Universität Rostock

Traditio et Innovatio

Visuelle Aufmerksamkeitsmodelle basierend auf neuronalen Netzwerken

Masterarbeit

Mathematisch-Naturwissenschaftliche Fakultät

Institut für Mathematik

Name:	Johannes Michael
Matrikelnummer:	211203921
Betreuer und Gutachter:	Dipl.-Math. Tobias Grüning
Gutachter:	Prof. Dr. Roger Labahn
Abgabedatum:	16.09.2016

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Als Teilgebiete des maschinellen Lernens bieten die Bild- und Mustererkennung zahllose Anwendungsmöglichkeiten, wenn es darum geht, in einer Menge von Daten Regelmäßigkeiten, Ähnlichkeiten oder Muster zu erkennen. Sei es Spracherkennung, Texterkennung, Gesichtserkennung oder diverse Klassifikationsprobleme, man ist stets auf der Suche nach zuverlässigen und effektiven Algorithmen, um solche Aufgaben zu lösen. Computersysteme, die aus Erfahrung und vorgegebenen Daten lernen, ohne explizit dafür programmiert zu sein, eröffnen Möglichkeiten für andauernde Forschung und entwickeln sich ständig weiter.
Ein etablierter Zweig der künstlichen Intelligenz sind *künstliche neuronale Netzwerke*. Vor allem Faltungsnetzwerke haben große Fortschritte erzielt und waren erst kürzlich sehr erfolgreich bei diversen Aufgaben der Bild- und Mustererkennung ([9], [14], [19], [28]). Ein bestehender Nachteil von Faltungsnetzwerken ist, dass ihre Berechnungskosten schlecht mit einer zunehmenden Bildgröße skalieren. Den Hauptanteil verursachen die Faltungen am Eingabebild, wodurch die Kosten mindestens linear mit der Anzahl der Pixel ansteigen. Trotz ihrer exzellenten Performance leiden sie also zunehmend unter hohen Kosten, sowohl während des Lernprozesses, als auch bei ihrer Nutzung, da die Herangehensweise bei großen Bildern und komplexeren Erkennungsaufgaben, wie Sequenzen mehrerer Objekte, meist durch das *Sliding-Window Paradigma* bestimmt ist. Dabei werden die Systeme nur darauf trainiert, Objekte in relativ kleinen Bildausschnitten zu erkennen, um dann unabhängig auf viele potentielle Fenster des eigentlichen Bildes angewandt zu werden. Außerdem werden Informationen aus vorigen Schritten nur sehr begrenzt genutzt, um zukünftige Verarbeitungen des Bildes zu beeinflussen.
Motiviert durch die Publikationen von Mnih et al. [24] und Ba et al. [1] war es das Ziel dieser Arbeit, eine global trainierbare Architektur aufzubauen, welche sowohl die Lokalisierung, als auch die Erkennung von Objekten ermöglicht und effizient skaliert. Als Inspiration diente die Art und Weise der menschlichen Wahrnehmung. Anstatt dass wir beispielsweise eine gesamte Szene mit einem Blick einfangen, verarbeiten wir diese sequentiell, indem wir unsere *Aufmerksamkeit* selektiv auf bestimmte Teile fokussieren, Informationen sammeln, diese nach und nach kombinieren und eine interne Repräsentation der Szene aufbauen, welche unsere zukünftigen Entscheidungen beeinflusst [27]. Wenn man es schaffen würde, dass ein System diesen Prozess simuliert, so würde man einerseits Kosten sparen, da immer nur ein ausgewählter Bereich des Bildes verarbeitet wird und andererseits würde es die Komplexität des Problems reduzieren, da irrelevante Merkmale außerhalb des Fokus automatisch ignoriert werden. Daraus entstand das Konzept eines aufmerksamkeitsbasierten Systems zur zielgerichteten visuellen Bildverarbeitung mit neuronalen Netzen.
Den Kern des *Aufmerksamkeitsmodells* bildet also ein neuronales Netz, welches verschie-

dene Bildausschnitte sequentiell als Eingaben verarbeitet und die extrahierten Informationen kombiniert, um eine dynamische interne Repräsentation der Szene aufzubauen. Abhängig vom aktuellen internen Zustand des Netzes, entscheidet dieses, welche Position als nächste fokussiert werden soll. Das Aufmerksamkeitsmodell ist so konstruiert, dass die Anzahl der Parameter und jegliche Berechnungskosten unabhängig von der Größe der Eingabebilder sind, was ein großer Vorteil im Kontrast zu Faltungsnetzwerken ist (vergleiche [24]). Es wird ein Verfahren aufgebaut, welches ein globales Training der gesamten Architektur ermöglicht, um ein problemspezifisches Ziel zu optimieren. Dabei werden einerseits für das *überwachte Lernen* typische Fehlerterme durchs Netz zurück propagiert und andererseits nutzt man Konzepte aus dem *bestärkenden Lernen*, um nicht differenzierbare Anteile einzubeziehen. Das Modell wird an verschiedenen Datenmengen im Kontext statischer Klassifikations- und Regressionsprobleme getestet, um zu sehen, welche Art von Strategien es im Stande ist zu lernen.

2 Neuronale Netzwerke

Künstliche neuronale Netzwerke wurden ursprünglich als biologisch inspiriertes mathematisches Modell der Informationsverarbeitung des menschlichen Gehirns entwickelt. Sie stellen einen Zweig der künstlichen Intelligenz dar und versuchen einen Einblick in den Aufbau und die Funktionsweise eines Gehirns zu gewähren, indem seine elementaren Bestandteile modelliert werden und so eine Struktur erstellt wird, die in der Lage ist zu lernen. Das Gehirn ist abstrakt gesehen ein hochkomplexes, nicht-lineares, parallel arbeitendes Informationsverarbeitungssystem. Im Wesentlichen besteht es aus untereinander verbundenen Neuronen, deren Verbindungen unterschiedlich stark ausgeprägt sind und über welche die Neuronen in der Lage sind, Impulse aneinander zu senden. Anhand dieses Vorbilds wird ein Netz aus *künstlichen Neuronen* und Vernetzungen erstellt, dass die Wechselwirkung von Neuronen auf abstrakter Ebene realisiert. Für den Rest dieser Arbeit setzen wir die Bezeichnung eines künstlichen neuronalen Netzes mit der eines neuronalen Netzes gleich. Grundsätzlich besteht ein neuronales Netz aus mehreren kleinen Verarbeitungseinheiten, den künstlichen Neuronen, welche mittels *gewichteter Verbindungen* miteinander verbunden sind. Aktiviert wird das Netz, indem einigen Neuronen Eingaben zur Verfügung gestellt werden, welche sich daraufhin entlang der Verbindungen durch das Netz verteilen und bei einigen Neuronen Ausgaben erzeugen. Neuronale Netze können dazu genutzt werden, eine breite Spanne an Aufgaben, wie Zeitreihenanalysen, Bildverarbeitungen und Mustererkennungen zu lösen.
Mit der Zeit haben sich verschiedene Variationen von neuronalen Netzwerken herausgebildet. Auf einige wird im Folgenden eingegangen. In diesem Kapitel werden zunächst wichtige Grundlagen für neuronale Netze eingeführt und anschließend ausgewählte Netzarchitekturen vorgestellt, die für das in dieser Arbeit aufgebaute Aufmerksamkeitsmodell relevant sind. Praxisbezogene Anpassungen und Optimierungen für neuronale Netze sollen das Kapitel abschließen.

2.1 Grundlagen

Dieser Abschnitt umfasst die grundlegenden Bausteine künstlicher neuronaler Netze und wie man diese zu einem neuronalen Netz zusammensetzen kann. Weiterhin werden erste Gedanken aufgeführt, auf welche Arten Maschinen etwas lernen können.

2.1.1 Neuronen und Verbindungen

Bevor wir uns mit dem strukturellen Aufbau und der Funktionsweise diverser neuronaler Netze beschäftigen können, muss zunächst geklärt werden, wie die künstlichen

Neuronen innerhalb des Netzes modelliert sind. Jedes Neuron stellt eine einfache Recheneinheit dar, welche fundamental für die Funktionsweise des Netzes ist. Jedes Neuron erhält eine bestimmte Anzahl von Eingaben x_i, denen entlang der Verbindungen je ein *Gewicht* w_i zugeordnet ist. Jedes eingehende Signal wird dabei mit seinem entsprechenden Gewicht multipliziert und anschließend mit allen anderen Werten aufsummiert, um die *Netzeingabe* a des Neurons zu bilden.

Definition 2.1 (Netzeingabe)**.** Es sei $\{x_1, \ldots, x_n\}$ die Menge der Eingaben eines Neurons, deren Verbindungen durch die Werte $w_1, \ldots, w_n$ gewichtet sind. Dann berechnet sich die *Netzeingabe* a des Neurons durch die gewichtete Summe

$$a = \sum_{i=1}^{n} w_i x_i. \tag{2.1}$$

Diese Summe wird an eine *Aktivierungsfunktion* $f(\cdot)$ weitergegeben, welche die eigentliche Ausgabe b, die *Aktivierung* des Neurons erzeugt. Die eingehende Summe wird dabei je nach Form der Aktivierungsfunktion (siehe Kapitel 2.1.3) in ein nützliches Ausgabesignal umgewandelt. Im einfachsten Fall handelt es sich dabei um eine Stufenfunktion, die nur die Werte 0 oder 1 annehmen kann. Die Ausgabe lässt sich somit als Feuern oder Nicht-Feuern des biologischen Neurons interpretieren.

Definition 2.2 (Aktivierungsfunktion und Aktivierung)**.** Die *Aktivierungsfunktion* f verarbeitet die Netzeingabe a eines Neurons, um dessen *Aktivierung* b zu bestimmen:

$$b = f(a). \tag{2.2}$$

Üblicherweise wird von der Summe noch ein *Schwellenwert* θ abgezogen, bevor sie der Aktivierungsfunktion übergeben wird. Dieser stellt biologisch gesehen die Reizschwelle dar, ab der ein Neuron feuert. Der Einfachheit halber kann man den Schwellenwert aber auch durch einen *Bias* ersetzen, indem man dem Neuron eine zusätzliche konstante Eingabe x_0 mit Wert 1 und Gewichtung $w_0 = -\theta$ hinzufügt. Die Funktionsweise eines künstlichen Neurons kann dann folgendermaßen festgehalten werden:

$$a = \sum_{i=0}^{n} w_i x_i \tag{2.3}$$

$$b = f(a). \tag{2.4}$$

Damit entspricht es einem deterministischem Eingabe-Ausgabe-Verhalten. Für einige Anwendungen von neuronalen Netzen kann es gewünscht sein, Berechnungen auf stochastischen Neuronen-Modellen durchzuführen, was für diese Arbeit aber nicht weiter von Relevanz ist. Abbildung 2.1 zeigt das allgemeine Modell eines künstlichen Neurons, auch bekannt als *einfaches Perzeptron*.

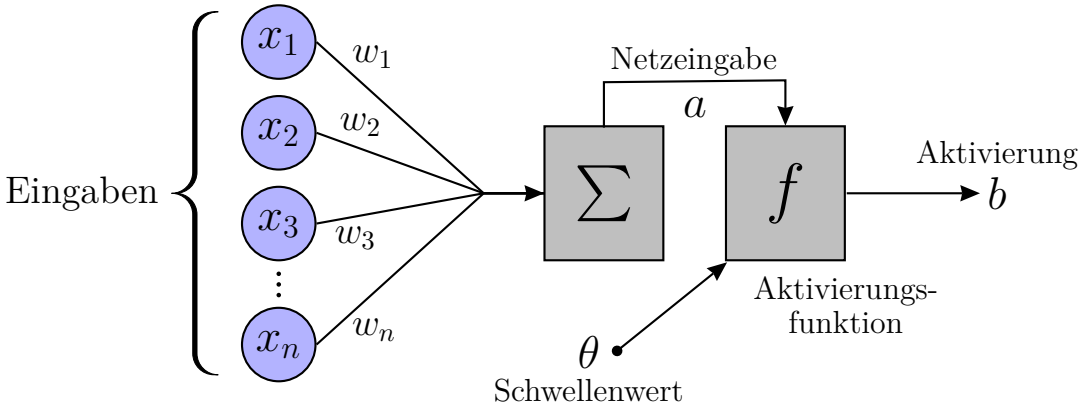


Abbildung 2.1: Modell eines künstlichen Neurons. Jeder Eingabe x_i ist ein Gewicht w_i zugeordnet. Eine Summationseinheit berechnet die Netzeingabe a des Neurons, welche an die Aktivierungsfunktion f übergeben wird, um die Aktivierung b zu berechnen.

2.1.2 Netztopologien

Wie bereits besprochen, besteht ein neuronales Netz aus einer Menge künstlicher Neuronen und Verbindungen.

Definition 2.3 (Neuronales Netz)**.** Ein (*künstliches*) *neuronales Netz* ist ein Tripel (N, V, w) mit zwei Mengen N, V sowie einer Funktion w, wobei $N \subset \mathbb{N}$ die Menge der Neuronen bezeichnet und $V \subset N^2$ eine Menge von *Verbindungen* zwischen den Neuronen angibt. Die Funktion $w\colon V \to \mathbb{R}$ definiert die *Gewichte*, wobei $w(i, j)$, das Gewicht der Verbindung von Neuron i zu Neuron j, kurz mit w_{ij} bezeichnet wird.

Bemerkung 2.1. Man kann die Menge V in Definition 2.3 auch auf $V = N^2$ erweitern und nicht vorhandene Verbindungen mit einem 0-Gewicht identifizieren. Die Gewichtungen lassen sich dann in einer quadratischen *Gewichtsmatrix* W oder einem *Gewichtsvektor* W implementieren.

Bemerkung 2.2. Neuronenspezifische Größen werden durch einen Index angegeben. Beispielsweise ist die Netzeingabe von Neuron j mit a_j bezeichnet. Gleichermaßen verhält es sich mit der Aktivierungsfunktion f_j und der Aktivierung b_j.

Um ein funktionierendes Netzwerk aufzubauen und ein Training zu ermöglichen, muss eine eindeutige Struktur unter den Neuronen festgelegt werden. Man gruppiert beliebige Mengen von Neuronen und ordnet diese in hintereinander liegende Schichten an: Es gibt eine *Eingabeschicht* (input layer), in der externe Eingaben ins Netz gefüttert werden, eine *Ausgabeschicht* (output layer), bei der die Aktivierungen der Neuronen als Ausgaben des Netzes interpretiert werden und eine beliebige Anzahl *verborgener Schichten* (hidden layers) zwischen der Eingabe- und Ausgabeschicht. Die Topologie eines Netzes beschreibt im Allgemeinen, wie viele künstliche Neuronen sich auf wie vielen Schichten befinden und wie diese miteinander verbunden sind.
Lässt man nur Verbindungen zwischen Neuronen einer Schicht und der direkt benachbarten zu (in Richtung der Ausgabeschicht), so spricht man von *Feed-Forward-Netzen*.

Verbindungen von Neuronen innerhalb einer Schicht sind dabei nicht gestattet. Oft begegnet man Fällen, in denen jedes Neuron eine Verbindung zu allen Neuronen der Folgeschicht besitzt.

Definition 2.4 (Feed-Forward-Netz)**.** Ein *Feed-Forward-Netz* (feedforward neural network, FNN) ist ein neuronales Netz, dessen Neuronen in klar abgetrennte Schichten eingeteilt sind: Eine Eingabeschicht, eine Ausgabeschicht und beliebig viele verborgene Schichten. Dabei sind nur vorwärtsgerichtete Verbindungen erlaubt, das heißt Verbindungen zu Neuronen der jeweils nächsten Schicht.

Definition 2.5 (Voll-verknüpfte Schichten)**.** Sind die Neuronen einer Schicht zu allen Neuronen der Folgeschicht verbunden, so nennt man die beiden beteiligten Schichten *voll-verknüpft* beziehungsweise *voll-verbunden*.

Die Frage nach der optimalen Anzahl an Schichten und Neuronen jeder Schicht hängt von der Problemstellung ab. Meist muss die Architektur experimentell bestimmt werden, da es keine pauschalen Lösungen gibt. Ebenso ist anzumerken, dass unterschiedliche Aktivierungsfunktionen in verschiedenen Schichten zur Anwendung kommen können. In Abbildung 2.2 ist ein einfaches FNN mit einer verborgenen Schicht abgebildet (in Form eines gerichteten azyklischen Graphen).

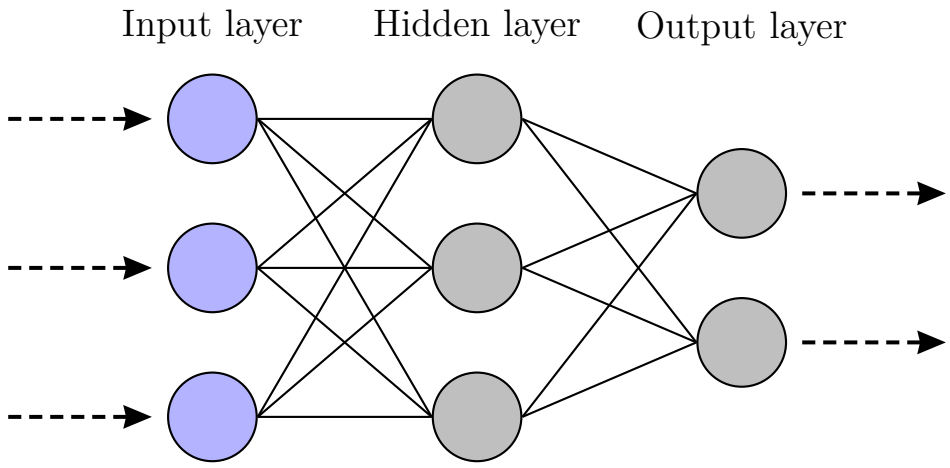


Abbildung 2.2: Modell eines einfachen Feed-Forward-Netzes mit einer verborgenen Schicht. Die Eingaben werden ausgehend von der Eingabeschicht vorwärts über die verborgene Schicht und bis zur Ausgabeschicht durchs Netz propagiert.

Hier wird auch erkenntlich, warum diese Art von Netz Feed-Forward-Netz genannt wird, denn die Eingaben werden stets nur vorwärts von Schicht zu Schicht durchs Netz propagiert.

Definition 2.6 (Vorwärtsphase)**.** Der Prozess, bei dem die Netzeingaben verarbeitet und Schicht für Schicht durchs Netzwerk propagiert werden, bis die Ausgabeschicht die Netzausgaben berechnet, wird als *Vorwärtsphase* bezeichnet.

Allgemein kann man sich ein FNN als Funktion vorstellen, die den Eingabevektoren bestimmte Ausgabevektoren zuordnet und welche aufgrund der Parametrisierung durch

die Gewichte in der Lage ist, viele verschiedene Funktionen zu modellieren. Dabei ermöglicht die Einführung verborgener Schichten, dem Netz nicht-lineare Funktionen zu approximieren. Tatsächlich wurde bewiesen, dass ein FNN mit nur einer Schicht genügend vieler verborgener nicht-linearer Neuronen jede stetige Funktion mit einem kompakten Definitionsbereich zu beliebiger Genauigkeit approximieren kann [13].
Ausgehend von den Feed-Forward-Netzen scheint es ein natürlicher Schritt zu sein, die Einschränkung der Verbindungen zu lockern. Lässt man *zyklische Verbindungen* zu, wird eine Art Rückkopplung beziehungsweise Rekurrenz in das Netz integriert.

Definition 2.7 (Zyklische Verbindungen)**.** Als *zyklische Verbindungen* bezeichnet man alle nicht vorwärtsgerichteten Verbindungen eines neuronalen Netzwerks. Man unterscheidet dabei:

- Direkte Rückkopplung: Verbindung von einem Neuron zu sich selbst
- Seitliche Rückkopplung: Verbindung zwischen zwei Neuronen derselben Schicht
- Indirekte Rückkopplung: Verbindung von einem Neuron zu einer vorhergehenden Schicht

Definition 2.8 (Rekurrente Schicht und Rekurrentes Netz)**.** Eine *rekurrente Schicht* ist eine Schicht, die neben vorwärtsgerichteten Verbindungen auch zyklische Verbindungen enthält. Besitzt ein neuronales Netz rekurrente Schichten, so wird es *rekurrentes Netz* (recurrent neural network, RNN) genannt.

Wir werden uns auf direkte und seitliche Rückkopplungen beschränken, um Informationen zu zeitlichen Abhängigkeiten innerhalb einer Schicht zu speichern. Abbildung 2.3 zeigt das Modell eines einfachen RNN mit einer rekurrenten verborgenen Schicht.

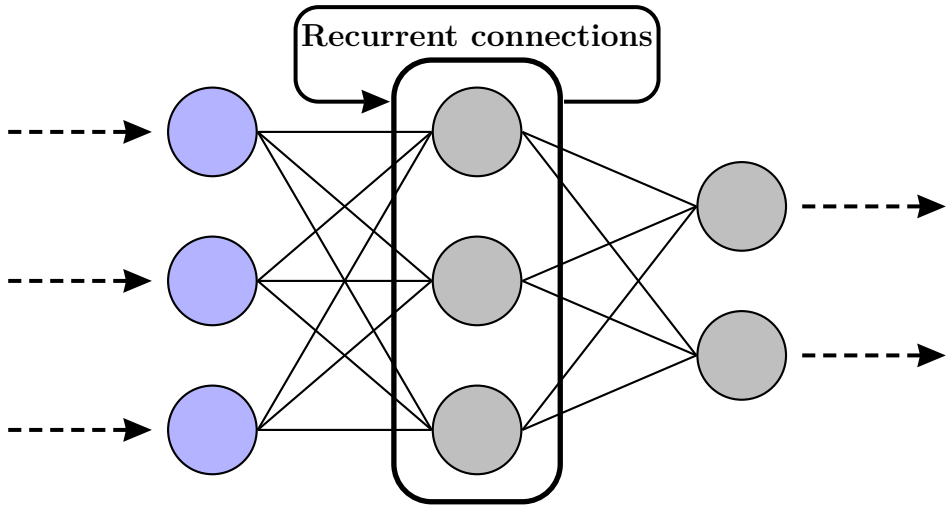


Abbildung 2.3: Modell eines einfachen rekurrenten Netzes. Die verborgenen Neuronen sind mit der gesamten verborgenen Schicht verknüpft.

Im Verlauf dieses Kapitels wird noch genauer auf zwei Vertreter der Feed-Forward-Netze eingegangen: Auf das *Mehrschicht-Perzeptron* als klassisches FNN und auf *Faltungsnetzwerke*, die sich spezieller Techniken aus der digitalen Signalverarbeitung bedienen

und konzeptionell unterschiedliche Arten von Schichten einführen, um besonders gut mit Bilddaten umgehen zu können. Rekurrente Netze werden auch erläutert, da sie die Grundstruktur für das in dieser Arbeit verwendete Aufmerksamkeitsmodell bilden. Die Feed-Forward-Netze werden hingegen für einzeln eingesetzte Komponenten genutzt.

2.1.3 Aktivierungsfunktionen

Die Aktivierungsfunktion $f(a)$ definiert die Ausgabe eines Neurons in Abhängigkeit von seiner Netzeingabe a. Innerhalb eines neuronalen Netzes ist die Aktivierungsfunktion oft global für alle oder zumindest eine Menge von Neuronen (zum Beispiel die einer Schicht) festgelegt. Im Folgenden sollen die gängigsten Varianten vorgestellt werden.

- Die wohl einfachste Aktivierungsfunktion ist die *Schwellenwertfunktion* (auch *Heaviside-Funktion* oder *Stufenfunktion* genannt, Abbildung 2.4a)

$$f(a) = \begin{cases} 1 & \text{wenn } a \geq 0 \\ 0 & \text{wenn } a < 0 \end{cases}, \tag{2.5}$$

  welche nur die Werte 0 oder 1 annehmen kann. Sie wechselt am Schwellenwert von einem Wert zum anderen und ist ansonsten konstant. An dieser Stelle ist sie somit nicht differenzierbar und ansonsten ist die Ableitung 0. Dies macht ein Lernen mittels Fehlerrückführung unmöglich (siehe Kapitel 2.2.3).

- Beliebt ist die *ReLU-Aktivierungsfunktion* (Rectifier Linear Unit, Abbildung 2.4b)

$$f(a) = \max(0, a) = \begin{cases} a & \text{wenn } a \geq 0 \\ 0 & \text{wenn } a < 0 \end{cases}, \tag{2.6}$$

  mit der Ableitung

$$f'(a) = \begin{cases} 1 & \text{wenn } a \geq 0 \\ 0 & \text{wenn } a < 0 \end{cases}. \tag{2.7}$$

  Es gibt Abwandlungen der Form $f(a) = \max(0.01a, a)$, die kleine Gradienten ungleich Null erlauben, wenn das Neuron nicht aktiviert wird. Diese Funktionen heißen *Leaky ReLUs*.

- Oft verwendet man die auch *Logistische Funktion* oder *Fermifunktion* genannte *Sigmoid-Funktion* (Abbildung 2.4c)

$$f(a) = \sigma(a) = \frac{1}{1 + \exp(-\beta a)}, \tag{2.8}$$

  welche einen Wertebereich von $(0, 1)$ aufweist und ein variables Steigungsmaß β besitzt, das die Krümmung des Funktionsgraphen beeinflusst. Damit lässt sich die

Sigmoid-Funktion beliebig an die Schwellenwertfunktion annähern und ist trotzdem differenzierbar mit der Ableitung

$$\sigma'(a) = \beta\sigma(a)(1 - \sigma(a)). \tag{2.9}$$

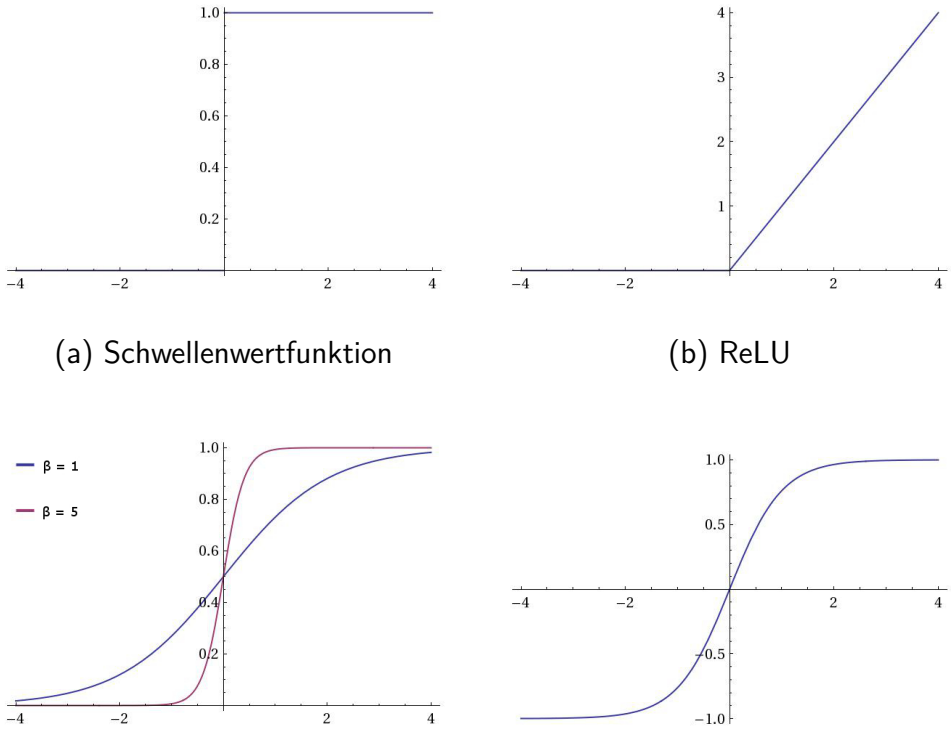
- Eine ähnliche Funktion ist der *Tangens Hyperbolicus* (Abbildung 2.4d)

$$f(a) = \tanh(a) = \frac{\exp(2a) - 1}{\exp(2a) + 1}, \tag{2.10}$$

welcher den Wertebereich auf $(-1, 1)$ erweitert und durch eine einfache Umskalierung und Verschiebung in die Sigmoid-Funktion überführt werden kann: $\tanh(x) = 2\sigma(2x) - 1$. Seine Ableitung lautet

$$\tanh'(a) = 1 - \tanh^2(a). \tag{2.11}$$

Es gibt noch weitere Aktivierungsfunktionen, die vor allem eine Rolle für die Neuronen der Ausgabeschicht spielen, auf die wir aber erst bei passender Gelegenheit eingehen werden.



