technische universität
dortmund

Bachelorarbeit

Generierung von Syntax-Bäumen mit Generative Adversarial Networks

Jan-Philip Richter
Januar 2020

Gutachter:
Prof. Dr. Morik, Katharina
Pfahler, Lukas

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
https://www-ai.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Diese Arbeit ist motiviert durch ein Fachprojekt „Data-Mining und Datenanalyse“ 2018 / 2019 am Lehrstuhl für Künstliche Intelligenz, LS VIII, der Technischen Universität Dortmund. Ziel des Projektes war die Konstruktion einer Suchmaschine, die auf einem Datensatz wissenschaftlicher Arbeiten von arxiv.org arbeitet.[1] Statt einer Schlagwortsuche sollte die Suche über die Eingabe mathematischer Formeln möglich sein, so dass Arbeiten mit semantisch oder thematisch ähnlichen Formeln als Ergebnis vorgeschlagen werden. Um das mit Mitteln maschinellen Lernens zu erreichen, wurde ein Trainingsdatensatz benötigt, der zu mathematischen Formeln semantisch ähnliche Formeln bereitstellt. Zu diesem Zweck wurden mit einem Skript Syntaxbäume solcher Formeln generiert und semantisch äquivalent umgeformt, sodass für jede generierte Formel ein Satz von ähnlichen Formeln vorlag. Die Generierung erfolgte nach einem einfachen Zufallsprinzip, mathematische Operatoren und Bezeichner für Variablen wurden dabei nach einer Gleichverteilung ausgewählt.

Dieser Ansatz hat sich insofern als wenig fruchtbar erwiesen, als dass nach einer Gleichverteilung generierte Formeln zum größten Teil keine im Sinne der Zielsetzung sinnvollen äquivalenten Umformungen ergeben haben. Fraglich blieb außerdem, ob die Modellierung eines Ähnlichkeitsmaßes nicht nur dann gelingen kann, wenn mit Daten trainiert wird, die denen, auf denen später eine Suche nach ähnlichen Datenpunkten laufen soll, auch tatsächlich ähnlich sind. So mag ein neuronales Netz dahingehend optimiert werden, dass es semantische Ähnlichkeit zwischen Formeln aus dem Bereich Elektrotechnik bemessen kann. Deshalb muss es diese aber nicht auch auf Formeln aus dem Bereich der Statistik bemessen können. Das muss natürlich umso mehr gelten, wenn auf Formeln aus einer Gleichverteilung trainiert wurde.

[1] Informationen und Datensatz unter https://whadup.github.io/EquationLearning/, vlg. [31]

Ziel dieser Arbeit ist es, für einen gegebenen Datensatz von Bildern mathematischer Formeln Syntaxbäume für künstliche Formeln zu generieren, die möglichst nicht von den echten Formeln zu unterscheiden sind.

1.2 Problemstellung

Im Verlauf dieser Arbeit wird oft das Wort Sequenz synonym für Syntaxbaum verwendet. Rechtfertigung findet diese vermeintliche Ungenauigkeit darin, dass nur eine Teilmenge von Sequenzen in dieser Arbeit besprochen wird - nämlich solche, die eindeutig einem Syntaxbaum zugeordnet werden können. In Kapitel 5 werden wir eine Äquivalenz für die hier besprochenen Syntaxbäume zu jeweiligen Sequenzen zeigen. Das vorgestellte Problem kann so unter der Perspektive der Sequenzgenerierung mit neuronalen Netzen betrachtet werden, eine Thematik, zu der in den letzten Jahren einige Forschung betrieben wurde.[6][12][20][29][48] Im Gegensatz zu vielen aktuellen Forschungsansätzen, die keine Garantien für grammatische Korrektheit geben[2], soll in dieser Arbeit die Struktur generierter Sequenzen korrekter mathematischer Syntax entsprechen. Dabei lehnt der hier vorgestellte Ansatz stark an den von Yu et al. (2017) vorgestellten Sequence Generative Adversarial Networks (SeqGANs) an, deren Methodik für natürlichsprachliche Sequenzen zur Generierung von Sequenzen mathematischer Formeln adaptiert werden soll.[48] Formal handelt es sich dabei um das Problem für eine gegebene Verteilung p_{echt} mathematischer Formeln ein Modell p_{unecht} zu konstruieren, so dass $p_{echt} = p_{unecht}$[16].

Die Optimierung einer Funktion mit diskretem Wertebereich ist nicht trivial.[48] Zur Optimierung eines Modells werden beim maschinellen Lernen typischerweise sukzessiv Parameter einer Funktion so angepasst, dass eine kleine Änderung des Modells bewirkt wird. Voraussetzung dafür ist die Differenzierbarkeit der zu optimierenden Funktion. Diese Eigenschaft liegt für eine Funktion mit einem diskreten Lösungsraum nicht vor, eine kleine Änderung der Ausgabe ist hier nicht definiert.[48] Für das Generieren von Syntaxbäumen muss daher eine Möglichkeit gefunden werden, dieses Problem zu umgehen. Konkret ergibt sich außerdem eine gewisse Hürde aus der Ungleichförmigkeit der Daten. Ziel ist das Generieren von Syntaxbäumen, der zur Verfügung stehende Datensatz besteht aus Bildern. Es stellt sich also die Frage, in welcher Form der Datensatz zur Optimierung eines Modells verwendet werden kann, da keine Informationen zu den Syntaxbäumen der echten Formeln oder über ihre Verteilung p_{echt} vorliegen.

Bei SeqGANs handelt es sich um eine Variante der von Goodfellow et al. (2014) entwickelten Generative Adversarial Networks (GANs).[48] Diese haben gerade bei der Generierung von Daten große Aufmerksamkeit erfahren.[6][12][43] Durch ihre, im Vergleich zu herkömmlichen generativen Modellen besondere Struktur, die in Kapitel 4 genauer erläutert werden soll, bringen sie jedoch auch einige Probleme mit sich, die es zu bewälti-

[2] vgl. [6][12][20][29][48]

gen gilt. So gelten die bei der Optimierung verwendeten Trainingsimpulse als notorisch instabil.[12][32][26]

1.3 Vorgehen

Zu Zwecken der Übersicht soll das Vorgehen ohne weitere Erläuterungen vorweg skizziert werden. Auf die verwendeten Methoden wird im Anschluss genauer eingegangen. In Abbildung 1.1 ist die Methode visualisiert.

- Ein generatives Modell G soll über Zeichen eines Alphabets Sequenzen Y erzeugen. Um das Problem der Generierung diskreter Daten zu lösen, wird der Generierungsprozess als sequentielles Entscheidungsproblem aufgefasst.[2][3] Um eine Sequenz $Y_{1:T}$ mit Zeichen $y_1, \ldots, y_T \in \Sigma$ zu generieren, wird G so trainiert, dass es für eine Eingabe $Y_{1:t-1}$ eine Verteilung über Σ repräsentiert. Nach einem stochastischen Auswahlprozess wird die Sequenz $Y_{1:t-1}$ mit einem Zeichen y_t fortgesetzt, bis die Sequenz beendet wird.[48]

- Es soll ein Alphabet Σ und eine Grammatik P so definiert werden, dass die Zielmenge der Verteilung p_{echt} durch das trainierte Modell möglichst gut abgebildet werden kann. Das Alphabet Σ soll die einzelnen Zeichen und Operatoren enthalten, während die Grammatik ihren Kontext definieren muss.

- Die Sequenzen werden in der Postfixnotation interpretiert, was eine Übersetzung in Syntaxbäume erlaubt. Diese werden zu LaTeX-Code übersetzt, zu PDF-Dateien kompiliert und schließlich in ein Bildformat formatiert, das den Bildern des Datensatzes entspricht.

- Ein klassifizierendes neuronales Netz D wird trainiert, Bilder aus p_{echt} von Bildern aus p_{unecht} zu unterscheiden. $D(Y)$ entspricht der Wahrscheinlichkeit, dass $Y \sim p_{unecht}$, also Y künstlich generiert wurde. $D(Y_{Y \sim p_{unecht}})$ dient als Feedback im Optimierungsprozess von G.[16]

- Durch ein abwechselndes Training von D und G soll G so optimiert werden, dass Sequenzen erzeugt werden können, die nicht mehr von denen des Datensatzes unterschieden werden können.[48]

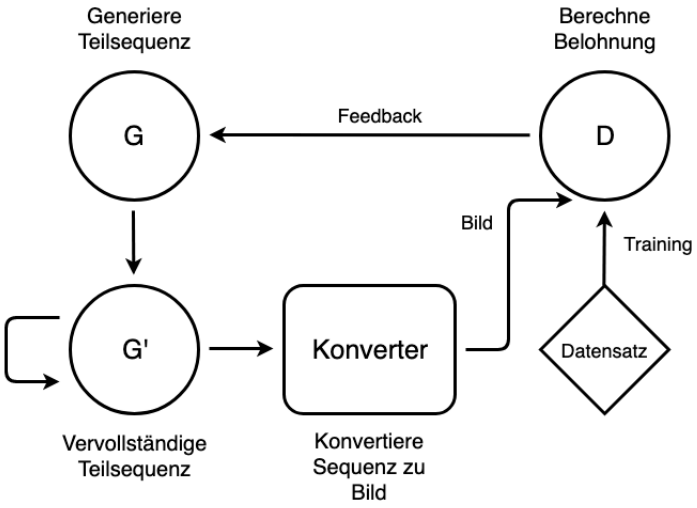

Abbildung 1.1: Der Optimierungsprozess für G - Teilsequenzen werden durch ein zweites Modell G' vervollständigt. Zur Vereinfachung kann angenommen werden, dass $G = G'$. Warum eine Unterscheidung sinnvoll ist, wird in Kapitel 4 besprochen.

Kapitel 2

Grundlagen zu neuronalen Netzen

Bei einem neuronalen Netz handelt es sich, einfach formuliert, um eine komplizierte, parametrisierte Funktion.[15]$^{S.164}$ Das Training eines neuronalen Netzes bedeutet, die Parameter der Funktion schrittweise so anzupassen, dass für eine Eingabe eine gewünschte Ausgabe erfolgt.[49] Beispielsweise wäre für die Eingabe einer Teilsequenz in unserem Fall die Ausgabe eines Zeichens erwünscht, das die Formel zu einer echt aussehenden Formel vervollständigt. Bei der Einführung zu neuronalen Netzen beschränken wir uns auf die Grundlagen und die Einordnung einiger Begrifflichkeiten. Es werden im Folgenden aus Gründen der Lesbarkeit oft die in der Literatur geläufigen, englischen Begriffe verwendet. Außerdem werden Kommazahlen mit einem Punkt statt einem Komma notiert, um Uneindeutigkeiten bei der Aufzählung mit einem separierenden Komma zu vermeiden.

2.1 Struktur neuronaler Netze

Formal ist das Ziel beim Training neuronaler Netze die Approximation einer Funktion f^*.[15]$^{S.164}$ Das neuronale Netz selbst repräsentiert dabei eine Funktion f_θ mit Parametern θ. Bei der Optimierung des Netzes bezüglich der Funktion f^* lernt das Netz dabei einen Wert θ, so dass f^* bestmöglich approximiert wird. Ein neuronales Netz f besteht in der Regel aus einer Komposition $f = f^n \circ \cdots \circ f^i \circ \cdots \circ f^1$ von Funktionen $f^1, \ldots, f^i, \ldots, f^n$.[15]$^{S.164}$ Wir bezeichnen eine Funktion f^i dieser Komposition als Layer. Das erste Layer f^1 wird auch Input-Layer und das letzte Layer f^n Output-Layer genannt. Layer f^i zwischen diesen Layern nennen wir Hidden-Layer.[15]$^{S.164}$

Strukturell lässt sich ein neuronales Netz als eine Menge von Neuronen genannten Knoten und eine Menge von gerichteten Kanten zwischen diesen Knoten interpretieren. Mit jedem Neuron sind eine Menge von j eingehenden Kanten und j' ausgehenden Kanten assoziiert, über die es Eingaben $x_1, \ldots, x_j$ empfängt und Ausgaben $y_1, \ldots, y_{j'}$ sendet.[15]$^{S.164}$ Alle Knoten der gleichen Tiefe bilden jeweils ein Layer, das eine Funktion aus der Komposition des gesamten Netzes repräsentieren.[15]$^{S.164}$ Eine Eingabe wird Layer für Layer durch das

Netz propagiert, bis eine Ausgabe im Output-Layer erfolgt. Die Ausgabe für ein i-tes Neuron eines Layers mit m eingehenden Kanten berechnet sich dabei mit $y_i = \varphi(\Sigma_{j=0}^{m} w_{ij} x_j)$, wobei die Koeffizienten w_{ij} mit den m eingehenden Kanten assozierte Gewichte sind, so zu sehen in Abbildung 2.1.[15]$^{S.168}$ Bei dem Wert x_0 handelt es sich üblicherweise um einen sogenannten Bias Wert, der nicht Teil der Eingabe, sondern Teil der Parameter des Netzes ist.[15]$^{S.168}$ Wir werden ihn im Laufe der Ausführungen teilweise mit einem eigenen Bezeichner b kennzeichnen. Die Funktion φ bezeichnet man als Aktivierungsfunktion.[15]$^{S.168}$

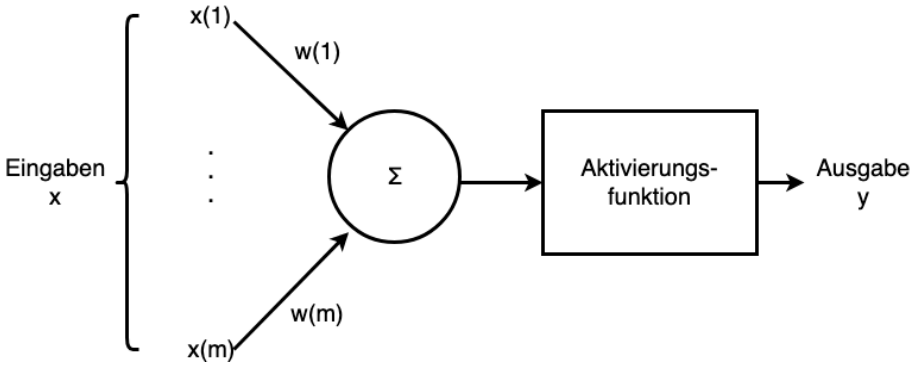

Abbildung 2.1: Ein einzelnes Neuron eines neuronalen Netzes mit m eingehenden Kanten und einer Ausgabe y.

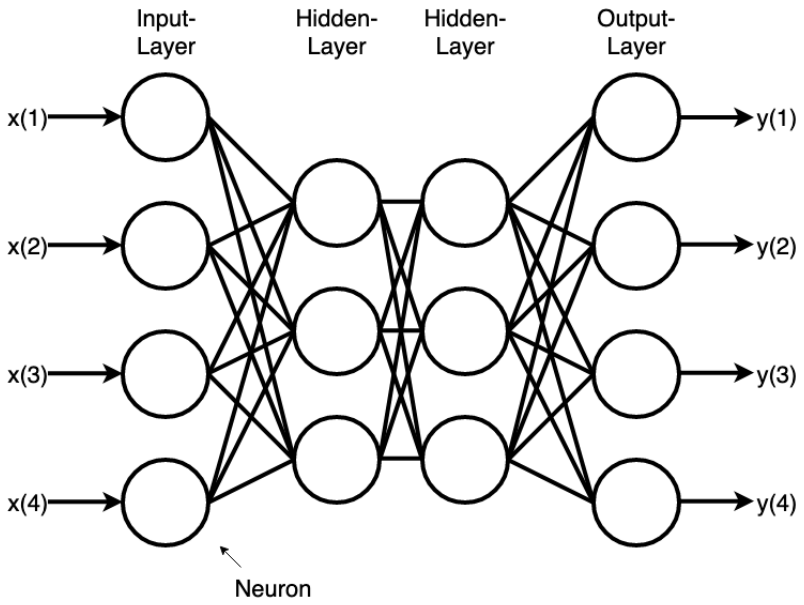

Abbildung 2.2: Ein neuronales Feedforward-Netz mit zwei Hidden-Layern - eine Eingabe x wird durch das Netz propagiert, bis im Output-Layer eine Ausgabe y erzeugt wird. Ein einzelnes Neuron ist in 2.1 detailliert abgebildet.

Nimmt man nun die Layer eines Netzes zusammen, wie in Abbildung 2.2 skizziert, ergibt sich die Funktion parametrisiert durch die Gewichtsmatrizen w und die Biasvektoren

b pro Layer. Varianten neuronaler Netze können dabei durchaus weitere lernbare Parameter haben.[1]

2.2 Backpropagation-Algorithmus

Um eine günstige Belegung der Parameter θ mit Hinblick auf die zu approximierende Funktion zu finden, wird in der Regel der Backpropagation-Algorithmus eingesetzt.[15]$^{S.173}$[39] Bei dem Backpropagation-Algorithmus handelt es sich um ein Gradientenverfahren.[2][39] Als Gradientenverfahren bezeichnet man schrittweise die Optimierung einer Funktion f durch Anpassung ihrer Parameter θ nach der Form

$$\theta_{t+1} = \theta_t + \alpha \nabla f(\theta) \tag{2.1}$$

für Schritt t, Gradienten $\nabla f(\theta)$ von $f(\theta)$ und eine Schrittgröße $\alpha > 0 \in \mathbb{R}$.[49] Der Gradient einer Funktion $f(x_1, \ldots, x_n)$ ist definiert als der Vektor der partiellen Ableitungen von f für $\frac{\partial f}{\delta x} = [\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}]$. Er zeigt also in die Richtung, in die eine Funktion f hinsichtlich ihrer Parameter steigt.[35]$^{S.684}$ Abhängig von der Schrittgröße α konvergiert die Folge θ_t dabei zu einem stationären Punkt von f, also einem Punkt, in dem die Ableitungen entweder 0 sind oder die Funktion an diesem Punkt nicht differenzierbar ist.[15]$^{S.80f.}$[49] Ziel ist es, mit diesem Punkt ein Extremum zu finden. Die Schrittgröße wird im Folgenden auch als Lernrate bezeichnet. Minimiert werden soll nicht die Ausgabe des Netzes, sondern die Abweichung der ausgebenen Ist-Werte von den Soll-Werten. Um diese Abweichung zu messen wird eine Fehlerfunktion definiert.[49] Ein klassisches Beispiel für eine Fehlerfunktion ist der quadratische Fehler $e(y) = \frac{1}{2}\Sigma_{i=1}^{n}(y_i' - y_i)^2$ für n Eingaben, Sollwerte y' und die errechneten Ausgaben y.[49] Der Backpropagation-Algorithmus wiederholt dazu folgende Phasen[39]:

- Eine Eingabe wird durch das Netz propagiert, bis im Output-Layer eine Ausgabe vorliegt.[15]$^{S.208}$
- Mithilfe der Fehlerfunktion wird ein Fehler berechnet.[15]$^{S.209}$
- Der Fehler wird durch das Netz zurückpropagiert und die Gewichte werden so verändert, dass der Fehler minimiert wird (der Backpropagation-Schritt).[39]

Im letzten Schritt wird der Gradient der Fehlerfunktion hinsichtlich der Netzparameter berechnet. Der Einfachheit halber gehen wir an dieser Stelle davon aus, dass innerhalb

[1] vgl. Kapitel 6 „Recurrent Neural Networks“

[2] Goodfellow et al. (2016) unterscheiden zwischen Backpropagation als reinem Differenzierungsverfahren und dem Gradientenverfahren, das zur Aktualisierung der Gewichte genutzt wird.[15]$^{S.200}$ Wir folgen Rumelhart et al. (1986) und bezeichnen mit dem Backpropagation-Algorithmus im Folgenden eine Prozedur, die schrittweise die Gewichte des neuronalen Netzes anpasst.[39]

des Netzes keine Zykel existieren, dass es sich also um ein sogenanntes „Feedforward"-Netz handelt.[15]$^{S.164}$ Wie der Backpropagation-Algorithmus dahingehend angepasst werden kann, dass auch Rückwärtskanten erlaubt sind, wird in Kapitel 6 besprochen.

Sei die Ausgabe y_j eines Neurons j durch $y_j = \varphi(z_j)$ definiert, wobei $z_j = \Sigma_{i=1}^{m} x_i w_{ij}$. Dann kann die partielle Ableitung der Fehlerfunktion e durch Verwendung der Kettenregel berechnet werden[15]$^{S.205}$[39]:

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}. \tag{2.2}$$

Das Inkrement Δ für ein einzelnes Gewicht w_{ij} bestimmt sich über

$$\Delta w_{ij} = \alpha \cdot -\frac{\partial e}{\partial w_{ij}} \cdot y_i = \alpha(-\delta_j) y_i, \tag{2.3}$$

wobei δ_j einer Art Fehlersignal des Neurons j entspricht.[15]$^{S.209}$ Da wir den Fehler minimieren wollen, folgen wir der absteigenden Richtung des Gradienten $-\frac{\partial e}{\partial w_{ij}}$. Der Faktor δ_j berechnet sich wie folgt:[21]

$$\delta_j = \begin{cases} e'(y_j) \cdot \varphi'(z_j), & \text{falls Neuron j im Output-Layer} \\ \Sigma_k(\delta_k w_{jk}) \cdot \varphi'(z_j), & \text{falls j in einem Hidden-Layer.} \end{cases} \tag{2.4}$$

Dabei sind die Werte k die Indizes der nachfolgenden Neuronen von j im nächsten Layer. So wird der Beitrag zum Fehler von Neuronen eines Layers zurückgeführt auf Neuronen der vorderen Layer, deren Eingaben sie verarbeiten. Nach dem Gradientenverfahren können die Gewichte dann mit $w_{ij}^{t+1} = w_{ij}^{t} + \Delta w_{ij}^{t}$ aktualisiert werden.[15]$^{S.209}$

Wir bezeichnen die Optimierung eines Netzes hinsichtlich einer Fehlerfunktion auch als Training des Netzes. Wenn in Verbindung mit einem Training von einem Datensatz geschrieben wird, dann ist damit die Menge der Eingabedaten gemeint, mit deren Hilfe das Netz trainiert wird. Diese Daten werden üblicherweise aus Performanzgründen nicht einzeln durch das Netz propagiert, sondern parallel in kleineren Mengen, den sogenannten Mini-Batches. Um das zu realisieren, werden die berechneten Fehler über einen Mini-Batch gemittelt und der gemittelte Fehler zurückpropagiert.[49] In dieser Arbeit wird eine Variante des Backpropagation-Algorithmus, der Adam-Optimierungsalgorithmus eingesetzt.[27] Eine Erläuterung dazu findet sich im Anhang in Abschnitt 1.

2.3 Aktivierungsfunktionen

In der Praxis sind je nach Anwendungsgebiet viele verschiedene Aktivierungsfunktionen gebräuchlich. In der Regel haben sie bis auf Ausnahmen aber einige Eigenschaften gemein: 1. Sie sind nicht linear. Ziel ist es typischerweise, mit dem neuronalen Netz eine nicht-lineare Funktion zu approximieren.[15]$^{S.165,186}$ Die Komposition linearer (Aktivierungs-) Funktionen ist aber selbst auch linear. Nicht-lineare Funktionen wären also nicht darstellbar.

2. Sie sind differenzierbar.[15]$^{S.188}$ Das ist Voraussetzung dafür, dass gradientenbasierte Optimierungsalgorithmen angewendet werden können. Dabei ist anzumerken, dass Aktivierungsfunktionen diese Eigenschaften nicht zwangsweise aufweisen müssen. Üblicherweise verwendete und auch im Rahmen dieser Arbeit vorgeschlagene Aktivierungsfunktionen sind zum Beispiel die Sigmoidfunktion $\varphi(x) = \frac{1}{1+exp(-x)}$ und die Rectified Linear Unit $\varphi(x) = max(0, x)$.[15]$^{S.189ff.}$

Kapitel 3

Policy Gradient

Den Prozess der Generierung von Sequenzen als sequentielles Entscheidungsproblem zu interpretieren, bedeutet konkret, Sequenzen Zeichen für Zeichen schrittweise zu erzeugen. Im Kapitel zu neuronalen Netzen wurde beschrieben, wie ein generatives Modell G mithilfe eines Fehlersignals optimiert werden kann. Im Bereich des maschinellen Lernens bezeichnet Reinforcement Learning eine Kategorie von Methoden, deren Basis die Interaktion eines Agenten, das heißt eines Akteurs mit seiner Umgebung bildet. Dabei führt der Agent Aktionen aus und erhält in der Folge Feedback aus der Umgebung.[41]$^{S.2}$ Das vorgestellte Problem soll nun vor dem Hintergrund des Reinforcement Learnings betrachtet werden.

