Universität Siegen

Wirtschaftswissenschaften
Wirtschaftsinformatik
Wirtschaftsrecht

Projektarbeit

zur Erlangung des Moduls „Projektarbeit“ Bachelor of Science Wirtschaftsinformatik

Kryptoanalyse der Enigma m.H. von LLMs: Anwendung der Transformer-Architektur zur Bestimmung des Enigma-Steckerbretts

Betreuer	Maik Bastian
Erstprüfer	Prof. Bernhard Esslinger
vorgelegt von	Tom Heindrichs
Matrikelnummer	1770733
Abgabedatum	30. Juni 2026 (Update 09.07.2026)

Kurzzusammenfassung

Diese Arbeit untersucht, ob ein Transformer-basiertes Modell unter *Ciphertext-Only*-Bedingungen in der Lage ist, das Steckerbrett der Enigma-Maschine rein datengetrieben zu erlernen. Im Zentrum steht damit das Element der Enigma mit dem größten Beitrag zum Schlüsselraum — das Steckerbrett. Die Kryptoanalyse wird als Multi-Label-Klassifikationsproblem formuliert, bei dem die aktiven Steckerpaare aus dem Geheimtext und den bekannten Enigma-Maschinenparametern vorhergesagt werden. Zur Validierung der Trainingspipeline wird in dieser Arbeit die Caesar-Chiffre verwendet.

Für die Experimente wird eine Pipeline zur Datenaufbereitung, Verschlüsselung, Kodierung der Enigma-Maschinenparameter, zum Trainingsablauf und zur Modellbewertung implementiert. Als Klartextbasis dient der *TinyStories*-Datensatz, aus dem mit Hilfe generierter Enigma-Schlüssel Trainings-, Validierungs- und Testdaten erstellt werden. Die Enigma-Schlüssel werden vor dem eigentlichen Training vorab berechnet, in einer SQLite-Datenbank persistiert und über ein PyTorch-Dataset geladen. Das eingesetzte Modell basiert auf einer *Encoder-Only*-Transformer- Architektur, die den Geheimtext gemeinsam mit dem bekannten Teil der Enigma-Maschinenparameter verarbeitet.

Die Ergebnisse zeigen, dass die Caesar-Chiffre vom Modell erlernt werden kann und damit die Validierungsbaseline erfüllt ist. Bei der Enigma hingegen erweist sich die Vorhersage der Steckerpaare auf dem Test-Set mit einer Genauigkeit von 48,20 % bei 10 Steckerpaaren als schwieriger. Insbesondere Modellparameter, Datenverteilung, Klassenbalancierung, Sampling-Strategie und die Komplexität der Enigma-Maschinenparameter beeinflussen das Lernverhalten. Die Arbeit liefert damit sowohl eine reproduzierbare Implementierungs- und Evaluationspipeline als auch eine Einordnung der Grenzen einer Kryptoanalyse bei der Enigma.

Abstract

This paper focuses on the ability of Transformer-based models to learn to reconstruct the plugboard of the Enigma machine under *ciphertext-only* conditions. Therefore, the focus lies on the most key-space-dominating element of the Enigma, the plugboard. The cryptanalytic task is formulated as a multi-label classification problem in which the active plugboard connections are predicted from the structure of the ciphertext together with the remaining machine settings. The Caesar cipher is included only as a baseline and is used to validate the training pipeline.

To conduct the experiments, a complete pipeline for data preparation, encryption, machine-setting encoding, training, and model evaluation is implemented. The *TinyStories*

dataset serves as the plaintext source, from which training, validation, and test data are generated with prepared Keys. The Enigma keys are precomputed before training, stored in an SQLite database, and loaded through a PyTorch dataset. The model itself is based on an *encoder-only* Transformer architecture that processes the ciphertext together with the known subset of machine settings.

The results show that the Caesar cipher can be learned reliably and therefore fulfills its role as a validation baseline. In the Enigma setting, however, predicting plugboard connections proves to be substantially more challenging. On the test set, the model reaches an accuracy of 48.20 % for 10 connections, which shows that it can learn the task but not master it. In particular, model parameters, data distribution, class balancing, sampling strategy, and the variability of Enigma machine settings have a strong influence on the learning behavior. The thesis therefore contributes both a reproducible implementation and evaluation pipeline and an empirical assessment of the limits of data-driven cryptanalysis for polyalphabetic historical ciphers.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Informationssicherheit basiert maßgeblich auf der Robustheit kryptografischer Verfahren, welche die Vertraulichkeit und Integrität digitaler Kommunikation gewährleisten. Gleichzeitig haben die jüngsten Fortschritte im Bereich der Künstlichen Intelligenz (KI), insbesondere bei der Transformer-Architektur [25] und Large Language Models (LLMs), gezeigt, dass diese Modelle in der Lage sind, komplexe Muster und Zusammenhänge in großen Datenmengen zu erkennen. Während in der aktuellen Sicherheitsforschung zunehmend evaluiert wird, inwieweit KI-Systeme zur automatisierten Schwachstellenanalyse eingesetzt werden können [12], bleibt eine zentrale Frage der Kryptoanalyse weitgehend offen: Sind deep-learning-basierte Modelle in der Lage, zustandsabhängige und kombinatorische Transformationen zu erlernen?

Im Gegensatz zu modernen Verschlüsselungsmethoden gelten historische Chiffren, wie das Caesar-Verfahren oder die polyalphabetische Enigma-Maschine, nach heutigen Maßstäben als historisch gebrochen [23, Kap. 1.1.2.2]. Dennoch bieten sie aufgrund ihrer Struktur ein gut kontrollierbares Szenario für moderne Machine-Learning-Ansätze. Während klassische Angriffe auf kryptografische Verfahren meist auf dem mathematischen Ausnutzen spezifischer Schwächen oder statistischer Häufigkeitsanalysen basieren, bietet Deep Learning einen weiteren Ansatz: Die statistische Mustererkennung in chiffrierten Datenströmen. Erste Ansätze, wie die Arbeit von Greydanus [8], haben gezeigt, dass Machine-Learning-Modelle in der Lage sind, Teile der Enigma zu lernen, jedoch unter stark vereinfachten Bedingungen. Ein weiteres Beispiel ist das CrypTool-Projekt [1] mit seinem KI-basierten Verschlüsselungstypen-Erkenner[1]: Es zeigt, dass es möglich ist, den Verschlüsselungstyp allein durch die Analyse eines Geheimtextes zu identifizieren. Diese Erfolge werfen die Frage auf, ob es möglich ist, die komplexeren Schlüsselstrukturen historischer Chiffren, wie der Enigma-Maschine, zu rekonstruieren.

Die besondere Herausforderung dieser Arbeit liegt in der Durchführung eines Angriffs unter *Ciphertext-Only*-Bedingungen, bei dem kein Wissen über den zugrundeliegenden Klartext existiert [6, Kap. 1.8.1]. Um die mathematische Komplexität besser untersuchbar zu machen, konzentriert sich diese Untersuchung auf das Element der Enigma mit dem größten Beitrag zum Schlüsselraum: das Steckerbrett. Während die fundamentalen mechanischen Parameter (wie Walzenlage und Ringstellung) im Rahmen des experimentellen Setups als bekannt vorausgesetzt oder systematisch variiert werden, verbleibt das Steckerbrett aufgrund seiner kombinatorischen Vielfalt der entscheidende Faktor, der den Schlüsselraum vergrößert [23, Kap. 1.1.5.1]. Die Kryptoanalyse wird hierbei als ein Multi-Label-Klassifikationsproblem [24] formuliert, bei dem das Modell die korrekten

[1] https://www.cryptool.org/de/cto/ncid/

Steckerpaare aus dem Geheimtext in Kombination mit den bekannten beziehungsweise vorgegebenen Enigma-Maschinenparametern ableiten muss. Hinsichtlich des Klartexts handelt es sich damit um ein *Ciphertext-Only*-Szenario, jedoch nicht hinsichtlich aller Schlüsselparameter.

Die Hauptforschungsfrage ist: Kann ein gezielt trainiertes Transformer-Modell eine Approximation der Zuordnung von Geheimtext und bekannten Maschinenparametern auf das Steckerbrett lernen und den verbleibenden Teilschlüssel in Form der verwendeten Steckerbrettpaare korrekt klassifizieren?

1.2 Ziele der Arbeit

Das Hauptziel dieser Arbeit besteht in der Untersuchung der Fähigkeiten Transformer-basierter Architekturen bei der Kryptoanalyse unter *Ciphertext-Only*-Bedingungen. Konkret wird analysiert, ob ein gezielt trainiertes Modell die Zuordnung von Geheimtext und bekannten Enigma-Maschinenparametern auf das Steckerbrett hinreichend gut approximieren und den zum Geheimtext gehörigen Teilschlüssel, in Form von bis zu zehn Steckerpaaren, korrekt klassifizieren kann.

Um dieses übergeordnete Forschungsziel systematisch zu erreichen, gliedert sich die Arbeit in folgende methodische Teilschritte:

1. **Konzeption und Validierung der Baseline:** Aufbau einer modularisierten Trainings- und Evaluationspipeline für eine einfache monoalphabetische Substitution, um die grundsätzliche Eignung der gewählten Modellarchitektur zu verifizieren.

2. **Infrastrukturelle Erweiterung für die Enigma:** Funktionale Erweiterung der Pipeline zur Abbildung der polyalphabetischen Drei-Walzen-Enigma.

3. **Optimierung durch Datenaufbereitung und Hyperparameter-Tuning:** Ermittlung einer dateneffizenten Strategie sowie verschiedener Hyperparameter zur Optimierung der Datenverteilung, um die Lernfähigkeit des Modells zu maximieren und das benötigte Trainingsvolumen zu minimieren.

4. **Initiales Modelltraining auf statischem Setup:** Training und Evaluation des Modells auf eine statische mechanische Grundeinstellung der Enigma bei einer variablen Belegung des Steckerbretts, ohne zusätzliche Variationen der anderen Enigma-Maschinenparameter.

5. **Generalisierung des Ansatzes auf variablem Setup:** Aufhebung der statischen Restriktionen durch die Erweiterung des Trainings auf variable Walzenlagen, Ringstellungen und Grundstellungen zur Überprüfung der Modellrobustheit.

6. **Evaluation und Einordnung:** Bewertung der Konvergenzgeschwindigkeit, Genauigkeit (*Accuracy*), Verlust (*Loss*) und weiterer Metriken. Abschließend folgt eine Einordnung der Ergebnisse in den aktuellen Forschungsstand.

1.3 Experimentierumgebung

Die praktischen Evaluierungen und Modelltrainings dieser Arbeit wurden in einer hybriden Hardware-Umgebung durchgeführt. Erste Prototypisierungen sowie lokale Vorversuche erfolgten auf einer dedizierten Workstation, ausgestattet mit einer NVIDIA GeForce GTX 1080. Für das finale Training der rechenintensiven Transformer-Modelle wurde auf die Ressourcen eines leistungsstarken GPU-Clusters mit NVIDIA A40 und A100 Tensor Core GPUs zurückgegriffen.

Die programmiertechnische Umsetzung erfolgte vollständig in der Programmiersprache Python. Zur Gewährleistung der Reproduzierbarkeit der Ergebnisse wurden sämtliche Abhängigkeiten innerhalb einer virtuellen Umgebung (`venv`) isoliert und in einer `requirements.txt`-Datei dokumentiert. Als fundamentale Deep-Learning- und Optimierungs-Frameworks kamen *PyTorch* [20] und *Ray Tune* [14] zum Einsatz. Die Simulation der Enigma-Maschine wurde über das Modul *py-enigma* [17] realisiert. Die Softwareentwicklung fand in der integrierten Entwicklungsumgebung Visual Studio Code statt, ergänzt durch Jupyter Notebooks für interaktive Datenanalysen. Die Quellcodeverwaltung sowie die fortlaufende Versionierung wurden mittels Git realisiert, während die Erstellung der architektonischen Diagramme über Draw.io erfolgte.

Im Zuge der Ausarbeitung kamen Large Language Models (LLMs) als assistive Werkzeuge zum Einsatz. Diese dienten primär der iterativen Generierung von Code- und Textfragmenten sowie der initialen Strukturierung der Quellcodedokumentation. Jedes KI-generierte Artefakt wurde anschließend kritisch validiert, manuell an den spezifischen Anwendungsfall adaptiert und auf fachliche Korrektheit geprüft. Sofern nicht explizit durch externe Quellenangaben gekennzeichnet, stellen alle Erklärungen, Diagramme und Implementierungen die Eigenleistung innerhalb dieser Arbeit dar.

Der Quellcode der Implementierung, inklusive aller Skripte zur Datenaufbereitung, Modellarchitektur, sowie Trainings- und Evaluationspipelines, ist eingeschränkt auf GitHub verfügbar[2].

[2] https://github.com/taaooum/projektarbeit

1.4 Aufbau der Arbeit

Die Arbeit gliedert sich in sechs Kapitel. Nach der **Einleitung** legt das Kapitel **Grundlagen** die theoretischen Grundlagen dar: Es führt in die Methoden der Kryptoanalyse ein, beschreibt Aufbau sowie Schwachstellen der Caesar-Chiffre und der Enigma-Maschine und gibt anschließend eine Einführung in Deep Learning sowie die Transformer-Architektur.

Das Kapitel **Implementierung** beschreibt den konkreten Aufbau der Trainingspipeline: Von der Datenaufbereitung und der Kodierung der Chiffren-Klassen über die Modellarchitektur und das Training bis hin zur Evaluation.

Im Kapitel **Methodik & Experimente** werden die durchgeführten Experimente beschrieben. Es wird dargestellt, welche Hyperparameter und Datenstrategien systematisch variiert wurden und wie die einzelnen Versuchsreihen aufgebaut sind.

Das Kapitel **Ergebnisse & Diskussion** präsentiert die erzielten Ergebnisse, bewertet Konvergenzverhalten und Genauigkeit und ordnet die Befunde in den Kontext aktueller Forschung ein.

Abschließend fasst das **Fazit** die wichtigsten Erkenntnisse zusammen und gibt einen Ausblick auf mögliche Erweiterungen, Verbesserungen und offene Forschungsfragen.

2 Grundlagen

2.1 Kryptoanalyse

Die Kryptoanalyse untersucht Methoden und Algorithmen, mit denen sich verschlüsselte Informationen analysieren und kryptografische Verfahren auf ihre Vertraulichkeit prüfen lassen. Die folgenden Erläuterungen basieren, sofern nicht explizit anders gekennzeichnet, auf den methodischen Grundlagen von [6, Kap. 1.8.1], [23, Kap. 1.1.2.2] und [4, Kap. 3.4.2].

2.1.1 Klassifikation von Angriffsmodellen

Die Stärke eines kryptoanalytischen Angriffs wird durch die verfügbaren Informationen bestimmt. Das grundlegende Szenario dieser Arbeit ist der **Ciphertext-Only-Angriff**. Der Angreifer hat dabei ausschließlich Zugriff auf eine Menge von Geheimtexten. Informationen über den zugrunde liegenden Klartext, wie sie beim *Known-Plaintext-Angriff* vorliegen, oder die Möglichkeit, gezielt Texte verschlüsseln zu lassen, wie bei dem *Chosen-Plaintext-Angriff*, bestehen nicht.

2.1.2 Kryptoanalytische Ansätze

Zur praktischen Durchführung eines Angriffs existieren verschiedene methodische Ansätze. Bei einer **vollständigen Suche** (*Brute-Force-Angriff*) wird der gesamte Schlüsselraum systematisch durchsucht, bis der resultierende Klartext ein definiertes Abbruchkriterium, etwa semantische Plausibilität, erfüllt. Ist der Schlüsselraum dafür zu groß, kommen analytische Verfahren wie die **statistische Häufigkeitsanalyse** zum Einsatz. Sie nutzt die charakteristische Verteilung von Zeichen natürlicher Sprachen. Bei einfachen monoalphabetischen Substitutionsverfahren wird ein Klartextbuchstabe immer durch dasselbe Geheimtextzeichen ersetzt. Dadurch bleibt das statistische Profil, im Deutschen etwa die Häufigkeit des Buchstabens „E“, auch im Geheimtext erhalten und kann ausgenutzt werden [3, Kap. 1.3].

2.1.3 Konfusion und Diffusion

Die theoretische Grundlage moderner, sicherer Verschlüsselung bilden die von Claude Shannon vorgestellten Prinzipien [22]:

- **Konfusion:** Ziel ist es, den Zusammenhang zwischen Schlüssel und Geheimtext möglichst komplex und nichtlinear zu gestalten, damit sich mathematische Korrelationen nur schwer erkennen lassen.

- **Diffusion:** Sie beschreibt die Verteilung der statistischen Eigenschaften eines einzelnen Klartextzeichens über möglichst viele Stellen des Geheimtextes. Fehlt diese Eigenschaft, bleiben lokale sprachliche Muster im Text leichter erkennbar.

Moderne Verschlüsselungsverfahren kombinieren in der Regel beide Prinzipien. Einfache historische Chiffren erfüllen dagegen meist nur eines oder keines davon.