(a) Schwellenwertfunktion (b) ReLU

(c) Sigmoid-Funktion (d) Tangens Hyperbolicus

Abbildung 2.4: Aktivierungsfunktionen neuronaler Netzwerke.

2.1.4 Paradigmen des Lernens

Die beeindruckendste Eigenschaft neuronaler Netze ist ihre Fähigkeit zum Lernen. Wir erinnern uns daran, dass künstliche neuronale Netze durch das menschliche Gehirn inspiriert wurden. Obwohl man grundlegende Erkenntnisse über das Gehirn in Bezug aufs Lernen hat, ist das Verständnis dieses Prozesses immer noch ziemlich elementar. Es wird angenommen, dass sich während des Lernprozesses die neuronalen Strukturen verändern. Verbindungen zwischen Neuronen werden abhängig von ihrer Aktivität verstärkt oder abgeschwächt, um in Zukunft optimaler zu reagieren. Vereinfacht gesehen, modellieren künstliche neuronale Netze diesen Prozess, indem die Gewichte der Verbindungen im Verlauf des Trainings angepasst werden, um somit bessere Ergebnisse bezüglich eines bestimmten Kriteriums (meist in Form einer Zielfunktion) zu erzielen. Nach ausreichendem Training können auch bis dahin unbekannte Probleme derselben Klasse gelöst werden, was man als *Generalisierung* bezeichnet.
Es gibt viele Trainingsalgorithmen für neuronale Netze, wobei sich ihnen drei verschiedenen Paradigmen des Lernens zuordnen lassen, welche wesentliche Unterschiede in der Art des Trainings aufweisen.

- *Unüberwachtes Lernen* ist biologisch gesehen am plausibelsten, eignet sich praktisch aber nicht für alle Fragestellungen. Hier werden dem Netz beim Training nur Eingabemuster zur Verfügung gestellt. Aufgabe des Netzes ist es nun, ähnliche Muster ohne externe Hilfe zu identifizieren. Unüberwachtes Lernen wird in dieser Arbeit nicht verwendet.

- Beim *überwachten Lernen* werden dem Netz neben den Eingaben auch die zugehörigen (korrekten) Ausgaben zur Verfügung gestellt. Damit ist es möglich, eine Art Fehler zu berechnen, abhängig davon, wie stark die eigentliche Ausgabe des Netzes von der bereitgestellten Ausgabe abweicht. Mithilfe des Fehlers können dann Korrekturen an den Gewichten vorgenommen werden. Dieses Lernverfahren ist biologisch nicht sehr plausibel, aber zielgerichteter und meist sehr praktikabel. Im Regelfall umfasst überwachtes Lernen drei disjunkte Mengen von Paaren (x, t), wobei x ein Trainingsmuster ist und t eine dazugehörige richtige Ausgabe. Die *Trainingsmenge* wird für das eigentliche Training des Netzes verwendet, mit dem Ziel, die aufgabenabhängige Fehlermessung auf der *Testmenge* zu minimieren. Die *Validierungsmenge* dient zur Validierung der Performance des Netzes während des Trainings und die Testmenge dient der Beurteilung des Netzes nach Abschluss des Trainings. Üblicherweise umfasst das Vorgehen zur Fehlerminimierung die Optimierung einer Zielfunktion auf der Trainingsmenge, indem man iterativ die Parameter des Netzwerks anpasst. Die Zielfunktion kann dabei der Fehler selbst sein oder sich auf diesen beziehen. Das Verfahren des überwachten Lernens wird am Beispiel des Mehrschicht-Perzeptrons deutlich werden.

- *Bestärkendes Lernen* ähnelt dem überwachten Lernen derart, dass dem Netz nach Durchlauf eine Rückmeldung über Erfolg oder Misserfolg gegeben wird. Dies erfolgt allerdings nicht durch einen Abgleich mit Referenzwerten, sondern beispielsweise durch Ausgabe eines reellen Wertes, bei dem 0 für falsch und 1 für richtig

steht. Man kann auch einen kontinuierlichen Wert wählen, wenn wichtig ist, „wie“ falsch oder richtig das Netz lag. Das Ziel ist es also, diese Belohnung durch Versuch und Irrtum zu maximieren. Bestärkendes Lernen spielt eine wichtige Rolle für diese Arbeit. Die Thematik wird deswegen in Kapitel 3 noch weiter vertieft.

2.1.5 Stochastisches Klassifizieren

Dieser Abschnitt behandelt eine effektive Herangehensweise zur Klassifizierung im Bereich der Mustererkennung (vergleiche [10, Kapitel 2.2]). Mustererkennung, die Fähigkeit, in einer Menge von Daten Regelmäßigkeiten oder Ähnlichkeiten zu erkennen, wurde bereits ausführlich in der Literatur des maschinellen Lernens dargelegt und diverse Algorithmen werden laufend für kommerzielle Zwecke genutzt. Die folgenden Betrachtungen seien im Rahmen des überwachten Lernens zu verstehen. Ziel ist es, stets die Fehlerrate der Klassifizierung zu minimieren. Eine intuitive Herangehensweise ist, einen Klassifikator zu trainieren, der Trainingsmuster direkt Klassen zuordnet. Ein bekanntes Beispiel hierfür sind die *Stütz-Vektor-Maschinen* (support vector machines).
Eine andere Möglichkeit ist *stochastisches Klassifizieren*, bei dem zunächst die bedingten Wahrscheinlichkeiten $p(C_k \mid x)$ der K verschiedenen Klassen $C_1, \ldots, C_K$, abhängig durch ein Eingabemuster x, bestimmt werden und dann die wahrscheinlichste als Klassifizierung $h(x)$ des Algorithmus interpretiert wird:

$$h(x) = \arg\max_k p(C_k \mid x). \tag{2.12}$$

Das eröffnet die Möglichkeit, mehrere stochastische Klassifikatoren konsistent miteinander zu kombinieren. Außerdem können die Wahrscheinlichkeiten bei Bedarf dazu genutzt werden, verschiedenen Missklassifikationen unterschiedlich hohe Strafen zuzuordnen.
Stochastisches Klassifizieren induziert eine einheitliche Methodik zum Trainieren von Algorithmen. Es sei h_w ein stochastischer Klassifikator mit variablen Parametern w und $p(C_k \mid x, w)$ eine von h_w erzeugte bedingte Wahrscheinlichkeitsverteilung über die Klassen. Weiterhin sei

$$p(S \mid w) = \prod_{(x,t)\in S} p(t \mid x, w) \tag{2.13}$$

ein Produkt über den unabhängig und identisch verteilten Eingabe-Ziel-Paaren (x, t) der Trainingsmenge S. Mithilfe von Bayes' Theorem erhalten wir dann

$$p(w \mid S) = \frac{p(S \mid w)p(w)}{p(S)}. \tag{2.14}$$

Analytisch gesehen lässt sich die A-posteriori-Verteilung der Klassen für eine Eingabe x berechnen, indem man über alle möglichen Parameterkonstellationen w integriert:

$$p(C_k \mid x, S) = \int_w p(C_k \mid x, w)p(w \mid S)dw. \tag{2.15}$$

Allerdings ist w meist sehr hochdimensional und somit die obige Berechnung unpraktisch. Abhilfe verschafft die sogenannte *Maximum-a-posteriori-Schätzung* (MAP). Gesucht ist der Vektor w_{MAP}, welcher (2.14) maximiert und dann benutzt wird, um Vorhersagen über die Wahrscheinlichkeit der Klassen zu treffen:

$$p(C_k \mid x) = p(C_k \mid x, w_{\text{MAP}}). \tag{2.16}$$

Da $p(S)$ unabhängig von w ist, erhält man

$$w_{\text{MAP}} = \arg\max_w p(S \mid w)p(w), \tag{2.17}$$

wobei die A-priori-Verteilung $p(w)$ der Parameter als Regularisierungsterm bezeichnet wird. Er begünstigt die Parameter, die a-priori wahrscheinlicher sind. Nimmt man allerdings eine Gleichverteilung an, dann kann der Term $p(w)$ in (2.17) vernachlässigt werden und man erhält die *Maximum-likelihood-Schätzung* (ML) der Form

$$w_{\text{ML}} = \arg\max_w p(S \mid w) = \arg\max_w \prod_{(x,t)\in S} p(t \mid x, w). \tag{2.18}$$

Standardmäßig lässt sich der Vektor w_{ML} finden, indem man den negativen natürlichen Logarithmus von $p(S \mid w)$ minimiert, da der natürliche Logarithmus monoton steigend ist und die Minimierung von $-\ln p(S \mid w)$ der Maximierung von $p(S \mid w)$ entspricht:

$$w_{\text{ML}} = \arg\min_w -\ln \prod_{(x,t)\in S} p(t \mid x, w) = \arg\min_w -\sum_{(x,t)\in S} \ln p(t \mid x, w). \tag{2.19}$$

Die Maximum-likelihood-Schätzung spielt in Kapitel 2.2.2 eine Rolle, da sie zur Herleitung von Zielfunktionen neuronaler Netze genutzt werden wird.

2.2 Mehrschicht-Perzeptron

Die verbreitetste Form eines FNN ist das sogenannte Mehrschicht-Perzeptron, welches mehr als eine Schicht variabel gewichteter Verbindungen enthält.

Definition 2.9 (Mehrschicht-Perzeptron)**.** Ein FNN mit mindestens einer verborgenen Schicht nennt man *Mehrschicht-Perzeptron* (multilayer perceptron, MLP).

Als FNN sind die Neuronen des MLP in Schichten angeordnet. Die vorwärtsgerichteten Verbindungen propagieren die Signale von Schicht zu Schicht durchs Netz. Im Regelfall handelt es sich bei MLPs um voll-verknüpfte Netze, welche den Prozess der *Fehlerrückführung* (error backpropagation, BP) nutzen, um überwacht zu lernen. Dieser Lernprozess umfasst die bereits erwähnte Vorwärtsphase und im Anschluss eine *Rückwärtsphase*. Letztere nutzt die Ausgaben der Vorwärtsphase, um eine Fehlerkorrektur vorzunehmen. Dabei arbeitet man sich rückwärts, beginnend bei der Ausgabeschicht, durchs Netz.

2.2.1 Vorwärtsphase

Wir betrachten ein MLP mit einer Eingabeschicht der Dimension I, m verborgenen Schichten der Dimensionen $H_1, \ldots, H_m$ und einer Ausgabeschicht der Dimension K (siehe Abbildung 2.5). Eingabemuster werden als Vektor $x = (x_1, \ldots, x_I)$ repräsentiert, Gewichte von Neuron i zu Neuron j werden mit w_{ij} bezeichnet und die Aktivierungsfunktion eines Neurons j sei $f_j(\cdot)$. Einfachheitshalber lassen wir jegliche Bias-Terme im Folgenden aus.

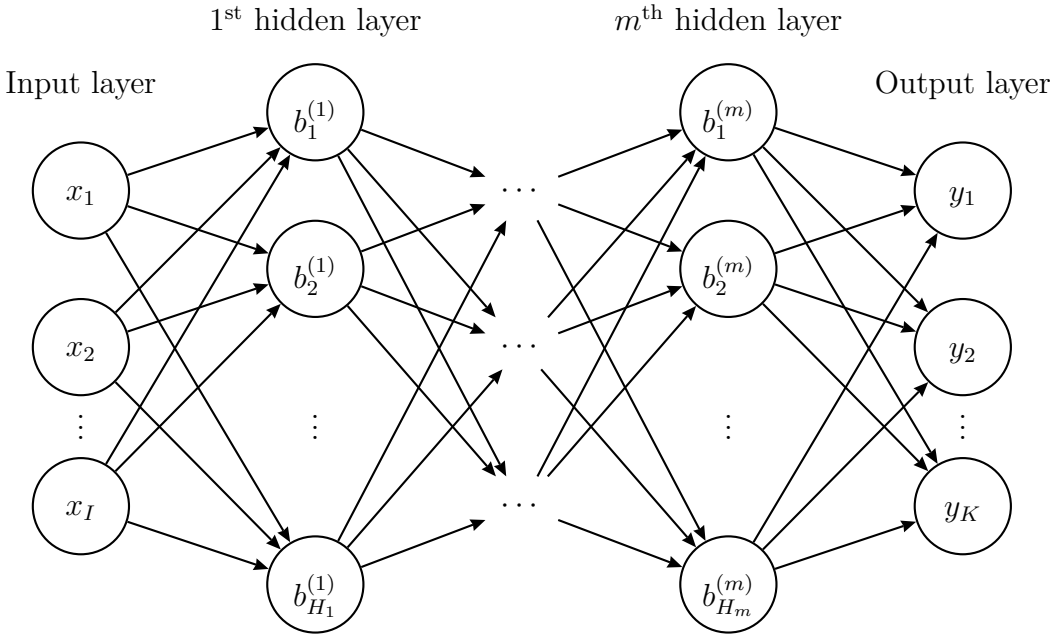


Abbildung 2.5: Ein Mehrschicht-Perzeptron mit I Eingaben, m verborgenen Schichten und K Ausgaben. Die l-te verborgene Schicht enthält H_l Neuronen.

Zunächst berechnet jedes Neuron h der ersten verborgenen Schicht seine Netzeingabe $a_h^{(1)}$ als gewichtete Summe seiner Eingaben und wendet darauf seine Aktivierungsfunktion f_h an, um die Aktivierung $b_h^{(1)}$ zu erzeugen:

$$a_h^{(1)} = \sum_{i=1}^{I} w_{ih} x_i \tag{2.20}$$

$$b_h^{(1)} = f_h(a_h^{(1)}). \tag{2.21}$$

Gebräuchlich für die Aktivierungsfunktionen der verborgenen Schichten sind nicht-lineare, differenzierbare Funktionen wie die Sigmoid-Funktion oder der Tangens Hyperbolicus. Die Nicht-Linearität ist essentiell, da das Netzwerk somit nicht-lineare Klassifikationsgrenzen finden kann und auch in der Lage ist, nicht-lineare Gleichungen zu modellieren.
Mit den Aktivierungen der ersten verborgenen Schicht kann der obige Prozess nun für die zweite verborgene Schicht wiederholt werden und sukzessive auch für alle weiteren.

Für ein beliebiges Neuron h der l-ten verborgenen Schicht ergibt sich also

$$a_h^{(l)} = \sum_{h'=1}^{H_{l-1}} w_{h'h} b_{h'}^{(l-1)} \tag{2.22}$$

$$b_h^{(l)} = f_h(a_h^{(l)}). \tag{2.23}$$

Angekommen bei der Ausgabeschicht lassen sich nun analog die Netzeingaben der Ausgabeneuronen berechnen:

$$a_k = \sum_{h=1}^{H_m} w_{hk} b_h^m. \tag{2.24}$$

Die Aktivierungen $b_k = f_k(a_k)$ der Ausgabeneuronen k werden als Ausgaben des neuronalen Netzes interpretiert und man fasst diese im Ausgabevektor $y = (y_1, \ldots, y_K) = (b_1, \ldots, b_K)$ zusammen. Die Anzahl der Ausgabeneuronen und die Wahl der Aktivierungsfunktion hängt von der Problemstellung ab.
Behandelt man eine Regression, so ist man bei den Ausgaben nicht an einer binären Entscheidung interessiert, welche beispielsweise durch die Sigmoid-Funktion oder den Tangens Hyperbolicus modelliert wird. Vielmehr braucht man einen kontinuierlichen Übergang reeller Zahlen und kann als Aktivierungsfunktion einfach die Identität $f_k(a_k) = a_k$ als lineare Funktion ansetzen. Die Aktivierungsfunktionen der verborgenen Neuronen bleiben dabei unangetastet.
Für Klassifikationsprobleme nutzt man eine andere Herangehensweise. Im binären Fall ($K = 2$) reicht die Konfiguration eines einzelnen Ausgabeneurons mit einer Sigmoid-Aktivierung. Da die Sigmoid-Funktion auf den Wertebereich $(0, 1)$ abbildet, kann die einzelne Ausgabe y des Netzes als Näherung der Wahrscheinlichkeit dafür interpretiert werden, dass das Eingabemuster x zur ersten Klasse gehört. Es folgt automatisch, dass die Wahrscheinlichkeit dafür, dass es zur zweiten Klasse gehört, 1 minus der Ausgabe ist:

$$p(C_1 \mid x) = y = \sigma(a) \tag{2.25}$$

$$p(C_2 \mid x) = 1 - y. \tag{2.26}$$

Wir können die (eindimensionalen) Zielvektoren in der Form darstellen, dass $t = 1$, wenn das Eingabemuster zur Klasse C_1 gehört und $t = 0$ wenn es zur Klasse C_2 gehört. Damit erhält man eine kompakte Form für die Wahrscheinlichkeiten der Ziele:

$$p(t \mid x) = y^t (1 - y)^{1-t}. \tag{2.27}$$

Für den Fall mehrerer Klassen ($K > 2$) bieten sich K Ausgabeneuronen an, deren Aktivierungen über die sogenannte *Softmax-Funktion* normalisiert werden, um Näherungen für die jeweiligen Klassenwahrscheinlichkeiten zu erhalten:

$$p(C_k \mid x) = y_k = \text{softmax}(a_k) = \frac{\exp(a_k)}{\sum_{k'=1}^{K} \exp(a_{k'})}. \tag{2.28}$$

Die Zielvektoren lassen sich mithilfe einer 1-aus-K-Kodierung darstellen. Dabei werden sie durch einen binären K-dimensionalen Vektor repräsentiert, bei dem jeweils nur das k-te Bit auf 1 gesetzt wird, das der korrekten Klasse C_k entspricht, während die restlichen Bits 0 sind. Beispielsweise wird der Zielvektor t im Fall $K = 10$ und korrekter Klasse C_3 durch den Vektor $(0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$ repräsentiert. Damit erhält man wieder eine kompakte Form für die Wahrscheinlichkeiten der Ziele:

$$p(t \mid x) = \prod_{k=1}^{K} y_k^{t_k}. \tag{2.29}$$

Mit diesen Betrachtungen ist die Vorwärtsphase abgeschlossen und die Nutzung eines MLP für Klassifikationsprobleme beschreibt dann folgender simpler Prozess: Gib dem Netz das zu klassifizierende Eingabemuster, berechne alle Aktivierungen, propagiere die Signale bis zur Ausgabeschicht und wähle die Klasse, die gemäß (2.27) beziehungsweise (2.29) am wahrscheinlichsten ist, das heißt, deren zugehöriges Ausgabeneuron die höchste Aktivierung erzeugt hat.

2.2.2 Zielfunktionen

Das Training neuronaler Netzwerke umfasst das Anpassen der Gewichte aller vorhandenen Verbindungen derart, dass die Fehlermessung zwischen den Netzausgaben und den gewünschten Zielen minimiert wird. In Abhängigkeit dieser beiden Größen definiert man eine sogenannte *Zielfunktion*, welche in unserem Kontext durch den Prozess der Fehlerrückführung optimiert wird. Im Folgenden betrachten wir zwei populäre Varianten.

Definition 2.10 (Zielfunktion)**.** Eine *Zielfunktion* $O(\cdot)$ (objective function, cost function, loss function) beschreibt die zu optimierende Größe eines Optimierungsproblems. Im Kontext neuronaler Netzwerke modelliert sie die Fehlermessung zwischen der Netzausgabe und der korrekten Lösung.

Eine sehr universelle Zielfunktion, welche sich primär für Regressionen eignet, ist die *Summe der quadratischen Fehler*.

Definition 2.11 (Summe der quadratischen Fehler)**.** Die *Summe der quadratischen Fehler* (sum of squared errors, SSE) ist definiert als

$$O(S) = \sum_{(x,t)\in S} E(x,t) = \sum_{(x,t)\in S} \sum_{k=1}^{K} (y_k - t_k)^2. \tag{2.30}$$

Oft erhält diese Zielfunktion noch den Vorfaktor $\frac{1}{2}$, der sich als praktisch beim Bilden von Ableitungen für die Fehlerrückführung erweist. Warum Ableitungen eine Rolle spielen, wird im nächsten Abschnitt angerissen und genauer in Kapitel 2.5.1 behandelt. Alternativ kann man zusätzlich durch die Anzahl der Trainingspaare $|S|$ mitteln, um den *mittleren quadratischen Fehler* zu erhalten:

Definition 2.12 (Mittlerer quadratischer Fehler)**.** Der *mittlere quadratische Fehler* (mean squared error, MSE) ist definiert als

$$O(S) = \frac{1}{|S|} \sum_{(x,t)\in S} E(x,t) = \frac{1}{|S|} \sum_{(x,t)\in S} \sum_{k=1}^{K} (y_k - t_k)^2. \tag{2.31}$$

Es ist anzumerken, dass die Abhängigkeiten der Zielfunktion (wie zum Beispiel die Gewichte w) der Einfachheit halber in der obigen Darstellung unterdrückt wurden. Die einzelnen Fehlerwerte $E(x,t)$ sind intuitiv klar: Der Fehler wird klein ($E \approx 0$) genau dann, wenn der Ausgabevektor y das gewünschte Ziel t gut approximiert. Offensichtlich müssen alle Ausgabeneuronen k betrachtet werden, um die Fehler zum aktuellen Eingabemuster aufzusummieren. Ist das Netz hingegen schlecht trainiert, wird E groß, da die Netzausgaben von den Zielen stark abweichen. Die eigentliche Zielfunktion setzt sich dann aus den quadratischen Fehlern der gesamten Trainingsmenge zusammen. Das Quadrieren gibt allen Fehlern das gleiche Vorzeichen und verhindert damit, dass sich unterschiedliche positive und negative Terme aus verschiedenen Neuronen oder Trainingsmustern aufheben.
Die Herleitung der zweiten Zielfunktion richtet sich nach Kapitel 2.1.5 und findet daher auch hauptsächlich Anwendung in Klassifikationsproblemen. Es gab Versuche, die vollständige A-posteriori-Verteilung aus Gleichung (2.15) für neuronale Netze zu approximieren. Wir legen unseren Fokus aber auf die ML-Schätzung. Wir erinnern uns an die Gleichung (2.19) und die Berechnung des Vektors w_{ML}, der als Parameterkonstellation zur Approximation der A-posteriori-Verteilung der Klassen genutzt wird:

$$w_{\mathrm{ML}} = \arg\min_{w} - \sum_{(x,t)\in S} \ln p(t \mid x, w). \tag{2.32}$$

Beachte, dass die ML-Schätzung zur MAP-Schätzung (2.17) erweitert werden kann, indem man den Regularisierungsterm $p(w)$ wieder einführt, der die A-priori-Verteilung der Parameter beschreibt. Auf dieser Grundlage definieren wir zwei Zielfunktionen, welche beide als *Kreuzentropie* bekannt sind.

Definition 2.13 (Kreuzentropie)**.** Die Form der *Kreuzentropie* (cross-entropy function, CE) richtet sich nach der Anzahl der zu klassifizierenden Klassen. Für eine binäre Klassifikation ist sie definiert als

$$O(S) = - \sum_{(x,t)\in S} t \ln y + (1-t) \ln (1-y) \tag{2.33}$$

und für eine Klassifikation mit $K > 2$ Klassen ist sie definiert als

$$O(S) = - \sum_{(x,t)\in S} \sum_{k=1}^{K} t_k \ln y_k. \tag{2.34}$$

Für die Form der binären Klassifikation ($K = 2$) haben wir (2.27) in die zu minimierende Funktion aus (2.32) substituiert und analog für eine Klassifikationen mit $K > 2$ Klassen haben wir (2.29) anstatt (2.27) substituiert.

2.2.3 Rückwärtsphase

Da MLPs nach Konstruktion differenzierbare Operatoren sind, können sie mittels *Gradientenverfahren* jede differenzierbare Zielfunktion minimieren. Gradientenverfahren zum Training neuronaler Netze werden näher in Kapitel 2.5.1 behandelt, aber der Kerngedanke ist, dass man den Gradienten der Zielfunktion bezüglich der Gewichte bildet und dann Anpassungen in Richtung des negativen Anstiegs durchführt. Die benötigten Gradienten werden in der Rückwärtsphase berechnet.

Definition 2.14 (Rückwärtsphase)**.** Der Prozess, bei dem die Netzausgaben der Vorwärtsphase und die vorgegebenen Lösungen verarbeitet werden, um den Gradienten $\nabla_w O$ der Zielfunktion bezüglich der Gewichte zu berechnen, wird als *Rückwärtsphase* bezeichnet. Dabei wird die Fehlermessung der Zielfunktion mittels mehrfacher Anwendung der Kettenregel partieller Ableitungen rückwärts, beginnend bei der Ausgabeschicht, durchs Netz propagiert.

