und Tom M. Mitchell. „Machine Learning: Trends, Perspectives, and Prospects“. *Science* 349.6245 (2015), S. 255–260.

[7] Aurélien Géron. *Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow: Konzepte, Tools und Techniken für intelligente Systeme. Aktuell zu TensorFlow 2*. 2. Aufl. Heidelberg: O'Reilly, 2020.

[8] Tom M. Mitchell. *The Discipline of Machine Learning*. 9. Aufl. Pittsburgh: Carnegie Mellon University, School of Computer Science, Machine Learning, 2006.

[9] Paul Wilmott. *Grundkurs Machine Learning*. 1. Aufl. Bonn: Rheinwerk Computing, 2020.

[10] Yann LeCun, Yoshua Bengio und Geoffrey Hinton. „Deep Learning“. *Nature* 521.7553 (2015), S. 436–444.

[11] Haohan Wang und Bhiksha Raj. „On the Origin of Deep Learning“. *arXiv preprint arXiv:1702.07800* (2017).

[12] Stuart Russell und Peter Norvig. *Künstliche Intelligenz*. 3. Aufl. Hallbergmoos: Pearson, 2012.

[13] Roland Schwaiger und Joachim Steinwendner. *Neuronale Netze programmieren mit Python*. 2. Aufl. Bonn: Rheinwerk Computing, 2019.

[14] Jean-Pierre Didier und Emmanuel Bigand. *Rethinking Physical and Rehabilitation Medicine: New Technologies Induce New Learning Strategies*. 1. Aufl. Paris: Springer Science & Business Media, 2010.

[15] Warren S. McCulloch und Walter Pitts. „A Logical Calculus of the Ideas Immanent in Nervous Activity“. *The Bulletin of Mathematical Biophysics* 5.4 (1943), S. 115–133.

[16] Jürgen Schmidhuber. „Deep Learning in Neural Networks: An Overview“. *Neural Networks* 61 (2015), S. 85–117.

[17] Frank Rosenblatt. „The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain“. *Psychological review* 65.6 (1958), S. 386.

[18] Marvin Minsky und Seymour Papert. *Perceptrons*. 1. Aufl. Cambridge, MA: MIT press, 1969.

[19] Paul Werbos. „Applications of Advances in Nonlinear Sensitivity Analysis“. In: *Proceedings of the 10th IFIP Conference* (31. Aug.–4. Sep. 1981). Hrsg. von Rudolf F. Drenick und Frank Kozin. New York, 1982, S. 762–770.

[20] Paul Werbos. „Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences“. Diss. Cambridge, MA: Harvard University, Jan. 1974.

[21] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. *Learning Internal Representations by Error Propagation*. Techn. Ber. ADA164453. Defense Technical Information Center, 1. Sep. 1985.

[22] Kunihiko Fukushima. „Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position“. *Biological Cybernetics* 36.4 (1980), S. 193–202.

[23] Sepp Hochreiter. „Untersuchungen zu dynamischen neuronalen Netzen“. Diplomarbeit. München: Technische Universität München, 15. Juni 1991.

[24] Sepp Hochreiter und Jürgen Schmidhuber. „Long Short-Term Memory“. *Neural Computation* 9.8 (1997), S. 1735–1780.

[25] Geoffrey Hinton, Simon Osindero und Yee-Whye Teh. „A Fast Learning Algorithm for Deep Belief Nets“. *Neural Computation* 18.7 (Juli 2006), S. 1527–1554.

[26] Kevin P. Murphy. *Probabilistic Machine Learning: An Introduction*. 1. Aufl. Cambridge, MA: MIT press, 2022.

[27] Lu Lu u. a. „Dying Relu and Initialization: Theory and Numerical Examples“. *arXiv preprint arXiv:1903.06733* (2019).

[28] Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng u. a. „Rectifier Nonlinearities Improve Neural Network Acoustic Models“. In: Bd. 30. 1. 2013, S. 3.

