UNIVERSITÄT SIEGEN

Clientseitiges Deep Learning durch Klassifizierung von deutschsprachigen Clickbaits

Ugur Tigu

Master-Thesis

zur Erlangung des akademischen Grades Master of Science (M.Sc.)

Studiengang Wirtschaftsinformatik

Fakultät IV - Institut für Wissensbasierte Systeme und Wissensmanagement

Universität Siegen

24. Dezember 2020

Betreuer

Prof. Dr.-Ing. Madjid Fathi, Universität Siegen

Johannes Zenkert, Universität Siegen

Tigu, Ugur:
Clientseitiges Deep Learning durch Klassifizierung von deutschsprachigen Clickbaits / Ugur Tigu. –
Master-Thesis, Aachen: Universität Siegen, 2020. 35 Seiten.

Tigu, Ugur:
Client-side deep learning through classification of German clickbaits / Ugur Tigu. –
Master Thesis, Aachen: University of Siegen, 2020. 35 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Universität Siegen öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Aachen, 24. Dezember 2020

Ugur Tigu

Abstract

Clientseitiges Deep Learning durch Klassifizierung von deutschsprachigen Clickbaits

Ein im Internet weit verbreitetes Phänomen sind *Clickbaits-Nachrichten* (auf deutsch „Klickköder“). Ziel dieser Arbeit ist die Entwicklung eines Deep Learning Verfahrens, welches deutsche Clickbait Nachrichten automatisch erkennen soll. Die Arbeit stellt einen Datensatz vor, welches aus zwei Klassen von Nachrichten Überschriften besteht und zum trainieren eines Deep Learning Ansatzes verwendet wird. Dieser Datensatz wird durch Web Scraping erstellt und gelabelt. Das Ergebnis dieser Arbeit ist ein Modell für die Textklassifizierung, entwickelt in TensorFlow.js. Dieses Modell wird vollständig clientseitig in den Browser eingebettet und benötigt somit keinen Server.

Client-side deep learning through classification of German clickbaits

A widespread phenomenon on the Internet are clickbaits. The aim of this thesis is the development of a deep learning model which should automatically recognize German clickbait titles. The thesis presents a data set, which consists of two classes of news headlines and is used to train a deep learning approach. This data set is created using web scraping and hand-labeled. The result of this work is a model for text classification, developed in TensorFlow.js. This model is completely embedded in the browser on the client side and therefore does not require a server.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Kapitel 2

Deep Learning

2.1 Einleitung

Künstliche neuronale Netze stellen eine Klasse von Modellen des maschinellen Lernens dar, die vom Zentralnervensystem von Säugetieren inspiriert sind. Jedes Netz besteht aus mehreren miteinander verbundenen „Neuronen“, die in „Schichten“ organisiert sind. Neuronen in einer Schicht leiten Nachrichten an Neuronen in der nächsten Schicht weiter. Erste Studien wurden in den frühen 50er Jahren mit der Einführung des „Perzeptrons“ [1] begonnen, eines zweischichtigen Netzwerks, das für einfache Operationen verwendet wird, und in den späten 60er Jahren mit der Einführung des „Back-Propagation-Algorithmus“ (effizientes mehrschichtiges Netzwerktraining) (gemäß [2], [3]) weiter ausgebaut.

Deep Learning ermöglicht es Computermodellen, die aus mehreren Verarbeitungsebenen bestehen, Darstellungen von Daten mit mehreren Abstraktionsebenen zu erlernen. Diese Verfahren haben den Stand der Technik in Bezug auf Spracherkennung, visuelle Objekterkennung, Objekterkennung und viele andere Bereiche wie die Genomik dramatisch verbessert. Deep Learning entdeckt komplexe Strukturen in großen Datenmengen, indem es den Backpropagation-Algorithmus verwendet. Dieses ist ein Verfahren, welches der Maschine zeigt, wie sie ihre internen Parameter ändern soll, mit denen die Darstellung in jeder Schicht aus der Darstellung in der vorherigen Schicht berechnet wird. Tiefe Faltungsnetze haben Durchbrüche bei der Verarbeitung von Bildern, Video, Sprache und Audio gebracht, während wiederkehrende Netze sequentielle Daten wie Text und Sprache beleuchtet haben [4].

2.2 Überwachtes Lernen

Die häufigste Form des maschinellen Lernens, ob tief oder nicht, ist das überwachte Lernen. Beim überwachten Lernen werden zunächst große Mengen an Daten gesammelt, die jeweils mit ihrer Kategorie gekennzeichnet sind. Währen des Trainings wird der Maschine ein Exemplar aus den Daten gezeigt und es wird eine Ausgabe in Form eines Bewertungsvektors erzeugt, einer für jede Kategorie. Der Wunsch ist es, für jede Kategorie die höchste Punktzahl zu erreichen. Es wird eine Zielfunktion berechnet, die den Fehler (oder Abstand) zwischen den Ausgabewerten und dem gewünschten Bewertungsmustern misst. Die Maschine ändert dann ihre internen einstellbaren Parameter um, um den Fehler zu minimieren. Diese einstellbaren Parameter werden **Gewichte** genannt. Gewichte sind reelle Zahlen die als „Knöpfe“ gesehen werden können, die die Eingabe-Ausgabe-Funktion der Maschine ist. In einem typischen Deep-Learning-System können Hunderte Millionen dieser einstellbaren Gewichte vorkommen [4]. Neben den Gewichten taucht der Begriff **Bias** auf, welches im Deep Learning ein Maß dafür ist, das Perzeptron zum „feuern zu bringen“ [5, S. 7].

2.3 Das Perzeptron

Das Perzeptron ist ein einfacher Algorithmus mit einem Eingabevektor x mit m Werten $(x_2, ..., x_m)$. Es wird oft als „Eingabe-Features“ oder einfach als „Features“ bezeichnet. Es gibt entweder eine 1 „Ja“ oder eine 0 „Nein“ zurück (siehe Formel 2.1). Dabei ist w ein Vektor welches das Gewicht darstellt, und wx das Punktprodukt aus $\sum_{j=1}^{m} w_j x_j$, b ist der Bias. Aus $wx + b$ ist die Grenzhyperebene definiert, die die Position gemäß den w und b zugewiesenen Werten ändert.

$$fx = \begin{cases} 1 & wx + b > 0 \\ 0 & ansonsten \end{cases} \tag{2.1}$$

Beispielsweise kann das Perzeptron bei drei Eingabemerkmalen (Rot, Grün und Blau) unterscheiden, ob die Farbe weiß ist oder nicht. Es soll beachtet werden, dass das Perzeptron keine „Vielleicht“-Antwort ausdrücken kann. Es kann mit „Ja“ (1) oder „Nein“ (0) antworten. Das Perzeptron-Modell kann dafür verwendet werden, indem durch Anpassung von w und b, das Modell trainiert wird.

2.4 Mehrschichtiges Perzeptron

Wenn ein Modell nicht nur eine einzige lineare Schicht hat, sondern wenn mehrere Schichten in Form von Perzeptronen zusammengebracht werden, handelt es sich dabei um ein mehrschichtiges Perzeptron. Die Eingabe- und Ausgabeebene ist von außen sichtbar, während alle anderen Ebenen in der Mitte ausgeblendet oder verborgen sind (hidden layers). Es werden mehrere lineare Funktionen (einzelne Schichten) nacheinander gestapelt und somit eine mehrschichtiges Perzeptron erzeugt.

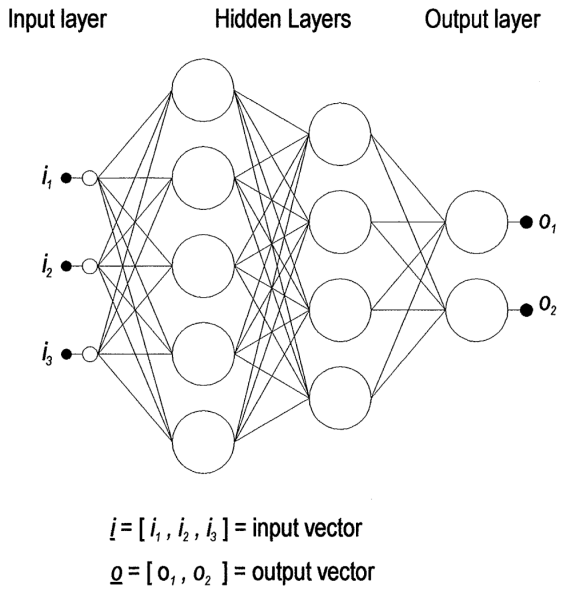

Abbildung 2.1: Das mehrschichtige Perzeptron mit zwei verborgenen Schichten, entnommen aus [6]

2.5 Sigmoid-Neuron

Neben dem Perzeptron gibt es das Sigmoid Neuron. Das Perzeptron gibt bekannt nur eine 0 oder eine 1 zurück. Es ist also eine Funktion nötig, die sich ohne ohne Diskontinuität schrittweise von 0 auf 1 ändert. Mathematisch bedeutet dies, dass eine stetige Funktion nötig ist, mit der die Ableitung berechnet werden kann. Dieses Problem kann überwunden werden, indem einen neuer Typ eines künstlichen Neurons eingeführt wird, der als **Sigmoid-Neuron** bezeichnet wird. Sigmoid Neuronen ähneln Perzeptronen, sind jedoch so modifiziert, dass kleine Änderungen ihres Gewichts und ihres Bias nur eine geringe Änderung ihrer Leistung bewirken. Dies ist der Grund dafür, dass Netzwerke lernen können [5, S. 8].

Genau wie ein Perzeptron hat das Sigmoid-Neuron die Eingaben $x_1, x_2, ...$, aber anstatt nur 0 oder 1 zu sein, können diese Eingänge auch beliebige Werte zwischen 0 und 1 annehmen. Also zum Beispiel 0,123 welches eine gültige Eingabe für ein Sigmoid-Neuron ist. Ebenso wie ein Perzeptron hat das Sigmoid-Neuron Gewichte für jede Eingabe, $w_1, w_2, ...$ und einen Bias, b. Die Ausgabe ist jedoch nicht 0 oder 1, stattdessen ist es σ, $(wx + b)$, wobei σ als Sigmoidfunktion bezeichnet wird und durch Formel 2.2 definiert ist.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

2.6 Aktivierungsfunktionen

Ohne eine **Aktivierungsfunktion** (auch als Nichtlinearität bezeichnet) würde die dichte Schicht (dense layer) nur aus zwei linearen Operationen bestehen - einem Punktprodukt und einer Addition: $Ausgabe = Punkt(w, Eingabe)+b$. Die Schicht konnte also nur lineare Transformationen (affine Transformationen) der Eingabedaten lernen. Um Zugang zu einem viel umfangreicheren Hypothesenraum zu erhalten, wird eine Nichtlinearitäts- oder Aktivierungsfunktion benötigt [7, S. 72].

2.6.1 Sigmoid

Die Sigmoidfunktion wird mit der Formel 2.2 definiert und in der Abbildung 2.2 dargestellt, die Ableitung der Sigmoidfunktion wird in der Formel 2.3 definiert. Sie hat kleine Ausgangsänderungen im Bereich (0, 1), wenn der Eingang im Bereich $(-\infty, \infty)$ variiert. Mathematisch ist die Funktion stetig. Ein Neuron kann das Sigmoid zur Berechnung der nichtlinearen Funktion $\sigma(z = wx + b)$ verwenden. Wenn $z = wx + b$ sehr groß und positiv wird, dann wird $e^z \to 0$ also $\sigma(z) \to 1$, während wenn $z = wx + b$ sehr groß und negativ wird, wird $e^{-z} \to 0$ also $\sigma(z) \to 0$. Mit anderen Worten, ein Neuron mit Sigmoidaktivierung hat ein ähnliches Verhalten wie das Perzeptron, aber die Änderungen sind allmählich und Ausgabewerte wie 0,54321 oder 0,12345 sind vollkommen legitim. In diesem Sinne kann ein Sigmoid-Neuron auch mit „vielleicht“ antworten [8, S. 10].