Wir werden uns im Folgenden beim Berechnen jeglicher Ableitungen immer auf ein Trainingspaar $(x, t) \in S$ beziehen, obwohl die Zielfunktionen des vorigen Abschnitts als Summen über der gesamten Trainingsmenge definiert wurden. Jedes Trainingspaar trägt in den Summen von (2.33) und (2.34) nur zu genau einem Term bei. Wir können die Zielfunktion also durch

$$O(S) = \sum_{(x,t)\in S} O(x, t) \tag{2.35}$$

beschreiben, wobei

$$O(x, t) = -\ln p(t \mid x) \tag{2.36}$$

gilt. Da es sich beim Gradienten um einen linearen Operator handelt, gilt

$$\frac{\partial O(S)}{\partial w} = \sum_{(x,t)\in S} \frac{\partial O(x, t)}{\partial w} \tag{2.37}$$

und dementsprechend reicht es aus, die Gradienten $\frac{\partial O(x,t)}{\partial w}$ zu berechnen, um den vollständigen Gradienten $\frac{\partial O(S)}{\partial w}$ zu bestimmen.
Um die nötigen Gradienten für das Training zu berechnen, nutzen wir die Fehlerrückführung. Diese beinhaltet eine wiederholte Anwendung der Kettenregel für partielle Ableitungen, um Gradienten berechnen zu können, die man auf direktem Weg sonst nicht erhalten würde. Wir beschränken uns auf den Fall mehrerer Klassen mit $K > 2$, da der binäre Fall sich ganz analog herleiten lässt. Gesucht sind also die Ableitungen $\frac{\partial O}{\partial w_{ij}}$ für alle Gewichte w_{ij} des Netzwerks. Leider lassen sich diese nicht direkt berechnen und wir müssen die Kettenregel anwenden. Der erste Schritt ist, die Kreuzentropie (2.34) bezüglich der Netzausgaben y_k abzuleiten:

$$\frac{\partial O}{\partial y_k} = -\frac{t_k}{y_k}. \tag{2.38}$$

Wenn man bedenkt, dass die Aktivierung jedes Neurons k der Ausgabeschicht aufgrund der Softmax-Funktion von allen Netzeingaben der Ausgabeneuronen abhängt, dann liefert die Kettenregel für die partiellen Ableitungen bezüglich der Netzeingaben a_k:

$$\frac{\partial O}{\partial a_k} = \sum_{k'=1}^{K} \frac{\partial O}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial a_k}. \tag{2.39}$$

Den ersten Term in der Summe kennen wir aus (2.38) und den zweiten können wir berechnen, indem wir (2.28) direkt mittels Quotientenregel ableiten:

$$\frac{\partial y_{k'}}{\partial a_k} = y_k \delta_{kk'} - y_k y_{k'}, \tag{2.40}$$

wobei $\delta_{kk'}$ das *Kronecker-Delta* ist, dass heißt

$$\delta_{ij} = \begin{cases} 1 & \text{falls } i = j \\ 0 & \text{falls } i \neq j \end{cases}. \tag{2.41}$$

Nun können wir (2.38) und (2.40) in (2.39) einsetzen und erhalten

$$\begin{aligned} \frac{\partial O}{\partial a_k} &= -\sum_{k'=1}^{K} \frac{t_k}{y_{k'}} (y_k \delta_{kk'} - y_k y_{k'}) = -\frac{t_k}{y_k}(y_k - y_k^2) + \sum_{\substack{k'=1 \\ k' \neq k}}^{K} t_k y_k \\ &= -t_k + y_k \sum_{k'=1}^{K} t_k = y_k - t_k, \end{aligned} \tag{2.42}$$

da $\sum_{k'=1}^{K} t_k = 1$ gilt. Bevor wir uns nun rückwärts durchs Netz arbeiten, führen wir noch eine abkürzenden Notation für ein beliebiges Neuron j ein:

$$\delta_j \stackrel{\text{def}}{=} \frac{\partial O}{\partial a_j}. \tag{2.43}$$

Bedenkt man, dass jede Zielfunktion O von jedem Neuron h der letzten verborgenen Schicht nur indirekt über dessen Einfluss auf die Ausgabeneuronen abhängt (nämlich durch den Anteil der Aktivierung b_h an den Netzeingaben der Ausgabeneuronen), so erhält man unter Anwendung der Kettenregel

$$\delta_h = \frac{\partial O}{\partial a_h} = \frac{\partial O}{\partial b_h} \frac{\partial b_h}{\partial a_h} = \frac{\partial b_h}{\partial a_h} \sum_{k=1}^{K} \frac{\partial O}{\partial a_k} \frac{\partial a_k}{\partial b_h}. \tag{2.44}$$

Die Ableitungen $\frac{\partial b_h}{\partial a_h}$ und $\frac{\partial a_k}{\partial b_h}$ können direkt berechnet werden (siehe (2.21) und (2.24)) und es folgt

$$\delta_h = f_h'(a_h) \sum_{k=1}^{K} \delta_k w_{hk}. \tag{2.45}$$

Analog lassen sich die δ-Terme für alle weiteren verborgenen Schichten rekursiv berechnen. Für die Neuronen h der l-ten verborgenen Schicht gilt beispielsweise

$$\delta_h = f'_h(a_h) \sum_{h'=1}^{H_{l+1}} \delta_{h'} w_{hh'}. \tag{2.46}$$

Haben wir alle δ-Terme, können wir endlich die Ableitungen der Zielfunktion bezüglich der Gewichte w berechnen. Für ein beliebiges Gewicht w_{ij} sagt die Kettenregel zunächst

$$\frac{\partial O}{\partial w_{ij}} = \frac{\partial O}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}}, \tag{2.47}$$

und nach Substitution der Ableitung $\frac{\partial a_j}{\partial w_{ij}}$ (siehe (2.22)) erhält man

$$\frac{\partial O}{\partial w_{ij}} = \delta_j b_i, \tag{2.48}$$

wobei die b_i für den Fall, dass Neuron i ein Eingabeneuron ist, den Eingaben x_i entsprechen. Mit nur einem Durchlauf der Rückwärtsphase haben wir somit jegliche Ableitungen der Zielfunktion O bezüglich der Gewichte w erhalten. Diese werden verwendet, um mittels eines Gradientenverfahren die Gewichte so anzupassen, dass die Zielfunktion immer weiter optimiert wird (siehe Kapitel 2.5.1).

2.3 Rekurrente neuronale Netze

Die Idee rekurrenter Netze wurde bereits in Kapitel 2.1.2 eingeführt. Im Gegensatz zu den FNNs zeichnen sich RNNs durch Verbindungen von Neuronen einer Schicht zu Neuronen derselben oder einer vorangegangenen Schicht aus. Letzteren Fall werden wir in unseren Betrachtungen nicht berücksichtigen. Obwohl diese Erweiterung neuronaler Netze relativ trivial scheint, hat sie weitreichende Implikationen in der Praxis. In einem traditionellen neuronalen Netz nimmt man an, dass alle Eingaben (und Ausgaben) unabhängig voneinander sind. Diese Annahme ist für viele Problemstellungen allerdings unzureichend. Möchte man im Rahmen der Schrifterkennung beispielsweise das nächste Wort in einem Satz vorhersagen, so ist es wichtig zu wissen, welche Wörter davor auftraten. Rekurrente Verschaltungen von Neuronen können genutzt werden, um solche zeitlich codierten Informationen in den Daten zu entdecken. Ein MLP kann beispielsweise nur einzelnen Eingaben bestimmte Ausgaben zuordnen. Ein RNN kann hingegen prinzipiell ganzen Folgen von Eingaben bestimmte Ausgaben zuordnen und ist somit in der Lage, zeitlich dynamische Abhängigkeiten zu modellieren. Frühere Eingaben werden implizit in netzinternen Zuständen (als eine Art Gedächtnis) gespeichert und können somit die Netzausgaben beeinflussen. Theoretisch können RNNs Informationen aus beliebig langen Eingabesequenzen nutzen, aber in der Praxis stellt sich heraus, dass sie ohne Implementierung spezieller Module effektiv auf kürzere Sequenzen beschränkt sind. Auch bei RNNs haben sich diverse Architekturen herausgebildet. Beispiele sind *Elman-Netze*, *Jordan-Netze*, *Hopfield-Netze*, *voll-rekurrente Netze* und *bidirektionale re-*

kurrente Netze. Wir werden uns in diesem Abschnitt aber auf ein einfaches RNN mit nur einer selbst verbundenen verborgenen Schicht beschränken.

2.3.1 Vorwärtsphase

Die Vorwärtsphase eines RNN überträgt sich von der eines MLP mit einer verborgenen Schicht mit nur einer Erweiterung, nämlich dass sich die Eingaben, die an der verborgenen Schicht ankommen, nicht nur aus den Aktivierungen der vorhergehenden Schicht zusammensetzen, sondern dass auch die Aktivierungen der verborgenen Neuronen aus dem vorigen Zeitschritt berücksichtigt werden müssen. Dies wird deutlich in Abbildung 2.6, bei der unser RNN entlang der Eingabesequenz *zeitlich entfaltet* wird.

Definition 2.15 (Zeitlich entfalten)**.** Wird ein RNN *zeitlich entfaltet* (unfolding in time), entsteht ein zum zyklischen Netz äquivalentes azyklisches Netz in der Form eines FNN. Jede Schicht des entfalteten Netzes entspricht einer Kopie aller Neuronenschichten des zyklischen Netzes.

Dies ist eine nützliche Methode, um RNNs zu visualisieren. Die zeitlichen Abhängigkeiten, die sonst nur implizit durch die rekurrenten Verbindungen dargestellt sind, werden am vollständigen Netzwerk direkt sichtbar. Dabei wird das Netzwerk für die komplette Eingabesequenz x dargestellt und es entsteht ein zum zyklischen Netz äquivalentes azyklisches Netz. Die Azyklizität ist wichtig, da ansonsten die Vorwärts- und Rückwärtsphase nicht wohldefiniert wären.

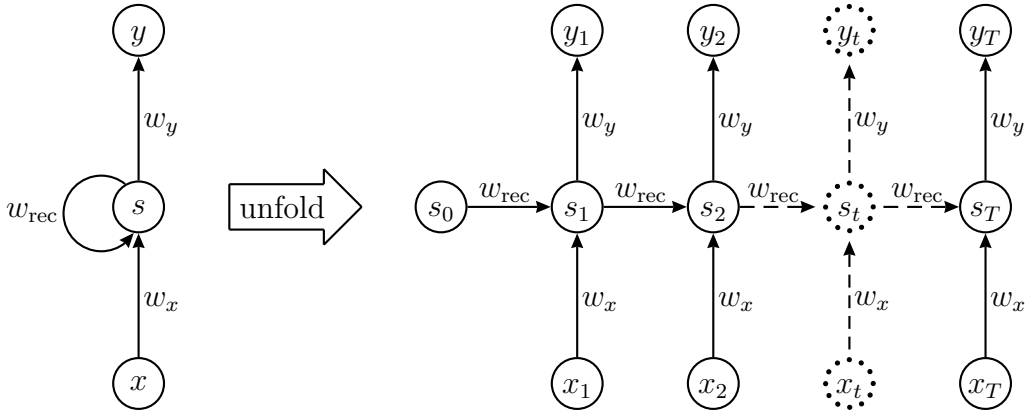


Abbildung 2.6: Ein einfaches rekurrentes Netz wird für T Zeitschritte entfaltet. Jeder Knoten entspricht einer Neuronenschicht zu einem bestimmten Zeitpunkt. Die Eingaben jeder verborgenen Schicht setzten sich aus dem internen Zustand der vorigen Schicht und dem Eingabemuster des jeweiligen Zeitschritts zusammen. Die Gewichte der kopierten Schichten stimmen alle überein.

Man konstruiert ein Netz mit T Schichten, wenn die Eingabesequenz die Länge T hat, wobei jede Schicht eine Kopie aller Neuronen des zyklischen Netzes ist. Ein einfaches RNN wird also in ein neuronales Netz mit T Schichten entfaltet, wobei jede Schicht als Eingabe ein Muster x_t der Sequenz und den internen Zustand s_{t-1} der vorigen Schicht

erhält und die Möglichkeit hat, ihre eigene Ausgabe y_t zu erzeugen. Es ist zu beachten, dass ein entfaltetes RNN die gleichen Gewichte in allen Zeitschritten verwendet (w_x, w_{rec}, w_y in Abbildung 2.6). Das verdeutlicht den Fakt, dass wir in jedem Zeitschritt den gleichen Prozess nur mit unterschiedlichen Eingaben durchführen. Durch das Auflösen der Rückkopplungen kann das Netz mit Verfahren für nicht rekurrente Netze trainiert werden.
Wir betrachten ein RNN mit einer Eingabeschicht der Dimension I, einer verborgenen Schicht der Dimension H und einer Ausgabeschicht der Dimension K. Ein Modell dieses RNN ist in Abbildung 2.7 zu sehen.

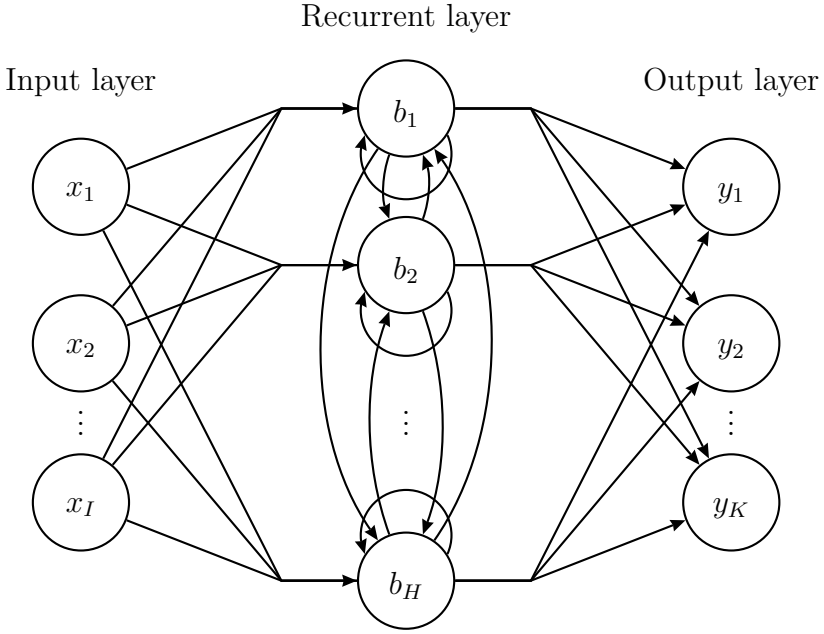


Abbildung 2.7: Ein rekurrentes Netz mit I Eingaben, einer verborgenen rekurrenten Schicht der Dimension H und K Ausgaben. Jedes Neuron der rekurrenten Schicht ist mit allen Neuronen derselben Schicht verbunden.

Dem Netz werden Eingabesequenzen x der Länge T präsentiert, wobei x_i^t den Wert des Eingabeneurons i zum Zeitpunkt t bezeichnet. Ebenso seien a_j^t und b_j^t die jeweilige Netzeingabe und Aktivierung von Neuron j zum Zeitpunkt t. Für die Netzeingabe a_h^t eines beliebigen verborgenen Neurons h zum Zeitpunkt t erhält man

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}. \tag{2.49}$$

Die Anwendung nicht-linearer differenzierbarer Aktivierungsfunktionen $f(\cdot)$ gleicht der beim MLP und es wird hier auch keine zeitliche Differenzierung vorgenommen. Die Aktivierung b_h^t berechnet sich also mittels

$$b_h^t = f_h(a_h^t). \tag{2.50}$$

Beginnend im Zeitschritt $t = 1$ lassen sich die beiden Vorschriften (2.49) und (2.50)

dann rekursiv durch Inkrementieren der Zeitvariable nach jedem Schritt anwenden, um die komplette Sequenz aller verborgenen Aktivierungen zu bestimmen. Dabei ist zu beachten, dass für die Berechnung der Werte a_h^1 Initialwerte $b_{h'}^0$ benötigt werden, welche den internen Zustand s_0 des RNN repräsentieren, bevor dieses irgendeine Eingabe der Datensequenz erhalten hat. Als einfachste Lösung bietet sich hierfür die Initialisierung mit Nullwerten an. Allerdings können anders gewählte Startwerte die Stabilität und Robustheit des RNN verbessern [38]. In jedem Zeitschritt t werden auch die Netzeingaben a_k^t der Ausgabeneuronen k berechnet:

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t. \tag{2.51}$$

Für Klassifikationsprobleme lassen sich die gleichen Aktivierungsfunktionen in der Ausgabeschicht einsetzen wie bei MLPs (wir nutzen die Softmax-Funktion), wobei die eigentliche Klassifizierung meist erst am Ende der gesamten Sequenz ($t = T$) durchgeführt wird. Ebenso übertragen sich die Überlegungen zu den Zielfunktionen.

2.3.2 Zeitliche Fehlerrückführung

Zwei bekannte Algorithmen, die zum Trainieren rekurrenter Netze entworfen wurden, um die Ableitungen der Zielfunktion bezüglich der Gewichte zu berechnen, sind das *rekurrente Lernen in Echtzeit* (real time recurrent learning, RTRL) und die *zeitliche Fehlerrückführung* (backpropagation through time, BPTT). Auch wurden hybride Varianten dieser beiden Algorithmen vorgeschlagen. Wir fokussieren uns auf BPTT als konzeptionell einfacheren Algorithmus, der außerdem effizienter in der Rechenzeit ist (allerdings auch speicherhungriger). Das Training eines RNN mittels BPTT ähnelt dem Training eines MLP, da, wie der Name schon vermuten lässt, eine abgewandelte Form der Fehlerrückführung angewandt wird. Dadurch, dass die Gewichte von allen Zeitschritten geteilt werden, hängt der Gradient bei jeder Ausgabe von den aktuellen Berechnungen und zusätzlich von allen vorigen ab. Deswegen wird in der Rückwärtsphase eine Fehlerrückführung durchgeführt, die sich in der Zeit zurückbewegt. Man kann sich BPTT als Kombination der zeitlichen Entfaltung eines RNNs mit einer Fehlerrückführung in sequenzieller Reihenfolge durchs Netz vorstellen. Durch das Entfalten entsteht ein FNN, welches mit der bekannten BP trainiert werden kann. BPTT nutzt also genau wie BP die mehrfache Anwendung der Kettenregel zur Berechnung der Gradienten.
Wie bereits erwähnt, wird dem Netz der Zielvektor t bei Klassifikationsproblemen erst im letzten Zeitschritt präsentiert. Hier wird dann auch der eigentliche Fehler der Ausgabeschicht berechnet. Der Fehler der Ausgabeschicht in allen vorigen Zeitschritten kann dann einfach Null gesetzt werden. Die partiellen Ableitungen $\frac{\partial O}{\partial y_k^T}$ der Zielfunktion bezüglich der Netzausgaben im letzten Zeitschritt gleichen also den partiellen Ableitungen in (2.38) beim MLP und die restlichen partiellen Ableitungen $\frac{\partial O}{\partial y_k^t}, t \in \{1, \ldots, T-1\}$

werden auf Null gesetzt. Wir nutzen wieder die δ-Notation

$$\delta_j^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial a_j^t} \tag{2.52}$$

und erhalten sofort $\delta_k^t = 0, t \in \{1, \ldots, T-1\}$, da

$$\delta_k^t = \frac{\partial O}{\partial a_k^t} = \sum_{k'=1}^{K} \frac{\partial O}{\partial y_{k'}^t} \frac{\partial y_{k'}^t}{\partial a_k^t} = \sum_{k'=1}^{K} 0 \cdot \frac{\partial y_{k'}^t}{\partial a_k^t}. \tag{2.53}$$

Analog zu (2.42) erhält man

$$\delta_k^T = \frac{\partial O}{\partial a_k^T} = y_k^T - t_k. \tag{2.54}$$

Für die Berechnung der δ-Terme der verborgenen Neuronen muss man sich folgendes klarmachen: Die Zielfunktion hängt jetzt von den Aktivierungen der verborgenen Neuronen ab, nicht mehr nur noch durch deren Einfluss auf die Ausgabeschicht, sondern auch durch ihren Einfluss auf die verborgene Schicht des nächsten Zeitschritts. Gleichung (2.45) muss also erweitert werden und es gilt für ein beliebiges verborgenes Neuron h im Zeitschritt t:

$$\delta_h^t = f_h'(a_h^t) \left(\sum_{k=1}^{K} \delta_k^t w_{hk} + \sum_{h'=1}^{H} \delta_{h'}^{t+1} w_{hh'} \right). \tag{2.55}$$

Beginnend im Zeitschritt $t = T$ lässt sich die Vorschrift (2.55) dann rekursiv durch Dekrementieren der Zeitvariable nach jedem Schritt anwenden, um die komplette Sequenz aller δ-Terme zu berechnen. Dabei gilt $\delta_h^{T+1} = 0$ für alle h, da es keine Fehlerterme über das Ende der Eingabesequenz hinaus gibt. Unter Berücksichtigung des Faktes, dass die Gewichte des entfalteten Netzes in jeder Schicht gleich sind (da jede Schicht einer Kopie des zyklischen Netzes entspricht), folgt, dass man für die Berechnung der partiellen Ableitungen bezüglich der Gewichte mehrere mögliche Anwärter hat (siehe (2.48)). Mit diesen lässt sich auf unterschiedliche Weise verfahren. Man kann alle Werte kumulieren, was unter Umständen aber zu große Änderungen pro Gewicht zur Folge hat, falls alle Werte das gleiche Vorzeichen haben. Bei einer Durchschnittsbildung würde dieses Problem etwas abgeschwächt werden. Eine andere Möglichkeit wäre es, einen Diskontierungsfaktor einzuführen, der den Einfluss weiter in der Vergangenheit liegender Werte abschwächt. Wir entscheiden uns an dieser Stelle für das einfache Kumulieren und erhalten

$$\frac{\partial O}{\partial w_{ij}} = \sum_{t=1}^{T} \frac{\partial O}{\partial a_j^t} \frac{\partial a_j^t}{\partial w_{ij}} = \sum_{t=1}^{T} \delta_j^t b_i^t, \tag{2.56}$$

womit die Rückwärtsphase abgeschlossen ist.
Ein Nachteil von BPTT ist, dass das Training eines entfalteten Netzes sehr lange dauern kann, da man unter Umständen eine große Anzahl an Schichten produziert, die wieder-

um eine sehr große Anzahl an Gewichten besitzen. Außerdem haben RNNs bei normaler BPTT Probleme damit, Abhängigkeiten über längere Zeiträume zu modellieren, da sie unter dem sogenannten *Problem des verschwindenden Gradienten* leiden.

Definition 2.16 (Problem des verschwindenden Gradienten)**.** Das *Problem des verschwindenden Gradienten* (vanishing gradient problem) ist ein fundamentales Problem gradientenbasierter Lernmethoden, ausgelöst durch bestimmte Aktivierungsfunktionen. In großen Bereichen ist der Gradient dieser Funktionen sehr klein, sodass eine Änderung der Netzwerkparameter kaum eine Auswirkung auf die Netzausgaben hat. Somit wird ein Training des Netzes fast unmöglich. Der Effekt wird verstärkt, wenn sich mehrere Schichten aneinander reihen, da die Gradienten durch die Kettenregel nach vorne hin exponentiell abnehmen.

Die Ableitungen der Sigmoid-Funktion, beschränkt durch $\frac{\beta}{4}$, und des Tangens Hyperbolicus, beschränkt durch 1, werden an beiden Rändern 0. Tritt dieser Fall ein, so spricht man von *saturierten* Neuronen. Die Fehlerrückführung berechnet die Gradienten mittels der Kettenregel. Wird ein RNN in n Schichten entfaltet, so hat das den Effekt, dass n dieser Ableitungen miteinander multipliziert werden, um die Ableitungen der vorderen Schichten zu berechnen. Also fällt der Gradient exponentiell in n ab und die Gradienten für die vorderen Schichten werden potentiell sehr klein oder verschwinden sogar endgültig nach ein paar Schritten. Das hat zur Folge, dass die netzinternen Zustände von frühen Zeitschritten kaum zum Training beitragen: Langzeitabhängigkeiten werden nicht modelliert. Verschwindende Gradienten sind kein exklusives Problem rekurrenter Netze. Auch *tiefe* FNNs, also FNNs mit sehr vielen Schichten, sind davon betroffen. Bei RNNs ist es deshalb so vordergründig, da sie durch das Entfalten meist automatisch sehr tiefe Architekturen bilden. Ebenso kann es sein, dass abhängig von der Wahl der Aktivierungsfunktionen und der Netzwerkparameter die Gradienten exponentiell ansteigen (explodieren), was analog im *Problem des explodierenden Gradienten* (exploding gradient problem) resultiert. Um diesen Phänomenen entgegenzuwirken, wurden bereits verschiedene Methoden entwickelt: Die ReLU als Aktivierungsfunktion hat immer die konstante Ableitung 0 oder 1 und speziell entwickelte Typen rekurrenter Netze ermöglichen eine Art Steuerung des Informationsflusses. Dazu zählen *Netze mit langem Kurzzeitgedächtnis* und *Netze mit gesteuerten rekurrenten Einheiten*.

2.4 Faltungsnetzwerke

Obwohl MLPs in der Theorie universelle Schätzer für Funktionen sind, können sie mit vielen Aufgaben in der Praxis nicht gut umgehen. Es entstehen diverse Probleme, wenn Objekte oder Muster erkannt werden sollen, die visuell repräsentiert werden. Zum einen steigt die Anzahl der Gewichte enorm schnell mit der Dimension der Eingaben, da jedes Neuronen einer Schicht mit allen Neuronen der Folgeschicht verbunden wird, was ein langsames Training zur Folge hat. Zum anderen werden keine räumlich codierten Informationen des Bildes genutzt: Für gewöhnlich hat jede einzelne Verbindung zweier Neuronen ihr eigenes Gewicht. Lernt das Netz nun ein Objekt in einer bestimmten Position zu erkennen, überträgt sich diese Fähigkeit nicht auf andere Bereiche im Bild,

da dort andere Gewichte für die Berechnungen zuständig sind. Um diesen Problemen entgegenzuwirken, wurden Faltungsnetzwerke entworfen.

Definition 2.17 (Faltungsnetzwerk)**.** Bei *Faltungsnetzwerken* (convolutional neural networks, CNN) handelt es sich um biologisch inspirierte Variationen von MLPs, die versuchen, den Prozess der visuellen Verarbeitung eines Tieres zu modellieren. Dabei werden lokale Verbindungsstrukturen zwischen Neuronen benachbarter Schichten aufgebaut und der Prozess der diskreten Faltung genutzt, um räumliche Informationen zu extrahieren.

CNNs haben das Ziel, die zweidimensionalen räumlichen Informationen zwischen den Pixeln eines Bildes auszunutzen und gleichzeitig die Anzahl der benötigten Parameter zu reduzieren. Außerdem soll bei der Objekterkennung ein hoher Grad an Invarianz bezüglich Translationen, Skalierungen und anderen Verformungen erreicht werden. CNNs gehören zur Klasse der FNNs, da ihr Aufbau immer noch in Schichten gegliedert ist und Signale stets vorwärts durchs Netz propagiert werden. Im Gegensatz zu MLPs nutzen sie aber neben den in Kapitel 2.1.1 eingeführten Perzeptron-Neuronen noch andere Techniken aus der Signalverarbeitung.

2.4.1 Struktur und Funktionsweise

Grundsätzlich machen sich CNNs drei simple Konzepte zu Nutze:

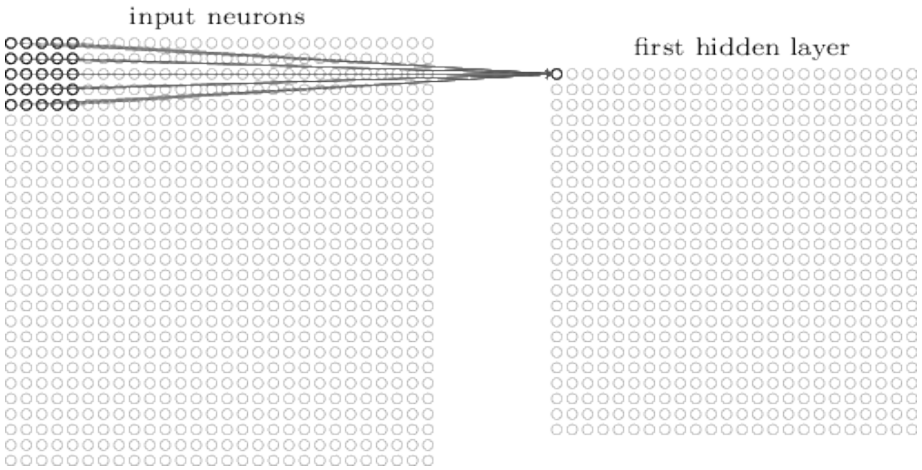
- Die Netzwerkarchitektur wird durch die Einführung lokaler Verbindungen in der Form von *lokalen rezeptiven Feldern* (local receptive fields) eingeschränkt.
- Bestimmte Gewichte werden von mehreren Neuronen geteilt, das heißt mehrere Verbindungen nutzen die gleichen Gewichte.
- Extrahierte Informationen werden zwischenzeitlich vereinfacht und zusammengefasst (pooling, subsampling).

