SE5012 Modellgetriebene Softwareentwicklung in der Praxis - Projekt Dokumentation -

Ben Riegel 5178979
Mirko Lehn 5054637

Technische Hochschule Mittelhessen, Gießen, Germany
ben.riegel@mni.thm.de
mirko.lehn@mni.thm.de

1 Anforderung an das Projekt

Im Rahmen der Veranstaltung 'Modellgetriebene Softwareentwicklung in der Praxis' (gekürzt aus dem Englischen: MDDP) wurde das Model-Driven Development neben der Vorlesung an einem Projekt praktisch erprobt und umgesetzt. Endgültig sollte man in der Lage sein per Modelltransformation aus einem Modell eine statische Webseite zu generieren. Gewünscht waren, neben dem Generat, mindestens zwei Abstraktionsstufen. Das erste Modell (M0) der obersten Abstraktionsebene sollte hierbei die Legalität von zu verwendenden HTML Elementen festlegen (Metamodell -Konzepte einer Modellierungstechnik). Darunter fallen beispielsweise deren Typen, ihre hierarchische Anordnungen im HTML Dokument oder mögliche zuweisbare Attribute. In einer tieferen Abstraktionsschicht (M1) wird anschließend eine Webseite modelliert dargestellt. Es gilt nun zu Überprüfen ob sich M1 an die Regeln von M0 hält.
Der letzte Schritt ist die Generation des HTML Codes.
Die Zusammenfassung in diese drei Schritte ist für diese Dokumentation vereinfacht doch ausreichend um das Ziel das Projektes zu verdeutlichen.

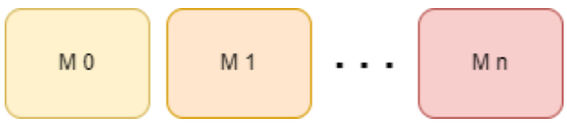


Fig. 1: Vereinfachte Darstellung der Modelle verschiedener Abstraktionsebenen.

Um das Projekt erfolgreich Abzuschließen gab es vorgegebene Meilensteine an denen man sich inhaltlich und zeitlich orientieren sollte. Diese lauteten wie folgt:

1. Manuelle Erstellung einer **Webseite**
2. Entwicklung einer **Modellierungssprache**
3. Entwicklung eines **Codegenerators**
4. Entwicklung eines **Modell-Discoverers**
5. Entwicklung eines **Webseitengenerators**

Der Gedanke hierbei ist die Arbeit rückwärts zu beginnen, angefangen mit einem gewünschten Ergebnis (Die manuell erstellte Webseite) über die Entwicklung der dazu benötigten Modellierungssprache (Metamodell und spezifisches Modell der Webseite), bis hin zum Codegenerator, der aus dem Modell die entsprechende Webseite erstellt.

[1] https://keras.io/

Ein weiterer Schritt ist der Modell-Discoverer, der per Reverse-Engineering ein Modell erstellen soll. Normalerweise wird eine solche Technik häufig für Code-Refactoring oder Modellbasierte Migration (Beispielsweise beim Überführen von Programmcode in eine andere Sprache) verwendet. In diesem Fall jedoch soll das erstellen der Webseite erleichtert werden indem z.B. eine Ordnerstruktur genutzt wird, um die Webseite (inkl. deren Inhalte) abzubilden. Der Modell-Discoverer erstellt aus den hinterlegten Daten und deren Hirachie ein Modell (entsprechend den Vorgaben des Metamodelles) der Webseite. Der Codegenerator kann dann die gewünschte Webseite generieren.

Vorgehensweise: Für die Umsetzung dieser Aufgaben wählten wir eine durchaus ungewöhnliche Herangehensweise. Mit Hilfe eines Neuronalen Netzes sollen aus Modellen die gewünschten Webseiten generiert werden. Die Wahl 'Machine Learning' zu verwenden führte zu einigen typischen Herausforderungen (wie die Erstellung von Trainingsdaten) die wir in folgenden Abschnitten dieser Dokumentation erläutern werden.
Inwieweit die vorgesehenen Meilensteine umgesetzt wurden wird ebenfalls beschrieben.

Vision: Durch die Verwendung eines neuronalen Netzes sollen aus einem Modell, indem nur die nötigsten Informationen hinterlegt sind, dennoch anschauliche Webseiten generiert werden. Das Netz wäre somit ein Tool welches das Styling und Design vollständig übernimmt. Hierzu darf das Netz auch zusätzliche HTML Elemente hinzufügen oder vorgegebene Elementtypen ändern. Ziel ist es also aus einer zügig kreierten minimalistischen Modelldarstelung eine statische Webseite zu erhalten - in der Praxis verwendbar beispielsweise als Mockup, Ausgangspunkt weiterer Implementierungen oder auch als fertiges Produkt.

2 Arbeitsumgebung und Frameworks

Da wir uns für die Umsetzung des Projektes mit einem Neuronalen Netzes entschieden bot sich, auch aufgrund von Erfahrung aus früheren Projekten, der Framework Keras[1] an. Keras ist ein Framework zum einfachen erstellen von Netzen oder Manipulation von Daten. Er basiert auf Tensorflow, ein low-level Framework der für Tensorarethmetik auf GPU's beziehungsweise TPU's ausgelegt ist.

Da der Framework in Form von Pythonlibraies verfügbar ist, nutzten wir ein Google-Colab Notebook mit Python Kernel. Dies ermöglichte wiederum die Zusammenarbeit im Team am Code.

Fig. 2: Die Logos von Keras, Tensorflow und Google Colab in selbiger Reihenfolge. Dies sind die drei Frameworks auf die in diesem Projekt zurückgegriffen wurde.

3 Modelldesign

Zwar begannen wir damit, wie in Meilenstein 1 vorgesehen, statische Webseiten manuell zu erstellen, bemerkten jedoch das eine strikte Orientierung an diesen nicht Zielführend sein kann, da Webseiten nach unserer Vorstellung zu einem hohen Grad variabel erstellt werden sollen. Frühzeitig festzulegen welche Styling-Optionen und Funktionen letztendlich möglich sein sollen ()festgelegt durch die entsprechenden Webseiten) hätte zudem den Aufwand der Projektumsetzung möglicherweise maßgeblich erschwert.
Aus diesem Grund übersprangen wir diesen Meilenstein zunächst und fokussierten den **2. Meilenstein** und begannen zuerst mit der erstellung einer **Modellierungssprache**.

Beispiel PlantUML Instanzdiagramm

```
@startuml

object Body
object Header
object Logo {
+ href = "logo.png"
}
object Main {
- Wilkommen auf meiner Webseite
}
object Footer

'SPLIT

Body : <body>
Header : <header>
Main : <main>
Footer : <footer>

'SPLIT

Body --> Header
Body --> Main
Body --> Footer
Header --> Logo

@enduml
```

Nach einigen versuchen selbst eine listen-artige Struktur als Modell zur Repräsentation einer Webseite (M1 in Abbildung 1) zu designen wurde uns bewusst, dass die benötigte Komplexität der Liste nicht annähernd geringer ist als das resultierende HTML Dokument. Die Länge des Codes war nach einigen Versuchen jeweils nur unbedeutend kürzer. Zudem fehlte eine grafische Darstellung des Modells.

Ein weiterer Versuch bestand nun darin ein UML Instanzdiagram unter Verwendung von Bibliotheken als Python code zu exportieren. Die wichtigsten Elemente dieses Python codes können anschließend entnommen werden und als Eingabedaten für das Neuronale Netz verwendet werden.

Während der Umsetzung bemerkten wir jedoch die Vorzüge von PlantUML. Als heruntergebrochene Art der UML-Notation, die auf das wesentlichste beschränkt ist, erfüllt sie exakt unsere Anforderungen. Unter Verwendung von PlantUML Instanzdiagrammen wird nicht nur das Modell einfach und übersichtlich gehalten, verschiedene Tools[2] bieten auch eine grafische Darstellung.

Metamodell: Für unser Metamodell M0 entschieden wir keine weiteren Restriktionen vorzunehmen. Die Benennung von HTML Tags und deren Verschachtelung ist in einem echten HTML Dokument freigestellt und sollte immer von gängigen Browsern verarbeitet werden. Da wir zunächst anstrebten einzelne Webpages (und keine vollständige Website) zu generieren und der Header jeweils unverändert bleibt (und somit nicht mit generiert werden muss) verbleiben lediglich die Elemente mit beliebiger Verschachtelung (siehe Abbildung 3).

Modell Validierung: Da unser Neuronales Netz jedes gültige PlantUML Instanzdiagramm verarbeiten kann (nach einem umfangreichen Pre-Processing wie in Kapitel 6 beschrieben) umgehen wir uns somit auch das Vergleichen eines M0 (siehe Abbildung 1) sowie eine Validierung ob das Modell korrekt ist, da diese Funktionalität durch Tools ja bereits abgedeckt wird. Es verbleibt somit nur noch die Umsetzung des HTML Codegenerators.

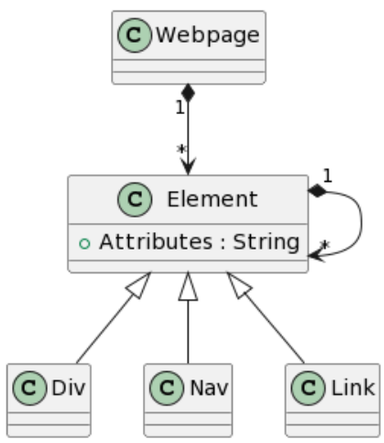


Fig. 3: Darstellung des vereinfachten Metamodells. Die Vererbung ist nur rudimentär repräsentiert und soll zeigen wie alle Elementtypen mit einer Klasse modelliert werden können.

Anmerkung: Für unser Konzept ist es jedoch relevant das Objektbenennungen im PlantUML keine Namen von später existierenden HTML Elementen aufweisen, da durch Ersetzung mit Platzhaltern dann auch die HTML-Tags ersetzt werden würden (beispielsweise *body*). Hierzu reicht es jedoch alle Benennungen im PlantUML am Anfang groß zu schreiben (beispielsweise *Body*). Diese Tatsache wird von uns nicht überprüft.

[2] https://planttext.com

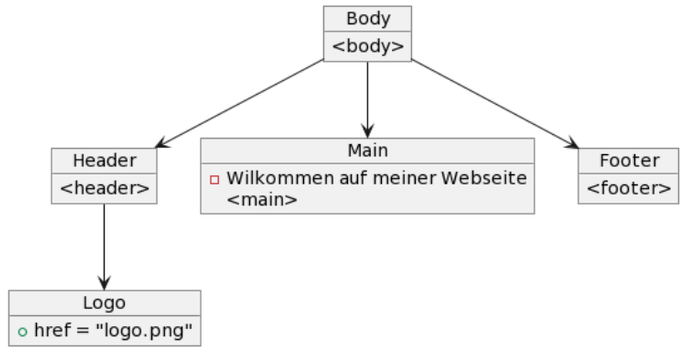


Fig. 4: Darstellung des Beispiel PlantUML Instanzdiagrammes.