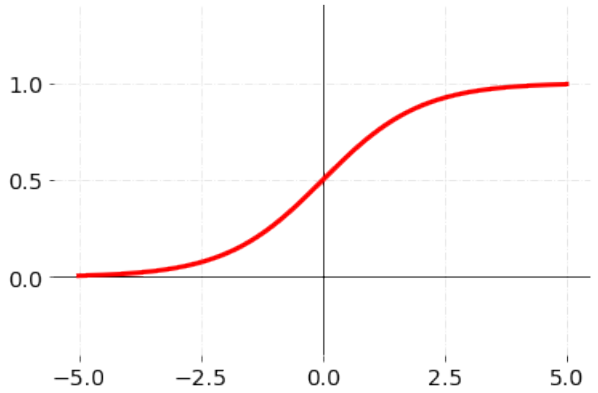

Abbildung 2.2: Darstellung der Sigmoid-Aktivierungsfunktion (eigene Darstellung)

$$\sigma'(z) = \frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = \frac{1}{(1+e^{-z})^{-2}}\frac{d}{dz} = (e^{-z}) = \frac{e^{-z}}{(1+e^{-z})}\frac{1}{(1+e^{-z})} =$$

$$\frac{e^{-z}+1-1}{(1+e^{-z})}\frac{1}{(1+e^{-z})} = \left(\frac{(1+e^{-z})}{(1+e^{-z})} - \frac{1}{(1+e^{-z})}\right)\frac{1}{(1+e^{-z})} = \tag{2.3}$$

$$\left(1 - \frac{1}{(1+e^{-z})}\right)\left(\frac{1}{(1+e^{-z})}\right) = (1-\sigma(z))\sigma(z)$$

2.6.2 Tanh

Die Tanh-Aktivierungsfunktion wird mit der Formel 2.4 definiert ihre Ableitung wird in der Formel 2.5 berechnet. Sie hat ihre Ausgangsänderungen im Bereich (-1, 1). Sie hat eine Struktur, die der Sigmoid-Funktion sehr ähnlich ist. Der Vorteil gegenüber der Sigmoidfunktion besteht darin, dass ihre Ableitung steiler ist, was bedeutet, dass sie mehr Werte enthalten kann (vergleiche Abbildung 2.3). Für die Ableitung der Tanh-Aktivierungsfunktion gilt, $e^z = \frac{d}{dz}e^z$ und $e^{-z} = \frac{d}{dz}e^{-z}$.

$$tanh(z) = \frac{e^z - e^{-z}}{e^z - e^{-z}} \tag{2.4}$$

$$\frac{d}{dz}tanh(x) = \frac{\left(e^z+e^{-z}\right)\left(e^z+e^{-z}\right)-\left(e^z-e^{-z}\right)\left(e^z-e^{-z}\right)}{\left(e^z+e^{-z}\right)^2} = \tag{2.5}$$

$$1 - \frac{\left(e^z-e^{-z}\right)^2}{\left(e^z+e^{-z}\right)^2} = 1 - tanh^2(z)$$

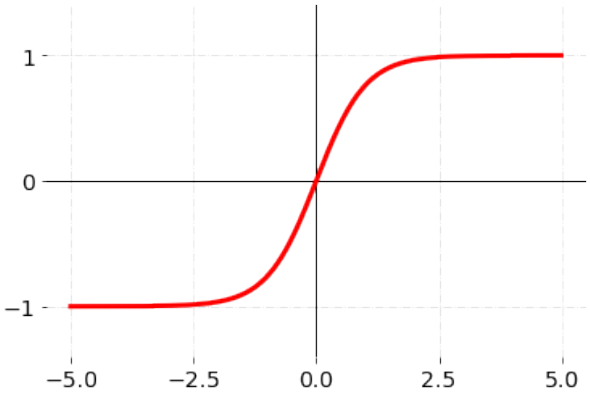

Abbildung 2.3: Darstellung der Tanh-Aktivierungsfunktion (eigene Darstellung)

2.6.3 ReLu

Vor kurzem wurde eine sehr einfache Funktion namens ReLU (REctified Linear Unit) sehr beliebt, da sie dazu beiträgt, einige Optimierungsprobleme die bei Sigmoiden beobachtet werden, zu lösen [8, S. 11]. Sie wird in Formel 2.6 definiert, die zugehörige Ableitungsfunktion wird in der Formel 2.7 berechnet. Wie in Abbildung 2.4 zu sehen, ist die Funktion für negative Werte Null und wächst für positive Werte linear. Die ReLU ist relativ einfach zu implementieren.

$$f(x) = \begin{cases} 0 & wenn\ x < 0 \\ x & wenn\ x \geq 0 \end{cases} \tag{2.6}$$

$$f'(x) = \begin{cases} 1, & wenn\ x > 0 \\ 0, & sonst \end{cases} \tag{2.7}$$

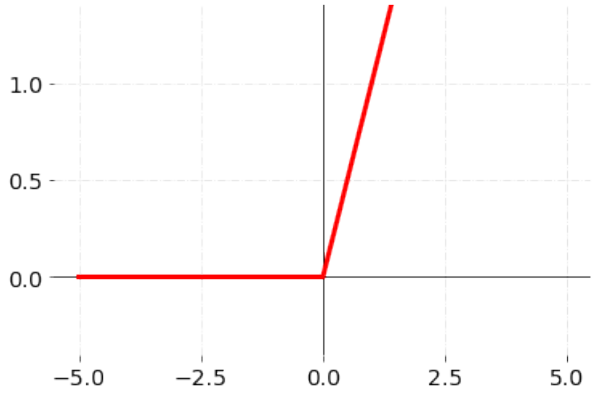

Abbildung 2.4: Darstellung der ReLu-Aktivierungsfunktion (eigene Darstellung)

2.6.4 softmax

Die grundlegende Schwierigkeit bei der Durchführung kontinuierlicher Mathematik auf einem digitalen Computer besteht darin, dass unendlich viele reelle Zahlen mit einer endlichen Anzahl von Bitmustern dargestellt werden müssen. Dabei entstehen Rundungsfehler die problematisch sein können, insbesondere wenn sie sich über viele Operationen hinweg zusammensetzen. Sieführen dazu, dass theoretisch funktionierende Algorithmen in der Praxis fehlschlagen. Ein Beispiel für eine Funktion, die gegen Rundungsfehler stabilisiert, ist die softmax-Funktion [9, S. 80–81].

$$softmax(x)_i = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)} \tag{2.8}$$

2.7 Optimierungsalgorithmen

Die meisten Deep-Learning-Algorithmen beinhalten irgendeine Art von Optimierung. Optimierung bezieht sich auf die Aufgabe, eine Funktion $f(x)$ durch Ändern von x entweder zu minimieren oder zu maximieren. Die meisten Optimierungsprobleme werden in Bezug auf Minimierung von $f(x)$ formuliert. Die Maximierung kann über einen Minimierungsalgorithmus durch Minimieren von $-f(x)$ erreicht werden. Die Funktion die minimiert oder maximiert werden soll, wird als **Objektive Funktion** bezeichnet. Es wird auch **Kostenfunktion** oder **Verlustfunktion** bezeichnet. Zum Beispiel ist die Funktion $x^* = arg\ min\ f(x)$ eine solche Funktion.

2.7.1 Gradientenverfahren

Der Gradientenabstieg ist einer der beliebtesten Algorithmen zur Optimierung und bei weitem der häufigste Weg zur Optimierung neuronaler Netze. Der Gradientenabstieg ist ein Weg, um eine Zielfunktion $J(\theta)$ zu minimieren, die durch die Parameter eines Modells $\theta \in \mathbb{R}^d$ parametrisiert wird. Die Parameter werden in der entgegengesetzten Richtung des Gradienten der Zielfunktion $\nabla\theta J(\theta)$ angepasst. Die Lernrate η bestimmt die Größe der Schritte, die unternommen werden, um ein lokales Minimus zu erreichen. Mit anderen Worten, wird der Richtung bergab gefolgt, die durch die Zielfunktion erzeugt wird, bis ein „Tal“ erreicht wird (siehe Abbildung 2.5) [10].

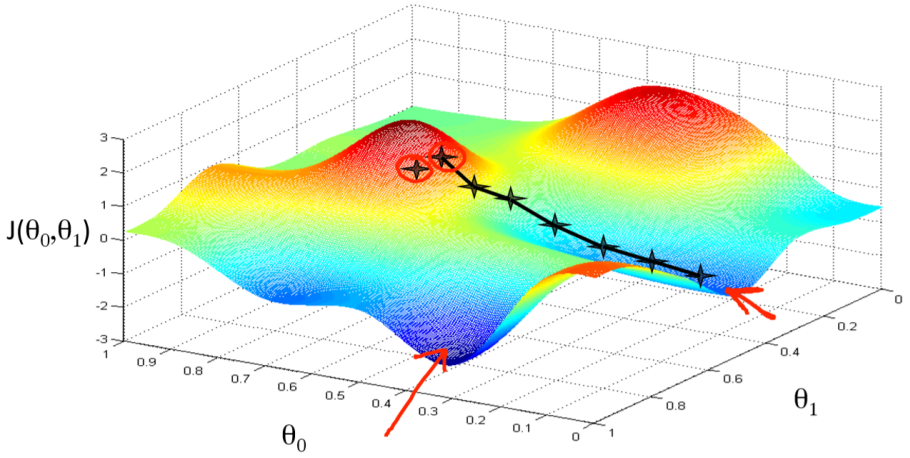

Abbildung 2.5: Darstellung des Gradientenabstiegs, entnommen aus [11].

2.7.2 Batch Gradientenabstiegsverfahren

Der Standart Gradientenabstiegsverfahren, auch Batch Gradientenabstiegsverfahren genannt, berechnet den Gradienten der Verlustfunktion zu den Parametern θ für den gesamten Trainingsdatensatz.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{2.9}$$

Da der gesamte Datensatz berechnet werden muss, um nur eine Aktualisierung durchzuführen, kann der Batch Gradientenabstieg sehr langsam sein und ist für Datensätze, die nicht in den Speicher passen, nicht zu handhaben. Der Batch Gradien-

tenabstieg ermöglicht es auch nicht, das Modell mit neuen Beispielen im laufenden Betrieb zu aktualisieren [10].

2.7.3 Stochastische Gradientenabstiegsverfahren

Im Gegensatz dazu führt der stochastische Gradientenabstiegsverfahren (SGD) eine Parameteraktualisierung für jedes Trainingsbeispiel durch. Der Batch Gradientenabstieg führt redundante Berechnungen für große Datenmengen durch, da Gradienten für ähnliche Beispiele vor jeder Parameteraktualisierung neu berechnet werden. Der Stochastische Gradientenabstiegsverfahren beseitigt diese Redundanz, indem jeweils ein Update durchgeführt wird. Es ist daher in der Regel viel schneller und kann auch beim laufendem Lernen verwendet werden [10].