Lokale rezeptive Felder

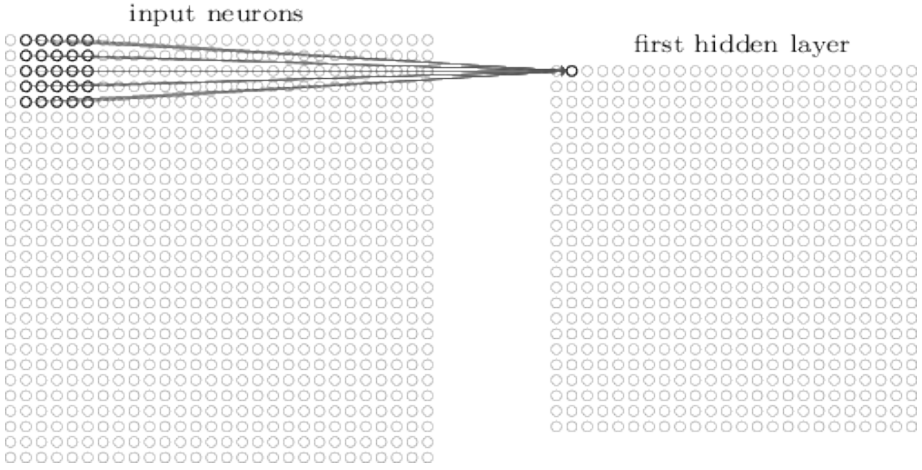
Bisher haben wir Eingaben immer als Vektor von Eingabeneuronen dargestellt. Für CNNs macht es Sinn, sich die Eingaben als Matrix von Neuronen vorzustellen, da CNNs meist Bilddaten präsentiert werden und man jedem Pixel des Bildes somit ein Neuron der Matrix zuordnen kann. Haben wir als Eingabe beispielsweise ein Bild bestehend aus 28×28 Pixeln, so entspricht dies einer Eingabeschicht von 28×28 Neuronen. Weiterhin waren wir es gewöhnt, dass ein Neuron mit allen Neuronen der vorhergehenden Schicht verbunden war. Bei CNNs hingegen setzen wir nur Verbindungen in kleinen lokalen Regionen, den sogenannten lokalen rezeptiven Feldern.

Definition 2.18 (Lokales rezeptives Feld)**.** Bei CNNs ist ein Neuron nur mit einer ausgewählten Menge an Neuronen der vorhergehenden Schicht verbunden. Diese Menge wird durch das dem Neuron zugeordnete *lokale rezeptive Feld* bestimmt.

Ein Neuron der ersten verborgenen Schicht wird mit einer kleinen ausgewählten Menge an Eingaben verbunden, beispielsweise mit einer 5×5 Region, welche 25 Eingabeneuronen umfasst. Jeder dieser 25 Verbindungen wird wie gehabt ein Gewicht w zugeordnet und das verborgene Neuron erhält zusätzlich ein Bias b. Konzeptionell gesehen lernt das verborgene Neuron also, Informationen aus seinem zugeordneten lokalen rezeptiven Feld zu extrahieren. Dieses Feld wird dann über das gesamte Eingabebild geschoben und jedem Schritt wird ein eigenes verborgenes Neuron zugeordnet. Dieser Zusammenhang ist für die ersten beiden Neuronen der ersten verborgenen Schicht in Abbildung 2.8 dargestellt, wobei das lokale rezeptive Feld immer um ein Pixel verschoben wird. Bei der Verschiebung ist es möglich, mit unterschiedlichen Werten zu experimentieren.



(a) Es bestehen nur Verbindungen zwischen dem verborgenen Neuron und seinem zugeordneten lokalen rezeptiven Feld.



(b) Verschieben des Feldes um einen Pixel liefert das lokale rezeptive Feld für das zweite verborgene Neuron.

Abbildung 2.8: Konzept und Zuordnung lokaler rezeptiver Felder [25].

Aufgrund der Begrenzungen am Rand des Bildes verkleinert sich die 28×28 Eingabeschicht auf eine 24×24 verborgene Schicht im ersten Schritt, wenn ein 5×5 Feld verwendet wird. Für unser Beispiel wäre die Ausgabe für ein verborgenes Neuron an der Stelle (i, j) dann folgendermaßen zu berechnen:

$$f\left(b + \sum_{u=0}^{4} \sum_{v=0}^{4} w_{u,v} y_{i+u,j+v}\right), \tag{2.57}$$

wobei f wieder die Aktivierungsfunktion bezeichnet, b den Bias, w eine 5×5 Matrix mit Gewichten und $y_{r,s}$ die Eingabe an der Position (r, s) (Ausgabe der vorigen Schicht). Beachte, dass die Terme b und $w_{u,v}$ keine Indizierung bezüglich i oder j enthalten. Der Grund dafür ist das Konzept der *gemeinsam genutzten Gewichte*, welches wir am Anfang dieses Abschnitts aufgelistet haben und welches in Kürze behandelt wird.
Die Doppelsumme in (2.57) hat dabei die Form einer *diskreten Faltung* (discrete convolution), woher auch der Name Faltungsnetzwerk stammt. In der digitalen Bildverarbeitung wird die Faltung zum Filtern von Bildern genutzt. Die Matrix w der Gewichte wird dabei *Faltungskern* genannt (auch Faltungsmatrix, Filtermaske, Filterkern, englisch convolutional kernel) und mit K notiert. Es handelt sich meist um quadratische Matrizen ungerader Dimension in unterschiedlichen Größen. Formal ist die Faltung nur für eine Anzahl von zwei Funktionen definiert. Wir können Graustufenbilder als diskrete zweidimensionale Funktionen darstellen:

$$I \colon \{1, \dots, n_1\} \times \{1, \dots, n_2\} \to W \subset \mathbb{R},\ (i, j) \mapsto I_{i,j}, \tag{2.58}$$

sodass das Bild I durch eine $n_1 \times n_2$ Matrix repräsentiert wird. Die Menge W ist meistens $\{0, \dots, 255\}$ und Farbbilder werden dann mit den drei RGB-Kanälen rot, grün und blau als $n_1 \times n_2 \times 3$ Matrix repräsentiert.

Definition 2.19 (Diskrete Faltung)**.** Sei $K \in \mathbb{R}^{2h_1+1 \times 2h_2+1}$ ein Faltungskern. Die *diskrete Faltung* eines Bildes I am Kern K ist durch

$$I^*_{x,y} = (I * K)_{x,y} \stackrel{\text{def}}{=} \sum_{u=-h_1}^{h_1} \sum_{v=-h_2}^{h_2} K_{u,v} I_{x+u,y+v} \tag{2.59}$$

gegeben, wobei

$$K = \begin{pmatrix} K_{-h_1,-h_2} & \cdots & K_{-h_1,h_2} \\ \vdots & K_{0,0} & \vdots \\ K_{h_1,-h_2} & \cdots & K_{h_1,h_2} \end{pmatrix} \tag{2.60}$$

gilt.

Bemerkung 2.3. Bei dieser Definition ist zu beachten, dass sie an den Rändern des Bildes nicht wohldefiniert ist, da Pixel außerhalb des Bildes referenziert werden. Wie in unserem obigen Beispiel lässt sich dieses Problem lösen, indem die Faltung nur an Positionen angewandt wird, für die keine Probleme entstehen. Dementsprechend verkleinert sich das Ergebnisbild I^* um $2h_1$ Pixel in der Breite und $2h_2$ Pixel in der Höhe.

Gemeinsam genutzte Gewichte

Jedem verborgenen Neuron wurde in unserem Beispiel ein Bias und 25 Gewichte für sein lokales rezeptives Feld zugeordnet. Das Interessante bei CNNs ist, dass wir diese 26 Parameter für jedes der 24×24 verborgenen Neuronen der ersten verborgenen Schicht nutzen. Deswegen fehlten in (2.57) auch die Indizes für j und k, da die Werte für alle Neuronen dieser Schicht übereinstimmen. Das bedeutet, dass alle Neuronen der ersten verborgenen Schicht die gleichen *Merkmale* (features) extrahieren, nur an unterschiedlichen Positionen. Ein von einem Neuron erkanntes Merkmal ist als eine Art Eingabemuster zu verstehen, dass die Aktivierung des Neurons zur Folge hat, zum Beispiel eine bestimmte Form innerhalb des Bildes. Die Abbildung von einer Schicht zur nächsten heißt deshalb auch *Merkmalsabbildung* (feature map). Damit lässt sich schon intuitiv erkennen, dass Faltungsnetzwerke eine Translationsinvarianz implementieren: Sind die Gewichte und der Bias so trainiert, dass das verborgene Neuron eine Ecke oder Kante in seinem lokalen rezeptiven Feld erkennen kann, dann ist es höchstwahrscheinlich auch nützlich, diese Fähigkeit auf andere Positionen im Netz zu übertragen, was durch die gemeinsam genutzten Gewichte und Bias realisiert wird.

Definition 2.20 (Gemeinsam genutzte Gewichte)**.** In einer feature map eines CNN werden für alle lokalen rezeptiven Felder der Neuronen die gleichen Gewichte und der gleiche Bias verwendet. Deshalb spricht man von *gemeinsam genutzten Gewichten* (auch *geteilte* oder *verteilte Gewichte*, englisch shared weights).

Eine feature map ist also nur in der Lage, bestimmte lokale Merkmale zu erkennen. Für die Bilderkennung sind aber viele verschiedene Merkmale erforderlich und deswegen werden *Faltungsschichten* im Netzwerk aus mehreren feature maps zusammengesetzt, wobei jede feature map ihren eigenen Bias und Faltungskern zugeordnet bekommt. Ein großer Vorteil von dieser Verteilung der Gewichte ist die enorme Reduzierung der Parameteranzahl in einem CNN im Vergleich zu Netzen, die voll-verknüpfte Schichten verwenden.

Definition 2.21 (Faltungsschicht)**.** Eine *Faltungsschicht* ist eine Neuronenschicht innerhalb eines CNN, die sich aus mehreren feature maps zusammensetzt, wobei jede feature map ihre eigenen verteilten Gewichte nutzt.

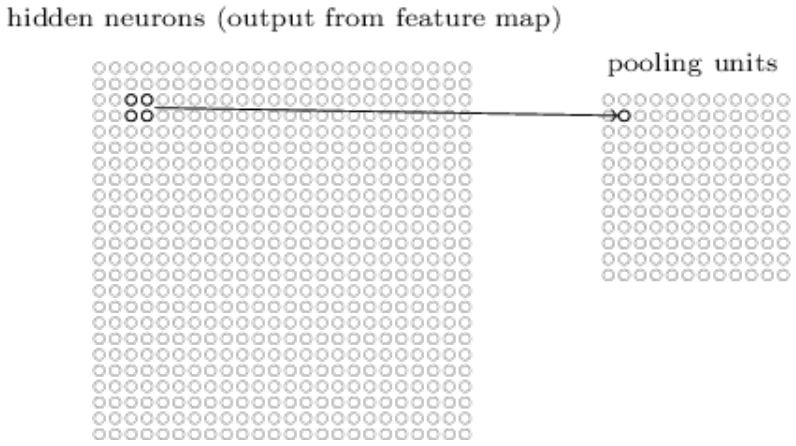
Unterabtastung

Zusätzlich zu den Faltungsschichten enthalten CNNs noch weitere Schichten, welche die Informationen der Ausgabe einer Faltungsschicht vereinfachen, beziehungsweise zusammenfassen, indem sie eine Unterabtastung der Daten durchführen oder über diese mitteln (pooling or subsampling layers). Motiviert wurde dieser Prozess durch eine wünschenswerte Robustheit gegen Rauschen und Verzerrungen in den Daten [15].

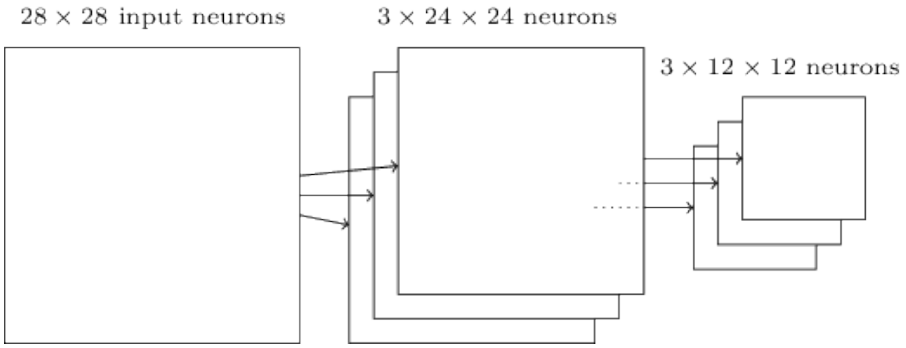
Definition 2.22 (Pooling-Schicht)**.** Eine *Pooling-Schicht* bildet die Ausgaben einer feature map auf eine feature map kleinerer Dimension ab. Dafür wird ein Fenster variabler Größe über die feature map geschoben und pro Fenster wird nur ein Wert in die neue feature map eingetragen. Man unterscheidet zwei Arten für das Abtasten:

- *Average-pooling*: Ausgabe ist der Mittelwert aller im Fenster liegenden Werte.
- *Max-pooling*: Ausgabe ist der größte im Fenster vorliegende Wert.

Für gewöhnlich werden die Fenster so platziert, dass sie sich nicht überlappen. Nehmen wir beispielsweise ein 2×2 Fenster, wird dieses immer um zwei Neuronen weitergeschoben. Die entstehende feature map wird dabei in ihren Dimensionen halbiert. Da wir in jeder Faltungsschicht mehrere feature maps bilden, wenden wir auf jede feature map eine eigene Abtastung an, sodass sich die Anzahl der feature maps durch eine Pooling-Schicht nicht verändert (siehe Abbildung 2.9). Der Prozess der Unterabtastung lässt sich so interpretieren, dass falls ein Merkmal gefunden wurde, die exakte Position nicht mehr so wichtig ist, solange die relative Position zu anderen Merkmalen beibehalten wird.



(a) Die feature map wird mit einem 2×2 Fenster abgetastet und ihre Dimension halbiert sich.



(b) Die Pooling-Schicht erhält die Anzahl der feature maps aus der Faltungsschicht.

Abbildung 2.9: Unterabtastung durch Pooling-Schichten [25].

Architektur

Der strukturelle Aufbau eines CNN ähnelt dem eines MLP, da es auch aus Schichten zusammengesetzt wird. Dazu zählen die Eingabeschicht, die verborgenen Schichten und die Ausgabeschicht. Im Gegensatz zu herkömmlichen FNNs besteht ein Großteil der verborgenen Schichten aus einer Kombination mehrerer Faltungs- und Pooling-Schichten. Traditionell folgt jeder Faltungsschicht eine Pooling-Schicht und der Kern eines CNNs setzt sich dann aus einer abwechselnden Folge von Faltungs- und Pooling-Schichten zusammen. Oft wird bei reduzierter Dimension der feature maps die Anzahl an feature maps der nächsten Faltungsschicht erhöht und Neuronen einer verborgenen Schicht können dann Verbindungen zu lokalen rezeptiven Feldern in mehreren feature maps der vorhergehenden Schicht haben. Die Ausgabeschicht ist dann meist wieder eine normale, voll-verknüpfte Schicht zur Klassifizierung. Ein solches Beispielnetz, welches strukturell dem LeNet ähnelt [21], ist in Abbildung 2.10 zu sehen.

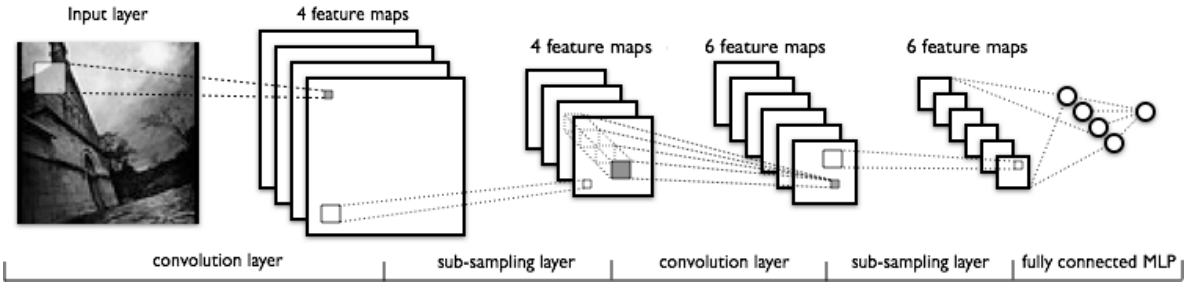


Abbildung 2.10: Ein Faltungsnetzwerk. Mehrere Faltungs- und Pooling-Schichten wechseln sich ab. Feature maps der zweiten Faltungsschicht beziehen Eingaben aus allen feature maps der vorigen verborgenen Schicht. Die Ausgabe wird durch voll-verknüpfte Schichten realisiert [39].

Modernere Varianten von CNNs führen noch weitere Arten von Schichten und strukturellen Kombinationen ein. In [19] wird beispielsweise eine Architektur genutzt, die 5 Faltungsschichten enthält. Jede dieser Schichten wird zunächst von einer Schicht gefolgt, welche die ReLU-Aktivierungsfunktion anwendet, um eine Nicht-Linearität ins Netz zu integrieren. Diese ist bei uns als Aktivierungsfunktion in der Faltungsschicht enthalten. Als nächstes folgt je eine Schicht, die eine Helligkeitsnormalisierung durchführt. Letztlich wird je eine Pooling-Schicht integriert. Hier werden sogar überlappende Fenster zugelassen, da dies die Chance hat, eine *Überanpassung* an die Trainingsmenge zu verringern.

2.4.2 Vorwärtsphase

Die Vorwärtsphase eines CNN verläuft ganz analog zu der eines MLP. Dem Netz werden Eingaben präsentiert, welche dann vorwärts durchs Netz propagiert werden, um die Ausgaben zu erzeugen. Dabei werden die oben beschriebenen Vorschriften der speziellen Schichten eingehalten.
Ist l eine Faltungsschicht, so besteht ihre Eingabe aus $d_1^{(l-1)}$ feature maps der vorigen Schicht, wobei jeder dieser feature maps die Dimension $d_2^{(l-1)} \times d_3^{(l-1)}$ hat. Ist $l = 1$,

so handelt es sich bei der vorigen Schicht um die Eingabeschicht, welche ein Bild I repräsentiert. In diesem Fall wäre $d_2^{(l-1)} \times d_3^{(l-1)}$ die Dimension des Bildes und $d_1^{(l-1)}$ entspräche der Anzahl der Farbkanäle (beispielsweise 1 für ein Graustufenbild oder 3 für ein RGB-Bild). Die Ausgabe der Schicht l besteht dann aus $d_1^{(l)}$ feature maps der Dimensionen $d_2^{(l)} \times d_3^{(l)}$. Die Netzeingabe $x_{i,j}^{(k,l)}$ eines Neurons an der Stelle (i,j) in der k-ten feature map der Schicht l berechnet sich mittels

$$x_{i,j}^{(k,l)} = b^{(k,l)} + \sum_{m=1}^{d_1^{(l-1)}} \sum_{u=0}^{h^{(l)}-1} \sum_{v=0}^{h^{(l)}-1} w_{u,v}^{(k,l)} y_{i+u,j+v}^{(m,l-1)}, \tag{2.61}$$

wobei $b^{(k,l)}$ der Bias ist, welcher innerhalb jeder feature map k einer Schicht l konstant ist, $w^{(k,l)}$ bezeichnet den Faltungskern für die feature map k der Schicht l und $y_{r,s}^{(m,l-1)}$ die Ausgabe des Neurons an der Stelle (r,s) in der m-ten feature map der vorigen Schicht $(l-1)$. Gleichzeitig haben wir uns auf quadratische Faltungskerne der Dimension $h^{(l)} \times h^{(l)}$ beschränkt und gehen davon aus, dass die Faltung nur auf zulässige Gebiete angewendet wird. Das heißt, die entstehenden feature maps haben die Größen $d_2^{(l)} = d_2^{(l-1)} - h^{(l)} + 1$ und $d_3^{(l)} = d_3^{(l-1)} - h^{(l)} + 1$. Außerdem ist zu beachten, dass die Gewichte $w_{u,v}^{(k,l)}$ in (2.61) nicht von der eingehenden feature map m abhängen. Das liegt daran, dass die Filter zur Berechnung einer fixierten feature map für gewöhnlich übereinstimmen. Gleichung (2.61) suggeriert ebenfalls, dass die Netzeingabe eines Neurons von allen feature maps der vorigen Schicht abhängt. Die erste Summe kann allerdings auch über eine Teilmenge der feature maps laufen. Die Netzeingabe $x^{(k,l)}$ wird innerhalb der Faltungsschicht l dann noch der Aktivierungsfunktion $f^{(l)}$ übergeben, um die Netzausgabe $y^{(k,l)}$ zu berechnen:

$$y_{i,j}^{(k,l)} = f^{(l)}(x_{i,j}^{(k,l)}). \tag{2.62}$$

Ist l eine Pooling-Schicht, so besteht ihre Eingabe aus $d_1^{(l-1)}$ feature maps der vorigen Schicht, wobei jeder dieser feature maps die Dimension $d_2^{(l-1)} \times d_3^{(l-1)}$ hat. Die Ausgabe enthält $d_1^{(l)} = d_1^{(l-1)}$ feature maps reduzierter Dimension. Gemäß der bereits vorgestellten Vorschriften (average-pooling, max-pooling) wird ein $p \times p$ Fenster über die einzelnen feature maps geschoben und in jedem Schritt wird nur 1 Wert in die neue feature map geschrieben. Findet die Unterabtastung ohne überlappende Fenster statt, so haben die ausgegebenen feature maps die Größen $d_2^{(l)} = \frac{d_2^{(l-1)}}{p}$ und $d_3^{(l)} = \frac{d_3^{(l-1)}}{p}$.
Sind l und $(l-1)$ voll-verknüpfte Schichten, so lassen sich die Ausgaben analog zum MLP berechnen. Andernfalls erwartet l als Eingabe $d_1^{(l-1)}$ feature maps der Dimension $d_2^{(l-1)} \times d_3^{(l-1)}$ und das i-te Neuron berechnet:

$$y_i^{(l)} = f^{(l)} \left(\sum_{m=1}^{d_1^{(l-1)}} \sum_{r=1}^{d_2^{(l-1)}} \sum_{s=1}^{d_3^{(l-1)}} w_{i,m,r,s}^{(l)} y_{r,s}^{(m,l-1)} \right), \tag{2.63}$$

wobei $w_{i,m,r,s}^{(l)}$ das Gewicht bezeichnet, welches das Neuron an der Stelle (r,s) in der m-ten feature map der Schicht $l-1$ mit dem i-ten Neuron der Schicht l verbindet.

2.4.3 Rückwärtsphase

Die Rückwärtsphase umfasst die Anwendung der bekannten Fehlerrückführung, um die Gradienten der Zielfunktion bezüglich der Netzparameter zu bestimmen. Da wir drei verschiedene Arten von Schichten haben, müssen allerdings einige Änderungen beim Herleiten der Ableitungen beachtet werden. Für voll-verknüpfte Schichten haben wir alle Terme bereits in Kapitel 2.2.3 hergeleitet. Das bedeutet, wir müssen uns noch um die Faltungs- und Pooling-Schichten kümmern. Wir nehmen außerdem ein paar strukturelle Vereinheitlichungen an den Schichten vor, um die Herleitung der Fehlerrückführung übersichtlicher zu gestalten. Die einzelnen Schritte lassen sich aber dennoch auf den allgemeinen Fall übertragen. Die folgenden Berechnungen sind an [7] angelehnt.
Der Einfachheit halber nehmen wir zunächst $d_1^{(l)} = 1$ für alle l an, sodass die Schichten nur je eine feature map enthalten. Dementsprechend benötigen wir auch nur einen Faltungskern $w^{(l)}$. Wir betrachten weiterhin eine quadratische $d_2^{(l-1)} \times d_3^{(l-1)} = d \times d$ feature map, welche von unserer Faltungsschicht l gefolgt wird. Der Faltungskern sei $w^{(l)} = w$ mit der Dimension $h^{(l)} \times h^{(l)} = h \times h$. Dementsprechend hat die ausgehende feature map die Dimension $d_2^{(l)} \times d_3^{(l)} = (d-h+1) \times (d-h+1)$ und die Berechnungen haben die folgende Form:

$$x_{i,j}^{(l)} = \sum_{u=0}^{h-1} \sum_{v=0}^{h-1} w_{u,v} y_{i+u,j+v}^{(l-1)}, \tag{2.64}$$

$$y_{i,j}^{(l)} = f^{(l)}(x_{i,j}^{(l)}), \tag{2.65}$$

wobei wir an dieser Stelle den Bias unterdrückt haben. Unter diesen Voraussetzungen leiten wir nun die Fehlerrückführung her.
Wir haben eine Zielfunktion O gegeben und nehmen an, dass wir die Fehlerterme $\frac{\partial O}{\partial y_{i,j}^{(l)}}$ der aktuellen Faltungsschicht l bereits kennen. Bedenke, dass wir diese Fehlerterme für jegliche voll-verknüpfte Schichten am Ende des CNNs mittels der bekannten Fehlerrückführung herleiten und zurück propagieren können. Die Frage ist, wie man diese Fehlerterme durch eine Faltungsschicht zur davor liegenden Schicht hindurch propagiert. Zunächst überlegen wir uns aber, wie die Ableitungen $\frac{\partial O}{\partial w_{u,v}}$ der Zielfunktion bezüglich der Gewichte einer Faltungsschicht aussehen, da diese zur Anwendung eines Gradientenverfahrens benötigt werden. Dabei muss beachtet werden, dass die Gewichte eines Faltungskerns über einer feature map gleich sind und man die beteiligten Terme somit alle zusammenfassen muss. Mithilfe der Kettenregel erhält man

$$\frac{\partial O}{\partial w_{u,v}} = \sum_{i=0}^{d-h} \sum_{j=0}^{d-h} \frac{\partial O}{\partial x_{i,j}^{(l)}} \frac{\partial x_{i,j}^{(l)}}{\partial w_{u,v}} = \sum_{i=0}^{d-h} \sum_{j=0}^{d-h} \frac{\partial O}{\partial x_{i,j}^{(l)}} y_{i+u,j+v}^{(l-1)}, \tag{2.66}$$

wobei der zweite Term in der Doppelsumme direkt mit (2.64) berechnet werden konnte. Somit fehlen uns nur noch die bekannten δ-Terme $\frac{\partial O}{\partial x_{i,j}^{(l)}}$. Diese erhält man ebenfalls über

die Kettenregel:

$$\delta_{i,j}^{(l)} = \frac{\partial O}{\partial x_{i,j}^{(l)}} = \frac{\partial O}{\partial y_{i,j}^{(l)}} \frac{\partial y_{i,j}^{(l)}}{\partial x_{i,j}^{(l)}} = \frac{\partial O}{\partial y_{i,j}^{(l)}} f'(x_{i,j}^{(l)}). \tag{2.67}$$

Da wir die Terme $\frac{\partial O}{\partial y_{i,j}^{(l)}}$ bereits kennen, lassen sich die δ-Terme ohne weitere Probleme berechnen und damit auch der Gradient der Zielfunktion bezüglich der Gewichte. Bleibt noch die Frage, wie wir die Fehler zur davor liegenden Schicht propagieren. Gesucht ist also $\frac{\partial O}{\partial y_{i,j}^{(l-1)}}$. Auch hier muss beachtet werden, dass die Ausgabe $y_{i,j}^{(l-1)}$ des Neurons an der Stelle (i, j) der Schicht $(l-1)$ mehrere Neuronen der Folgeschicht über deren lokale rezeptive Felder beeinflusst. Wir benutzen die Kettenregel und erhalten:

$$\frac{\partial O}{\partial y_{i,j}^{(l-1)}} = \sum_{u=0}^{h-1} \sum_{v=0}^{h-1} \frac{\partial O}{\partial x_{i-u,j-v}^{(l)}} \frac{\partial x_{i-u,j-v}^{(l)}}{\partial y_{i,j}^{(l-1)}} = \sum_{u=0}^{h-1} \sum_{v=0}^{h-1} \delta_{i,j}^{(l)} w_{u,v}, \tag{2.68}$$

wobei der zweite Term in der Doppelsumme wiederum direkt mit (2.64) berechnet werden konnte. Also haben wir an dieser Stelle auch die Fehlerterme für die einer Faltungsschicht vorangehende Schicht berechnet. In (2.68) ist zu beachten, dass der Ausdruck nur für Werte wohldefiniert ist, die mindestens h Stellen vom Rand entfernt sind. Abhilfe verschafft hier das Hinzufügen eines Rahmens voller Nullen. Betrachtet man (2.68) noch genauer, so erkennt man, dass es sich tatsächlich um eine Faltung handelt, wobei der Faltungskern einer Spiegelung von w entlang beider Achsen entspricht.
Letztlich stehen noch die Pooling-Schichten aus. Hier ist allerdings nicht viel zu tun, da die Pooling-Schichten kein eigentliches Lernen durchführen, sondern hauptsächlich der Dimensionsreduktion dienen. In der Vorwärtsphase wurden $p \times p$ Fenster zu einem einzelnen Neuron reduziert. Gelangt nun ein Fehlerterm an diesem Neuron an, so verteilt dieses den Fehler einfach auf die Neuronen des zugehörigen Fensters in der davor liegenden Schicht. Die Verteilung richtet sich dabei nach der Art der Unterabtastung: Beim max-pooling wird der gesamter Fehlerterm an das ausgewählte Neuron geleitet und beim average-pooling wird der Fehlerterm gleichmäßig auf alle Neuronen im Fenster verteilt.