[29] Djork-Arné Clevert, Thomas Unterthiner und Sepp Hochreiter. „Fast and Accurate Deep Network Learning by Exponential Linear Units (elus)“. *arXiv preprint arXiv:1511.07289* (2015).

[30] Günter Klambauer u. a. „Self-Normalizing Neural Networks“. In: *Advances in Neural Information Processing Systems*. Hrsg. von I. Guyon u. a. Bd. 30. Curran Associates, Inc., 2017, S. 971–981.

[31] TensorFlow Documentation. *tf.keras.layers.LSTM*. Version TensorFlow Core v2.8.0. 18. März 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM.

[32] TensorFlow Documentation. *tf.keras.layers.GRU*. Version TensorFlow Core v2.8.0. 18. März 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU.

[33] Boris T. Polyak. „Some Methods of Speeding Up the Convergence of Iteration Methods“. *USSR Computational Mathematics and Mathematical Physics* 4.5 (1964), S. 1–17.

[34] John Duchi, Elad Hazan und Yoram Singer. „Adaptive Subgradient Methods for Online Learning and Stochastic Optimization“. *Journal of Machine Learning Research* 12.7 (2011).

[35] Tijmen Tieleman und Geoffrey E. Hinton. *Lecture 6.5 - RMSProp*. Techn. Ber. COURSERA: Neural Networks for Machine Learning, 2012.

[36] Diederik P Kingma und Jimmy Ba. „Adam: A Method for Stochastic Optimization“. *arXiv preprint arXiv:1412.6980* (2014).

[37] Sebastian Ruder. „An Overview of Gradient Descent Optimization Algorithms“. *arXiv preprint arXiv:1609.04747* (2016).

[38] Anna Choromanska u. a. „The Loss Surfaces of Multilayer Networks“. In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics* (9.–12. Mai 2015). Hrsg. von Guy Lebanon und S. V. N. Vishwanathan. Bd. 38. Proceedings of Machine Learning Research. San Diego: PMLR, 2015, S. 192–204.

[39] Xavier Glorot und Yoshua Bengio. „Understanding the Difficulty of Training Deep Feedforward Neural Networks“. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (13.–15. Mai 2015). Hrsg. von Yee Whye Teh und Mike Titterington. Bd. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, S. 249–256.

[40] TensorFlow Documentation. *tf.keras.layers.Dense*. Version TensorFlow Core v2.8.0. 18. März 2022. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense.

[41] Kaiming He u. a. „Delving Deep Into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification“. In: *Proceedings of the IEEE international Conference on Computer Vision* (7.–13. Dez. 2015). Santiago, Chile: IEEE, 2015, S. 1026–1034.

[42] Francois Chollet. *Deep Learning with Python*. 1. Aufl. Shelter Island: Manning, 2021.

[43] Nitish Srivastava u. a. „Dropout: A Simple Way to Prevent Neural Networks From Overfitting“. *The Journal of Machine Learning Research* 15.1 (2014), S. 1929–1958.

[44] Mike Schuster und Kuldip Paliwal. „Bidirectional Recurrent Neural Networks“. *IEEE Transactions on Signal Processing* 45 (Dez. 1997), S. 2673–2681.

[45] Felix Gers, Jürgen Schmidhuber und Fred Cummins. „Learning to Forget: Continual Prediction with LSTM“. *Neural Computation* 12 (Okt. 2000), S. 2451–71.

[46] Kyunghyun Cho u. a. „Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation“. *arXiv preprint arXiv:1406.1078* (2014).

[47] Klaus Greff u. a. „LSTM: A Search Space Odyssey“. *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (2016), S. 2222–2232.

[48] Samy Bengio u. a. „Scheduled Sampling for Sequence Prediction With Recurrent Neural Networks“. *Advances in Neural Information Processing Systems* 28 (2015), S. 1171–1179.

[49] Kyunghyun Cho u. a. „On the Properties of Neural Machine Translation: Encoder-Decoder Approaches“. *arXiv preprint arXiv:1409.1259* (2014).