2.2 Caesar-Chiffre

Die Caesar-Chiffre dient in dieser Arbeit als methodischer Einstieg und grundlegendes Referenzmodell. Die Caesar-Chiffre wird als monoalphabetische Substitutionschiffre sowie als Verschiebe-Chiffre (*displacement cipher*) klassifiziert. Der Verschlüsselungsmechanismus basiert auf einer zyklischen Verschiebung jedes Klartextbuchstabens um eine feste Anzahl von k Positionen im Alphabet. [23, Kap. 1.1.2]

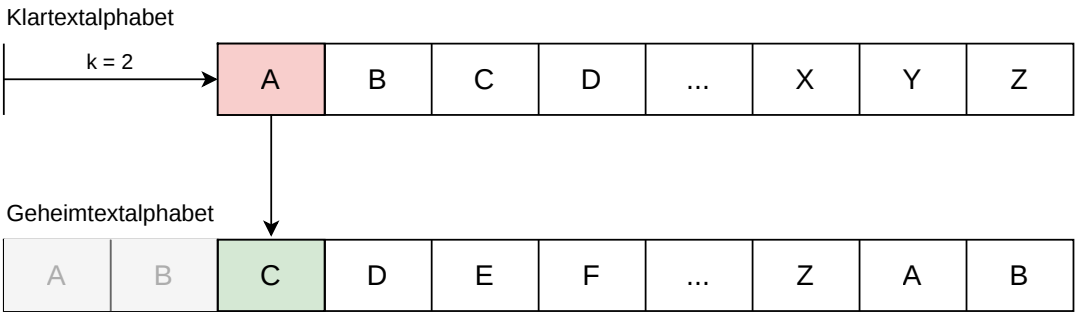

Abb. 1: Visualisierung der Caesar-Verschiebung des Alphabets um $k = 2$.

Abbildung 1 zeigt beispielhaft eine Verschiebung um $k = 2$. Der Klartext **HALLO** wird dabei zum Geheimtext **JCNNQ**. Gerade diese Einfachheit macht das Verfahren anschaulich, aber auch leicht angreifbar.

Hypothese 1

Es ist anzunehmen, dass ein Transformer-Modell die zugrundeliegende Buchstabenverschiebung der Caesar-Chiffre aufgrund ihres sehr kleinen Schlüsselraums und der einfachen algorithmischen Struktur bereits mit einer geringen Menge an Trainingsdaten erlernen kann.

2.3 Chiffrier-Maschine Enigma

Die von den deutschen Streitkräften im Zweiten Weltkrieg eingesetzte elektromechanische Rotormaschine realisiert eine komplexe polyalphabetische Substitution. Im Verlauf ihrer Einsatzzeit entstanden verschiedene Versionen der Enigma mit unterschiedlichen Konfigurationen. [23, Kap. 1.1.5][10, Kap. 9]

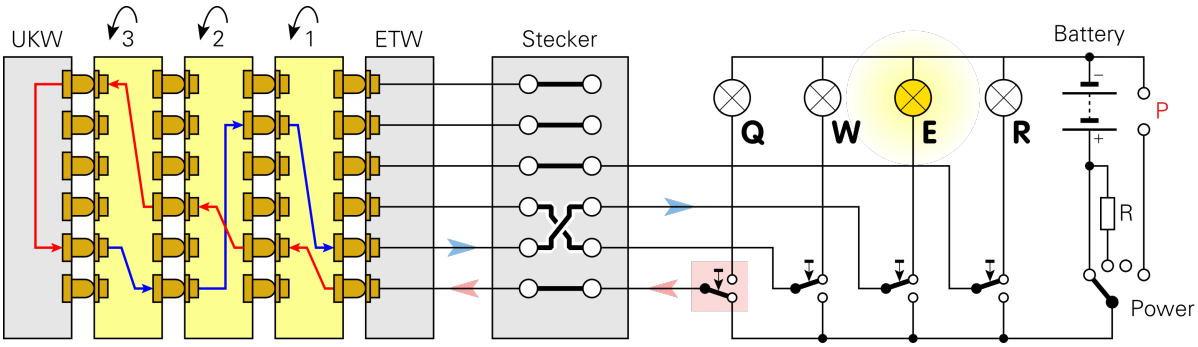

Abb. 2: Vereinfachte Visualisierung einer 3-Rotor Enigma-Maschine [21].

Anhand der Abbildung 2 lässt sich der elektrische Stromlauf des Verschlüsselungsvorgangs gut nachvollziehen: Ein eingegebenes Signal passiert zunächst das **Steckerbrett**, auf dem (je nach Konfiguration) bis zu zehn Steckerpaare die jeweiligen Buchstaben vertauschen. Danach durchläuft es die mechanischen Walzen, bestehend aus typischerweise drei rotierenden **Walzen (Rotoren)**. Ähnlich einem Kilometerzähler dreht sich die rechte Walze nach jedem Tastendruck weiter und löst nach einer vollen Umdrehung einen mechanischen Übertrag auf die benachbarte Walze aus. Am Ende des Walzensatzes lenkt eine feststehende **Umkehrwalze (Reflektor)** das Signal auf einem veränderten Pfad durch die Rotoren und das Steckerbrett zurück zum Lampenfeld.

Im Einsatz der Maschine wurden sogenannte Tagesschlüssel verwendet, die die initiale Konfiguration definierten. Diese setzten sich aus vier Variablen zusammen: Der Auswahl der **Walzen (Walzenlage)**, der Justierung des **Übertragspunktes (Ringstellung)**, der **Startposition** der **Rotoren (Grundstellung)** sowie den Kabelverbindungen auf dem **Steckerbrett**. Während die mechanischen Walzenkombinationen noch überschaubar waren, erhöhte allein das Steckerbrett mit über 150 Billionen Möglichkeiten die Komplexität drastisch. Insgesamt ergab sich ein Schlüsselraum von etwa $2 \cdot 10^{23}$ Kombinationen. [23, Kap. 1.1.5.1]

Wendet man die zuvor definierten Prinzipien der Kryptoanalyse 2.1 auf die Enigma an, zeigt sich: Der große Schlüsselraum schützte die Maschine wirksam vor bloßen **Brute-Force-Angriffen** ab. Im Gegensatz zur monoalphabetischen Caesar-Chiffre ändert sich bei dieser polyalphabetischen Substitution das wirksame Alphabet durch die rotierenden Walzen nach jedem Tastendruck. Diese ständige Neuverdrahtung erzeugt **Konfusion** und erschwert gewöhnliche **statistische Häufigkeitsanalysen** erheblich. Dennoch

besaß das System zwei zentrale Schwachstellen. Erstens besaß die Enigma keine **Diffusion** im Shannon'schen Sinn, da ein Tastendruck stets genau einen Buchstaben im Geheimtext veränderte. Über längere Texte entsteht durch die Rotorbewegung zwar eine gewisse statistische Durchmischung, diese ist jedoch nicht mit der Diffusion moderner Blockchiffren gleichzusetzen. Zweitens sorgte die Umkehrwalze zwar für eine selbstinverse Bedienung, da Ver- und Entschlüsselung identisch waren, dies führte jedoch zu dem grundlegenden Konstruktionsmerkmal, dass **kein Buchstabe jemals auf sich selbst abgebildet werden konnte**. Genau diese hartkodierten statistischen Eigenschaften boten eine Angriffsfläche. [23, Kap. 1.1.5.2]

Hypothese 2

Es ist anzunehmen, dass ein Transformer-Modell die zustandsabhängige kombinatorische Komplexität der Enigma-Maschine aufgrund des riesigen Schlüsselraums und der zustandsabhängigen Transformationen nur mit einer sehr großen Menge an Trainingsdaten erlernen kann. Im Gegensatz zur Caesar-Chiffre ist bei der Enigma anzunehmen, dass die Geheimtext-basierende Mustererkennung des Transformers hier voraussichtlich an ihre Grenzen stoßen wird. Grund dafür ist der riesige Schlüsselraum und die Komplexität der zustandsabhängigen kombinatorischen Transformationen, die die Enigma während der Verschlüsselung ausführt.

2.4 Deep Learning

Deep Learning ist ein Teilbereich des maschinellen Lernens, der auf künstlichen neuronalen Netzen basiert. Ziel dieser Netze ist es, komplexe, nichtlineare Zusammenhänge in Daten zu approximieren. Die nachfolgenden Erläuterungen basieren, sofern nicht explizit anders gekennzeichnet, auf den methodischen Grundlagen von [7].

2.4.1 Aufbau künstlicher neuronaler Netze

Ein künstliches neuronales Netz besteht aus mehreren Schichten von Neuronen, die durch gewichtete Verbindungen miteinander verbunden sind. Jedes Neuron empfängt Eingaben, verarbeitet sie und gibt ein Signal an die nächste Schicht weiter. Durch die Anpassung der Gewichte (w) und Biases (b) während des Trainingsprozesses lernt das Netz, komplexe Muster in den Daten zu erkennen und Vorhersagen zu treffen.

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{1}$$

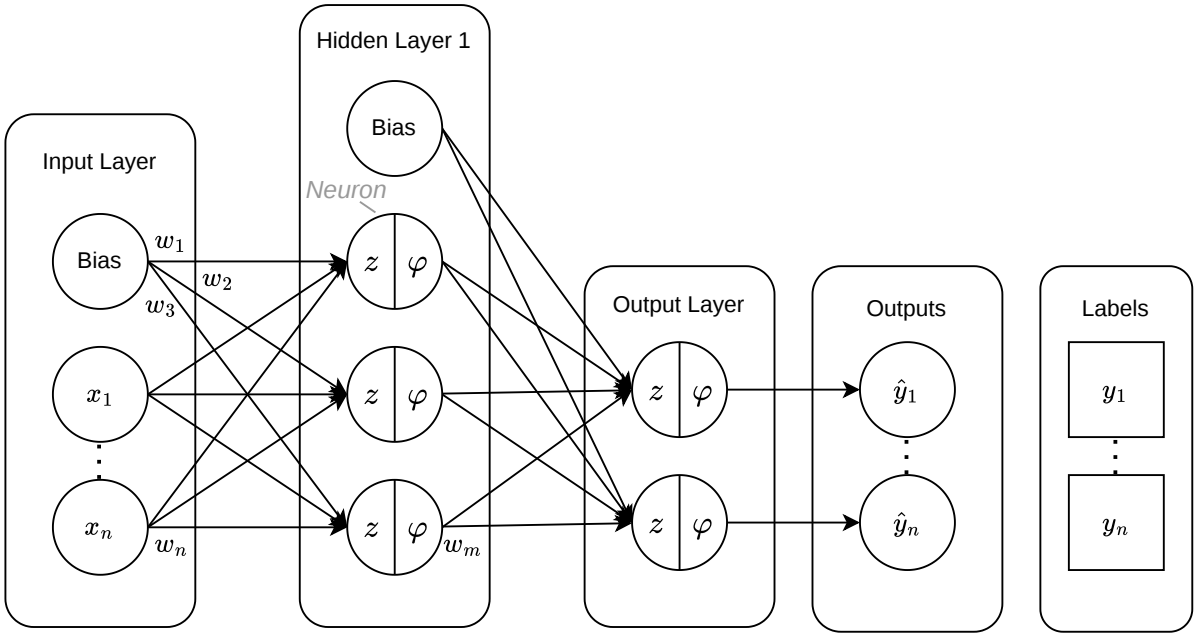

Abb. 3: Vereinfachte Visualisierung eines künstlichen neuronalen Netzes mit einem Hidden Layer.

Wie in Abb. 3 veranschaulicht, durchlaufen die Eingabedaten $(x_1 \dots x_n)$ das Netzwerk schichtweise über künstliche Neuronen. Jedes Neuron in einer verborgenen Schicht (*Hidden Layer*) berechnet zunächst wie in Gleichung (1) die gewichtete Summe seiner Eingaben zuzüglich des Bias-Wertes.

Diese lineare Kombination (z) allein reicht jedoch nicht aus, da auch die Hintereinanderausführung vieler linearer Operationen mathematisch wieder nur eine lineare Transformation ergibt. Ein lineares Netzwerk könnte daher allenfalls einfache Verschlüsselungen wie die Caesar-Chiffre abbilden, würde aber an den dynamischen Zustandswechseln der Enigma-Walzen scheitern.

Aus diesem Grund wird das Zwischenergebnis z an eine nichtlineare Aktivierungsfunktion φ übergeben. Erst das Zusammenspiel aus anpassbaren Parametern und Nichtlinearität ermöglicht es dem Netzwerk, abstrakte Repräsentationen der kryptografischen Struktur zu bilden und in der Ausgabeschicht eine Vorhersage $\hat{y}$ (auch „Label“ genannt) zu erzeugen.

2.4.2 Training und Optimierung

```
# Training Loop:
for each epoch do
    for each batch do
        Load data
        Pass data through network          # Forward Pass
        Calculate losses                   # Loss Calculation
        Calculate gradients                # Backpropagation
        Adjust network weights             # Optimization
    end
end
```

Listing 1: Pseudocode eines vereinfachten Trainingsloops eines neuronalen Netzwerks.

Wie aus Listing 1 ersichtlich, gliedert sich das Training eines neuronalen Netzwerks in mehrere aufeinander aufbauende zyklische Phasen:

1. **Forward Pass:** Das Modell verarbeitet die Eingabesequenz Batchweise[3] und erzeugt eine Vorhersage ($\hat{y}$).

2. **Loss-Berechnung (Loss Calculation):** Die Vorhersage wird mit dem tatsächlichen Label (y, *Ground Truth*) über eine Loss-Funktion (*Loss Function*) verglichen. Das Ergebnis ist der Loss-Wert (L), der die Fehlergröße quantifiziert.

3. **Backpropagation:** Der berechnete Loss-Wert wird mittels partieller Ableitungen rückwärts durch das Netzwerk geleitet. Das Ergebnis ist der Gradient (∇L).

4. **Optimization:** Ein Optimierer nutzt den berechneten Gradienten, um die Gewichte des Netzwerks schrittweise entgegen der Gradientenrichtung anzupassen. Die klassische Aktualisierungsregel lautet:

$$w_{t+1} = w_t - \eta \cdot \nabla L \tag{2}$$

   wobei η die Lernrate bezeichnet.

Damit ein Modell effektiv lernt, benötigt es eine Information darüber, in welche Richtung seine Parameter verändert werden sollen. Diese Information liefert der Gradient. Er beschreibt die Richtung und Stärke der Anpassung für jedes Gewicht im Netzwerk. Die Richtung des Gradienten zeigt an, ob ein Gewicht erhöht oder verringert werden sollte, um den Loss zu minimieren. Die Stärke des Gradienten gibt an, wie stark die Anpassung sein sollte.

[3] Ein Batch ist eine Gruppe von Eingabesequenzen, die gleichzeitig durch das Netzwerk geleitet werden, um die Effizienz zu erhöhen.

Mathematisch ist der Gradient ∇L ein Vektor, der sich aus den partiellen Ableitungen der Ergebnisse der Loss-Funktion nach allen lernbaren Parametern (Gewichten w_i) des Netzwerks zusammensetzt:

$$\nabla L = \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_d} \right)^T \tag{3}$$

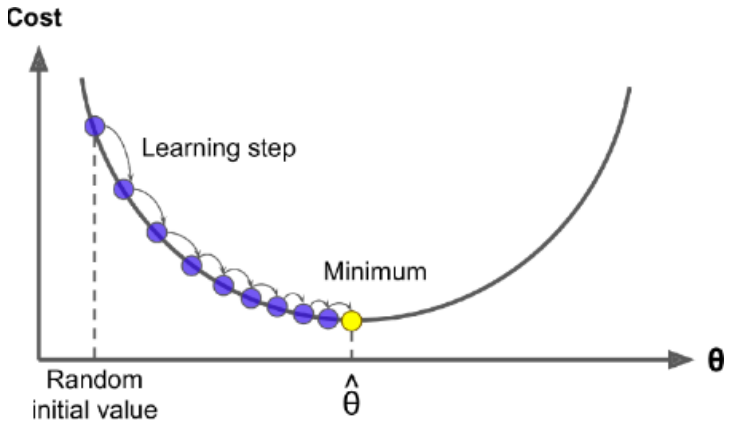

Abb. 4: Einfacher Gradient mit zwei Dimensionen und Optimierungsschritten. [7, Kap. 4]

Geometrisch kann die Loss-Funktion als topografische Landschaft (siehe Abb. 4 eines zweidimensionalen Beispiels) im hochdimensionalen Parameterraum verstanden werden (*Loss Landscape*). Der Gradientenvektor zeigt an jedem Punkt in die Richtung des steilsten Fehleranstiegs. Da das Training den Fehler minimieren soll, erfolgt die Parameteraktualisierung in die entgegengesetzte Richtung des Gradienten ($-\nabla L$). Dieses Prinzip wird als *Gradient Descent* bezeichnet.

Die Lernrate η (*Learning Rate*) bestimmt die Schrittweite der Anpassung im Parameterraum. Moderne Optimierungsverfahren variieren diese Schrittweite dynamisch für einzelne Parameter oder nutzen einen *Scheduler*, um die Lernrate über den Trainingsverlauf hinweg anzupassen und das Konvergenzverhalten zu stabilisieren.

2.4.3 Evaluationsmetriken

Um die Leistungsfähigkeit und das Konvergenzverhalten eines Modells zu bewerten, werden verschiedene Metriken herangezogen. Da bei Klassifikationsaufgaben eine isolierte Metrik (wie Accuracy) meist nicht ausreicht, werden im Trainings- und Evaluationsprozess verschiedene Maße kombiniert. Dies gilt insbesondere für Multi-Label-Probleme mit stark ungleicher Klassenverteilung, bei denen viele negative Klassen eine hohe Accuracy

vortäuschen können, obwohl positive Klassen nur unzureichend erkannt werden. Daher werden in dieser Arbeit zusätzlich Precision, Recall und F1-Score verwendet, um den Umgang des Modells mit positiven Klassen getrennt von den vielen True Negatives zu bewerten. Grundlage sind die vier Elemente der Fehlermatrix um die vorhergesagten Klassen[4] zu bewerten: *True Positives (TP)*, *True Negatives (TN)*, *False Positives (FP)* und *False Negatives (FN)*.