3.1 Markov Decision Process

Die Interaktion zwischen Agent und Umgebung wird typischerweise als Markov Decision Process modelliert.[41]$^{S.47}$ Ein Markov Decision Process ist eine Sequenz von Zuständen und Aktionen, die einen Entscheidungsprozess simulieren soll. Dabei wird der Prozess in diskrete Zeitschritte unterteilt. In jedem Zeitschritt herrscht ein bestimmter Zustand und es stehen zustandsspezifisch eine Reihe möglicher Aktionen zur Auswahl. Wird eine bestimmte Aktion ausgewählt, erfolgt abhängig vom Zustand und der Auswahl eine Transition in einen neuen Zustand. Der Auswahl wird außerdem ein numerischer Wert zugeordnet, der als Belohnungssignal dient.[41]$^{S.48ff.}$ Formal kann ein Markov Decision Process als Tupel (S, A, T, p, r) beschrieben werden.[41]$^{S.48}$ Dabei sei

- S die Menge der möglichen Zustände
- A die Menge der möglichen Aktionen
- $T \subseteq \mathbb{N}$ die Menge der Zeitschritte
- $p : S \times A \times T \times S \to [0..1]$ eine Funktion, die die Wahrscheinlichkeit $P(s_{t+1}|s_t, a_t)$ einer Transition in einen Zustand s_{t+1} aus einem Zustand s_t mit einer Aktion a_t angibt

- $r : S \times A \times T \to \mathbb{R}$ eine Funktion, die einer Aktion eine numerische Belohnung zuordnet

Außerdem muss der Prozess die Markov Eigenschaft erfüllen. Informell beschreibt die Markov Eigenschaft die Eigenschaft eines Zustands $s_t \in S$, alle entscheidungsrelevanten Informationen aus dem vorangegangenen Entscheidungsprozess zu beinhalten.[41]$^{S.49}$ So wird sichergestellt, dass die Entscheidung zu einem Zeitpunkt t ausschließlich von s_t abhängt. Sei

$$\mathbb{P}(s_{t+1}, r_{t+1}|s_t, a_t) \tag{3.1}$$

also die Wahrscheinlichkeit in den Zustand s_{t+1} zu wechseln und dabei Belohnung r_{t+1} zu erhalten, gegeben s_t und a_t, und

$$\mathbb{P}(s_{t+1}|\ s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, r_1, s_0, a_0) \tag{3.2}$$

die Wahrscheinlichkeit in den Zustand s_{t+1} zu wechseln und dabei Belohnung r_{t+1} zu erhalten, wenn alle Werte des bisherigen Entscheidungsprozesses gegeben sind. Dann gilt die Markov Eigenschaft genau dann, wenn 3.1 gleich 3.2 für alle $s_{t+1} \in S$, $r \in \mathbb{R}$ und alle möglichen Verläufe s_t,a_t,r_t,s_{t-1},a_{t-1},r_{t-1}, $\ldots$,r_1,s_0,a_0 gilt.[41]$^{S.49}$

Aus der Perspektive des Reinforcement Learning ist diese Eigenschaft deshalb hilfreich, weil sie es erlaubt, zukünftige Zustände und Belohnungen anhand eines Ausgangszustands abzuschätzen. Damit ist es möglich, einzelnen Zuständen oder Aktionen in gegebenen Zuständen einen Performanzwert zuzuordnen, abhängig von den Belohnungen, die von Folgezuständen aus erreicht werden können.

3.2 Grundlagen zum Reinforcement Learning

Sei nun ein Agent ein Entscheidungsträger in einem Markov Decision Process. Er befindet sich also zu jedem Zeitpunkt in einem fest definierten Zustand und hat eine Auswahl von Aktionen zur Verfügung. Wird eine dieser Aktionen ausgewählt, bekommt der Agent von der Umgebung Feedback in Form von Folgezustand und Belohnung für die ausgewählte Aktion, wie Abbildung 3.1 dargestellt. In diesem Zusammenhang wird auch vom Modell gesprochen. Gemeint ist das mathematische Modell, dem der Entscheidungsprozess zugrunde liegt. Kenntnis des Modells bedeutet also Kenntnis über die Verteilung der Transitionswahrscheinlichkeiten und die Berechnung des Belohnungssignals.[41]$^{S.48}$

Der Prozess der Auswahl von Aktionen wird Strategie genannt und mit π notiert. Eine Strategie kann entweder deterministisch sein oder als eine Wahrscheinlichkeitsverteilung über die verschiedenen Aktionen modelliert werden.[41]$^{S.58}$ Im Folgenden wird mit π eine stochastische Strategie impliziert. Mit $\pi_t(s, a) = \mathbb{P}(a|s)$ sei die Wahrscheinlichkeit bezeichnet, dass der Agent im Zustand s_t die Aktion a_t auswählt.

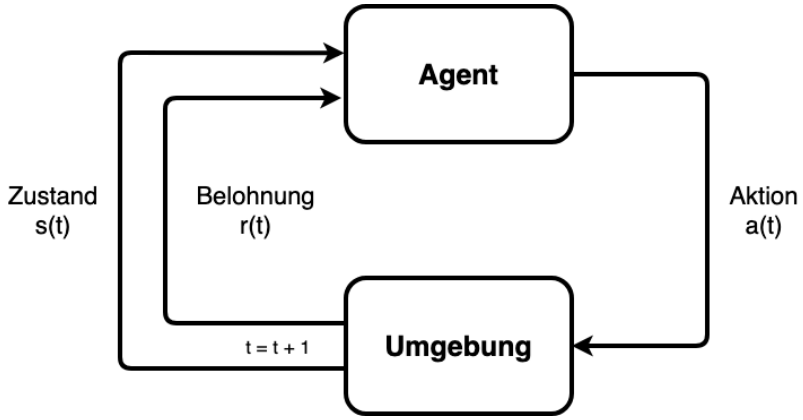

Abbildung 3.1: Ein Agent führt im Kontext einer Umgebung Aktionen aus und bekommt Feedback in Form eines veränderten Zustands und eines Belohnungssignals. Frei aus dem Englischen nach [41]$^{S.48}$.

Das Problem, das der Agent durch Anwendung der Strategie lösen soll, wird als Task bezeichnet. Es können grundsätzlich zwei verschiedene Kategorien unterschieden werden: kontinuierliche und episodische Tasks. Episodische Tasks zeichnen sich durch einen terminierenden Zustand aus. Kontinuierliche Tasks sind solche, die sich ohne terminierenden Zustand beliebig fortsetzen.[41]$^{S.54f.}$ Im Folgenden sind mit Tasks ausschließlich episodische Tasks gemeint, da die Problemstellung der Natur nach episodisch ist und diese Beschränkung eine formale Vereinfachung darstellt.

Ziel ist es eine Strategie zu finden oder zu approximieren, die unter Berücksichtigung der gesammelten Belohnungen optimal ist. Dazu wird eine Zielfunktion $\mathbb{J}(\pi)$ formuliert, die maximiert werden soll. In der Zielfunktion werden die erwarteten Belohnungen akkumuliert, die der Agent durch das Befolgen der Strategie erhalten kann.[41]$^{S.53}$ Typischerweise geschieht das zum Beispiel durch das Berechnen der Durchschnittsbelohnung pro Zeitschritt mit[41]$^{S.53}$

$$\mathbb{J}(\pi) = \lim_{n\to\infty} \frac{1}{n}\mathbb{E}[r_1 + r_2 + \cdots + r_n|\pi] \tag{3.3}$$

oder auch durch das Berechnen der Summe der Langzeitbelohnungen mit einem sogenannten discount $\gamma \in [0..1]$ über[41]$^{S.54}$

$$\mathbb{J}(\pi) = \mathbb{E}[\sum_{t=1}^{\infty} \gamma^{t-1} r_t|\ s_0, \pi] \tag{3.4}$$

Der discount γ sorgt dafür, dass kurzfristige sicherere Belohungen stärker gewichtet werden als Belohnungen, die noch viele Schritte entfernt liegen.[41]$^{S.54}$ In der Praxis wird man selten eine optimale Strategie berechnen können und sich mit einer Approximation begnügen müssen. Selbst wenn das Modell bekannt ist, scheitert es schnell an der Komplexität der Berechnung.[41]$^{S.66}$ So sind zum Beispiel die Regeln des Schachspiels bekannt,

trotzdem ist ein optimaler Zug zu Beginn des Spiels in einem vernünftigen Zeitraum nicht berechenbar. Auf Grund der Menge an möglichen Zuständen würde auch der benötigte Speicher zum Flaschenhals.

3.3 Approximation einer optimalen Strategie

Das Ziel ist die Optimierung der Strategie unter Berücksichtigung der Zielfunktion. Im Bereich des Reinforcement Learnings wird zu diesem Zweck typischerweise eine sogenannte value function $V_\pi(s)$ approximiert, die unter Berücksichtigung einer Strategie π für einen Zustand s angibt, wie hoch die Summe der Belohnungen ist, die erwartungsgemäß von diesem Zustand aus gesammelt werden können.[41]$^{S.58}$ Formal lässt sich $V_\pi(s)$ definieren als

$$V_\pi(s) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s]. \tag{3.5}$$

Alternativ lässt sich auch eine value function $Q_\pi(s, a)$ definieren, die statt eines Zustandes eine Aktion in einem bestimmten Zustand bewertet mit[41]$^{S.58}$

$$Q_\pi(s, a) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a]. \tag{3.6}$$

Anhand dieser Funktionen lässt sich leicht eine optimale Strategie, zum Beispiel über eine gierige Auswahl $\pi^*(s) = \arg\max_{a \in A(s)} Q(s, a)$, modellieren. Methoden, die eine solche Funktion $Q(s, a)$ approximieren, werden auch als „actor critic“ Methoden bezeichnet.[41]$^{S.331}$

Eine andere Möglichkeit eine optimale Strategie zu bestimmen ist die Optimierung einer Strategie direkt im Suchraum aller möglichen Strategien.[41]$^{S.67}$ Dazu wird die Strategie selbst als parametrisierte Funktion modelliert, die auf die Wahrscheinlichkeiten der möglichen Aktionen abbildet. Wenn eine Aktion ausgeführt und eine Belohnung erhalten wird, wird die Strategie verändert, um in der Zukunft Aktionen mit hohen langfristigen Belohnungen wahrscheinlicher zu machen. Das wird wiederholt, bis die Funktion zu einer optimalen Strategie konvergiert.[41]$^{S.321ff.}$ Formal definieren wir eine Strategie π_θ mit einem Vektor von Parametern $\theta \in \mathbb{R}^k$. Sei s_0 ein beliebiger, aber fest gewählter Startzustand und $r_{\pi_\theta}(s)$ eine Funktion, die für einen Zustand s den Wert des Zustands unter der Strategie π_θ angibt. Dann können wir mit

$$\mathbb{J}(\theta) = \mathbb{E}_\pi[r_{\pi_\theta}(s_0)] \tag{3.7}$$

eine Zielfunktion angeben, die wir über die Optimierung der Parameter θ maximieren wollen.[41]$^{S.324}$ Ist eine Parameterbelegung θ gefunden, die $\mathbb{J}(\theta)$ maximiert, dann ist π_θ auch eine optimale Strategie.[41]$^{S.321}$ Da die Wahl der Aktion und die Einwirkung der

Umgebung durch Eingabe und Belohnungssignal zufällig sind, muss mit dem Erwartungswert gerechnet werden. Zur Optimierung wird dabei das Gradientenverfahren eingesetzt und dem Gradienten in aufsteigender Richtung gefolgt bis keine numerische Verbesserung mehr erzielt wird.[41]$^{S.321}$ Die Strategie wird entsprechend des Gradienten verändert

$$\theta_{k+1} = \theta_t + \alpha \nabla_\theta \mathbb{J}(\theta_k), \tag{3.8}$$

wobei mit α die nicht-negative Lernrate und mit k der Index der Aktualisierung bezeichnet sind.[41]$^{S.321}$ Dabei ist der Index der Aktualisierung nicht unbedingt gleich einem Zeitschritt t des markov'schen Entscheidungsprozesses. Insbesondere bei episodischen Tasks kann eine Aktualisierung der Parameter auch erst zum Ende der Episode erfolgen. Es werden also solche Aktionen wahrscheinlicher, die eine höhere Belohnung zur Folge haben. Voraussetzung ist, dass π_θ differenzierbar ist.[41]$^{S.321f.}$ Die Wahrscheinlichkeit für eine Aktion kann durch die Normalisierung der Werte berechnet werden.[41]$^{S.322}$ Auf diesem Ansatz basierende Verfahren werden Policy Gradient Verfahren genannt.[41]$^{S.321}$

Das Hauptproblem bei Policy Gradient Verfahren ist die Berechnung einer guten Abschätzung des Gradienten $\nabla_\theta \mathbb{J}(\theta)$. Die Berechnung hängt bei einem Startzustand s_0 von den zukünftigen Belohnungen ab. Diese wiederum sind abhängig von der Verteilung der Folgezustände und der Verteilung der Auswahl der Aktionen.[41]$^{S.324}$ Die Verteilung der Folgezustände wird bestimmt durch die Transitionsfunktion des Modells. Das Modell ist aber oft unbekannt oder zu komplex, um entsprechende Verteilungen zu bestimmen.[41]$^{S.66}$

Eine Lösung bietet das von Sutton et al. (2000) formulierte Policy Gradient Theorem:[42]

$$\nabla \mathbb{J}(\theta) \propto \sum_{s \in S} \mu(s) \sum_{a \in A} q_\pi(s,a) \nabla_\theta \pi(a|s,\theta). \tag{3.9}$$

Das Symbol $\propto$ meint das Verhältnis „proportional zu". μ bezeichnet die stationäre Verteilung der Zustände unter der Strategie π. Intuitiv berechnet $q_\pi(s,a)\nabla_\theta \pi(a|s,\theta)$ gegeben $s \in S, a \in A$ einen Vektor, der als Inkrement auf θ Auswahl der Aktion a umso wahrscheinlicher macht, je größer die erhaltene Belohnung für die Auswahl von a in s ist.[42] Für einen Zustand s werden diese Aktionen in einer Summe zu einem Vektor akkumuliert. Für die Akkumulation aller Zustände wird jeder Summand mit der Wahrscheinlichkeit des Auftretens $\mu(s)$ des zugehörigen Zustands gewichtet, auch Likelihood Ratio Methode genannt.[14] Die Proportionalität sagt aus, dass der Gradient sich von diesem Term nur um einen Faktor unterscheidet, beide zeigen also in die gleiche Richtung.[42] Dieser Faktor ließe sich auch über die Lernrate α kontrollieren.

Ein großer Vorteil von Policy Gradient Verfahren ist die Möglichkeit, stochastische Strategien zu modellieren. Mit Ansätzen, die auf value functions basieren, ist das nicht ohne weiteres möglich. Diese zielen darauf ab, mithilfe des approximierten Wertes eine deterministische Auswahl einer Aktion zu treffen. Je nach Umgebung kann eine Verhaltensstrategie außerdem weniger komplex als die qualitative Bewertung von Zuständen oder

Aktionen sein. In diesem Fall ist eine Strategiefunktion auch einfacher zu approximieren und entsprechende Methoden erzielen bessere Ergebnisse.[41]$^{S.323f.}$

3.4 REINFORCE

Williams stellt 1992 in seiner Arbeit zu gradientenbasierten Algorithmen eine Klasse von Algorithmen vor, die beim Berechnen der Abschätzung des Gradienten der Likelihood Ratio Methode folgen.[45]

Die Klassifizierung beruht auf fünf Einschränkungen beziehungsweise Eigenschaften:[45] 1) Die Tasks sind assoziativ. Darunter werden Aufgaben gefasst, die eine Abbildung von Eingaben auf Ausgaben beinhalten. Ein Beispiel dafür ist die Verteilung π_θ, anhand derer Ausgaben zufällig generiert werden. Offensichtlich ist das eine Voraussetzung für jedes Policy Gradient Verfahren. 2) Die Belohnungssignale, anhand derer der Agent lernt, werden unmittelbar zur Verfügung gestellt und nicht erst mehrere Zeitschritte später. 3) Das Verhalten des Agenten ist nicht deterministisch. Damit wird das Problem des Abwägens zwischen Exploitation und Exploration umgangen. Der Agent soll generell die vorteilhaftesten Aktionen auswählen, muss aber auch zunächst nicht vorteilhaft scheinende Handlungszweige erforschen. Im Extremfall würde er sonst sofort in einem potentiell lokalen Maximum stecken bleiben. 4) Die vorgestellten Algorithmen folgen einem Gradienten, kommen aber ohne die direkte Berechnung des Gradienten $\nabla_\theta \mathbb{J}(\theta)$ aus und speichern keine Informationen, aus denen dieser Gradient berechnet werden könnte. Sie werden daher auch als modell-frei bezeichnet, da sie keine Informationen über die Berechnung der Belohnungssignale oder die Verteilung der Zustände benötigen.[45]

Wir wollen die Voraussetzungen einmal unter die bisherigen Ausführungen zu neuronalen Netzen subsumieren: Die Strategie π wird durch ein Feedforward-Netz aus mehreren Einheiten dargestellt. Ein Lernschritt besteht aus einem Zyklus, in dem die Einheiten eine Eingabe aus der Umgebung bekommen, diese durch das Netz propagieren und die Ausgabe wieder zur Evaluierung an die Umgebung zurückschicken. Die Umgebung bestimmt anhand der Ausgabe ein Belohnungssignal, welches an alle Einheiten des Netzes gesendet wird. Diese korrigieren danach je nach verwandtem Verfahren ihre Gewichte entsprechend des Belohnungssignals. Sei x^i der Eingabevektor der i-ten Einheit des Netzwerks und y_i dessen Ausgabe. Es sei mit W die Gewichtsmatrix aller Gewichte w_{ij} im Netzwerk bezeichnet, wobei w_{ij} das Gewicht ist, welches mit dem j-ten Element des Eingabevektors der i-ten Einheit assoziert ist. Für jede Einheit i wird $g_i(\xi, w^i, x^i) = \mathbb{P}(y_i = \xi | w^i, x^i)$ definiert. g_i ist also die Dichtefunktion, die für eine Aktion ξ bei entsprechend festgelegten Parametern ihre Wahrscheinlichkeit y_i angibt. Im Folgenden wird die Konsistenz zeitlicher Abhängigkeiten von Parametern impliziert, sofern sie zusammen in einer Gleichung stehen. Es wird also davon ausgegangen, dass sie in diesem Fall Werte aus dem gleichen Zyklus t repräsentieren.

Analog zur allgemeinen Definition der Zielfunktion in Gleichung 3.7 wird die Zielfunktion mit $\mathbb{J} = \mathbb{E}(r, W)$ definiert.[41]$^{S.321}$ Die Parameter θ der Strategie aus Gleichung 3.8 sind die Gewichte des neuronalen Netzes. Zur Vereinfachung wird vorausgesetzt, dass die Eingabe und die Berechnung des Belohnungssignals durch die Umgebung stationären Verteilungen folgt, die Verteilungen also unabhängig vom Zyklus t sind und sich nicht ändern. Dann ist die Zielfunktion deterministisch und im Suchraum aller Gewichtsmatrizen W gibt es einen Punkt, für den $\mathbb{E}(r, W)$ maximal ist.[41]$^{S.322f.}$[45]

Unter diesen Voraussetzungen ist nun jeder Algorithmus ein REINFORCE Algorithmus, der die Gewichte W mit folgendem Wert inkrementiert[45]

$$\Delta w_{ij} = \alpha_{ij}(r - b_{ij})e_{ij} = \alpha_{ij}(r - b_{ij})\frac{1}{g_i}\frac{\delta g_i}{\delta w_{ij}}. \tag{3.10}$$

Dieses Inkrement findet sich zum Vergleich in allgemeiner Form in Gleichung 3.8 wieder. Analog bezeichnet α_{ij} die Lernrate, r die insgesamt erhaltene Belohnung, b_{ij} eine von y_i unabhängige Baseline und e_{ij} die sogenannte charakteristische Eignung. Letztere repräsentiert einen Wert, der eine Aussage über den Einfluss des entsprechenden Gewichts auf die Belohnung gibt. Der Name REINFORCE ist dabei ein Akronym der englischen Begriffe: „**RE**ward **I**ncrement = **N**onnegative **F**actor × **O**ffset **R**einforcement × **C**haracteristic **E**ligibility“.[45]

Im Folgenden wird gezeigt, dass die Abschätzung des Gradienten nach Gleichung 3.10 äquivalent zu einer Berechnung über das später formulierte Policy Gradient Theorem in Gleichung 3.9 ist. In Gleichung 3.9 werden die Zustände s durch $\mu_\pi(s)$ dementsprechend gewichtet, wie oft sie erwartungsgemäß beim Befolgen der Strategie π auftreten. Es gilt also

$$\begin{aligned} \nabla\mathbb{J}(\theta) &\propto \sum_{s\in S}\mu(s)\sum_{a\in A} q_\pi(s,a)\nabla_\theta\pi(a|s,\theta) \\ &= \mathbb{E}_\pi[\sum_{a\in A} q_\pi(S_t,a)\nabla_\theta\pi(a|S_t,\theta)] \end{aligned} \tag{3.11}$$

mit einem Stichprobenzustand S_t.[41]$^{S.326}$ Um analog die Summe über die möglichen Aktionen mit den Aktionen einer Stichprobe A_t über einen Erwartungswert zu ersetzen, kann die Wahrscheinlichkeit des Auftretens der Aktion a $\pi(a|S_t,\theta)$ zum Term multipliziert und dividiert werden. Damit ergibt sich[41]$^{S.327}$

$$\begin{aligned} \nabla\mathbb{J}(\theta) &= \mathbb{E}_\pi[\sum_{a\in A} q_\pi(S_t,a)\nabla_\theta\pi(a|S_t,\theta)] \\ &= \mathbb{E}_\pi[\sum_{a\in A} \pi(a|S_t,\theta)q_\pi(S_t,a)\frac{\nabla_\theta\pi(a|S_t,\theta))}{\pi(a|S_t,\theta)}] \\ &= \mathbb{E}_\pi[q_\pi(S_t,A_t)\frac{\nabla_\theta\pi(a|S_t,\theta))}{\pi(a|S_t,\theta)}]. \end{aligned} \tag{3.12}$$

Dabei ist $\mathbb{E}_\pi[q_\pi(S_t, A_t)]$ nichts anderes als die Belohnung r aus Gleichung 3.10. Substituiert man nun den Parametervektor θ aus Gleichung 3.8 mit den Gewichten des neuronalen Netzes W und betrachtet den Zustand S als Teil der Eingabe des neuronalen Netzes ergibt sich mit

$$\pi(A_t|\ S_t, \theta) = \mathbb{P}(y_i = \xi | w_i, x_i) = g_i(\xi, w_i, x_i) \tag{3.13}$$

die Gleichheit der Terme.[41]$^{S.327}$ Zu Algorithmen dieser Art stellt Williams das folgende Theorem auf:[45]

3.4.1 Theorem. *Für jeden REINFORCE Algorithmus ist das innere Produkt von* $\mathbb{E}(\Delta W|W)$ *und* $\nabla_W \mathbb{E}(r|W)$ *nicht-negativ. Weiter, wenn* $\alpha_{ij} > 0$ *für alle* i *und* j *gilt, dann ist das innere Produkt nur dann null, wenn* $\nabla_W \mathbb{E}(r|W) = 0$. *Wenn* $\alpha_{ij} = \alpha$ *von* i *und* j *unabhängig ist, dann gilt außerdem* $\mathbb{E}(\Delta W|\ W) = \alpha \nabla_W \mathbb{E}(r|W)$.

Die Aussagen dieses Theorems decken sich mit denen des später formulierten Policy Gradient Theorems.[41]$^{S.325.}$ Zum einen, wenn das innere Produkt des durchschnittlichen Inkrements und des echten Gradienten positiv ist, zeigen beide Vektoren etwa in eine gleiche Richtung. Eine positive Lernrate α_{ij} ändert daran nichts.[45] Sofern α_{ij} die Proportionalität der Inkrementierung einzelner Gewichte durch eine Abhängigkeit von i und j nicht verzerrt, kann die Aussage so verschärft werden, dass sich das durchschnittliche Inkrement und der Gradient nur um einen Faktor α unterscheiden.[45] Damit folgt die Proportionalitätsaussage des Policy Gradient Theorems.