4 Wunschresultate - das HTML

Nachdem die Struktur der PlantUML Modelle festgelegt wurde blieb die Überlegung welche Elemente und styling Optionen bereits im Modell festgelegt oder von unserem Neuronalen Netz umgesetzt werden sollen. Dieser Vorgang, der dem **1. Meilenstein** entspricht, war ein langer Prozess, der sich über die gesamte Projektdauer erstreckte. Regelmäßig waren hier Anpassungen erforderlich.

Das Hinzufügen von Elementen wie *
* oder weiteren * * um Bereiche zu separieren oder zu gruppieren war bereits seit Projektbeginn geplant. Das Styling jedoch ermöglichte uns einige unterschiedliche Vorgehensweisen. Von eigens angelegten CSS Dokumenten über im header embedded CSS oder inline css bis hin zur Benutzung von Frameworks gab es zunächst keine Einschränkungen. Wir entschieden uns dafür es dem Neuronalen Netz so einfach wie möglich zu gestalten durch die reine Verwendung von Bootstrap[3]. Es bietet zahlreiche Möglichkeiten für Styling ohne zu viele unterschiedliche Begriffe zu verwenden. Auch die Kombinatorik unterschiedlicher Klassen ist in Bootstrap nicht zu kompliziert und es gibt in der Praxis häufige Kombinationen von Klassen für einen bestimmten Tag. Anhand von Trainingsdaten kann ein Netz also typische Styling-Schemata erkennen und anschließend generieren beziehungsweise reproduzieren.

Um Bootstrap erfolgreich zu verwenden wurde das Bootstrap-CSS Dokument online eingebunden. Da der Header unserer HTML Dokumente sich somit nicht mehr untereinander unterscheidet trainierten wir das Netz darauf nur den *<body>* zu generieren. Den Rahmen um das Generat, inklusive des *<head>* mit dem eingebundenden Bootstrap Link, wird letztendlich im Post-Processing (siehe Kapitel 6) hinzugefügt.

5 Trainingsdaten

Um ein Netz zu trainieren benötigt man im häufigsten Falle, dem *supervised training*, eine sogenannte ground truth. Dies ist das gewünschte Ergebnis zu einem jeweiligen Daten-Input, an welchem sich ein Netz messen und seine Gewichte anpassen kann.
In unserem Fall entspricht ein solches Trainingspaar also einem PlantUML und dem dazugehörigen HTML, wie es das Netz später generieren soll. Solche Daten gibt es noch nicht, wodurch wir gezwungen waren die Daten selbst zu erstellen. Zeitdruck jedoch zwang uns zu effektiveren Methoden, als alle Trainingspaare händisch zu erstellen.
Wir verwendeten also die API von ChatGPT[4] in unserem Python Notebook um einige PlantUML Diagramme zu generieren. Bereits hier waren die Ergebnisse nicht besonders zufriedenstellend und bedurften manueller Bearbeitung. Anschließend ließen wir chatgpt den HTML Code generieren. Hier stießen wir schnell an die Grenzen des bekannten Sprachmodells, da sich die Aufgabe als zu komplex erwies um zufriedenstellende Ergebnisse zu produzieren. Nach einigen Versuchen schafften wir es durch verkettung kleinschrittiger Anfragen Stück für Stück das gewünschte HTML zu erhalten. Diese Anfragen wurden in eine Schleife implementiert um zügig eine beliebige Menge an HTML Dokumenten abzuspeichern oder direkt während des Training Prozesses zu erstellen.

Bespiel Anfrage an ChatGPT

```
<--! Add pevious output here -->
```

this is a simple static html webpage (only the body). Add class = " " attributes to a lot of these html elements: Use the bootstrap grid system to allign the elements nicely. (like centered or next to each other) Also add or change the bootstrap magins and paddings to display everything as pleasing as possible. Use only bootstrap 4.0 classes, no style tags. If necessary add some html tags.

Bis die manuelle Erstellung, sowie Generierung von PlantUML und den dazugehörigen HTML Webpages zufriedenstellend funktionierte standen wir jedoch mittlerweile unter Zeitdruck. Aus diesem Grund wurden letztendlich nur 30 Trainingspaare erstellt.

6 Pre- und Post-Processing

Eine Webseite enthält teilweise Informationen, die ein Netz unmöglich generieren kann. Beispielsweise eine Referenz auf ein Bild, das lokal an einem bestimmten Pfad hinterlegt ist. Das Netz hat keine Kenntnis über das System und dessen Verzeichnisse, in dem das HTML Dokument später hinterlegt wird. Solche Informationen müssen also bereits im PlantUML angegeben werden (siehe Beispiel PlantUML Instanzdiagramm).

Die Angaben könnten nun an das Netz übergeben werden und müssten als relevant und unveränderbar erkannt werden. Dies jedoch führt zu einem größeren Lernaufwand für das Netz und einem stetigen Risiko von Fehlverhalten des Netzes. Es ist also sinnvoll diese Daten erst später in das fertige Generat einzufügen.
Um hierfür die entsprechende Stelle im HTML code wiederzufinden nutzen wir das *id* Attribut. Wir erstellen zunächst ein Dictionary um Elementbenennungen eine Platzhalter ID zuzuweisen. Diesen Platzhalter kann das Netz anschließend generieren. In einem Postprocessing

[3] https://getbootstrap.com/
[4] https://platform.openai.com/overview

Schritt können nun dieser ID alle weiteren Attribute (wie Beispielsweise Referenzen) zugewiesen und der Platzhalter selbst ausgetauscht werden.

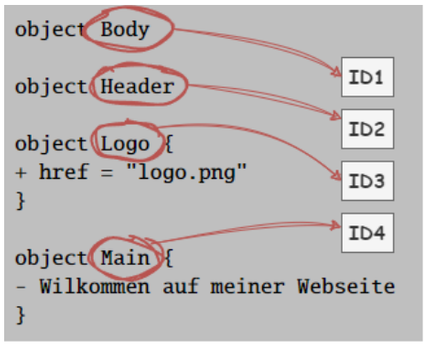


Fig. 5: Verwendung von Platzhaltern als ID's.

Platzhalter verwenden wir jedoch nicht ausschließlich für ID's. Für alle wiederkehrenden Strukturen ist es sinnvoll einen Platzhalter einzuführen, der im Postprocessing wieder ersetzt wird. Ein Beispiel sind die Idents zu Beginn jeder Zeile. Sie sind nicht ntig um HTML code von einem Browser verarbeiten zu lassen, sind aber immer Teil eines ordentlichen und leserlichen Programmcodes. Das Netz jedoch unzählige Leerzeichen in Folge generieren zu lassen ist ein gutes Beispiel für Redundanz und die Gefahr die Falsche Anzahl zu generieren. So ersetzten wir die Anzahl an Leerzeichen in den Trainingsdaten gegen einen Platzhalter, der jederzeit mit einer einfachen Methode wieder in die entsprechende Anzahl Leerzeichen umgewandelt werden kann.

Anmerkung: Im selben Schritt werden auch nicht gewünschte Elemente entfernt, die möglicherweise durch ChatGPT beim erstellen der Trainingsdaten (siehe Kapitel 5) erzeugt wurden. Diese werden selbstverständlich nicht im Preprocessing gegen Platzhalter ersetzt und im Postprocessing ignoriert.

```
00SPACES <body >
04SPACES <header id1 class = " container text - center bg - light " >
08SPACES <div class = " row " >
12SPACES <div class = " col " >
16SPACES <img id20 >
12SPACES </div >
08SPACES </div >
08SPACES <nav id13 class = " navbar navbar - expand - lg navbar - light bg - light " >
12SPACES <div class = " container " >
16SPACES <a id10 class = " navbar - brand " > </a >
16SPACES <button class = " navbar - toggler " data - toggle = " collapse " data -
20SPACES aria - controls = " navbarNav " aria - expanded = " false " aria - label = " Toggle navigation " >
20SPACES <span class = " navbar - toggler - icon " > </span >
16SPACES </button >
16SPACES <div class = " collapse navbar - collapse "  >
20SPACES <ul class = " navbar - nav ml - auto " >
24SPACES <li class = " nav - item " >
28SPACES <a id22 class = " nav - link text - warning " > </a >
24SPACES </li >
24SPACES <li class = " nav - item " >
28SPACES <a id19 class = " nav - link text - danger " > </a >
24SPACES </li >
```

Fig. 6: Ein Beispiel HTML Dokument der Trainingsdaten nach dem Schritt der Vorverarbeitung (Preprocessing). Aus Platzgründen ist hier nur der Beginn des Dokumentes abgebildet.

7 Einführung: Künstliche neuronale Netze

Anmerkung: Dieses Kapitel ist für den Projektbericht redundant, sollte jedoch bei Bedarf technische Details und Begriffe erläutern (vor allem in Bezug auf Kapitel 8). Die Beschriebenen Grundlagen wurden alle in Versuchen während dieses Projektes verwendet und finden sich fast alle in unserer endgültigen Netzstruktur wieder.

Künstliche neuronale Netze (gekürzt aus dem Englischen: ANNs) sind maschinelle Lernmodelle, die von der Struktur und Funktion des menschlichen Gehirns inspiriert sind. Sie bestehen aus Schichten von miteinander vernetzten Knoten oder "Neuronen" (siehe Abbildung 7) die einfache Berechnungen mit Eingabedaten durchführen und die Ergebnisse an die nächste Schicht von Neuronen weiterleiten. Die Verbindungen zwischen den Neuronen sind gewichtet, was bedeutet, dass alle eingehenden Verbindungen unterschiedliche Relevant sein können. [1]

ANN's verwenden einen "Trainings"-Prozess (Backpropagation) um zu lernen, genaue Vorhersagen und richtige Entscheidungen bei neuen Daten zu treffen. Während des Trainings werden dem Netzwerk (normalerweise) eine Reihe von gekennzeichneten Beispielen beziehungsweise Trainingspaaren vorgelegt. Nach der Berechnung des Resultatespasst es seine Gewichte als Reaktion auf die Fehler die es macht an. Dieser Prozess wird in der Regel mit einer Variante des Gradientenabstiegs durchgeführt, bei dem die Gewichte iterativ angepasst werden, um den Fehler zwischen der vorhergesagten Ausgabe und der wahren Ausgabe (ground truth) zu minimieren. [2]

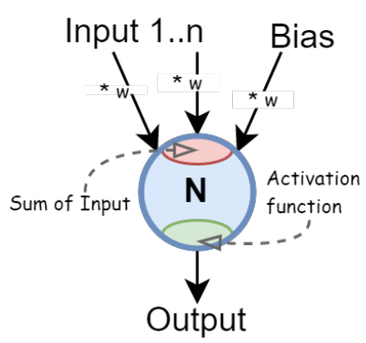


Fig. 7: Ein einzelnes Neuron eines Netzwerkes.