- **Verlust (Loss):** Der Wert der Loss-Funktion selbst dient als kontinuierliches Maß für die Fehlergröße. Er wird während des Trainings und der Validierung fortlaufend überwacht, um die Lernkurve zu analysieren und quantitative Trends wie eine Stagnation des Lernerfolgs sichtbar zu machen.

- **Genauigkeit (Accuracy):** Misst den relativen Anteil der exakt korrekt klassifizierten Beispiele an der Gesamtheit aller abgegebenen Vorhersagen.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4}$$

- **Präzision (Precision):** Bestimmt den Anteil der korrekt als positiv vorhergesagten Klassen an allen positiven Vorhersagen, um fälschlicherweise positive Klassifikationen (*False Positives*) zu bestrafen.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{5}$$

- **Sensitivität (Recall):** Gibt an, welchen Anteil der tatsächlich positiven Klassen das Modell korrekterweise identifiziert hat, um das Übersehen relevanter Merkmale (*False Negatives*) zu bewerten.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{6}$$

- **F1-Score:** Bildet das harmonische Mittel aus Precision und Recall, um beide Dimensionen zu einer gemeinsamen, balancierten Kennzahl zusammenzufassen.

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7}$$

Die Metriken werden in dieser Arbeit mikro-gemittelt berechnet. Dazu werden die Elemente der Fehlermatrix zunächst über alle Klassen hinweg global summiert und erst anschließend die Kennzahlen bestimmt. Makro-gemittelte Metriken berechnen die Kennzahlen dagegen zunächst für jede Klasse separat und bilden danach den Durchschnitt.

[4] In der Klassifikation bezeichnet eine Klasse eine diskrete Kategorie oder ein Ziel-Label, das das Modell auf Basis der Eingabedaten vorhersagen soll. Im Kontext dieser Multi-Label-Klassifikation beschreibt eine Klasse eine mögliche Verbindung von zwei Buchstaben auf dem Steckerbrett.

2.4.4 Generalisierung und Regularisierung

Da das Netzwerk während des Trainings nur einen winzigen Bruchteil des großen Enigma-Schlüsselraums sehen kann, besteht die fundamentale Herausforderung darin, zu *generalisieren*, also die zugrunde liegende Zuordnung von Geheimtext und bekannten Maschinenparametern auf das Steckerbrett zu approximieren, statt einzelne Datenpunkte auswendig zu lernen.

Tritt Letzteres ein, spricht man von **Overfitting** (Überanpassung). Das Modell erzielt einen minimalen Fehler auf den Trainingsdaten, scheitert jedoch auf den ungesehenen Validierungsdaten. Das Gegenstück ist das **Underfitting** (Unteranpassung), bei dem das Modell aufgrund zu geringer Komplexität oder unzureichendem Training die grundlegenden Muster der Daten überhaupt nicht erfasst. Bei der Regularisierung werden verschiedene Techniken eingesetzt, um die Komplexität des Modells zu kontrollieren und die Gefahr der Überanpassung zu reduzieren.

2.5 Transformer-Architektur

Wie im vorherigen Abschnitt 2.4 erläutert, verarbeiten klassische neuronale Netze Eingaben isoliert. Für die Modellierung polyalphabetischer Chiffren wie der Enigma ist dieser Ansatz unzureichend: Da sich die Rotoren nach jedem Tastendruck weiterdrehen, hängt die kryptografische Bedeutung eines Zeichens von seiner exakten Position und dem Kontext der vorangegangenen Sequenz ab. Dieses Problem löst die *Transformer-Architektur* durch den sogenannten *Attention*-Mechanismus. Die folgenden Erläuterungen basieren, sofern nicht explizit anders gekennzeichnet, auf den methodischen Grundlagen von [25].

Abbildung 5 zeigt die klassische Transformer-Architektur mit einem Encoder- und einem Decoder-Teil. Der *Encoder* (links) dient primär der Merkmalsextraktion und dem Aufbau eines globalen Kontextes, während der *Decoder* (rechts) diese verdichteten Informationen nutzt, um autoregressiv eine neue Sequenz zu generieren. Für die Modellierung der Kryptoanalyse der Enigma als Klassifikationsaufgabe wird in dieser Arbeit eine *Encoder-Only-Architektur* eingesetzt. Die Begründung dieser Architekturentscheidung sowie der konkrete Datenfluss werden in Abschnitt 3.4 beschrieben.

2.5.1 Embeddings und Positionskodierung

Zur Vorverarbeitung der Eingabesequenzen werden die Token zunächst in ein Embedding überführt, anschließend durch eine *Positionskodierung* um Positionsinformationen

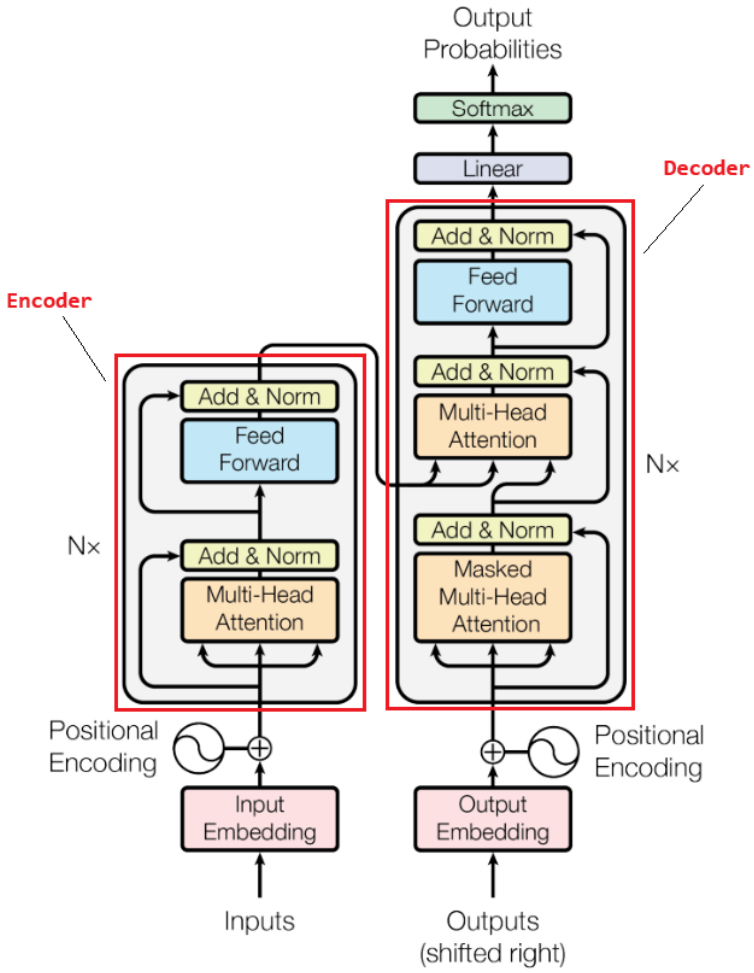

Abb. 5: Visualisierung der originalen Transformer-Architektur nach [25].

ergänzt und schließlich auf eine einheitliche Sequenzlänge gepaddet. Ein *Embedding* überführt diskrete Token-IDs in einen kontinuierlichen Vektorraum. Die *Positionskodierung* ergänzt Informationen über die Reihenfolge der Token, da die Transformer-Architektur selbst keine inhärente Positionsstruktur besitzt. Das *Padding* sorgt dafür, dass alle Sequenzen dieselbe Länge haben. Eine *Padding-Maske* stellt sicher, dass der Self-Attention-Mechanismus die hinzugefügten Füll-Token bei der Gewichtung ignoriert.

2.5.2 Multi-Head Self-Attention und Maskierung

Das funktionale Herzstück des Encoders bildet die *Self-Attention*. Hierbei wird die Beziehung jedes Tokens zu allen anderen Token der Sequenz berechnet. Dazu wird die eingebettete Sequenzmatrix durch lineare Projektionen in drei funktionale Repräsenta-

tionen überführt: *Query* (Q), *Key* (K) und *Value* (V). Die gewichtete Relevanz ergibt sich über das skalierte Tensorprodukt:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{8}$$

wobei d_k die Dimension der Keys bezeichnet. Um komplexe kryptoanalytische Muster parallel zu erfassen, wird dieses Prinzip als *Multi-Head Self-Attention* umgesetzt. Die Projektionen werden dabei in h getrennte Repräsentationsräume (Heads) aufgeteilt. So kann das Modell unterschiedliche mathematische und statistische Abhängigkeiten, etwa periodische Verschiebungen der Enigma-Walzen, gleichzeitig auf verschiedenen Abstraktionsebenen analysieren.

2.5.3 Residual Connections und Normalisierung (Add & Norm)

Residual Connections [9] erleichtern das Training tiefer Modelle, indem sie die ursprüngliche Eingabe einer Schicht an der eigentlichen Berechnung vorbeiführen und direkt zum Ergebnis addieren. Dadurch entsteht eine „Abkürzung“ für den Gradientenfluss, über die Fehlerinformationen auch frühe Schichten des Netzwerks zuverlässig erreichen.

Ergänzend dazu stabilisiert eine direkt nachgeschaltete *Layer Normalization* [2] die Verteilung der berechneten Werte über die Features hinweg. Während die Residual Connection den Signalfluss sichert, verhindert die Normalisierung, dass Werte im tiefen Netzwerk unkontrolliert anwachsen oder abfallen. Dieses kombinierte **Add & Norm**-Prinzip umschließt sowohl die Multi-Head-Self-Attention-Schicht als auch das nachfolgende Feed-Forward-Netzwerk.

2.5.4 Positionsweises Feed-Forward-Netzwerk (FFN) und Stacking

Um zusätzliche nichtlineare Abbildungen der Sequenzen zu ermöglichen, folgt auf die Multi-Head-Self-Attention ein *Feed-Forward-Netzwerk* (FFN). Ein vollständiger **Encoder-Block** besteht somit aus der Abfolge Multi-Head-Self-Attention, Add & Norm, FFN sowie einem zweiten Add & Norm-Schritt. Da ein einzelner Block in der Regel nicht ausreicht, um die komplexen Mechanismen der Enigma-Maschine abzubilden, werden N identische Encoder-Blöcke hintereinandergeschaltet. Dieser **Encoder-Stack** ermöglicht es dem Netzwerk, mit zunehmender Tiefe immer komplexere Abhängigkeiten in den Geheimtexten zu erfassen.

2.5.5 Lineare Ausgabeschicht

Die kontextualisierte Ausgabe des letzten Encoder-Blocks bildet einen Vektor der Dimension d_{model}, der die verdichteten Merkmale der gesamten Eingabesequenz enthält. Um daraus konkrete Vorhersagen über das Steckerbrett zu erzeugen, wird er durch eine vollständig vernetzte lineare Schicht (*Linear Layer*) in den Zielraum der Dimension M projiziert. Formal entspricht dies einer linearen Transformation mit zusätzlichem Bias-Vektor, wobei M die Anzahl der definierten Zielklassen bezeichnet.

Das Resultat dieses Schritts ist ein Vektor aus unnormierten Rohwerten, den sogenannten *Logits*. Welche Aktivierungs- und Loss-Funktion auf diesen Logits aufbaut, hängt von der konkreten Vorhersageaufgabe ab und wird im Rahmen der Modellimplementierung in Abschnitt 3.5.1 begründet und spezifiziert.

3 Implementierung

Während Kapitel 2 die theoretischen Grundlagen beschreibt, konzentriert sich dieses Kapitel auf die praktische Umsetzung. Im Mittelpunkt stehen die architektonischen Entscheidungen, die technische Repräsentation der Chiffren und die konkrete Datenaufbereitung. Die für dieses Kapitel relevante Projektstruktur ist in Abbildung 6 dargestellt.

```
project-root/
├── data/
│   ├── tiny-stories/
│   │   ├── prepare.py (Vorabberechnung und Datenaufbereitung)
│   │   └── analyse.ipynb (Analyse bereits erzeugter Daten)
│   └── vocab.json
├── src/
│   ├── ciphers/ (Caesar- und Enigma-Implementierung)
│   ├── config/ (Pfad- und Prepare-Konfiguration)
│   ├── dataset.py (SQLite-Dataset)
│   ├── precompute_worker.py (Parallele Verschluesselung)
│   ├── model.py
│   ├── train.py
│   └── eval.py
└── requirements.txt
```

Abb. 6: Auszug der Projektstruktur

3.1 Datenaufbereitung

Die Datenaufbereitung erfolgt in dieser Architektur nicht während des Trainings, sondern im Vorfeld als *Precomputation*. Das Skript `prepare.py` übernimmt dabei die Erzeugung der Daten. Konfiguriert wird der Lauf primär über `prepare_config.py` sowie `paths.py`; ergänzend erlaubt das Skript gezielte Kommandozeilen-Overrides für Stichprobengrößen, Sampling-Strategie und Zufallsparameter. Das Notebook `analyse.ipynb` ist demgegenüber nicht Teil der Vorberechnung, sondern dient der Visualisierung und Prüfung der bereits erzeugten Daten. Die Abb. 7 zeigt die Aufteilung dieser Pipeline.

3.1.1 TinyStories Dataset

Als Klartextbasis dient der *TinyStories*-Datensatz [5], welcher eine Vielzahl syntaktisch einfacher, englischer Kurzgeschichten enthält. Die Wahl fiel bewusst gegen klassische

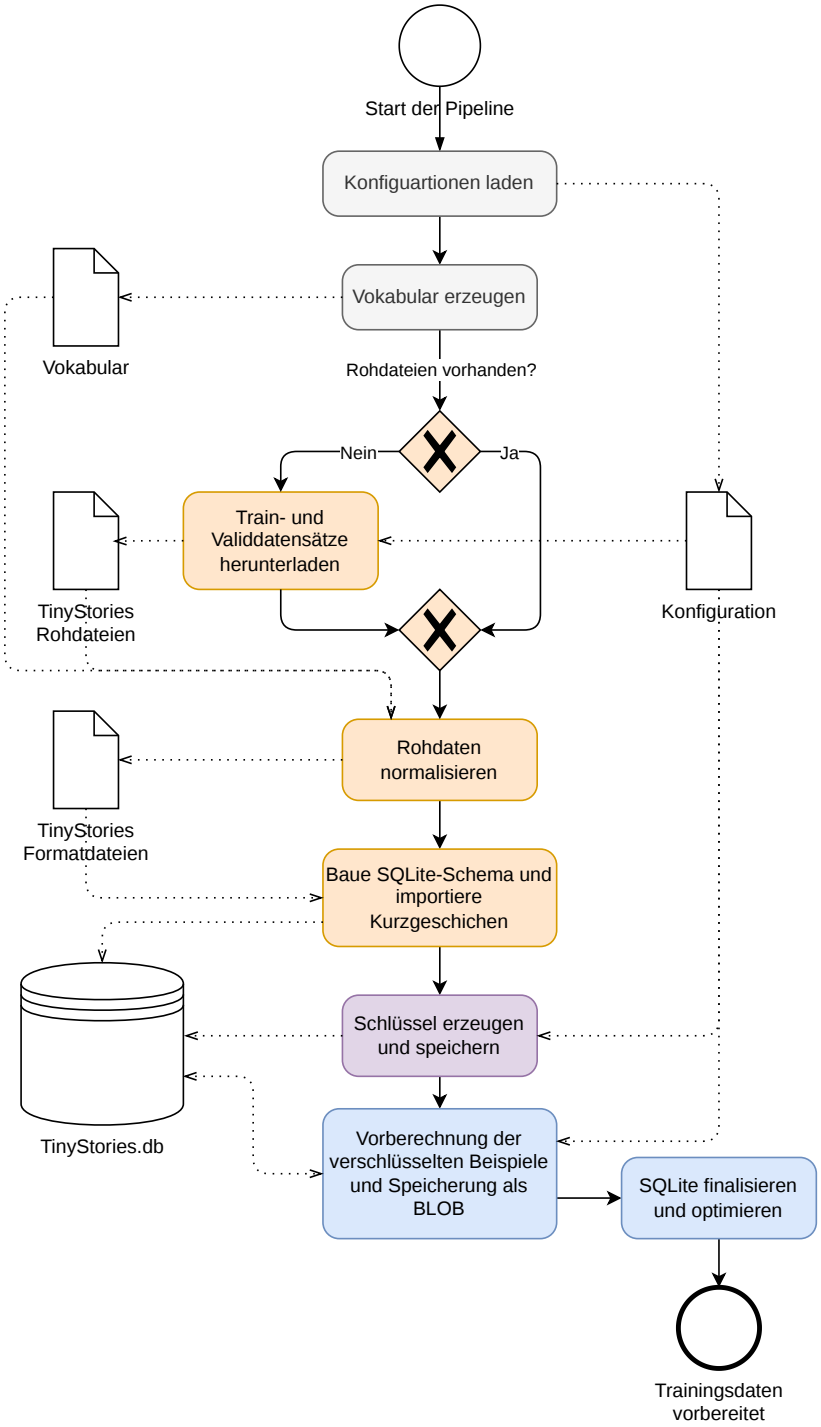

Abb. 7: Schematische Darstellung der Datenvorbereitungspipeline aus `prepare.py` und den Konfigurationsdateien

Literaturquellen wie den *Gutenberg*-Datensatz [13], da diese häufig komplexere Satz-

strukturen und ein breiteres Vokabular aufweisen. *TinyStories* liefert dagegen eine konsistentere Textgrundlage, die für die zeichenbasierte Verarbeitung besser geeignet ist und den Fokus stärker auf die mathematische Struktur der Chiffre lenkt.