3.5 Baselines

Das Inkrement Δw_{ij} bezieht für ein Gewicht die insgesamt erhaltene Belohnung mit ein, die von allen zukünftigen Aktionen und Zuständen abhängt. Für kleine Änderungen im Verhalten kann das Modell der Umgebung stark unterschiedliche Belohnungssignale geben. Das wird durch das Zufallselement der stochastischen Strategie verstärkt. Policy Gradient Verfahren leiden daher unter einer großen Varianz der ermittelten Belohnungen, die den Optimierungsprozess erschwert.[41]$^{S.329f.}$ Eine geschickt gewählte Baseline b_{ij} dient dabei zur Verringerung der Varianz.[41]$^{S.329f.}$ Typischerweise wird zum Beispiel der Durchschnitt der bisher von diesem Zustand aus erreichten Belohnungswerte als Baseline gewählt. Die Baseline muss von y_i unabhängig sein, damit der Erwartungswert nicht beeinflusst wird.[18]

Kapitel 4

Generative Adversarial Networks

Mit der Einführung von Generative Adversarial Networks durch Goodfellow et al. (2014) wurden im Bereich der generativen[1] Modelle große Fortschritte gemacht. Die zugrunde liegende Idee liegt in der Konzeption zweier Antagonisten, dem generativen Modell G und einem Diskriminator D.[16] In einem Minimax-Spiel versuchen sich diese im Training gegenseitig zu überlisten: G soll echt aussehende Daten erzeugen - also solche, die wie Stichproben der echten Verteilung p_{echt} wirken. Der Diskriminator lernt, die Daten des Generators $x \sim p_{unecht}$ von denen der echten Verteilung $x \sim p_{echt}$ zu unterscheiden.[16] Der Vorteil von GANs besteht darin, dass sie beliebige Verteilungen modellieren können, die durch rechnerische Methoden nur schwer zu approximieren sind.[16]

4.1 Grundlagen

Wir betrachten den Fall, dass es sich bei G und D um mehrschichtige neuronale Feedforward-Netze mit differenzierbaren Aktivierungsfunktionen handelt. Sei $G(z, \theta_g)$ eine Funktion mit Parametervektor θ_g, die Zufallswerte z auf Werte x abbildet. Die Zufallseingaben werden wir im Folgenden auch als „noise“ bezeichnen. D sei eine Funktion $D(x, \theta_d)$, die parametrisiert durch einen Vektor θ_d für eine Eingabe x ein Skalar ausgibt, d.h. die Wahrscheinlichkeit prognostiziert, dass $x \sim p_{echt}$, also x aus der echten Verteilung stammt. Damit ist $1 - D(x, \theta_d)$ also die Wahrscheinlichkeit, dass $x \sim p_{unecht}$. Für das Training beider Netze ergibt sich damit folgende Zielfunktion V mit

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{echt}}[\log(1 - D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))], \tag{4.1}$$

die G minimieren und D maximieren möchte.[16] Ziel für G ist es, einen möglichst hohen Wert $D(G(z))$ und damit einen möglichst kleinen Wert $log(1 - D(G(z))$ zu erzielen,

[1] Mit generativen Modellen sind solche Modelle gemeint, die Verteilungen $P(X)$ über eine Menge X definieren in einem Raum $\mathcal{X}$ repräsentieren.[11]

während für D der Wert dann maximal wird, wenn er die Daten $x \sim p_{data}$ und $G(z) \sim p_{unecht}$ korrekt klassifiziert.

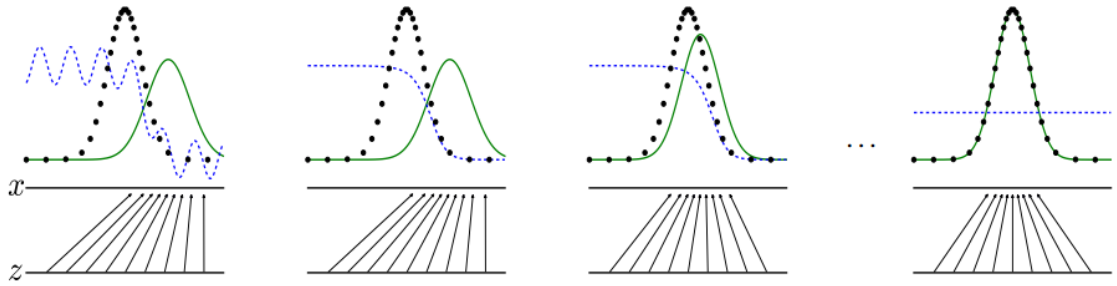

Abbildung 4.1: Quelle: Goodfellow et al. (2014) / GANs[16]. Der Bilduntertitel wurde sinngemäß aus dem Englischen übersetzt. | Beim GAN-Training wird eine klassifizierende Verteilung D (blaue, gestrichelte Linie) so trainiert, dass sie zwischen Stichproben aus der echten Verteilung p_{echt} (schwarz, gepunktete Linie) und der generativen Verteilung p_{unecht} (G, grün, durchgezogene Linie) unterscheiden kann. Die untere horizontale Linie repräsentiert den Raum, aus dem die Noise Stichproben z gezogen werden, die Pfeile stellen die Abbildung $G(z) = x$ dar. In a) bis d) ist der Prozess des GAN-Trainings dargestellt. a) Sei der Abstand zwischen p_{echt} und p_{unecht} gering und D bei der Unterscheidung nur teilweise genau. b) D wird trainiert bis $D^*_G(x) = \frac{p_{echt}(x)}{p_{echt}(x)+p_{unecht}(x)}$. c) Nach einem Update von G bildet G nun auf Punkte ab, bei denen $D(G(z))$ größer wird. d) Nach mehreren Trainingsschritten, sofern G und D geeignete Netze sind, erreichen G und D einen Punkt an dem $p_{echt} = p_{unecht}$ gilt.

Für das Training der beiden Netze werden iterativ beide abwechselnd trainiert, bis das Generator-Netz konvergiert. Goodfellow et al. schlagen einen Ansatz vor, bei dem der Diskriminator möglichst nah an einem Optimum gehalten wird, um genauere Gradienten sicherzustellen.[16] Es wechseln sich daher im vorgestellten Algorithmus 4.1 k Diskriminator-Trainingsschritte mit jeweils einem Generator-Trainingsschritt ab, bis G konvergiert und $p_{unecht} = p_{echt}$, wie in Abbildung 4.1 dargestellt. Für ein festes G konvergiert dabei D gegen $D^*_G(x) = \frac{p_{echt}(x)}{p_{echt}(x)+p_{unecht}(x)}$.[16] Intuitiv kann D natürlich nicht besser schätzen als es die Distanz zwischen $p_{echt}(x)$ und $p_{unecht}(x)$ erlaubt. Gilt beispielsweise $p_{echt}(x) = p_{unecht}(x)$, dann bleibt nur noch zu „raten“ aus welcher Verteilung ein Eingabewert x stammt und es gilt $D^*_G(x) = \frac{1}{2}$.[16]

4.2 Generative Adversarial Networks mit REINFORCE

Im Kapitel 3 wurde zum Thema „Reinforcement Learning“ besprochen, wie ein handelnder Agent aus dem Feedback seiner Umgebung lernen kann. Die Frage nach der Umgebung, und wie dieses Feedback konstruiert werden kann, ist dabei offen geblieben. Für viele Aufgaben ist es nicht trivial, Handlungen eines Agenten so zu bewerten, dass es das präzise Feedback eines Skalars als Belohnungssignal erlaubt. Oft ist selbst eine ungefähre Einordnung durch

Algorithmus 4.1 Iteratives Training von Generative Adversarial Networks | Quelle: Goodfellow et al. (2014) / GANs[16]. Der Algorithmus wurde frei aus dem Englischen übersetzt. Das Ziehen von Stichproben wird als „Samplen“ bezeichnet.

Eingabe: G: Generator

Eingabe: D: Diskriminator

Eingabe: k: Anzahl von Trainingsschritten für D

Eingabe: $p_{echt}(x)$: Trainingsdaten aus der echten Verteilung

procedure GAN-TRAINING
  for Anzahl an Trainingsschritten **do**
    for k Schritte **do**
      Sample Mini-Batch von m Noise Proben $\{z_1, \dots z_m\}$
      Sample Mini-Batch von m Proben $\{x_1, \dots x_m\}$ aus p_{echt}
      Aktualisiere D anhand seines Gradienten:

$$-\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x_i) + \log(1 - D(G(z_i)))]$$

    end for
    Sample Mini-Batch von m Noise Proben $\{z_1, \dots z_m\}$
    Aktualisiere G anhand seines Gradienten:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z_i)))$$

  end for
end procedure

einen menschlichen Betrachter mit Fachwissen schwierig oder gar unmöglich.[41]$^{S.66}$ Gleiches gilt somit auch für die Modellierung einer Umgebung „von Hand“. Das Konzept des Gegenspielers in Form eines Diskriminators löst dieses Problem mit Hilfe des maschinellen Lernens, in dem also das Modell der Umgebung durch den Diskriminator approximiert wird.

Es ergeben sich dabei für die Aufgabenstellung aus der Verwendung von GANs aber zwei weitere Probleme: Erstens funktioniert die GAN-Methode nur mit der Komposition zweier differenzierbarer Funktionen.[48] Da wir diskrete Daten erzeugen, ist die generative Funktion nicht differenzierbar. An diesem Punkt wird an das Kapitel zum Reinforcement Learning angeknüpft. Statt direkt diskrete Daten zu generieren, wird das Generieren einer Formel als sequentieller Entscheidungsprozess eines Agenten verstanden.[2][3] Ein Zustand s wird durch die bisher generierte Teilformel repräsentiert, die möglichen Aktionen a sind die Zeichen eines Alphabets, aus denen der Agent wählen kann. Um die Frage der Differenzierbarkeit zu lösen, modelliert G eine stochastische Strategie, der der Agent folgt. Es werden also nicht direkt Folgezeichen, sondern eine Verteilung über das Alphabet generiert, bedingt durch die bisher gegebene Teilsequenz. Für das Training kann gemäß Verteilung eine Aktion durch ein Zufallsexperiment ausgewählt werden. Die so generierten Formeln können vom Diskriminator bewertet und der Generator mit dem ausgegebenen Belohungssignal mithilfe des REINFORCE Algorithmus aus Gleichung 3.10 trainiert werden.[48]

Zweitens kann aber der Diskriminator nicht jede Aktion des Agenten sinnvoll bewerten. Die Handlungen des Agenten entsprechen den Schritten, in denen der Generator einzelne Zeichen produziert. Wird der Diskriminator auf einem Trainingsdatensatz von Bildern vollständiger mathematischer Formeln trainiert, lassen sich im Gegenzug auch nur Bilder vollständiger mathematischer Formeln bewerten. Damit kann der Diskriminator für jeden Schritt, in dem der Generator keine Formel vervollständigt, kein Belohnungssignal beisteuern.[48]

Eine Lösung bieten sogenannte Monte-Carlo-Simulationen.[48] Eine Monte-Carlo-Simu -lation bezeichnet die Approximation eines unbekannten Wertes durch eine hohe Anzahl gleichartiger Zufallsexperimente.[41]$^{S.91}$ Gesucht ist das arithmetische Mittel der Summe der Belohnungen $Q_\pi(s, a)$, die von einem Zustand s mit einer Aktion a beim Befolgen einer Strategie π gesammelt werden können. Beim Zustand s handelt es sich um eine bisher generierte Teilfolge von Zeichen $y_1, \ldots, y_{t-1}$ für einen Zeitschritt t, a ist das in Zeitschritt t generierte Zeichen und die Strategie π wird durch den Generator G repräsentiert. Gesucht ist ein Mittelwert, weil $Q_\pi(s, a)$ von der stochastischen Strategie π abhängt.[41]$^{S.66}$ Der Belohnungswert des Diskriminators $D(Y)$ für eine vervollständigte Zeichenfolge $Y = y_1, \ldots, y_T$ ist hingegen beim Training des Generators fest.[16] Für eine maximale Länge T wird zur Mittelung eines Wertes im Schritt t $Q_\pi(s_t, a)$

$$Q(y_{1:t-1}, a) = \begin{cases} D(y_{1:t}))) & \text{falls } t = T \\ Q(y_{1:t}), G(y_{1:t})) & \text{falls } t < T \end{cases} \tag{4.2}$$

berechnet (Gleichung 3.6), wobei die Parametervektoren θ_d und θ_g implizit sind und $y_{1:t-1}$ konkateniert mit a die Folge $y_{1:t}$ ergibt.[48] So kann jeder Teilsequenz ein Belohnungswert zugewiesen werden.

An dieser Stelle ist es wichtig anzumerken, dass wir nun zwei Verteilungen betrachten. In der Einleitung wurde mit p_{echt} eine Verteilung über Sequenzen bezeichnet, die modelliert werden sollte. Mit der Annahme eines sequentiellen Prozesses kann das allerdings nicht die Verteilung sein, die G modellieren soll. Wenn also in der Folge von p_{echt} geschrieben wird, ist eine Verteilung über die Menge der Zeichen gemeint.

4.3 Sequence Generative Adversarial Networks

Den oben beschriebenen und in Abbildung 4.2 visualisierten Ansatz nutzen auch Yu et al. (2017) in ihrer Arbeit zu SeqGANs.[48] Sie verwenden dabei Algorithmus 4.2. Grundsätzlich scheint das Vorgehen für Sequenzen gut übertragbar für jegliche Art von Sequenzen, so auch grammatisch strukturierte Sequenzen wie die der Syntaxbäume.

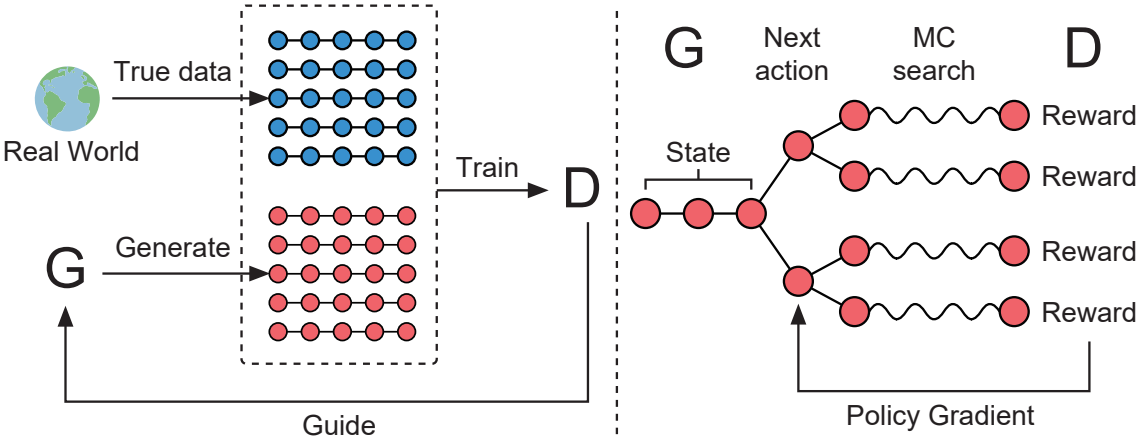

Abbildung 4.2: Quelle: Yu et al. (2017) / SeqGANs[48]. Der Bildtext wurde sinngemäß aus dem Englischen übersetzt. Links: D wird mit Daten aus der echten Verteilung gegen die Daten aus G trainiert. Rechts: Für eine Teilsequenz wird der Wert eines generierten Folgezeichens durch mehrere Simulationen des Generationsprozesses bestimmt.

Im Unterschied zu klassischen GANs akkumuliert der Algorithmus alle Belohnungen einer Episode, das heißt einer Sequenz auf, bevor die Parameter G_θ aktualisiert werden. Für jede Teilsequenz wird also mit Hilfe der Monte-Carlo-Simulation ein Belohnungswert gemittelt, der alle noch zu erzielenden Belohnungen aufaddiert.[48] Die „roll-out policy“ $G_{rollout}$ ist die Strategie, der bei der Monte-Carlo-Simulation zum Fortsetzen der Sequenzen gefolgt wird. Zur Verbesserung der Performanz kann es Sinn machen, ein einfacheres

Algorithmus 4.2 Sequence Generative Adversarial Networks, Quelle: Yu et al. (2017) / SeqGANs [48]. Sinngemäß aus dem Englischen übersetzt.

Eingabe: G_θ: Generator policy
Eingabe: $G_{rollout}$: Generator roll-out policy
Eingabe: D: Diskriminator
Eingabe: S_{echt}: Datensatz von Sequenzen
Eingabe: g: Anzahl an Generator-Trainingsschritten
Eingabe: d: Anzahl an Diskriminator-Trainingsschritten
Eingabe: k: Anzahl von Diskriminator-Trainingsepochen

procedure SeqGAN-Training
    Initialisiere G_θ und D mit Zufallsgewichten
    Trainiere G_θ mithilfe von S_{echt}
    $G_{rollout} \leftarrow G_\theta$
    Generiere künstliche Daten S_{unecht} mit G_θ
    Trainiere D mit S_{unecht} und S_{echt}
    while SeqGAN nicht konvergiert **do**
        for g Schritte **do**
            Generiere eine Sequenz $Y_{1:T} = (y_1, \ldots, y_T) \sim G_\theta$
            for t $\in 1 :$ T **do**
                Berechne $Q(a = y_t, s = Y_{1:t-1})$ mit Gleichung 4.2
            end for
            Aktualisiere G mit dem Policy Gradient Verfahren in Gleichung 3.10
        end for
        for d Schritte **do**
            Generiere S_{unecht} mit G_θ
            Erstelle Datensatz S aus S_{unecht} und S_{echt}
            Trainiere D mit S für k Epochen gemäß Gleichung 4.1
        end for
        $G_{rollout} \leftarrow G_\theta$
    end while
end procedure

Modell für $G_{rollout}$ zu benutzen. Wir folgen Yu et al. und setzen $G_\theta = G_{rollout}$.[48] Vor dem eigentlichen Training der beiden Netze wird ein Pretraining, d.h. ein vorgelagertes Training, durchgeführt. Die Aufgabe wird dabei als Regressionsproblem interpretiert. D.h. für die Eingabe einer Teilsequenz soll G ein Folgezeichen vorhersagen.[15]$^{S.99}$[48] Dafür wird als Fehlerfunktion die sogenannte Kreuzentropie verwendet, ein Abstandsmaß für Verteilungen.[48]

4.3.1 Definition. Sei X eine Zufallsvariable mit Zielmenge Ω, die gemäß einer Verteilung p_a verteilt ist, $s_1, \ldots, s_n \in \Omega$ Realisierungen von X und sei p_b eine Approximation der Verteilung p_a auf der gleichen Menge, dann lässt sich die Kreuzentropie wie folgt schätzen:[15]$^{S.73}$

$$H(p_a, p_b, n) = -\frac{1}{n} \sum_{i=1}^{n} \log\ p_b(x_i). \tag{4.3}$$

Im vorliegenden Fall ist Ω eine Menge von Zeichen eines Alphabets Σ, $p_a = p_{echt}$ und $p_b = p_{unecht}$. Die Verteilung ist bedingt durch die vorangehende Teilsequenz. Zur Verdeutlichung lässt sich die Gleichung also zu $-\frac{1}{n} \sum_{i=1}^{n} \log\ p_g(x_i|x_1, \ldots, x_{i-1})$ umschreiben. Die Arbeit zu SeqGANs behandelt natürlichsprachliche Zeichensequenzen, für diese liegen Datensätze vor und damit auch Stichproben für die nach Teilsequenzen bedingte Verteilung p_{echt}.[48] Für die vorliegende Aufgabenstellung gestaltet sich das anders, es liegen lediglich Datensätze zu Daten im Bildformat vor. Formeln und Bilder stellen offensichtlich keine im Sinne der Kreuzentropie verwendbare Stichprobe S dar, sie können nicht als Eingabe für G dienen, um $\log\ p_g(x_i|x_1, \ldots, x_{i-1})$ zu berechnen. Ein Pretraining ist deshalb nicht möglich. In der Arbeit zu SeqGANs kommen auf 100 Iterationen Pretraining lediglich 50 Iterationen GAN-Training.[48] Che et al. (2017) stellen fest, dass SeqGANs nicht ohne ein sorgfältiges Pretraining auskommen.[6] Dieser Faktor stellt also für diese Arbeit eine große Herausforderung dar.

Um den Algorithmus für unsere Zwecke anzupassen, muss außerdem eine Übersetzung von Sequenzen zu Bäumen bzw. zu Bildern zwischengeschaltet werden, bevor Bewertungsschritte durch den Diskriminator erfolgen können. Für das genauere Vorgehen dabei sei auf das Kapitel 5 verwiesen. Eine angepasste Version des Algorithmus findet sich in Kapitel 8 in Algorithmus 8.1.

4.4 Weitere verwandte Arbeiten

Da sich SeqGANs im Vergleich zu anderen Methoden als performant erwiesen haben, adaptiert diese Arbeit die SeqGAN-Methode.[48] Dennoch sollen einige zu SeqGANs vergleichbare Arbeiten erwähnt werden, die im Bereich der Sequenzgenerierung einige Aufmerksamkeit erfahren haben.

Maximum-Likelihood Augmented GAN (MaliGAN) Che et al. (2017) setzen Importance Sampling bei der Gradientenberechnung zur Optimierung von G ein.[6] Importance Sampling ist eine Technik zur Verringerung der Varianz, zum Beispiel beim Schätzen von Mittelwerten, bei der Stichproben nach einem definierten Gewicht in die Mittelwertbildung eingehen. Einen äquivalenten Ansatz verfolgen Hjelm et al. (2017) mit **Boundary-Seeking GANs (BGANs)**.[20] In MaliGANs wird als Belohnung $\frac{D(Y)}{1-D(Y)}$ statt $D(Y)$ verwendet, dabei werden hohe Belohnungen für $D(Y) \to 1$ also viel stärker gewichtet als kleine Belohnungen. Die Aktualisierung bestimmt sich insgesamt über

$$(\frac{r_D(Y_{1:T})}{\sum_{i=1}^{m} r_D(Y^i_{1:T})} - b) \sum_{t=1}^{T} \nabla_{\theta_G} log G_\theta(y_i|Y_{1:t-1}),$$

für m Sequenzen eines Mini-Batches und $r_D = \frac{D(Y)}{1-D(Y)}$.[20] Die Normalisierung über den Mini-Batch gewichtet die Aktualisierung zu Gunsten der Sequenzen Y^i mit den höchsten Belohnungen $D(Y^i)$.[20] In Experimenten zur Sequenzgenerierung hat SeqGAN jedoch zu Beginn des Trainings bessere Ergebnisse erzielt.[23]

MaskGAN Fedus et al. (2018) schlagen eine actor critic[2] Variante vor, bei der zum Training von G die Sequenzgenerierung als Vervollständigung echter Teilsequenzen durch künstliche Zeichen verstanden wird.[12] Dabei werden Zeichen in Sequenzen aus p_{echt} durch Lücken ersetzt, die von G durch künstlich generierte Zeichen aufgefüllt werden. Dieser Ansatz erfordert das Vorliegen von Sequenzen aus p_{echt}, er ist also hier nicht anwendbar.

Reward for Every Generation Step (REGS) In ihrer Arbeit zur Generierung von Dialogen, wobei G die Rolle des Antwortenden einnimmt, verwenden Li et al. (2017) ein Verfahren, dass erstens für jedes generierte Zeichen eine Belohnung durch Vervollständigung der Sequenz berechnet.[12] Im Vergleich zum regulären REINFORCE Algorithmus ist das insofern eine Verbesserung, als das dort zum Training nur die einmal erhaltene Belohnung am Ende der generierten Sequenz genutzt wird. Zweitens Wird D so trainiert, dass auch unvollständige Sequenzen bewertet werden können. Punkt 1 entspricht einer verkürzten Variante der SeqGAN Monte Carlo Simulationen, da bei SeqGANs die Belohung aus n Vervollständigungen gemittelt wird.[48]

[2] siehe Kapitel 3, Abschnitt 3

Kapitel 5

Alphabet und Grammatik

5.1 Alphabet

Wir wollen zunächst die Problemstellung unter einige formale Definitionen subsumieren.

5.1.1 Definition. Ein **Alphabet** ist definiert als eine endliche nicht-leere Menge von unterscheidbaren Zeichen.[24]

5.1.2 Definition. Ein **Wort** über einem Alphabet Σ ist eine endliche Sequenz von Zeichen aus Σ.[24]

5.1.3 Definition. Eine **Sprache** L über einem Alphabet Σ ist definiert als eine Menge von Wörtern über Σ.[24]

Wir betrachten die Menge mathematischer Formeln, deren Verteilung modelliert werden soll. Diese sind Sequenzen aus mathematischen Operatoren wie $+$, $-$, $\prod$, Variablen wie a, α und Zahlen. Bei den Operatoren und Variablen handelt es sich um die Zeichen des Alphabets Σ. Formal handelt es sich also bei mathematischen Formeln um Wörter aus einer formalen Sprache L über ein Alphabet Σ. Zu Anfang wurde das Ziel gesetzt, grammatische Korrektheit mathematischer Sequenzen sicherzustellen. An dieser Stelle soll es aber zunächst nur um die Definition einer Menge von Zeichen gehen. Im Kontext der GAN-Umgebung handelt es sich um die Zielmenge der Verteilung p_{echt}[1], da wir keine ganzen Sequenzen, sondern nur einzelne Zeichen abhängig von einer Teilsequenz generieren.