Einer der Hauptvorteile von ANNs ist ihre Fähigkeit, automatisch Merkmale und Muster in Daten ohne menschliches Eingreifen zu erkennen und zu erlernen. Dies macht sie zu einem leistungsstarken Werkzeug für Aufgaben wie Bilderkennung, natürliche Sprachverarbeitung und Datengenerierung. Darüber hinaus ermöglicht die hierarchische Struktur von ANNs das Erlernen immer komplexerer Darstellungen von Daten, wenn sie sich durch mehrere Schichten von Neuronen bewegen. Die grundlegenden Konzepte, Varianten und Ansätze neuronaler Netze werden in den folgenden Abschnitten kurz erläutert.

Anmerkung: Keines der folgenden Konzepte schließt ein Anderes aus. In den meisten modernen Netzwerken finden sich häufig eine Vielzahl von Kombinationen wieder.

7.1 Aktivierungs Funktionen

Aktivierungsfunktionen spielen in künstlichen neuronalen Netzen eine entscheidende Rolle, da sie dem Ausgang eines Neurons eine Nichtlinearität verleihen. Diese Nichtlinearität ermöglicht es neuronalen Netzen, komplexe Beziehungen in Daten zu modellieren, die mit linearen Modellen nur schwer zu erfassen wären. Es gibt mehrere Aktivierungsfunktionen, die in neuronalen Netzen verwendet werden, darunter die beliebten Sigmoid-, ReLU- und tanh-Funktionen.

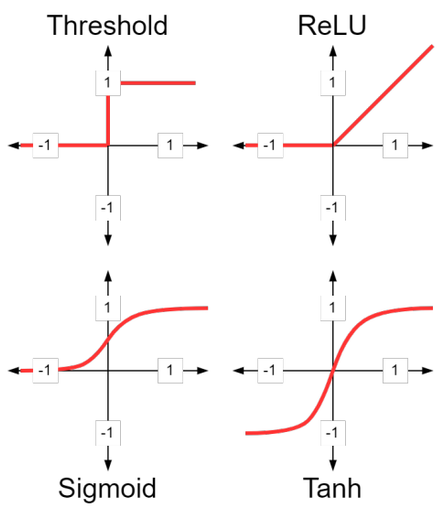


Fig. 8: Beliebte Aktivierungsfunktionen, die in künstlichen neuronalen Netzen verwendet werden. Die horizontale Achse stellt die Summe aller Neuroneneingänge dar. Die vertikale Achse zeigt den entsprechenden Ausgangswert.

Die Sigmoidfunktion ist eine glatte, S-förmige Funktion, die jede Eingabe auf einen Wert zwischen 0 und 1 abbildet. Die Sigmoidfunktion ist differenzierbar und eignet sich daher für die Backpropagation, eine beliebte Technik für das Training neuronaler Netze. Allerdings kann die Sigmoidfunktion unter dem Problem der verschwindenden Gradienten leiden, was das Training von tiefen Netzen schwierig machen kann.

Die ReLU-Funktion (Rectified Linear Unit) ist eine lineare Funktion, die unverändert die Eingabe zurückgibt solange sie positiv ist (ansonsten 0). Die ReLU-Funktion ist in den letzten Jahren aufgrund ihrer Einfachheit, Berechnungseffizienz und guten Leistung in tiefen Netzen populär geworden. Die ReLU-Funktion kann jedoch unter dem Problem der toten Neuronen (dead neurons) leiden, bei dem einige Neuronen während des Trainings dauerhaft inaktiv werden können.[3]

Die tanh-Funktion (hyperbolischer Tangens) ähnelt der Sigmoid-Funktion, bildet aber jede Eingabe auf einen Wert zwischen -1 und 1 ab. Wie die Sigmoid-Funktion ist auch die tanh-Funktion differenzierbar, so dass sie sich für die Backpropagation eignet. Allerdings kann auch die tanh-Funktion unter dem Problem der verschwindenden Gradienten (vanishing gradients) leiden.

In den letzten Jahren wurden neue Aktivierungsfunktionen vorgeschlagen, die darauf abzielen, einige der Einschränkungen der traditionellen Funktionen zu überwinden. Dazu gehören die Swish-Funktion, eine glatte Funktion, die die Vorteile der Sigmoid- und der ReLU-Funktion kombiniert, und die GELU-Funktion, eine glatte Annäherung an die ReLU-Funktion, die das Problem der toten Neuronen entschärfen kann.

7.2 Shallow Networks

Flache neuronale Netze (Shallow Networks) sind neuronale Netze mit maximal zwei "versteckten" Schichten zwischen der Eingabe- und der Ausgabeschicht. Diese Netze sind relativ einfach zu trainieren und zu interpretieren und werden häufig als Ausgangspunkt für das Erlernen tiefer neuronaler Netze verwendet. Im Allgemeinen sind flache neuronale Netze gut für Aufgaben geeignet, die eine Merkmalsextraktion aus Daten erfordern. Bei Bildklassifizierungsaufgaben können flache neuronale Netze beispielsweise trainiert werden, um Merkmale wie Kanten, Ecken und andere Low-Level-Bildmerkmale zu extrahieren.

Die Verwendung flacher neuronaler Netze ist jedoch mit einigen Einschränkungen verbunden. So sind sie möglicherweise nicht in der Lage, komplexe Beziehungen zwischen Eingabemerkmalen zu erfassen, und haben Schwierigkeiten, Abstraktionen auf höherer Ebene aus Daten zu lernen. Daher eignen sie sich möglicherweise nicht für komplexere Aufgaben, wie z. B. die Verarbeitung natürlicher Sprache oder Computer-Vision-Aufgaben, die das Verständnis komplexer visueller Szenen erfordern.

Trotz dieser Einschränkungen sind flache neuronale Netze nach wie vor ein wichtiges Instrument auf dem Gebiet des Deep Learning. Sie können als Sprungbrett für komplexere neuronale Netze dienen und zur Lösung vieler realer Probleme eingesetzt werden.

Anmerkung: Selbst ein flaches neuronales Netz mit nur einer versteckten Schicht kann durch eine komplexe mathematische Funktion dargestellt werden, die theoretisch jedes lösbare mathematische Problem lösen kann. Um dies zu erreichen, wäre jedoch eine unpraktische Anzahl von Neuronen in der versteckten Schicht erforderlich und die Trainingszeit wäre übermäßig lang.

7.3 Deep Neural Networks

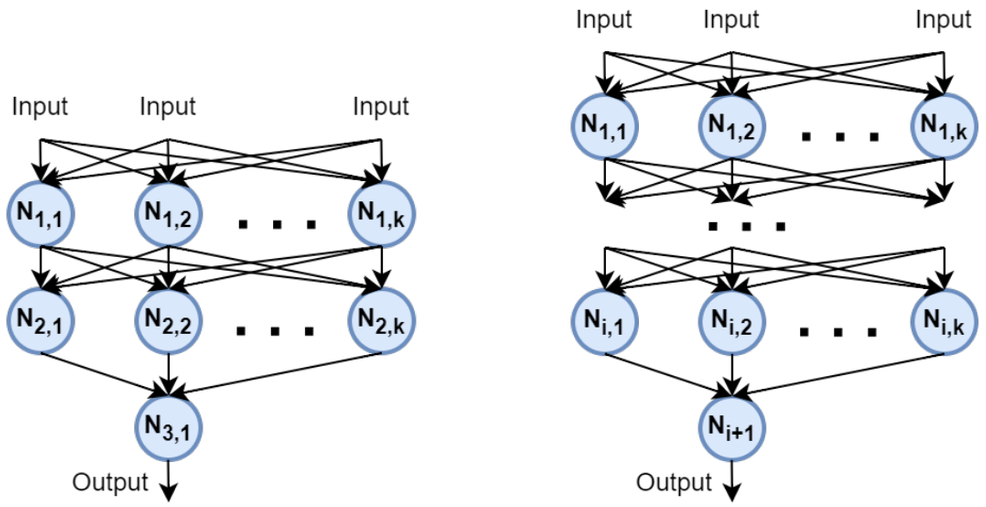
Tiefe neuronale Netze sind eine Klasse von neuronalen Netzen mit mehreren Schichten von Neuronen, die es ihnen ermöglichen, komplexe Darstellungen von Daten zu lernen. Im Gegensatz zu flachen neuronalen Netzen, die nur eine oder zwei verborgene Schichten enthalten, bestehen tiefe neuronale Netze in der Regel aus einer wesentlich größeren Anzahl verborgener Schichten, die manchmal mehrere Hundert oder sogar Tausende erreichen.

Die zusätzlichen Schichten in tiefen neuronalen Netzen ermöglichen es ihnen, immer komplexere Merkmale der

Daten zu lernen, auf denen sie trainiert werden. Bei einer Bilderkennungsaufgabe zum Beispiel könnte die erste Schicht eines tiefen neuronalen Netzes einfache Merkmale wie Kanten und Kurven lernen, während die mittleren Schichten komplexere Merkmale wie Texturen und Muster lernen könnten und die letzte Schicht lernen könnte, Objekte im Bild zu erkennen.

Ein entscheidender Vorteil von tiefen neuronalen Netzen ist ihre Fähigkeit, automatisch Merkmalsdarstellungen aus Rohdaten zu lernen, ohne dass eine manuelle Merkmalserstellung erforderlich ist. Dies macht sie besonders effektiv für Aufgaben wie Bild- und Spracherkennung, natürliche Sprachverarbeitung und andere Aufgaben, bei denen die Eingabedaten komplex und hochdimensional sind.

Die Ausbildung tiefer neuronaler Netze kann jedoch eine Herausforderung darstellen, da die große Anzahl von Schichten und Parametern sie anfällig für Überanpassung und andere Optimierungsprobleme macht. Um diese Probleme zu lösen, haben Forscher eine Reihe von Techniken entwickelt, wie z. B. Dropout, Batch-Normalisierung und frühzeitiges Stoppen, um die Leistung und Stabilität von tiefen neuronalen Netzen zu verbessern.