Die Pipeline (siehe Abb. 7 in Orange markiert) lädt die Train- und Valid-Dateien bei Bedarf automatisiert aus dem Hugging-Face-Repository herunter und segmentiert die Rohdaten anhand des Markers `<|endoftext|>` in Geschichten mit genau einer Story pro Zeile. Dabei werden die Texte in Großbuchstaben überführt, sodass sie mit der buchstabenbasierten Verarbeitung der Chiffren kompatibel bleiben. Zusätzlich entfernt ein nachgelagerter Bereinigungsschritt hochfrequente, invariante Texteinstiege wie „Once upon a time“. Ohne diese Maßnahme bestünde die Gefahr, dass das Modell den Schlüssel primär anhand charakteristischer Satzanfänge errät, anstatt die mathematische Permutationslogik der Chiffre zu erlernen. Am Ende der Aufbereitung werden die bereinigten Geschichten in einer zentralen SQLite-Datenbank gespeichert.

Die Dateivorverarbeitung endet damit bewusst vor der eigentlichen Vokabularprojektion. Zeichen außerhalb des Alphabets A–Z verbleiben zunächst in den normalisierten Textdateien und werden nicht bereits in `prepare.py` entfernt. Die eigentliche Filterung und Vokabularabbildung wird somit in den Tokenizer (siehe Abschnitt 3.2.1) verlagert. Dies gewährt der Pipeline ein hohes Maß an Erweiterbarkeit, falls zusätzliche Token (wie Ziffern) in das Vokabular integriert werden sollen.

Die Daten werden deterministisch aufgeteilt, um Data Leakage auszuschließen:

- **Trainings- und Validierungs-Splits:** Sie werden aus `TinyStories-train` im Verhältnis 80 : 20 erzeugt.
- **Finaler Test-Split:** Er wird ausschließlich aus der separaten Datei `TinyStories-valid` generiert.

Diese strikte Trennung stellt sicher, dass die Generalisierungsleistung des Modells ausschließlich an Texten evaluiert wird, die zu keinem Zeitpunkt in die Gradientenoptimierung eingegangen sind.

3.1.2 Schlüsselgenerierung

Nach der Textaufbereitung erfolgt die Generierung der Zielklassen (siehe Abb. 7 in Lila markiert). Für die *Caesar-Chiffre* ist dieser Prozess trivial, da die vollständige Schlüsselmenge aus lediglich 26 diskreten Verschiebungsfaktoren ($k \in \{0, \ldots, 25\}$) besteht (siehe 2.2). Für die *Enigma-Maschine* hingegen ist eine vollständige Abdeckung aufgrund des großen theoretischen Schlüsselraums ausgeschlossen (siehe 2.3). Daher erzeugt die Pipeline zunächst eine Enigma-Schlüsseltabelle, deren Einträge vollständige Enigma-Schlüssel

enthalten. Diese Enigma-Schlüssel werden gezielt nach der Anzahl der Steckerpaare aufgeteilt. Die generierten Enigma-Schlüssel werden über `pair_count`, also die Anzahl der Steckerpaare, balanciert. Die Obergrenze wird über `PRECOMPUTE_PLUGBOARD_UPPER_LIMIT` konfiguriert und kann bei Bedarf zusätzlich über `INCLUDE_ZERO_PAIRS` beeinflusst werden. Für jedes Steckerpaaren werden bis zu `MAX_KEYS_PER_PAIR_COUNT` Enigma-Schlüssel[5] erzeugt.

Die Randomisierung der Enigma-Maschinenparameter ist bewusst getrennt aufgebaut. Separate Konfigurationsflags steuern, ob Walzenlage, Umkehrwalze, Ringstellung und Grundstellung variiert oder mit festen Standardwerten belegt werden. Dies ermöglicht, die Schwierigkeit der Aufgabe gezielt zu steuern, indem die Enigma-Maschinenparameter individuell angepasst werden können, z.B. durch die Fixierung der Ringstellung oder lediglich randomisierte Walzenlage. Werden die Enigma-Maschinenparameter randomisiert, kann zusätzlich über `SETTINGS_PLUGBOARD_RATIO` gesteuert werden, wie viele unterschiedliche Kombinationen aus Enigma-Schlüsseln pro Steckerbrettmuster erzeugt werden. Dies ermöglicht eine feinere Kontrolle über die Vielfalt der erzeugten Schlüssel, die das Modell zu sehen bekommt. Der fertige Schlüssel wird anschließend als standardisierter JSON-String abgespeichert. Dieses JSON-Objekt enthält die Walzenlage, die Umkehrwalze, die Ringstellung, die Grundstellung, die Anzahl der Steckerpaare sowie die eigentlichen Steckerbrettpaare.

```python
# Separate Flags fuer die Randomisierung einzelner Enigma-Maschinenparameter
RANDOMIZE_ROTORS: bool = True
RANDOMIZE_REFLECTOR: bool = True
RANDOMIZE_RING_SETTINGS: bool = False
RANDOMIZE_START_POSITIONS: bool = False

AVAILABLE_ROTORS: list[str] = ["I", "II", "III", "IV", "V", "VI", "VII", "VIII"]
AVAILABLE_REFLECTORS: list[str] = ["B", "C"]
RING_SETTINGS_RANGE: tuple[int, int] = (1, 26)
START_POSITIONS_ALPHABET: list[str] = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")

DEFAULT_ROTORS: str = "I II III"
DEFAULT_REFLECTOR: str = "B"
DEFAULT_RING_SETTINGS: str = "None"
DEFAULT_START_POSITIONS: str = "A A A"

SETTINGS_PLUGBOARD_RATIO = 1
MAX_KEYS_PER_PAIR_COUNT = 50_000
INCLUDE_ZERO_PAIRS: bool = False
PRECOMPUTE_PLUGBOARD_UPPER_LIMIT = 3
```

[5] Der Enigma-Schlüssel ist ein strukturiertes JSON-Objekt, das sowohl die Enigma-Maschinenparameter als auch die Steckerbrettpaare enthält.

Listing 2: Verfügbare mechanische Parameter einer 3-Rotor-Enigma-Chiffriermaschine aus `prepare_config.py`

Um Redundanzen und Mehrfachrepräsentationen desselben Steckerpaares zu vermeiden, werden die Buchstabenpaare bereits bei der Schlüsselerzeugung kanonisch sortiert und erst danach in den JSON-Payload übernommen. Dadurch wird sichergestellt, dass etwa `AV` und `VA` nicht als zwei verschiedene Enigma-Schlüsselvarianten persistiert werden (siehe Listing 3).

```
letters = rng.permutation(alphabet).tolist()
pairs: list[str] = []
for i in range(0, pair_count * 2, 2):
    pair = "".join(sorted((letters[i], letters[i + 1])))
    pairs.append(pair)

pairs.sort()
return " ".join(pairs)
```

Listing 3: Kanonische Sortierung von Steckerpaaren bei der Schlüsselerzeugung aus `prepare.py`

3.1.3 Verschlüsselung und Persistierung

Die Generierung der Trainingsbeispiele verschlüsselt die aufbereiteten Klartexte mit den erzeugten Enigma-Schlüsseln. Dabei wird jedoch bewusst kein vollständiges Kreuzprodukt zwischen allen Klartexten und allen Schlüsseln gebildet. Ein solches Vorgehen wäre wegen der großen Anzahl möglicher Kombinationen zu teuer und würde viele redundante Beispiele erzeugen. Stattdessen wird bereits in der Vorabberechnung eine Abbildung aus Story- und Enigma-Indizes[6] erstellt. Diese Abbildung legt fest, welcher Klartext mit welchem Schlüssel kombiniert wird.

Die Zusammensetzung dieser Paare ist wichtig, weil sie die Datenverteilung im späteren Training direkt beeinflusst. Ohne zusätzliche Steuerung könnten manche Anzahlen von Steckerpaaren im Datensatz zu oft vorkommen, andere dagegen nur selten. Das würde das Lernverhalten des Modells verzerren. Eine Hilfsroutine verteilt die geforderte Anzahl Beispiele daher zunächst möglichst gleichmäßig über alle aktivierten `pair_count`-Klassen. Erst danach wird die gewählte Sampling-Strategie angewendet. Auf diese Weise bleibt die Verteilung der Anzahl der Steckerpaare kontrollierbar, ohne dass kleine Klassen im Gesamtkorpus untergehen.

Zusätzlich unterstützt die Pipeline drei Sampling-Strategien und passende Verhältnisparameter (siehe Listing 4). Sie wurden eingeführt, um gezielt zu steuern, welche Art von

[6] Indizes sind die fortlaufenden Integer-IDs der Klartexte bzw. Schlüssel in der SQLite-Datenbank.

Vielfalt im Datensatz stärker betont wird, also eher unterschiedliche Klartexte oder eher unterschiedliche Enigma-Schlüssel. Die Evaluation, wie sich diese Strategien auf das Training auswirken, wird in Kapitel 4 dargelegt. Folgende Strategien stehen zur Auswahl:

- **Random Sampling:** Erzeugt rein zufällige Kombinationen aus Klartexten und Enigma-Schlüsseln. Diese Strategie dient als einfache Referenz ohne zusätzliche strukturelle Gewichtung.
- **Text Focus Sampling:** Wählt zunächst eine kleinere Menge von Klartexten aus und kombiniert jeden dieser Texte mit mehreren unterschiedlichen Enigma-Schlüsseln. Dadurch steigt die Schlüsselvielfalt pro Klartext.
- **Key Focus Sampling:** Wählt zunächst eine kleinere Menge von Enigma-Schlüsseln aus und kombiniert jeden dieser Schlüssel mit mehreren unterschiedlichen Klartexten. Dadurch steigt die Textvielfalt pro Schlüssel.

```
PRECOMPUTE_ENCRYPT_STRATEGY = "key_focus"
PRECOMPUTE_ENCRYPT_RATIO = 50
PRECOMPUTE_SAMPLE_SIZES = {
        "train": 1_000_000,
        "valid":   100_000,
        "test":    100_000,
    }
```

Listing 4: Sampling-Strategien und Verhältnisparameter aus `prepare_config.py`

Eine zentrale Implementierungsentscheidung betrifft die Datenübergabe an das Modell (siehe Abb. 7 in Blau markiert). Erste Prototypen führten die Chiffrierung *on-the-fly* auf der CPU während des Trainings aus. Das erzeugte einen deutlichen CPU-Flaschenhals. Während dieses Vorgehen für die Caesar-Chiffre noch vertretbar wäre, ist es für die Enigma-Maschine zu teuer. Die Simulation drosselte den Datenfluss so stark, dass die A100-GPUs wiederholt auf neue Daten warten mussten (*GPU Idling*). Die Vorabberechnung verschiebt diesen Aufwand in die Vorbereitung und erhöht damit zwar den Speicherbedarf, entlastet aber den Trainingspfad deutlich. Dadurch ist das Experimentieren verschiedener Konfigurationen der Datenzusammensetzung und -größe jedoch deutlich langsamer möglich, da vor jedem neuen Training die Daten erneut vorbereitet werden müssen.

Die finale Zuordnung der Beispiele zu den Daten-Splits (*Train*, *Valid*, *Test*) erfolgt ebenfalls deterministisch in der Vorberechnung. Die Split-Informationen werden direkt in der SQLite-Datenbank persistiert und später von der Klasse `SQLDataset` ohne zusätzlichen Rechenaufwand gelesen. Die Daten werden über einen `ProcessPoolExecutor` parallel verschlüsselt und als BLOBs (*Binary Large Objects*) in einer zentralen SQLite-

Datenbank abgelegt. Die Logik der Worker liegt in dem Modul `precompute_worker.py` und sorgt dafür, dass der Prozesspool auf Windows- und Cluster-Systemen über den `spawn`-Mechanismus stabil initialisiert werden kann. Jeder Worker instanziiert genau einmal ein `EnigmaCipher`-Objekt und serialisiert die verschlüsselten Token, die Maschinenparameter-Tensoren und die Multi-Label-Zielvektoren direkt in BLOB-Form. So bleibt der Datenladepfad kompakt und eine erneute Chiffrierung zur Trainingszeit entfällt.

3.2 Chiffren-Abstraktion und -Implementierung

Um eine modulare Codebasis für das Training zu gewährleisten, sind die Chiffren als Python-Klassen realisiert (siehe Abb. 8), wobei eine abstrakte Oberklasse als Basis dient. So kann die chiffrenspezifische Logik vollständig aus der allgemeinen Trainingspipeline herausgelöst werden.

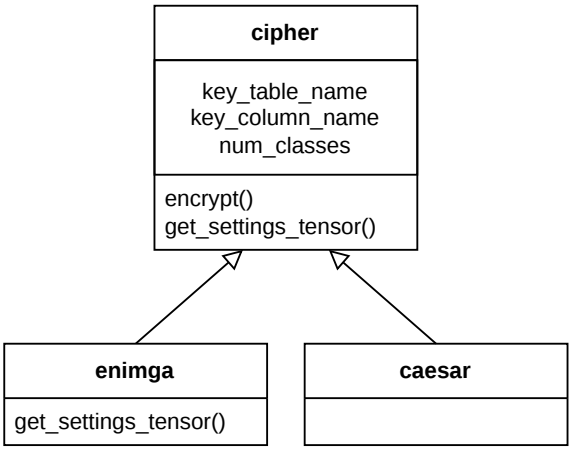

Abb. 8: UML-Diagramm der Chiffren-Klassenhierarchie.

3.2.1 Verschlüsselung und zeichenbasierte Tokenisierung

Die zentrale Methode `encrypt` nimmt einen Klartext sowie einen Enigma-Schlüssel entgegen und gibt die verschlüsselten Daten sowie den dazugehörigen Enigma-Schlüssel zurück. Während die Caesar-Chiffre die mathematische Modulo-Addition nativ in Python berechnet, wird für die mechanische Simulation der Enigma-Rotoren auf die Open-Source-Bibliothek `py-enigma` [17] zurückgegriffen.

Da beide Chiffren auf Buchstabenebene operieren, erfolgt die Überführung der Textdaten in den Tensorraum über eine strikt zeichenbasierte Tokenisierung (*Character-Level Tokenization*). Eine moderne, wort- oder subwortbasierte Tokenisierung würde die deterministische, positionsabhängige Mechanik der Enigma-Rotoren verschleiern und die Mustererkennung erschweren (siehe Abschnitt 2.2 und 2.3). Der eingesetzte `CharTokenizer` arbeitet mit einem 26-symboligen Vokabular der Großbuchstaben A bis Z und weist jedem Zeichen einen eindeutigen Integer-Index zu, also etwa `A` $\mapsto 0$, `B` $\mapsto 1$, $\ldots$, `Z` $\mapsto 25$. Zeichen, die nicht im definierten Zielvokabular enthalten sind, werden während der Tokenisierung verworfen. An dieser Stelle erfolgt die Reduktion auf das Alphabet des Modells: In der Caesar-Klasse vor der Verschiebung des Klartexts und in der Enigma-Klasse nach der Verschlüsselung des Textes. Das für die Batch-Bildung benötigte Padding-Token gehört hingegen nicht zum Vokabular der Chiffren selbst, sondern wird erst nachgelagert im DataLoader- und Modellpfad als zusätzlicher Index eingeführt.

3.2.2 Zustandskodierung der Maschinenparameter

Da sich diese Arbeit mit der Kryptoanalyse des Enigma-Steckerbretts beschäftigt, ist es notwendig, den mechanischen Ausgangszustand der Maschine an das Modell zu übermitteln. Über die Methode `get_settings_tensor` wird der aktuelle Zustand der Enigma-Maschinenparameter (exklusive des Steckerbretts) in einen 10-dimensionalen ganzzahligen Tensor[7] transformiert:

$$\mathbf{key} = [r_1, r_2, r_3, u, g_1, g_2, g_3, p_1, p_2, p_3] \tag{9}$$

Das Setup aus Listing 2 basiert dabei auf der historischen Simulation einer Drei-Walzen-Enigma. Dieser Einstellungsvektor (siehe Gleichung 9) kodiert die IDs der drei aktiven Walzen ($r_i \in \{I \ldots VIII\}$), die ID der Umkehrwalze ($u \in \{B, C\}$), die drei numerischen Werte der Ringstellung ($g_i \in \{1 \ldots 26\}$) sowie die initialen Werte der Grundstellung ($p_i \in \{1 \ldots 26\}$), die innerhalb der Konfiguration in Listing 2 definiert sind.

3.2.3 Ziel-Label-Kodierung

Neben der `encrypt`-Methode liefert `get_label_tensor` die mathematische Repräsentation der gesuchten Zielgröße (Ground Truth).

[7] Ein Tensor ist eine mehrdimensionale Datenstruktur, die verwendet wird, um Daten effizient zu speichern und zu verarbeiten.

Caesar-Chiffre Da der Schlüssel aus einer einzelnen, exklusiven Verschiebung $k \in \{0, \ldots, 25\}$ besteht, beläuft sich die Klassenzahl auf `num_classes = 26`. Das Label wird folglich als klassischer *One-Hot-Vektor* [7, Kap. 2, 13] der Länge 26 kodiert, bei dem exakt an der Position der korrekten Verschiebung eine Eins steht.