Die Menge der Zeichen, die sinnvollerweise in einer mathematischen Formel aus jedwedem Teilgebiet der Mathematik vorkommen könnten, kann kaum definiert werden. Die Anzahl der Knoten im Input-Layer eines neuronalen Netzes ist fix. Abhängig von der Kodierung der Zeichen als Eingabe für den Generator, steigt also mit zunehmender Mächtigkeit der Zeichenmenge auch die Dimensionalität des Netzes und damit auch die Anzahl

[1] siehe Kapitel 4, Abschnitt 4

der durchzuführenden Berechnungen. Außerdem erhöht sich mit zunehmender Dimensionalität auch die Anzahl der zu optimierenden Parameter. Aus technischen Gründen macht es deshalb Sinn, die Anzahl an Zeichen möglichst stark zu beschränken. Schließlich hängt es aber auch von den echten Formeln ab, auf denen der Diskriminator trainiert wird. Enthalten sie viele Zeichen, die im definierten Alphabet nicht enthalten sind und vice versa, unterscheiden sich die Verteilungen p_{unecht} und p_{echt} schon in ihren Zielmengen. Der D hat dann leichtes Spiel, wenn er anhand von Zeichen unterscheidet, die G gar nicht berücksichtigen kann. Insgesamt muss also ein Mittelmaß gefunden werden, das größtmögliche Flexibilität in der Modellierung von Sequenzen erlaubt, aber die Anzahl der Zeichen auf ein geeignetes Maß beschränkt.

Ein weiterer Punkt, der Beachtung verdient, ist die Unterscheidung zwischen Semantik von Zeichen und ihrer Syntax. Beispielsweise kann der Punktoperator eine Multiplikation von Zahlen $1 \cdot 1$, aber genauso ein Skalarprodukt $\overrightarrow{v} \cdot 2$ oder eine Matrixmultiplikation $M \cdot M^T$ beschreiben. Es muss nicht einmal eindeutig sein, welche Operation gemeint ist, wenn das nur aus dem Kontext von Variablen hervorgeht, die außerhalb der Formel definiert wurden. Genauso kann ein Symbol f eine Funktionsapplikation $f(x)$ aber auch eine Konstante $f = 1$ meinen. Versteht man formal die Konstante als nullstellige Funktion, so geht doch zumindest aus dem Bezeichner f seine Stelligkeit oder sein Definitionsbereich nicht immer hervor, zum Beispiel in der Funktionskomposition $f \circ g$. Bei der regelbasierten Synthese von Syntaxbäumen, auf denen Äquivalenzumformungen durchgeführt werden sollen, ist die Semantik ausschlaggebend. Für die Trainingsdaten liegt aber nur eine visuelle Repräsentation vor. Modelliert werden kann deshalb allein die Visualisierung der Formeln, nicht ihre Semantik. Für die Definition eines Alphabet heißt das konkret, dass es sich bei der betrachteten Grundmenge zunächst um die Menge der visuellen Zeichen und nicht die der mathematischen Operatoren handelt. Statt der verschiedenen Multiplikationsoperatoren wird also nur ein Zeichen aufgenommen, nämlich $\cdot$. Eine semantische Interpretation kann nur retrospektiv erfolgen und soll nicht Teil dieser Arbeit sein. Das bedeutet nicht, dass eine Form von grammatischer Struktur nicht garantiert werden kann, denn Information über die Stelligkeit der Operatoren bleibt erhalten. Viel mehr bedeutet der Verzicht auf semantische Interpretation die Mehrdeutigkeit von generierten Sequenzen.

Eine vorläufige Definition des Alphabets zu dieser Arbeit findet sich in Abbildung 5.1, sie wird im nächsten Abschnitt noch etwas modifiziert. Nicht aufgezählt sind englische und griechische Buchstaben, sowie die Zahlen 0 bis 9. Insbesondere enthält das Alphabet keine Kommazahlen. Bei nur einer Nachkommastelle kommt man bereits auf 100 verschiedene Zeichen $0.0, 0.1, \ldots 9.9$. Ein rekursiver . Operator, über den sich Kommazahlen konstruieren lassen, wäre unverhältnismäßig kompliziert und kaum intuitiv. Die verwendeten Zeichen richten sich dabei nach dem zum Training von D verwendeten Datensatz, der Formeln aus dem Bereich Maschinelles Lernen enthält.

$\sqrt{x}$	!	max	min	argmax	argmin	x^{-1}	sin	cos	tan
sinh	cosh	tanh	x^T	x'	|x|	||x||	$\mathbb{E}$	$\mathbb{P}$	+
-	$\cdot$	$\times$	$\frac{x}{y}$	mod	x^y	$\frac{\partial x}{\partial y}$	$\sum$	$\prod$	$\int$
$=$	$<$	$>$	$\leq$	$\geq$	$\subset$	$\subseteq$	$\cup$	$\cap$	
$\in$	x(y)	(x)	∞	$\propto$	$\emptyset$	- (Negation)			

Abbildung 5.1: Ein Alphabet ohne Zahlen und englische oder griechische Buchstaben. Variablen x,y sind nur zur Visualisierungszwecken verwendet. Es werden weitere Elemente für hoch- und tiefgestellte Ausdrücke hinzugefügt, das Alphabet wird im nächsten Abschnitt noch einmal modifiziert.

5.2 Grammatik, Übersetzung und Kodierung

Schließlich muss auch eine Übersetzung von Zeichenfolgen zu Syntaxbäumen erfolgen können. Dafür muss zunächst definiert werden, wie die Sequenzen und ihre entsprechenden Syntaxbäume konstruiert sein sollen. Das ist abhängig von der Sprache mathematischer Formeln. Eine Sprache über einem Alphabet kann über ihre Menge von Wörtern definiert werden, aber auch durch eine Menge von Konstruktionsregeln über ihrem Alphabet. Eine solche Menge von Regeln wird Grammatik genannt.

5.2.1 Definition. Eine **Grammatik** ist ein Tupel $G = (\Sigma, V, P, S)$, für das gilt:[8]

- Σ ist eine endliche Menge von Symbolen, auch Terminalsymbole genannt.
- V Ist eine endliche Menge von Nichtterminalsymbolen, disjunkt von Σ und disjunkt mit der Menge der Wörter, die über G konstruiert werden können.
- P ist eine Menge von Konstruktionsregeln der Form $(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$, wobei $*$ die kleenesche Hülle notiert.
- S ist ein Nichtterminalsymbol, auch Startsymbol genannt.

5.2.2 Definition. Sei L eine Sprache über einem Alphabet Σ. Sei $L^0 = \{\epsilon\}$ und $L^i = \{xy | x \in L, y \in L^{i-1}\}$ für $i \geq 1$. Die **kleenesche Hülle** L^* von L ist definiert als $L^* = \bigcup_{i \geq 0} L^i$.[24]

Die kleenesche Hülle einer Sprache L ist also die Menge aller Wörter, die sich durch die Verknüpfung aller Wörter in L ergibt, wobei das leere Wort inbegriffen ist. Intuitiv sind Konstruktionsregeln P solche Regeln, anhand derer Folgen von Nichtterminalen und Terminalen zu anderen Folgen von Nichtterminalen und Terminalen abgeleitet werden können, bis die Folge nur noch Terminale enthält und nicht weiter abgeleitet werden kann. Damit ist die Folge der Terminale ein Wort der Sprache L. Seien Σ, N und P nun wie folgt definiert:

- Die Operatoren, die durch die Zeichen in Σ aus Abbildung 5.1 repräsentiert werden, werden in nullstellige, ein-, zwei- und dreistellige Operatoren O_0, O_1, O_2, O_3 unterschieden. Die Mengen müssen nicht disjunkt sein. Nullstellige Operatoren sind alle englischen und griechischen Buchstaben, die Zahlen 0-9, ∞ und $\emptyset$. Einstellige Operatoren sind $-$, $\surd$, !, max, min, x^{-1}, x^T, x', sin, cos, tan, $sinh$, $cosh$, $tanh$, $|x|$, $||x||$, $\mathbb{E}$, $\mathbb{P}$, (x). Einstellige Operatoren wie $argmax$ können auch tiefgestellte Ausdrücke wie $argmax_{x \in X}(y)$ oder $\mathbb{E}_{x \sim X}(y)$ beinhalten. Insofern werden sie auch den zweistelligen Operatoren zugerechnet. Zweistellige Operatoren sind max, min, $argmax$, $argmin$, $\mathbb{E}$, $\mathbb{P}$, $+$, $-$, $\cdot$, $\times$, $\frac{x}{y}$, mod, x^y, $\frac{\partial x}{\partial y}$, $=$, $<$, $>$, $\leq$, $\neq$, $\geq$, $\subset$, $\subseteq$, $\cap$, $\cup$, $\in$, $x(y)$, $\propto$. Dreistellige Operatoren sind $\sum$, $\int$, $\prod$. Die Indizierung und die obere Schranke, zum Beispiel $i = 1$ und n für $\sum_{i=1}^{n}$ werden dabei als Operanden gewertet.

- Sei $\Sigma = O_0 \cup O_1 \cup O_2 \cup O_3$, so dass alle Symbole für Operatoren gleichen Typs und verschiedener Stelligkeit unterscheidbar sind.

- Sei $N = \{Ausdruck\}$.

- Sei $P = \{S \rightarrow Ausdruck\}$.

$$P = P \cup \{Ausdruck \rightarrow o\}, \forall o \in O_0.$$

$$P = P \cup \{Ausdruck \rightarrow o\ Ausdruck\}, \forall o_N \in O_1.$$

$$P = P \cup \{Ausdruck \rightarrow o\ Ausdruck\ Ausdruck\}, \forall o_N \in O_2.$$

$$P = P \cup \{Ausdruck \rightarrow o\ Ausdruck\ Ausdruck\ Ausdruck\}, \forall o_N \in O_3.$$

Wir folgen sowohl bei Erläuterungen, bei der Grammatik, als auch bei der Kodierung von Sequenzen der polnischen Notation, bei der zunächst die Operatoren und darauffolgend die Operanden niedergeschrieben werden, so zum Beispiel $+xy$ statt $x + y$.[4] Die oben definierte Menge P vereinfacht die Übersetzung von Sequenzen in Bäume, da Operatoren eine fest definierte Stelligkeit haben. Eine Addition muss beispielsweise mit $+x+yz$ erfolgen statt mit $+xyz$. Der Vorteil ist, dass die Operanden für einen Operator leicht abzulesen sind. Anders könnten sich Uneindeutigkeiten bei der Assozation ergeben, zum Beispiel, mit $/ab$ für $\frac{a}{b}$, könnte $/a/bcd$ sowohl mit $/a(/bc)d$, als auch mit $/a(/bcd)$ ausgewertet werden. Da Division nicht assoziativ ist, muss aber $/a(/bc)d = /a(/bcd)$ nicht gelten. Kenntnis von Syntaxbäumen wurde bisher vorausgesetzt. Sie sollen an dieser Stelle dennoch für den vorliegenden Fall angepasst und definiert werden.

5.2.3 Definition. Ein **Syntaxbaum** zu einem Wort $w = o_1, \ldots, o_n$ aus der über die Grammatik G definierte Sprache ist ein Tupel $B = (V, E)$, für das gilt, dass $V = V_{alg}$ und $E = E_{alg}$ für die Ausgabe des Algorithmus 5.1 (w', V_{alg}, E_{alg}) bei Eingabe $(w, V = \emptyset, E = \emptyset)$.

Algorithmus 5.1 Algorithmus zur Übersetzung von Wörtern in Syntaxbäume

Eingabe: $O = \{O_0, O_1, O_2, O_3\}$: Terminalmenge
Eingabe: w: Wort
Eingabe: (V, E): Menge der Kanten und Knoten, bei Aufruf $(\emptyset, \emptyset)$

```
1: procedure SYNTAX-BAUM(w, V, E)
2:     o_erstes, w_rest ← erstesZeichen(w)        ▷ w_rest kann ein leeres Wort ϵ sein.
3:     if w_rest == ϵ then                        ▷ Syntaxbaum für unvollständige Sequenz
4:         i ← 0
5:     else
6:         i ← j, so dass o_erstes ∈ O_j gilt     ▷ Anzahl an hinzuzufügenden Kanten
7:     end if
8:     for j = 1, j ≤ i, j++ do                   ▷ Für i = 0 werden keine Kanten hinzugefügt.
9:         o_naechste, w_rest ← erstesZeichen(rest)
10:        E ← E ∪ (o_erstes, o_naechste)
11:        (w_rest, V, E) ← SYNTAX-BAUM(w_rest, V, E)
12:    end for
13:    return (w_rest, V, E)
14: end procedure
```

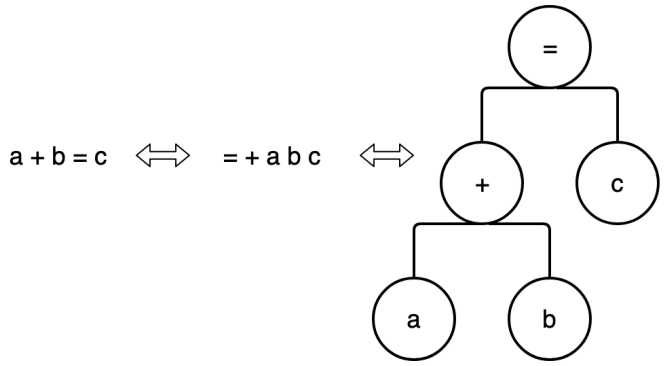

Abbildung 5.2: Äquivalenz von Bäumen und Sequenzen gemäß den vorgestellten Algorithmen. Links: Zielformel, Mitte: Postfixnotation, Rechts: Syntaxbaum.

Typischerweise enthalten innere Knoten von Syntaxbäumen Nichtterminale und Blätter die Terminale.[24] Da in diesem Fall aber rekursive Strukturen, nämlich die mehrstelligen Operatoren ebenfalls durch Zeichen repräsentiert und nicht substituiert, sondern nur ergänzt werden, ist es vorteilhaft ebenfalls Terminale als innere Knoten zu wählen. Damit kann die Übersetzung zu einer Sequenz durch ein simples Verfahren, dargestellt in Algorithmus 5.2, erfolgen. Beide Algorithmen können auch mit unfertigen Sequenzen umgehen, wenn also zu einem Operator nicht ausreichend Operanden in der Sequenz vorhanden sind. In Abbildung 5.2 finden sich Beispiele für die Anwendung beider Algorithmen.

Algorithmus 5.2 Algorithmus zur Übersetzung von Syntaxbäumen in Wörter

Eingabe: w: Wort, bei Aufruf das leere Wort ϵ
Eingabe: (V, E): Menge der Kanten und Knoten des Syntaxbaums
Eingabe: v_{wurzel}: Der Wurzelknoten des Syntaxbaums

```
1: procedure SEQUENZ(w, V, E, v_wurzel)
2:     w ← w + zeichenVon(v_wurzel)
3:     for (v, v') ∈ E do
4:         if v == v_wurzel then                ▷ v' ist ein Operand von v
5:             w ← Sequenz(w, V, E, v')         ▷ und seine Zeichen werden angehängt
6:         end if
7:     end for
8:     return w
9: end procedure
```

Eine Zeichenfolge muss außerdem als Eingabe für den Generator, und die Ausgabe des Generators als Verteilung über Folgezeichen interpretiert werden können. Zur Kodierung von kategorischen Daten wird typischerweise eine 1-aus-n-Kodierung verwendet.

5.2.4 Definition. Seien n kategorische Werte beliebig sortiert, dann ist für den i-ten Wert in der Sortierung ein Vektor $v = (v_1, \ldots, v_n)$ mit $v_i = 1$ und $v_j = 0$ für $0 \leq j \leq n$, $j \neq i$ eine 1-aus-n-Kodierung.[19]

Sei also das Alphabet $\Sigma = \{a, b\}$ würde a mit 10 und b mit 01 kodiert.

Kapitel 6

Recurrent Neural Networks

Im Kapitel 5 wurde besprochen, wie mit Hilfe einer 1-aus-n-Kodierung eine Sequenz in eine Kodierung $v = (v_1, \ldots, v_n)$ transformiert werden kann. Diese kann als Eingabe für das Eingabe-Layer des Generator-Netzes mit n Knoten dienen. Da wir auch die Parameter des Eingabe-Layers und der zugehörigen Kanten optimieren, ist diese Anzahl n für das Training fest. Das bedeutet, dass die Eingabelänge über den Verlauf des Trainings statisch ist. Folgen wir der Idee, den Erzeugungsprozess als sequentielle Entscheidung eines Agenten zu verstehen, erfordert die Generierung einer Sequenz der Länge n, zunächst Teilsequenzen der Längen 1 bis $n-1$ zu generieren. Eine Möglichkeit wäre, die Länge des Eingabevektors auf ein genügend großes n zu fixieren und mit einer Kodierung für ein leeres Zeichen $\{\epsilon\}$ aufzufüllen. Dann könnte für G ein einfaches Feedforward-Netz genutzt werden, wie es in Kapitel 2 vorgestellt wurde. Bezüglich der erzeugbaren Sequenzen stellt das eine starke Beschränkung dar, da nicht beliebig lange Sequenzen generiert werden können.

6.1 Grundlagen zu rekurrenten Netzen

Bei der Sequenzgenerierung werden deshalb typischerweise rekurrente neuronale Netze verwendet.[15]$^{S.367f.}$ Die Eingabe erfolgt sukzessiv Zeichen für Zeichen.[44] Durch zyklische Abhängigkeiten speichert das Netz durch Zeitschritte t hinweg einen Zustand h_t. Der Zustand ändert sich abhängig von den eingegebenen Zeichen.[15]$^{S.370}$ Er fungiert damit als Gedächtnis des Netzes. Damit ist auch das Problem der Sequenzlänge gelöst. Sukzessiv kann Zeichen für Zeichen eine beliebig lange Sequenz eingegeben werden. Die Dimension der Eingabe kann sich nach der Kodierung eines einzelnen Zeichens und damit der Anzahl an Zeichen richten, die bereits bei Definition der Sprache bekannt sind.

Einen einfachen Aufbau eines rekurrenten Netzes zeigt Abbildung 6.1. Für einen Knoten v eines rekurrenten Layers wird eine Kante (v, v) dem Netz hinzugefügt. Sei v der i-te Knoten eines Layers in einem vorwärtsgerichteten Netz mit m eingehenden und n ausgehenden Kanten. Für ein Eingabevektor $x = (x_1, \ldots, x_m)$ kommt also ein weiteres Eingabeelement

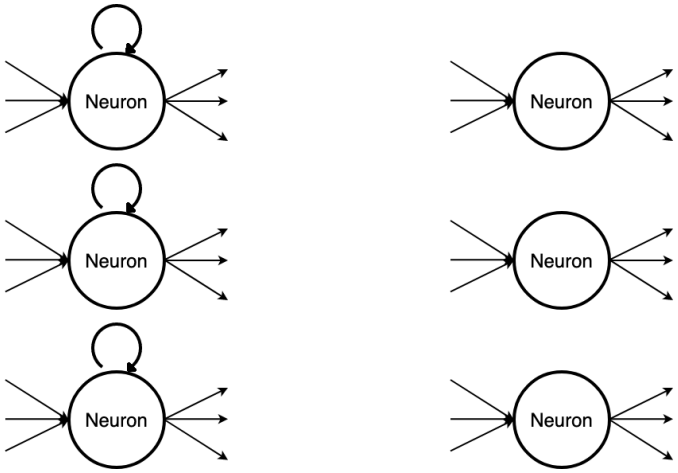

Abbildung 6.1: Links: Ein rekurrentes Layer mit drei Neuronen - Rechts: Ein Layer aus einem zykelfreien Feedforward-Netz zum Vergleich.

x_{m+1} hinzu, sodass der Ausgabewert in Zeitschritt t mit $y_t = \varphi(\Sigma_{j=0}^{m+1} w_{ij} x_j)$ für eine Aktivierungsfunktion φ berechnet wird.[15]$^{S.383}$ In Zeitschritt $t+1$ gilt dann $x_{m+1} = y_t$.

6.2 Backpropagation Through Time

Wegen der zyklischen Abhängigkeiten sind die lernbaren Parameter nicht nur von der Eingabe, sondern auch von der Zeit abhängig. Das macht es schwierig rekurrente Netze formal zu betrachten. Zur Vereinfachung kann ein rekurrentes Netz „abgerollt“ werden. Dieses Konzept wird auch „unfolding through time“ genannt.[15]$^{S.36ff.}$ Für ein Netz N, mit dessen Hilfe zyklisch eine Ausgabe über mehrere Zeitschritte erzeugt wird, kann ein Netz N^* wie in Abbildung 6.2 konstruiert werden, das keine zyklischen Abhängigkeiten mehr enthält. Zu diesem Zweck wird N, das heißt jeder zeitabhängige Parameter v für jeden Zeitschritt dupliziert, so dass das Netz N^* aus einer Verkettung der Netze $N_1, \ldots, N_k$, besteht, mit den Parametern $v_1, \ldots, v_k$, wie in Abbildung 6.2 dargestellt.[15]$^{S.370}$ Die Kopien der Netze teilen sich dabei die gleichen Parameter.[15]$^{S.371}$

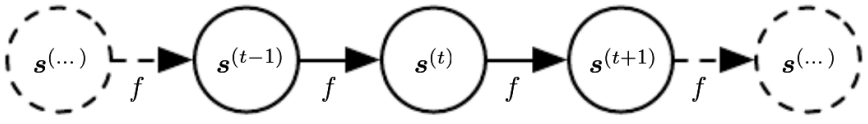

Abbildung 6.2: Quelle: Goodfellow et al. (2016) / Deep Learning[15]$^{S.369}$. Das unfold through time Verfahren für ein einzelnes Neuron, dessen Zustand von t abhängig ist. Eine Funktion f (mehr dazu in Abschnitt 6.4) bildet einen Zustand zum Zeitpuntk t auf einen Zustand zum Zeitpunkt $t+1$ ab. Für jeden Zeitschritt werden die gleichen Parameter für f benutzt.

Im Kapitel 2 wurde der Backpropagation Algorithmus erläutert, der zur Anpassung der Parameter beim Optimierungsprozess verwendet wird. Das Rückpropagieren eines Fehlersignals durch ein rekurrentes Netz erscheint nicht trivial. Das Abrollen eines rekurrenten Netzes ergibt jedoch ein einfaches Feedforward-Netz, für das der Backpropagation Algorithmus angewendet werden kann. Backpropagation in Verbindung mit unfolding through time wird auch als Backpropagation through time bezeichnet.[15]$^{S.376}$ Sei N das neuronale Netz, über das der Backpropagation Algorithmus nach n Vorwärtsschritten laufen soll. Sei N wie in Abbildung 6.3 abgerollt. Da sich $N_1, \ldots, N_n$ die Parameter teilen, können die n Aktualisierungswerte der Gewichte einfach über alle n Netze aufsummiert werden.

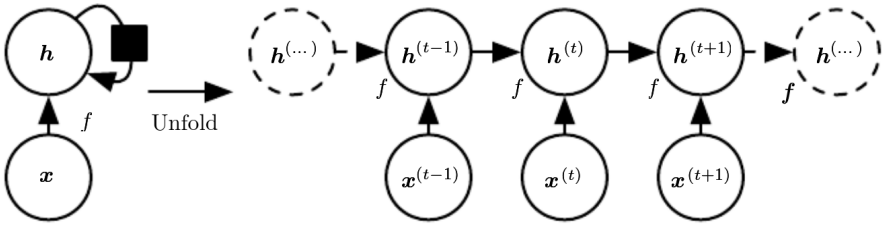

Abbildung 6.3: Quelle: Goodfellow et al. (2016) / Deep Learning[15]$^{S.370}$. Ein rekurrentes Netz, für das keine Ausgaben abgebildet sind - Das Netz verarbeitet eine Eingabe und speichert abhängig von der Eingabe einen Zustand h_t. Links: Ein rekurrentes Neuron, vgl. Abbildung 6.1. Das schwarze Rechteck notiert einen Zeitschritt Verzögerung beim Propagieren der Eingabe durch das Neuron. Rechts: Das gleiche Neuron abgerollt nach der unfold through time Methode. Jede Kopie ist assozieert mit einem bestimmten Zeitschritt.