(a) shallow network

(b) deep network

Fig. 9: **(a)** Ein flaches Netz mit zwei versteckten Schichten. Der Bias für jeden Knoten und die trainierbaren Gewichte für jede Verbindung werden nicht dargestellt. Das einzelne Neuron in der letzten Schicht steht für eine *True* | *False* Klassifizierung. **(b)** Eine version des flachen Netzes als tiefes neuronales Netz mit i "hidden layers".

7.4 Auto Encoders

Autoencoder sind eine Art von tiefen neuronalen Netzen, die für eine Vielzahl von Aufgaben wie Dimensionalitätsreduktion, Anomalieerkennung sowie Bild- und Spracherkennung eingesetzt werden können. Die Hauptidee hinter Autoencodern besteht darin, eine komprimierte Darstellung der Eingabedaten zu erlernen, indem sie diese in einen niedriger-dimensionalen Raum kodieren und dann wieder in den ursprünglichen Raum dekodieren. Die Kodierungs- und Dekodierungsfunktionen werden gleichzeitig durch Backpropagation und Gradientenabstieg erlernt. [4]

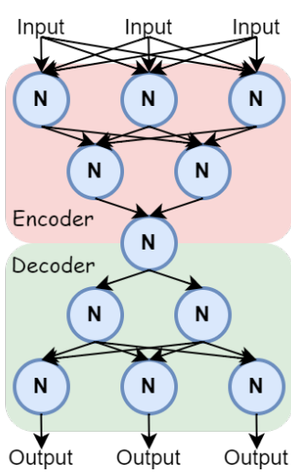


Fig. 10: Ein simples Modell eines Autoencoders.

Autoencoder bestehen in der Regel aus einem Encodernetzwerk, das die Eingabedaten auf eine komprimierte Darstellung abbildet, und einem Decodernetzwerk, das die komprimierte Darstellung zurück in den ursprünglichen Eingaberaum abbildet. Das Ziel des Netzwerks ist es, den Rekonstruktionsfehler zwischen der ursprünglichen Eingabe und der rekonstruierten Ausgabe zu minimieren.

Ein beliebter Typ von Autoencodern ist der Denoising-Autoencoder [5], der das Rauschen aus den Eingabedaten entfernt, indem er das ursprüngliche Signal aus einer verrauschten Version davon rekonstruiert. Ein anderer Typ ist der Variations-Autoencoder, der darauf ausgelegt ist, eine niedrigdimensionale Darstellung der Eingabedaten zu erlernen, die die zugrunde liegende Struktur der Daten erfasst.

Autoencoder wurden erfolgreich in einer Vielzahl von Anwendungen eingesetzt, zum Beispiel bei der Erkennung von Anomalien im "Network traffic", der Bild- und Spracherkennung und der Verarbeitung natürlicher Sprache. Einer der Hauptvorteile von Autoencodern ist ihre Fähigkeit, nützliche Darstellungen der Daten zu lernen, ohne dass eine explizite Überwachung erforderlich ist. Das Training von Autoencodern kann jedoch eine Herausforderung darstellen, insbesondere bei großen Datensätzen, und die Wahl der Hyperparameter und der Architektur kann einen erheblichen Einfluss auf die Leistung haben.

7.5 Convolution

Die Faltung ist eine Schlüsseloperation beim Deep Learning, die eine entscheidende Rolle bei der Bild- und Signalverarbeitung spielt. Bei der Faltung wird ein Filter auf ein Eingangssignal angewendet und ein neues Signal erzeugt, indem der Filter über das Eingangssignal geschoben wird (siehe Abbildung 11). Im Zusammenhang mit der Bildverarbeitung ist der Filter eine kleine Matrix von Gewichten, auch Kernel genannt, die auf das Bild an jeder Pixelposition angewendet wird. Das Ergebnis der Faltungsoperation ist ein neues Bild (oder eine andere Ausgabe, die dem Eingabetyp entspricht), das bestimmte Merkmale des Eingabebildes hervorhebt, je nach Art des verwendeten Filters.

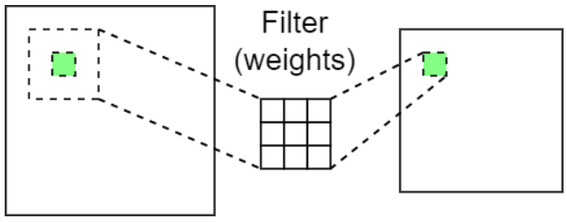


Fig. 11: Ein Schritt eines zweidimensionalen Filters, der auf zweidimensionale Daten angewendet wird. Der Filter wird schrittweise über die Daten bewegt und ein neuer Wert wird in Abhängigkeit von den Nachbarwerten und den entsprechenden Gewichten in der Filtermatrix erstellt. Gezeigt ist ein 3x3 Filter (9 Gewichte). Alle Werte der Eingangsaten werden mit dem Gewicht multipliziert und die Summe als neuer Wert abgebildet.

Convolutional Neural Networks (CNNs) nutzen diese Faltungsoperation, um Bildklassifizierungs- und Objekterkennungsaufgaben durchzuführen. In einem CNN wird eine Reihe von Faltungsschichten verwendet, um Merkmale aus dem Eingangsbild zu extrahieren. Jede Faltungsschicht wendet einen Satz von Filtern auf die Ausgabe der vorhergehenden Schicht an und erzeugt einen Satz von Merkmalskarten. Die Anzahl der Filter in jeder Schicht kann angepasst werden, um die Tiefe des Netzwerks und die Komplexität der extrahierten Merkmale zu steuern. [6]

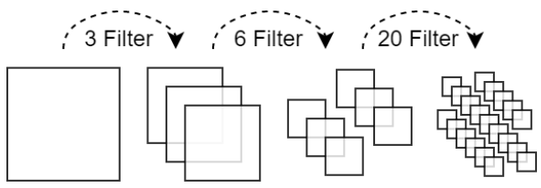


Fig. 12: Ein einfaches Beispiel für ein Faltungsnetz. In diesem Fall werden einige zweidimensionale Daten (behandelt als dreidimensional mit einer Dimension der Größe *1*) als Eingabe verwendet. In der ersten Schicht des Netzes gibt es drei Filter mit trainierbaren Gewichten, die auf die Daten angewendet werden, um neue Daten für jeden Filter zu erstellen. Sie werden Feature-Maps genannt. Die Dimension der Filtergröße ist ein Hyperparameter, der vom Ersteller des Netzes festgelegt wird, mit Ausnahme der letzten Dimension. Die Größe dieser Dimension entspricht der Anzahl der vorherigen Karten, so dass jeder Filter unterschiedliche Informationen aus jeder einzelnen Karte der vorherigen Schicht erhält. Wenn dies nicht vermieden wird, sind die Ausgangsmaps der Filter aufgrund des "sliding window" und des nicht zugänglichen Randbereichs der Eingangs-map etwas kleiner als die diese.

Auf Faltungsschichten folgen häufig Pooling-Schichten, die die Größe der Merkmalskarten durch Downsampling verringern. Dies verringert die Rechenkomplexität des Netzes und macht das Netz auch robuster gegenüber kleinen Änderungen im Eingangsbild. [6]

Anmerkung: Auch wenn Faltung hauptsächlich in der "computer vision" zum Einsatz kommt gibt es Anwendungsspezifische Situationen in denen sie sich anbietet. Bei ein oder mehrdimensionalen Daten, bei denen die Nachbarschaft zueinander eine Bedeutung hat, ist eine Faltungsoperation immer in Erwägung zu ziehen. Im Falle von Sequenzdaten könnte Beispielsweise eine Paarbildung aus immer zwei Eingangsdaten möglich sein.

7.6 Recurrent Neural Networks

Rekurrente neuronale Netze (RNNs) sind eine Art von tiefen neuronalen Netzen, die sich besonders für die Verarbeitung von sequentiellen Daten, wie Zeitreihen oder natürlichsprachliche Texte, eignen. RNNs unterscheiden sich von Feedforward-Netzwerken dadurch, dass sie Schleifen enthalten, die es ermöglichen, dass Informationen über die Zeit hinweg bestehen bleiben und verarbeitet werden.

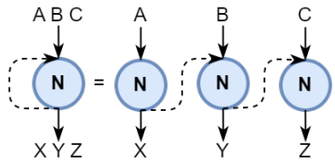


Fig. 13: Ein rekurrentes Netz. Sequentielle Daten werden schrittweise verarbeitet. In modernen rekurrenten Netzen werden frühere Schritte erinnert und haben Auswirkungen auf die folgenden Schritte.

Der Grundgedanke von RNNs besteht darin, einen internen Zustand oder "Speicher" aufrechtzuerhalten, der bei jedem Zeitschritt aktualisiert werden kann, indem die aktuelle Eingabe und der vorherige Zustand berücksichtigt werden. Der interne Zustand wird verwendet, um eine Ausgabe zu erzeugen und den Zustand beim nächsten Zeitschritt zu aktualisieren. Dieser Prozess wird für jeden Zeitschritt wiederholt, so dass das Netz die gesamte Sequenz verarbeiten kann.

Ein gängiger Typ von RNN ist das Long Short-Term Memory (LSTM)-Netz. LSTMs lösen das Problem der verschwindenden Gradienten, die bei herkömmlichen RNNs auftreten können und sie daran hindern, langfristige Abhängigkeiten effektiv zu lernen. LSTMs verwenden eine komplexere Architektur, die es ihnen ermöglicht, frühere Eingaben selektiv zu vergessen oder sich an sie zu erinnern, wodurch sie besser für die Verarbeitung längerer Sequenzen geeignet sind. RNNs und LSTMs werden in einer Vielzahl von Anwendungen eingesetzt, z. B. in der Spracherkennung, Sprachmodellierung und maschinellen Übersetzung. Insbesondere bei der Verarbeitung natürlicher Sprache wie der Analyse von Gefühlen, der Erkennung benannter Entitäten und der maschinellen Übersetzung haben sie sich aufgrund ihrer Fähigkeit, die sequentielle Natur der Sprache zu modellieren, als sehr erfolgreich erwiesen. [7]

RNNs können jedoch auch sehr rechenintensiv sein und neigen zur Überanpassung, insbesondere bei langen Sequenzen. Aus diesem Grund haben Forscher verschiedene Strategien entwickelt, um diese Probleme zu lösen, wie z.

B. Aufmerksamkeitsmechanismen und Regularisierungstechniken.

Trotz ihrer Herausforderungen sind RNNs und LSTMs nach wie vor ein leistungsfähiges Werkzeug für die Verarbeitung sequenzieller Daten und werden in der im Deep-Learning häufig eingesetzt.