Enigma-Maschine Die zu erlernende Zielgröße ist die Belegung des Steckerbretts. Da bis zu zehn Steckerkabel simultan aktiv sein können, liegt ein Multi-Label-Klassifikationsproblem vor [24]. Ein exklusiver One-Hot-Vektor ist hierfür unzureichend; stattdessen wird ein *Multi-Hot-Vektor* [7, Kap. 3, 10] generiert. Die Dimension des Zielvektors entspricht der Anzahl aller theoretisch möglichen, ungeordneten Buchstabenpaare. Mathematisch definiert sich dieser Raum über den Binomialkoeffizienten aus dem 26-teiligen Alphabet ohne Zurücklegen:

$$\binom{26}{2} = \frac{26!}{2!(26-2)!} = 325 \tag{10}$$

Die finale Klassenzahl für das Enigma-Ausgabe-Label beträgt somit `num_classes = 325`. Jeder Index des Tensors repräsentiert ein spezifisches, alphabetisch sortiertes Paar, beginnend bei Index 0 für `AB` bis hin zu Index 324 für `YZ`. Die programmiertechnische Überführung der extrahierten Steckerpaare in diese Tensorrepräsentation ist in Listing 5 dargestellt.

```
1
2  label = torch.zeros(self.num_classes, dtype=torch.float32)
3  if plugboard_pairs:
4      pairs = plugboard_pairs.split()
5      for pair in pairs:
6          # [...]
7
8          # Konvertiere die Buchstaben in Indizes
9          idx_a = self._char_to_idx[char_a]
10         idx_b = self._char_to_idx[char_b]
11
12         # Hole die Position der beiden Buchstaben im Alphabet
       und sortiere sie
13         i, j = (idx_a, idx_b) if idx_a < idx_b else (idx_b,
       idx_a)
14
15         # Hole den Index in der oberen Dreiecksmatrix an der
       Position (i, j)
16         tri_idx = self._upper_triangle_index(i, j)
17         label[tri_idx] = 1.0
18
19 return label
```

Listing 5: Abbildung von Steckerpaaren auf einen Multi-Hot-Vektor aus `enigma.py`

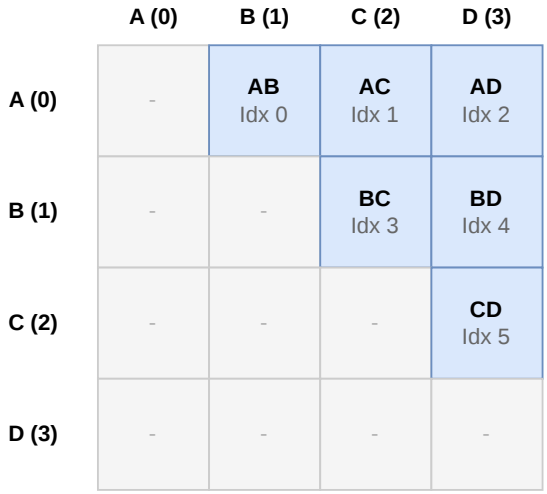

Abb. 9: Visualisierung einer 4×4-Dreiecksmatrix für das Alphabet $\{A, B, C, D\}$.

Um die ungeordneten Buchstabenpaare auf den eindimensionalen Ausgabe-Tensor abzubilden, nutzt der Algorithmus das Prinzip einer oberen Dreiecksmatrix (siehe Abb. 9). Wie am Beispiel der Abbildung 9 beschreibt die folgende Formel:

$$\binom{4}{2} = 6$$

aus Gleichung 10 die Anzahl der möglichen Steckerbrettpaare. Die Berechnung der flachen 1D-Indexposition aus den zweidimensionalen Koordinaten (i, j) erfolgt über die interne Hilfsfunktion `_upper_triangle_index` (siehe Listing 6). Dadurch wird sichergestellt, dass das Modell keine isolierten Einzelzeichen erlernt, sondern die Aktivierung des gesamten kombinatorischen Steckerraums erfasst.

```
1 def _upper_triangle_index(self, i: int, j: int) -> int:
2     n = self.alphabet_size
3     return i * (n - 1) - (i * (i - 1)) // 2 + (j - i - 1)
```

Listing 6: Berechnung des Dreiecksmatrix-Index aus den 2D-Koordinaten aus `enigma.py`

3.3 Dataset & DataLoader

Die Schnittstelle zwischen der vorberechneten SQLite-Datenbank und der Modellarchitektur wird durch ein PyTorch-Dataset realisiert. Der Vorteil liegt darin, dass die Datensätze effizient durch einen PyTorch-DataLoader ausgelesen und in der vom Modell erwarteten Tensorform bereitgestellt werden können.

Das `SQLDataset` kapselt diesen Ladeprozess. Über den Parameter `db_split` greift die Instanz gezielt auf die entsprechenden Teilmengen (*train*, *valid* oder *test*) der Datenbank zu. Ein optionaler Parameter `train_combinations` erlaubt zudem die künstliche Limitierung der genutzten Datenmenge für kontrollierte Skalierungsexperimente. Ein einzelnes Datenelement (*Sample*) besteht dabei aus den verschlüsselten Text-Token (`input_ids`), dem Zielvektor (`label`) und dem Maschinenparameter-Vektor (`settings`).

Für einen hohen Datendurchsatz wurden zwei zentrale Optimierungsstrategien in das `SQLDataset` integriert:

I/O- und Laufzeitoptimierungen Um beim parallelen Datenzugriff I/O-Kollisionen und Sperren zwischen asynchronen Worker-Prozessen auszuschließen, wird die SQLite-Verbindung strikt über eine schreibgeschützte URI (`file:{path}?mode=ro`) initialisiert. Beim Instanziieren der Klasse werden lediglich die Zeilen-IDs der Datenbank im RAM hinterlegt. Dies ermöglicht ein hocheffizientes Zeilen-Mapping in $\mathcal{O}(1)$, ohne die großen Daten-BLOBs vorab laden zu müssen. Anstatt für jedes Sample eine isolierte SQL-Abfrage an die Festplatte zu senden, nutzt das Dataset die PyTorch-spezifische `__getitems__`-Schnittstelle. Alle angeforderten Indizes eines vollständigen Batches werden aggregiert und über eine einzige, gebündelte Datenbankabfrage (siehe Listing 7) extrahiert.

```
# Batchweise Abfrage der Datenbank mit den gesammelten
    Zeilen-IDs
self._cursor.execute(f"SELECT id, input_ids, label, settings
    FROM precomputed_samples WHERE id IN ({placeholders});",
    row_ids)

# Rekonstruktion der Tensoren aus den geladenen BLOBs
input_ids = torch.from_numpy(
    np.frombuffer(raw_input, dtype=np.int64).copy()
)
settings = torch.from_numpy(
    np.frombuffer(raw_settings, dtype=np.int64).copy()
)
label = torch.from_numpy(
    np.frombuffer(raw_label, dtype=np.float32).copy()
)
```

Listing 7: Batchweises Laden und Rekonstruieren der Tensoren aus `dataset.py`

Wie in Listing 7 gezeigt, werden diese Rohdaten beim Laden mittels `np.frombuffer` zunächst effizient aus den BLOBs rekonstruiert und anschließend per `copy()` in für PyTorch schreibbare Arrays überführt. Die `input_ids` und der `settings`-Vektor liegen dabei als `int64`-Tensoren vor, um mit den Embedding-Schichten des Modells kompatibel

zu sein. Die Zielvektoren (`label`) werden als `float32`-Tensoren für die spätere Loss-Berechnung via BCE-Loss bereitgestellt.

Batch-Bildung und Multiprocessing Die Koordination der Batches übernimmt der PyTorch-*DataLoader*. Da Samples variierende Längen aufweisen, füllt die Funktion `pad_collate_fn` die Sequenzen dynamisch bis zur Maximallänge des jeweiligen Batches auf. Auf diese Weise erhält das Modell einen einheitlich geformten Eingabetensor. Als Padding-Wert wird dabei ein zusätzlicher Token-Index verwendet, der aus der Vokabulargröße abgeleitet und später im Modell konsistent als `padding_token_id` behandelt wird.

Um die Hardware-Ressourcen effizient zu nutzen, werden dem *DataLoader* erweiterte Multiprocessing-Attribute übergeben. Mit `pin_memory=True` wird der Datentransfer in den GPU-Speicher beschleunigt. Über den `spawn`-Kontext in Kombination mit `persistent_workers=True` wird sichergestellt, dass die initialisierten Worker-Prozesse zwischen den Epochen im RAM verbleiben. Der zeitaufwändige Overhead für das wiederholte Neuinstanziieren der Datenbankverbindungen wird dadurch eliminiert.

3.4 Modellarchitektur

Das Modell implementiert eine *Encoder-Only-Transformer-Architektur*, bei der nur der Encoder-Teil der Transformer-Architektur aus Abschnitt 2.5 verwendet wird. Da die Kryptoanalyse in dieser Arbeit als Klassifikationsproblem modelliert wird, erzeugt der Encoder zunächst eine kontextualisierte Repräsentation der Eingabe. Erst der anschließende Klassifikations- beziehungsweise Prediction-Head projiziert diese auf die festen Zielklassen und gibt deren Wahrscheinlichkeiten aus. Im Gegensatz zu einem generativen Ansatz, bei dem das Modell eine variable Token-Sequenz erzeugt, erhält der Klassifikator eine feste Anzahl von Zielklassen. Der generative Decoder-Teil wird daher nicht benötigt. Der Vorwärtslauf (*Forward Pass*) des Modells wird in Abb. 10 schematisch dargestellt.

3.4.1 Einbettung und Verdichtung der Maschinenparameter

Die bekannten Enigma-Maschinenparameter (Walzenlage, Umkehrwalze, Ringstellung und Grundstellung) werden zunächst als zehn diskrete Integerwerte (siehe Gleichung 9) über eine separate Embedding-Schicht (`key_emb`) übergeben. Anders als in einer reinen Präfix-Strategie verbleiben diese zehn Einzelrepräsentationen jedoch nicht als eigene Token in der Sequenz. Stattdessen werden sie zu einem flachen Vektor konkateniert und über eine Projektionsschicht zu einem globalen Maschinenparameter-Token verdichtet.

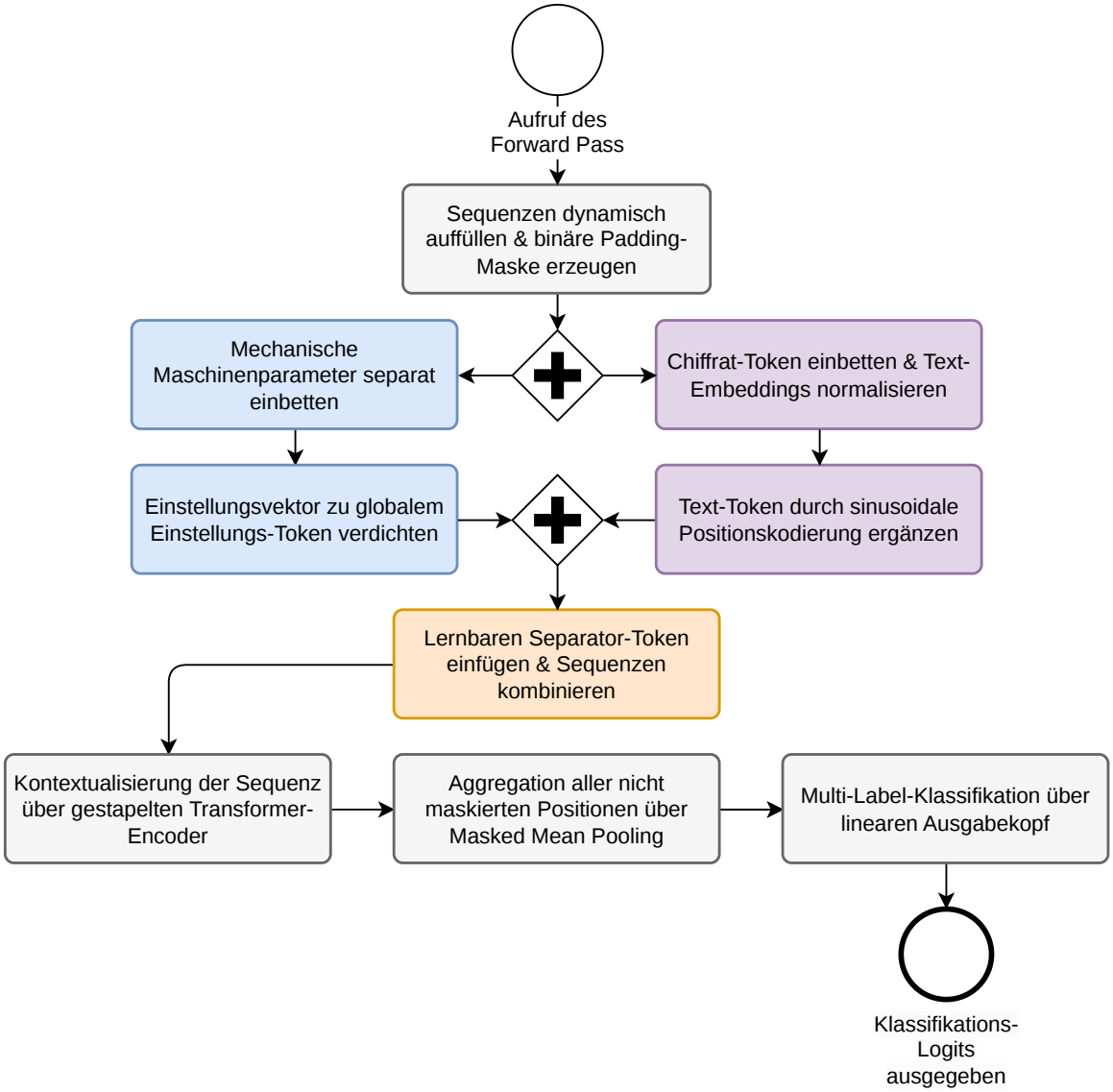

Abb. 10: Visualisierung des Forward Pass des Modells aus `model.py`.

Dieser Token fasst die Enigma-Maschinenparameter kompakt zusammen und gibt dem Modell einen globalen Maschinenparameterkontext, statt zehn einzelne Tokens lernen zu müssen. Zusätzlich wird ein lernbarer Separator-Token eingefügt, der die Grenze zwischen globalem Maschinenparameterkontext und Chiffrat markiert (siehe Listing 8). Der Separator-Token ist ein trainierbarer Vektor, der dem Modell signalisiert, dass die nachfolgenden Tokens den Chiffrat-Text repräsentieren. Diese Trennung erleichtert dem Modell die Unterscheidung zwischen den beiden Informationen.

```
text_tokens = self.pos_encoder(self.text_norm(self.text_emb(x)))

key_embeds = self.key_emb(key_info)
flat_keys = key_embeds.view(x.size(0), -1)
key_token = self.key_compressor(flat_keys).unsqueeze(1)

sep = self.sep_token.expand(x.size(0), -1, -1)
combined = torch.cat([key_token, sep, text_tokens], dim=1)

encoded = self.transformer_encoder(combined,
    src_key_padding_mask=combined_mask)
pool_mask =
    (~combined_mask).unsqueeze(-1).expand_as(encoded).float()
pooled = torch.sum(encoded * pool_mask, dim=1) /
    torch.clamp(pool_mask.sum(dim=1), min=1e-9)
```

Listing 8: Ausschnitt der `forward`-Methode aus `model.py` mit Maschinenparameter-Token, Separator und Pooling

3.4.2 Padding, Maskierung und Aggregation

Das eigentliche Padding des Geheimtextes erfolgt bereits während der Batch-Bildung im `DataLoader`. Im Modell wird dieser Zustand über einen zusätzlichen Token-Index außerhalb des Alphabets abgebildet, also `padding_token_id = vocab_size`. Dadurch bleibt das Vokabular der 26 Buchstaben unverändert, während die Padding-Positionen eindeutig markiert sind. Die resultierende Sequenzstruktur aus globalem Maschinenkontext, Separator, Chiffrat und Padding ist in Abbildung 11 dargestellt. Auf dieser Grundlage wird eine binäre Padding-Maske erzeugt. Sie stellt sicher, dass der Transformer-Encoder nur Auffüllpositionen ignoriert, nicht jedoch den vorangestellten Maschinenkontext. Nur Tokens mit einem Maskenwert von 1 werden in die Kontextualisierung einbezogen.

Nach der Kontextualisierung aggregiert das Modell alle nicht maskierten Positionen der kombinierten Sequenz mittels *Masked Mean Pooling*. Dabei wird die Summe der Token-Embeddings durch die Anzahl der gültigen, also nicht maskierten, Positionen geteilt. Das Ergebnis ist eine stabile Repräsentation, die weitgehend unabhängig von der Länge der Chiffratsequenz und dem Anteil des Paddings bleibt. Dieser zusammengefasste Vektor dient anschließend als Eingabe für den linearen Klassifikationskopf.

Die finale Vorhersage der Zielklassen erfolgt über eine lineare Projektion des gepoolten Vektors auf die Dimension der Zielklassen. Die Ausgabe ist ein Logit-Vektor[8] , der

[8] Der Logit x_i ist der rohe, unskalierte Ausgabewert des Modells für die Klasse i. Er repräsentiert die Vorhersage vor der Transformation in eine Wahrscheinlichkeit, dass das Paar i aktiv ist. Er kann Werte zwischen $-\infty$ und $+\infty$ annehmen, wobei ein höherer Wert für eine höhere Wahrscheinlichkeit des Modells steht.

die Wahrscheinlichkeit jeder Klasse vor der Anwendung einer Aktivierungsfunktion darstellt.

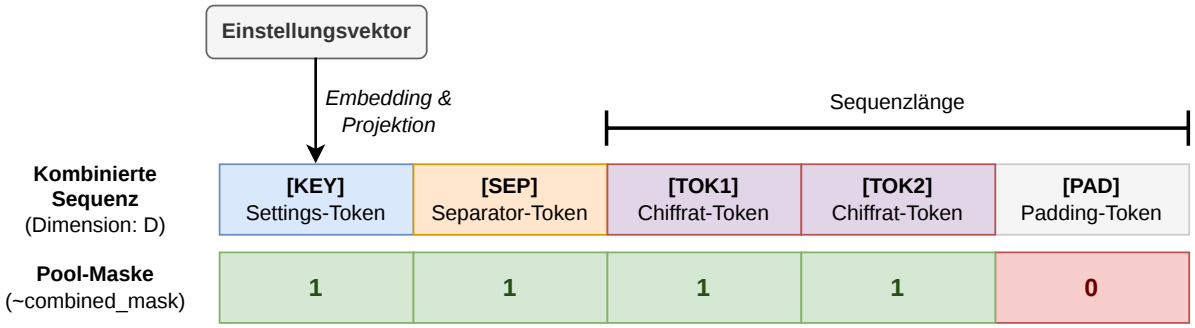

Abb. 11: Visualisierung des kombinierten Eingabevektors mit globalem Maschinenkontext, Separator-Token, Chiffrat-Token und Padding.