6.3 Vanishing Gradients

Ohne weitere Modifikationen ist das Training rekurrenter Netze oft problematisch. Das ergibt sich vor allem aus der „vanishing gradient"-Problematik.[5] Bei Anwendung des Backpropagation Algorithmus werden einzelne Gewichte proportional zum Wert der partiellen Ableitung einer Fehlerfunktion nach eben diesen Gewichten aktualisiert. Wir betrachten das Netz aus Abbildung 6.4 mit Sigmoid-Aktivierungsfunktionen in den Hidden-Layern. Zur Vereinfachung ignorieren wir dabei Eingaben x und Biasterme. Berechnen wir den Gradienten für w_1, dann gilt (vgl. Gleichung 2.2):

$$\frac{\partial e}{\partial w_1} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial h_n}\frac{\partial h_n}{\partial h_{n-1}} \cdot \cdots \cdot \frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w_1} \tag{6.1}$$

Mit $h_t = \sigma(w_h \cdot h_{t-1} + w_x \cdot x_t)$ und $\frac{\partial h_i}{\partial h_{i-1}} = w_i \cdot \sigma'(h_{i-1})$ gilt also insgesamt

$$\frac{\partial e}{\partial w_1} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial h_n}\left(\prod_{k=2}^{n} w_h \cdot \sigma'(w_h \cdot h_{k-1} + w_x \cdot x_k)\right)\frac{\partial h_1}{\partial w_1}$$

wobei σ' die erste Ableitung der Sigmoidfunktion notiert. Es wird also für jeden Zeitschritt ein entsprechendes Gewicht w aufmultipliziert. Wegen des Terms $\prod_{k=2}^{n} w_k$ wird der berechnete Wert mit wachsenden n exponentiell kleiner, wenn $w < 1$ gilt.[22] Das Problem wird größer, wenn Aktivierungsfunktionen wie die Sigmoidfunktion verwendet werden.[15]$^{S.178f.}$ Diese bildet auf Werte im Intervall $(0, 0.25)$ ab, wie in Abbildung 6.5 erkennbar. Insgesamt wird für die vorderen Layer mit steigenden n der Gradient beliebig klein. Es gilt für w < 1 $\frac{\partial e}{\partial w_1} \to 0$. Für die Aktualisierung des Parameters gilt dann $w_1 = w_1 - \alpha \frac{\partial e}{\partial w_1} \to 0$. Für genügend große n tritt also kein Lerneffekt mehr ein.[5]

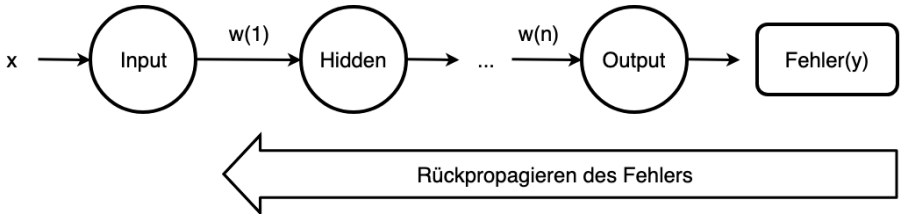

Abbildung 6.4: Ein einfaches neuronales Netz mit $n-1$ Hidden Layern. Mittels Backpropation wird bei der Berechnung des Gradienten für w_1 nach Gleichung 6.1 der Fehler durch das Netz zurückpropagiert.

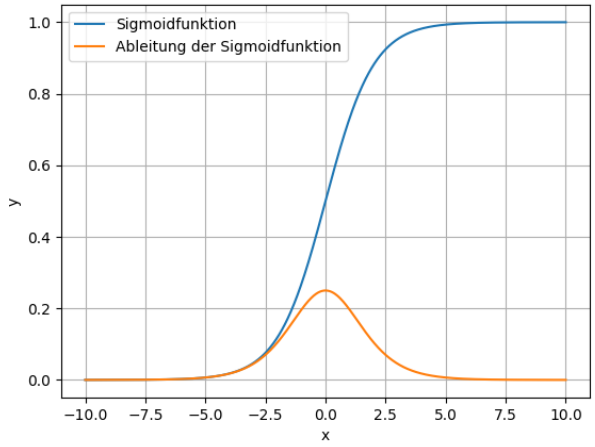

Abbildung 6.5: Die Sigmoidfunktion und ihre Ableitung im Vergleich.

6.4 Gated Recurrent Units

Zur Vermeidung von vanishing gradients kann eine Art von Gedächtnis-Zelle eingesetzt werden. Dabei wird für einen Knoten im Netz ein expliziter Zustand h_t einer Zelle eingeführt, der neben dem Eingabesignal durch das Netz propagiert wird.[22] Wir betrachten eine Variante solcher Zellen, die auch gated recurrent units (GRUs) genannt werden.[7] Abgesehen von der Speicherzelle selbst besitzen diese ein Aktualisierungsgatter (Update)

und ein Rücksetzgatter (Reset). Über diese Gatter wird der Informationsfluss zu und von der Zelle kontrolliert, in der über eine beliebige Anzahl von Zeitschritten Informationen gespeichert werden können.[7] Diese Zellen bieten nicht nur eine Möglichkeit die Gradienten über lange Sequenzen stabil zu halten, sie ermöglichen auch, komplizierte Abhängigkeiten von Eingabeelementen untereinander zu modellieren. Während rekurrente Netze in einem Zeitschritt t nur genau die Ausgabe im Schritt $t-1$ berücksichtigen, können in die Speicherzellen zu beliebigen Schritten Informationen in und aus der Zelle fließen.[22] Wir notieren im Folgenden die Sigmoidfunktion mit σ und den tangens hyperbolicus mit $tanh$. Sei für eine GRU-Zelle in einem Layer von n GRU-Zellen mit jeweils m Eingaben für eine Zelle:

$x_t \in \mathbb{R}^m$ ein Eingabevektor,

$h_t \in \mathbb{R}^n$ ein Ausgabevektor,

$z_t \in \mathbb{R}^n$ ein Aktivierungsvektor im Aktualisierungsgatter,

$r_t \in \mathbb{R}^n$ ein Aktivierungsvektor im Rücksetzgatter,

$W_z, W_r \in \mathbb{R}^{n \times m}$ Gewichtsmatrizen für die Eingaben,

$U_z, U_r \in \mathbb{R}^{n \times n}$ Gewichtsmatrizen für die Eingaben h_{t-1} aus dem letzten Zeitschritt.[7]

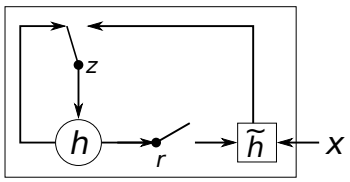

Abbildung 6.6: Quelle: Cho et al. (2014) [7] - Eine Visualisierung einer GRU-Zelle. Update-Gatter z wählt, ob der neu berechnete Zustand $h' = \tilde{h}$ in den Zustand h der Zelle einfließen soll. Reset-Gatter r entscheidet, ob der alte Zustand h vergessen werden soll.

Die Aktivierungen des Aktualisierungsgatters und des Rücksetzgatters werden mit

$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1}) \\ r_t &= \sigma(W_r x_t + U_r h_{t-1}) \end{aligned} \tag{6.2}$$

berechnet.[7] Intuitiv bestimmt das Update-Gatter, wieviel Informationen aus vergangenen Rechenschritten in den neuen Zustand mit einfließen sollen. Das Reset-Gatter bestimmt, wieviel Information vergessen werden soll. In einem Zeitschritt t wird ein neuer Zustand h'_t mit

$$h'_t = tanh(W x_t + r_t \odot U h_{t-1}) \tag{6.3}$$

berechnet, wobei $\odot$ das Hadamard-Produkt (Definition 6.4.1) bezeichnet.[7]

6.4.1 Definition. Sind $A \in \mathbb{R}^{m \times n}$ und $B \in \mathbb{R}^{m \times n}$ zwei Matrizen über $\mathbb{R}$, dann wird das **Hadamard-Produkt** $\odot$ von A und B durch

$$A \odot B = \begin{bmatrix} a_{11} \cdot b_{11} & \dots & a_{1n} \cdot b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} \cdot b_{m1} & \dots & a_{mn} \cdot b_{mn} \end{bmatrix}$$

definiert. Es berechnet sich also durch die komponentenweise Multiplikation der Einträge der Matrizen A und B.[33]

Durch die elementweise Multiplikation von r_t und Uh_{t-1} wird bestimmt, welche Informationen aus vorherigen Zeitschritten vergessen werden sollen. Damit liegen ein potentieller neuer Zustand h'_t, der den Eingabevektor x_t berücksichtigt, und ein alter Zustand h_{t-1} vor. Danach wird mit

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t$$

der letztendliche Speicherstand und die Ausgabe der Zelle berechnet.[7] Intuitiv wird hier über das Update-Gatter entschieden, zu welchen Anteilen Informationen aus dem alten und aus dem neu berechneten Speicherstand übernommen werden sollen. Gehen wir vom gleichen Modell aus Abschnitt 6.3 aus. Für die Anwendung von Backpropagation through time rollen wir ein rekurrentes Netz augmentiert mit GRU-Zellen ab, sodass wir ein Netz mit n Hidden-Layern erhalten. Im Vergleich zu Gleichung 6.1 gilt nun mit $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t$:

$$\frac{\partial h_i}{\partial h_{i-1}} = \frac{\partial}{\partial h_{i-1}}(z_i \odot h_{i-1}) + \frac{\partial}{\partial h_{i-1}}((1 - z_i) \odot h'_i)$$

Durch das Gatter wird ein additives Element eingefügt. Im Unterschied zu simplen rekurrenten Netzen tendiert der Gradient nicht garantiert zu einem Wert w, da die Gradienten verschiedene Werte annehmen können.[9] Auch wenn das Problem der vanishing gradients nicht garantiert verhindert wird, zumal auch das Problem der Sigmoidfunktion verbleibt, so führen die eingeführten Gatter doch zu stabileren Ergebnissen.[7][15]$^{S.178f.}$[22]

Kapitel 7

Convolutional Neural Networks

Für den Diskriminator liegen als Eingabe Bilddaten vor. Klassischerweise werden zur Bildklassifizierung sogenannte Convolutional Neural Networks (CNNs) genutzt, da diese räumliche Beziehung von Eingabedaten gut verarbeiten können.[28] CNNs sind grundsätzlich nichts anderes als eine spezielle Form von Feedforward-Netzen. Durch einige besondere Arten von Layern kommen sie allerdings mit weniger Resourcen aus ohne dabei an Robustheit einzubüßen.[15]$^{S.330ff.}$ Ein CNN besteht in der Regel aus einer Komposition dreier Layer-Arten: dem Convolutional Layer, dem Pooling Layer und einem sogenannten vollverknüpften Layer.

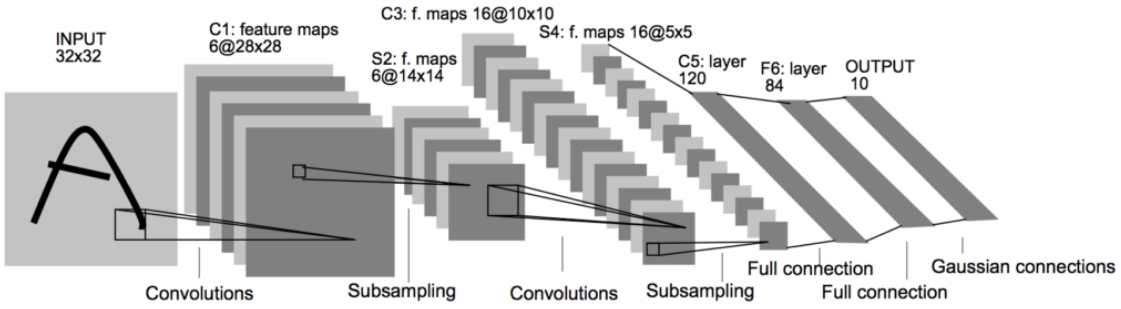

Abbildung 7.1: Quelle: Lecun et al. (1998)[28] - Abbildung einer CNN-Architektur. Mit den Subsampling Layern sind die Pooling Layer aus Abschnitt 7.2 gemeint. Bei einem fully connected Layer handelt es sich um ein vollverknüpftes Layer aus Abschnitt 7.3.

7.1 Convolutional Layer

Die Eingabe besteht typischerweise aus Matrizen. Handelt es sich um Bildformate, weisen diese zwei Dimensionen für die Kodierung des Bildes und eine weitere für die verschiedenen Kanäle auf, zum Beispiel Grau-, Alpha- oder Rot-, Grün- und Blauwerte (RGB).[15]$^{S.342,354}$

In einem Convolutional Layer wird über Filter die räumliche Nähe von Pixeln rechnerisch verarbeitet. Ein Filter besteht aus einer Matrix $F \in R^{m \times n}$, die über die Eingabe

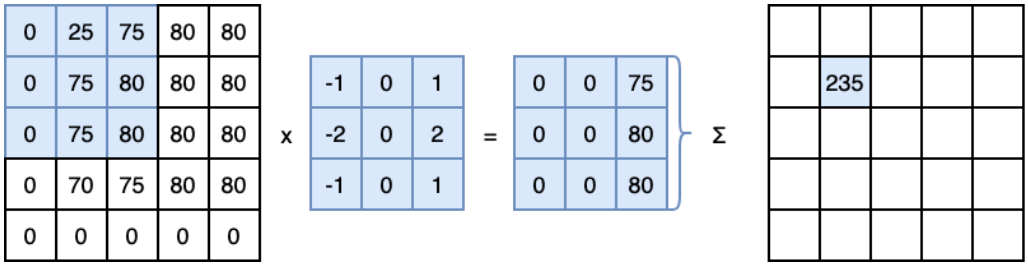

Abbildung 7.2: Ein 3x3 Filter wird auf einer Matrix angelegt. Die Einträge der Matrix wird mit den Gewichten des Filters multipliziert und die berechneten werde werden aufaddiert. Ein einer Zielmatrix wird der Wert an der Position des Filters eingetragen, bevor dieser sich weiter über die Matrix bewegt.

bewegt wird. Die Werte der Eingabe werden dabei mit denen des Filters multipliziert und zu einem Skalar aufaddiert, wie in Abbildung 7.2 zu sehen, bis jedes Element der Eingabe verarbeitet wurde.[15]$^{S.328}$ Um Eingaben mit mehreren Kanälen zu verarbeiten, werden für Eingaben mit c_i Kanälen Filter von der Dimension m x n x c_i verwendet. Für c_o Ausgabekanäle werden dabei c_o Filter angewendet.[15]$^{S.342f.}$ Üblicherweise wird der Filter Element für Element von links nach rechts und oben nach unten über die Eingabe bewegt. Es kann aber auch ein Parameter „stride “ definiert werden, der die Schrittgröße bestimmt.[15]$^{S.342}$ Sollen die Dimensionen durch das Filtern nicht verringert werden, können am Rand der Eingabematrix Nullfelder so hinzugefügt werden, dass die Dimension nicht verändert wird.[15]$^{S.343}$ Ein Convolutional Layer besteht also insgesamt aus einer Anzahl an Filtern, die eine Eingabe verarbeiten. Jedes Element der Eingabe repräsentiert einen Eingabeknoten. Jedes Element des Filterergebnisses repräsentiert einen Knoten in einem Hidden-Layer. Elemente der Ein- und Ausgabematrizen werden auch als Features bezeichnet. Das zweidimensionale Ergebnis eines Filters wird auch „feature map“ genannt.[15]$^{S.328}$

Durch Aktivierungsfunktionen werden nicht-lineare Transformationen eingeführt. Da auch CNNs unter dem vanishing gradient Problem leiden, ist die ReLU eine beliebte Variante, da diese anders als die Sigmoidfunktion stabile Gradienten erzeugt.[15]$^{S.189}$

7.2 Pooling Layer

Auch das Pooling Layer arbeitet mit Filtern. Seine Aufgabe ist es, die Eingabe auf einen niedrig-dimensionalen Raum abzubilden.[15]$^{S.337}$ Genau wie in einem Convolutional Layer wird ein Filter über die Eingabe bewegt. Die Elemente in der Eingabemaske können über verschiedene Operationen zusammengefasst werden, üblich ist zum Beispiel ein Max-Pooling, bei dem das Maximum der Werte übernommen wird, wie in Abbildung 7.3 dargestellt. [15]$^{S.335}$ Beim Pooling soll das Modell idealerweise die wichtigen Elemente aus den weniger wichtigen Elementen herausfiltern.[15]$^{S.336}$

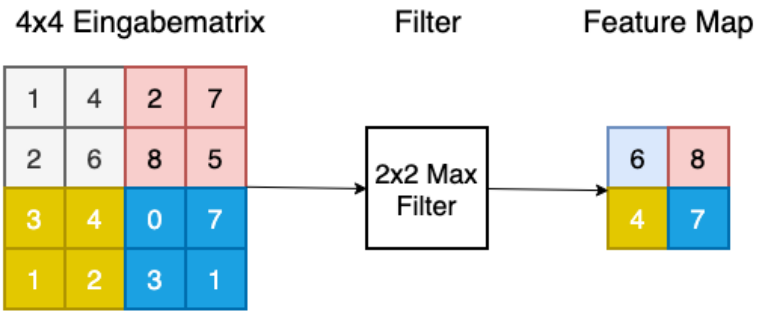

Abbildung 7.3: Ein 2x2 Max-Pooling Filter wird auf eine 4x4 Matrix angewandt, der Parameter stride ist dabei auf 2 gesetzt.

7.3 Vollverknüpftes Layer

Wie auch im vorliegenden Fall löst man mit Convolutional Neural Networks oft Klassifizierungsprobleme. [10]

7.3.1 Definition. Sei $M = \{(x_i, y_i) | i \in \{1, \ldots, m\}\}$ eine Menge von m Stichproben aus einer Verteilung D, wobei $x_i \in \mathbb{R}^n$ und $y_i \in \{1, \ldots, k\}$. Ein Klassifizierungsproblem ist das Problem mit Hilfe von M eine Funktion $f : \mathbb{R}^n \to \{1, \ldots k\}$ zu finden, die für weitere Stichproben x aus D den Wert y bestimmen kann.[15]$^{S.98}$

y wird auch als Klasse bezeichnet. Gemeint ist die Zugehörigkeit einer Beobachtung x zu einer Kategorie y.[15]$^{S.98}$ Bei der Anwendung von CNNs lässt sich das so ausdrücken, dass das Ausgabe-Layer eine Dimension von k für k Klassen hat. Dabei steht der i-te Wert im Ausgabe-Layer bei einer Eingabe x für die Wahrscheinlichkeit, dass x eine Ausprägung der i-ten Klasse ist. Dafür müssen die dreidimensionalen Ausgaben der Hidden-Layer zu einem eindimensionalen Vektor transformiert werden. Hier kommt ein vollverknüpftes Layer ins Spiel, das die Informationen der feature maps vor der Ausgabe in einem eindimensionalen Vektor aggregiert.[10]

Kapitel 8

Experimente

Der verwendete Code findet sich unter https://github.com/jp-richter/formelbaer-rnn. Für die Experimente wurde ein Datensatz von ca. 300000 Formeln aus Arbeiten verwendet, die von arxiv.org heruntergeladen wurden.[1] Die LaTeX-Dokumente wurden im Rahmen einer Projektarbeit nach Formeln durchsucht und diese im LaTeX- und PNG-Format bereitgestellt. Die Arbeiten stammen aus der Kategorie Maschinelles Lernen. Das experimentelle Vorgehen ist in Algorithmus 8.1 detailliert dargestellt.

8.1 Laufzeit

Eine große Herausforderung stellt dabei die Laufzeit des verwendeten Algorithmus in Abbildung 8.1. Oft wird beim Training neuronaler Netze die Laufzeit der Berechnungen auf den Netzen zum Flaschenhals. In diesem Fall nimmt allerdings das Kompilieren der LaTeX-Dokumente einen erheblichen Teil der Laufzeit ein. Performanztests mit Hilfe des von Python bereitgestellten cPython Profilers ergaben, dass das Kompilieren der Formeln und damit verbundene Prozesse 70% der Zeit einnahmen, während Berechnungen in Zusammenhang mit neuronalen Netzen weniger als 5% der Rechenzeit benötigten. Durchgeführt wurden die Tests auf einem System mit einem Intel(R) Core(TM) i5-7600K Prozessor, einer NVIDIA GTX 970 und einer NVMe M.2 SSD. Die vorgestellten Experimente wurden auf dem Rechencluster des SFB876[2] ausgeführt, wobei 48 Kerne (Intel(R) Xeon(R) CPU E5-2697), 4 NVIDIA Tesla k40m GPUs und 128 GB Ram zur Verfügung standen. Die Tests mit cProfile sind also nur grob aussagekräftig, dennoch bieten sie einen Anhaltspunkt für die Größenordnungen der Laufzeitanteile. Zur Verringerung der Laufzeit hat es sich neben Verwendung des RAMs als Hauptspeicher und Nebenläufigkeit bei der Kompilation der Formeln außerdem als sehr effektiv erwiesen, die Präambel der LaTeX-Dokumente gesondert für viele Dokumente auf einmal zu kompilieren und eine möglichst

[1] Der Datensatz ist verfügbar unter https://whadup.github.io/EquationLearning/.
[2] https://sfb876.tu-dortmund.de/index.html

Algorithmus 8.1 SeqGAN Adaption für Syntaxbäume

Eingabe: S_{echt}: Datensatz von Bildern echter Sequenzen
Eingabe: g: Anzahl an Generator-Trainingsschritten
Eingabe: d: Anzahl an Diskriminator-Trainingsschritten
Eingabe: k: Anzahl von Diskriminator-Trainingsepochen

1: **procedure** GAN-Training
2: Initialisiere G_θ und D mit Zufallsgewichten
3: $G_{rollout} \leftarrow G_\theta$
4: **for** i Iterationen **do**
5: **for** g Schritte **do**
6: Generiere eine Sequenz $Y_{1:T} = (y_1, \ldots, y_T) \sim G_\theta$ ▷ $i \cdot g \cdot T$
7: $R \leftarrow \{\}$
8: **for** t $\in 1 : \mathrm{T}$ **do**
9: **for** mc Monte Carlo Schritte **do**
10: Generiere $Y'_{1:T} = Y_{1:t} + Y_{t+1:T} \sim G_{rollout}$ ▷ $i \cdot g \cdot mc \cdot \sum_{l=1}^{T} l$
11: Übersetze $Y'_{1:T}$ zu Y'_{TREE}, Algorithmus 5.1 ▷ $i \cdot g \cdot mc \cdot T$
12: Übersetze Y'_{TREE} zu Y'_{LATEX} ▷ $i \cdot g \cdot mc \cdot T$
13: Kompiliere Y'_{LATEX} zu Y'_{BILD} ▷ $i \cdot g \cdot mc \cdot T$
14: $R \leftarrow R + D(Y'_{BILD})$ ▷ $i \cdot g \cdot mc \cdot T$
15: **end for**
16: **end for**
17: $R \leftarrow \frac{1}{|R|} \sum_{i=1}^{|R|} R_i$ ▷ $i \cdot g$
18: Aktualisiere G für Belohnung R über Gleichung 3.10 ▷ $i \cdot g$
19: **end for**
20: **for** d Schritte **do**
21: Generiere S_{unecht} mit G_θ ▷ $i \cdot d \cdot |S_{synth}| \cdot T$
22: ... (s. Zeilen 10-12 $\forall s \in S_{unecht}$)
23: Erstelle S aus S_{unecht} und S_{echt}, sodass $|S_{unecht}| = |S_{real}|$ ▷ $i \cdot d$
24: Trainiere D mit S für k Epochen mit Gleichung 4.1 ▷ $i \cdot d \cdot k$
25: **end for**
26: $G_{rollout} \leftarrow G_\theta$ ▷ i
27: **end for**
28: **end procedure**

niedrige Kompression der erzeugten PDF-Dokumente zu wählen. Bedingt durch den Overhead des LaTeX-Kompiliervorgangs ist es außerdem sinnvoll, viele Formeln in einem einzelnen Dokument zusammenzufassen. Die Performanztests mit cProfile wurden dabei auf dem bereits optimierten Code durchgeführt. In Algorithmus 8.1 finden sich Kommentare zur Laufzeit. Die Laufzeit hängt offensichtlich von der Laufzeit der einzelnen Aufrufe ab. Trotzdem lässt sich mit $\sum_{l=1}^{T} l = \frac{T^2+T}{2}$ ablesen, dass die Laufzeit quadratisch mit der Länge der zu generierenden Sequenzen wächst.