7.7 Overfitting & Regularisation

Beim Deep Learning ist die Überanpassung ein häufiges Problem, bei dem ein Modell lernt, sich zu gut an die Trainingsdaten anzupassen, was zu einer schlechten Generalisierungsleistung bei ungesehenen Daten führt. Eine beliebte Technik zur Lösung dieses Problems ist die Regularisierung, bei der der Verlustfunktion des Modells während des Trainings ein Strafterm hinzugefügt wird. Der Strafterm hält das Modell davon ab, sich zu sehr auf eine kleine Untergruppe von Merkmalen oder Parametern zu verlassen, und fördert so eine besser verallgemeinerbare Lösung. Es gibt mehrere Arten von Regularisierungstechniken, die häufig beim Deep Learning verwendet werden, darunter L1- und L2-Regularisierung, Dropout und frühzeitiges Stoppen. [8]

Die L1- und L2-Regularisierung fügt einen Strafwert hinzu, der auf der L1- bzw. L2-Norm der Modellgewichte basiert. Die L1-Regularisierung, die auch als Lasso-Regularisierung bekannt ist, fügt der Verlustfunktion des Modells eine Strafe hinzu, die proportional zur Summe der absoluten Werte der Gewichte ist. Dies führt dazu, dass einige der Gewichte auf Null geschrumpft werden, wodurch eine effektive Merkmalsauswahl erfolgt und die Komplexität des Modells reduziert wird. Die L2-Regularisierung, auch als Ridge-Regularisierung bekannt, fügt der Verlustfunktion des Modells eine Strafe hinzu, die proportional zur Summe der Quadrate der Gewichte ist. Dies führt dazu, dass alle Gewichte gegen Null geschrumpft werden, aber nicht genau auf Null, und reduziert somit die Größe der Gewichte, ohne eines von ihnen vollständig zu eliminieren. Im Allgemeinen könnte man dies als "Vergessen" der trainierten Gewichte im Laufe der Zeit verstehen.

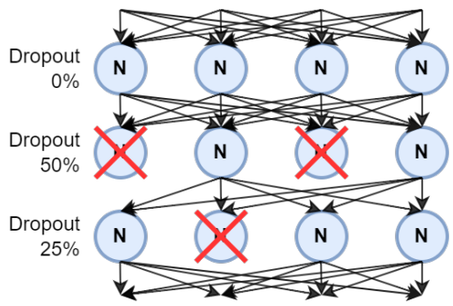


Fig. 14: Darstellung von unterschiedlichen Ausfallraten in Schichten. Die ausgeschiedenen Neuronen wechseln in jedem Trainingsschritt. Das Netzwerk ist gezwungen, ein breites Spektrum von Neuronen zu verwenden, um insgesamt bessere Ergebnisse zu erzielen, anstatt einzelne Neuronen für unterschiedliche Daten zu trainieren, was zu überangepassten Ergebnissen führt (overfitting).

Dropout ist eine weitere weit verbreitete Regularisierungstechnik, bei der ein Teil der Neuronen in einer Schicht während des Trainings nach dem Zufallsprinzip herausgenommen wird (siehe Abbildung 14). Dadurch wird das Modell gezwungen, robustere Merkmale zu erlernen, die nicht von bestimmten Neuronen abhängig sind, wodurch eine Überanpassung verhindert wird.

Das frühzeitige Abbrechen ist eine einfache, aber wirksame Regularisierungstechnik, die den Trainingsprozess stoppt, wenn sich die Leistung des Modells auf einem Validierungssatz nicht mehr verbessert. Dadurch wird verhindert, dass das Modell weiterhin zu stark an die Trainingsdaten angepasst wird, und seine Fähigkeit zur Generalisierung auf neue Daten verbessert.

7.8 Skip Connections

Skip Connections, auch bekannt als Residual Connections, sind eine Technik, die 2015 eingeführt wurde, um das Problem der verschwindenden Gradienten in tiefen neuronalen Netzen zu lindern. In sehr tiefen Netzen können die Gradienten so klein werden, dass sie praktisch verschwinden, was das Lernen des Netzes erschwert. Skip-Verbindungen beheben dieses Problem, indem sie es den Gradienten ermöglichen, einige der Schichten im Netzwerk zu "überspringen", wodurch es für die Gradienten einfacher wird, durch das Netzwerk zurückzufließen und die Gewichte zu aktualisieren.

In einem Netz mit Skip-Verbindungen wird ein Teil der Eingabe in eine Schicht zur Ausgabe einer späteren Schicht hinzugefügt, wodurch eine oder mehrere Schichten effektiv umgangen werden. Die Idee hinter diesem Ansatz ist, dass das Netz, wenn die Identitätsfunktion als mögliche Transformation hinzugefügt wird, sich dafür entscheiden kann, die Eingabe einfach in die Ausgabe der Schicht zu kopieren, so dass die Gradienten diese Schicht bei Bedarf effektiv umgehen können. [9]

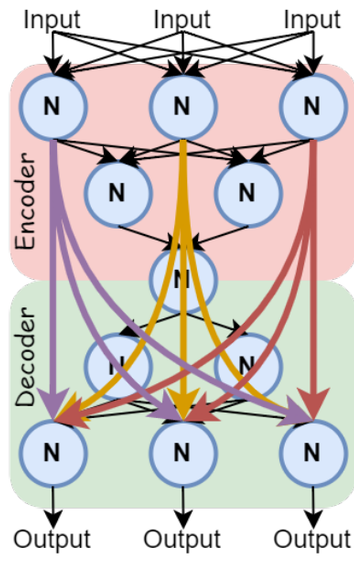


Fig. 15: Beispiel für eine Skip-Connection in einem Autoencoder. Die Eingabe der zweiten versteckten Schicht wird zusammen mit der Ausgabe der vierten Schicht verwendet und in die letzte Schicht eingespeist. Zur besseren Veranschaulichung ist die 'Sprungverbindung' für jedes Neuron unterschiedlich eingefärbt (lila, gelb, rot).

Skip-Connections wurden in einer Vielzahl von Deep-Learning-Aufgaben eingesetzt, darunter Bilderkennung, Verarbeitung natürlicher Sprache und Spracherkennung, und es hat sich gezeigt, dass sie die Leistung von tiefen neuronalen Netzen verbessern. Sie sind besonders nützlich in Netzwerken mit vielen Schichten, wo sie dem Netzwerk helfen können, Merkmale auf mehreren Abstraktionsebenen zu lernen.

7.9 Attention Mechanismen

"Attention" Mechanismen sind eine wichtige Entwicklung im Bereich des Deep Learning, die in einer Vielzahl von Anwendungen eingesetzt werden, darunter maschinelle Übersetzung oder Spracherkennung.

Der Grundgedanke hinter der "Attention" ist, dem Modell zu ermöglichen, sich auf bestimmte Teile der Eingabe zu konzentrieren, die für die anstehende Aufgabe am wichtigsten sind. Dies geschieht, indem jedem Eingabeelement eine Gewichtung zugewiesen wird, die bestimmt, wie viel Aufmerksamkeit das Modell diesem Element schenken sollte. Es gibt verschiedene Arten von Aufmerksamkeitsmechanismen, aber sie alle beinhalten die Berechnung von Aufmerksamkeitsgewichten für Eingabeelemente und deren Verwendung, um einen Kontextvektor zu erzeugen, der die relevantesten Informationen erfasst.

In CNNs wurden beispielsweise Aufmerksamkeitsmechanismen eingesetzt, um sich selektiv auf bestimmte Regionen des Eingangsbildes zu konzentrieren und Aufgaben wie Objekterkennung und Segmentierung durchzuführen. In Transformer-Netzwerken sind Aufmerksamkeitsmechanismen eine Schlüsselkomponente der Architektur und werden zur Berechnung kontextueller Darstellungen der Eingabesequenz für Aufgaben wie maschinelle Übersetzung und Sprachmodellierung verwendet.

7.10 Metriken

Metriken spielen eine entscheidende Rolle bei der Bewertung der Leistung von Deep-Learning-Modellen. Im Bereich der Cyber- und IT-Sicherheit sind genaue und zuverlässige Metriken unerlässlich, um die Wirksamkeit verschiedener Deep-Learning-basierter Ansätze bei der Erkennung und Abwehr von Cyber-Bedrohungen zu messen.

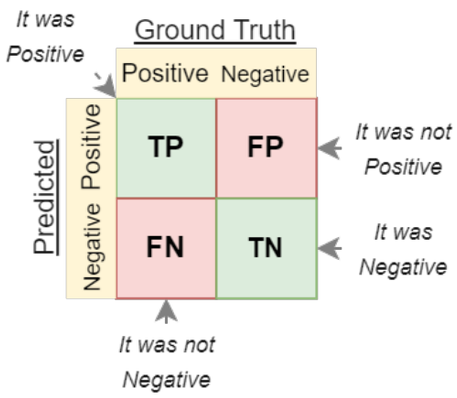


Fig. 16: Dies ist eine einfache Konfusionsmatrix für nur zwei Klassen (True | False). Alle vorhergesagten Klassen können mit der entsprechenden "ground truth" abgeglichen werden.

Eine häufig verwendete Metrik ist die Genauigkeit (Accuracy), die den Prozentsatz der korrekt klassifizierten Instanzen misst. Obwohl die Genauigkeit eine wichtige Kennzahl ist, ist sie für bestimmte Anwendungen nicht immer die am besten geeignete Kennzahl. Bei der Erkennung von Malware (Schad-Software) oder Krebszellen (Medizinbereich) beispielsweise ist die Genauigkeit allein möglicherweise nicht ausreichend, da falsch negative Ergebnisse (nicht erkannte Malware oder Krebszellen) schwerwiegende Folgen haben können. Im wirklichen Leben sind Fälle von Malware im Vergleich zu nicht bösartiger Software jedoch selten. Ein untrainiertes Netzwerk, das Software immer als nicht bösartig einstuft, könnte zwar eine Genauigkeit von über 99% aufweisen, aber seinen Zweck völlig verfehlen. In solchen Fällen sind Metriken wie Recall, Precision und F1-Score möglicherweise besser geeignet.