[50] Dzmitry Bahdanau, Kyunghyun Cho und Yoshua Bengio. „Neural Machine Translation by Jointly Learning to Align and Translate“. *arXiv preprint arXiv:1409.0473* (2014).

[51] Minh-Thang Luong, Hieu Pham und Christopher D. Manning. „Effective Approaches to Attention-based Neural Machine Translation“. *arXiv preprint arXiv:1508.04025* (2015).

[52] Ashish Vaswani u. a. „Attention is All You Need“. *Advances in Neural Information Processing Systems* 30 (2017).

[53] National Institute of Standards und Technology. *NIST Special Publication 800-57 Part 1 Revision 5*. Recommendation for Key Management: Part 1 – General. Mai 2020.

[54] National Institute of Standards und Technology. *NIST Special Publication 800-59*. Guideline for Identifying an Information System as a National Security System. Aug. 2003.

[55] Bernhard Esslinger, Hrsg. *Das CrypTool-Buch: Kryptographie lernen und anwenden mit CrypTool und SageMath*. 12. Aufl. Siegen: CrypTool-Projekt, 2018.

[56] Dietmar Wätjen. *Kryptographie*. 3. Aufl. Wiesbaden: Springer Fachmedien Wiesbaden, 2018.

[57] Paolo Bonavoglia. „Trithemius, Bellaso, Vigenère–Origins of the Polyalphabetic Ciphers“. In: *Proceedings of the 3rd International Conference on Historical Cryptology HistoCrypt 2020*. 171. Linköping University Electronic Press. 2020, S. 46–51.

[58] Blaise de Vigenère. *Traicté des chiffres,ou Secrètes manières d'escrire*. 1. Aufl. Paris: Abel L'Angelier, 1586.

[59] Jeffrey Hoffstein u. a. *An Introduction to Mathematical Cryptography*. Bd. 1. New York: Springer New York, 2008.

[60] Lester S. Hill. „Cryptography in an Algebraic Alphabet“. *The American Mathematical Monthly* 36.6 (1929), S. 306–312.

[61] Marian Rejewski. „How Polish Mathematicians Deciphered the Enigma“. *Annals of the History of Computing* 3.3 (1981), S. 213–234.

[62] George Lasry, Nils Kopal und Arno Wacker. „Cryptanalysis of Columnar Transposition Cipher With Long Keys“. *Cryptologia* 40.4 (2016), S. 374–398.

[63] Thomas Mahon und James Gillogly. *Decoding the IRA*. 1. Aufl. Blackrock: Mercier Press, 2008.

[64] George Lasry u. a. „Deciphering ADFGVX Messages From the Eastern Front of World War I“. *Cryptologia* 41.2 (2017), S. 101–136.

[65] Alex Graves, Abdel-rahman Mohamed und Geoffrey Hinton. „Speech Recognition With Deep Recurrent Neural Networks“. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, S. 6645–6649.

[66] Ilya Sutskever, Oriol Vinyals und Quoc V. Le. „Sequence to Sequence Learning With Neural Networks“. *Advances in Neural Information Processing Systems* 27 (2014).

[67] Andrej Karpathy und Li Fei-Fei. „Deep Visual-Semantic Alignments for Generating Image Descriptions“. In: *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition*. 2015, S. 3128–3137.

[68] Md Saiful Islam und Emam Hossain. „Foreign Exchange Currency Rate Prediction Using a GRU-LSTM Hybrid Network“. *Soft Computing Letters* 3 (2021).

[69] Riccardo Focardi und Flaminia L. Luccio. „Neural Cryptanalysis of Classical Ciphers“. In: *ICTCS*. 2018.

[70] Nada Aldarrab und Jonathan May. „Can Sequence-to-Sequence Models Crack Substitution Ciphers?“ *arXiv preprint arXiv:2012.15229* (2021).

[71] Ernst Leierzopf u. a. „A Massive Machine-Learning Approach for Classical Cipher Type Detection Using Feature Engineering“. In: *International Conference on Historical Cryptology*. 2021, S. 111–120.