2.5 Training und Optimierung

Bisher haben wir betrachtet, wie neuronale Netzwerke aufgebaut werden und wie sie in Bezug auf die Zielfunktion abgeleitet werden können, um sie dadurch mittels Gradientenverfahren zu trainieren. Wie genau das funktioniert und was beachtet werden muss, damit das Training effektiv verläuft beziehungsweise das Netz eine gute Generalisierung erreicht, wird in den folgenden Abschnitten behandelt. Obwohl die Zielfunktionen stets über der Trainingsmenge definiert wurden, ist das eigentliche Ziel, die Performance über einer während des Trainings nicht gesehenen Testmenge zu optimieren. Die Testmenge dient dabei als Indikator dafür, ob das Netzwerk in der Lage, ist gut zu generalisieren.

Definition 2.23 (Generalisierung und Überanpassung)**.** Die *Generalisierung* ist die Fähigkeit neuronaler Netze, nach erfolgreichem Training ähnliche Probleme derselben Klasse, die nicht explizit trainiert wurden, plausiblen Lösungen zuzuführen. Wurde das Netz zu lange oder zu intensiv trainiert und hat es somit die Eigenheiten der Trainingsmenge übernommen, so spricht man von einer *Überanpassung*.

2.5.1 Gradientenverfahren

Es wurde bereits erwähnt, dass wir die in den Rückwärtsphasen berechneten partiellen Ableitungen nutzen können, um mittels Gradientenverfahren unsere Netzwerke zu trainieren. Ziel des Trainings ist es, eine der gewählten Zielfunktionen zu minimieren, da wir diese als eine Art Fehlermessung eingeführt haben. Dabei ist das Trainieren als Anpassen der Gewichte w des Netzes zu verstehen (inklusive Bias). Allerdings haben viele verschiedene Faktoren Einfluss auf die Zielfunktion: Neben den Gewichten und dem Bias auch die Aktivierungsfunktionen, die allgemeine Netzarchitektur, die Trainingsdaten und so weiter. Deswegen ist es nützlich, diesen strukturellen Rahmen fürs Erste zu ignorieren und sich zunächst auf den Aspekt der Minimierung einer beliebigen Funktion zu konzentrieren. Bekannterweise gibt es analytische Methoden, um Extremalstellen von Funktionen zu finden. Aber für den Fall, dass die Funktion von sehr vielen Variablen abhängt, was bei neuronalen Netzen der Fall ist, sind diese sehr unpraktisch. Bewältigen lässt sich dieses Vorhaben mit Gradientenverfahren.
Wir erinnern uns daran, dass die Ableitung einer Funktion, die man beim Differenzieren erhält, die Richtung angibt, in welche die Funktionen ansteigt und abfällt. Die Ableitung ist auch ein Maß dafür, wie stark sich eine Funktion ändert, sobald man ihre Eingaben ändert [23]. Für eine einfache Funktion $O(x)$ gilt:

- $\frac{dO}{dx} > 0$ bedeutet, dass O wächst, sobald x wächst. Um das (lokale) Minimum von O zu finden, sollten wir x also verringern.
- $\frac{dO}{dx} < 0$ bedeutet, dass O abfällt, sobald x wächst. Um das (lokale) Minimum von O zu finden, sollten wir x also erhöhen.
- $\frac{dO}{dx} = 0$ bedeutet, dass wir uns an einer Extremalstelle von O befinden.

Um sich in Richtung eines Minimums zu bewegen, muss man also einfach der negativen Ableitung folgen. Diese Beobachtungen übertragen sich auf den Fall mehrerer Variablen, wobei man dann den Gradienten betrachten muss.

Definition 2.24 (Gradient)**.** Es sei $f(x_1, \ldots, x_n)$ eine n-dimensionale Funktion. Der *Gradient* g von f ist ein n-dimensionaler Vektor, dessen Einträge g_i den partiellen Ableitungen $\frac{\partial f}{\partial x_i}$ von f in x_i-Richtung entsprechen. Er ist die Verallgemeinerung der Ableitung für mehrdimensionale Funktionen, ist für jeden differenzierbaren Punkt einer Funktion f definiert, deutet genau in die Richtung des steilsten Anstiegs in diesem Punkt und gibt durch seinen Betrag $|g|$ den Steigungsgrad in diese Richtung an. Der Operator für einen Gradienten ∇ wird als *Nabla-Operator* bezeichnet.

Definition 2.25 (Gradientenabstieg)**.** Als *Gradientenabstieg* bezeichnet man den Prozess, bei dem man sich ausgehend von einem beliebigen Startpunkt einer Funktion f entgegen dem Gradienten ∇f schrittweise bergab bewegt, wobei die Schrittweite proportional zu $|\nabla f|$ ist.

Übertragen wir dieses Konzept auf unsere neuronalen Netze, so erhalten wir eine formale Update-Regel für die Gewichte in der Form

$$w^{(n+1)} = w^{(n)} + \Delta w^{(n)} = w^{(n)} - \eta \nabla O(w^{(n)}) = w^{(n)} - \eta \frac{\partial O}{\partial w^{(n)}}, \tag{2.69}$$

wobei $w^{(n)}$ der Gewichtsvektor im n-ten Schritt ist, $\Delta w^{(n)}$ die Anpassung der Gewichte im n-ten Schritt und $\eta \in [0, 1]$ die sogenannte *Lernrate*.

Definition 2.26 (Lernrate)**.** Die *Lernrate* η kontrolliert, wie stark sich die Gewichte in jeder Iteration des Gradientenabstiegs verändern sollen und ist ausschlaggebend für die Konvergenz des Algorithmus.

Der Prozess (2.69) wird dann solange wiederholt, bis ein *Abbruchkriterium* erfüllt wird, beispielsweise eine vorgegebene Anzahl an Durchläufen durch die gesamte Trainingsmenge, genannt *Epoche*, oder ein Punkt, an dem die Zielfunktion nicht weiter minimiert wird. Das Konzept des Gradientenabstiegs ist in Abbildung 2.11 zu sehen.

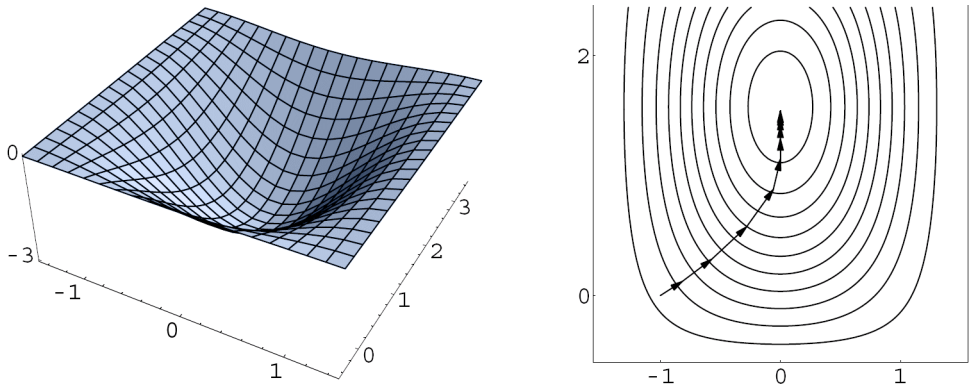


Abbildung 2.11: Veranschaulichung des Gradientenabstiegs anhand einer zweidimensionalen Fehlerfunktion. Links: Fehlergebirge in 3D. Rechts: Iterativer Abstieg und Höhenlinien in 2D. Man bewegt sich entgegengesetzt des Gradienten in Richtung einer Minimalstelle [18, Seite 64].

Es ist zu bedenken, dass Gradientenverfahren nicht perfekt arbeiten und ein paar potentielle Probleme mit sich bringen (siehe Abbildung 2.12):

- Gradientenverfahren können lokale Minima erreichen und dort stecken bleiben (Teil a).
- Ein Gradient nahe Null, beispielsweise beim Durchlaufen flacher Plateaus, kann das Training extrem verlangsamen oder ganz zum Stillstand bringen (Teil b).

- Steile Schluchten in der Fehlerfunktion können Oszillationen hervorrufen (Teil c).
- Gute Minima können bei zu großer Schrittweite verlassen werden (Teil d).

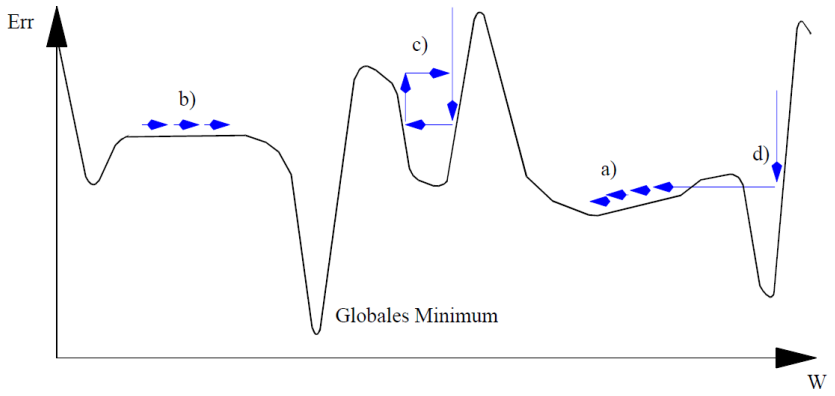


Abbildung 2.12: Potentielle Probleme während eines Gradientenabstiegs: a) Finden suboptimaler Minima, b) Quasi-Stillstand bei kleinem Gradienten, c) Oszillationen in Schluchten, d) Verlassen guter Minima [18, Seite 66].

Weiterhin können beim Ausführen des Gradientenabstiegs verschiedene Trainingsprotokolle angewandt werden, welche im folgenden Abschnitt besprochen werden.

2.5.2 Trainingsprotokolle

Wir erinnern uns an Kapitel 2.2.3, wo wir in (2.37) festgehalten haben, dass sich der Gradient der Zielfunktion, bezüglich der Gewichte über der gesamten Trainingsmenge aus den Gradienten der einzelnen Trainingspaare zusammensetzt. Man unterscheidet nun zwischen drei wesentlichen Trainingsprotokollen für das Anpassen der Gewichte mittels Gradientenabstieg:

- Stochastisches Training
- Online-Training
- Batch-Training

Beim *stochastischen Training* und beim *Online-Training* basieren die Gradienten in der Update-Regel (2.69) auf einem einzelnen Trainingspaar $(x, t) \in S$, das heißt die Gradienten haben die Form

$$\frac{\partial O}{\partial w^{(n)}} = \frac{\partial O(x,t)}{\partial w^{(n)}}. \tag{2.70}$$

Die beiden Vorgehen unterscheiden sich dahingehend, dass das Trainingspaar beim stochastischen Training stets zufällig ausgewählt wird und beim Online-Training alle Trainingspaare der Trainingsmenge sequenziell durchlaufen werden.

Beim *Batch-Training* hingegen wird die gesamte Trainingsmenge durchlaufen und die einzelnen Gradienten werden gemäß (2.37) aufsummiert, bevor ein Schritt der Gewichtsanpassung ausgeführt wird. Für alle Protokolle gilt, dass die Trainingsmenge im Verlauf des Trainings mehrmals durchlaufen wird. Es unterscheiden sich letztlich nur die Wahl der Trainingspaare und die Berechnung der verwendeten Gradienten.
Online-Training ist generell effizienter als Batch-Training, wenn die genutzten Daten große Mengen an Redundanzen oder Regularitäten enthalten, da sich Batch-Training immer erst die gesamte Trainingsmenge anschaut. Auch ist es möglich, dem Online-Training eine Art stochastisches Verhalten zu integrieren, indem vor jeder Epoche die Paare innerhalb der Trainingsmenge zufällig angeordnet werden. Das kann helfen, lokale Minima zu überwinden [22].
In der Praxis weit verbreitet ist eine Kombination von stochastischem Training und Batch-Training, genannt *Mini-Batch-Training*.

Definition 2.27 (Mini-Batch-Training)**.** Beim *Mini-Batch-Training* wird zunächst eine zufällige Teilmenge der Trainingsmenge bearbeitet, meist konstanter, wesentlich kleinerer Größe als die der Trainingsmenge und die jeweiligen Gradienten werden kumuliert, bevor die Gewichte angepasst werden.

Der Pseudocode für diesen Prozess ist in Algorithmus 2.1 zu sehen.

```
while Abbruchkriterium nicht erfüllt do
    Wähle zufällige Teilmenge der Trainingsmenge (Mini-Batch)
    foreach Trainingspaar im Mini-Batch do
        Durchlaufe Vorwärts- und Rückwärtsphase, um
        den aktuellen Gradienten zu berechnen
    end
    Summiere alle Gradienten vom Mini-Batch auf
    Passe die Gewichte mittels Gradientenabstieg an
end
```

Algorithmus 2.1: Mini-Batch-Training mit Gradientenabstieg

Mini-Batch-Training schließt also einen Kompromiss zwischen dem Berechnen des wahren Gradienten und dem Gradienten eines einzelnen Trainingspaares und integriert gleichzeitig durch die zufällige Wahl der Teilmenge einen gewissen Grad an stochastischem Verhalten. Die Berechnung mittels der Mini-Batches liefert eine bessere Approximation des wahren Gradienten als Online-Training oder stochastisches Training, ist aber immer noch schneller als Batch-Training, da in gleicher Zeit wesentlich mehr Schritte des Gradientenabstiegs durchgeführt werden. Die Schätzung ist zwar nicht perfekt, aber es reicht aus, wenn wir uns generell in eine Richtung bewegen, in die die Zielfunktion abnimmt. Das Nutzen von mehr als einem Trainingspaar zur Berechnung des Gradienten reduziert die Varianz in der Approximation des Gradienten und kann oft durch parallel arbeitende Architekturen in modernen Computern effizient implementiert werden. Bei der Wahl der Batch-Größe muss man abwägen, ob man die Varianz weiter reduzieren möchte oder öfter Schritte des Gradientenabstiegs ausführen möchte. Abhängig vom

Modell der Daten und der verfügbaren Hardware reichen die Teilmengen von einem bis zu mehreren Hundert Trainingspaaren. Für unsere Experimente haben wir meist Mini-Batch-Training mit 20 Trainingspaaren verwendet.

2.5.3 Normalisierung der Eingaben

Obwohl neuronale Netze relativ robust gegenüber der Darstellung der Eingabedaten scheinen, kann sich eine geeignete Normalisierung positiv auf die Performance auswirken. Jede Eingabevariable (Komponente der Eingabevektoren) sollte so vorverarbeitet werden, dass ihr Mittelwert über der gesamten Trainingsmenge nahe Null liegt mit einer Standardabweichung von 1. Man berechnet zunächst den Mittelwert

$$m_i = \frac{1}{|S|} \sum_{(x,t)\in S} x_i \tag{2.71}$$

und die Standardabweichung

$$\sigma_i = \sqrt{\frac{1}{|S|} \sum_{(x,t)\in S} (x_i - m_i)^2} \tag{2.72}$$

jeder Eingabevariable i und normalisiert diese dann mittels

$$\widetilde{x}_i = \frac{x_i - m_i}{\sigma_i}. \tag{2.73}$$

Die Informationen der Trainingsdaten gehen dadurch nicht verloren, aber die Werte der Eingaben liegen so in einem Bereich, der besser für die Aktivierungsfunktionen geeignet ist. Die berechneten Mittelwerte und Standardabweichungen der Trainingsmenge sollten auch für die Normalisierung der Validierungs- und Testmenge verwendet werden (vergleiche [10, Kapitel 3.3.3]).

2.5.4 Initialisierung der Gewichte

Beim Aufsetzen eines Netzwerks müssen vor Beginn des Trainings die Gewichte initialisiert werden. Die Initialisierung kann sich erheblich auf die Effizienz des Trainings und Performance des Netzwerks auswirken. Sind alle Gewichte Null, so findet gar kein Lernen statt. Die Gewichte sollten nicht alle den gleichen Startwert erhalten, da sonst beim Gradientenabstieg überall die gleichen Anpassungen stattfinden würden. Dementsprechend ist eine zufällige Initialisierung zu bevorzugen und es muss nur noch eine passende Größenordnung bestimmt werden. Wählt man die Gewichte anfangs zu hoch, so kann es sein, dass viele Neuronen sofort saturieren, wodurch das Training aufgrund der flachen Gradienten an den Enden der Aktivierungsfunktionen sehr langsam wird. Ebenso sollten auch zu kleine Gewichte vermieden werden. Viele Gradientenverfahren für neuronale Netze benötigen deswegen kleine, zufällig gewählte Gewichte nahe der Null. Diese Initialisierung mit kleinen positiven und negativen Gewichten hat im Idealfall zur Folge, dass die initialen Netzeingaben der Neuronen ebenfalls nahe Null liegen.

Das ist vorteilhaft, da dies der Bereich ist, an dem die Aktivierungsfunktionen linear agieren und ihren stärksten Gradienten besitzen. Somit werden gleich zu Anfang starke Lernimpulse ermöglicht (vergleiche [18, Seite 105]).
Die einfachste Möglichkeit der Initialisierung ist eine Gleichverteilung in einem kleinen Intervall um die Null zu wählen, das heißt $w_{ij} \in [-\varepsilon, \varepsilon]$ mit kleinem ε. Ebenso kann man eine Normalverteilung mit Mittelwert Null und Standardabweichung ε wählen. Häufig wird in der Literatur eine speziellere Methode vorgeschlagen (zum Beispiel [6]), bei der die Gewichte im Bereich

$$-\frac{1}{\sqrt{n_j}} < w_{ij} < \frac{1}{\sqrt{n_j}} \tag{2.74}$$

initialisiert werden, wobei n_j die Anzahl der eingehenden Gewichte zu Neuron j bezeichnet. Dieses Resultat basiert allerdings auf der Annahme, dass die Eingaben zu jedem Neuron normalverteilt sind und es sorgt dafür, dass die Netzeingaben der Neuronen sich im Bereich $(-1, 1)$ befinden. Für unsere Experimente haben wir meist die Methode der Normalverteilung mit einer Standardabweichung von 0.1 verwendet.

2.5.5 Lernkurven und frühes Stoppen

Jedes Netzwerk muss anfangs zufällig initialisiert werden, um einen Punkt zu haben, an dem das Training beginnen kann. Dementsprechend wird der Fehler auf der Trainingsmenge anfangs ziemlich hoch sein. Der Fehler wird dann durch das Training nach und nach verringert, was in der *Lernkurve* in Abbildung 2.13 zu sehen ist. Führt man parallel Messungen an der Testmenge durch, so sieht man, dass der Fehler generell auch sinkt, ab einem gewissen Zeitpunkt aber wieder ansteigt. Das sind Anzeichen einer Überanpassung des Netzwerks, unter welcher die Generalisierung leidet. Der Fehler auf der Trainingsmenge sinkt weiter, da das Netz anfängt, die Eigenheiten dieser Daten zu lernen, verliert gleichzeitig aber die Fähigkeit, andere Daten richtig einzuordnen.
Eine Möglichkeit, der Überanpassung entgegen zu wirken, ist das Konzept des *frühen Stoppens*, für das man einen Teil der Trainingsdaten abspaltet, um eine Validierungsmenge zu bilden.

Definition 2.28 (Frühes Stoppen)**.** *Frühes Stoppen* implementiert ein Abbruchkriterium, welches sich auf die Fehlermessungen der Validierungsmenge bezieht. Sobald ein minimaler Fehler auf der Validierungsmenge gefunden wurde, der Fehler für eine Weile stagniert oder der Fehler durch Überanpassung an die Trainingsdaten wieder anfängt zu wachsen, wird das Training abgebrochen.

Dieser Punkt des frühen Stoppens ist als gestrichelte Linie in Abbildung 2.13 eingezeichnet. Auswertungen der Testmenge sind der Vollständigkeit halber auch mit aufgeführt, obwohl diese während des Trainings typischerweise nicht genutzt werden, sondern erst nach Abschluss des Trainings zur Beurteilung der endgültigen Performance ausgewertet werden. Einziger Nachteil beim frühen Stoppen ist, dass ein gewisser Anteil der Trainingsdaten für die Validierungsmenge abgezogen werden muss, obwohl es für die Generalisierung besser ist, eine große Trainingsmenge repräsentativer Daten zu nutzen.

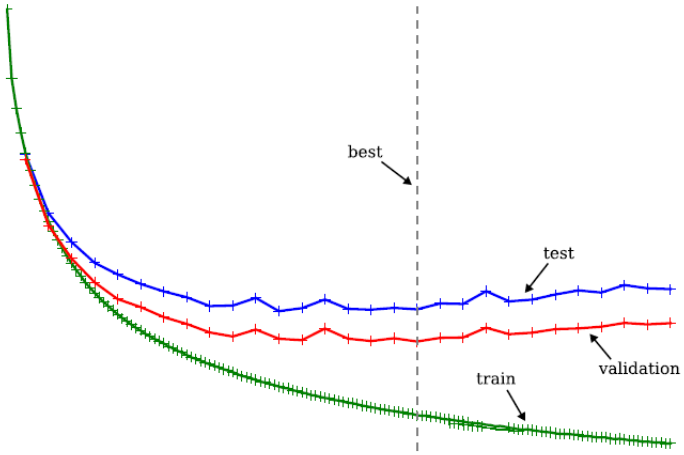


Abbildung 2.13: Typische Lernkurven während des Trainings. Anfangs sinkt der Fehler auf allen Datenmengen. Nach einer Weile fängt er allerdings auf den Daten, die nicht zum Training verwendet werden, wieder an zu steigen (Überanpassung). Die gestrichelte Linie indiziert den Punkt des frühen Stoppens, an dem der Fehler der Validierungsmenge am geringsten ist [10, Seite 27].

2.5.6 Lernrate

Die Lernrate η in (2.69) nimmt direkten Einfluss auf die Schrittgröße beim Gradientenabstieg und bedarf daher einer bedachten Wahl während des Trainings:

- Bei zu kleinem η wird das Training sehr lange dauern, da die Gewichte nur um kleine Beträge angepasst werden. Allerdings wird das Training auch stabiler, da Oszillationen entgegengewirkt wird und gute Minima nicht schnell verlassen werden.
- Bei zu großem η kann es sein, dass lokale Minima übersprungen werden, da die Gewichte zu große Änderungen erfahren. Im schlimmsten Fall oszillieren die Gewichte hin und her oder divergieren sogar.

Die einfachste Möglichkeit ist, eine konstante Lernrate für das gesamte Training zu wählen. Die Größenordnung kann man beispielsweise experimentell bestimmen, je nachdem welcher Ansatz bessere Ergebnisse liefert. Um die Konvergenz des Algorithmus zu verbessern, kann man in Betracht ziehen, die Lernrate während des Trainings zu verringern. Das Netzwerk sollte nämlich anfangs größere Anpassungen an den Gewichten durchführen und später immer kleinere Schritte ausführen, da man sich im Idealfall bereits in der Nähe eines Minimums befindet. Hier lässt sich mit linearen und exponentiellen Abnahmen der Lernrate in jeder Epoche experimentieren, wobei in unseren Experimenten eine exponentielle Abnahme schneller und effizienter trainiert hat. Es kann auch Sinn machen, darüber nachzudenken, unterschiedliche Lernraten in den verschiedenen Schichten zu nutzen, da wir durch Effekte wie den verschwindenden Gradienten wissen, dass die Gradienten der vorderen Schichten kleiner werden als die der hinteren. Durch

variable Lernraten in den Schichten kann dieser Effekt somit ausgeglichen werden. Wir haben stets eine globale, exponentiell abnehmende Lernrate verwendet.

2.5.7 Regularisierungen

Bereits beim frühen Stoppen haben wir gesehen, dass das eigentliche Trainingsziel ist, eine gute Generalisierung für neue Daten zu erreichen. Eine weitere Möglichkeit, eine Überanpassung an die Trainingsdaten zu vermindern, sind Regularisierungen, welche bestimmte Parameterkonstellationen bestrafen.

Definition 2.29 (L_p-Norm)**.** Die L_p-Norm $\|\cdot\|_p$ ist für einen beliebigen Vektor w definiert als:

$$\|w\|_p = \left(\sum_{i=0}^{|w|} |w_i|^p\right)^{\frac{1}{p}}. \tag{2.75}$$

Definition 2.30 (L_p-Regularisierung)**.** Für eine beliebige Zielfunktion $O(S, w)$ ist die *regularisierte Zielfunktion* $O_{\text{reg}}(S, w)$ definiert als

$$O_{\text{reg}}(S, w) = O(S, w) + \lambda L_p(w), \tag{2.76}$$

wobei $L_p(w) = \|w\|_p^p$ der *Regularisierungsterm* ist und λ der *Regularisierungsparameter*, der die Gewichtung der Regularisierung beeinflusst.

Solche Regularisierungsterme haben den Effekt, dass große Gewichte bestraft werden, da diese einen größeren Regularisierungsterm zur Zielfunktion addieren. Nur wenn sie die ursprüngliche Zielfunktion erheblich reduzieren, werden größere Gewichte zugelassen. Generell bevorzugt man kleine Gewichte, da große Gewichte für eine Anpassung an die Eigenheiten der Trainingsdaten und somit einer schlechteren Generalisierung sprechen. Das Hinzufügen solcher Regularisierungsterme wirkt sich offensichtlich auch auf den Gradientenabstieg aus. Für die L_2-Regularisierung im Speziellen gilt

$$\frac{\partial O_{\text{reg}}}{\partial w} = \frac{\partial O}{\partial w} + \lambda \frac{\partial L_2}{\partial w} = \frac{\partial O}{\partial w} + 2\lambda w, \tag{2.77}$$

wobei der Vorfaktor 2 auch vernachlässigt werden kann und wir die partiellen Ableitungen $\frac{\partial O}{\partial w}$ aus der Rückwärtsphase erhalten. Der Delta-Term in (2.69) verändert sich zu

$$\Delta w^{(n)} = -\eta \frac{\partial O}{\partial w^{(n)}} - \eta\lambda w^{(n)} \tag{2.78}$$

und somit gilt

$$w^{(n+1)} = (1 - \eta\lambda) w^{(n)} - \eta \frac{\partial O}{\partial w^{(n)}}, \tag{2.79}$$

das heißt, die Regularisierung entspricht einer Skalierung der Gewichte mit Faktor $(1 - \eta\lambda)$. Die L_2-Regularisierung ist auch als *weight decay* bekannt.