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

Recall (True-Positive-Rate) misst die Fähigkeit eines Modells, alle positiven Instanzen zu identifizieren, während Precision (Genauigkeit) den Anteil der wahrhaft positiven Instanzen an allen als positiv klassifizierten Instanzen misst. Der F1-Score ist ein harmonisches Mittel aus Recall und Precision und wird häufig verwendet, um den Kompromiss zwischen diesen beiden Metriken auszugleichen.

$$Recall : \frac{TP}{TP + FN}$$

Neben diesen Metriken werden auch andere Metriken wie die ROC-Kurve (Receiver Operating Characteristic) und die AUC-Kurve (Area under the Curve) häufig zur Bewertung der Leistung von Deep-Learning-Modellen im Bereich der Cybersicherheit verwendet. Die ROC-Kurve ist eine grafische Darstellung des Kompromisses zwischen wahrer Positivrate (Sensitivität) und Falsch-Positiv-Rate (1-Spezifität). Der AUC ist eine einzelne Zahl, die die Fläche unter der ROC-Kurve darstellt und ein Gesamtmaß für die Fähigkeit des Modells zur Unterscheidung zwischen positiven und negativen Instanzen liefert. Es ist wichtig, die geeigneten Metriken je nach Aufgabe und den spezifischen Anforderungen der Anwendung zu wählen. In einigen Fällen können auch domänenspezifische Metriken erforderlich sein, um die Leistung des Modells in einem bestimmten Kontext zu bewerten. In jedem Fall sind die sorgfältige Auswahl und Verwendung geeigneter Metriken entscheidend, um sicherzustellen, dass die für die Cybersicherheit entwickelten Deep-Learning-Modelle effektiv und zuverlässig sind.

8 Unsere Netzarchitektur

In unserem Anwendungsfall handelt es sich bei den Eingangsdaten, sowie dem gewünschten Generat, um Text (Programmcode) und damit um Sequenzdaten. Ein solcher Fall wird in im Bereich des Machine Learnings als Seq2Seq bezeichnet (Sequence to Sequence).

Der Output eines solchen Netzes ist jedoch keinesfalls die vollständige gewünschte Sequenz auf einmal, sondern immer nur die Vorhersage für ein Token, wie Beispielsweise ein einzelnes ASCII Zeichen oder ein Wort aus einem vorher angelegten begrenzten Wortschatz. Ein Netz 'kennt' lediglich die Nummerierung der Tokens, hinterlegt in einem Wörterbuch (Datenstruktur 'Dict' in Python).

Die letzte Schicht eines Seq2Seq Netzes ist immer ein Klassifizierer, also eine Anzahl an Neuronen die den möglichen generierbaren Tokens entspricht. Die möglichen Ausgangswerte bei jedem dieser Neurone liegt zwischen Null und Eins. Das Neuron mit dem höchsten Wert (also im besten Fall 1) entscheidet welcher Token der nächste sein wird.

Durch iteratives ausführen eines solchen Netzes bei Eingabe der bereits generierten Tokensequenz kann auf

diese Weise solange generiert werden bis ein festgelegtes 'Stop' Token generiert wird. Hier wird die Iteration beendet und der Prozess ist abgeschlossen. Der Vorgang ist vergleichbar mit dem Schreiben von Text. Stückweise baut sich so, Wort für Wort, ein Text auf.

Tokens: Anfänglich versuchten wir, wie in diesem Kapitel bereits angesprochen, alle Daten die wir haben mit ASCII nummern zu ersetzen. Dies ist ein einfacher und zeitsparender Vorgang und es gibt eine überschaubare Menge unterschiedlicher Tokens (127 ASCII Zeichen + ein 'Stop' Token). Wir bemerkten jedoch das hierbei die Länge der Eingaben, sowohl des PlantUML's sowie des HTML Codes schlichtweg zu groß wurde.
Um unserem Netz etwas Arbeit abzunehmen legten wir darum ein eigens Konzipiertes Wörterbuch an, mit allen Zeichenketten die wir benötigten. Dieses muss den gesamten PlantUML Code, HTML Code für statische Webseiten sowie Bootstrap Klassen und das 'Stop' Token abbilden können und dennoch nicht zu umfangreich werden. Unser Wörterbuch enthält insgesamt 546 Einträge um jede Form von Daten möglichst effizient zu komprimieren. Beispielsweise erstellten wir eine Liste mit allen Bootstrap Klassen, trennten alle Bezeichner bei Trennstrichen und entfernten alle mehrfachen Instanzen von Einträgen. Somit erhielten wir alle Einzelbegriffe aus denen sich jede Bootstrap-Klasse erstellen lässt. Diese Liste kombinierten wir mit den Begriffen für PlantUML's und HTML Code.

Tokens, Ausschnitt - Bootstrap Keywords mit **d**

```
d
danger
dark
deck
decoration
dialog
disabled
dismissible
divider
dropdown
dropleft
dropright
dropup
```

Der Framework Keras bietet einige Methoden zum erstellen eines Tokenizer an. Hiermit war es uns möglich auf Grundlage unserer Begriffsliste einen 'Übersetzer' zu kreieren, der Text-Daten in Zahlen und wieder zurück transformiert. Eine effektive Transformation war möglich durch das umfangreiche Pre-Processing, wie in Kapitel 6 und der dazugehörigen Abbildung 6 bereits erläutert. Der Code wurde teilweise gegen Platzhalter ersetzt (Beispielsweise Einrückungen zu beginn jeder Zeile) und mit zusätzlichen Leerzeichen versehen um eine reibungslose Übersetzung in Tokens zu ermöglichen.

Eingangsdaten: Üblicherweise gibt es bei einem Seq2Seq Anwendungsfall nur ein oder zwei Eingangsdaten für das Netz. Zum einen die Sequenz die man ursprünglich hatte und zum anderen die bereits generierte Sequenz. Diese Daten liegen entweder getrennt oder schlichtweg hintereinander konkateniert vor (in diesem Fall reicht ein Input). In unserem Fall entschieden wir uns jedoch aus einem jeweiligen PlantUML sowohl den Element-Typ wie die Hierarchie/Anordnung der Elemente getrennt zu entnehmen und als separaten Netzinput zu verwenden. Gemeinsam mit der bereits generierten HTML Sequenz erhalten wir somit drei getrennte Daten:
type, order, html (siehe 3 sowie 6).

Netzarchitektur: Die ersten Versuche eines Netzes basierten ausschließlich auf LSTM-Units (siehe Kapitel 7.6) gefolgt von einem Klassifizierer. Nur durch zwei solcher LSTM-Layer ist ein Netz theoretisch in der Lage bereits gute Ergebnisse zu erzielen, vorausgesetzt, das Netz ist Breit genug (Anzahl der LSTM Units in einem Layer). Bei den ersten Versuchen stellte sich heraus, dass das Problem des Overfittings auch durch Methoden wie Regularisierung oder Dropout (siehe Kapitel 7.7) nicht zu lösen ist. Auch wenn erste positive Ergebnisse erziehlt wurden ähnelten Sie in großem Umfang immer den zu letzt trainierten Trainingsdaten (beziehungsweise der zuletzt trainierten Webpage).

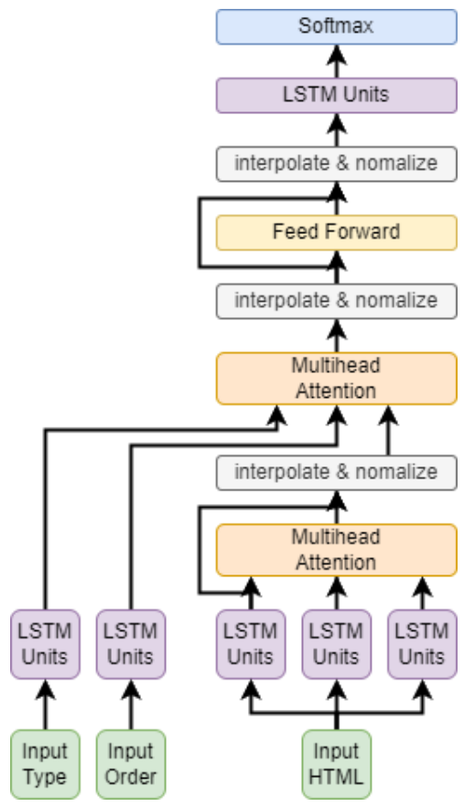


Fig. 17: Unsere entgültige Netzwerkarchitektur. Drei unterschiedliche Input-Daten werden zunächst durch LSTM-Layer vorverarbeitet. Die resultierenden Daten stellen einen abstrakten Informationsraum der ursprünglichen Daten da. Durch Multihead Attention werden diese Daten anschließend Kombiniert, wobei die bisher generierten Tokens (im Bild: *Input HTML*) durch die Information des PlantUML (Element-*Type/Order*) beeinflusst werden. Letztendlich werden die Informationen genutzt um in einem Letzten Layer mit 546 Neuronen eien Vorhersage über das nächste Token zu treffen. Sollte das erste Neuron den höchsten Wert aufweisen entspricht es index 0 unseres Wörterbuches und damit dem *Stop*-Token. Ansonsten wird der Index an die Liste *Input HTML* angehängt und der Vorgang nun wiederholt.

Aus diesem Grund veränderten wir die Netzarchitektur ständig, was ebenfalls neue Daten sowie erneutes Training erforderte. Letztendlich hielten wir uns an den als aktuell 'State of the Art' geltenden Transformer. Er beruht auf einigen Techniken wie Multihead Attention und Skip Connections. Ein großer Nachteil jedoch ist die Größe der Eingangsdaten, denn diese müssen üblicherweise eine bekannte und damit festgelegte Größe (und Dimension) aufweisen. Dies tun Sequenzdaten jedoch selten und erfordern somit ein umfangreicheres Padding der Daten. Für unseren Fall tauschten wir also die ersten Layer der standartmäßigen Transformer-Architektur gegen LSTM-Units aus, was zwar zu mehr trainierbaren

Gewichten/Parametern führte, jedoch dynamische Längen der Eingangsdaten ermöglichte. Eine detaillierte Übersicht des Netzes ist in Abbildung 17 zu sehen.

9 Ergebnisse

Ein neuronales Netzwerk wird initial (üblicherweise) mit zufälligen Gewichten als Verbindung zwischen den Neuronen erstellt. Erst durch das Training, beispielsweise mit Trainingspaaren aus Daten Input und erwünschten Output, erhält das Netz schrittweise die gewünschte Funktionalität. In unserem Fall entspricht ein trainiertes Netz welches seine Aufgabe erfüllt dem **Codegenerator**. Das Training selbst spiegelt also den **3. Meilenstein** wieder.

9.1 Ablauf des Trainings zum Erzeugen der Ergebnisse

Im ersten Ansatz des Trainings setzten wir auf ein Epochentraining. Dabei wurde pro Trainingsiteration ein zufälliges Trainingsobjekt aus den 30 Trainingsdaten ausgewählt Im Rahmen dieser Arbeit wurde die Funktion `create_dataset()` implementiert und das Modell daraufhin für jeweils 5 Epochen trainiert. t. Die Batchsize betrug dabei 16, was in unserem Netz bedeutete, dass jeweils die ersten 16 Token gleichzeitig eingelesen wurden. Wir betrachten die folgenden Ergebnisse.