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2.10}$$

2.8 Backpropagation

Das **Backpropagation** Verfahren bezieht sich auf die Gewichte eines mehrschichtigen Netzes und wendet dabei praktisch die Anwendung der Kettenregel der Differentialrechnung an. Es wird vom Gradienten in Bezug auf die Ausgabe (oder die Eingabe der Nachfolge) rückwärts gearbeitet. Die Kettenregel sagt dabei aus, wie zwei kleine Effekte (eine kleine Änderung von x auf y und der von y auf z) zusammengesetzt sind. Eine kleine Änderung von Δy in x wird zuerst in eine kleine Änderung Δy in y umgewandelt, indem sie mit $\frac{\partial y}{\partial y}$ multipliziert wird (die partielle Ableitung). In ähnlicher Weise erzeugt die Änderung Δy eine Änderung von Δz in z (siehe Formel 2.8) [4].

$$\begin{aligned}\Delta z &= \frac{\partial z}{\partial y}\Delta y \\ \Delta y &= \frac{\partial y}{\partial x}\Delta x \\ \Delta z &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\Delta x \\ \frac{\partial z}{\partial x} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\end{aligned} \tag{2.11}$$

Die Gleichungen, die zur Berechnung des **Vorwärtsdurchlaufs** (siehe Abbildung 2.6) in einem neuronalen Netz mit zwei verborgenen Schichten und einer Ausgangsschicht verwendet werden, bilden jeweils ein Modul, durch das man Gradienten zurückpropagieren kann. Auf jeder Ebene wird zuerst die Gesamteingabe z für jede Einheit berechnet. Dann wird eine nichtlineare Funktion auf z angewendet, um die Ausgabe der Einheiten zu erhalten (der Bias wurde hier der Einfachheit halber weggelassen). Eine nichtlineare Funktion kann z.B. eine ReLu- oder eine Sigmoid-Funktion oder eine tanh-Funktion sein. Im **Rückwärtsdurchlauf** (siehe Abbildung 2.6) wird in jeder verborgenen Schicht die Fehlerableitung in Bezug auf die Ausgabe jeder Einheit berechnet. Dies geschieht durch Vergleichen der Ausgaben mit der richtigen Antwort, um Fehlerableitungen zu erhalten [4].

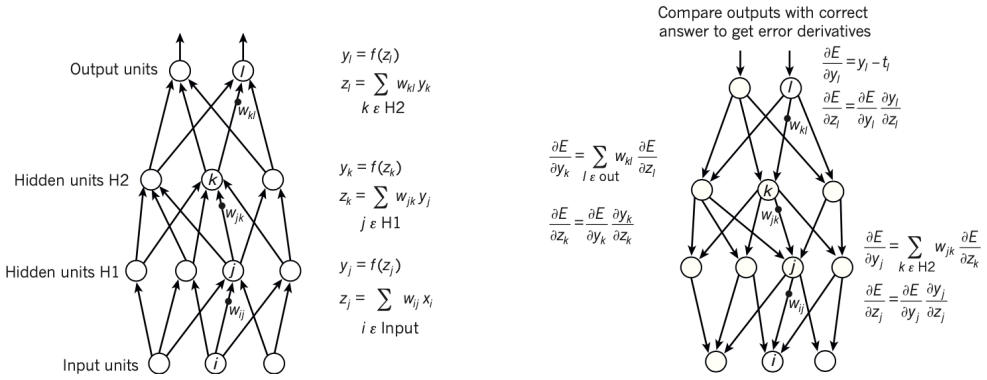

Abbildung 2.6: Darstellung vergleicht die Schritte des Backpropagation Verfahrens nach vorne und hinten (in Anlehnung an [4]).

Die Backpropagation-Gleichung wird wiederholt angewendet. Der Rückwärtsdurchlauf bezieht sich auf die Idee, die Differenz zwischen Vorhersage- und Istwerten zu verwenden, um die Hyperparameter der verwendeten Methode anzupassen. Für die Anwendung ist jedoch immer eine vorherige Vorwärtspropagation erforderlich.

2.9 Lernrate im Deep Learning

Die **Lernrate** beeinflusst den Betrag, um den die Parameter während der Optimierung angepasst werden, um den Fehler des neuronalen Netzwerks zu minimieren. Es ist ein Koeffizient, der die Größe der Schritte (Aktualisierungen) skaliert, die ein neuronales Netzwerk auf seinen Parameter (Vektor x) ausführt, wenn es den Verlustfunktionsraum durchquert. Ein großer Lernratenkoeffizient (z. B. 1) lässt die Parameter Sprünge machen, und kleine (z. B. 0,00001) lassen ihn langsam voranschreiten. Im Gegensatz dazu sollten kleine Lernraten letztendlich zu einem Fehlerminimum führen (es kann eher ein lokales als ein globales Minimum sein). Sehr kleine Lernraten können sehr lange dauern und die Belastung eines bereits rechenintensiven Prozesses erhöhen [12, S. 77].

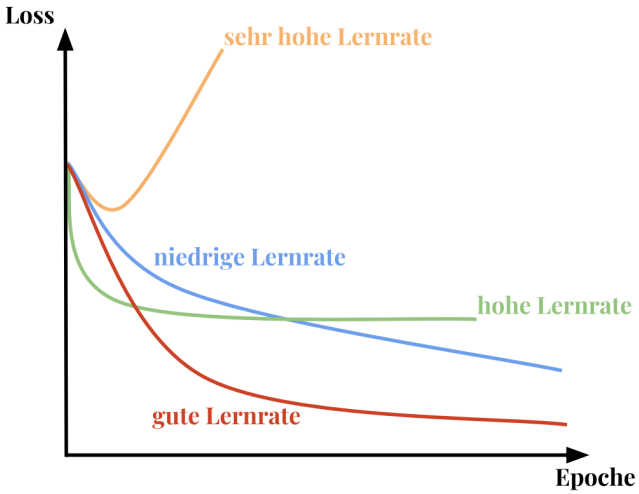

Abbildung 2.7: Vergleich unterschiedlicher Lernraten und deren Effekt auf den Verlust. Bei niedrigen Lernraten ist eine „lineare“ Verbesserungen zu sehen. Mit hohen Lernraten werden sie exponentieller. Höhere Lernraten verringern den Verlust schneller, bleiben jedoch bei schlechteren Verlustwerten hängen (grüne Linie). Dieses liegt daran, dass die Optimierung zu viel „Energie“ enthält und die Parameter „chaotisch herumspringen“ und sich nicht an einem Ort in der Optimierungslandschaft niederlassen können (in Anlehnung an [13]).

2.10 Unteranpassung und Überanpassung

Optimierungsalgorithmen versuchen zunächst, das Problem der **Unteranpassung** „Underfitting“ zu lösen. Das heißt, eine Linie zu nehmen, die sich den Daten nicht gut annähert, und sie besser an die Daten heranzuführen. Eine gerade Linie, die über ein gekrümmtes Streudiagramm schneidet, wäre ein gutes Beispiel für eine Unter-

anpassung, wie in Abbildung 2.8 dargestellt. Wenn die Linie zu gut zu den Daten passt, haben wir das gegenteilige Problem, welches als **Überanpassung** „Overfitting“ bezeichnet wird [12, S. 27].

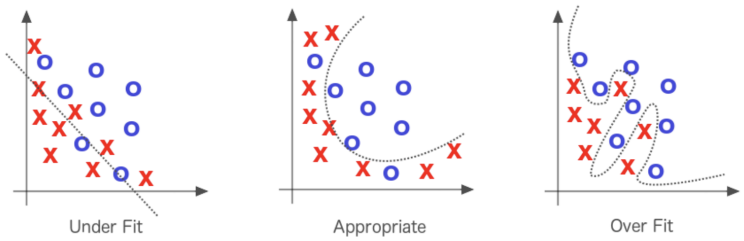

Abbildung 2.8: Die Abbildung vergleicht die Überanpassung mit der Unteranpassung. Die einzelnen Punkte passen sich dem trainierten Model zu sehr an, der Verlust wird also so klein, dass das Modell nicht mehr zuverlässige Ergebnisse liefern kann. Dies ist darauf zurückzuführen, dass das Modell „zu viel“ aus dem Trainingsdatensatz gelernt hat. Unteranpassung ist der Fall, wenn das Modell aus den Trainingsdaten „nicht genug gelernt“ hat, was zu einer geringen Verallgemeinerung und unzuverlässigen Vorhersagen führt (Grafik entnommen aus [12, S. 27]).

2.11 Regularisierung

Die Regularisierung lässt die Auswirkungen von außer Kontrolle geratenen Parametern minimieren, indem verschiedene Methoden oder Strategien verwendet werden, um die Parametergröße im Laufe der Zeit zu minimieren. Der Hauptzweck der Regularisierung besteht darin, die Überanpassung zu kontrollieren [12, S. 79].

Ein zentrales Problem beim maschinellen Lernen besteht darin, einen Algorithmus zu erstellen, der nicht nur bei den Trainingsdaten, sondern auch bei neuen Eingaben eine gute Leistung erbringt. Viele beim maschinellen Lernen verwendete Strategien sind explizit darauf ausgelegt, den Testfehler zu reduzieren, möglicherweise auf Kosten eines erhöhten Trainingsfehlers. Diese Strategien werden zusammen als Regularisierung bezeichnet. Tatsächlich war die Entwicklung effektiverer Regularisierungsstrategien eine der wichtigsten Forschungsanstrengungen auf diesem Gebiet. Regularisierung kann schließlich definiert werden als „jede Änderung, die wir an einem Lernalgorithmus vornehmen, um dessen Generalisierungsfehler, aber nicht seinen Trainingsfehler zu reduzieren“ [9, S. 228].

2.11.1 Early Stopping

Wenn große Modelle trainiert werden, um eine bestimmte Aufgabe lösen, wird häufig festgestellt, dass der Trainingsfehler mit der Zeit stetig abnimmt, der Fehler des Validierungssatzes jedoch wieder zunimmt. Dies bedeutet, dass ein Modell mit einem besseren Validierungssatzfehler (und damit einem besseren Testsatzfehler) erhalten werden kann, indem zu dem Zeitpunkt mit dem niedrigsten Validierungssatzfehler zur Parametereinstellung zurückgekehrt wird. Jedes Mal, wenn sich der Fehler im Validierungssatz verbessert, wird eine Kopie der Modellparameter gespeichert. Wenn der Trainingsalgorithmus beendet wird, wird diese Parameter anstelle der neuesten Parameter zurückgegeben. Diese Strategie wird als **Early Stopping** „frühes Stoppen“ bezeichnet. Es ist wahrscheinlich die am häufigsten verwendete Form der Regularisierung. Seine Popularität ist sowohl auf seine Wirksamkeit als auch auf seine Einfachheit zurückzuführen [9, S. 246].

2.11.2 Dropout

Eine weitere Strategie um Überanpassung zu vermeiden wird in [14] dargestellt. **Dropout** bietet eine rechnerisch kostengünstige, aber leistungsstarke Methode zur Regularisierung dar. Es ist das aufteilen des Netzwerkes in mehreren Teile. Es wird also ein Ensemble aus diesen kleinen Teilen (sub-networks) gebildet [9, S. 258].