2.5.8 Globales Minimum und Momentum

Der Gradientenabstieg stellt sicher, dass die Gewichte so angepasst werden, dass wir letztlich in einem Minimum der Zielfunktion landen. Ob es sich dabei um ein lokales oder globales Minimum handelt, hängt hauptsächlich davon ab, wo man durch die zufällige Initialisierung des Netzes gestartet ist. Ein schwerwiegendes Problem von Gradientenabstiegsverfahren ist, dass sie leicht in lokalen Minima stecken bleiben. Diesen Effekt will man soweit es geht verhindern. Neben der Möglichkeit, eine große Menge an Netzen zu trainieren und zu hoffen, dass eines davon das globale oder zumindest ein sehr gutes Minimum erreicht, kann man dem Update in (2.69) auch einen Term hinzufügen, welcher eine Art Trägheit der Bewegung des Algorithmus durch das Fehlergebirge modelliert.

Definition 2.31 (Momentum)**.** Der Gradientenabstieg lässt sich mittels *Momentum* erweitern, indem man der Anpassung einen *Momentum-Term* hinzufügt:

$$\Delta w^{(n)} = \Delta w^{(n)} + \alpha \Delta w^{(n-1)}. \tag{2.80}$$

Dieser entspricht einem Anteil der Gewichtsanpassung vom vorigen Schritt. Der *Momentumparameter* $\alpha \in [0, 1]$ beeinflusst dabei die Gewichtung der Trägheit.

Man kann sich die Bewegung beim Gradientenabstieg dann wie einen Ball vorstellen, der ein Gebirge hinunterrollt und durch seine Trägheit kleine Täler überwindet, bis er in einem tiefen liegen bleibt. Man muss allerdings wieder einen Kompromiss schließen: Ist der Momentumparameter zu hoch, kann es sein, dass gute Minima übersprungen oder verlassen werden. Ist er zu klein, wird es nicht ermöglicht, lokale Minima zu verlassen und gleichzeitig verlangsamt sich das Training. Ein hoher Momentumparameter kann auch die Konvergenz des Trainings beschleunigen, indem flache Plateaus schneller durchlaufen werden.

2.5.9 Strukturelle Komplexität

Wenn man anfängt, ein künstliches neuronales Netz aufzubauen, scheint einem nicht klar zu sein, wie genau das Netz strukturiert werden soll. Neben der Entscheidung darüber, ob es sich um ein FNN, ein CNN, ein RNN oder ein Netz anderer Architektur handeln soll, muss auch festgelegt werden, wie viele Neuronenschichten integriert werden und wie viele Neuronen diese jeweils umfassen sollen. Gemeint sind die verborgenen Schichten, da die Dimensionen der Ein- und Ausgabeschicht durch die Eingabedaten beziehungsweise die Anzahl an Klassen vorgegeben sind. Diese Hyperparameter hängen meist stark von der Problemstellung und den benutzten Daten ab. Sind die Muster beispielsweise linear separierbar, reichen wenige verborgene Neuronen aus. Je komplexer die Daten und deren Verteilung ist, desto größere Kapazitäten wird das Netz benötigen, um ein gutes Modell simulieren zu können. Gibt es keine problemspezifischen Bedingungen, so reicht es in der Regel aus, eine verborgene Schicht zu integrieren, da das Netzwerk bereits mit 3 Schichten in der Lage ist, jede beliebige Funktion zu modellieren.

2.5.10 Training mit Rauschen

Bis jetzt haben wir versucht, die Fähigkeit der Generalisierung eines Netzes mittels diversen Regularisierungen oder anderen Optimierungen zu verbessern. Es sollte aber intuitiv klar sein, dass auch die Menge und Vielfalt der Trainingsdaten einen großen Einfluss auf das Training hat. Bekommt das Netz nur wenige Muster zu sehen, wird es wahrscheinlich relativ schlecht trainiert werden und keine gute Performance für neue Daten erreichen. Hat man stattdessen sehr viele Trainingsmuster zur Verfügung, die ein weites Spektrum an Variationen abbilden, so wird es für das Netz leichter sein, Regularitäten und wichtige Merkmale zu extrahieren. Oft kann es aber teuer sein, neue Trainingsdaten zu erhalten. In diesem Fall kann man seine Datenmenge künstlich erweitern, indem man vorhandenen Mustern ein wenig Rauschen hinzufügt und somit neue Eingabemuster mit gleichem Zielvektor erhält. Beispielsweise addiert man normalverteilte Werte mit Mittelwert Null und fixierter Varianz auf die Eingabevektoren. Festzustellen, wie hoch die Varianz dabei sein soll, kann mitunter problematisch sein und muss experimentell bestimmt werden. Wie genau man die vorhandenen Daten künstlich erweitert, hängt aber immer von der Problemstellung ab. Prinzipiell möchte man solche Operationen anwenden, welche mögliche Variationen in der realen Welt widerspiegeln. Arbeitet man mit Bilddaten, wird das einfache Hinzufügen von Rauschen wahrscheinlich nicht nützlich sein. Hier will man vielmehr neue Daten erzeugen, indem vorhandene Bilder verschoben, gedreht, skaliert oder anderweitig verformt werden.
In ähnlicher Manier kann man während des Trainings auch den Gewichten Rauschen hinzufügen (Addieren normalverteilter Werte). Da dies keinen Einfluss auf die Repräsentation der Eingaben hat, kann dieser Prozess bei allen Problemstellungen angewandt werden, ist aber meist auch weniger effektiv. Rauschen in den Gewichten kann als Vereinfachung des Netzes verstanden werden, da es die Genauigkeit reduziert, mit der die Gewichte eingestellt werden müssen. Im Allgemeinen werden einfache Netze bevorzugt, da sie besser generalisieren - eine Ausprägung von *Ockhams Rasiermesser* (vergleiche [10, Seite 28]).

3 Bestärkendes Lernen

Bestärkendes Lernen (reinforcement learning, RL) steht für eine Reihe von Methoden des maschinellen Lernens. Es beschreibt das Problem eines *Agenten*, ein bestimmtes Verhalten durch trial-and-error Interaktionen mit seiner dynamischen *Umwelt* zu lernen.

Definition 3.1 (Agent)**.** Ein *Agent* ist ein Computersystem, das sich in einer bestimmten Umgebung befindet und welches fähig ist, eigenständige Aktionen in dieser Umgebung durchzuführen, um seine (vorgegebenen) Ziele zu erreichen [36, Seite 5].

Es reicht, wenn man sich den Agenten als den Lerner vorstellt, also denjenigen, der Entscheidungen trifft und diese im Verlauf anpasst. Die Idee, dass wir lernen, indem wir mit unserer Umwelt interagieren, ist wahrscheinlich eine der offensichtlichsten, wenn wir über das Konzept des Lernens nachdenken. Deswegen liegt es nahe, diesen Prozess im Rahmen der künstlichen Intelligenz zu modellieren. Bestärkendes Lernen beschreibt eine Herangehensweise des Lernens, bei der man idealisierte Lernsituationen erforscht. Der Agent soll selbstständig sein Verhalten anpassen, um erhaltene *Belohnungen* zu maximieren. Im Gegensatz zu vielen anderen Formen des Maschinellen Lernens wird dem Agenten nicht vorgezeigt, welche Aktion in welcher Situation die beste ist. Er muss vielmehr herausfinden, welche Aktionen die größte Belohnung erzielen. Mit anderen Worten verspricht bestärkendes Lernen, dass der Agent aufgrund von Belohnungen und Bestrafungen lernt, ohne dass man ihm mitteilt, wie er sein Ziel zu erreichen hat. Aktionen können dabei sogar Auswirkungen auf zukünftige Belohnungen haben, sodass es nicht ausreicht, ausschließlich die unmittelbaren Ergebnisse zu berücksichtigen. Agenten müssen also weitestgehend in der Lage sein, den Zustand der Umgebung einzusehen und sie müssen in der Lage sein, Aktionen auszuführen, welche den Zustand der Umgebung beeinflussen. Gleichzeitig benötigt der Agent eine Art Ziel, das er verfolgt, welches im einfachsten Fall seine unmittelbare Belohnung nach Ausführung einer Aktion ist.
In diesem Kapitel werden zunächst notwendige Grundlagen für das bestärkende Lernen besprochen. Anschließend werden wir das bestärkende Lernen in den Kontext von *Markov-Entscheidungsprozessen* einbetten, da diese ein universelles Framework zur Lösung verschiedener Problemstellungen bieten. Der letzte Abschnitt beschäftigt sich mit *Strategie-Gradientenverfahren*, einer Klasse von Algorithmen zur Lösung solcher Probleme, wobei im Speziellen auf die *Likelihood-Ratio-Methode* eingegangen wird.

3.1 Grundlagen

Dieser Abschnitt stellt zunächst das *RL-Framework* vor, welches später in das allgemeinere Framework der Markov-Entscheidungsprozesse eingebettet wird. Danach wird

besprochen, anhand welcher Kriterien der Agent sein Verhalten optimieren sollte. Abschließend wird noch eine Eigenheit des bestärkenden Lernens hervorgehoben, durch welche man beim Lernen stets einen Kompromiss zwischen entgegengesetzten Zielen eingehen muss.

3.1.1 Das Framework

Systeme des bestärkenden Lernens umfassen einen Agenten, der mit seiner Umwelt, also allem was außerhalb von ihm liegt, interagiert. Die Interaktionen sind in diskrete Zeitschritte gegliedert und umfassen die Wahrnehmung der Umgebung sowie die Ausführung verschiedener Aktionen, welche wiederum unterschiedlich belohnt werden. Formal entspricht das

- einer Menge $\mathcal{S}$ von möglichen Zuständen der Umgebung,
- einer Menge $\mathcal{A}$ von möglichen Aktionen des Agenten und
- einer Menge skalarer Belohnungen $\mathcal{R}$ (typischerweise $\{0, 1\}$ oder $\mathbb{R}$).

In jedem Schritt t nimmt der Agent den Zustand $s_t \in \mathcal{S}$ seiner Umgebung als Eingabe wahr. Dann wählt er eine Aktion $a_t \in \mathcal{A}$ beziehungsweise $a_t \in \mathcal{A}(s_t)$, falls die verfügbaren Aktionen vom aktuellen Zustand der Umgebung abhängen, als Ausgabe und erhält im nächsten Zeitschritt eine Belohnung $r_{t+1} \in \mathcal{R}$. Gleichzeitig geht die Umgebung in den Zustand $s_{t+1} \in \mathcal{S}$ über. Diese Wechselwirkungen sind in Abbildung 3.1 modelliert.

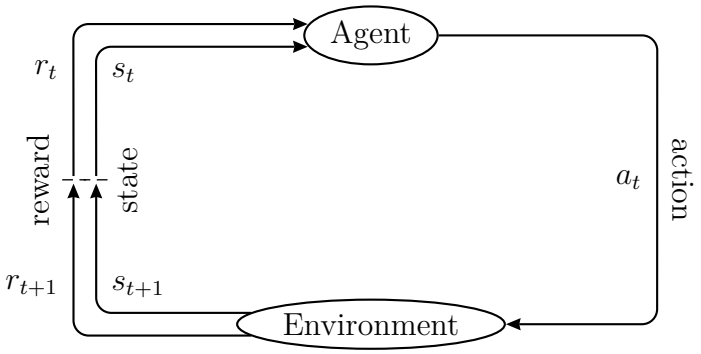


Abbildung 3.1: Die Interaktion zwischen Agent und Umgebung im RL-Framework.

Das allgemeine Framework umfasst neben dem Agenten und der Umgebung allerdings noch vier weitere Hauptbestandteile: Eine *Strategie* (policy), eine *Belohnungsfunktion*, eine *Nutzenfunktion* und optional ein *Modell der Umgebung*.
Das Verhalten des Agenten ist durch seine Strategie π definiert, welche im Allgemeinen Zustände der Umgebung auf Aktionen des Agenten abbildet, die in diesen Zuständen gewählt werden sollten. Strategien können verschiedenster Komplexität sein und stochastisches Verhalten implementieren.

Definition 3.2 (Strategie)**.** Die *Strategie* definiert das Verhalten des Agenten. Eine deterministische Strategie π ist eine Funktion $\pi\colon \mathcal{S} \to \mathcal{A}$, die jedem Zustand der Umgebung eine Aktion des Agenten zuweist. Eine stochastische Strategie π ist eine Wahrscheinlichkeitsverteilung, die jedem Zustand $s \in \mathcal{S}$ und jeder Aktion $a \in \mathcal{A}$ die Wahrscheinlichkeit $\pi(a \mid s)$ zuweist, dass Aktion a gewählt wird, wenn sich die Umgebung im Zustand s befindet.

Die Belohnungen des Agenten werden durch eine Belohnungsfunktion R repräsentiert, welche meistens deterministisch ist. Sie kann vom Agenten nicht beeinflusst werden, wird aber genutzt um sein Verhalten anzupassen.

Definition 3.3 (Belohnungsfunktion)**.** Nach der Ausführung einer Aktion a_t im Zeitschritt t erhält der Agent eine Belohnung r_{t+1} im nächsten Zeitschritt. Der Wert der Belohnung wird durch eine *Belohnungsfunktion* R vorgegeben. Diese kann abhängig vom Kontext verschiedene Formen annehmen:

- $R\colon \mathcal{S} \to \mathcal{R}$
- $R\colon \mathcal{S} \times \mathcal{A} \to \mathcal{R}$
- $R\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$

Die Belohnungsfunktion bildet also für gewöhnlich Zustände oder Zustand-Aktionspaare auf skalare Werte ab. Mitunter berücksichtigt sie den Übergang von einem Zustand in den nächsten.

Ein erweitertes Konzept sind Nutzenfunktionen. Während Belohnungsfunktionen ein Indikator dafür sind, was gut in einem unmittelbaren Sinne ist, spezifizieren Nutzenfunktionen, was gut über lange Sicht ist. Allgemein beschreibt der *Nutzen* eines Zustandes die Menge aller Belohnungen, die der Agent erwarten kann, wenn er in diesem Zustand startet. Nutzenfunktionen berücksichtigen also die womöglich folgenden Zustände und deren wahrscheinliche Belohnungen. Somit können Zustände beispielsweise immer geringe Belohnungen aber trotzdem hohen Nutzen erzeugen oder andersherum. Der einzige Sinn von Nutzenfunktionen ist, die Belohnungen auf lange Sicht zu maximieren, was gerade das Ziel des Agenten ist. Da nur die Belohnungen durch die Umgebung definiert sind, müssen Nutzenfunktionen über längere Sequenzen von Interaktionen abgeschätzt werden. Explizit werden der Nutzen eines Zustandes und die Nutzenfunktion in Kapitel 3.2.2 eingeführt.
Ein optionaler Bestandteil beim bestärkenden Lernen ist ein *Modell der Umgebung*. Dieses imitiert das Verhalten der Umgebung und kann genutzt werden, um das weitere Vorgehen des Agenten zu planen, da mögliche zukünftige Situationen berücksichtigt werden können. Beispielsweise sagt es bei einem vorgegebenen Zustand und einer Aktion den nächsten resultierenden Zustand und die zugehörige Belohnung vorher.

3.1.2 Optimales Verhalten

Dieser Abschnitt hält sich größtenteils an [16, Kapitel 1.2]. Bisher wurde davon gesprochen, dass der Agent sich so verhalten soll, dass er seine Belohnungen maximiert.

Dieses Verhalten ist aber nicht sehr eindeutig definiert und in der Tat gibt es verschiedene Modelle optimalen Verhaltens, welche im Folgenden vorgestellt werden. Neben der unmittelbaren Belohnung in jedem Zeitschritt nach Ausführung einer Aktion muss auch spezifiziert werden, wie der Agent die zukünftigen Belohnungen zu bewerten hat.

Definition 3.4 (Gewinn)**.** Der *Gewinn* G des Agenten bezeichnet im Allgemeinen die Summe all seiner Belohnungen. Im Speziellen unterscheidet man verschiedene Modelle des Gewinns, welche festlegen, in welcher Weise die Belohnungen unterschiedlicher Zeitschritte gewichtet werden.

Definition 3.5 (Modell mit endlichem Horizont)**.** Das *Modell mit endlichem Horizont* beschreibt einen Agenten, dessen Ziel es ist, seine erwartete Belohnung der nächsten T Schritte zu optimieren:

$$\mathbb{E}[G] = \mathbb{E}\left[\sum_{t=0}^{T} r_t\right]. \tag{3.1}$$

Dabei bezeichnet $\mathbb{E}[\cdot]$ den Erwartungswert-Operator.

Dieses Modell kann auf zweierlei Weise genutzt werden. In der ersten nutzt der Agent eine dynamische Strategie: Im ersten Schritt wählt er eine *T-Schritt optimale Aktion*, das heißt die beste Aktion unter der Einschränkung, dass noch T Schritte verbleiben. Im nächsten Schritt wählt er eine $(T-1)$-Schritt optimale Aktion und fährt so fort, bis er eine 1-Schritt optimale Aktion wählt und terminiert. In der zweiten nutzt der Agent eine gleitende Strategie: Er wählt immer die T-Schritt optimale Aktion, verhält sich also immer entsprechend der gleichen Strategie, welche nur T Schritte in die Zukunft schaut. Das Modell mit endlichen Horizont bietet sich an, wenn die Lebenszeit des Agenten im Vorhinein bekannt ist und somit nach einer vorgegebenen Anzahl an Schritten terminiert oder wenn das Problem *episodisch* ist.

Definition 3.6 (Episode)**.** Eine *Episode* entspricht einer Interaktionssequenz (des Agenten mit seiner Umwelt), die einen terminalen Zustand erreicht. Gliedert sich ein Problem in aneinander gereihte Episoden, so nennt man es *episodisch*.

In vielen Problemen zerfallen die Interaktionssequenzen aber nicht in identifizierbare Episoden, sondern laufen kontinuierlich fort. Hier wäre der finale Schritt also bei $T \to \infty$ und der erwartete Gewinn könnte ebenfalls unendlich sein, sodass der Agent nichts zu optimieren hätte.

Definition 3.7 (Modell mit unendlichem Horizont)**.** Das *Modell mit unendlichem Horizont* berücksichtigt die gesamten anfallenden Belohnungen des Agenten, wobei Belohnungen, die in der Zukunft liegen, geometrisch mit einem Diskontierungsfaktor $\gamma \in [0, 1]$ versehen werden:

$$\mathbb{E}[G] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]. \tag{3.2}$$

Der Diskontierungsfaktor γ kann dabei als Verzinsung, als Wahrscheinlichkeit des Überlebens für einen weiteren Schritt oder als mathematischer Trick zur Beschränkung der unendlichen Summe interpretiert werden.

Definition 3.8 (Modell der durchschnittlichen Belohnung)**.** Das *Modell der durchschnittlichen Belohnung* berücksichtigt den auf lange Sicht durchschnittlich erwarteten Gewinn:

$$\mathbb{E}[G] = \lim_{T \to \infty} \mathbb{E}\left[\frac{1}{T} \sum_{t=0}^{T} r_t\right]. \tag{3.3}$$

Dieses Modell kann als Grenzfall des Modells mit unendlichem Horizont interpretiert werden, wobei γ gegen 1 geht. Problematisch ist, dass hier nicht zwischen Strategien unterschieden werden kann, wenn eine den Großteil der Belohnungen in der Anfangsphase erhält und eine andere die Belohnungen auf weiter in der Zukunft liegenden Schritten verteilt. Außerdem werden die Belohnungen der Anfangsphasen oft von denen der auf lange Sicht erwarteten Belohnungen überwogen. Die Wahl des Modells und der optionalen Parameter kann in bestimmten Fällen die optimale Strategie beeinflussen. In unseren Experimenten haben wir stets das Modell mit endlichem Horizont verwendet, da es sich um ein episodisches Problem handelt.

3.1.3 Erkundung und Ausnutzung

Eine der Herausforderungen, die bestärkendes Lernen mit sich bringt, ist der Trade-Off zwischen *Erkundung* und *Ausnutzung* (exploration versus exploitation). Auf der einen Seite sollte ein Agent solche Aktionen bevorzugen, von denen er durch früheres Ausprobieren weiß, dass sie hohe Belohnungen erzeugen. Auf der anderen Seite muss er aber auch andere Aktionen wählen, die er noch nicht kennt, um herauszufinden, welche Aktionen am besten sind. Der Agent sollte also bekanntes Wissen *ausnutzen*, muss aber gleichzeitig neue Aktionen *erkunden*, um für die Zukunft optimalere Entscheidungen treffen zu können. Hier muss man einen Kompromiss eingehen, da ein zu hoher Fokus auf eine der beiden Seiten zu einem Misserfolg auf der anderen Seite führt. In einem stochastischen Rahmen müssen die gleichen Aktionen sogar mehrmals gewählt werden, um eine zuverlässige Schätzung für die zugehörigen erwarteten Belohnungen zu erhalten. Mit berücksichtigt werden muss auch die Dauer des Lernens beziehungsweise die Dauer der Interaktion zwischen Agent und Umgebung. Desto länger diese anhält, desto schwerwiegendere Konsequenzen hat es, wenn man sich früh dafür entscheidet, bestimmte Aktionen auszunutzen und desto mehr sollte der Agent erkunden. Dieses Dilemma zwischen Erkundung und Ausnutzung tritt beim überwachten Training beispielsweise nicht auf. Eine einfache Möglichkeit, eine stetige Erkundung zu gewährleisten, ist eine *ε-gierige Methode* zu verfolgen. Hier wählt der Agent stets die Aktion, von der er glaubt, dass sie die besten Belohnungen auf lange Sicht erzeugt mit einer Wahrscheinlichkeit von $1 - \varepsilon$ und wählt eine beliebige andere mit einer Wahrscheinlichkeit von $0 < \varepsilon < 1$. Der Parameter ε lässt sich offensichtlich dem Problem anpassen oder kann auch im Verlauf des Lernens adaptiv reguliert werden. In unserem Modell werden wir eine Erkundung durch stochastisches Verhalten integrieren.

3.2 Markov-Entscheidungsprozesse

Im RL-Framework trifft der Agent Entscheidungen anhand seiner Strategie, also anhand einer Funktion, die vom Zustandssignal der Umgebung abhängt. In diesem Abschnitt betrachten wir, welche Art von Informationen wir von dem Zustandssignal zu erwarten haben und definieren außerdem die sogenannte Markov-Eigenschaft. Weiterhin wird ein Spezialfall aus den resultierenden Markov-Entscheidungsprozessen vorgestellt, welcher für diese Arbeit von besonderem Interesse ist.

3.2.1 Die Markov-Eigenschaft

Mit dem Zustand der Umgebung wird die Menge an Informationen bezeichnet, die dem Agenten zur Verfügung steht. Der Agent nimmt den Zustand als Zustandssignal wahr. Abhängig vom Kontext kann dieses die unterschiedlichsten Formen annehmen. Neben direkten Wahrnehmungen wie sensorischen Messungen, kann der Zustand auch vorverarbeitete Signale oder komplex zusammengesetzte Strukturen enthalten. Schweifen wir mit unseren Augen beispielsweise über eine Landschaft, so ist zu jedem Zeitpunkt nur ein kleiner Ausschnitt wirklich detailliert zu sehen, der innerhalb der Fovea centralis liegt, dem Bereich des schärfsten Sehens der Netzhaut bei Säugetieren. Dennoch können wir eine fast perfekte Repräsentation der Landschaft aufbauen. Hier wurde der Zustand auf der Basis direkter Wahrnehmungen zusammen mit vorhergehenden Zuständen oder abgespeicherten Wahrnehmungen gebildet. Im Idealfall fasst das Zustandssignal alle vergangenen Wahrnehmungen kompakt zusammen, ohne dabei relevante Informationen zu verlieren. In der Regel wird dafür nicht eine unmittelbare Wahrnehmung ausreichen, aber es wird nie mehr als die gesamte Sequenz an Wahrnehmungen benötigt. Allgemein spricht man von einem Zustandssignal mit der *Markov-Eigenschaft*, wenn es alle relevanten Informationen erhält.
Wir betrachten eine beliebige Umgebung und nehmen an, dass die Anzahl an Zuständen und Belohnungen endlich ist. Im Allgemeinen hängt die Reaktion der Umgebung im Zeitschritt $t+1$ nach Ausführung einer Aktion im Zeitschritt t von der gesamten vorhergehenden Sequenz an Interaktionen ab. Diese Sequenzen bestehen aus den Aktionen a_t jedes Zeitschritts t nach Wahrnehmung des Zustands s_t und der Belohnung r_{t+1}. Zusammenfassen kann man diese bis zum Zeitpunkt t in einem *Verlauf* H_t.

Definition 3.9 (Verlauf)**.** Ein *Verlauf* H_t fasst alle Zustände, Aktionen und Belohnungen einer Interaktionssequenz bis zum Zeitpunkt t zusammen:

$$H_t = (s_0, a_0, r_1, \ldots, s_{t-1}, a_{t-1}, r_t, s_t, a_t). \tag{3.4}$$

Bemerkung 3.1. Für einige Betrachtungen spielen die Belohnungen keine Rolle. In diesen Fällen können die Belohnungen im Verlauf auch vernachlässigt werden.

Die möglichen Ereignisse müssen durch die gesamte Wahrscheinlichkeitsverteilung beschrieben werden:

$$p(r_{t+1} = r, s_{t+1} = s \mid H_t) \tag{3.5}$$

für alle $r \in \mathcal{R}, s \in \mathcal{S}$ und alle möglichen Verläufe H_t. Wenn das Zustandssignal allerdings die Markov-Eigenschaft hat, hängt die Reaktion der Umgebung im Zeitschritt $t+1$ ausschließlich von den Ereignissen im Zeitschritt t ab.

$$p(r_{t+1} = r, s_{t+1} = s \mid s_t, a_t) \tag{3.6}$$

für alle $r \in \mathcal{R}, s \in \mathcal{S}$ und alle möglichen Ereignisse s_t, a_t.

Definition 3.10 (Markov-Eigenschaft)**.** Ein Zustandssignal hat die Markov-Eigenschaft genau dann, wenn (3.5) und (3.6) für alle $r \in \mathcal{R}, s \in \mathcal{S}$ und alle möglichen Verläufe H_t äquivalent sind.

Die Markov-Eigenschaft ermöglicht es uns, auf der Grundlage des aktuellen Zustands und einer Aktion, den nächsten Zustand inklusive Belohnung vorherzusagen. Damit können optimale Strategien auf Grundlage von Markov-Zuständen gewählt werden und nicht mehr nur noch auf Grundlage von kompletten Interaktionssequenzen. Die Markov-Eigenschaft ist wichtig im RL, da von Entscheidungen und Nutzenwerten angenommen wird, dass sie durch Funktionen repräsentiert werden können, die ausschließlich von Zuständen abhängen.
Unter der Annahme der Markov-Eigenschaft an die Zustandssignale lässt sich das RL-Framework als *Markov-Entscheidungsprozess* modellieren.

Definition 3.11 (Markov-Entscheidungsprozess)**.** Ein *Markov-Entscheidungsprozess* (Markov decision process, MDP) ist ein zeitdiskreter stochastischer Prozess bestehend aus

- einer Menge $\mathcal{S}$ von Zuständen,
- einer Menge $\mathcal{A}$ von Aktionen,
- *Übergangswahrscheinlichkeiten* $p(s' \mid s, a)$, in einen neuen Zustand s' zu gelangen, abhängig vom aktuellen Zustand s und der gewählten Aktion a,
- einer Belohnungsfunktion R

und optional

- einem Diskontierungsfaktor $\gamma \in [0, 1]$ für die Differenzierung zwischen aktuellen und zukünftigen Belohnungen
- sowie einer Startverteilung p_0 der Zustände.