8.2 Hyperparameter und Bildformat

Mit der Performanz in Sachen Rechenzeit ergibt sich eine weitere Schwierigkeit: Die Fülle an wählbaren Parametern für das Training, also solchen die nicht durch das Training optimiert werden. Diese werden im Folgenden als Hyperparameter bezeichnet.[15]$^{S.96}$ Die Anzahl an wählbaren Parameterkombinationen macht eine erschöpfende Suche nach geeigneten Parameterbelegungen im Rahmen dieser Arbeit unmöglich. Daher wurde für die unten beschriebenen Experimente die maximale Länge der Sequenzen fest gewählt, so wie auch einige andere Parameter, die einen starken Einfluss auf die Laufzeit haben. Anzumerken ist, dass der Algorithmus immer Sequenzen der maximalen Länge erzeugt. Trotzdem können mit den Algorithmen 5.1 (Übersetzung von Sequenzen zu Bäumen) und 5.2 (Übersetzung von Bäumen zu Sequenzen) Syntaxbäume zu Sequenzen verschiedener Länge erzeugt werden, da der Algorithmus 5.1 bei der Übersetzung zu Syntaxbäumen abbricht, sobald alle Operatoren mit Operanden saturiert sind. Es ist möglich, dass der Generator nicht saturierte Sequenzen erzeugt, zum Beispiel „+++++++“. Das soll aber nicht verhindert, sondern dem Optimierungsprozess überlassen werden. Für die Experimente gilt, soweit nicht ausdrücklich etwas anderes angegeben: Trainingsschritte (i): 150; Sequenzlänge (T): 20; Künstliche Trainingsformeln ($|S_{synth}|$): 2000; Generator Traingsschritte (g) = 10; Diskriminator Trainingsschritte (d): 1; Diskriminator Trainingsepochen (k): 1; Monte Carlo Schritte (mc): 10. Für SeqGANs wird $g = 1, d - 5, k = 3$ vorgeschlagen, das eingesetzte Pretraining schafft jedoch für den Generator bessere Voraussetzungen als sie in diesem Fall vorliegen.[48]

Zur Kompilation der LaTeX-Dokumente wurde pdflatex verwendet. Die PDF Kompression wurde dabei auf 0 gesetzt, da Speicherzugriffe auch bei größeren Dateien mit Nutzung des RAMs als Hauptspeicher vernachlässigt werden können. Die PDF-Dateien wurden mit Ghostscript in das PNG-Format mit Grauwert- und Alphakanal transformiert. Für das Training wurde lediglich der Alphakanal verwendet, da sich das im Rahmen der Projektarbeit zur Suchmaschine als performanter erwiesen hat. Die Bildgröße der Eingabebilder wurde auf 32x332 in HöhexBreite verkleinert, so dass die Formeln auf dem Bild zentriert erscheinen. Bei der gewählten Sequenzlänge kann es passieren, dass eine Formeln nicht ganz auf dem Bild zu sehen sind. In der Praxis kam das jedoch in allen Testläufen nur

zwei mal in Experiment 5 vor. In Abbildung 8.1 ist ein Beispiel für eine Eingabe aus dem Datensatz von arxiv.org zu sehen.

$$E_s[\sum_{u<v} \alpha[Q_s, X]_{uv}] = \sum_{u<v} p_{uv}\alpha[h, X]_{uv} + \sum_{u<v<w} p_{uvw}\beta[X]_{uvw} \ .$$

Abbildung 8.1: Formel aus einem Ranking Algorithmus, Quelle: Ailon et al. (2007) [1]

8.3 Netzarchitekturen

Das Alphabet besteht aus 148 Zeichen, die 1-aus-n-Kodierung für ein Zeichen besteht daher aus einem Vektor mit der Länge 148. Als Generator wurde ein rekurrentes Netz mit zwei GRU-Layern und einem vollverknüpften Layer mit ReLU verwendet. Einige Details zu beiden Netzen lassen sich der Abbildung 8.2 entnehmen. Um eine Verteilung zu modellieren, wird vor der Ausgabe ein Softmax-Layer verwendet, dass die Eingabewerte auf einen Bereich im Intervall (0,1) abbildet.[15]$^{S.79}$ Für die rekurrenten Layer wird, wie von Yu et al. bei SeqGANs vorgeschlagen, ein Dropout von 0.2 eingesetzt.[48]

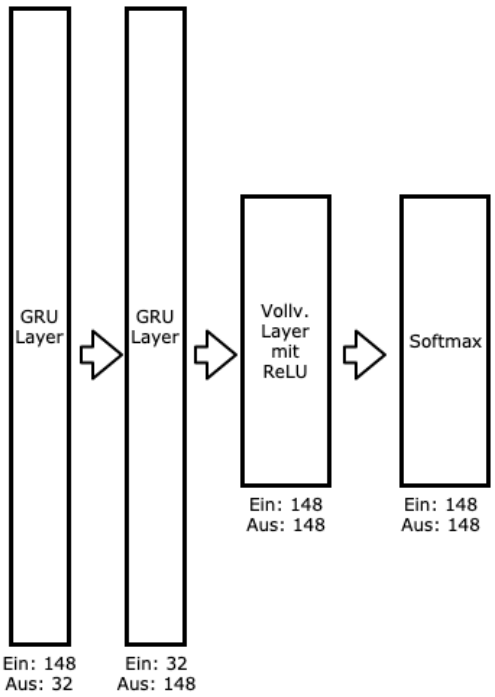

Abbildung 8.2: Die Architektur des eingesetzten Generator-Netzes. Unter den Layern sind die Ein- und Ausgabedimensionen vermerkt.

Der Diskriminator besteht aus drei Convolutional Layern, jeweils gefolgt von Pooling Layern. Die Ein- und Ausgabedimensionen sind wie (Kanäle x Höhe x Breite) zu lesen. K:(h,w) bezeichnet die Größe der Filtermaske. Der Stride-Parameter ist auf 1 gesetzt. Für die Ausgabe wird eine Sigmoidfunktion genutzt, um ein einzelnes Skalar $D(x) \in (0, 1)$ auszugeben. Die Architektur des Diskriminators wurde dabei mit kleinen Modifikationen von der Projektarbeit zur Suchmaschine übernommen. Für den Diskriminator wird eine Lernrate von $\alpha = 0.001$ festgelegt, die sich als geeignet erwiesen hat. Für beide Netze wird in den Experimenten der Adam-Optimierungsalgorithmus verwendet.

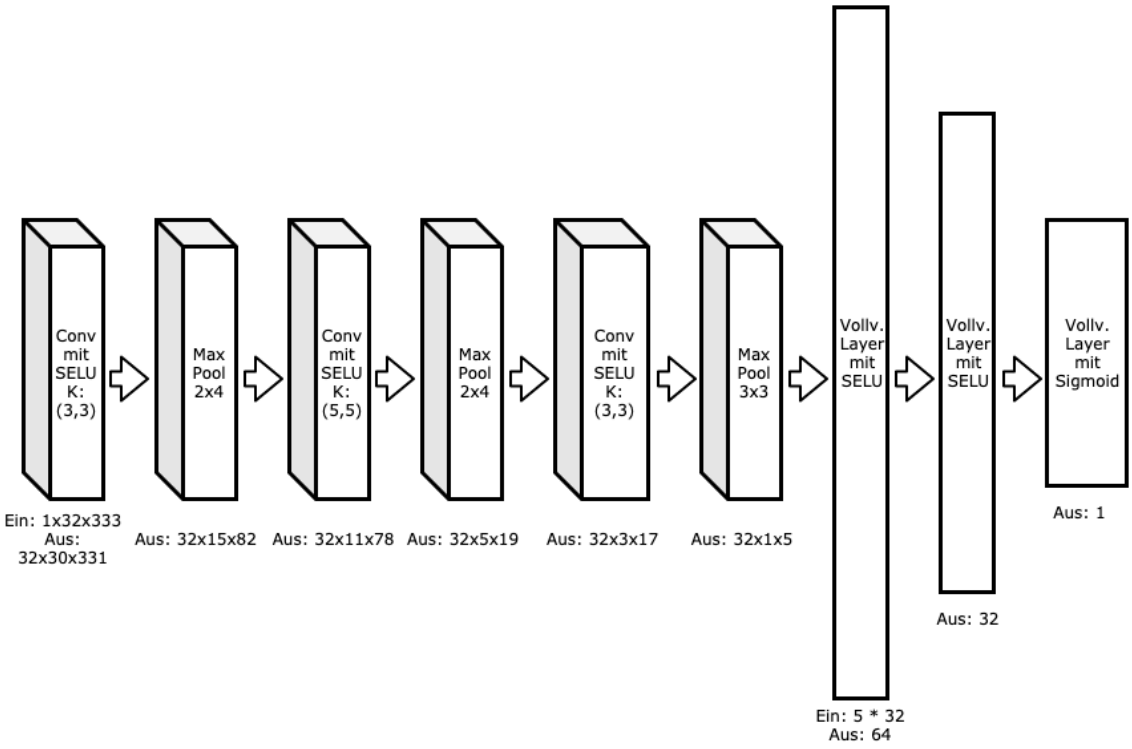

Abbildung 8.3: Die Architektur des eingesetzten Diskriminator-Netzes. Unter den Layern sind jeweils die Ein- und Ausgabedimensionen vermerkt.

8.4 Experiment 1 - Laufzeittests

Zum Aufbau der ersten Reihe von Experimenten kann nicht viel Neues gesagt werden, er entspricht den bisherigen Ausführungen. Zunächst sollte herausgestellt werden, wie die Parameter laufzeitgünstig gewählt werden können, welche Schranken für die Parameterbelegungen also verfügbar sind. Zu diesem Zweck wurde die Lernrate des Generators auf $\alpha = 0.05$ gesetzt. Es kam dabei zu folgenden Ergebnissen:

- Als laufzeittechnisch geeignet hat sich eine Batchsize von 192 erwiesen. Da der G bei allen durchgeführten Versuchen langfristig nur noch ein einzelnes Zeichen generierte,

wurde auf die Gefahr des Overfittings[3] keine Rücksicht genommen und mehr Wert auf die Genauigkeit der Gradienten gelegt. Der Vorteil einer größeren Batchsize ergibt sich aus der Anzahl von Formeln auf einem einzelnen LaTeX-Dokument. Jedem Kern wird ein LaTeX-Dokument zur Kompilation zugewiesen. Der Overhead des Kompiliervorgangs steigt stark mit zusätzlichen Dokumenten und nur zu einem kleinen Teil mit weiteren Formeln. Es ist daher laufzeittechnisch billig, weitere Formeln hinzuzufügen, insbesondere weil auch für die Nebenläufigkeit ein gewisser Overhead entsteht. Dieser lohnt sich erst, wenn jedem Kern ein LaTeX-Dokument mit genügend Formeln zugewiesen werden kann. So hat ein Test mit einer Batchsize von 48, also in einem Durchlauf jeweils eine Formel pro Kern, eine Laufzeit von 12,11h (Stunden) für 150 Iterationen ergeben. Für eine Batchsize von 192 ist die Laufzeit nur auf 20,19h gestiegen. Um die Laufzeit bei etwa 24h zu halten, wurde keine größere Batchsize gewählt. Für den Verlauf aller Experimente wurde die Batchsize nicht mehr verändert.

- Je kleiner der künstlich generierte Datensatz S_{synth} ist, umso geringer ist auch die Laufzeit. Für $|S_{synth}| = 2000$ erreichte D bereits während eines einzigen Trainingsschritts $d_{steps} = 1$ mit einer Epoche $k = 1$ eine mittlere Genauigkeit von etwa 0.8 wie in Abbildung 8.7. Für das Training von G ist es sinnvoll, dass D gerade zu Beginn nicht zu gut trainiert ist. In dem Fall gehen die Belohnungen des Generators gegen 0 und es wird kein Lerneffekt erzielt. Da dieser Wert arbiträr gewählt wurde und es nicht exakt auf das Erreichen eines bestimmten Wertes ankommt, wurde auf eine Messung der Genauigkeit auf einem Evalierungsdatensatz verzichtet.

- Ein geeigneter Parameter g, die Anzahl der Trainingsschritte des Generators, ist schwierig zu bestimmen. Die Anzahl an Parametern legt nahe, möglichst viele davon festzulegen. Für den Durchlauf in Abbildung 8.7 hat sich ergeben, dass D für $g = 10$ nach zwei Iterationen nur noch eine Genauigkeit von $0,5$ hatte. Ein höherer Wert für g hat großen Einfluss auf die Laufzeit, für die folgenden Experimenten wird daher $g = 10$ festgelegt.

- Eine Belegung für die Anzahl an Stichproben bei der Monte Carlo Simulation ist insofern einfach zu wählen, als dass ein höherer Wert immer besser ist, da damit die Genauigkeit des Mittelwerts für die Belohnung einer Teilsequenz steigt. Wir wählen $mc = 10$, da der Parameter neben der Sequenzlänge die drastischste Auswirkung auf die Laufzeit hat.

Einige Resultate sind in Abbildungen 8.4, 8.5, 8.6 zu sehen. Interessanterweise hat der erste Ansatz ohne weitere „Verbesserungen“ zumindest ansatzweise die besten Ergebnisse

[3]Ein Phänomen, bei dem ein Modell so gut auf die verwendeten Trainingsdaten abgestimmt ist, dass es nicht generalisieren kann.[15]$^{S.108}$ Für G ist das der Fall, wenn nur solche Formeln generiert werden, die schon im Datensatz vorkommen.

1)

$$\sum_{sinhX}^{sinhN \cap sinhsinhN} \sum_{N}^{sinhN} N$$

2)

$$argmin_N \sum_{N \cap N}^{7} N$$

3)

$$\sqrt{sinhN}$$

4)

$$\sum_{N \cap N}^{\sum_{sinhN}^{N} N} sinhN \cap sinhN$$

Abbildung 8.4: Ergebnisformeln aus einem Test mit $g = 20$ nach 150 Iterationen.

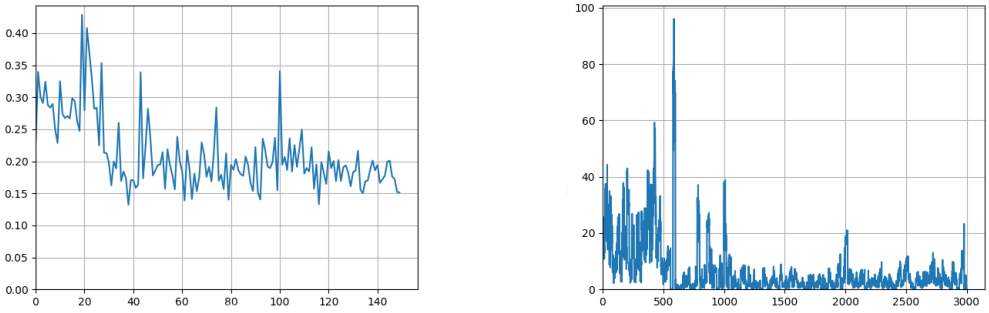

Abbildung 8.5: Experiment 1 | links: x: Iterationen, y: Mittlerer Diskriminator Fehler; rechts: x: Iterationen $*$ 20, y: Mittlere Generator Belohnung. In diesem Fall mit $g = 20$. Belohnungswerte sind aufsummiert für eine ganze Sequenz.

erzielt, wenn auch die Endresultate unbrauchbar waren. Abbildung 8.7 zeigt, dass der Generator kurzzeitig gut genug wird, um den Diskriminator hinreichend zu täuschen. Bei Iteration 20 ist im Minimax-Spiel des GAN-Settings der Diskriminator im Rückstand. In den Folgeiterationen gibt der Diskriminator keine geeigneten Belohnungsignale mehr.

1)

$$||\mathbb{E}_Y[sin\mathbb{E}_{argmax_{||\lambda||}||\prod_{||sin\omega||}^{a} sinsinU!||}[argmax_U]]||$$

2)

$$O$$

3)

$$||||||U||||||$$

4)

$$argmax_{sinsinU}||\Upsilon||$$

Abbildung 8.6: Experiment 1 | Ergebnisformeln aus einem Test aus Abbildung 8.7 mit $g = 10$ nach 150 Iterationen.

In Abbildung 8.8 ist das Verhältnis von rückpropagiertem Fehler zur Belohnung zu sehen. Es scheint, als würde der Fehler minimiert, obwohl die erhaltenen Belohnungen sinken. Es ist allerdings genau andersherum: der Fehler wird kleiner, weil die erhaltenen Belohnungen sinken. Wir minimieren $-[log(p) \cdot r]$, für $p = G(y_t|Y_{1:t-1})$ und $r = D(Y_{1:t} + Y_{t+1:T})$ für ein konstantes r. Es ist also nicht das Gradientenverfahren, das für die Verringerung

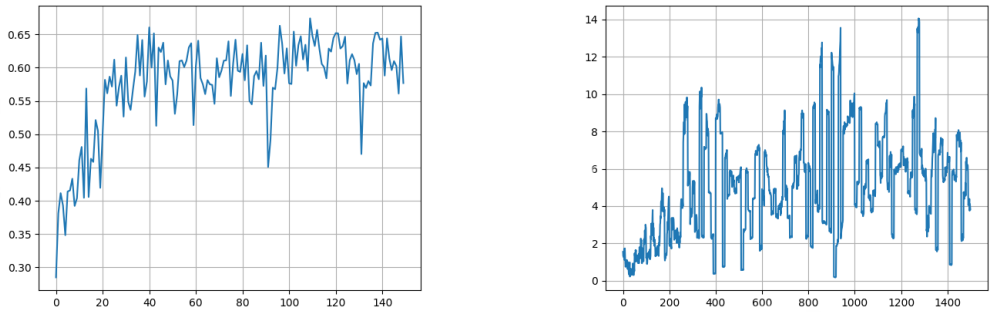

Abbildung 8.7: Experiment 1 | links: x: Iterationen, y: Diskriminator Fehler; rechts: x: Iterationen $*$ 10, y: Mittlere Generator Belohnung. Belohnungswerte sind aufsummiert für eine ganze Sequenz.

des Fehlers verantwortlich ist. Wir werden uns daher im Folgenden auf die Darstellung solcher Kurven beschränken, die die erhaltenen Belohnungswerte zeigen. Im Gegensatz zum berechneten Fehler, sind diese aussagekräftig bezüglich des Trainingsziels.

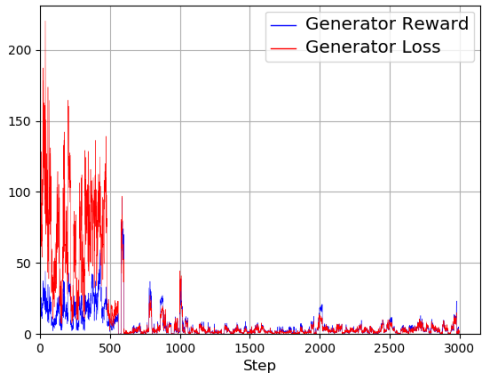

Abbildung 8.8: Experiment 1 | Mittlerer Fehler (Loss) und mittlere Belohnung (Reward) für einen Testlauf mit $g = 20$, $|S_{unecht}| = 1000$ und $Batchsize = 48$. Es ist gut zu sehen, wie die Belohnung den Fehler skaliert.

8.5 Experiment 2 - Bias, Baseline und Entropie

Um ein Ergebnis wie das aus 8.7 zu vermeiden und sicherzustellen, dass D nicht auf Daten lernt, die der künstlichen Verteilung nicht mehr entsprechen, wurde zwischen Generator- und Diskriminator-Training der Parametervektor des Diskriminators θ_D in den folgenden Experimenten zurückgesetzt. Außerdem wird in diesem Experiment alle 20 Iterationen die Größe des künstlichen Datensatzes S_{synth} um 2000 Formeln und damit auch die Größe des echten Datensatzes S_{real} um 2000 Formeln vergrößert.

In dieser Versuchsreihe wurde eine Baseline eingeführt. Eine simple, aber effektive Wahl ist die erwartete Durchschnittsbelohnung. Wir wählen einen etwas niedrigeren Wert von 0.05, um negative Belohnungen zwecks besserer Interpretierbarkeit zu vermeiden. Die zu minimierende Zielfunktion $-log(\mathbb{P}(Aktion)) * Belohnung$ würde sonst zwischen positiven und negativen Werten alternieren. Für den Optimierungsprozess wäre das unschädlich, da die Belohnung für einen Backpropagationschritt konstant ist.[16]

Um Ergebnisse wie aus Abbildung 8.7 verlässlich zu reproduzieren, wäre es vorteilhaft, dem Generator gerade zu Beginn des Trainings einen Vorteil zu verschaffen. Anstatt eines Pretrainings gibt es die Möglichkeit, eine günstige Initialisierung der Biaswerte für den Generator vorzunehmen. Typischerweise werden diese mit 0, bei rekurrenten Netzen auch mit 1 initialisiert.[25] Mit den Biaswerten lassen sich die Funktionen der einzelnen Neuronen verschieben, ein Beispiel dazu findet sich in Abbildung 8.9. Das lineare Ausgabelayer des Generators gibt Wahrscheinlichkeitswerte für Zeichen aus. Ist die ergodische Verteilung der Zeichen für die Menge der Sequenzen bekannt, ergibt das einen Anhaltspunkt für Werte, die proportional zu den echten Biastermen des linearen Ausgabelayers sind. Um diese Verteilung zu approximieren wurde auf ca. 100000 Formeln aus 2000 wissenschaftlichen Arbeiten ein Skript ausgeführt, das die absoluten Häufigkeiten der Formeln zählt. Das Ergebnis findet sich im Anhang A.2. Aus diesen Werten wurde ein Vektor mit den relativen Häufigkeiten berechnet und zur Initialisierung der Biaswerte des linearen Ausgabe-Layers verwendet.

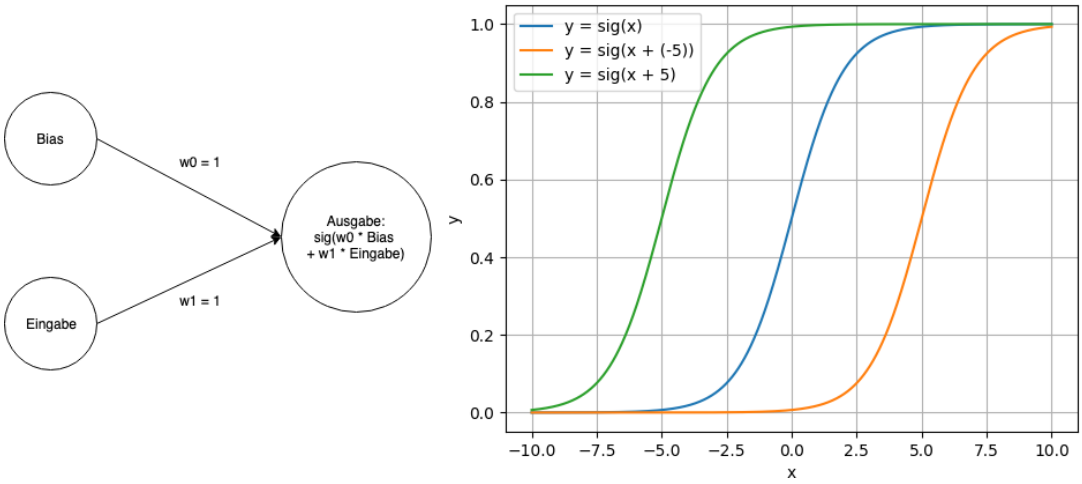

Abbildung 8.9: Beispiel für die Ausgabe eines einfachen Neurons mit verschiedenen Biaswerten

In allen bisherigen Durchläufen, so auch in den Abbildungen zu Experiment 1 zu sehen, ergaben sich hohe Wahrscheinlichkeiten für einige wenige Zeichen. Um die Ergebnisse zu diversifizieren und insbesondere den Generator anzuregen andere Möglichkeiten zu erkunden anstatt die relativ hohen Belohnungen bestimmter Zeichen auszunutzen, fügen wir die Entropie für die, durch den Generator modellierten Verteilung der Zielfunktion hinzu.[34][46] Statt $-[(log(G(y_t|Y_{1:t-1})) \cdot D(Y_{1:t} + Y_{t+1:T})]$ für eine Sequenz Y und eine mit

1)

$$\sum_{\alpha}^{M} a \subseteq argminZ < \beta < \int_{\Phi}^{x} O^X \cap Y$$

2)

$$\frac{maxY \subset b}{min\ min_{sinhI \cap \phi} M^{-1}} - V$$

3)

$$\epsilon$$

4)

$$-\sqrt{f}$$

Abbildung 8.10: Experiment 2 | Beispiele für Resultate einiger Testläufe aus Experiment 2

$G_{rollout}$ generierte Restsequenz wird nun $-[(log(G(y_t|Y_{1:t-1})) \cdot D(Y_{1:t} + Y_{t+1:T}) + \beta H(G)]$ minimiert, wobei $H(G)$ die Entropie für die durch G modellierten Verteilung bezeichnet.

8.5.1 Definition. Sei X eine diskrete Zufallsvariable mit den möglichen Werten $\{x_1, \ldots, x_n\}$ und einer diskreten Dichtefunktion $P(X)$. Dann ist die Entropie $H(X)$ definiert als:[15]$^{S.164}$

$$\begin{aligned} H(X) &= \mathbb{E}[-log(P(X)] \\ &= -\sum_{i=1}^{n} P(x_i) \log_b P(x_i). \end{aligned}$$

für $P(x_i) = 0$ wird $0 \cdot log_2(0) = 0$ definiert.

Die Entropie für eine Verteilung entspricht intuitiv ihrer Diversität.[15]$^{S.72}$ Wir wählen $b = 2$ und einen Faktor $\beta = 0.005$, um den Einfluss der Entropie auf die Zielfunktion zu kontrollieren. Ergebnisse finden sich in Abbildungen 8.11, 8.12.

Für alle Testversuche dieser Reihe ergibt sich dabei ein ähnliches Bild. Die Initialisierung der Biaswerte scheint keine auffallende Verbesserung des Generators in den ersten Iterationen zur Folge zu haben. Eine mögliche Erklärung wäre das große Verhältnis von nullstelligen Operatoren, also Bezeichmern, zu mehrstelligen Operatoren in den gezählten Häufigkeiten. Offensichtlich kommt auf jeden mehrstelligen Operator mindestens ein nullstelliger Operator. Steht ein nullstelliger Operator an erster Stelle, dann bricht der Übersetzungsalgorithmus an dieser Stelle ab. Die Ergebnisse in Abbildung 8.10 sind dabei stellvertretend für das Gesamtresultat, es kommen vermehrt sehr kurze Sequenzen vor.