3.5 Modelltraining

Der Trainingsprozess wird in Abb. 12 visualisiert. Die Orchestrierung und Parallelisierung des Trainingsprozesses wird über das *Ray Tune*-Framework [14] realisiert. Die eigentliche Trainingsschleife folgt dem üblichen PyTorch-Ablauf [20]: Zunächst werden die Gradienten mit `optimizer.zero_grad()` zurückgesetzt. Danach folgen Vorwärtslauf, Loss-Berechnung, Backpropagation mittels `loss.backward()` und schließlich die Aktualisierung der Modellparameter durch Optimierer und Lernraten-Scheduler.

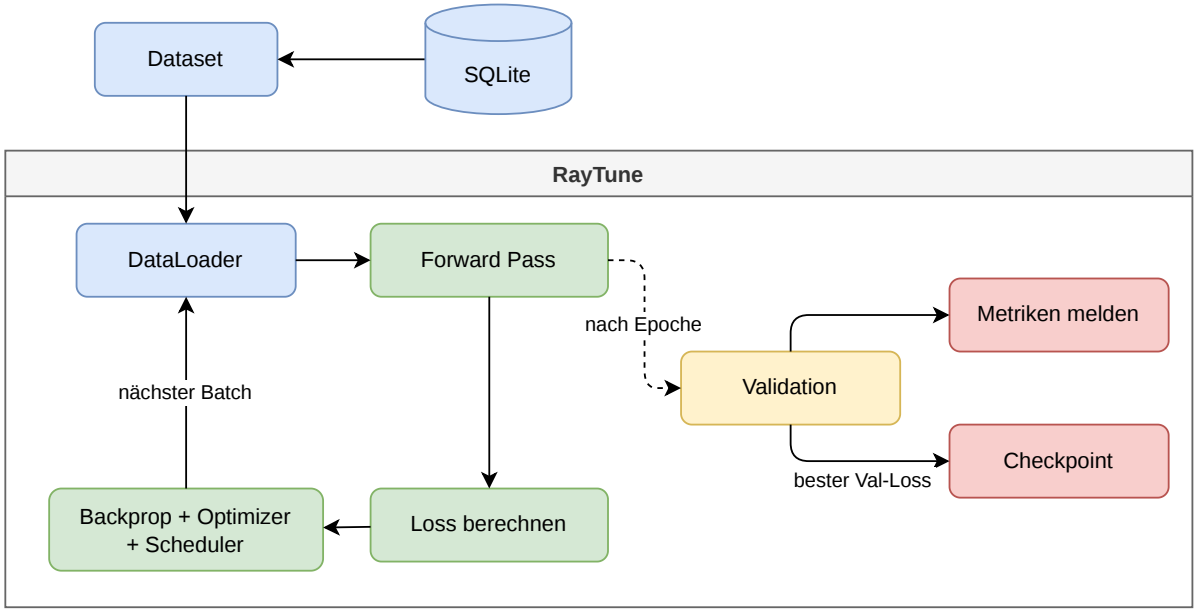

Abb. 12: Visualisierung der Trainingspipeline aus `train.py`.

3.5.1 Loss- und Aktivierungsfunktion

Die Vorhersage der Steckerpaare wird als Multi-Label-Klassifikationsproblem formuliert. Da das Steckerbrett der Enigma-Maschine bis zu zehn reziproke Buchstabenpaare vertauscht, müssen aus den 325 theoretisch möglichen Paarungen des 26-teiligen Alphabets jene identifiziert werden, die in der aktuellen Konfiguration aktiv sind. Jede der 325 Zielklassen wird damit als eigenes binäres Klassifikationsproblem behandelt.

Zur Optimierung dieses spärlich besetzten Zielraums wird `BCEWithLogitsLoss` verwendet. Die mathematische Formulierung für ein einzelnes Trainingsbeispiel lautet:

$$\mathcal{L}_{\mathrm{BCE}} = -\frac{1}{M}\sum_{i=1}^{M}\Big[y_i \log\Big(\sigma(x_i)\Big) + (1-y_i)\log\Big(1-\sigma(x_i)\Big)\Big] \tag{11}$$

Hierbei bezeichnet $M = 325$ die Anzahl der Klassen, $y_i \in \{0, 1\}$ das binäre Target für das Vorhandensein eines Steckerpaars i. Das Modell gibt für jede Klasse einen reellen Logit-Wert $x_i \in \mathbb{R}$ aus. Die integrierte Sigmoid-Funktion $\sigma(x_i)$ transformiert diese Logits intern wie folgt:

$$\sigma(x_i) = \frac{1}{1+e^{-x_i}} \tag{12}$$

Sie verhindert numerische Instabilitäten wie Unterläufe bei sehr kleinen Wahrscheinlichkeiten ($x_i \to -\infty$) oder Überläufe bei großen Logits ($x_i \to \infty$).

Da der Zielvektor aufgrund des Limits von maximal zehn aktiven Steckerpaaren stark spärlich besetzt ist, besteht die Gefahr, dass das Modell pauschal Nullen vorhersagt. Um diesem Klassenungleichgewicht entgegenzuwirken, wird der Gewichtungsfaktor p_i (`pos_weight`) eingeführt und an `BCEWithLogitsLoss` übergeben. Er skaliert den Loss bei Fehlklassifikationen tatsächlich vorhandener Verbindungen ($y_i = 1$) stärker:

$$p_i = \frac{N_{\mathrm{neg}}}{N_{\mathrm{pos}}} \tag{13}$$

wobei N_{neg} die Anzahl der inaktiven und N_{pos} die Anzahl der aktiven Steckerpaare bezeichnet. In der aktuellen Implementierung wird dieser Faktor als skalarer Hyperparameter `pos_weight` in der Trainingskonfiguration hinterlegt.

Die Berechnung des Gradienten erfolgt aus Effizienzgründen nicht über den gesamten Datensatz, sondern über *Mini-Batch Gradient Descent*. Dazu wird der Datensatz in Teil-

mengen der Größe B unterteilt. Der für den Optimierungsschritt genutzte Gradient ∇L wird als Mittelwert über den aktuellen Mini-Batch approximiert. Dieses Verfahren erlaubt eine effiziente Vektorisierung auf modernen GPUs. Zugleich wirkt das stochastische Rauschen der Mini-Batches als natürlicher Regularisierungseffekt.

3.5.2 Optimizer, Scheduler und Präzision

Für die Gewichtsanpassung wird der *AdamW*-Optimierer [15] mit entkoppeltem Weight Decay verwendet. Er gilt als etablierter Standard für Transformer-basierte Modelle und eignet sich hier, weil er adaptive Lernraten mit einer separaten Regularisierung der Gewichte verbindet.

Die konkrete Implementierung arbeitet hierbei mit Mini-Batches aus dem `DataLoader`. Der pro Batch berechnete Gradient wird somit nicht über den gesamten Datensatz, sondern über die jeweilige Teilmenge approximiert. Zur Vermeidung explodierender Gradienten[9] sowie zur zusätzlichen Stabilisierung des Trainings wird vor jedem Optimierungsschritt ein norm-basiertes Gradienten-Clipping [19] mit einem maximalen Schwellenwert von $\theta = 10$ durchgeführt. Dies lässt große Gradientenwerte schrumpfen, aber erlaubt noch größere Lernschritte.

Die Steuerung der Basis-Lernrate η folgt einem zweiphasigen Zeitplan. Im dargestellten Beispiel (siehe Abbildung 13) steigt die Lernrate in der ersten Phase, dem linearen Warmup, über die ersten 10 % der globalen Trainingsschritte von null auf die Basislernrate an. Dadurch werden instabile Updates zu Beginn des Trainings vermieden. In der zweiten Phase fällt die Lernrate entlang einer halben Kosinus-Kurve (*Cosine Decay*) auf eine minimale Lernrate ab. So bleibt das Training anfangs beweglich und wird gegen Ende feiner abgestimmt. Die beispielhafte Darstellung dieses Schemas ist in Abbildung 13 visualisiert[10]. Die ersten `warmup_factor` × `total_steps` Schritte werden für den linearen Anstieg der Lernrate genutzt; anschließend folgt der monotone Abfall über Cosine-Decay bis zur minimalen Lernrate.

[9] Explodierende Gradienten treten auf, wenn die Ableitungen der Loss-Funktion während des Backpropagation-Prozesses sehr groß werden. Dies kann dazu führen, dass die Gewichte des Modells instabil werden und der Trainingsfortschritt verloren geht.

[10] Gemini 3.1 Pro – Prompt: „Erstelle mir eine Grafik auf Basis dieser Methode `lr_lambda` (siehe Listing 9) und den Parametern `"base_learning_rate: 5e-4`, `mminimum_learning_rate: 1e-5`, `"weight_decay: 1e-3`, `"warmup_factor: 0.1`“, wobei `lr_lambda` aus `train.py` stammt

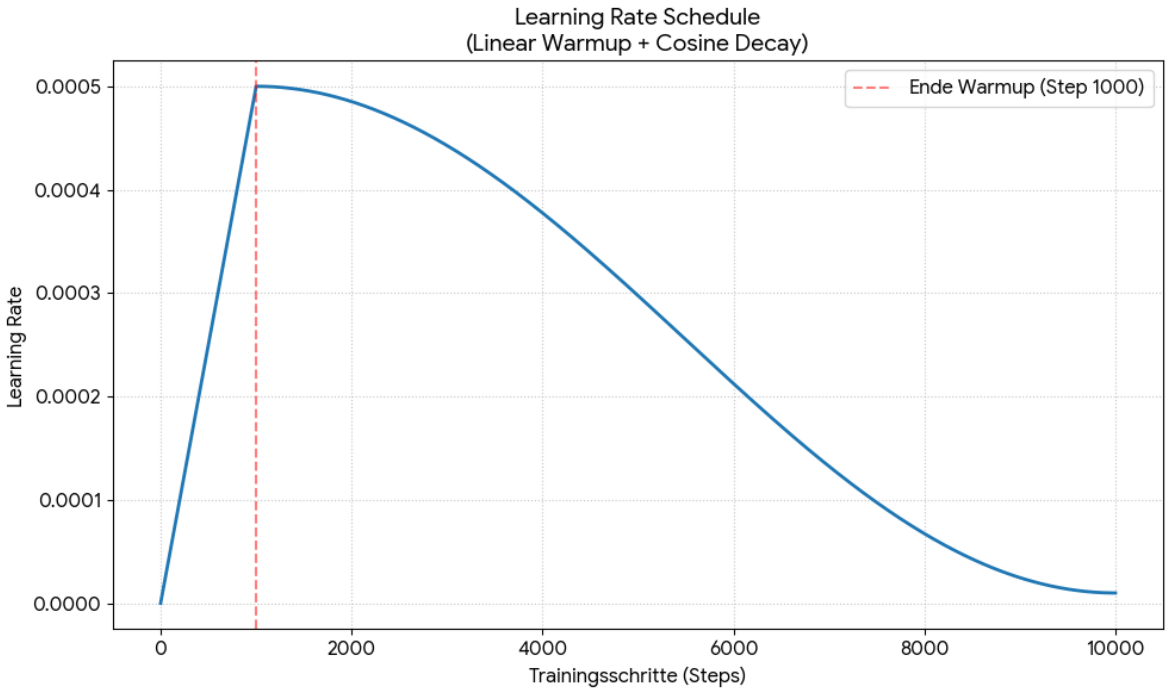

Abb. 13: Visualisierung des Lernraten-Schedulers mit linearem Warmup und Cosine-Decay.

```python
def lr_lambda(current_step: int) -> float:

    if current_step < warmup_steps:
        return float(current_step) / float(max(1, warmup_steps))

    progress = float(current_step - warmup_steps) / float(max(1, total_steps - warmup_steps))
    min_lr_factor = minimum_learning_rate / base_learning_rate
    cosine_decay = 0.5 * (1.0 + math.cos(math.pi * progress))

    return min_lr_factor + (1.0 - min_lr_factor) * cosine_decay
```

Listing 9: Funktion des Lernraten-Scheduler (`lr_lambda`) aus `train.py`

Zur Effizienzsteigerung während des Trainings auf dem GPU-Cluster wird *Automatic Mixed Precision* (AMP) eingesetzt. Rechenintensive Operationen, vor allem die Matrixmultiplikationen in den Attention-Layern, werden dabei ab der NVIDIA-Ampere-Generation automatisch von `float32` (FP32) in das kompaktere `bfloat16` (BF16)-Format umgestellt [18]. Dadurch sinken Rechenaufwand und Speicherbedarf, ohne dass der große Wertebereich von FP32 verloren geht [11]. Präzisionskritische Schritte, etwa in der Loss-Funktion, verbleiben hingegen im FP32-Format [16]. So wird das Training beschleunigt, ohne die numerische Stabilität des Modells wesentlich zu verschlechtern [11, 16].

4 Methodik und Ergebnisse

Dieses Kapitel beschreibt, wie die Modelle konfiguriert, trainiert und bewertet wurden. Die Projektarbeit unterteilt sich in zwei Phasen: Zunächst wird die Caesar-Chiffre als Vorversuch genutzt, um die grundsätzliche Lernfähigkeit der Transformer-Pipeline zu prüfen. Anschließend konzentriert sich die Untersuchung auf die polyalphabetische Enigma-Maschine.

4.1 Trainingsvalidierung mit Hilfe von Caesar

Als erstes wurde die Caesar-Chiffre untersucht. Dabei zeigte sich in der explorativen Phase, dass bereits kleine Transformer die lineare Verschlüsselung schnell erkannten. Der Loss sank und die Accuracy stieg nach nur wenigen Batches stark. Es näherte sich dabei 100 % Accuracy. Da das zentrale wissenschaftliche Interesse dieser Arbeit gemäß Abschnitt 1.2 auf der Enigma-Maschine liegt, wird die Caesar-Chiffre im Folgenden als abgeschlossene Vorphase behandelt. Hypothese 1 gilt damit als bestätigt. Im weiteren Verlauf steht daher die Prüfung von Hypothese 2 im Mittelpunkt.

4.2 Komplexitätssteigerung durch Enigma

Nachdem die grundsätzliche Funktionsweise der Trainingspipeline validiert wurde, begann die Untersuchung auf der Enigma-Maschine. Dabei zeigte sich, dass die Lernfähigkeit des Modells stark von der Komplexität der Enigma-Maschinenparameter abhängt.

4.2.1 Modellarchitektur und Hyperparameter

Zur Untersuchung der Lernfähigkeit des Transformers wurden für das Versuchsdesign Konstanten, Hyperparameter (unabhängige Variablen) und Evaluationsmetriken (abhängige Variablen) festgelegt.

Konstanten: Die über alle Experimente hinweg konstant gehaltenen Parameter sind in Tabelle 1 zusammengefasst.

Konstante	**Wert**	**Funktion**
Sequenzlänge	100	Definition der Kontextfenstergröße der Eingabe.
Batch-Größe	128	Balance zwischen stabiler Gradientenschätzung und GPU-Durchsatz.
Gradienten-Clipping	10.0	*Max_norm* zur Vermeidung explodierender Gradienten.
Dropout-Wert	0.0	Deaktiviert.
Positives-Gewicht	25	Gewichtung positiver Klassen in der Loss-Funktion.
Verhältnis d_{model} zu n_{hid}	1 : 4	Proportionalität der Feedforward-Kapazität (256 zu 1024).
Vokabulargröße	27	Alphabet (A–Z) zuzüglich des nachgelagerten Padding-Token.
Sampling-Strategie	*key_focus*	Erhöhung der Datenvielfalt pro Schlüssel.

Tab. 1: Invariante Konstanten der experimentellen Untersuchung.

Der *Dropout*-Wert wurde auf 0.0 gesetzt. Da die Enigma-Chiffre auf einer Folge von Transformationen beruht, die sich mit jedem Tastendruck durch die Fortschaltung der Rotoren verändert, ist die relative Position jedes Zeichens innerhalb der Sequenz wichtig. Das randomisierte Deaktivieren von Gewichten führte zu strukturellen Brüchen in den gelernten Netzen. In Vorversuchen induzierte jeglicher Dropout ausgeprägte Konvergenzinstabilitäten.

Das Verhältnis zwischen der Einbettungsdimension d_{model} und der verborgenen Dimension des Feedforward-Netzwerks (n_{hid}) wurde auf 1 : 4 beschränkt. Dies soll für ein gemeinsames Skalierungsverhältnis sorgen.

Die Sampling-Strategie *key_focus* zeigte in Vorversuchen, dass durch das Variieren der Klartexte für identische Enigma-Schlüssel das Modell die Zielzuordnung von dem Geheimtext und den bekannten Maschinenparametern auf das Steckerbrett besser approximieren kann. Daher wird diese als primäre Sampling-Strategie gewählt. In kleineren Vorversuchen zeigte sich, dass diese Strategie die Lernfähigkeit des Modells im Vergleich zu anderen Sampling-Strategien verbessert.

Hyperparameter (Unabhängige Variablen): Die Hyperparameter, die während der Untersuchung variiert wurden, sind in der nachfolgenden Liste dargestellt. Sie steuern die Skalierung und die Komplexität des Modells und der Trainingsdaten:

- **Datenbudget:** Legt fest, wie viele Sequenzen für das Training, die Validierung und den abschließenden Test erzeugt werden mit Hilfe von (`train_example_count`, `eval_example_count` und `test_example_count`).