Sind die Zustands- und Aktionsmengen endlich, so spricht man von einem endlichen Markov-Entscheidungsprozess.

3.2.2 Nutzenfunktionen und optimale Strategien

Fast alle RL-Algorithmen beinhalten die Approximation von den bereits erwähnten Nutzenfunktionen, um einschätzen zu können, wie gut es für den Agenten ist, sich in einem gewissem Zustand zu befinden, immer im Hinblick darauf, den erwarteten Gewinn zu maximieren. Da die erwarteten Belohnungen stets von der aktuellen Strategie des Agenten abhängen, werden Nutzenfunktionen in Abhängigkeit von diesen definiert. Die folgenden Betrachtungen sind im Kontext eines MDP zu verstehen.

Definition 3.12 (Nutzen und Nutzenfunktionen)**.** Ist π eine Strategie, die jedem Zustand $s \in \mathcal{S}$ und jeder Aktion $a \in \mathcal{A}(s)$ eine Wahrscheinlichkeit $\pi(a \mid s)$ zuordnet, mit der Aktion a im Zustand s ausgeführt wird, dann bezeichnet man den *Nutzen* des Zustands s unter der Strategie π mit $v_\pi(s)$. Er beschreibt den erwarteten Gewinn des Agenten, wenn er in s startet und danach Strategie π anwendet und ist definiert als

$$v_\pi(s) = \mathbb{E}_\pi[G \mid s_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_k \mid s_t = s \right], \tag{3.7}$$

wobei $\mathbb{E}_\pi$ den Erwartungswert, gegeben der Strategie π, bezeichnet und t ein beliebiger Zeitschritt ist. Die Funktion v_π wird *Zustand-Nutzenfunktion* genannt. Auf ähnliche Weise definiert man auch die *Aktion-Nutzenfunktion* q_π, welche den Nutzen einer Aktion a im Zustand s unter der Strategie π beschreibt:

$$q_\pi(s, a) = \mathbb{E}_\pi[G \mid s_t = s, a_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_k \mid s_t = s, a_t = a \right]. \tag{3.8}$$

Algorithmen können diese Nutzenfunktionen approximieren, indem sie einer Strategie π folgen und Durchschnitte für jeden Zustand aufrechterhalten. Gibt es zu viele Zustände, so approximiert man die Nutzenfunktionen eher als parametrisierte Funktionen und passt die Parameter an. Die Nutzenfunktionen definieren eine Halbordnung über den Strategien. Es gilt $\pi \geq \pi'$ genau dann, wenn $v_\pi(s) \geq v_{\pi'}(s)$ für alle $s \in \mathcal{S}$ gilt. Eine Strategie π ist also besser als π' oder genauso gut, wenn ihr erwarteter Gewinn größer oder gleich dem erwarteten Gewinn von π' ist.

Definition 3.13 (Optimale Strategie)**.** Eine Strategie π^*, für die $\pi^* \geq \pi$ für alle Strategien π gilt, nennt man *optimale Strategie*, da sie den maximal möglichen Gewinn erzielt.

Um optimale Strategien für MDPs zu finden, gibt es diverse Methoden. Zu den fundamentalsten Klassen gehören die *dynamische Programmierung*, *Monte-Carlo Methoden* und das *Temporal Difference Learning*. Angenommen, man hat ein Modell des Problems gegeben, kennt also die Übergangswahrscheinlichkeiten p und die Belohnungsfunktion R, so lassen sich Algorithmen der dynamischen Programmierung anwenden, die diese Informationen ausnutzen, um optimale Strategien zu finden. Dazu zählen unter anderem die *Nutzen-Iteration*, welche iterativ eine optimale Nutzenfunktion bestimmt und damit gleichzeitig eine optimale Strategie induziert und die *Strategie-Iteration*, welche die Strategie direkt manipuliert und vergleicht, ob die Nutzenfunktion dadurch verbessert wird.

RL beschäftigt sich allerdings hauptsächlich mit dem Problem des Findens einer optimalen Strategie, wenn man keine Informationen zum Modell gegeben hat. Hier muss der Agent mit der Umwelt interagieren, um sich entweder ein Bild vom Modell aufzubauen, welches verarbeitet wird, um eine optimale Strategie zu finden oder um direkt Änderungen an der Strategie durchzuführen. Monte-Carlo Methoden lernen Nutzenfunktionen und optimale Strategien mithilfe von Erfahrungen in der Form von Beispielsequenzen, in denen sie Informationen über Zustände, Aktionen und Belohnungen sammeln. Monte-Carlo Methoden werden deshalb hauptsächlich für episodische Probleme genutzt, da sie nach Durchlauf einer Episode Änderungen an den Nutzenfunktionen und Strategien durchführen können.
Temporal Difference Learning kombiniert Ideen aus der dynamischen Programmierung mit Monte-Carlo Methoden. So können sie beispielsweise ohne ein vorgegebenes Modell aus Erfahrungen lernen und passen ihre Schätzungen direkt an, ohne auf finale Ergebnisse zu warten. Dazu zählen unter anderem *Q-Learning*, der *SARSA Algorithmus* und der *TD-Lambda Algorithmus*.
Diese herkömmlichen Methoden sind allerdings auf Problemstellungen beschränkt, die mit einer überschaubaren Anzahl an Zuständen und Aktionen zu tun haben, da gerade bei der Abschätzung von Nutzenfunktionen Werte für jeden einzelnen Zustand oder jedes einzelne Zustand-Aktionspaar gesetzt werden. Wenn die Mengen allerdings kontinuierlich sind oder sehr komplexe Strukturen enthalten wie Bilder, dann besteht die einzige Möglichkeit zum Lernen darin, aus einer Teilmenge von erfahrenen Zuständen auf noch nicht Gesehenes zu generalisieren. Hier ist die Idee, RL-Methoden mit vorhandenen Methoden zum Generalisieren zu kombinieren. Dafür bieten sich Funktionsapproximatoren an, da sie Beispiele einer gewünschten Funktion nehmen (zum Beispiel einer Nutzenfunktion) und versuchen, daraus die gesamte Funktion zu konstruieren. So kann man beispielsweise die Nutzenfunktion parametrisieren und von einem neuronalen Netzwerk berechnen lassen. Trainiert wird dann standardmäßig mittels überwachten Trainings, wobei Trainingspaare durch die oben erwähnten Algorithmen geliefert werden könnten. Probleme entstehen dann, wenn dynamische Strategien genutzt werden oder auf spontane Zustandsänderungen der Umgebung reagiert werden muss, weil neuronale Netze und viele andere statistische Methoden auf der Annahme statischer Trainingsmengen und Zielfunktionen beruhen.
Eine weitere Möglichkeit sind Gradientenverfahren, wie wir sie vom Prinzip her schon in Kapitel 2.5.1 kennengelernt haben. Auch hier gibt es Varianten, die sich auf die Approximation von Nutzenfunktion beziehen. Wir wollen uns in Kürze aber mit dem sogenannten Strategie-Gradientenverfahren beschäftigen, welches direkt die Strategie des Agenten manipuliert. Dieses Verfahren ist sehr robust und kommt sogar mit einer eingeschränkten Wahrnehmung der Umgebung zurecht.

3.2.3 Eingeschränkte Wahrnehmung

Die allgemeine Herangehensweise beim bestärkenden Lernen ist also, Nutzenfunktionen zu bestimmen oder zu approximieren und anhand dieser eine optimale Strategie abzuleiten. In der Praxis sind diese Methoden aber oft nicht gut umsetzbar, wenn das Problem sehr viele Zustände und Aktionen oder sogar kontinuierliche Mengen enthält. Außerdem

leiden sie unter diversen Beeinträchtigungen. Einerseits versuchen sie, anhand der Nutzenfunktion deterministische Strategien zu finden, obwohl optimale Strategien oft stochastisch sind. Andererseits können beliebig kleine Änderungen in der Nutzenfunktion drastische Änderungen in der Strategie nach sich ziehen, was ein Hauptgrund dafür ist, dass diese Algorithmen oft Probleme mit der Konvergenz zu einem Optimum haben. Es ist auch zu bedenken, dass bestärkendes Lernen in der Praxis häufig mit unvollständigen beziehungsweise verrauschten Daten oder auch mit Nicht-Markov-Signalen zu arbeiten hat. In vielen Umgebungen der realen Welt wird es dem Agenten nicht möglich sein, den Zustand der Umgebung komplett wahrzunehmen. In diesem Fall macht der Agent nur *Beobachtungen* des wahren Zustands der Umgebung, wobei diese Beobachtungen unter Umständen nur unvollständige Informationen liefern. Das resultierende Modell wird *eingeschränkt wahrnehmbarer Markov-Entscheidungsprozess* genannt.

Definition 3.14 (eingeschränkt wahrnehmbarer Markov-Entscheidungsprozess)**.** Ein *eingeschränkt wahrnehmbarer Markov-Entscheidungsprozess* (partially observable Markov decision process, POMDP) erweitert das Konzept eines MDP und besteht aus

- einer Menge $\mathcal{S}$ von Zuständen,
- einer Menge $\mathcal{A}$ von Aktionen,
- *Übergangswahrscheinlichkeiten* $p(s' \mid s, a)$, in einen neuen Zustand s' zu gelangen, abhängig vom aktuellen Zustand s und der gewählten Aktion a,
- einer Belohnungsfunktion R,
- einer Menge $\mathcal{O}$ von Beobachtungen,
- *Beobachtungswahrscheinlichkeiten* $O(o \mid s', a)$, eine Beobachtung $o \in \mathcal{O}$ zu machen, abhängig vom neuen Zustand s' der Umgebung und der gewählten Aktion a

und optional

- einem Diskontierungsfaktor $\gamma \in [0, 1]$ für die Differenzierung zwischen aktuellen und zukünftigen Belohnungen
- sowie einer Startverteilung p_0 der Zustände.

Solche Einschränkungen verhindern die traditionelle Herangehensweise mit Nutzenfunktionen und benötigen komplexere Schätzungen meist in Verbindung mit einem internen Gedächtnis über vergangene Zustände.
Einen naiven Ausweg bilden *reaktive stochastische Strategien* [29], welche Beobachtungen direkt Aktionen zuweisen, ohne die Zustände der Umgebung zu bestimmen. Diese Strategien basieren allerdings auf den Annahmen, dass Zustandsinformationen keine all zu große Rolle spielen und dass das stochastische Verhalten ausreicht, um endlose Schleifen von mehrdeutigen Beobachtungen zu verhindern. Deswegen scheinen Gedächtnis basierte Herangehensweisen sinnvoller und sind essentiell für komplexere

Problemstellungen. Obwohl es intuitiv so scheint, als bräuchte man ein perfektes Gedächtnis, um Probleme wie POMDPs zu lösen, wird ein solches nicht realisierbar sein. Stattdessen kann man *stochastische Strategien mit eingeschränktem Gedächtnis* nutzen, welche Zuständen des *eingeschränkten Gedächtnisses* Wahrscheinlichkeitsverteilungen über Aktionen zuweisen. Eine effektive Möglichkeit, solche Strategien zu trainieren sind *Strategie-Gradientenverfahren*, welche im folgenden Abschnitt eingeführt werden. Allgemein werden wir diese allerdings im Kontext von MDPs besprechen und erst bei Bedarf auf POMDPs erweitern.

3.3 Strategie-Gradientenverfahren

Bei Strategie-Gradientenverfahren werden die Strategien explizit durch ihren eigenen Funktionsapproximator repräsentiert, um unabhängig von der Nutzenfunktion direkt angepasst zu werden. Die Anpassung richtet sich dabei nach dem Gradienten des erwarteten Gewinns bezüglich der Parameter der Strategie, sodass man einen Gradientenanstieg in die Richtung eines höheren Gewinns durchführt. Das Modell des Gewinns hängt wieder von der Problemstellung ab (siehe Kapitel 3.1.2). Für eine durch Parameter θ parametrisierte Strategie π_θ werden die Parameter also durch folgende Vorschrift angepasst:

$$\theta^{(n+1)} = \theta^{(n)} + \Delta\theta^{(n)} = \theta^{(n)} + \eta\nabla_\theta G = \theta^{(n)} + \eta\frac{\partial G}{\partial\theta^{(n)}}, \tag{3.9}$$

wobei $\theta^{(n)}$ der Parametervektor im n-ten Schritt ist, $\Delta\theta^{(n)}$ die Anpassung der Parameter im n-ten Schritt und η eine Lernrate. Die Kernidee ist also, dass wir, anstatt eine Nutzenfunktion zu approximieren, um mit dieser eine deterministische Strategie abzuleiten, direkt eine stochastische Strategie mittels eines unabhängigen Funktionsapproximators mit eigenen Parametern anpassen. Die Strategie kann beispielsweise durch ein neuronales Netzwerk repräsentiert werden, wobei die Eingaben durch das Zustandssignal der Umgebung realisiert werden, die Ausgaben durch eine Wahrscheinlichkeitsverteilung über die Aktionen dargestellt werden und dessen Gewichte den Parametern der Strategie entsprechen.
Strategie-Gradientenverfahren bieten diverse Vorteile gegenüber den herkömmlichen Methoden. Sie haben die Fähigkeit, mit komplexen und kontinuierlichen Zuständen und Aktionen, sowie eingeschränkter Wahrnehmbarkeit, wie sie in POMDPs modelliert ist, umzugehen. Außerdem können sie erweitert werden, um hochdimensionale Probleme zu lösen und sind theoretisch sogar in der Lage, jedes beliebige RL-Problem zu bewältigen. Weiterhin verursachen kleine Veränderungen in den Parametern auch nur kleine Änderungen in der Strategie, da diese stochastischer Natur ist. Dieser Punkt ist gleichzeitig mit verantwortlich dafür, dass die Strategie-Gradientenverfahren garantiert zu einer wenigstens lokal optimalen Strategie konvergieren. Voraussetzungen dafür sind, dass die Schätzung des Gradienten *erwartungstreu* ist, das heißt, dass ihr Erwartungswert dem wahren Wert des Gradienten gleicht und dass die Lernrate η die beiden Bedingungen $\sum_{n=0}^{\infty}\eta_n > 0$ und $\sum_{n=0}^{\infty}\eta_n^2 = \text{const}$ erfüllt, wobei η_n die Lernrate im n-ten Schritt bezeichnet [26]. Zusätzlich sind sie in der Lage, ohne ein gegebenes Modell der

Umgebung zu trainieren (bis auf eine Ausnahme). Letztlich lassen sie sich auf natürliche Weise mit Techniken der Fehlerrückführung in neuronalen Netzen kombinieren, wie wir in Kapitel 4.2 sehen werden. Damit werden wir in der Lage sein, komplexe neuronale Netze mit einer hohen Parameteranzahl zu trainieren, indem wir durch den Gewinn gewichtete *Eignungen* durch rekurrente Verbindungen zeitlich zurück propagieren. Dieses Vorgehen nennt man *rekurrentes Strategie-Gradientenverfahren* (Recurrent Policy Gradients).
Das Hauptproblem der Strategie-Gradientenverfahren besteht darin, eine gute Schätzung des eigentlichen Gradienten $\nabla_\theta G$ zu erhalten. Es haben sich verschiedene Ansätze entwickelt, von welchen wir einige erwähnen und einen genauer besprechen wollen. Eine der ältesten und die mitunter einfachste Herangehensweise sind *Finite-Differenzen-Methoden*. Die Parametrisierung θ der Strategie π_θ wird hier um kleine Beträge $\Delta\theta_i$ $(i \in I)$ angepasst, sodass neue erkundende Strategien entstehen. Für jede Variation $\theta^{(n)} + \Delta\theta_i$ wird dann ein Verlauf erzeugt. Zur Erinnerung, ein Verlauf fasst die bis zu einem bestimmten Zeitpunkt auftretenden Ereignisse, bestehend aus Zuständen, Aktionen und optional Belohnungen zusammen. Die Verläufe liefern Schätzungen für den zu erwartenden Gewinn G_i und der gewünschte Gradient lässt sich dann mittels linearer Regression approximieren. Finite-Differenzen-Methoden können sowohl für deterministische, als auch für stochastische Strategien angewandt werden. Ein Nachteil ist allerdings, dass die Wahl der Variationen die Performance erheblich beeinflussen kann.
Eine andere Möglichkeit bieten *Likelihood-Ratio-Methoden*, welche wir im kommenden Abschnitt genauer behandeln werden. Diese Methoden lassen sich erweitern, indem die Schätzung des Gradienten mit einem Faktor skaliert wird, der von der sogenannten *Fisher-Informationsmatrix* abhängt. Die *Fisher-Information* liefert dabei Aussagen über die bestmögliche Qualität von Parameterschätzungen in einem bestimmten Modell. In diesem Fall spricht man von *natürlichen Strategie-Gradientenverfahren*.

3.3.1 Likelihood-Ratio-Methode

Wir leiten das Verfahren im Kontext eines MDP her. Die Umgebung produziert in jedem Zeitschritt t einen Zustand $s_t \in \mathcal{S}$, welcher vom Agenten wahrgenommen wird. Die Zustandsübergange sind durch dem Agenten unbekannte Übergangswahrscheinlichkeiten $p(s_{t+1} \mid s_t, a_t)$ definiert. Weiterhin führt der Agent im gleichen Zeitschritt eine Aktion $a_t \in \mathcal{A}$ aus, wobei die Aktionen durch eine stochastische Strategie $a_t \sim \pi_\theta(a_t \mid s_t)$ modelliert werden, um erkundende Aktionen zu gewährleisten. Im nächsten Zeitschritt erhält er eine Belohnung $r_{t+1} \in \mathcal{R}$, abhängig vom Zustand der Umgebung und der gewählten Aktion. Diese Interaktionssequenzen erzeugen einen Verlauf $H = H_T = (s_0, a_0, \ldots, s_T, a_T)$, wobei T den Horizont definiert, welcher unendlich sein kann oder im Fall von episodischen Problemen endlich ist. Das Ziel ist nun, die Parameter θ so anzupassen, dass die Strategie π_θ den zu erwartenden Gewinn G maximiert. Die genaue Form des Gewinns spielt für die folgenden Betrachtungen keine Rolle.
Wir gehen davon aus, dass verschiedene Verläufe H durch eine Strategie π_θ generiert werden, dass heißt $H \sim p(H \mid \theta)$ und bezeichnen den Gewinn eines einzelnen Verlaufs

H mit $G(H)$. Damit lässt sich die Größe, die der Algorithmus maximieren soll zu

$$G = \mathbb{E}_H[G(H)] = \int_{\mathcal{H}} p(H \mid \theta) G(H) dH \tag{3.10}$$

umformulieren. Dies bezeichnet den erwarteten Gewinn über allen möglichen Verläufen $\mathcal{H}$, gewichtet durch ihre Wahrscheinlichkeiten unter der Strategie π_θ. Um an den Gradienten $\nabla_\theta G$ zu gelangen, nutzen wir einerseits den *Likelihood-Ratio-Trick*

$$\nabla_\theta p(H \mid \theta) = p(H \mid \theta) \nabla_\theta \log p(H \mid \theta), \tag{3.11}$$

welcher direkt mit der Kettenregel hergeleitet werden kann: $\nabla_\theta \log p(H \mid \theta) = \frac{\nabla_\theta p(H|\theta)}{p(H|\theta)}$ und andererseits nutzen wir den Fakt, dass $\nabla_\theta G(H) = 0$ für ein fixiertes H gilt. Damit ergibt sich

$$\begin{aligned}
\nabla_\theta G &= \nabla_\theta \int_{\mathcal{H}} p(H \mid \theta) G(H) dH && (3.12)\\
&= \int_{\mathcal{H}} \nabla_\theta p(H \mid \theta) G(H) + \nabla_\theta G(H) p(H \mid \theta) dH && (3.13)\\
&= \int_{\mathcal{H}} \nabla_\theta p(H \mid \theta) G(H) dH && (3.14)\\
&= \int_{\mathcal{H}} p(H \mid \theta) \nabla_\theta \log p(H \mid \theta) G(H) dH && (3.15)
\end{aligned}$$

und der Gradient kann als Erwartungswert dargestellt werden:

$$\nabla_\theta G = \mathbb{E}_H[\nabla_\theta \log p(H \mid \theta) G(H)]. \tag{3.16}$$

Der Erwartungswert kann dann durch eine *Monte-Carlo-Simulation* approximiert werden, indem man K Beispielverläufe H^k generiert und das gewünschte arithmetische Mittel bildet:

$$\nabla_\theta G \approx \frac{1}{K} \sum_{k=1}^{K} \nabla_\theta \log p(H^k \mid \theta) G(H^k). \tag{3.17}$$

Somit benötigt man nur noch den Gradienten $\nabla_\theta \log p(H^k \mid \theta)$ für die eigentliche Abschätzung des Gradienten $\nabla_\theta G$. Allerdings hat man keine direkten Kenntnisse über die Verteilung $p(H \mid \theta)$, da die Wahrscheinlichkeiten verschiedener Verläufe von einer unbekannten Initialverteilung der Zustände, sowie dem Agenten unbekannten Übergangswahrscheinlichkeiten $p(s_{t+1} \mid s_t, a_t)$ abhängen. Ist das System Markov, kann man die Wahrscheinlichkeit eines bestimmten Verlaufs H als Produkt aller Aktionen und Zustände, gegeben der vorigen Ereignisse darstellen

$$p(H \mid \theta) = p(s_0) \prod_{t=0}^{T} p(s_{t+1} \mid s_t, a_t) \pi_\theta(a_t \mid s_t) \tag{3.18}$$

und es fällt auf, dass die meisten Terme nicht von θ beeinflusst werden. Tatsächlich sind es nur die Wahrscheinlichkeiten der Aktionen, welche durch die Strategie π_θ von θ

abhängen und diese sind dem Agenten bekannt. Im Kontext des Gradienten ∇_θ kann man alle restlichen Terme also als Konstanten betrachten. Der Logarithmus wandelt das Produkt (3.18) zunächst in eine Summe um: $\log p(H \mid \theta) = (\text{const}) + \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t)$ und man erhält für den Gradienten

$$\nabla_\theta \log p(H \mid \theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t). \tag{3.19}$$

Also müssen die Gradienten von $p(s_{t+1} \mid s_t, a_t)$ nicht berechnet werden, weshalb diese Methode ohne ein Modell der Umgebung auskommt. Würde man anstatt einer stochastischen Strategie $a \sim \pi(a \mid s)$ eine deterministische Strategie $a = \pi(s)$ nutzen, müsste man die Gradienten $\nabla_\theta \log p(s_{t+1} \mid s_t, a_t) = \nabla_{a_t} \log p(s_{t+1} \mid s_t, a_t) \nabla_\theta \pi_\theta(s_t)$ berechnen, um an den Gradienten $\nabla_\theta \log p(H \mid \theta)$ zu gelangen. In diesem Fall würde man ein Modell der Umgebung benötigen.
Substituiert man (3.19) in (3.17), so erhält man schließlich die erwartungstreue Schätzung für den Strategie-Gradienten, welcher nur von wahrnehmbaren Größen abhängt und besser bekannt ist als REINFORCE [35]:

$$\nabla_\theta G \approx \frac{1}{K} \sum_{k=1}^{K} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) G(H^k). \tag{3.20}$$

Es ist anzumerken, dass man bei der Berechnung der Gewinne $G(H^k)$ nur die Belohnungen ausgehend vom Zeitschritt t berücksichtigen muss, da zukünftige Aktionen nicht von vergangenen Belohnungen abhängen. Ein Problem von REINFORCE ist, dass eine hohe Varianz in die Approximation des Gradienten eingeführt werden kann. Um diese zu reduzieren, lässt sich eine konstante *Baseline* $b \in \mathbb{R}$ in (3.20) integrieren, welche generell beliebig gewählt werden kann [35]:

$$\nabla_\theta G \approx \frac{1}{K} \sum_{k=1}^{K} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)(G(H^k) - b). \tag{3.21}$$

Typischerweise ist b der durchschnittlich erwartete Gewinn oder eine Art von (abgeschätzter) Nutzenfunktion. Aufgrund des Likelihood-Ratio-Tricks gilt

$$\int_{\mathcal{H}} p(H \mid \theta) \nabla_\theta \log p(H \mid \theta) dH = \int_{\mathcal{H}} \nabla_\theta p(H \mid \theta) dH = \nabla_\theta 1 = 0, \tag{3.22}$$

sodass wir

$$\mathbb{E}_H[\sum_{k=1}^{K} \nabla_\theta \log p(H^k \mid \theta)\, b] = 0 \tag{3.23}$$

garantieren können und die Gradientenschätzung erwartungstreu bleibt. Die Baseline verschwindet also für unendliche Datenmengen, während sie die Varianz der Approximation für endliche Datenmengen verringert.
Likelihood-Ratio-Methoden bieten den Vorteil, dass keine Variationen von Parame-

tern erzeugt werden müssen, welche die Performance beeinflussen könnten. Weiterhin hat sich herausgestellt, dass in der Praxis oft schon ein einziger Beispielverlauf ausreicht, um eine erwartungstreue Abschätzung des Gradienten zu erhalten ([2], [30]), wodurch die Laufzeit verbessert wird. Außerdem erreichen Likelihood-Ratio-Methoden die theoretisch schnellstmögliche Konvergenz des Fehlers für ein stochastisches System mit $O(K^{-\frac{1}{2}})$ [8].
Will man Likelihood-Ratio-Methoden allerdings mit deterministischen Strategien kombinieren, dann ist man auf ein Modell der Umgebung angewiesen, wie wir oben gesehen haben. So ein Modell ist schwierig aufrecht zu erhalten, wenn man mit kontinuierlichen Zuständen und Aktionen arbeitet. In diesem Fall greift man lieber auf Finite-Differenzen-Methoden zurück.

4 Das Aufmerksamkeitsmodell

Wir modellieren das *Aufmerksamkeitsproblem* als sequentiellen Entscheidungsprozess eines zielgerichteten Agenten, der mit seiner visuellen Umgebung interagiert. Zu keinem Zeitpunkt nimmt der Agent den Zustand seiner Umgebung, welche sowohl statisch als auch dynamisch sein kann, vollständig wahr. Er macht hingegen nur eingeschränkte Beobachtungen mittels eines visuell limitierten Sensors und kann somit nur Informationen aus lokalen Regionen extrahieren. Der Agent hat dabei die volle Kontrolle über den Sensor, das heißt er entscheidet, auf welche Position sich dieser fokussieren soll. Unter diesen Bedingungen ist der Agent gezwungen, Informationen aus vergangenen Beobachtungen zu speichern beziehungsweise intern zu kombinieren, um zukünftige Entscheidungen treffen zu können. Der Agent ist auch in der Lage, Aktionen auszuführen, die beispielsweise in dynamischen Szenarien die Umgebung beeinflussen oder den Klassifikationen beziehungsweise Lokalisierungen entsprechen. Diese Aktionen werden potentiell in jedem Zeitschritt mit einer skalaren Belohnung bewertet und der Agent versucht die Summe all dieser Belohnungen (seinen Gewinn) zu maximieren. Allgemein gliedert sich die Verarbeitung eines Bildes I mit dem Aufmerksamkeitsmodell also in einen sequentiellen Prozess, bestehend aus T Schritten. In jedem Zeitschritt t erhält das Modell eine Position l_{t-1}, auf die der Sensor fokussiert wird. Dieser liefert eine Beobachtung I_t der Umgebung an der Position l_{t-1}. Die Beobachtung wird genutzt, um den internen Zustand des Modells zu aktualisieren und die nächste zu fokussierende Position l_t zu bestimmen. Optional wird eine Aktion a_t ausgeführt, die von der Problemstellung abhängt.
Diese Formulierung ist allgemein genug, um diverse Problemstellungen wie Objektlokalisierungen und -klassifikationen in statischen Umgebungen und Kontrollprobleme, wie das Spielen eines Videospiels, zu umfassen. In unserer Arbeit beschränken wir uns allerdings auf Klassifikations- und Lokalisierungsprobleme in statischen Umgebungen (in der Form von Bildern). Damit ist der Zustand der Umgebung fixiert. Aktionen des Agenten entsprechen in diesem Fall den verschiedenen Klassifizierungen oder kontinuierlichen Ausgaben bei einer Lokalisierung. Belohnungen spiegeln dann die Korrektheit der Klassifikation beziehungsweise den Grad der Korrektheit bei Schätzungen wider.
Das Aufmerksamkeitsmodell kann in verschiedene Komponenten unterteilt werden. Wir werden diese als Netzwerke bezeichnen, da sie grundsätzlich durch künstliche neuronale Netze modelliert sind. Im nächsten Abschnitt werden wie die einzelnen Bestandteile näher beschreiben. Danach erläutern wir, wie genau das Training abläuft und auf welche Verfahren dabei zurückgegriffen wird.