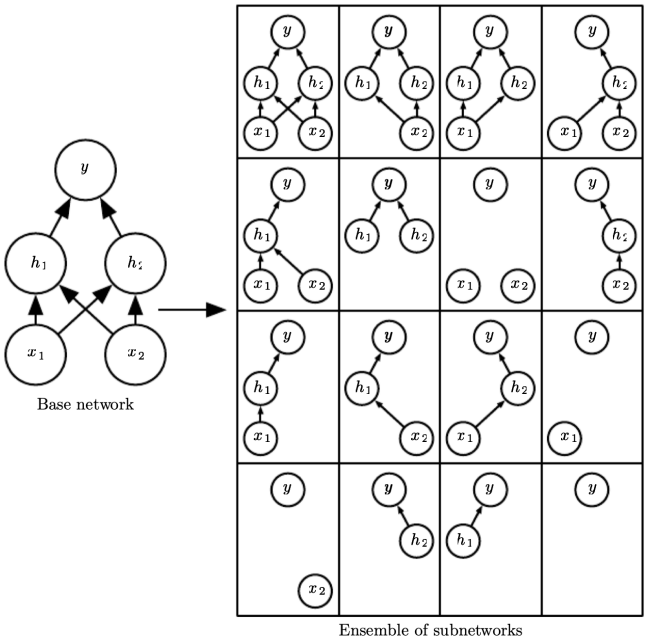

Abbildung 2.9: Dropout trainiert ein Ensemble. Ein Ensemble besteht aus allen Teilnetzwerken. Es wird durch das entfernen von Einheiten aufgebaut. Das Ensemble besteht aus 16 Teilmengen, aus den vier Einheiten des Basis-Netzwerks. Die 16 Subnetze werden durch das Löschen verschiedener Teilmengen von Einheiten aus dem ursprünglichen Netzwerk gebildet (entnommen aus [9, S. 260]).

2.11.3 L1 und L2

Eine weitere Strategie der Regularisierung ist die Modifikation der L1 und L2 Gewichte. Beim Deep Learning wird die Größe von Vektoren mit einer Funktion, die als „Norm“ bezeichnet wird [9, S. 39] gemessen. Formal ist diese Norm L^p gegeben als:

$$\|x\| = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}. \tag{2.12}$$

Normen, einschließlich der L^p-Norm, sind Funktionen, die Vektoren auf nicht negative Werte abbilden. Auf einer intuitiven Ebene misst die Norm eines Vektors x den Abstand vom Ursprung zum Punkt x. Die L^2-Norm mit $p = 2$ als euklidische Norm bekannt. Es ist der euklidische Abstand vom Ursprung zum Punkt x. Die L^2-Norm wird beim maschinellen Lernen häufig verwendet, sie wird einfach als $\|x\|$ bezeichnet, wobei der Index 2 weggelassen wird [9, S. 39].

Wenn zwischen Elementen zu unterscheiden ist, die genau Null sind und Elementen die klein, aber ungleich Null sind, wird die L^1-Norm angewendet. Die L^1-Norm kann vereinfacht werden als [9, S. 40]:

$$\|x\|1 \ = \ \sum_i \big|x_i\big|. \tag{2.13}$$

2.12 Verlustfunktion und Kreuzentropie

2.12.1 Verlustfunktionen

Innerhalb eines neuronalen Netzwerks wandelt eine **Verlustfunktion** oder **Kostenfunktion** alle möglichen Fehler, in eine Zahl um, die den Gesamtfehler des Netzwerks darstellt. Im Wesentlichen ist es ein Maß dafür, wie falsch ein Netzwerk liegt.

Das Neuron lernt dadurch, indem es Gewichte und Bias mit einer Rate ändert, die durch die partiellen Ableitungen der Kostenfunktion $\partial C/\partial w$ und $\partial C/\partial b$ bestimmt wird. Zu sagen, dass das „Lernen langsam ist“, ist also dasselbe wie zu sagen, dass diese partiellen Ableitungen klein sind [5, S. 61]. Gegeben sei die quadratische Verlustfunktion:

$$C = \frac{(y-a)^2}{2}, \tag{2.14}$$

dabei ist a die Ausgabe des Neurons, wenn die Trainingseingabe $x = 1$ ist, und $y = 0$ die entsprechende gewünschte Ausgabe. Um dies in Bezug auf Gewicht und Bias expliziter zu schreiben, sei daran erinnert, dass $a = \sigma(z)$ ist, wobei $z = wx+b$ ist. Es ergeben sich durch die Anwendung der Kettenregel folgende Gleichungen:

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z) \tag{2.15}$$

$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z). \tag{2.16}$$

Aus der Abbildung 2.2 ist die Kurve der Sigmoidfunktion zu sehen. Die Kurve wird sehr flach, wenn der Ausgang des Neurons nahe bei 1 liegt, und daher wird $\sigma'(z)$ sehr klein. Die Gleichungen 2.15 und 2.16 sagen dann aus, dass $\partial C/\partial w$ und $\partial C/\partial b$ sehr klein werden. Dies ist der Grund warum das lernen langsamer wird.

2.12.2 Kreuzentropie

Nach [5, S. 62] kann die Lernverlangsamung gelöst werden, indem die quadratische Verlustfunktion durch eine andere Verlustfunktion ersetzt wird. Diese Funktion wird als als **Kreuzentropie** bezeichnet. Die Abbildung 2.10 zeigt die Kreuzentropie mit mehreren Eingabevariablen und entsprechenden Gewichten und dem Bias. Die Ausgabe des Neurons ist $a = \sigma(z)$, wobei $z = \sum_j w_j b_j + b$ ist, die gewichtete Summe des Inputs. Die Kreuzentropiekostenfunktion für dieses Neuron wird definiert durch:

$$C = -\frac{1}{n} \sum_x [y \, ln \, a \, + \, (1 - y) \, ln(1 - a)] \tag{2.17}$$

wobei n die Gesamtzahl der Trainingselemente darstellt. Die Summe gibt die entsprechende gewünschte Ausgabe x und y über alle Trainingseingaben an. Zusammenfassend ist die Kreuzentropie positiv und tendiert gegen Null, wenn das Neuron „besser“ wird in der Berechnung der gewünschten Ausgabe y für alle Trainingseingaben x ist. Die entropieübergreifende Kostenfunktion hat jedoch den Vorteil, dass sie im Gegensatz zu den quadratischen Kosten das Problem der Verlangsamung des Lernens vermeidet. Um dies zu sehen, wird die partielle Ableitung der Kreuzentropiekosten in Bezug auf die Gewichte berechnet [5, S. 63].

$$\begin{aligned} \frac{\partial C}{\partial w_j} &= -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{1 - y}{1 - \sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} = \\ &-\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{1 - y}{1 - \sigma(z)} \right) \sigma'(z) x_j \end{aligned} \tag{2.18}$$

$$\frac{\partial C}{\partial w_j} \; = \; \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1 - \sigma(z))} (\sigma(z) - y) \tag{2.19}$$

$$\frac{\partial C}{\partial w_j} \; = \; \frac{1}{n} \sum_x x_j (\sigma(z) - y) \tag{2.20}$$

Die Geschwindigkeit, mit der das Gewicht w lernt, wird durch $\sigma(z) - y$ gesteuert, also durch den Fehler in der Ausgabe. Je größer dieser Fehler wird, desto schneller lernt das Neuron. Dies ist ein gewünschtes Verhalten. Insbesondere wird die Lernverlangsamung vermieden, die durch den Term $\sigma'(z)$ in der analogen Gleichung

für die quadratischen Kostenfunktion 2.15 verursacht wird. Wenn die Kreuzentropie verwendet wird, wird der Term $\sigma'(z)$ aufgehoben und somit ist es egal, ob es klein ist. Diese Aufhebung ist das Besondere, welches durch die Kreuzentropie-Kostenfunktion gewährleistet wird [5, S. 63–64].

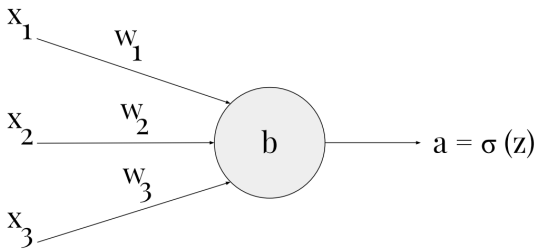

Abbildung 2.10: Das Neuron wird mit 3 Eingabewerten (x_1, x_2, x_3) den dazugehörigen Gewichten (W_1, W_2, W_3) trainiert. Der Bias ist durch b angegeben und die Ausgabe mit $a = \sigma(z)$ (in Anlehnung an [5])

2.13 Convolutional Neural Network

ConvNets/CNNs oder **Convolutional Neural Networks** dienen zur Verarbeitung von Daten in Form mehrerer Arrays, beispielsweise eines Farbbilds aus drei 2D-Arrays mit Pixelintensitäten in den drei Farbkanälen. Viele Datenmodalitäten liegen in Form mehrerer Arrays vor: 1D für Signale und Sequenzen, einschließlich Sprache; 2D für Bilder oder Audiospektrogramme; und 3D für Video- oder Volumenbilder. Hinter ConvNets stehen vier Schlüsselideen, die die Eigenschaften natürlicher Signale nutzen: lokale Verbindungen, gemeinsame Gewichte, Pooling und die Verwendung vieler Schichten [4].

CNNs arbeiten mit gitterstrukturierten Eingaben, die in lokalen Regionen des Netzes starke räumliche Abhängigkeiten aufweisen. Das offensichtlichste Beispiel für gitterstrukturierte Daten ist ein zweidimensionales Bild. Diese Art von Daten weisen räumliche Abhängigkeiten auf, da benachbarte räumliche Orte in einem Bild häufig ähnliche Farbwerte der einzelnen Pixel aufweisen. Eine zusätzliche Dimension erfasst die verschiedenen Farben, wodurch ein dreidimensionales Eingabevolumen entsteht. Andere Formen von sequentiellen Daten wie Text, Zeitreihen und Sequenzen können ebenfalls als Sonderfälle von Daten mit Gitterstruktur mit verschiedenen Arten von Beziehungen zwischen benachbarten Elementen betrachtet werden. Die überwiegende Mehrheit der Anwendungen von CNNs konzentriert sich

auf Bilddaten, obwohl man diese Netze auch für alle Arten von zeitlichen, räumlichen und raumzeitlichen Daten verwenden kann. Ein wichtiges definierendes Merkmal von CNNs ist eine Operation, die als Faltung (convolution) bezeichnet wird. [15, S. 315–316].

2.13.1 Architektur

In CNNs sind mehrere Schichten miteinander verbunden und jeder Schicht hat eine Gitterstruktur. Die Bezihungen zwischen den Schichten werden von einer Schicht zur nächsten vererbt, da jeder Merkmalswert auf einem kleinen lokalen Bereich aus der vorherigen Schicht basiert. Es ist wichtig, diese räumlichen Beziehungen zwischen den Gitterzellen aufrechtzuerhalten, da die Faltungsoperation und die Transformation zur nächsten Schicht von diesen Beziehungen abhängen. Jede Schicht im Faltungsnetzwerk ist eine dreidimensionale Gitterstruktur mit einer Höhe, Breite und Tiefe. Die Tiefe einer Schicht in einem Faltungsnetzwerk ist nicht die Tiefe des Netzwerks selbst. Das Wort „Tiefe“ bezieht sich auf die Anzahl der Kanäle in jeder Ebene, z. B. die Anzahl der Primärfarbkanäle (z. B. Blau, Grün und Rot) im Eingabebild [15, S. 318].

Die Architektur eines typischen ConvNets ist in mehrere Phasen unterteilt. Die ersten Stufen bestehen aus zwei Arten von Schichten: Faltungsschichten und Poolschichten. Einheiten in einer Faltungsebene sind in **Feature-Maps** organisiert, in denen jede Einheit über eine Reihe von Gewichten, die als Filterbank bezeichnet werden, mit lokalen Patches in den Feature-Maps der vorherigen Ebene verbunden ist. Das Ergebnis dieser lokal gewichteten Summe wird dann durch eine Nichtlinearität wie z. B.eine ReLU geleitet. Alle Einheiten in einer Feature-Map verwenden dieselbe Filterbank[4].