Die Funktion `create_dataset()` wurde entwickelt, um das Datenset für das Training des Modells vorzubereiten. Dabei wurden verschiedene Schritte wie Datenladen, Datenverarbeitung und Aufteilung in Trainings- und Testdaten durchgeführt. Die genaue Implementierung der Funktion kann dem Quellcode entnommen werden.

9.2 Training des Modells

Das Modell wurde für insgesamt 5 Epochen trainiert, um eine ausreichende Anpassung an die Daten zu gewährleisten. Die Wahl der Epochenanzahl kann je nach Problemstellung und Datenmenge variieren. In diesem Fall wurde eine Trainingseinheit auf 5 Epochen festgelegt.

Die Batchsize wurde während des Trainings auf *Batchsize* gesetzt. Die Batchsize bestimmt die Anzahl der Datensätze, die in einem einzigen Durchlauf des Trainingsalgorithmus verarbeitet werden. Eine angemessene Batchsize kann das Training beschleunigen und den Speicherbedarf optimieren. dabei 16, was in unserem Netz bedeutete, dass jeweils die ersten 16 Token gleichzeitig eingelesen wurden. Wir betrachten die Ergebnisse aus der Figure Trainingsergebnisse im Anhang.

Hierbei konnten wir nach 1,5 Stunden Training mit jeweils 100 Iterationen ein Overfitting-Problem feststellen. Die Genauigkeit des Netzes auf dem Trainingssatz nahm mit jeder Epoche schnell zu und erreichte schnell Werte von über 90auf die spezifischen Trainingsdaten überanpasste und die Ergebnisse auf anderen Trainingspaaren deutlich schlechter wurden. Es kam also zu einem dauerhaften Overfitting auf das aktuelle Trainingspaar. Aus diesen Erkenntnissen zogen wir zwei Konsequenzen. Erstens erweiterten wir unser Netz um Dropouts, die wir zwischen dem Encoder und dem Decoder platzierten. Zweitens verzichteten wir darauf, die Trainingspaare zufällig in das Netz einzuspeisen, sondern fütterten sie der Reihe nach ein. Dies reduzierte das Auftreten von Dopplungen und führte zu besseren Ergebnissen. Für das Modell "lastChanceAlternative.h5" konnten wir durch eine Verringerung der Batchsize auf 8 sogar bei nur einer Epoche pro Trainingsdatensatz dafür sorgen, dass die Gesamtgenauigkeit bei über 90 Prozent. /

9.3 Analyse einer generierten Seite durch unser Netz

Wir betrachten dazu Abbildung 4, welche durch unser Modell *"lastChanceAlternative.h5"* generiert wurde. Dieses Modell basiert auf dem Trainingspaar *"2.txt"*. Insgesamt hat das Modell eine zufriedenstellende Leistung gezeigt, indem es viele Elemente richtig erkannt und die Struktur der HTML-Seite weitgehend korrekt wiedergegeben hat. Es ist wichtig zu betonen, dass das Modell einige Bereiche gut gemeistert hat.

```
<body>
    <header id="Image" href="football.jpg" alt="Football Image" class=" card bg-primary ">
        Goaaaaal!
        <div class=" card-body ">
            <div class=" form-group ">
                <input id="InputName" type="text" name="name" placeholder="name" class=" form-control ">
            </div>
            <div class=" form-group ">
                <input id="Navigation" class=" form-control ">
            </div>
            <button id="Main" class=" btn btn-primary "> </button >
            </form >
          </p >
        </div >
      </div >
    </div >
  </section >
</main >
<footer id = "ListItem2" class = " container " >
   UEFA Champions League<div class = " row " >
    <div class = " col-md-" >
      <div class = " card bg-light " >
        <div class = " card-body " >
          <select id = "Option3"  value = "option3" class = " form-control bg-info text-white " >
             Option 3<option > </option >
            <option > </option >
            <option > </option >
          </select >
        </div >
      </div >
    </div >
  </footer >
</body >

</html>
```

Fig. 18: Ergebnis der Generierung durch das Modell "lastChanceAlternative.h5" basierend auf dem Trainingspaar "2.txt". Das Modell erkennt die grundlegende Struktur der HTML-Seite und viele HTML-Elemente korrekt. Es beweist die Funktionialität des Netzes

Es hat die allgemeine Struktur des HTML-Dokuments richtig erkannt, insbesondere die erforderlichen HTML-Elemente wie *"body"*, *"header"*, *"footer"* und *"section"*, die in der richtigen Konstellation erzeugt wurden. Dies zeigt, wie das Netzwerk wesentliche Strukturen lernen und verarbeiten kann. Auch konnte durch richtiges Postprocessing die Verlinkungen und auch die IDs richtig zugeordnet werden. Die Listenstruktur mit den `<ul>` und `<li>`-Objekten ist ebenfalls erfolgreich erzeugt worden. Auch die Bildintegration wurde korrekt durchgeführt, indem die Bilder mit den richtigen Attributen eingebettet wurden. Die allgemeine Struktur der Formularerstellung mit den `<form>`, `<input>` und `<button>`-Objekten kann als korrekt klassifiziert werden.

Dennoch gibt es einige Bereiche, die durch weiteres Training mit mehr vergleichbaren Eingangsdaten verbessert werden könnten. Hierbei spielen die Angaben von Attributen eine große Rolle. Auch werden einige Objekte ignoriert und nicht in die Hierarchie aufgenommen, wie z.B. `<div class="card-body">`. Zudem werden nicht immer alle Attribute vernünftig übernommen und zugeordnet.

Es lässt sich hier erkennen, dass das allgemeine Ergebnis erfolgreich war. Die grundlegende Struktur ist korrekt erkannt und viele der HTML-Elemente sind richtig identifiziert worden. Es gibt dennoch Raum für Verbesserungen. Hierbei ist jedoch sicher, dass durch mehr Trainingsdaten noch genauere Ergebnisse erzielt werden könnten.

9.4 Aktuelle Probleme

Problem der 'endlosen' Generierung in Listen Betrachten wir hier ein Ergebnis für die Generierung für das PlantUML unter:

`\Input_data\training_pairs\plantUML\1.txt`

Hierbei haben wir das neuronale Netz *webgene.h1* verwendet.

```
<ul class = " navbar - nav ml - auto " >
    <li class = " nav - item " >
        <a id22 class = " nav - link text - dark " > </a >
    </li >
    <li class = " nav - item " >
        <a id15 class = " nav - link text - dark " > </a >
    </li >
    <li class = " nav - item " >
        <a id9 class = " nav - link text - dark " > </a >
    </li >
    <li class = " nav - item " >
        <a id9 class = " nav - link text - dark " > </a >
    </li >
    <li class = " nav - item " >
        <a id9 class = " nav - link text - dark " > </a >
    </li >
    <li class = " nav - item " >
        <a id9 class = " nav - link text - dark " > </a >
    </li >
    <li class = " nav - item " >
        <a id9 class = " nav - link text - dark " > </a >
    </li >
    <li class = " nav - item " >
        <a id9 class = " nav - link text - dark " > </a >
    </li >
    <li class = " nav - item " >
        <a id9 class = " nav - link text - dark " > </a >
```

Fig. 19: Abbildung 2: Problem bei der Generierung der Navbar durch das neuronale Netzwerk.

Im vorliegenden Beispiel zeigt das neuronale Netz eine korrekte Generierung für den "head" und den "body" des HTML-Codes. Diese Teile werden erfolgreich aufgebaut, und die Token-Vorhersage ist genau. Allerdings gibt es ein Problem beim Generieren der Navbar, wie in Abbildung 19 zu sehen ist.

Die Navbar wird wie erwartet mit dem richtigen Anfangs-Tag `<ul class="navbar-nav ml-auto">` aufgebaut. Innerhalb der Navbar werden auch die einzelnen Listenelemente (`<li>`) richtig vorhergesagt, was vielversprechend aussieht. Das Problem tritt jedoch auf, wenn es darum geht, die Navbar zu schließen. Statt das Ende der Navbar korrekt zu generieren, wird stattdessen immer wieder ein neues Listenelement (`<li>`) vorhergesagt, was zu einer Endlosschleife führt. Dieses Verhalten entsteht, weil das Netzwerk wahrscheinlicher davon ausgeht, dass ein weiteres Listenelement folgt, anstatt die Navbar zu beenden. Die Vorhersage wird somit in einer Schleife gefangen und kann die korrekte Struktur nicht abschließen.

Ein großer Faktor spielt hierbei natürlich, dass es in den Trainingsdaten eine höhere Anzahl an Elementen gibt, welche nachfolgende Elemente von Listenelementen sind, als den Listenabschluss selbst.

```
def generator(plant_uml):
    # GENERATE: html
    with open("PlantUML_versuch/gpt_prompt_1.txt", "r") as file:
        prompt = file.read()
    response = generate_chat_response(plant_uml + prompt)
    response = cut_body(response)
    response = remove_html_comments(response)
    response = remove_attribute("style", response)
    response = remove_attribute("script", response)

    # GENERATE: more divs
    with open("PlantUML_versuch/gpt_prompt_2.txt", "r") as file:
        prompt = file.read()
    response = generate_chat_response(response + prompt)
    response = cut_body(response)
    response = remove_html_comments(response)
    response = remove_attribute("style", response)
    response = remove_attribute("script", response)

    # GENERATE: bootstrap cards
    with open("PlantUML_versuch/gpt_prompt_3.txt", "r") as file:
        prompt = file.read()
    response = generate_chat_response(response + prompt)
    response = cut_body(response)
    response = remove_html_comments(response)
    response = remove_attribute("style", response)
    response = remove_attribute("script", response)
    print("50% from generator done....")

    # GENERATE: bootstrap grid system
    with open("PlantUML_versuch/gpt_prompt_4.txt", "r") as file:
        prompt = file.read()
    response = generate_chat_response(response + prompt)
    response = cut_body(response)
    response = remove_html_comments(response)
    response = remove_attribute("style", response)
    response = remove_attribute("script", response)

    # GENERATE: bootstrap colors
    with open("PlantUML_versuch/gpt_prompt_5.txt", "r") as file:
        prompt = file.read()
    response = generate_chat_response(response + prompt)
    response = cut_body(response)
    response = remove_html_comments(response)
    response = remove_attribute("style", response)
    response = remove_attribute("script", response)

    return response
```

Fig. 20: Abbildung 3:Automatisierte Überarbeitung der durch OpenAI generierten Daten.

Um dieses Problem zu beheben, könnten verschiedene Ansätze verfolgt werden. Eine Möglichkeit besteht darin, dem Netzwerk beizubringen, dass die Navbar nach einer bestimmten Anzahl von Listenelementen abgeschlossen wird, oder dass es spezielle Endmarker für die Navbar verwendet, die ihm signalisieren, dass es die Navbar abschließen sollte.