- **Modellkapazität:** Bestimmt die Größe des Transformers über die Einbettungsdimension (`d_model`), die Anzahl der Aufmerksamkeitsköpfe (`nhead`) und die Anzahl der Encoder-Schichten (`num_layers`).

- **Optimierungsparameter:** Steuern den Trainingsverlauf über die beiden Lernraten (`base_learning_rate` und `minimum_learning_rate`), das Aufwärmen der Lernrate (`warmup_factor`), die Anzahl der Epochen (`epochs`), die Gewichtung positiver Klassen (`pos_weight`), die Regularisierung (`weight_decay`), sowie den Zufallswert (`seed`) für reproduzierbare Läufe.

- **Datenzusammensetzung:** Steuert die Komplexität und Vielfalt der Trainingsdaten über das Steckerpaar-Limit K (`plugboard_limit`), das Flag für leere Steckerbretter (`include_zero_pairs`) und die Randomisierung der Enigma-Maschinenparameter. Außerdem wird das Verhältnis von Klartext-Verschlüsselungen pro Enigma-Schlüssel (`precompute_encrypt_ratio`) sowie dessen Zusammensetzung aus Steckerpaaren und den übrigen Enigma-Maschinenparametern (`precompute_settings_pair_ratio`) reguliert.

Evaluationsmetriken (Abhängige Variablen): Zur Überprüfung des Lernerfolgs werden während der Validierungs- und Testphase folgende abhängige Variablen erfasst:

- **Validierungs-Loss:** Berechnet über die *Binary Cross-Entropy mit Logits* als kontinuierlicher Indikator für die mathematische Konvergenz.

- **Exact-Match-Accuracy:** Strenge Hauptmetrik, bei der eine Vorhersage nach der Sigmoid-Aktivierung mit einem Schwellenwert von 0,5 nur dann als korrekt gewertet wird, wenn alle 325 Klassen des Zielvektors fehlerfrei bestimmt wurden.

- **Exact-Pair-Match-Accuracy:** Eine detaillierte Hilfsmetrik, welche die exakte Übereinstimmung auf der Ebene einzelner Steckerpaare misst.

- **Mikro-gemittelte Precision, Recall und F1-Score:** Aufgrund der harten Accuracy-Metrik und der vielen negativen Klassen dienen diese Metriken als gleichwertige Kontrollinstrumente, um zu verifizieren, ob das Modell die Zielzuordnung

sinnvoll approximiert oder lediglich von der hohen Anzahl an Nullklassen profitiert (triviale Baseline).

4.2.2 Das Problem der trivialen Null-Klassen-Kompensation

Der Zielvektor für das Steckerbrett ist sehr ungleich verteilt. Bei einem maximalen Limit von $K = 10$ aktiven Steckern sind im Höchstfall nur 10 von 325 möglichen Verbindungen aktiv und tragen eine Eins. Die restlichen mindestens 315 Dimensionen sind Null. Ohne Gegenmaßnahmen wählt das Modell eine einfache Abkürzung: Es sagt für alle 325 Klassen pauschal immer Null voraus. Dadurch wirkt der Fehler auf den ersten Blick klein, ohne dass das Modell ein echtes Steckerpaar identifiziert.

Um dies zu verhindern, wird die Loss-Funktion (`BCEWithLogitsLoss`) über den Parameter `pos_weight` angepasst, wodurch positive Klassen (die Einsen) ein höheres Gewicht erhalten. Hierbei muss eine feine Balance gehalten werden:

- **Zu niedriges `pos_weight`:** Das Modell neigt dazu, fast überall Nullen vorherzusagen, da die wenigen Einsen im Label innerhalb der Loss-Funktion kaum ins Gewicht fallen.

- **Zu hohes `pos_weight`:** Das Modell neigt dazu, fast überall Einsen vorherzusagen. Es versucht dadurch, die sehr hohe Fehlerstrafe für eine nicht erkannte Eins durch konsequentes Vorhersagen von positiven Klassen zu umgehen.

In Vorversuchen hat sich ein Wert von 25 als brauchbarer Kompromiss gezeigt. Das bedeutet, dass eine positive Klasse aus dem Label 25-mal stärker gewichtet wird als eine negative Klasse. Damit bekommt das Modell einen Anreiz, Klassen als positiv vorherzusagen.

Wie stark die ungleiche Verteilung der Daten die Metriken verfälscht, zeigt die naive statistische Baseline. Wenn die Anzahl der Steckerpaare im Datensatz gleichmäßig von 0 bis 10 verteilt ist, existieren 11 mögliche Klassen. In exakt 1/11 aller Fälle (ca. $9,09\%$) hat die Enigma-Maschine überhaupt keine Steckerpaare (Klasse 0). Daher könnte es sein, dass das Modell eine Exact-Match-Accuracy von $9,09\%$ aufweist, aber noch kein echter Lernerfolg vorliegt, da Precision und Recall bei $0,00\%$ verharren. Ein echter Fortschritt zeigt sich erst, wenn das Modell dieses Plateau durchbricht. Dies lässt sich daran erkennen, dass der mikro-gemittelte F1-Score (*Micro-F1-Score*), welcher die reinen Null-Treffer (True Negatives) definitionsgemäß ignoriert, synchron mit der Exact-Match-Accuracy ansteigt.

4.2.3 Schrittweises Vorgehen bei steigender Komplexität

Die Untersuchung der Enigma-Maschine erfolgte stufenweise. Zunächst wurde ein vereinfachter Vorversuch mit fester Walzenlage durchgeführt. Dabei blieben Walzen, Umkehrwalze, Ringstellung und Grundstellung konstant, während ausschließlich das Steckerbrett variiert wurde.

Anschließend wurde die Komplexität erhöht, indem die Enigma-Maschinenparameter randomisiert und zunächst direkt ein Steckerpaar-Limit von $K = 10$ untersucht wurde. Da sich dieser Ansatz in der explorativen Phase als schwer auswertbar erwies und über längere Zeit kein klarer Lernfortschritt erkennbar war, wurde die Komplexität anschließend reduziert. Die weitere Analyse konzentrierte sich daher zunächst auf Läufe mit $K = 3$. Dadurch ließ sich das Trainingsverhalten besser nachvollziehen, und kleinere Änderungen an der Modellarchitektur konnten zuverlässiger bewertet werden.

Nach diesem vereinfachten Einstieg wurde die randomisierte Walzenlage systematisch untersucht. Zunächst wurde ein reduziertes Steckerpaar-Limit von $K = 3$ betrachtet. Dabei wurden Walzenlage, Umkehrwalze, Ringstellung und Grundstellung randomisiert. Die Ergebnisse des Trainingslaufs in Epoche 30 nach rund 17,2 Stunden sind zusammen mit den verwendeten Hyperparametern in Tabelle 2 dargestellt. Für den späteren Vergleichslauf mit $K = 10$ wurden dieselben Hyperparameter verwendet. Geändert wurde nur das Steckerpaar-Limit von 3 auf 10 mögliche Steckerpaare.

Die Lernkurven des $K = 3$-Laufs in Abbildung 14 zeigen somit das Trainingsverhalten über die Epochen, während Tabelle 3 die zugehörigen Endmetriken aus Epoche 30 sowie die Werte der Exact-Pair-Match-Accuracy für beide Läufe zusammenfasst.

Leistungsabfall bei höherer Komplexität Die Endmetriken aus Tabelle 3 zeigen einen Leistungsabfall mit steigender Komplexität. Auf dem Test-Set erreicht das Modell bei $K = 3$ in Epoche 30 eine Accuracy von 79,88 % und einen F1-Score von 87,12 %. Beim Lauf mit $K = 10$ sinken diese Werte auf 48,20 % beziehungsweise 51,66 %. Da Architektur und Trainingsdauer in beiden Läufen gleich gehalten wurden und in beiden Fällen nur 12 Millionen Trainingsbeispiele verwendet werden konnten, spricht dieser Unterschied dafür, dass die größere Zahl möglicher Enigma-Schlüssel die Aufgabe schwerer macht.

Verhältnis von Trainings- und Validierungs-Loss Die Lernkurven in Abbildung 14 und Abbildung 15 zeigen für beide Läufe einen weiter sinkenden Trainings-Loss. Unterschiede zeigen sich vor allem im Validierungsverlauf. Beim Lauf mit $K = 3$ verbessert sich der Validierungs-Loss zunächst und steigt danach nur moderat an. Beim Lauf mit $K = 10$ ist der Verlauf ungünstiger. Dort ist der Validierungs-Loss bereits in der ersten Epoche

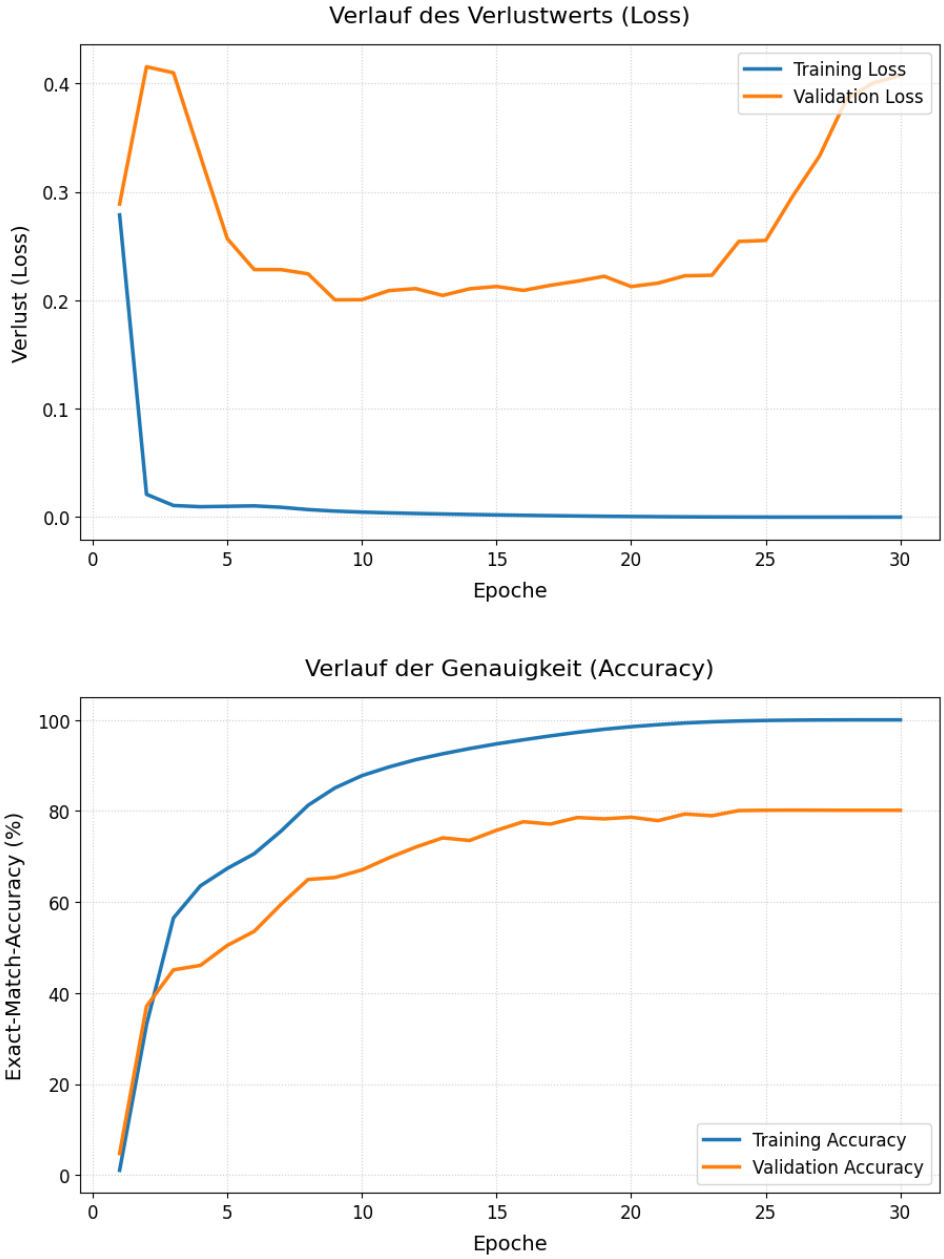

Abb. 14: Verlauf von Trainings- und Validierungs-Loss (oben) sowie der Accuracy (unten) über 30 Epochen bei 3 maximalen Steckerbrettpaaren.

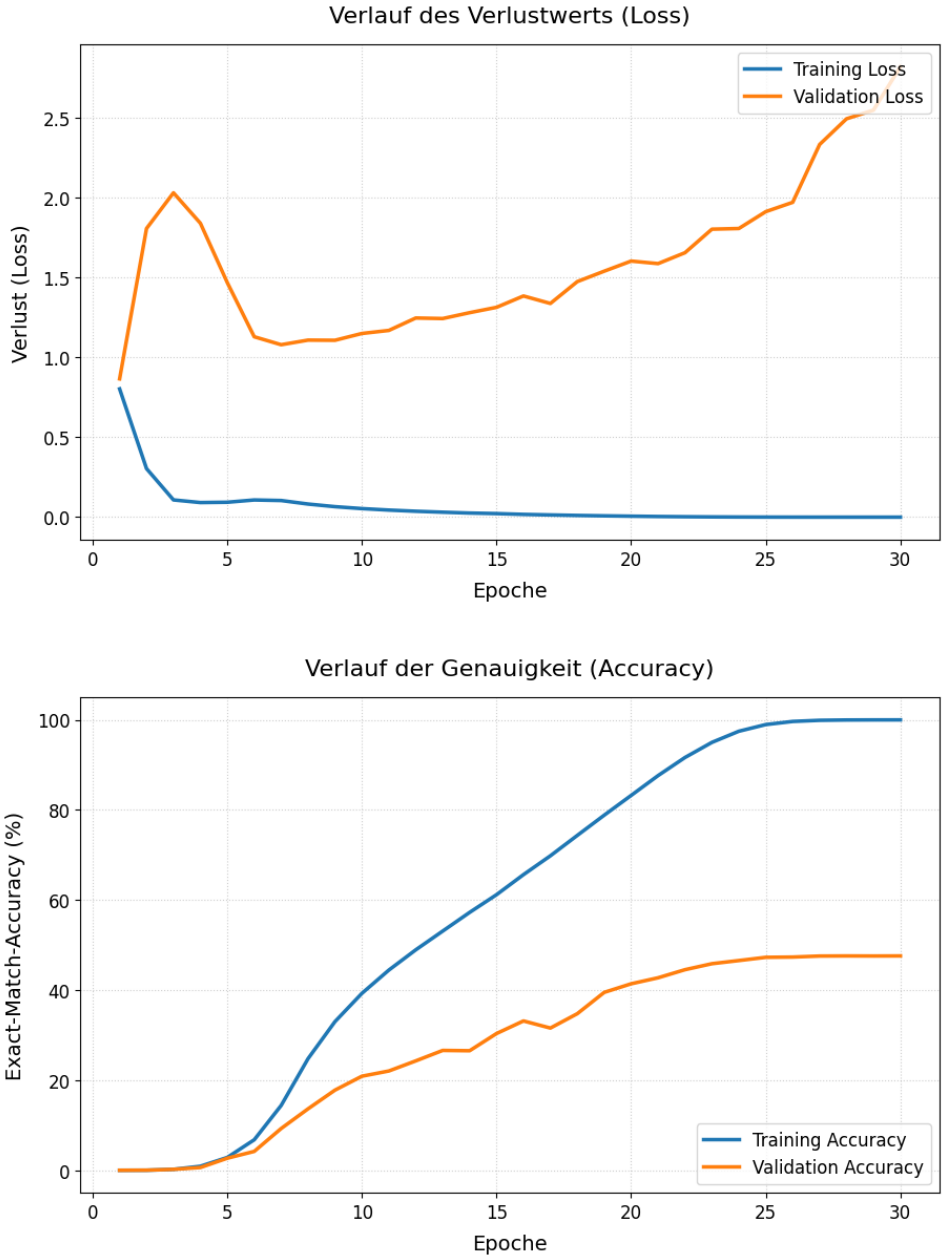

Abb. 15: Verlauf von Trainings- und Validierungs-Loss (oben) sowie der Accuracy (unten) über 30 Epochen bei 10 maximalen Steckerbrettpaaren.

Parameter	Wert
include_zero_pairs	True
randomize_rotors	True
randomize_reflector	True
randomize_ring_settings	True
randomize_start_positions	True
plugboard_limit	3
train_example_count	12 Mio.
settings_plugboard_ratio	350
precompute_encrypt_ratio	50
precompute_encrypt_strategy	key_focus
d_model	384
num_layers	6
nhead	8
base_learning_rate	$2 \cdot 10^{-4}$
minimum_learning_rate	$1 \cdot 10^{-5}$
weight_decay	$1 \cdot 10^{-2}$
warmup_factor	0,2

Tab. 2: Modellkonfiguration des randomisierten Laufs mit $K = 3$.

am niedrigsten und nimmt danach über weite Teile des Trainings zu. Dieses Verhalten ist eine Folge der für die Aufgabe gewählten Metrik-Kombination aus `BCEWithLogitsLoss`, positivem Klassengewicht und strenger Exact-Match-Accuracy. Der Loss bestraft schon moderate, aber systematische Abweichungen auf den vielen negativen Klassen, während die Accuracy nur prüft, ob nach dem Schwellenwert von 0,5 der gesamte Zielvektor exakt getroffen wurde. Konkret liegt der beste Validierungs-Loss bei $K = 3$ bei 0,2003 und steigt bis Epoche 30 auf 0,4072. Bei $K = 10$ wird der beste Wert von 0,8655 bereits in Epoche 1 erreicht und wächst bis Epoche 30 auf 2,8183 an. Für diese Aufgabe ist der niedrigste Loss daher nicht automatisch der passendste Checkpoint, da frühe Modelle trotz günstigerem Loss noch zu konservativ sein können und kaum positive Klassen aktivieren. Für die Bewertung der kryptanalytischen Leistungsfähigkeit sind deshalb Exact-Match-Accuracy und F1-Score hilfreicher als der reine Validierungs-Loss. Zusammen mit der weiterhin hohen Trainings-Accuracy deutet der Verlauf dennoch auf stärkeres Overfitting im komplexeren Szenario hin.