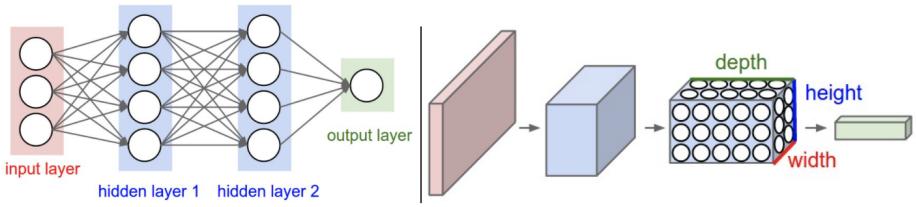

Abbildung 2.11: Links: Ein reguläres 3-Schicht-Neuronales Netz. Rechts: Ein ConvNet ordnet seine Neuronen in drei Dimensionen (Breite, Höhe, Tiefe) an, wie in einer der Ebenen dargestellt. Jede Schicht eines ConvNet wandelt das 3D-Eingangsvolumen in ein 3D-Ausgangsvolumen von Neuronenaktivierungen um. In diesem Beispiel enthält die rote Eingabeebene das Bild, sodass seine Breite und Höhe den Abmessungen des Bildes entsprechen. Die „Tiefe“ sind die 3 Farbkanäle (rot, grün, blau) aus [16].

2.13.2 Convolutional Layer

Die Parameter der **Convolutional Layer** oder **Faltungsschicht** bestehen aus einer Reihe von lernbaren Filtern. Jedes dieser Filter ist räumlich klein (Breite und Höhe), erstreckt sich jedoch über die gesamte Tiefe des Eingangsvolumens. Beispielsweise könnte ein typischer Filter auf einer ersten Schicht eines ConvNet die Größe $5 \times 5 \times 3$ haben (d. H. 5 Pixel Breite und Höhe und 3, weil Bilder die Tiefe 3 haben, also die Farbkanäle). Während des Vorwärtsdurchlaufs wird jedes der Filter über die Breite und Höhe des Eingangsvolumens *gefaltet* und es werden die Punktprodukte zwischen den Einträgen des Filters und dem Eingang an einer beliebigen Position berechnet. Wenn der Filter über die Breite und Höhe des Eingangsvolumens gefaltet wird, wird eine zweidimensionale *Aktivierungskarte* (feature map) erstellt, die die Antworten dieses Filters an jeder räumlichen Position angibt. Intuitiv lernt das Netzwerk Filter, die aktiviert werden, wenn sie eine Art visuelles Merkmal sehen, z. B. eine Kante mit einer bestimmten Ausrichtung oder einen Farbfleck auf der ersten Ebene oder schließlich radähnliche Muster auf höheren Ebenen des Netzwerks. Es entstehen somit ein ganzer Satz von Filtern in jeder Faltungsschicht (z. B. 12 Filter), und jeder von ihnen erzeugt eine separate zweidimensionale Aktivierungskarte. Diese Karten werden entlang der Tiefendimension gestapelt und das Ausgabevolumen erzeugt [16].

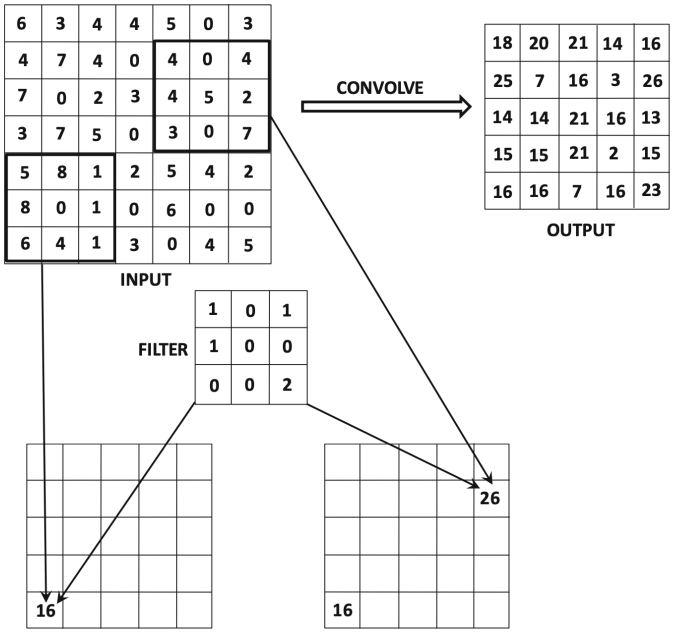

Abbildung 2.12: Die Faltungsoperation geschieht durch eine Punktprodukt Operation des Filters, was über alle räumlichen Positionen wiederholt wird aus [15, S. 321].

Padding

Die Faltungsoperation verringert die Größe der $(q+1)$-ten Schicht im Vergleich zur Größe der q-ten Schicht. Diese Art der Größenreduzierung ist im Allgemeinen nicht wünschenswert, da sie dazu neigt, einige Informationen entlang der Bildränder zu verlieren. Dieses Problem kann durch **Auffüllen** (padding) gelöst werden. Beim Auffüllen werden neue Werte rund um die Ränder der Feature-Map hinzugefügt. Der Wert jedes dieser aufgefüllten Feature-Werte wird auf 0 gesetzt, unabhängig davon, ob die Eingabe oder die ausgeblendeten Ebenen aufgefüllt werden. Diese Bereiche tragen nicht zum endgültigen Punktprodukt bei, da ihre Werte auf 0 gesetzt sind. Ein Teil des Filters aus den Rändern der Schicht wird „herausragt“ und dann durch das Durchführen des Punktprodukts nur über den Teil der Ebene, in dem die Werte definiert sind ersetzt [15, S. 323].

2.13.3 Pooling Layer

Es ist üblich, regelmäßig eine **Pooling Ebene** zwischen aufeinanderfolgenden Faltungsebenen in eine ConvNet-Architektur einzufügen. Die Funktion dieser Ebenene besteht darin, die räumliche Größe der Darstellung schrittweise zu verringern, um die Anzahl der Parameter und die Berechnung im Netzwerk zu verringern und damit auch die Überanpassung zu steuern. Die Pooling-Ebene arbeitet unabhängig mit jedem Tiefenabschnitt der Eingabe und ändert die Größe räumlich mithilfe der MAX-Operation. Die gebräuchlichste Form ist eine Pooling-Ebene mit Filtern der Größe *2x2*, die mit einem **Schritt** (Stride) von 2 Downsamples pro Tiefenscheibe in der Eingabe um 2 entlang der Breite und Höhe angewendet werden, wobei 75% der Aktivierungen verworfen werden. Jede MAX-Operation würde in diesem Fall maximal 4 Zahlen annehmen. Der *Volumen* bleibt unverändert [16].

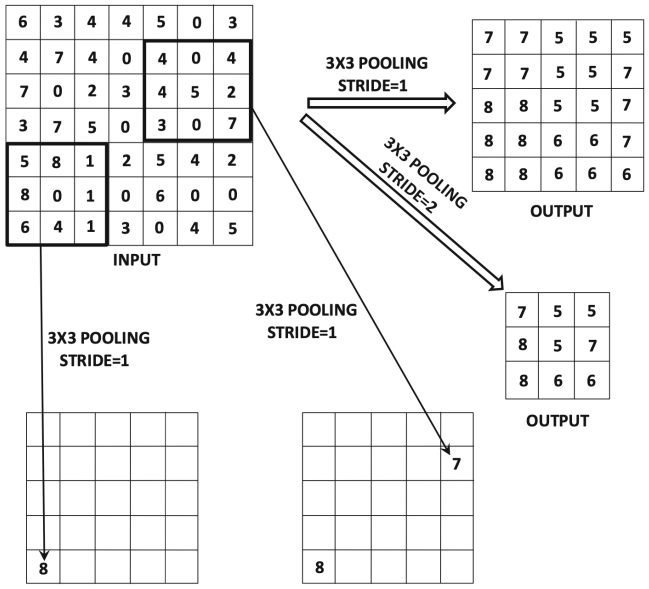

Abbildung 2.13: Ein Beispiel für ein Max-Pooling einer Aktivierungskarte der Größe 7 × 7 mit Stride von 1 und 2. Ein Stride von 1 erzeugt eine 5 × 5-Aktivierungskarte mit stark wiederholten Elementen aufgrund der Maximierung in überlappenden Regionen. Ein Stride von 2 erzeugt eine 3 × 3-Aktivierungskarte mit weniger Überlappung [15, S. 326].

2.13.4 Vollständig verbundene Ebenen

Am Ende wird die sogenannte **vollständig verbundene Ebenen** des Netzwerkes verwendet, um Klassenwerte zu berechnen, die als Ausgabe des Netzwerks dienen sollen. In einem traditionellen vorwärtsgerichteten Neuronalen Netzwerk wird jedes Eingangsneuron mit jedem Ausgangsneuron in der nächsten Schicht verbunden. Dies wird auch als vollständig verbundene Schicht bezeichnet. Der einzige Unterschied zwischen der vollständig verbundene Ebene und Faltungsschichten besteht darin, dass die Neuronen in der Faltungsschicht nur mit einer lokalen Region der Eingabe verbunden ist. Die Neuronen in beiden Schichten berechnen jedoch immer noch Punktprodukte, sodass ihre funktionale Form identisch ist. Zwischen beiden Schichten ist eine Konvertierung möglich [16].

Kapitel 3

Die Natürliche Sprache

Die **Verarbeitung natürlicher Sprache** (NLP) ist ein theoretisch motivierter Bereich von Computertechniken, zum Analysieren und Darstellen natürlich vorkommender Texte auf einer oder mehreren Ebenen der Sprachanalyse, um eine menschenähnliche Sprachverarbeitung für eine Reihe von Aufgaben oder Anwendungen zu erreichen [17].

Der Umgang mit Textdaten ist problematisch, da unsere Computer, Skripte und Modelle für maschinelles Lernen, keinen Text im menschlichen Sinne lesen und verstehen können. Wörter können viele verschiedene Assoziationen aufrufen, diese sprachlichen Assoziationen sind das Ergebnis recht komplexer neurologischer Berechnungen. ML-Modelle sind haben dieses vorgefertigte Verständnis der Wortbedeutung nicht.

3.1 Wie können Deep Learning Modelle die Natürliche Sprache lernen?

Die verborgenen Schichten eines mehrschichtigen neuronalen Netzwerks lernen, die Eingaben des Netzwerks so darzustellen, dass die Zielausgaben leicht vorhergesagt werden können. Dies wird gut demonstriert, indem ein mehrschichtiges neuronales Netzwerk trainiert wird, um das nächste Wort in einer Sequenz aus einem lokalen Kontext früherer Wörter vorherzusagen [18]. Jedes Wort im Kontext wird dem Netzwerk als Eins-aus-N-Vektor dargestellt, dh eine Komponente hat den Wert 1 und der Rest ist 0, dieses Verfahren wird auch als *One-Hot-Encoding* bezeichnet. In einem *Sprachmodell* lernen die anderen Schichten des Netzwerks, die Eingangs-

wortvektoren in einen Ausgangswortvektor für das vorhergesagte nächste Wort umzuwandeln, welches verwendet werden kann, um die Wahrscheinlichkeit vorherzusagen, dass ein Wort im Vokabular als nächstes erscheinen wird [4]. Das Netzwerk lernt **Wortvektoren**, die viele aktive Komponenten enthalten, von denen jede als separates Merkmal des Wortes interpretiert werden kann. Dieses Vorgehen wurde erstmals im Zusammenhang mit dem Lernen verteilter Darstellungen für Symbole demonstriert [19]. Die semantischen Merkmale waren in der Eingabe nicht explizit vorhanden. Das Lernverfahren wurde als eine gute Möglichkeit entdeckt, die strukturierten Beziehungen zwischen den Eingabe- und Ausgabesymbolen in mehrere „Mikroregeln“ zu zerlegen. Das Lernen von Wortvektoren hat sich auch als sehr gut erwiesen, wenn die Wortsequenzen aus einem großen Korpus von echtem Text stammen und die einzelnen Mikroregeln unzuverlässig sind [18].