Es ist wichtig, dieses Verhalten im Trainingsprozess zu berücksichtigen und das Modell so anzupassen, dass es die Hierarchie der HTML-Elemente korrekt erfasst und keine Endlosschleifen erzeugt. Nur durch die Behebung dieses Problems kann das neuronale Netzwerk zuverlässig und konsistent korrekten PlantUML HTML-Code generieren.

9.5 Schlussfolgerungen aus dem Training

1. Mehr Trainingsdaten Wie unser Team bereits in der Präsentation erwähnt hat, war es unsere Absicht, mit OpenAI eine große Menge an Webseiten und passendem PLANT-UML-Code zu generieren. Allerdings stießen wir auf Schwierigkeiten, da trotz mehrfacher automatisierter Überarbeitung (siehe Abbildung 20) die durch OpenAI generierten Daten immer noch viele Fehler aufwiesen. Dadurch war es nicht möglich, in einem schnellen und automatisierten Vorgang eine ausreichende Menge an qualitativ hochwertigen Daten zu erzeugen. Jedes der generierten Daten benötigte weitere Überarbeitungsschritte, um die Fehler zu korrigieren.

Obwohl wir am Ende über einen vielversprechenden Datensatz von 30 Trainingspaaren verfügten, stellte sich heraus, dass diese Datenmenge nicht ausreichend war, um eine umfassende Abdeckung der HTML-Strukturen und Varianten zu gewährleisten. Das führte schnell zu einem Overfitting-Problem, bei dem das Modell in Endlosschleifen geriet und repetitive Strukturen erzeugte. Dadurch war es nicht in der Lage, komplexe und vielfältige Codeabschnitte zu erstellen und angemessen zu verarbeiten.

2. Größe und Komplexität des Netzwerks Eine weitere Fehleinschätzung zu Beginn der Arbeit war, dass das Netz wesentlich größer und komplexer wurde als ursprünglich geplant. Wie in Abbildung 21 zu sehen ist, hat dieses Netz weit über 22 Millionen Parameter. Das Training und die Trainingszeit für eine Iteration mit allen Daten können dabei schon mit der uns zur Verfügung stehenden GeForce RTX 3060 bei geringer Batchsize mehrere Stunden dauern. Für die Zukunft wäre es ratsam, Server zu mieten, die Tensor Processing Units (TPUs) zur Verfügung stellen. Gerade wenn die Anzahl der Trainingsdaten steigern würde, würde dies die Trainingszeit erheblich verkürzen und die Effizienz des Modells steigern.

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, None, 547)]       0         []

 input_2 (InputLayer)        [(None, None, 547)]       0         []

 html_input (InputLayer)     [(None, None)]            0         []

 reshape_type (Reshape)      (None, None, 1)           0         ['input_1[0][0]']

 reshape_order (Reshape)     (None, None, 1)           0         ['input_2[0][0]']

 hytml_embed (Embedding)     (None, None, 547)         299209    ['html_input[0][0]']

 conv_type (Conv1D)          (None, None, 1024)        3072      ['reshape_type[1][0]']

 conv_order (Conv1D)         (None, None, 1024)        3072      ['reshape_order[1][0]']

 encoder_query (LSTM)        (None, None, 547)         2395860   ['hytml_embed[1][0]']

 encoder_value (LSTM)        (None, None, 547)         2395860   ['hytml_embed[1][0]']

 encoder_key (LSTM)          (None, None, 547)         2395860   ['hytml_embed[1][0]']
...
Total params: 21,337,188
Trainable params: 21,337,188
Non-trainable params: 0
```

Fig. 21: Abbildung 4: Die Komplexität unseres Models übersteigt mehr als 20 milionen Paramter

3. Batchsize Aktuell wird im Training die Batchsize als die Anzahl von Tokens definiert, die pro Trainingsiteration eingelesen werden. Allerdings ist diese Methode ineffizient. Eine bessere Herangehensweise wäre es gewesen, die Batchsize auf die Größe eines einzelnen Trainingsdatensatzes anzupassen, was viele Vorteile mit sich gebracht hätte.

Ein wesentlicher Vorteil wäre eine erheblich effizientere Nutzung der Hardware-Ressourcen. Das Training hätte in deutlich weniger Iterationen viel mehr Fortschritte erzielen können. Durch parallele Verarbeitung wären auch deutlich mehr Trainingspaare gleichzeitig trainiert worden, was einen weiteren Vorteil darstellt. Durch die gleichzeitige Verarbeitung mehrerer Trainingspaare würde nicht nur die Effizienz des Trainings verbessert, sondern auch das Overfitting reduziert. Die Modelle würden nicht mehr ausschließlich auf einzelne Muster fixiert, da mehrere Paare gleichzeitig trainiert werden. Dies hätte zu einer besseren Generalisierung des Modells geführt. Zudem würde durch das Training mehrerer Trainingspaare eine gleichmäßigere Gewichtsaktualisierung stattfinden. Somit würde nicht ständig zwischen verschiedenen Teilen des HTML-Files wie dem Header und dem Body hin und her gewechselt werden, was zu Schwankungen und Overfitting führen könnte.

Zusätzlich können beim gleichzeitigen Einlesen von Daten bestimmte Muster wie z.B. Navigationsleistenstrukturen besser erkannt werden. Durch die Betrachtung mehrerer Trainingsdaten auf einmal würden bestimmte zusammenhängende Muster als einfachere Muster erkannt werden können. Mit dieser Methode könnten also die Trainingsressourcen effizienter genutzt, Overfitting verhindert und die Generalisierungsfähigkeit des Modells verbessert werden.

10 Fazit

Das Resultat zeigt sich positiv in Bezug auf den Codegenerator. Mit im Endeffekt nur kleinen Mängeln zeigt sich, dass mit mehr Trainingsdaten ein erwünschtes Ergebnis erreicht worden wäre. Häufige Umstrukturierungen in der Vorgehensweise und das umständliche generieren der Daten waren jedoch zu Zeitintensiv um das Projekt vollständig umzusetzen. Wären zu einem früheren Zeitpunkt erfolgreich Webpages generiert worden, hätte ein Modelldiscoverer aus einer Ordnerstruktur mit allen nötigen Ressurcen (Beispielsweise Bilddateien limitierter Dateiformate) mehrere separate PlantUML's (pro Webpage) erstellen sollen. Diese hätte man anschließend einzeln mit unserem Netz generiert und in unserem Post-Processing Schritt mit den nötigen Verlinkungen untereinander versehen.

References

1. Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.
2. David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.
3. Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20:121–136, 1975.
4. Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.
5. Pascal Vincent, H. Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *J. Mach. Learn. Res.*, 11:3371–3408, 2010.
6. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.
7. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
8. Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Understanding regularization in deep learning. *arXiv preprint arXiv:1801.05134*, 2018.
9. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 06 2016.

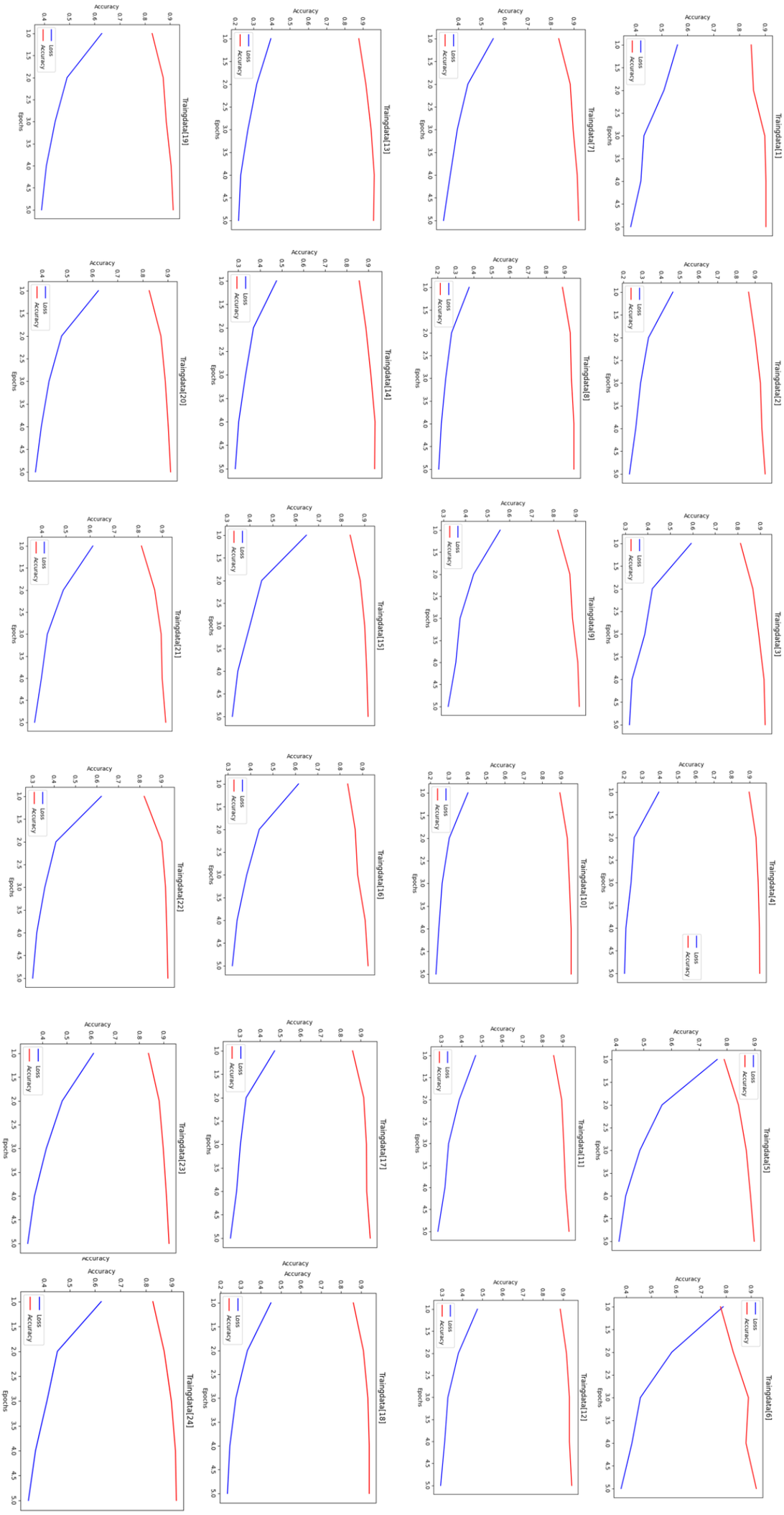


Fig. 22: Trainingsergebnisse