Precision und Recall Auch Precision und Recall aus Tabelle 3 zeigen den Unterschied zwischen beiden Läufen. Bei $K = 3$ liegt die Precision mit 95,94 % über dem Recall von 80,24 %. Das Modell arbeitet in diesem Fall eher zurückhaltend und erzeugt vergleichsweise wenige falsche positive Vorhersagen. Bei $K = 10$ liegen Precision und Recall mit 53,76 % beziehungsweise 48,33 % dagegen näher beieinander. Gleichzeitig blei-

	3-Pair Run			10-Pair Run		
Metrik	**Train**	**Valid**	**Test**	**Train**	**Valid**	**Test**
Loss	$1{,}15 \cdot 10^{-7}$	0,4072	0,4147	$2{,}10 \cdot 10^{-5}$	2,8183	2,7847
Accuracy	100,00 %	80,13 %	79,88 %	99,99 %	47,59 %	48,20 %
Baseline	25,00 %	25,00 %	25,00 %	9,09 %	9,09 %	9,09 %
Precision	-	95,94 %	95,74 %	-	53,76 %	54,55 %
Recall	-	80,24 %	79,92 %	-	48,33 %	49,06 %
F1-Score	-	87,39 %	87,12 %	-	50,90 %	51,66 %
TP	-	962.847	-	-	1.594.892	-
FP	-	40.769	-	-	1.371.771	-
FN	-	237.153	-	-	1.705.108	-
Exact-Pair-Match-Accuracy						
1 Paar	-	80,02 %	-	-	49,50 %	-
2 Paare	-	79,82 %	-	-	46,77 %	-
3 Paare	-	80,55 %	-	-	43,25 %	-
4 Paare	-	-	-	-	46,50 %	-
5 Paare	-	-	-	-	51,25 %	-
6 Paare	-	-	-	-	48,25 %	-
7 Paare	-	-	-	-	46,75 %	-
8 Paare	-	-	-	-	50,92 %	-
9 Paare	-	-	-	-	46,58 %	-
10 Paare	-	-	-	-	46,08 %	-

Tab. 3: Zusammengeführter Vergleich der Trainings-, Validierungs- und Testmetriken in Epoche 30 sowie der Exact-Pair-Match-Accuracy für die Läufe mit $K = 3$ und $K = 10$.

ben beide Werte deutlich unter dem Niveau des $K = 3$-Laufs. Das spricht dafür, dass das Modell unter höherer Komplexität aktive und inaktive Klassen insgesamt schlechter voneinander trennt.

Exact-Pair-Match-Accuracy Die Werte der Exact-Pair-Match-Accuracy in Tabelle 3 ergänzen dieses Bild. Beim Lauf mit $K = 3$ liegen die Werte für 1, 2 und 3 aktive Paare jeweils nahe bei 80 %. Beim Lauf mit $K = 10$ streuen die Werte zwischen 43,25 % und 51,25 %. Ein einzelner Bereich mit besonders guten oder besonders schlechten Werten ist dabei nicht erkennbar. Stattdessen verteilt sich die geringere Leistung relativ gleichmäßig über die verschiedenen Steckerzahlen. Das spricht dafür, dass der Leistungsabfall nicht auf einzelne Teilbereiche beschränkt ist, sondern die komplexere Aufgabe insgesamt betrifft.

5 Diskussion & Ausblick

Wie in Abschnitt 1.1 beschrieben, war das Ziel dieser Arbeit die Untersuchung, ob Transformer-Modelle unter *Ciphertext-Only*-Bedingungen hinsichtlich des Klartexts für die Kryptoanalyse geeignet sind. Im Mittelpunkt stand die Frage, ob ein trainierter Transformer-Encoder die Zuordnung von Geheimtext und bekannten Enigma-Maschinenparametern auf das Steckerbrett hinreichend gut approximieren und die Steckerbrettpaare als Teilschlüssel korrekt klassifizieren kann.

Interpretation Die Ergebnisse zeigen, dass das Modell die Steckerbrettpaare im vereinfachten Szenario ($K = 3$) im Test mit einer Exact-Match-Accuracy von 79,88 % vorhersagen kann. Bei der Skalierung auf die höhere Komplexität ($K = 10$) sinkt die Generalisierung im Test auf 48,20 %. Beide Versuchsreihen liegen jedoch klar über ihren jeweiligen trivialen Baselines von 25,00 % bei $K = 3$ beziehungsweise 9,09 % bei $K = 10$ (siehe Abschnitt 4.2.2). Das deutet darauf hin, dass der Transformer Strukturen des Geheimtexts nutzt, um sie auf die zugrunde liegenden Steckerbrettpaare abzubilden. Gleichzeitig zeigt der Rückgang von $K = 3$ auf $K = 10$, dass die Generalisierung mit wachsender kombinatorischer Komplexität schwieriger wird. Bezogen auf das Verhältnis von Trainingsdaten zu Schlüsselraum bleibt dennoch festzuhalten, dass das Modell mit 12 Millionen Trainingsbeispielen in einem Schlüsselraum mit mehr als 150 Billionen möglichen Konfigurationen von Steckerbrettpaaren lernte.

Limitationen Die Arbeit weist mehrere Limitationen auf, die sowohl die Trainingsergebnisse als auch die Generalisierbarkeit der Ergebnisse einschränken:

- **Regularisierung:** Ein Kompromiss war der Verzicht auf Dropout. In den Vorversuchen störte das stochastische Ausblenden von Features die Permutationsketten der Walzenfortschaltung. Beim $K = 10$-Modell zeigt sich dies im *Overfitting*: Der Validierungs-Loss stagnierte oder stieg bereits ab der zweiten Epoche, während der Trainingsfehler weiter sank.

- **Eingeschränkte Datengenerierung:** Die Datengenerierung war durch Randbedingungen des GPU-Clusters eingeschränkt, sodass eine Beschränkung auf 12 Millionen Samples erforderlich war. Es ist wahrscheinlich, dass zusätzliche Trainingsdaten die Generalisierung bei $K = 10$ verbessert hätten.

- **Fehlendes Hyperparameter-Tuning:** Aus Zeitgründen wurde kein systematisches Tuning durchgeführt. Die Parameter wurden anhand der Lernkurven angepasst. Eine bessere Konfiguration ist daher weiterhin möglich.

Ausblick Die Ergebnisse dieser Arbeit weisen auf mehrere weiterführende Fragestellungen hin:

1. **Alternative Regularisierung:** Da Dropout in den Vorversuchen die Positionsketten störte, könnten künftige Arbeiten untersuchen, wie sich ein höherer Weight Decay oder ein größerer Datensatz auf das Generalisierungsverhalten auswirken. Zusätzlich könnte man die Deaktivierung überdenken und prüfen, ob gezielt eingesetztes Dropout dennoch möglich ist.

2. **Maximale Label-Komplexität:** Ein naheliegender nächster Schritt wäre ein stufenweises Training bis an das mechanisch mögliche Limit von $K = 13$ Steckerpaaren.

3. **Erweiterung auf eine 4-Rotor-Enigma:** Ein potenzieller Folgeschritt wäre die Übertragung der Pipeline auf eine 4-Rotor-Enigma mit vier Rotoren. Hierdurch würden Schlüsselraum und Modellanforderungen weiter steigen.

4. **Erhöhung der Kontext-Sequenzlänge:** Eine Erweiterung der Eingabesequenzen über 100 Tokens hinaus könnte dem Modell zusätzliche Kontextinformationen bereitstellen. So ließen sich zyklische Muster der Walzenbewegung möglicherweise besser erfassen.

5. **Evaluation anderer Datensätze:** Der Wechsel von *TinyStories* zu komplexeren Textquellen wie dem *Gutenberg*-Datensatz könnte zeigen, wie robust das Modell gegenüber realistischeren und komplexeren Sprachmustern ist.

6. **Alternative Evaluationsmetriken:** Ergänzend zur strengen Exact-Match-Accuracy könnten künftige Arbeiten Metriken wie *Top-K-Accuracy* einbeziehen, um Teilerfolge bei der Vorhersage aktiver Steckerpaare differenzierter zu erfassen.

7. **Metadaten der Anzahl der Steckerpaare:** Ein weiterer interessanter Ansatz wäre die Frage, ob die exakte Anzahl der Steckerpaare dem Modell als zusätzlicher Eingabeparameter bereitgestellt oder als separates Nebenziel vorhergesagt werden könnte.

8. **Alternative Loss-Funktion:** Da der eingesetzte `BCEWithLogitsLoss` in Kombination mit `pos_weight` und der strengen Exact-Match-Accuracy zu schwer interpretierbaren Ergebnissen zwischen Loss und Accuracy führte, wäre eine angepasste Loss-Funktion ein naheliegender nächster Schritt.

6 Reflexion & Danksagung

Diese Projektarbeit hat mich mehr beschäftigt, als ich anfangs erwartet hatte. Schwierigkeiten gab es an vielen Stellen: In der Implementierung der Architektur, der Datenaufbereitung und in der wiederkehrenden Frage, ob schwache Ergebnisse auf die Hyperparameter, die Daten oder das Modell selbst zurückzuführen sind. Rückblickend hat sich dadurch mein Blick auf Machine Learning deutlich verändert.

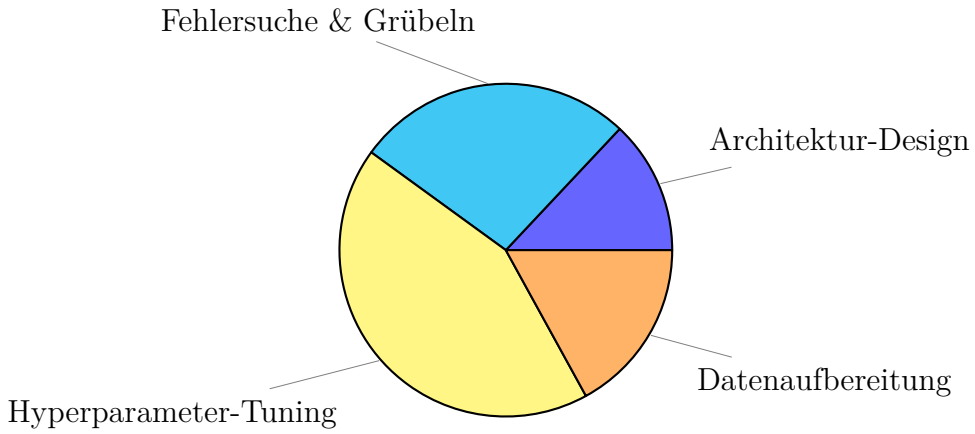

Abb. 16: Gefühlte Aufteilung der aufgewendeten Zeit.

Am meisten Spaß gemacht haben mir die Datenaufbereitung und der erste Testlauf mit der Caesar-Chiffre. Das Modell war klein, die Datenmenge überschaubar, und dennoch lernte es die Chiffre sehr schnell. Dieser Moment zeigte mir, dass das grundlegende Prinzip funktioniert. Auch wenn das finale Modell hinter meinen ursprünglichen Erwartungen zurückblieb, hat die Arbeit viele neue Ansatzpunkte für weitere Experimente eröffnet.

Ich bedanke mich bei meinem Betreuer Maik Bastian für die kontinuierliche Begleitung. Einen Ansprechpartner zu haben, auch wenn ich mal nicht weiter wusste, war für mich sehr hilfreich. Die vielen Mails und Gespräche haben mir geholfen, mich nicht im Detail zu verlieren und den Blick darauf zu behalten, was das Modell am Ende tatsächlich leisten soll.

Herrn Professor Bernhard Esslinger danke ich für sein direktes Feedback. Ich habe dabei gelernt, Ergebnisse nüchterner einzuschätzen und Fragen knapper zu beantworten, als es mir zunächst lag.

Als Erkenntnis nehme ich aus dieser Arbeit mit, dass Machine Learning in der Praxis deutlich „unordentlicher“ ist als in der Theorie. Es gibt selten den einen richtigen Weg, sondern viele kleine Entscheidungen, die das Ergebnis beeinflussen. Das war manchmal frustrierend, aber letztlich der Teil, der mich am meisten gelehrt hat.

Literaturverzeichnis

[1] URL: https://www.cryptool.org/de/ (besucht am 15. 06. 2026).

[2] Jimmy Lei Ba, Jamie Ryan Kiros und Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML]. URL: https://arxiv.org/abs/1607.06450.

[3] Alfred Beutelspacher. *Kryptologie Eine Einführung in die Wissenschaft vom Verschlüsseln, Verbergen und Verheimlichen*. 10. Aufl. Springer Spektrum, 2015.

[4] Johannes Buchmann. *Einführung in die Kryptographie*. 6. Aufl. Springer-Lehrbuch. Berlin, Heidelberg: Springer Vieweg, 2016. ISBN: 978-3-642-39774-5. DOI: 10.1007/978-3-642-39775-2.

[5] Ronen Eldan und Yuanzhi Li. *TinyStories: How Small Can Language Models Be and Still Speak Coherent English?* 2023. arXiv: 2305.07759 [cs.CL]. URL: https://arxiv.org/abs/2305.07759.

[6] Bernhard Esslinger. *Learning and Experiencing Cryptography with CrypTool and SageMath*. https://us.artechhouse.com/Learning-and-Experiencing-Cryptography-with-CrypTool-and-SageMath-P2378.aspx. Artech House, Norwood, 2024.

[7] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 978-1491962299.

[8] Sam Greydanus. „Learning the Enigma with Recurrent Neural Networks". In: *CoRR* abs/1708.07576 (2017). arXiv: 1708.07576. URL: http://arxiv.org/abs/1708.07576.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun. „Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[10] Zhichen Hu, Bowen Liu, Xufan Ren und Yuhe Tang. *Analysis and Implementation of the Enigma Machine*. IEEE, 2022.

[11] Dhiraj D. Kalamkar u. a. „A Study of BFLOAT16 for Deep Learning Training". In: *arXiv preprint arXiv:1905.12322* (2019). URL: https://arxiv.org/abs/1905.12322.

[12] Ramanpreet Kaur, Dušan Gabrijelčič und Tomaž Klobučar. „Artificial intelligence for cybersecurity: Literature review and future research directions". In: *Information Fusion* 97 (2023), S. 101804. ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2023.101804. URL: https://www.sciencedirect.com/science/article/pii/S1566253523001136.

[13] Shibamouli Lahiri. „Complexity of Word Collocation Networks: A Preliminary Structural Analysis“. In: *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*. Gothenburg, Sweden: Association for Computational Linguistics, Apr. 2014, S. 96–105. URL: http://www.aclweb.org/anthology/E14-3011.

[14] Richard Liaw u. a. *Tune: A Research Platform for Distributed Model Selection and Training*. 2018. arXiv: 1807.05118 [cs.LG]. URL: https://arxiv.org/abs/1807.05118.

[15] Ilya Loshchilov und Frank Hutter. „Decoupled weight decay regularization“. In: *arXiv preprint arXiv:1711.05101* (2017). arXiv: 1711.05101. URL: http://arxiv.org/abs/1711.05101.

[16] Paulius Micikevicius u. a. „Mixed Precision Training“. In: *International Conference on Learning Representations (ICLR)*. 2018. URL: https://arxiv.org/abs/1710.03740.

[17] Brian Neal. *Py-Enigma: A Library for Historically Accurate Enigma Machine Simulation in Python*. https://github.com/gremmie/enigma. Version 1.0.2. 2022.

[18] NVIDIA Corporation. *NVIDIA Ampere Architecture Whitepaper*. White Paper. NVIDIA Corporation, 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[19] Razvan Pascanu, Tomas Mikolov und Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG]. URL: https://arxiv.org/abs/1211.5063.

[20] Adam Paszke u. a. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703.

[21] Paul Reuvers und Marc Simons. *How does an Enigma machine work?* Abbildung: Simplified circuit diagram of a 3-rotor Service Enigma. Crypto Museum. 2024. URL: https://www.cryptomuseum.com/crypto/enigma/working.htm (besucht am 07. 05. 2026).

[22] Claude E. Shannon. „Communication Theory of Secrecy Systems“. In: *Bell System Technical Journal* 28.4 (1949), S. 656–715.

[23] Stephan Spitz, Michael Pramateftakis und Joachim Swoboda. *Kryptographie und IT-Sicherheit. Grundlagen und Anwendungen*. 2. Aufl. Wiesbaden: Vieweg+Teubner Verlag, 2011. ISBN: 978-3-8348-8120-5. DOI: 10.1007/978-3-8348-8120-5.

[24] Adane Nega Tarekegn, Mohib Ullah und Faouzi Alaya Cheikh. *Deep Learning for Multi-Label Learning: A Comprehensive Survey*. 2024. arXiv: 2401.16549 [cs.LG]. URL: https://arxiv.org/abs/2401.16549.

[25] Ashish Vaswani u. a. „Attention Is All You Need“. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen aus dem Internet. Diejenigen Paragraphen der für mich geltenden Prüfungsordnungen, die etwaige Betrugsversuche betreffen, habe ich zur Kenntnis genommen.

Der Speicherung der Projektarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

(Datum) (Unterschrift)