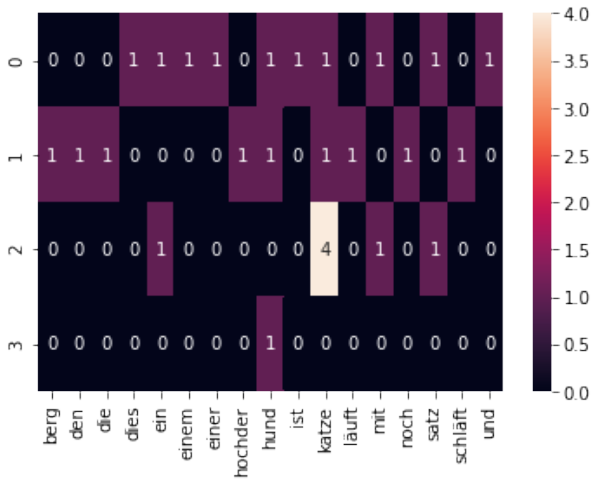

Abbildung 3.1: Die Grafik zeigt wie ein Beispielkorpus welches aus 5 Sätzen besteht, in einer Matrix dargestellt werden kann. Je nach Häufigkeit wird jedes Wort im Wortschaft entsprechend den Sätzen im Korpus abgebildet. Der Korpus besteht aus 5 Sätzen („Dies ist ein Satz mit einer Katze und einem Hund.“, „Die Katze läuft den Berg hoch.“, „Der Hund schläft noch.“, „Ein Satz mit Katze Katze Katze Katze.“ und „Ein Hund.“) (eigene Darstellung).

Die numerischen Werte sollten so viel wie möglich von der sprachlichen Bedeutung eines Wortes erfassen. Eine gut ausgewählte, informative Eingabedarstellung kann einen massiven Einfluss auf die Gesamtleistung des Modells haben. **Worteinbettungen** sind der vorherrschende Ansatz für dieses Problem und so weit verbreitet, dass ihre Verwendung praktisch in jedem NLP-Projekt angenommen wird. Unabhängig davon, ob Sie ein Projekt in den Bereichen Textklassifizierung, Sentimentanalyse oder maschinelle Übersetzung starten [4].

3.2 Worteinbettungen

Vektoren zur Darstellung von Wörtern werden im Allgemeinen als Einbettungen bezeichnet, da das Wort in einen bestimmten Vektorraum eingebettet wird [20, S. 99].

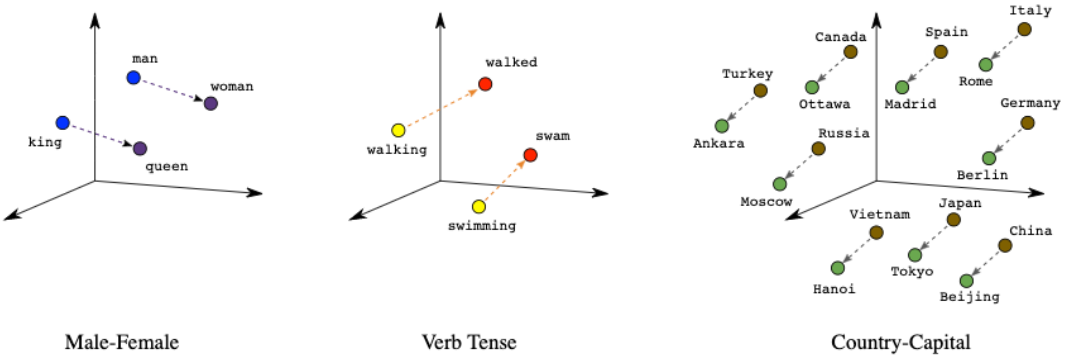

Abbildung 3.2: Durch Worteinbettungen können interessante Analogien zwischen einzelnen Wörtern gefunden werden. (Entnommen aus [21])

Word2Vec [22] Worteinbettungen sind Vektordarstellungen von Wörtern, die normalerweise von einem Modell gelernt werden, wenn große Textmengen als Eingabe eingegeben werden (z. B. Wikipedia, Wissenschaft, Nachrichten, Artikel usw.). Diese Darstellung von Wörtern erfasst die semantische Ähnlichkeit zwischen Wörtern unter anderen Eigenschaften. Word2Vec-Worteinbettungen werden so gelernt, dass der Abstand zwischen Vektoren für Wörter mit enger Bedeutung (z. B. „König“ und „Königin“) näher ist als der Abstand für Wörter mit völlig unterschiedlichen Bedeutungen (z. B. „König“ und „Katze“).

Bei der *One-Hot-Codierung* sind die Wörter „gut“ und „großartig“ genauso unterschiedlich wie „Tag“ und „Nacht“. Hier kommt die Idee, verteilte Darstellungen zu erzeugen. Intuitiv wird eine gewisse Abhängigkeit eines Wortes von den anderen Wörtern eingeführt. Die Wörter im Kontext dieses Wortes würden einen größeren Anteil dieser Abhängigkeit erhalten. In der One-Hot Darstellung dagegen, sind alle Wörter unabhängig voneinander.

3.2.1 Word2Vec mit der Skip-Gram Architektur

Wörter können als spärliche, lange Vektoren mit vielen Dimensionen dargestellt werden. Eine alternative Methode ist die Darstellung eines Wortes mit der Verwendung von *kurzen Vektoren*, mit einer Länge von 50-1000 und einer *großen dichte*

(die meisten Werte sind nicht Null). Es stellt sich heraus, dass dichte Vektoren in jeder NLP-Aufgabe besser funktionieren als spärliche Vektoren. Erstens können dichte Vektoren erfolgreicher als Features in maschinellen Lernsystemen aufgenommen werden.

Wenn beispielsweise 100-dimensionale Worteinbettungen als Merkmale verwendet werden, kann ein Klassifikator nur 100 Gewichte lernen, um die Bedeutung des Wortes darzustellen. Wenn stattdessen einen 50.000-dimensionalen Vektor eingeben wird, müsste ein Klassifikator Zehntausende von Gewichten für jede der spärlichen Dimensionen lernen. Zweitens können dichte Vektoren besser verallgemeinern und helfen, eine Überanpassung zu vermeiden, da sie weniger Parameter als spärliche Vektoren mit expliziten Zählungen enthalten. Schließlich können dichte Vektoren die Synonyme besser erfassen als spärliche Vektoren. Zum Beispiel sind *Auto* und *Automobil* Synonyme. In einer typischen spärlichen Vektordarstellung sind beide Dimensionen unterschiedliche Dimensionen. Da die Beziehung zwischen diesen beiden Dimensionen nicht modelliert wird, können spärliche Vektoren möglicherweise die Ähnlichkeit zwischen Auto und Automobil als Nachbarn nicht erfassen [20, S. 110–111].

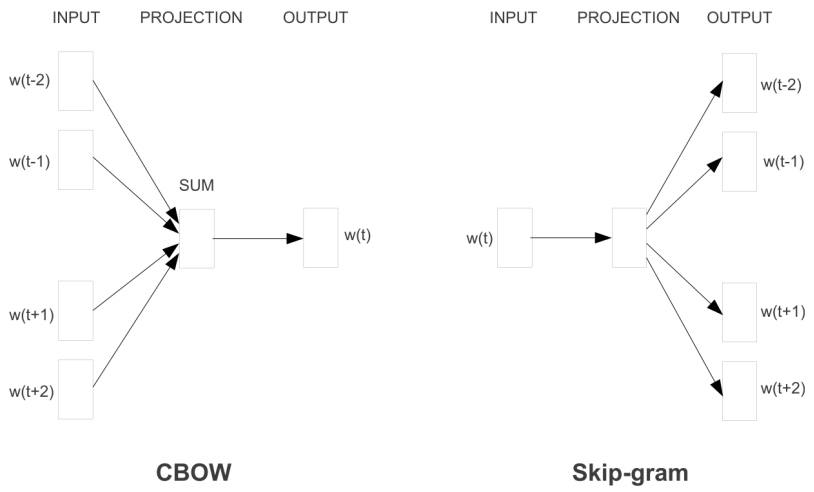

Abbildung 3.3: Die CBOW-Architektur sagt das aktuelle Wort basierend auf dem Kontext voraus, während das Skip-Gramm die umgebenden Wörter voraussagt, wenn das aktuelle Wort gegeben ist aus [23]).

Der Skip-Gram-Algorithmus ist einer von zwei Algorithmen in einem Softwarepaket namens **word2vec** [22][23]. Die Intuition von word2vec ist, dass anstatt zu zählen, wie of jedes Wort w in der Nähe von einem anderen Wort vorkommt, einen

Klassifikator für eine binäre Vorhersageaufgabe zu trainieren. Die erlernten Klassifikatorgewichte werden dann als Worteinbettungen genommen [20, S. 111]. Dabei wird ein Zielwort t mit Kandidaten aus dem Kontext *c* in ein *Tupel* gesetzt. *P(+|t,c)* sagt dann aus, wie wahrscheinlich es ist, dass ein Kontextwort *c* ein echter Kontext ist. Zum Beispiel sei gegeben der Satz „Wir essen Spaghetti zum Abendessen...“. Wenn der Kontext von ± 2 Wörter betrachtet wird, und *t* das Zielwort „essen“ ist, wird die Klassifikation für das Tupel *(essen,spaghetti)* „true“ und für das Tupel *(essen,auto)* „false“ zurückgeben [20, S. 111].

$$P(-|t,c) = 1 - P(+|t,c) \tag{3.1}$$

Die Ähnlichkeit eines Wortes zu einem anderen Wort, kann mittels Skalarprodukt berechnet werden. Dies ist zunächst nur eine Zahl zwischen $-\infty$ und $+\infty$. Um daraus eine Wahrscheinlichkeit zu berechnen, wird die *sigmoid* Funktion $\sigma(x)$ angewendet. Die Logistikfunktion gibt eine Zahl zwischen 0 und 1 zurück. Um die Wahrscheinlichkeit zu berechnen, muss gewährleistet werden, dass die Summe *c ist das Kontextwort* und *c ist nicht das Kontextwort* eine 1 ergeben [20, S. 112].

$$P(-|t,c) = 1 - P(+|t,c) = \frac{e^{-t \cdot c}}{1 + e^{-t \cdot c}} \tag{3.2}$$

Skip-Gramm macht die starke, aber sehr nützliche vereinfachende Annahme, dass alle Kontextwörter unabhängig sind, so dass ihre Wahrscheinlichkeiten multipliziert werden können:

$$P(+|t, c_{1:k}) = \prod_{i=1}^{k} \frac{1}{1 + e^{-t \cdot c_i}} \tag{3.3}$$

$$\log P(+|t, c_{1:k}) = \sum_{i=1}^{k} \log \frac{1}{1 + e^{-t \cdot c_i}}. \tag{3.4}$$

Word2vec lernt Einbettungen, indem es mit einem anfänglichen Satz von Einbettungsvektoren beginnt und dann die Einbettung jedes Wortes w iterativ verschiebt, um mehr in der Nähre der Einbettungen von Wörtern zu kömmen die ähneln. Für das Training eines binären Klassifikators werden negative Beispiele nötig. Das Skip-Gram benötigt mehr negative als positive Beispiele für das Training. Das Verhältnis zwischen positiven und negativen Beispielen wird mit einem Parameter k festge-

legt. Für jedes der Trainingseinheiten *t, c* werden *k* negative Stichproben erstellt, die jeweils aus dem Ziel *t* und einem „noise word“ besteht. Ein „noise word“ is ein zufälliges Wort aus dem Lexikon, das nicht das Zielwort „t“ sein darf [20, S. 113].