8.6 Experiment 3 - Aktualisierung pro Schritt

Testweise wurde probiert, ob eine Aktualisierung des Generators nach jedem Abschluss einer Monte-Carlo-Simulation bessere Ergebnisse erzielt. Auf die Vergrößerung des Trainingsdatensatzes für den Diskriminator alle 20 Iterationen wurde vorerst wieder verzichtet. Wie auch bei den Experimenten 1 und 2 wurden mehrere Testläufe durchgeführt, die unergiebig blieben. Einige Resultate sind in Abbildung 8.13 zu sehen. Für alle durchgeführten Testläufe bestand der weit überwiegende Teil der Sequenzen nur aus einzelnen Variablen.

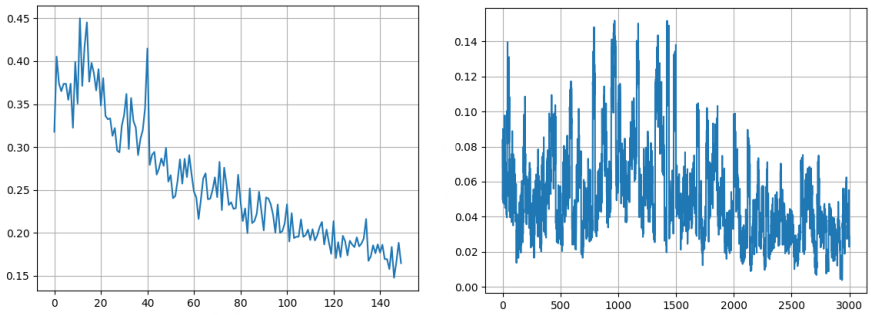

Abbildung 8.11: Experiment 2 | links: x: Iterationen, y: Mittlerer Diskriminator Fehler; rechts: x: Iterationen $*$ 20, y: Mittlere Generator Belohnung für $g = 20$. Belohnungswerte sind aufsummiert für eine ganze Sequenz.

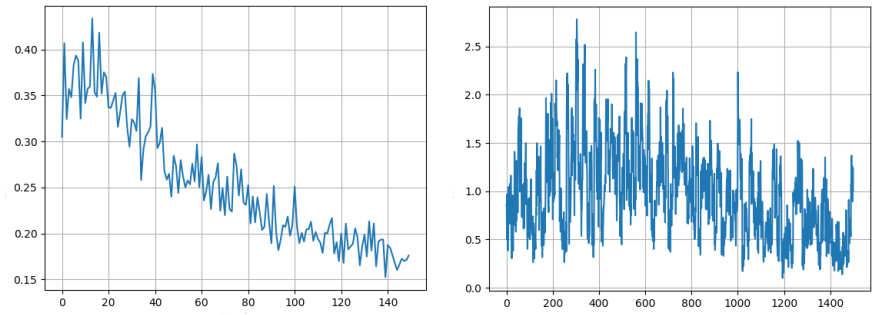

Abbildung 8.12: Experiment 2 | links: x: Iterationen, y: Mittlerer Diskriminator Fehler; rechts: x: Iterationen $*$ 10, y: Mittlere Generator Belohnung für $g = 10$. Belohnungswerte sind aufsummiert für eine ganze Sequenz.

8.7 Experiment 4 - Suche nach einer geeigneten Lernrate

Bisher wurden nur vereinzelt einige Parameter verändert, der Schwerpunkt lag auf der Funktionsweise des Algorithmus. Da das wenig erfolgreich war und die erfolgversprechendsten Ergebnisse in Experiment 1 erzielt wurden, wird die Zielfunktion in diesem Experiment ohne Entropie berechnet und die Policy Gradient Aktualisierung erfolgt wieder für eine ganze Sequenz anstatt für die Belohnungen jedes einzelnen Schrittes. Viele Parameter sind durch die Laufzeit beschränkt, viele andere Parameter sind nur indirekt für den Erfolg des Trainings verantwortlich. Wir setzen die Parameter so fest wie in der Einleitung beschrieben und führen Tests für verschiedene Lernraten $\alpha_G \in \{0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2\}$ durch. Yu et al. (2016) wählen in einem vergleichbaren Aufbau eine Lernrate von $\alpha_G = 0.2$.[48]

Die Ergebnisse fallen erneut unbefriedigend aus. Dennoch lassen die Abbildungen in 8.16 Raum für Interpretation bezüglich der Ursachen für den ausbleibenden Erfolg des Generators. Sobald die Wahrscheinlichkeit für ein bestimmtes Zeichen genügend hoch ist, wird für viele Iterationen fast ausschließlich dieses Zeichen generiert. Handelt es sich um

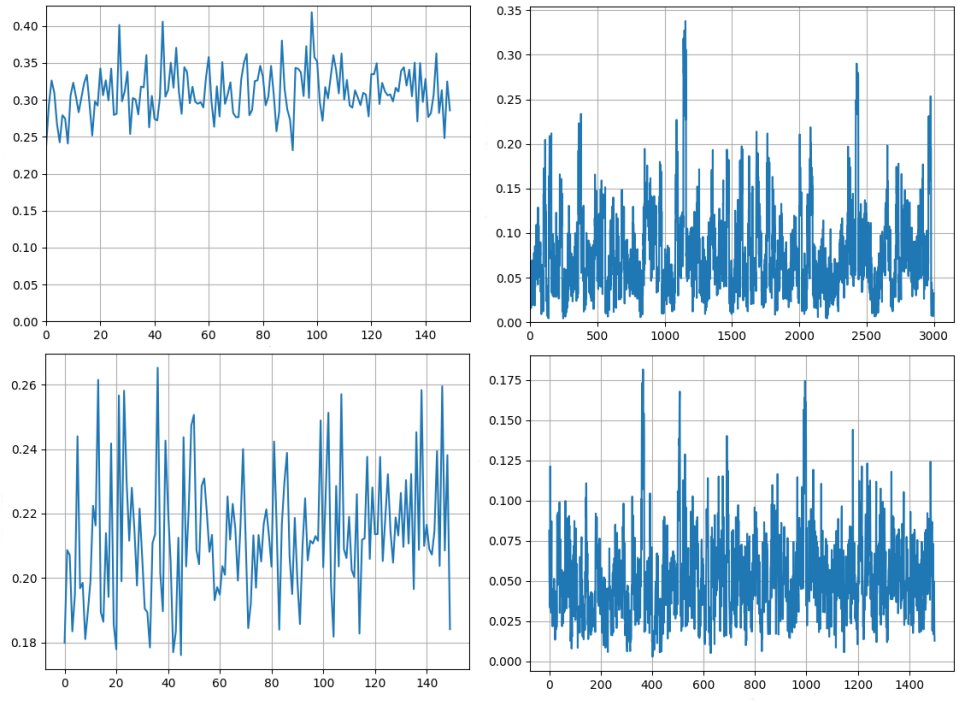

Abbildung 8.13: Experiment 3 | oben-links: x: Iterationen, y: Mittlerer Diskriminator Fehler; oben-rechts: x: Iterationen $*20$, y: Mittlere Generator Belohnung für $g = 20, |S_{synth}| = 1000$. | unten-links: x: Iterationen, y: Mittlerer Diskriminator Fehler; unten-rechts: x: Iterationen $*10$, y: Mittlere Generator Belohnung für $g = 10, |S_{synth}| = 2000$. Belohnungswerte sind aufsummiert für eine ganze Sequenz.

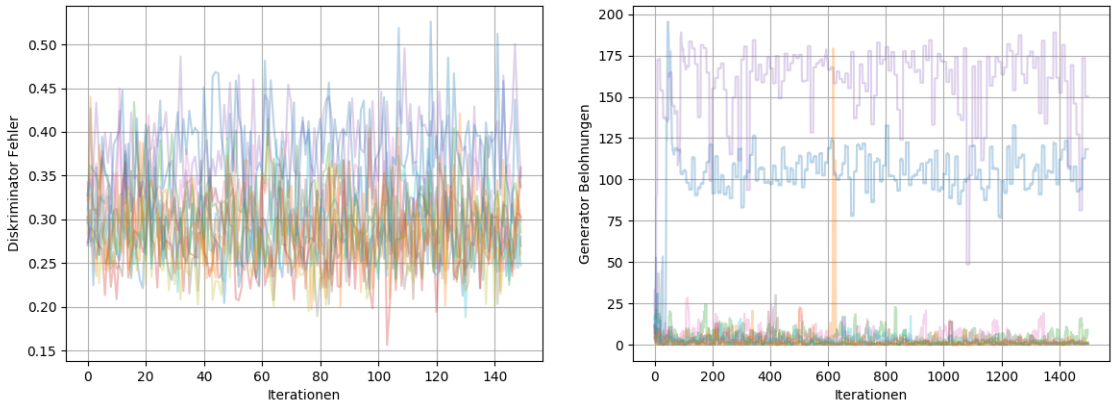

Abbildung 8.14: Experiment 4 | Zu sehen sind die Resultate für alle Durchläufe. Belohnungswerte sind aufsummiert für eine ganze Sequenz. Die hohen Belohnungen wurden für lange Sequenzen rekursiver Operatoren erreicht, die niedrigen durch nullstellige Operatoren.

einen mehrstelligen Operator, fällt die Gesamtbelohnung insgesamt besser aus, so zu sehen in Abbildung 8.14. Die Funktionsweise der Parameteraktualisierung proportional zu den

1)

||||||||||||||||||||||||||||||||||||||||

2)

maxmaxmaxmaxmaxmaxmaxmaxmaxmaxmaxmaxmaxmaxmaxmaxmaxmaxmax

Abbildung 8.15: Experiment 4 | Die höchsten Belohnungen in 8.14 wurden für die Sequenzen 1) erzielt, die zweithöchsten für 2). Alle anderen Durchläufe resultierten in Sequenzen mit einem einzelnen Zeichen, wobei für einen Durchlauf das Zeichen das Gleiche blieb.

erhaltenen Belohnungen setzt voraus, dass möglichst viele Aktionen eine Belohung erhalten, damit die Aktualisierungen in Relation zueinander stehen.[45] Wird nur noch eine einzige Handlung ausgeführt, steht nur noch ein Absolutwert zur Verfügung. Das Gradientenverfahren aktualisiert die Gewichte so, dass die Wahrscheinlichkeiten gesteigert werden, da das Vorzeichen der Belohnungen bisher immer positiv war. Es wird zunehmend unwahrscheinlicher, dass andere Zeichen generiert werden. Die bisherigen Ergebnisse lassen vermuten, dass das Training des Generators an dieser Stelle ins Leere läuft.

8.8 Experiment 5 - Lernrate bei modifizierter Zielfunktion

Für dieses Experiment wurde daher die Baseline für den Generator auf 0.1 gesetzt, sodass der Generator auch negative Belohnungen bekommt. Wird die Wahrscheinlichkeit für ein Zeichen zu hoch, werden mit negativen Belohungen Gewichte also auch so aktualisiert, dass Wahrscheinlichkeiten für generierte Zeichen sinken. Zusätzlich wurde die um die Entropie bereicherte Zielfunktion wieder eingeführt, diese erzielt einen ähnlichen Effekt. Für dieses Experiment setzen wir den Faktor der Entropie β auf $\beta = 0.01$.

Bis jetzt wurde $-log(\mathbb{P}(Aktion)) * Belohnung$ minimiert. Die Belohnung ist die vom Diskriminator ausgegebene Wahrscheinlichkeit $D(x)$, dass eine Sequenz echt ist. Da der Diskriminator bisher leichtes Spiel gegen den Generator hatte, fallen die Belohnungen in einen Wertebereich nahe 0. Da die Gradienten mit den Belohnungen skalieren (vgl. Gleichung 3.10), wird also auch der Lerneffekt beeinflusst. Um stabilere Gradienten zu erzeugen, modifizieren wir noch einmal die Zielfunktion e wie folgt:

$$\begin{aligned} e(y_t) &= -[(log(G(y_t|Y_{1:t-1})) \cdot D(Y_{1:t} + Y_{t+1:T}) + \beta H(G)] \\ &\rightarrow -[log(G(y_t|Y_{1:t-1})) \cdot -(1 - D(Y_{1:t} + Y_{t+1:T})) + \beta H(G)], \end{aligned}$$

für ein Zeichen y_t einer Sequenz $Y_{1:t}$, wobei $Y_{t+1:T}$ mit $G_{rollout}$ vervollständigt wird. $H(G)$ bezeichnet die Entropie der von G generierten Verteilung. Intuitiv wird also nicht die negierte Belohnung minimiert, sondern die von D ausgegebene Wahrscheinlichkeit, dass

$Y_{1:t}$ künstlich ist. Bezüglich des Trainingsziels führt beides zum gleichen Ziel. Zuletzt wird auch der Trainingsdatensatz für den Diskriminator wieder alle 20 Iterationen vergrößert, da der Diskriminator im letzten Experiment teilweise eine Genauigkeit von 40% erreicht hat.

Für dieses Setup wurden die Lernraten $\alpha_G \in \{0.008, 0.02, 0.04, 0.06, 0.08, 0.1, 0.2, 0.3, 0.4, 0.5\}$ getestet. In Abbildungen 8.17, 8.18, 8.19 sind einige Ergebnisse des Experiments dargestellt. Abgesehen wenigen Testläufen wurden entweder lange Formeln mit einem einzigen rekursiven Operator oder Formeln mit nur einem einzelnen Zeichen generiert. Die Ergebnisse sind in dieser Hinsicht für Experiment 5 etwas besser ausgefallen. Die besten Ergebnisse erzielte der Testlauf mit $\alpha_G = 0.1$.

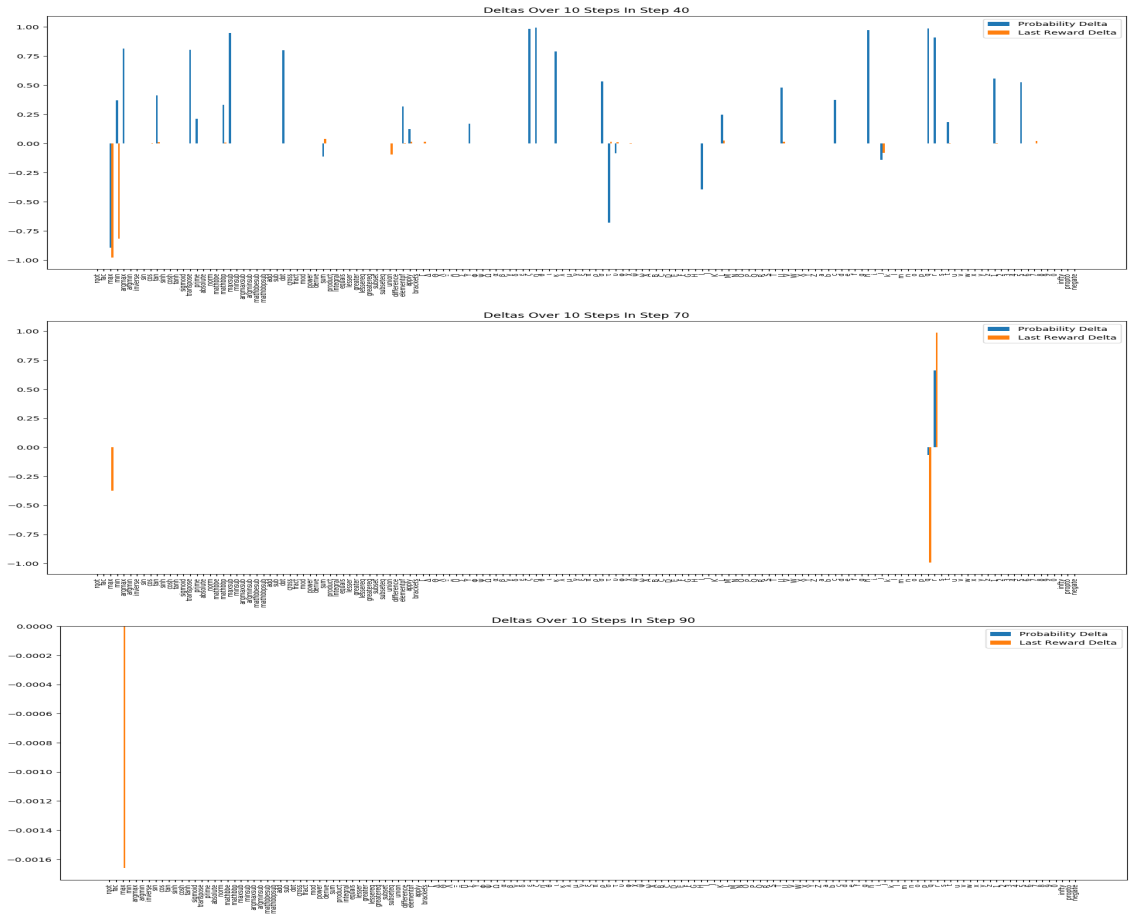

Abbildung 8.16: Experiment 4 | In jedem der Diagramme sind die Veränderungen von Wahrscheinlichkeiten und Belohungen über einen Zeitraum von jeweils 10 Schritten dargestellt, so dass für einen Schritt t die Wahrscheinlichkeiten (blau, linksseitig) der Schritte t-10 bis t und die Belohnungen (gelb, rechtsseitig) für die Schritte t-20 bis t-10 aufgezeichnet sind. So kann dargestellt werden, ob eine Veränderung der Belohnung zeitnah auch eine Veränderung der Verteilung zur Folge hat. Für alle Testläufe gilt, dass zu Anfang viele Veränderungen stattfinden. Nach ca. 10 Iterationen wird bei allen Testläufen nur noch ein einzelnes Zeichen ausgegeben, für das sich abwechselnd positive und negative Veränderungswerte bei der Belohnung ergeben, während die Verteilung weitestgehend unverändert bleibt. Für die obere Abbildung gilt $t = 40$, für die mittlere Abbilderung $t = 70$ und für die untere Abbildung $t = 90$ (das entspricht Iterationen 4, 7 und 9). In der untersten Abbildung sieht man klar, dass die Belohnung gesunken ist, die Wahrscheinlichkeit aber nicht mehr sinkt. Es handelt sich bei allen drei Graphen um den Testlauf mit den zweithöchsten Belohnungen (Abbildung 8.14). Beim Zeichen im untersten Graph handelt es sich um *max*, vgl. Abbildung 8.15.

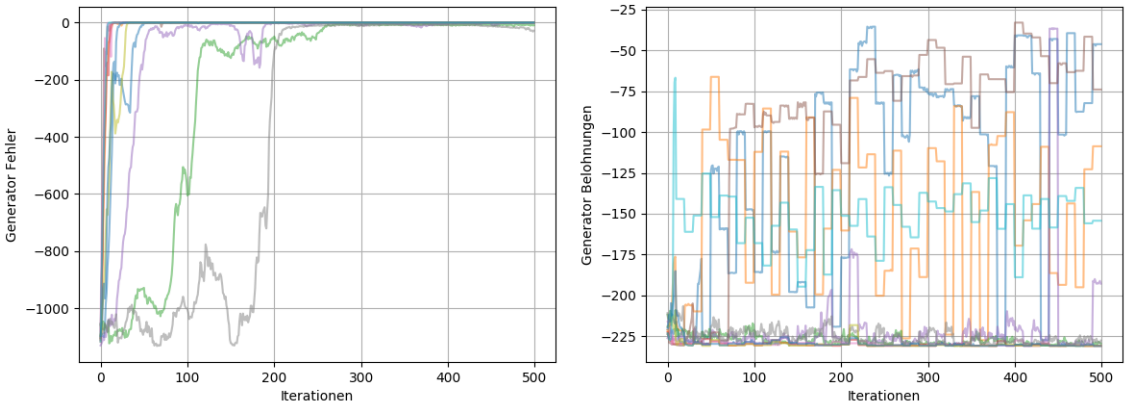

Abbildung 8.17: Experiment 5 | Zu sehen sind die Resultate für alle Durchläufe. Links: Mittlerer Generator Fehler. Für $\alpha_G = 0.1$ bleibt der Fehler am längsten stabil (lila), darauf folgt $\alpha_G = 0.008$ (grün), dann $\alpha = 0.02$ (lila). Rechts: Mittlere Generator Belohnung. Die höchsten Belohnungen (im Mittel ca. > -175) wurden für $\alpha = 0.04, 0.1, 0.3, 0, 4$.

1)	3)
$\mathbb{E}[\theta]$	$\mathbb{E}[X]^{-1} mod X$
2)	4)
$X \subseteq X \cap X$	$\mathbb{E}[min\ \mathbb{E}_X[X]]$

Abbildung 8.18: Experiment 5 | Beispiele für Resultate einiger Testläufe aus Experiment 5 mit einer Lernrate $\alpha_G = 0.1$. Visuell machen die Ergebnisse dieses Testlaufs den besten Eindruck. Die Formeln sind allerdings handverlesen, ein großer Teil der Formeln bestand nur aus einem Zeichen.

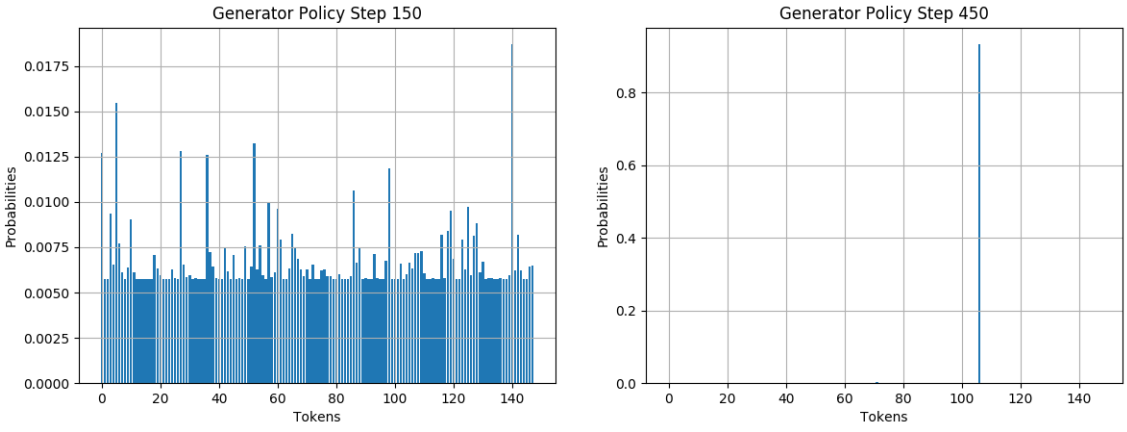

Abbildung 8.19: Experiment 5 | Links: Die mittlere von G generierte Verteilung im Schritt 150. Rechts: Die mittlere Verteilung im Schritt 450. Beide Kurven stammen aus dem Testlauf mit $\alpha_G = 0.1$.

Kapitel 9

Evaluation und Ausblick

Betrachtet man die Formeln, die in den durchgeführten Experimenten generiert wurden, ergibt sich insgesamt ein unbefriedigendes Ergebnis. Keine der Formeln sieht denen aus dem Datensatz besonders ähnlich oder könnte aus einer wissenschaftlichen Arbeit aus dem Bereich Maschinelles Lernen stammen. Auch wenn man die Experimente nicht ergebnisorientiert, sondern mit Hinblick den Verlauf des Optimierungsprozesses von G untersucht, ergeben sich kaum Anhaltspunkte für Teilerfolge. Das erhoffte Ergebnis beim Training von G wäre eine im Mittel konstante Steigerung der Belohnungen. Wie bereits in Experiment 1 beschrieben, wird der Fehler in vielen Testläufen minimiert. Das indiziert aber nicht die Konvergenz von G, sondern ist durch eine Skalierung des Fehlers durch kleiner werdende Belohnungen zu erklären. Offensichtlich war also keines der durchgeführten Experimente erfolgreich. Die Probleme, die sich bei der Verwendung von GANs mit REINFORCE ergeben haben, wurden zum Teil in der Grundlagenerörterung und der Beschreibung der Experimente dargelegt. Sie sollen noch einmal zusammengefasst werden und ergänzt werden:

- Das grundlegende Problem: Die mit REINFORCE über Monte Carlo Simulationen approximierten Gradienten für die Optimierung von G sind zu schlechte Näherungen der echten Gradienten.[43] Bei einer intuitiven Betrachtung lässt sich die Situation des Agenten mit einem großen Labyrinth vergleichen. Aus 20^{148} möglichen Wegen, die er gehen kann, führt nur ein kleiner Bruchteil zum Ausgang. Da die Sequenzen durch Variablen schnell saturiert sind, führt der größte Teil der Wege direkt in eine Sackgasse, aus der der Agent nicht wieder herauskommt und von vorne beginnen muss. Nicht nur das, er trägt sich für jede Wegkreuzung ein, wie gut seine Entscheidung war und wird sich in der Zukunft daran orientieren. Angenommen es liegt eine Teilsequenz der Länge 10 vor, dann wird die mittlere Belohnung anhand von gerade einmal 10 Stichproben aus 10^{148} möglichen Belohnungen geschätzt. Die Notizen, die dem Agenten in der Zukunft helfen sollen, sind also mit einem hohen Unsicherheitsgrad behaftet.