[72] Ernst Leierzopf u. a. „Detection of Classical Cipher Types with Feature-Learning Approaches“. In: *Australasian Conference on Data Mining*. Springer. 2021, S. 152–164.

[73] Timothy Dozat. „Incorporating Nesterov Momentum Into Adam“ (2016).

[74] Matthew D. Zeiler. „Adadelta: An Adaptive Learning Rate Method“. *arXiv preprint arXiv:1212.5701* (2012).

[75] Richard W. Hamming. „Error Detecting and Error Correcting Codes“. *The Bell System Technical Journal* 29.2 (1950), S. 147–160.

Medien

[76] *Basics of Cryptology – Part 1 (Cryptography – Terminology & Classical Ciphers)*. 27. Feb. 2020. URL: https://www.youtube.com/watch?v=jbumW7Ym03o.

Online-Quellen

[77] CrypTool. *Zeittafel / Zeitreise durch Kryptografie und Kryptoanalyse*. URL: https://www.cryptool.org/de/education/history (besucht am 02. 03. 2022).

[78] Microsoft. *What is a Machine Learning Model?* 30. Dez. 2021. URL: https://docs.microsoft.com/en-us/windows/ai/windows-ml/what-is-a-machine-learning-model (besucht am 10. 03. 2022).

[79] Unbekanter Autor. *File:Neuron (deutsch)-1.svg*. 19. Dez. 2006. URL: https://commons.wikimedia.org/w/index.php?curid=1474963 (besucht am 27. 03. 2022).

[80] Sebastian Raschka. *Is the Logistic Sigmoid Function Just a Rescaled Version of the Hyberpolic Tangent (tanh) Function?* 2022. URL: https://sebastianraschka.com/faq/docs/tanh-sigmoid-relationship.html (besucht am 22. 03. 2022).

[81] Jason Brownlee. *How to Choose an Activation Function for Deep Learning*. 22. Jan. 2021. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/ (besucht am 22. 03. 2022).

[82] Jason Brownlee. *A Gentle Introduction to Cross-Entropy for Machine Learning*. 21. Okt. 2019. URL: https://machinelearningmastery.com/cross-entropy-for-machine-learning/ (besucht am 16. 03. 2022).

[83] Jason Brownlee. *A Gentle Introduction to Mini-Batch Gradient Descent and How to Configure Batch Size*. 19. Sep. 2019. URL: https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/ (besucht am 17. 03. 2022).

[84] Jason Brownlee. *Weight Initialization for Deep Learning Neural Networks*. 3. Feb. 2021. URL: https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/ (besucht am 18. 03. 2022).

[85] Michael Copeland. *What's the Difference Between Deep Learning Training and Inference?* 16. Aug. 2016. URL: https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/ (besucht am 20. 03. 2022).

[86] Christopher Olah. *Understanding LSTM Networks*. 27. Aug. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (besucht am 25. 03. 2022).

[87] Jason Brownlee. *What is Teacher Forcing for Recurrent Neural Networks?* 6. Dez. 2017. URL: https://machinelearningmastery.com/teacher-forcing-for-recurrent-neural-networks/ (besucht am 28. 03. 2022).

[88] Cornell University. *Polyalphabetic Substitution Ciphers*. 18. März 2004. URL: https://pi.math.cornell.edu/~mec/2003-2004/cryptography/polyalpha/polyalpha.html (besucht am 15. 04. 2022).

[89] Loki 66. *Datei:Buchstabenhäufigkeit Deutsch.svg*. 13. Juni 2011. URL: https://de.m.wikipedia.org/wiki/Datei:Buchstabenh%C3%A4ufigkeit_Deutsch.svg (besucht am 16. 04. 2022).

[90] CrypTool. *Autokey*. URL: https://www.cryptool.org/de/cto/autokey (besucht am 16. 04. 2022).

[91] Messer Woland. *File:Enigma wiring kleur.svg*. 15. März 2007. URL: https://commons.wikimedia.org/wiki/File:Enigma_wiring_kleur.svg (besucht am 18. 04. 2022).