Das Ziel des Lernalgorithmus besteht darin, mittels gegebenen positiven und negativen Beispielen diese Einbettungen so anzupassen, dass die Ähnlichkeit der Ziel- und Kontextwortpaare *(t, c)* aus den positiven Beispielen maximiert werden und die Ähnlichkeit der Paare *(t, c)* aus den negativen Beispielen minimiert wird. Formell lässt sich dieses mit folgender Formel ausdrücken:

$$
\begin{aligned}
L(\theta) = \sum_{(t,c)\in+} \log P(+|t,c) + \sum_{(t,c)\in-} \log P(-|t,c) = \\
\log \sigma(c \cdot t) + \sum_{i=1}^{k} \log \sigma(-n_i \cdot t) = \\
\log \frac{1}{1+e^{-c \cdot t}} + \sum_{i=1}^{k} \log \frac{1}{1+e^{n_1 \cdot t}}.
\end{aligned} \tag{3.5}
$$

Die stochastische Gradientenabstieg kann verwendet werden, um dieses Ziel zu erreichen, indem die Parameter (die Einbettungen für jedes Zielwort t und jedes Kontextwort oder „noise word“ c im Vokabular) iterativ modifiziert werden [20, S. 114].

3.2.2 CNN in der Textverarbeitung

Anstelle von Bildpixeln können die Eingaben für die meisten NLP-Aufgaben Sätze oder Dokumente, als eine Matrix dargestellt werden. Jede Zeile der Matrix entspricht einem Token, normalerweise einem Wort, aber es kann sich auch um ein Zeichen handeln. Das heißt, jede Zeile ist ein Vektor, der ein Wort darstellt. Typischerweise sind diese Vektoren Worteinbettungen (niedrigdimensionale Darstellungen) wie word2vec oder GloVe, aber sie können auch One-Hot-Vektoren sein, die das Wort in ein Vokabular indizieren. Für einen 10-Wort-Satz unter Verwendung einer 100-dimensionalen Einbettung hätten wir eine 10×100-Matrix als Eingabe [24].

In der Vision „gleiten“ die Filter über lokale „pathces“ eines Bildes, in NLP wird jedoch der Filter über die ganze Zeilen der Matrix (Wörter) gleiten. Daher ent-

spricht die Breite der Filter normalerweise der Breite der Eingabematrix. Die Höhe oder Regionsgröße kann variieren, aber „Schiebefenster“ mit jeweils mehr als 2-5 Wörtern sind typisch. Pixel, die nahe beieinander liegen, sind wahrscheinlich semantisch verwandt (Teil desselben Objekts), aber das Gleiche gilt nicht immer für Wörter. In vielen Sprachen können Teile von Phrasen durch mehrere andere Wörter getrennt werden. Der kompositorische Aspekt ist ebenfalls nicht offensichtlich. Es ist klar, dass Wörter in gewisser Weise zusammengesetzt sind, wie ein Adjektiv, das ein Substantiv modifiziert, aber wie genau dies funktioniert, was Darstellungen auf höherer Ebene tatsächlich „bedeuten“, ist nicht so offensichtlich wie im Fall von Computer Vision. Angesichts all dessen scheinen CNNs nicht gut für NLP-Aufgaben geeignet zu sein. Wiederkehrende neuronale Netze (RNNs) sind intuitiver. Sie ähneln der Art und Weise, wie wir Sprache verarbeiten (oder zumindest wie wir denken, dass wir Sprache verarbeiten). Lesen nacheinander von links nach rechts. Glücklicherweise bedeutet dies nicht, dass CNNs nicht funktionieren. Alle Modelle sind falsch, aber einige sind nützlich. Es stellt sich heraus, dass CNNs, die auf NLP-Probleme angewendet werden, recht gut funktionieren. Ein großes Argument für CNNs ist, dass sie schnell sind. Faltungen sind ein zentraler Bestandteil der Computergrafik und werden auf Hardwareebene auf GPUs implementiert. Mit einem großen Wortschatz kann das Berechnen schnell „teuer“ werden. Faltungsfilter lernen automatisch gute Darstellungen, ohne das gesamte Vokabular darstellen zu müssen [25].

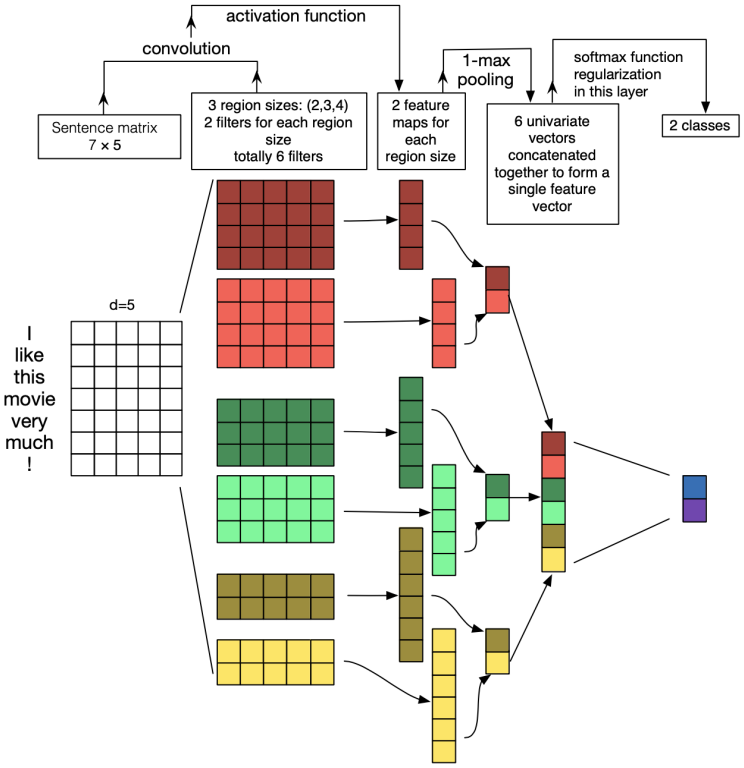

Abbildung 3.4: Illustration einer CNN-Architektur zur Satzklassifizierung. Es sind drei Filterbereichsgrößen vorhanden: 2, 3 und 4. Filter führen Faltungen in der Satzmatrix durch und generieren Feature-Maps (mit variabler Länge). Über jede Karte wird ein 1-Max-Pooling durchgeführt, d. H. die größte Anzahl von jeder Merkmalskarte wird aufgezeichnet. Somit wird aus allen sechs Karten ein univariater Merkmalsvektor erzeugt, und diese sechs Merkmale werden verkettet, um einen Merkmalsvektor für die vorletzte Schicht zu bilden. Die letzte Softmax-Schicht empfängt dann diesen Merkmalsvektor als Eingabe und verwendet ihn zur Klassifizierung des Satzes an. Hier wird eineeine binäre Klassifikation angewendet und es gibt daher zwei mögliche Ausgangszustände dar [24].

Kapitel 4

Verwandte Arbeiten

4.1 Einleitung

Clickbaits besitzen semantische und syntaktische Nuancen, auf die besonders vorgegangen werden muss. Dies geschieht in Form von analyse und vorverarbeitung der Titel. In diesem Abschnitt werden Ansätze aus der Literatur verglichen, die diese semantischen und syntaktischen Besonderheiten bearbeiten. In diesem Kapitel werden Ansätzte aus der Literatur vorgestellt, die die Aufgabe Clickbait Klassifizierung und die damit entstehenden Unteraufgaben wie das Sammeln von Daten, Einbetten der Wörter und schließlich entwickeln der Deep Learning Modelle haben.

4.2 Analyse der Literatur

Mit der Arbeit aus [26] wurde eine Browser-Erweiterung erstellt, welches Clickbaits erkennen soll. Es wurden umfangreiche Daten sowohl für Clickbaits als auch für Nicht-Clickbait-Kategorien gesammelt. Für die Nicht-Clickbaits-Kategorien wurden 18.513 Wikinews Artikeln gesammelt. Der Vorteil dieser Artikel ist, dass diese von einer Community erstellt werden und jeder Nachrichtenartikel vor der Veröffentlichung von der Community geprüft werden. Es gibt Stilrichtlinien, die eingehalten werden müssen. Um Clickbaits zu finden, haben die Autoren manuell aus Seiten wie „Buzzfeed“ oder „Upworthy“ ca. 8000 Titel gecrawlt. Um falsche Negative zu vermeiden (d. h. die Artikel in diesen Bereichen, bei denen es sich nicht um Clickbaits handelt), wurden sechs Freiwillige rekrutiert um die Überschriften zu labeln. Schließlich wurden 7500 Titel zu jeder der beiden Kategorien zugefügt.

Laut [26] sind die herkömmlichen Nicht-Clickbait Schlagzeilen kürzer als Clickbait Schlagzeilen. Traditionelle Schlagzeilen enthalten in der Regel meistens Wörter, die sich auf bestimmte Personen und Orte beziehen, während die Funktionswörter den Lesern zur Interpretation aus dem Kontext überlassen bleiben. Es wird hier als Beispiel gegeben „Visa-Deal oder kein Migranten-Deal, Türkei warnt EU“. Hier sind die meisten Wörter Inhaltswörter, die die wichtigsten Erkenntnisse aus der Geschichte zusammenfassen, und es gibt nur sehr wenige Verbindungsfunktionswörter zwischen den Inhaltswörtern. Auf der anderen Seite sind Clickbait Schlagzeilen länger. Die Sätze, enthalten sowohl Inhalts- als auch Funktionswörter. Ein Beispiel für solche Schlagzeilen ist „Ein 22-Jähriger, dessen Ehemann und Baby von einem betrunkenen Fahrer getötet wurden, hat ein Facebook-Plädoyer veröffentlicht“. Obwohl die Anzahl der Wörter in Clickbait-Schlagzeilen höher ist, ist die durchschnittliche Wortlänge kürzer. Es werden häufig Wörter verwendet wie „Sie werden“, „Sie sind“. Im Durchschnitt haben die Wörter bei den Clickbaits längere Abhängigkeiten als Nicht-Clickbaits. Der Hauptgrund ist die Existenz komplexerer Phrasensätze im Vergleich zu Schlagzeilen ohne Clickbait. Es ist außerdem zu sehen, dass in Clickbait-Schlagzeilen Stoppwörter häufiger verwendet werden. Clickbait Überschriften verwenden häufig Determinantien wie „ihre“, „meine“, die auf bestimmte Personen oder Dinge im Artikel verweisen. Die Verwendung Wörter dient in erster Linie dazu, den Benutzer neugierig auf das Objekt zu machen, auf das verwiesen wird, und ihn zu überzeugen, den Artikel weiter zu verfolgen. Um Daten zu erfassen, können auch Soziale Medien wie Twitter herangezogen werden, wie im Beispiel von [27].

Um lexikalischer und semantische und orthografische und morphologische Merkmale zu erfassen, benutzen die Autoren aus [28] Worteinbettungen und Zeicheneinbettungen, anstatt übermäßig Feature Selection auszuüben. Um Informationen außerhalb einzelner oder fester Wortfenster zu erfassen, untersuchten die Autoren dabei verschiedene RNN-Architekturen (Recurrent Neural Network) wie LSTM (Long Short Term Memory), GRU (Gated Recurrent Units) und Standard-RNNs. Dieses sind Wiederkehrende neuronale Netzwerkmodelle, die für sequentielle Daten wie Sprache und Text gut modelliert werden können.