Generell ist das Training von GANs selbst dann schwierig, wenn die Funktionskomposition aus beiden Netzen differenzierbar ist.[40][17][30] Durch die Anwendung von REINFORCE kommt die Unsicherheit der Monte Carlo Simulationen dazu. Laufzeitbedingt ist es im Rahmen dieser Arbeit nicht möglich herauszufinden, wie viele Simulationen durchgeführt werden müssten, um ein geeignetes Intervall für die Lage des echten Mittelwerts zu bestimmen, zumal auch keine Informationen über die Verteilung der Belohungen vorliegen.

- Bei möglichen Lösungsansätzen ergeben sich weitere Hürden, so der Mangel an Trainingsdaten für ein Pretraining, das für den Lernerfolg ausschlaggebend war. So kommen auf 50 Iterationen GAN-Training beim Training von SeqGANs 100 Iterationen Pretraining.[48]

- Ein weiteres Problem blieb bisher unerwähnt und ist in den Resultaten durch die Aussparung der Fehlerkurven zunächst nicht ersichtlich. Tatsächlich geht in den meisten Versuchen der Fehler im Laufe des Trainings gegen 0, auch wenn die Belohnungswerte stabil blieben. Das ist allerdings aus folgendem Grund nicht als Erfolg, sondern als Indikator für ein Problem zu bewerten: Nehmen wir an, dass für ein einzelnes Zeichen y $\mathbb{P}(y) \rightarrow 1$ gilt (im Kapitel zum Experimentalteil wurde dieses Verhalten dokumentiert). Dann gilt $-log(\mathbb{P}(y)) \rightarrow 0$ und damit auch $-(log(\mathbb{P}(y) \cdot r) \rightarrow 0$ für eine beliebige Belohnung $r \in \mathbb{R}$. Abbildung 9.1 bildet das Verhalten ab. Die Abbildungen sind repräsentativ für den größten Teil der Testläufe. Während die Fehlerkurven bis zu diesem Punkt ein typisches „Zickzack“-Muster aufweisen, wird dieses Muster schwächer. Es liegt deshalb nahe, dass die Gradienten schwächer werden, wenn die Softmax-Funktion in der Ausgabe von G saturiert, d.h. für die Wahrscheinlichkeit einer Klasse einen Wert gegen 1 ausgibt.[13][36]

- Die Menge an wählbaren Hyperparametern macht eine erschöpfende Suche nach geeigneten Belegungen schwierig. Dazu zählen die Hyperparameter für beide Netze, die für das GAN-Training und solche, die das Format der Formeln betreffen.

- Der Zwischenschritt der LaTeX-Kompilation potenziert mit seiner Auswirkung auf die Laufzeit die oben genannten Probleme.

Um das Problem zu lösen, bei dem einzelne Wahrscheinlichkeiten so hoch werden, dass der Fehler gegen 0 geht, wäre es möglich die Entropie bei der Berechnung des Fehlers stärker zu gewichten. Eine Normalisierung der durch das Netz in die Softmax-Funktion propagierten Werte, könnte möglicherweise das Problem lösen, bei dem bei Fehlerwerten gegen 0 kein Lernerfolg mehr erzielt wird.[13][36] Eine umfangreichere Erforschung des Problems würde aber weitere Laufzeitverbesserungen oder mehr Rechenzeit vorraussetzen. Die durchschnittliche Laufzeit eines Testlaufs betrug etwa einen Tag. Die Ergebnisse sind

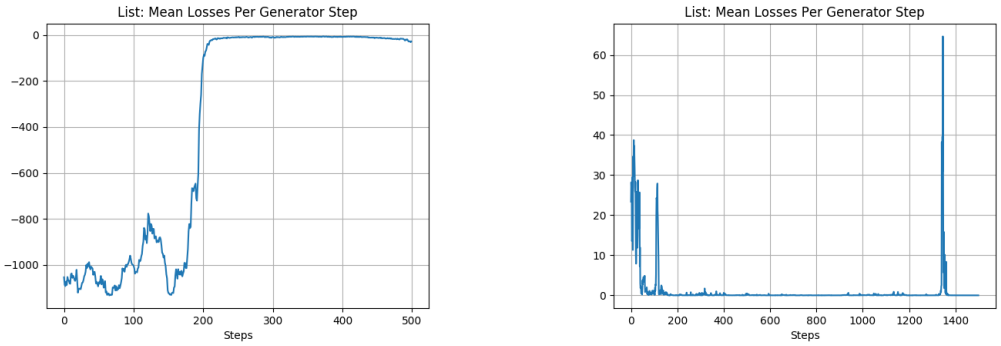

Abbildung 9.1: Links: Eine Fehlerkurve aus Experiment 5 mit modifzierter Zielfunktion und negativem Fehler. Rechts: Eine Fehlerkurve aus einem vorangegangenen Experiment. Beide sind jeweils repräsentativ für vergleichbare Testläufe. Der Fehler geht gegen 0, wenn die Wahrscheinlichkeit für ein Zeichen gegen 1 geht.

wegen der stochastischen Zeichenauswahl insbesondere bei der Approximation von Belohnungen für Teilsequenzen nicht deterministisch. Es wäre möglich, dass eine hier getestete Einstellung grundsätzlich funktioniert, aber nur in jedem zehnten Versuch. Es wäre daher optimal jeden Testlauf mehrfach auszuführen.

Rekurrente Netze mit GRU-Zellen haben sich in vielen Bereichen als sehr performant erwiesen. Bei der Auswahl zwischen GRU-Netzen und anderen rekurrenten Netzen wie den „long short-term memory“-Netzen (LSTM) gibt es keinen klaren Gewinner.[25][47] Die GRU-Architektur wurde gewählt, weil sie weniger lernbare Parameter als bei klassischen LSTMs bedeuten. Dennoch könnte es sich lohnen mit LSTMs eine andere Generatorarchitektur zu wählen.[48] Weiterhin wurden im Kapitel zu GANs neben SeqGANs auch andere Konzepte vorgestellt. Diese Arbeit hat sich stark an SeqGANs orientiert. Im Bereich der Sequenzgeneration setzen einige andere Vorgehensweisen das Vorliegen von Trainingsdaten in Sequenzform voraus, sie sind also nicht direkt anwendbar.[1] Von Interesse könnte jedoch, wie zum Beispiel von Tuan et al. (2019) vorgeschlagen, ein neuronales Netz zur Approximation der Belohnungen $Q(y_t|Y_{1:t-1})$ für Teilsequenzen $Y_{1:t}$ sein, um die Problematik der Monte Carlo Simulation zu umgehen.[43]

[1] vlg. Kapitel 4, Abschnitt 4

Anhang A

Weitere Informationen

A.1 Adam-Optimierungsalgorithmus

Eine große Klasse von gradientenbasierten Verfahren nutzt die Stochastic Gradient Descent Methode, bei der der echte Gradient aus Performanzgründen mit einem Bruchteil der echten Gradienten approximiert wird.[15]$^{S.290ff.}$ Im experimentellen Teil der Arbeit wird ein Algorithmus dieser Klasse, der Adam-Optimierungsalgorithmus verwendet, der zur Approximation sogenannte Momente nutzt. Der Name ergibt sich aus dem englischen Begriff „Adaptive Momentum Estimation“. Beim reinen Backpropagation-Algorithmus ist die Lernrate α statisch und gilt global für alle Parameter des Netzes (vlg. Kapitel 2, Abschnitt 2). Im Unterschied dazu wird beim Adam-Optimierungsalgorithmus für jeden Parameter separat eine Lernrate gespeichert und im Laufe des Optimierungsverfahrens mit Hilfe der geschätzten Gradienten aktualisiert.[27]

Das n-te Moment m_n einer Zufallsvariable X ist definiert als der Erwartungswert der n-ten Potenz der Variable, also[37]$^{S.109ff.}$

$$m_n = \mathbb{E}[X^n]. \tag{A.1}$$

Da die Trainingsdaten für eine Aktualisierung typischerweise aus einem zufällig zusammengestellten Mini-Batch bestehen, ist auch der berechnete Gradient eine Zufallsvariable. Um die Momente des Gradienten abzuschätzen, werden bei jedem Aktualisierungsschritt zwei gleitende Mittelwerte nachgehalten:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla e_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla e_t)^2, \end{aligned} \tag{A.2}$$

wobei es sich bei β_1 und β_2 um zwei Parameter handelt, die die Gradienten vorangegangener Schritte t umso weniger gewichten, je weiter sie zurückliegen. Sie werden typischerweise mit 0.9 und 0.999 initialisiert.[27] Werden m_t und v_t mit 0 initialisiert, lassen sie sich am Zeitschritt t auch mit

$$m_t = (1 - \beta_1) \sum_{i=1}^{t} \beta_2^{t-i} \cdot (\nabla e_i)^2 \tag{A.3}$$

schreiben. Durch die Initialisierung mit 0 wird dieser Schätzwert allerdings verzerrt, bzw. „biased".[27] Der mathematische Bias eines Schätzwertes $\hat{x}_m$ für einen echten Wert x ist wie folgt definiert:[15]$^{S.122ff.}$

$$bias(\hat{x}_m) = \mathbb{E}[\hat{x}_m] - x. \tag{A.4}$$

Ein Schätzwert wird „unbiased" genannt, wenn $bias(\hat{x}_m) = 0$, bzw. auch asymptotisch unbiased, wenn $lim_{m\to\infty} bias(\hat{x}_m) = 0$ gilt.[15]$^{S.122ff.}$ Für den gleitenden Mittelwert des ersten Moments m_t gilt

$$\begin{aligned} \mathbb{E}[m_t] &= \mathbb{E}[(1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} \nabla e_i] \\ &= \mathbb{E}[\nabla e_t](1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} + E \\ &= \mathbb{E}[\nabla e_t](1 - \beta_1^t) + E, \end{aligned} \tag{A.5}$$

wobei E ein zu vernachlässigender, da hinreichend kleiner Fehlerwert ist, der durch das Abschätzen des Teilterms entsteht.[27] Damit gilt auch

$$bias(m_t) = \mathbb{E}[m_t] - \mathbb{E}[\nabla e_t] \neq 0, \tag{A.6}$$

der Mittelwert ist also biased. Es wird daher ein korrigierter Schätzwert

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{A.7}$$

verwendet. Das Gleiche gilt analog für das zweite Moment des Gradienten und v_t.[27] Die Aktualisierungsregel des Adam-Optimierungsalgorithmus lautet für einzelne Gewichte dann wie folgt:

$$w_t = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{A.8}$$

für einen kleinen Wert ϵ, der die Division durch 0 verhindern soll.[27] Insgesamt ergibt sich damit Algorithmus A.1. Empirisch hat sich der Algorithmus gegenüber anderen Optimierungsmethoden größtenteils als sehr performant erwiesen.[38]

A.2 Absolute Häufigkeiten von Zeichen im Trainingsdatensatz

Die gezählten Häufigkeiten sind in Abbildung A.1 zu sehen. Es handelt sich dabei ausdrücklich nicht um die tatsächlichen Häufigkeiten in den gescannten Dokumenten. Die Zählung ist mit Hilfe von regulären Ausdrücken erfolgt. Bei der Zählung der Variablen und Zahlen

Algorithmus A.1 Simplifizierter Adam-Optimierungsalgorithmus. Quelle: [27], frei übersetzt aus dem Englischen.

Eingabe: α: Schrittgröße

Eingabe: $\beta_1, \beta_2 \in [0, 1)$: Verringerungsraten für die Schätzung der Momente

Eingabe: $e(\theta)$: Zielfunktion mit Parametern θ

Eingabe: θ_0: Initialer Parametervektor

procedure ADAM
    $m_0 \leftarrow 0$ ▷ Initialisiere Vektor des ersten Moments
    $v_0 \leftarrow 0$ ▷ Initialisiere Vektor des zweiten Moments
    $t \leftarrow 0$
    while θ_t nicht konvergiert **do**
        $t \leftarrow t + 1$
        $g_t \leftarrow \nabla_\theta e_t(\theta_{t-1})$ ▷ Gradient hinsichtlich der Zielfunktion
        $m_t \leftarrow \beta_1 \cdot m_{t-1} + (-\beta_1) \cdot g_t$
        $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
        $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t)}$
        $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
        $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$
    end while
    return θ_t
end procedure

```
|root: 12262        |fac: 18579         |max: 3050          |min: 4839
|argmax: 143        |argmin: 176        |inverse: 0         |sin: 5556
|cos: 4738          |tan: 1730          |sinh: 0            |cosh: 0
|tanh: 0            |sigmoid: 771       |transpose: 2936    |prime: 0
|absolute: 2750     |norm: 0            |mathbbe: 1340      |mathbbp: 611
|maxsub: 0          |minsub: 0          |argmaxsub: 0       |argminsub: 0
|mathbbesub: 0      |mathbbpsub: 0      |add: 120829        |sub: 171736
|dot: 16222         |cross: 10982       |fract: 88231       |mod: 1845
|power: 286494      |derive: 0          |sum: 23000         |product: 2805
|integral: 18661    |equals: 118363     |lesser: 122660     |greater: 29547
|lessereq: 0        |greatereq: 0       |subset: 1360       |subseteq: 0
|union: 1741        |difference: 2702   |elementof: 64181   |apply: 2363
|brackets: 8811     |infty: 142         |propto: 5213       |negate: 4
|Γ: 8151            |Δ: 9324            |Θ: 2019
|Λ: 4211            |Ξ: 876             |Π: 2480            |Σ: 4294
|Υ: 276             |Φ: 6222            |Ψ: 3726            |Ω: 7280
|α: 27499           |β: 17447           |γ: 14968           |δ: 15515
|ε: 13213           |ζ: 4003            |η: 10040           |θ: 17324
|ι: 666             |κ: 4402            |λ: 17351           |μ: 22116
|ν: 26122           |ξ: 22812           |π: 19493           |ρ: 15482
|σ: 16256           |τ: 15202           |υ: 282             |φ: 21414
|x: 7327            |ψ: 11270           |ω: 11103           |A: 42215
|B: 44236           |C: 34264           |D: 29140           |E: 51069
|F: 37382           |G: 20181           |H: 22173           |I: 42861
|J: 11104           |K: 16037           |L: 35043           |M: 33731
|N: 33657           |O: 11291           |P: 33969           |Q: 12442
|R: 34681           |S: 39629           |T: 38175           |U: 13131
|V: 24321           |W: 12460           |X: 20546           |Y: 8589
|Z: 10564           |a: 549032          |b: 239352          |c: 218361
|d: 250922          |e: 496029          |f: 205538          |g: 194518
|h: 220117          |i: 359190          |j: 42719           |k: 93388
|l: 307097          |m: 345521          |n: 308710          |o: 264560
|p: 163120          |q: 95492           |r: 431672          |s: 271147
|t: 581265          |u: 120760          |v: 85764           |w: 85586
|x: 120201          |y: 90568           |z: 31984           |1: 226659
|2: 212862          |3: 47881           |4: 33564           |5: 18054
|6: 13058           |7: 8024            |8: 10045           |9: 5954
|0: 98352
```

Abbildung A.1: Die gezählten absoluten Häufigkeiten der Operatoren auf einem Satz von 2000 wissenschaftlichen Arbeiten aus dem Bereich Maschinelles Lernen von arxiv.org.

beispielsweise können diese Teil eines Wortes oder einer anderen Zahl gewesen sein. Die Scans sind dabei nach Priorität zunächst für zusammenhängende Werte wie „max“ erfolgt, die Buchstaben m, a und x wären in der Folge nicht mehr als Variablen gezählt worden. Bei vielen Operatoren sind die Zählungen auch dadurch verfälscht, dass sie in vielfältiger-weise in LaTeX-Code implementiert werden können. So kann *log* beispielsweise mit $\backslash log$, *log* oder einem eigens definierten Befehl $\backslash xyz$ geschrieben werden.

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Ailon, Nir und Mehryar Mohri: *An efficient reduction of ranking to classification.* CoRR, abs/0710.2889, 2007.

[2] Bachman, Philip und Doina Precup: *Data Generation as Sequential Decision Making.* CoRR, abs/1506.03504, 2015.

[3] Bahdanau, Dzmitry, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron C. Courville und Yoshua Bengio: *An Actor-Critic Algorithm for Sequence Prediction.* CoRR, abs/1607.07086, 2016.

[4] Bauer, Goos: *Informatik, Eine einführende Übersicht*, Band 1. Springer-Verlag, Berlin, 1982. S. 224.

[5] Bengio, Y., P. Simard und P. Frasconi: *Learning long-term dependencies with gradient descent is difficult.* IEEE Transactions on Neural Networks, 5(2):157–166, March 1994.

[6] Che, Tong, Yanran Li, Ruixiang Zhang, R. Devon Hjelm, Wenjie Li, Yangqiu Song und Yoshua Bengio: *Maximum-Likelihood Augmented Discrete Generative Adversarial Networks.* CoRR, abs/1702.07983, 2017.

[7] Cho, Kyunghyun, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk und Yoshua Bengio: *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* CoRR, abs/1406.1078, 2014.

[8] Chomsky, N.: *Three models for the description of language.* IRE Transactions on Information Theory, 2(3):113–124, Sep. 1956.

[9] Chung, Junyoung, Çaglar Gülçehre, KyungHyun Cho und Yoshua Bengio: *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.* CoRR, abs/1412.3555, 2014.

[10] Ciresan, Dan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella und Jürgen Schmidhuber: *Flexible, High Performance Convolutional Neural Networks for Image Classification.* Seiten 1237–1242, 07 2011.

[11] Doersch, Carl: *Tutorial on Variational Autoencoders*, 2016.

[12] Fedus, William, Ian Goodfellow und Andrew M. Dai: *MaskGAN: Better Text Generation via Filling in the.* In: *International Conference on Learning Representations*, 2018.

[13] Glorot, Xavier und Yoshua Bengio: *Understanding the difficulty of training deep feedforward neural networks.* In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[14] Glynn, Peter W.: *Likelihood Ratio Gradient Estimation for Stochastic Systems.* Commun. ACM, 33(10):75–84, Oktober 1990.

[15] Goodfellow, Ian, Yoshua Bengio und Aaron Courville: *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[16] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville und Yoshua Bengio: *Generative Adversarial Nets.* In: Ghahramani, Z., M. Welling, C. Cortes, N. D. Lawrence und K. Q. Weinberger (Herausgeber): *Advances in Neural Information Processing Systems 27*, Seiten 2672–2680. Curran Associates, Inc., 2014.

[17] Goodfellow, Ian J.: *NIPS 2016 Tutorial: Generative Adversarial Networks.* CoRR, abs/1701.00160, 2017.

[18] Greensmith, Evan, Peter L. Bartlett und Jonathan Baxter: *Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning.* In: *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, Seite 1507–1514, Cambridge, MA, USA, 2001. MIT Press.

[19] Harris, David und Sarah Harris: *Digital Design and Computer Architecture, Second Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd Auflage, 2012. S.129.

[20] Hjelm, Devon R, Athul Paul Jacob, Tong Che, Kyunghyun Cho und Yoshua Bengio: *Boundary-Seeking Generative Adversarial Networks.* arXiv e-prints, abs/1702.08431, 2017.

[21] Hochreiter, Sepp, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber et al.: *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*, 2001.

[22] Hochreiter, Sepp und Jürgen Schmidhuber: *Long Short-term Memory.* Neural computation, 9:1735–80, 12 1997.

[23] Hung, Christina und Brendan Corcoran: *What's good for the goose is good for the GANder.*

[24] Jiang, Tao, Ming Li, Bala Ravikumar und Kenneth W. Regan: *Formal Grammars and Languages*, Seite 20. Chapman & Hall/CRC, 2 Auflage, 2010.

[25] Jozefowicz, Rafal, Wojciech Zaremba und Ilya Sutskever: *An Empirical Exploration of Recurrent Network Architectures.* In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, Seite 2342–2350. JMLR.org, 2015.

[26] Kawthekar, Prasad, Raunaq Rewari und Suvrat Bhooshan: *Evaluating generative models for text generation*, 2017.

[27] Kingma, Diederik P. und Jimmy Ba: *Adam: A Method for Stochastic Optimization*, 2014. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[28] Lecun, Y., L. Bottou, Y. Bengio und P. Haffner: *Gradient-based learning applied to document recognition.* Proceedings of the IEEE, 86(11):2278–2324, Nov 1998.

[29] Li, Jiwei, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter und Dan Jurafsky: *Adversarial Learning for Neural Dialogue Generation.* In: Palmer, Martha, Rebecca Hwa und Sebastian Riedel (Herausgeber): *EMNLP*, Seiten 2157–2169. Association for Computational Linguistics, 2017.

[30] Lucic, Mario, Karol Kurach, Marcin Michalski, Olivier Bousquet und Sylvain Gelly: *Are GANs Created Equal? A Large-Scale Study.* In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, Seite 698–707, Red Hook, NY, USA, 2018. Curran Associates Inc.

[31] Lukas Pfahler, Jonathan Schill und Katharina Morik: *The Search for Equations – Learning to Identify Similarities between Mathematical Expressions.* Machine Learning and Knowledge Discovery in Databases. ECML. PKDD, 2019.

[32] Mescheder, Lars, Sebastian Nowozin und Andreas Geiger: *The Numerics of GANs.* In: *Proceedings from the conference Neural Information Processing Systems 2017.* Curran Associates, Inc., Dezember 2017.

[33] Million, E.: *The Hadamard Product Elizabeth Million April 12 , 2007 1 Introduction and Basic Results*. 2007.

[34] Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver und Koray Kavukcuoglu: *Asynchronous Methods for Deep Reinforcement Learning*. CoRR, abs/1602.01783, 2016.

[35] Moise, E.E.: *Calculus*. Nummer Teil 2 in *Addison-Wesley series in mathematics*. Addison-Wesley Publishing Company, 1967.

[36] Øland, Anders, Aayush Bansal, Roger B. Dannenberg und Bhiksha Raj: *Be Careful What You Backpropagate: A Case For Linear Output Activations & Gradient Boosting*. CoRR, abs/1707.04199, 2017.

[37] Papoulis, Athanasios und S. Unnikrishna Pillai: *Probability, Random Variables, and Stochastic Processes*. McGraw Hill, Boston, Fourth Auflage, 2002.

[38] Ruder, Sebastian: *An overview of gradient descent optimization algorithms*, 2016. cite arxiv:1609.04747Comment: 12 pages, 6 figures.

[39] Rumelhart, David E., Geoffrey E. Hinton und Ronald J. Williams: *Learning Representations by Back-Propagating Errors*, Seite 696–699. MIT Press, Cambridge, MA, USA, 1988.

[40] Salimans, Tim, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford und Xi Chen: *Improved techniques for training gans*. In: *Advances in neural information processing systems*, Seiten 2234–2242, 2016.

[41] Sutton, Richard S. und Andrew G. Barto: *Reinforcement Learning: An Introduction*. The MIT Press, Second Auflage, 2018.

[42] Sutton, Richard S, David A. McAllester, Satinder P. Singh und Yishay Mansour: *Policy Gradient Methods for Reinforcement Learning with Function Approximation*. In: Solla, S. A., T. K. Leen und K. Müller (Herausgeber): *Advances in Neural Information Processing Systems 12*, Seiten 1057–1063. MIT Press, 2000.

[43] Tuan, Yi-Lin und Hung-yi Lee: *Improving Conditional Sequence Generative Adversarial Networks by Stepwise Evaluation*. CoRR, abs/1808.05599, 2018.

[44] Wen, Tsung-Hsien, Milica Gasic, Nikola Mrksic, Pei-hao Su, David Vandyke und Steve J. Young: *Semantically Conditioned LSTM-based Natural Language Generation for Spoken Dialogue Systems*. CoRR, abs/1508.01745, 2015.

[45] Williams, Ronald J.: *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning.* Mach. Learn., 8(3-4):229–256, Mai 1992.

[46] Wu, Yuxin und Yuandong Tian: *Training Agent for First-Person Shooter Game with Actor-Critic Curriculum Learning.* In: *ICLR*, 2017.

[47] Yin, Wenpeng, Katharina Kann, Mo Yu und Hinrich Schütze: *Comparative Study of CNN and RNN for Natural Language Processing*, 2017. cite arxiv:1702.01923Comment: 7 pages, 11 figures.

[48] Yu, Lantao, Weinan Zhang, Jun Wang und Yong Yu: *Seqgan: Sequence generative adversarial nets with policy gradient.* In: *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[49] Zhang, Jiawei: *Gradient Descent based Optimization Algorithms for Deep Learning Models Training.* CoRR, abs/1903.03614, 2019.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 6. Januar 2020

Jan-Philip Richter