Die Arbeit von [29] schlägt ein Modell vor, welches CNNs benutzt. CNNs werden für verschiedene Deep-Learning-Aufgaben verwendet. Es wurde nur eine Faltungsschicht in das CNN-Modell eingebaut. Die erste Schicht wird für das Einbetten der Wörter in Vektoren verwendet. Dabei wurden 2 verschiedene Worteinbettungen

in Bezug genommen. Ein vortrainiertes und eines welches von Grund auf lernen musste und sich während des Trainings weiterentwickelt. In der nächsten Schicht werden Filter in verschiedenen Größen verwendet um Faltungen über Wortvektoren zu erzeugen.

Die Autoren der Arbeit aus [30] Clustern zunächst die aus den Schlagzeilen erstellten Vektoren mit der sogenannten t-SNE Methode nach [31]. Dieser Algorithmus „rekategorisiert“ die Schlagzeilen in mehrere Gruppen und reduziert die vielen Dimensionen von Clickbaits im Datensatz. Die Autoren beginnen erst im nächsten Schritt mit dem Training. Dabei entstehen 11 Kategorien von Clickbaits (Mehrdeutig, Übertreibung oder Neckerei).

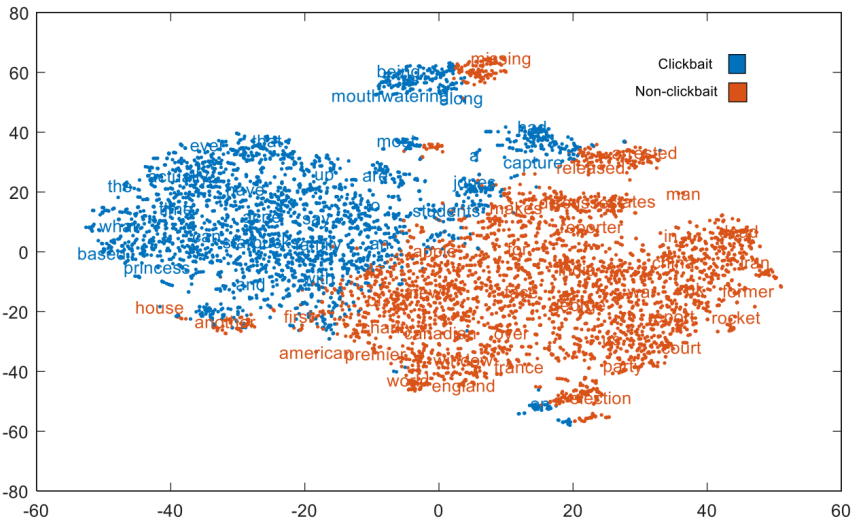

Abbildung 4.1: Die Autoren verwenden das t-SNE Algorithmus nach [31] um die Dimensionen zu Reduzieren und es entstehen mehrere Unterkategorien von Clickbaits. Entnommen aus [30].

4.3 Schluss

In diesem Kapitel wurden einige Arbeiten aus der Literatur vorgestellt, jedoch nicht alle. Es gibt außerdem noch weitere Arbeiten wie [32][33][34][35][36][37] und [38] die unter diese Literaturstudie fallen könnten. Alle diese Arbeiten in Ihrer ganzen Fülle zu beschreiben würde den Rahmen dieser Arbeit sprengen, sodass nur ein Einblick über mögliche Aufgaben, wie das Sammeln von Daten und das Bauen eines Deep Learning Ansatzes sowie die Besonderheiten die Clickbaits Schlagzeilen haben, verschafft wurde.

Abkürzungsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literatur

[1] F Rosenblatt, „THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN 1“, Techn. Ber. 6, S. 19–27.

[2] Paul J. Werbos, „Backpropagation Through Time: What It Does and How to Do It“, *Proceedings of the IEEE*, Jg. 78, Nr. 10, S. 1550–1560, 1990. DOI: 10.1109/5.58337.

[3] Geoffrey E Hinton und Simon Osindero, „A Fast Learning Algorithm for Deep Belief Nets Yee-Whye Teh“, Techn. Ber.

[4] Yann Lecun, Yoshua Bengio und Geoffrey Hinton, „Deep learning“, *Nature*, Jg. 521, Nr. 7553, S. 436–444, 2015. DOI: 10.1038/nature14539.

[5] MA Nielsen, „Neural networks and deep learning“, 2015.

[6] M. W. Gardner und S. R. Dorling, „Artificial neural networks (the multilayer perceptron) - a review of applications in the atmospheric sciences“, *Atmospheric Environment*, Jg. 32, Nr. 14-15, S. 2627–2636, 1998. DOI: 10.1016/S1352-2310(97)00447-0.

[7] Francois Chollet, *Deep learning with Python*. 2017. DOI: 10.23919/ICIF.2018.8455530.

[8] Antonio Guili; Amita Kapoor; Sujit Pal, *Deep Learning with TensorFlow 2 and Keras: Regression, ConvNets, GANs, RNNs, NLP, and More with TensorFlow 2 and the Keras API*. 2019.

[9] Aaron Courville Ian Goodfellow, Yoshua Bengio, *Deep Learning*. 2016.

[10] Sebastian Ruder, „An overview of gradient descent optimization algorithms“, Sep. 2016. arXiv: 1609.04747. Adresse: http://arxiv.org/abs/1609.04747.

[11] *Gradient Descent: All You Need to Know | Hacker Noon*. Adresse: https://hackernoon.com/gradient-descent-aynk-7cbe95a778da (besucht am 24. 12. 2020).

[12] Josh Patterson und Adam Gibson, *Deep Learning a Practitioner'S Approach*, 7553. 2019, Bd. 29, S. 1–73.

[13] Stanford University Course cs231n, „CS231n Convolutional Neural Networks for Visual Recognition", *Stanford University Course cs231n*, S. 30, 2018. Adresse: https://cs231n.github.io/neural-networks-3/%20 http://cs231n.github.io/convolutional-networks/%7B%5C%%7D0 Ahttp://cs231n.github.io/neural-networks-3/.

[14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky und Ruslan Salakhutdinov, „Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Techn. Ber., 2014, S. 1929–1958.

[15] Charu C. Aggarwal, *Neural Networks and Deep Learning*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-94463-0.

[16] Stanford University Course cs231n, „CS231n Convolutional Neural Networks for Visual Recognition", *Stanford University Course cs231n*, S. 30, 2018. Adresse: https://cs231n.github.io/convolutional-networks/.

[17] Elizabeth D Liddy, „SURFACE SURFACE Center for Natural Language Processing School of Information Studies (iSchool) 2001 Natural Language Processing Natural Language Processing Natural Language Processing 1", Techn. Ber. Adresse: https://surface.syr.edu/cnlp.

[18] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, Christian Jauvin, Jauvinc@iro Umontreal Ca, Jaz Kandola, Thomas Hofmann, Tomaso Poggio und John Shawe-Taylor, „A Neural Probabilistic Language Model", Techn. Ber., 2003, S. 1137–1155.

[19] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams, „Learning representations by back-propagating errors", *Nature*, Jg. 323, Nr. 6088, 1986. DOI: 10.1038/323533a0.

[20] Daniel Jurafsky und James H Martin, „Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition Third Edition draft", Techn. Ber.

[21] *Embeddings: Translating to a Lower-Dimensional Space*. Adresse: https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space (besucht am 15. 12. 2020).

[22] Tomas Mikolov, Kai Chen, Greg Corrado und Jeffrey Dean, „Distributed Representations of Words and Phrases and their Compositionality“, Techn. Ber., 2013. arXiv: 1310.4546v1.

[23] ——, „Efficient Estimation of Word Representations in Vector Space“, Techn. Ber. arXiv: 1301.3781v3. Adresse: http://ronan.collobert.com/senna/.

[24] Ye Zhang und Byron C Wallace, „A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification“, Techn. Ber. arXiv: 1510.03820v4. Adresse: http://nlp.stanford.edu/projects/.

[25] Rohit Francois Chaubard und Richard Socher Mundra, *Understanding Convolutional Neural Networks for NLP – WildML.* Adresse: http://www.wildml.com/2015/11 /understanding-convolutional-neural-networks-for-nlp/ (besucht am 15. 12. 2020).

[26] Abhijnan Chakraborty, Bhargavi Paranjape, Sourya Kakarla und Niloy Ganguly, „Stop Clickbait: Detecting and Preventing Clickbaits in Online News Media“, Techn. Ber. arXiv: 1610.09786v1.

[27] Martin Potthast, Tim Gollub, Matthias Hagen und Benno Stein, „The Clickbait Challenge 2017: Towards a Regression Model for Clickbait Strength“, Techn. Ber. arXiv: 1812.10847v1. Adresse: https://clickbait-challenge.org.

[28] Ankesh Anand, Tanmoy Chakraborty und Noseong Park, „We used Neural Networks to Detect Clickbaits: You won't believe what happened Next!“, Techn. Ber., 2019. arXiv: 1612.01340v2.

[29] Amol Agrawal, „Clickbait detection using deep learning“, *Proceedings on 2016 2nd International Conference on Next Generation Computing Technologies, NGCT 2016*, Nr. October, S. 268–272, 2017. DOI: 10.1109/NGCT.2016.7877426.

[30] Abinash Pujahari und Dilip Singh Sisodia, „Clickbait Detection using Multiple Categorization Techniques“, Techn. Ber. Adresse: www.clickbait-challenge.org.

[31] Laurens Van Der Maaten und Geoffrey Hinton, „Visualizing Data using t-SNE“, Techn. Ber., 2008, S. 2579–2605.

[32] Sarjak Chawda, Aditi Patil, Abhishek Singh und Ashwini Save, „A Novel Approach for Clickbait Detection“, in *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, IEEE, 2019, S. 1318–1321.

[33] Savvas Zannettou, Sotirios Chatzis, Kostantinos Papadamou und Michael Sirivianos, „The good, the bad and the bait: Detecting and characterizing clickbait on youtube“, *Proceedings - 2018 IEEE Symposium on Security and Privacy Workshops, SPW 2018*, S. 63–69, 2018. DOI: 10.1109/SPW.2018.00018.

[34] Vaibhav Kumar, Dhruv Khattar, Siddhartha Gairola, Yash Kumar Lal und Vasudeva Varma, „Identifying Clickbait: A Multi-Strategy Approach Using Neural Networks * Identifying Clickbait: A Multi-Strategy Approach Using“, DOI: 10.1145/3209978.3210144. Adresse: https://doi.org/10.1145/3209978.3210144.

[35] Philippe Thomas, „Clickbait Identification using Neural Networks The Whitebait Clickbait Detector at the Clickbait Challenge 2017“, Techn. Ber. arXiv: 1710.08721v1.

[36] Feng Liao, Hankz Hankui Zhuo, Xiaoling Huang und Yu Zhang, „Federated Hierarchical Hybrid Networks for Clickbait Detection“, Techn. Ber. arXiv: 1906.00638v1.

[37] Maria Glenski, Ellyn Ayton, Dustin Arendt und Svitlana Volkova, „Fishing for Clickbaits in Social Images and Texts with Linguistically-Infused Neural Network Models The Pineapplefish Clickbait Detector at the Clickbait Challenge 2017“, Techn. Ber. arXiv: 1710.06390v1. Adresse: http://www.clickbait-challenge.org/.

[38] Prakhar Biyani, Kostas Tsioutsiouliklis und John Blackmer, „Detecting Clickbaits in News Streams Using Article Informality“, *Thirtieth AAAI Conference on Artificial Intelligence*, S. 94–100, 2016.