4.1 Architektur

Der Kern des Modells entspricht einem rekurrenten neuronalen Netz, welches um weitere Komponenten erweitert wird.

Glimpse-Sensor

In jedem Zeitschritt t erhält das Modell eine eingeschränkte Beobachtung I_t der Umgebung. Diese Beobachtung wird von einem visuellen Sensor ρ geliefert, welcher sich auf eine gegebene Position l_{t-1} fokussiert. Der Sensor erfasst einen Bildausschnitt $I_t = \rho(I, l_{t-1})$ um die Position l_{t-1}, welcher annähernd der Funktionsweise der Netzhaut des Menschens nachempfunden ist. Dabei wird die Region direkt um l mit einer hohen Auflösung extrahiert und weiter von l entfernte Bereichen werden mit einer zunehmend geringeren Auflösung extrahiert. Die Ausgabe des Sensors entspricht dann einem Vektor wesentlich geringerer Dimension als der des ursprünglichen Bildes.

Definition 4.1 (Glimpse)**.** Die Repräsentation eines Bildausschnitts mit einer hoch auflösenden Aufnahme im Zentrum um eine gegebene Position und einer abnehmenden Auflösung nach außen bezeichnen wir als *Glimpse* (glimpse, [20]).

Im Spezifischen extrahiert der Glimpse-Sensor k Bildausschnitte, die um l zentriert sind. Dabei wird die Größe $s_x \times s_y$ des ersten Ausschnitts in Pixeln vorgegeben und jeder weitere Ausschnitt verdoppelt die Größe des Vorherigen. Im Anschluss werden alle k Ausschnitte auf $s_x \times s_y$ skaliert und zusammengefügt. Die spezifische Wahl der Größen s_x und s_y hängt von der Problemstellung ab. Die Glimpse-Position l wurde als zweidimensionaler Vektor $l = (x, y) \in [-1, 1]^2$ kodiert, wobei der Punkt $(0, 0)$ dem Zentrum des Bildes entspricht und der Punkt $(-1, -1)$ der linken oberen Ecke des Bildes entspricht. Die allgemeine Funktionsweise des Glimpse-Sensors ist in Abbildung 4.1 modelliert.

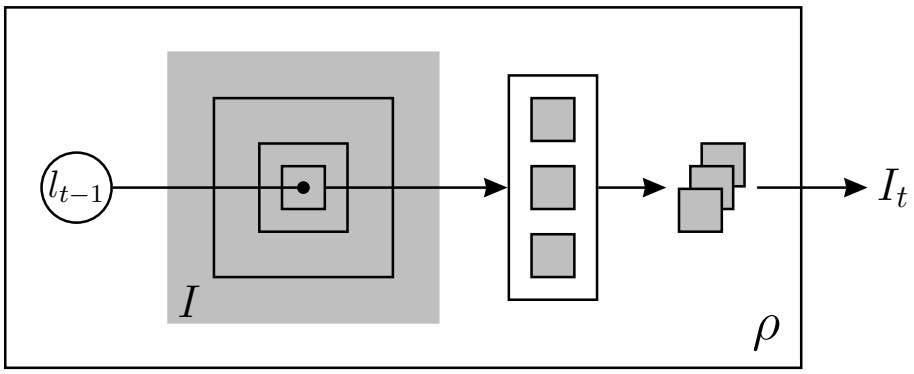


Abbildung 4.1: Aufbau des Glimpse-Sensors ρ. Bei gegebener Glimpse-Position l_{t-1} und Eingabebild I extrahiert der Sensor einen Glimpse $I_t = \rho(I, l_{t-1})$, der mehrere Auflösungen enthält und um l_{t-1} zentriert ist.

Glimpse-Netzwerk

Der Glimpse-Sensor wird in das sogenannte Glimpse-Netzwerk N_g integriert, welches die Aufgabe hat, nützliche Merkmale von der fokussierten Position des Bildes zu extrahieren. Es nutzt den vom Sensor erzeugten Glimpse I_t und die zugehörige Position l_{t-1} als Eingaben. Jeder dieser Vektoren wird in voneinander unabhängige verborgene Räume abgebildet, indem das Netz unabhängige verborgene, durch θ_g^0 beziehungsweise θ_g^1 parametrisierte, voll-verbundene Schichten implementiert. Diese Schichten haben die gleiche Dimension und nutzen stets ReLUs als nicht lineare Aktivierungsfunktionen. In unseren Experimenten wurden für die beiden Schichten jeweils 128 verborgene Neuronen gewählt. Die Informationen der beiden verborgenen Schichten werden dann durch eine lineare Schicht θ_g^2 kombiniert, welche den Merkmalsvektor g_t erzeugt, der bei uns stets die Dimension 256 hatte. Fasst man die verschiedenen Parameter des Glimpse-Netzwerks zu $\theta_g = \{\theta_g^0, \theta_g^1, \theta_g^2\}$ zusammen, lässt sich die Ausgabe des Netzes folgendermaßen formulieren:

$$g_t = N_g(I_t, l_{t-1} \mid \theta_g). \tag{4.1}$$

Die Zusammensetzung des Glimpse-Netzwerks wird in Abbildung 4.2 verdeutlicht.

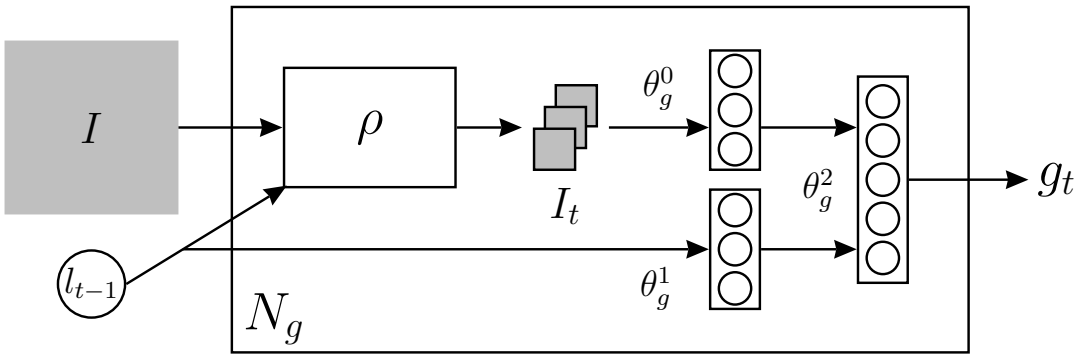


Abbildung 4.2: Aufbau des Glimpse-Netzwerks N_g. Mithilfe des Glimpse-Sensors wird in Abhängigkeit vom Eingabebild I und der Glimpse-Position l_{t-1} der Glimpse I_t erzeugt. Dieser und die Glimpse-Position werden in zwei voneinander unabhängige verborgene Schichten θ_g^0 und θ_g^1 abgebildet. Eine weitere lineare Schicht θ_g^2 kombiniert diese beiden Schichten und gibt den Glimpse-Vektor g_t aus.

Kernnetzwerk

Wie bereits erwähnt, muss das Modell einen internen Zustand (ein internes Gedächtnis) verwalten, um Informationen aus vorigen Zeitschritten zusammenfassen zu können. Damit kodiert es das Wissen des Agenten über seine Umgebung, sodass dieser bessere Entscheidungen in der Zukunft treffen kann. Realisiert wird dieses Konzept durch die verborgenen Einheiten h_t des rekurrenten Kernnetzwerks N_h, welche im Laufe der Zeit aktualisiert werden. Als Eingabe für das Kernnetzwerk dient der Merkmalsvektor g_t des

Glimpse-Netzwerks:

$$h_t = N_h(h_{t-1}, g_t \mid \theta_h). \tag{4.2}$$

Für unsere Experimente wurde das Kernnetzwerk stets aus einer einzelnen, voll-verbundenen, rekurrenten Schicht mit ReLUs als nicht lineare Aktivierungsfunktionen geformt. In unseren Experimenten hatte der interne Zustand stets die Dimension 256.

Positionsnetzwerk

In jedem Zeitschritt t muss der Agent entscheiden, auf welche Position l_t der Glimpse-Sensor als nächstes fokussiert werden soll. Dieser Vorgang wird durch das Positionsnetzwerk N_l realisiert, welches den internen Zustand h_t des Kernnetzwerks als Eingabe nutzt und eine bivariate Normalverteilung parametrisiert. Die Normalverteilung hat eine fixierte Varianz und der Mittelwert wird durch das Positionsnetzwerk in jedem Schritt ausgegeben. Die nächste Position wird anschließend stochastisch gezogen:

$$l_t \sim p(l_t \mid N_l(h_t \mid \theta_l)). \tag{4.3}$$

Das Positionsnetzwerk kontrolliert somit, worauf die Aufmerksamkeit des Modells gelenkt wird. Es besteht aus einer einzelnen, voll-verbundenen Schicht, welche einen *harten Tangens Hyperbolicus* als Aktivierungsfunktion implementiert, das heißt eine stückweise lineare Funktion auf $(-\infty, -1) \cup [-1, 1] \cup (1, \infty)$, die den Graphen des Tangens Hyperbolicus approximiert.

Aktionsnetzwerk

Das Aktionsnetzwerk N_a (Klassifikationsnetzwerk) realisiert die verschiedenen Aktionen des Agenten, welche von der Problemstellung abhängen. Bei Klassifikationen werden die Aktionen ähnlich den Positionen stochastisch aus einer Verteilung gezogen:

$$a_t \sim p(a_t \mid N_a(h_t \mid \theta_a)), \tag{4.4}$$

wobei die Verteilung durch das Aktionsnetzwerk parametrisiert wird und durch eine voll-verbundene Schicht vom internen Zustand des Kernnetzwerks abhängt. Für gewöhnlich wird dafür eine Softmax-Funktion als Ausgabe verwendet, die für jede Klasse angibt, wie wahrscheinlich eine Zuordnung der aktuellen Eingabe zu dieser ist. Bei Lokalisierungen hingegen nutzt man einfach eine lineare Schicht, welche die Position des gesuchten Objektes oder den Mittelpunkt der gesuchten Region ausgibt (wieder transformiert auf $[-1, 1]$). In diesen statischen Szenarien führt der Agent meist erst im finalen Zeitschritt T die Klassifikation beziehungsweise Lokalisierung aus, während er in allen vorigen Zeitschritten nur die nächste Position des Sensors wählt.

Kontextnetzwerk

In Anlehnung an [1] wurde auch mit einem optionalen Kontextnetzwerk N_c experimentiert. Dieses liefert den Initialzustand des rekurrenten Kernnetzwerks und wird gleich-

zeitig für die Bestimmung der ersten Glimpse-Position genutzt. Es erhält das gesamte Bild als Eingabe und versucht Kontextinformationen über möglicherweise interessante Regionen des Bildes zu extrahieren. Aufgebaut wurde es aus zwei Faltungsschichten, die jeweils von einer Pooling-Schicht gefolgt wurden und einer abschließenden vollverbundenen Schicht, die den Ausgabevektor erzeugt.

Eine allgemeine graphische Darstellung des (entfalteten) Aufmerksamkeitsmodells ist in Abbildung 4.3 zu sehen.

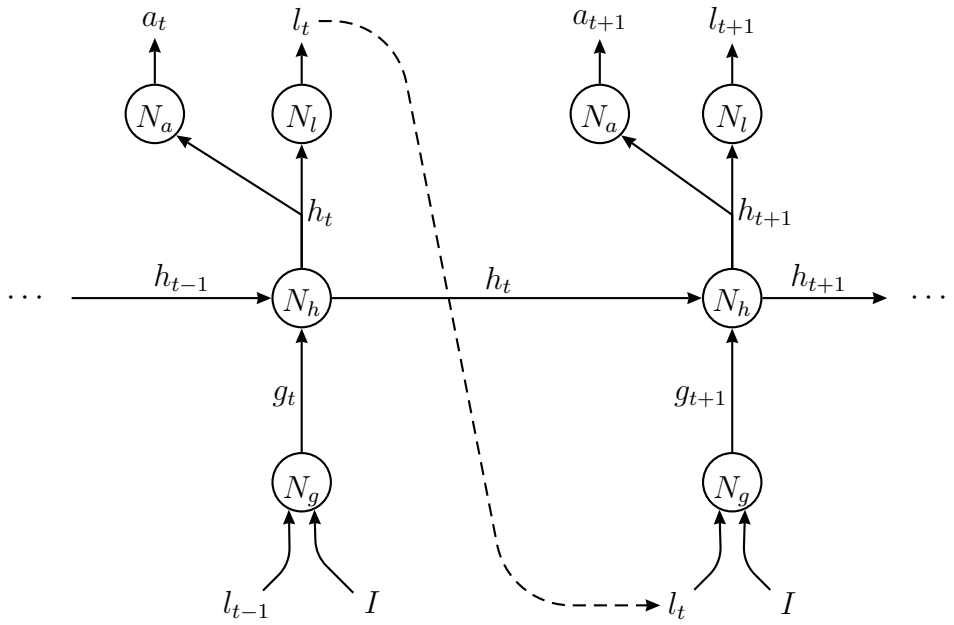


Abbildung 4.3: Aufbau des zeitlich entfalteten rekurrenten Aufmerksamkeitsmodells. Das Glimpse-Netzwerk N_g nutzt die Glimpse-Position l_{t-1} und das Eingabebild I, um den Glimpse-Vektor g_t zu bestimmen. Das Kernnetzwerk N_h bildet mit seinem vorigen Zustand h_{t-1} und dem aktuellen Glimpse-Vektor g_t seinen neuen internen Zustand h_t. Dieser dient als Eingabe für das Aktionsnetzwerk N_a und das Positionsnetzwerk N_l, welche die aktuelle Aktion a_t und die neue Glimpse-Position l_t erzeugen. Die gestrichelte Linie indiziert, dass die Position l stochastisch aus der parametrierten Verteilung gezogen wird. Das optionale Kontextnetzwerk N_c ist nicht enthalten.

4.2 Training

Die Konstruktion und Beschreibungen wurden so gewählt, dass sich das Aufmerksamkeitsproblem als POMDP in das RL-Framework einbetten lässt. Der wahre Zustand der Umgebung wird nicht vom Agenten wahrgenommen, da er zu jedem Zeitpunkt nur einen Glimpse erhält. Er hat das Ziel, eine (stochastische) Strategie $\pi_\theta((l_t, a_t) \mid H_t)$ zu lernen, die abhängig vom bisherigen Verlauf H_t eine Wahrscheinlichkeitsverteilung über die möglichen Aktionen (l_t, a_t) des aktuellen Zeitschritts definiert. In unserem Fall wird die Strategie π_θ durch das gesamte Aufmerksamkeitsmodell repräsentiert, wobei

die Verteilungen für die Positionen l_t und die Aktionen a_t durch das Positionsnetzwerk und das Aktionsnetzwerk geliefert werden. Der bisherige Verlauf H_t wird dabei durch den internen Zustand des rekurrenten Kernnetzwerks zusammengefasst und die Parameter θ umfassen die gesamten Parameter aller enthaltenen Netze. Ziel des Agenten ist es nun, seinen Gewinn $G = \sum_{t=0}^{T} r_t$, also die Summe all seiner Belohnungen, zu maximieren, weshalb es nahe liegt, Verfahren aus dem bestärkenden Lernen anzuwenden. Als Gewinn haben wir das Modell mit endlichem Horizont gewählt, da das Aufmerksamkeitsproblem episodisch ist, wobei eine Episode durch die Anzahl an vorgegebenen Glimpses beschränkt ist. Wir nutzen das Strategie-Gradientenverfahren aus Kapitel 3.3 und kombinieren dieses mit den Strukturen neuronaler Netzwerke, um das Aufmerksamkeitsmodell zu trainieren.

4.2.1 Rekurrentes Strategie-Gradientenverfahren

Wir wollen die Parameter θ direkt anpassen, sodass die Wahrscheinlichkeit gewählter Aktionen und dementsprechend Verläufen, die zu einem hohen Gewinn geführt haben, erhöht werden, während die, die zu einem geringen Gewinn geführt haben, verringert werden. Dazu führen wir den Gradientenanstieg (3.9) durch. Um den Gradienten $\nabla_\theta G$ abzuschätzen, wollen wir die Likelihood-Ratio-Methode nutzen. In Kapitel 3.3.1 haben wir diese allerdings nur im Kontext von MDPs eingeführt. Bei unserem Problem handelt es sich um ein POMDP, weshalb wir ein paar Änderungen vornehmen müssen, wobei sich das Grundgerüst des Algorithmus nicht verändert.
Anstatt, dass der Agent in jedem Zeitschritt t den wahren Zustand s_t der Umgebung wahrnimmt, erhält er jetzt nur noch Beobachtungen o_t in der Form von Glimpses, welche unvollständige Informationen enthalten, da sie nur Bildausschnitten mit abnehmender Auflösung entsprechen. Wir nehmen an, dass die Beobachtungen o_t durch Beobachtungswahrscheinlichkeiten $p(o_t \mid s_t, a_{t-1})$ bestimmt werden, die vom aktuellen Zustand der Umgebung und der gewählten Aktion des Agenten abhängen. Es ist zu beachten, dass die Aktionen a_t in diesem Kontext zum RL-Framework gehören und nicht mit den Ausgaben des Aktionsnetzwerks zu verwechseln sind. Weiterhin erhält der Agent in jedem Zeitschritt t eine Belohnung r_t. Wir verallgemeinern die Herleitung sogar noch weiter, indem wir auch Nicht-Markov Zustände zulassen, da diese Annahme für Strategie-Gradientenverfahren mit einem internen Gedächtnis nicht nötig ist. In diesem Fall sind die Übergangswahrscheinlichkeiten $p(s_{t+1} \mid s_{0:t}, a_{0:t})$ der Zustandsübergänge dem Agenten immer noch unbekannt, hängen aber nun von allen vorigen Aktionen $a_{0:t} = (a_0, \ldots, a_t)$ des Agenten und allen vorigen Zuständen $s_{0:t} = (s_0, \ldots, s_t)$ der Umgebung ab. In diesem Kontext erweitern wir das Konzept klassischer Verläufe und führen *Teilverläufe* ein.

Definition 4.2 (Teilverlauf)**.** Ein *Teilverlauf* setzt sich aus allen Beobachtungen o und Aktionen a bis zum aktuellen Zeitpunkt t zusammen:

$$\widetilde{H}_t = (o_0, a_0, \ldots, o_{t-1}, a_{t-1}, o_t). \tag{4.5}$$

Der Gesamtverlauf H enthält zusätzlich die wahren Zustände der Umgebung und ist durch $H = H_T = (\widetilde{H}_T, s_{0:T})$ gegeben.

Optimale Strategien in POMDPs oder in Nicht-Markov Umgebungen setzten theoretisch voraus, dass die Aktionen a_t in Abhängigkeit vom gesamten Teilverlauf $\widetilde{H}_t$ gewählt werden. Für gewöhnlich reicht es aber aus, wenn man anstatt der kompletten Sequenz von Beobachtungen und Aktionen ausreichende Informationen $M(\widetilde{H}_t)$ der vergangenen Ereignisse speichert. Diese Informationen $M(\widetilde{H}_t)$ lassen sich im internen Zustand des Aufmerksamkeitsmodells zusammenfassen und wir bezeichnen diesen als *eingeschränktes Gedächtnis* des Agenten. Damit lässt sich eine stochastische Strategie als $\pi_\theta(a_t \mid \widetilde{H}_t) = p(a_t \mid M(\widetilde{H}_t), \theta)$ definieren, welche von unserem rekurrenten Aufmerksamkeitsmodell repräsentiert wird, mit Gewichten θ und stochastisch interpretierbaren Ausgabeneuronen. Die Aktionen werden dann aus der erzeugten Wahrscheinlichkeitsverteilung gezogen: $a_t \sim \pi_\theta(a_t \mid \widetilde{H}_t)$.
Die Herleitung des Verfahren verläuft analog zu Kapitel 3.3.1. Wir gehen davon aus, dass verschiedene Verläufe H durch die Strategie π_θ generiert werden, das heißt $H \sim p(H \mid \theta)$ und bezeichnen den Gewinn eines einzelnen Verlaufs H mit $G(H)$. Den zu maximierenden Gewinn G formulieren wir dann mittels

$$G = \mathbb{E}_H[G(H)] = \int_{\mathcal{H}} p(H \mid \theta) G(H) dH \tag{4.6}$$

und nutzen den Likelihood-Ratio-Trick, sowie eine Monte-Carlo-Simulation, um die Abschätzung

$$\nabla_\theta G \approx \frac{1}{K} \sum_{k=1}^{K} \nabla_\theta \log p(H^k \mid \theta) G(H^k) \tag{4.7}$$

für den gesuchten Gradienten $\nabla_\theta G$ zu erhalten. Die Wahrscheinlichkeiten verschiedener Verläufe $p(H \mid \theta)$ hängen hier neben einer unbekannten Initialverteilung der Zustände und dem Agenten unbekannten Übergangswahrscheinlichkeiten $p(s_{t+1} \mid s_{0:t}, a_{0:t})$ noch zusätzlich von unbekannten Beobachtungswahrscheinlichkeiten in jedem Zustand ab. Die Wahrscheinlichkeit eines bestimmten Verlaufs H entspricht in diesem Fall dem Produkt aller Aktionen, Zustände und Beobachtungen, gegeben aller Teilverläufe:

$$p(H \mid \theta) = p(H_0) \prod_{t=0}^{T} p(o_{t+1} \mid s_{t+1}, a_t) p(s_{t+1} \mid s_{0:t}, a_{0:t}) \pi_\theta(a_t \mid \widetilde{H}_t) \tag{4.8}$$

Die meisten Terme werden nicht von θ beeinflusst und somit gilt

$$\nabla_\theta \log p(H \mid \theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid \widetilde{H}_t). \tag{4.9}$$

Substituiert man (4.9) in (4.7), so erhält man die gewünschte Approximation für den Strategie-Gradienten, welcher nur von Größen des eingeschränkten Gedächtnisses abhängt:

$$\nabla_\theta G \approx \frac{1}{K} \sum_{k=1}^{K} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid \widetilde{H}_t^k) G(H^k). \tag{4.10}$$

Die Bemerkungen zu den Belohnungen und der Varianzreduktion übertragen sich aus unseren früheren Betrachtungen, sodass der Gradient folgende Form annimmt:

$$\nabla_\theta G \approx \frac{1}{K} \sum_{k=1}^{K} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid \widetilde{H}_t^k)(G(H^k) - b). \tag{4.11}$$

Wählt man als Baseline b den erwarteten Gewinn unter der aktuellen Strategie, so kann man dem Aufmerksamkeitsmodell ein *Baseline-Netzwerk* N_b hinzufügen, welches eine voll-verknüpfte Schicht implementiert und abhängig vom internen Zustand des Kernnetzwerks die Baseline ausgibt:

$$b_t = N_b(h_t \mid \theta_b). \tag{4.12}$$

Dieses lernt die Baseline b, indem es den quadratischen Fehler zwischen $G(H)$ und b mittels Fehlerrückführung minimiert. Einfacher und schneller ist es, wenn man eine exponentielle Glättung vornimmt:

$$b^k = \gamma G(H^{k-1}) + (1 - \gamma) b^{k-1}, \tag{4.13}$$

wobei b^k die Baseline im Verlauf H^k bezeichnet ($b^0 = 0$) und $\gamma \in (0, 1]$ den Anteil des Gewinns an der neuen Baseline festlegt. Diese Herangehensweise ist allerdings nicht optimal ([11], [33]), da die optimale Baseline formal als Minimierung der Varianz der Gradientenschätzung bezüglich aller möglichen Baselines definiert ist. Dennoch konnten wir bei unseren Experimenten mit dem Baseline-Netzwerk und der exponentiellen Glättung gute Ergebnisse erzielen.
Unser Aufmerksamkeitsmodell ist aufgrund seiner rekurrenten Verbindungen in der Lage, Zuständen des eingeschränkten Gedächtnis Aktionswahrscheinlichkeiten zuzuordnen, das heißt es trifft Entscheidungen auf der Grundlage von Verläufen anstatt von Beobachtungen. Das rekurrente Strategie-Gradientenverfahren ermöglicht es uns, nun alle Parameter θ gleichzeitig zu optimieren, indem wir durch den Gewinn gewichtete Eignungen (eligibilities, siehe [35]) zeitlich durchs Modell zurück propagieren (eligibility-backpropagation through time, [34]). Damit werden Verläufe, die einen hohen Gewinn erzeugt haben, wahrscheinlicher als solche, die einen geringen Gewinn erzeugt haben.
Der Trainingsverlauf gleicht strukturell dem von BPTT. In der Vorwärtsphase erhält das Modell in jedem Zeitschritt t die Beobachtung o_t und die Belohnung r_t als Eingaben. Zusammen mit den rekurrenten Verbindungen produziert es dann die Wahrscheinlichkeitsverteilungen der Aktionen. Da nur die Ausgabeneuronen stochastisch sind, reicht es in der Rückwärtsphase aus, die Eignungen an den Ausgabeneuronen zu approximieren. Die Gradienten der anderen Parameter können dann wie gewohnt durch BPTT berechnet werden, indem wir die Eignungen als normale Fehler (Delta-Terme) interpretieren.

4.2.2 Hybrides überwachtes Lernen

Das rekurrente Strategie-Gradientenverfahren ist so aufgebaut, dass die Lernimpulse durch die akkumulierten Belohnungen des Agenten geliefert werden, wenn die optimalen Aktionen an sich nicht bekannt sind. Beispielsweise wird man im Vorhinein nicht

wissen, welche Sequenz an Fokussierungen der Glimpse-Sensor durchlaufen sollte, um dem Agenten die bestmöglichen Informationen über ein unbekanntes Bild zu liefern. In dynamischen Szenarien, in denen Aktionen des Agenten sogar die Umgebung beeinflussen können, wird ebenso unklar sein, was auf lange Sicht den größten Erfolg erzielt. Nach dem Ende einer Episode und somit nach Ende eines Verlaufs, liefert uns der Gewinn allerdings einen Indikator dafür, wie gut die jeweilige Sequenz war. Diese Herangehensweise eignet sich also, um das Positionsnetzwerk N_l zu trainieren, da es solche Positionssequenzen liefern soll, die den Glimpse-Sensor auf Regionen fokussiert, welche nützliche Informationen zur Klassifikation beziehungsweise Lokalisierung der gewünschten Objekte enthalten. Die approximierten Eignungen an den Ausgaben des Positionsnetzwerk lassen sich dann nutzen, um entweder ausschließlich die Gewichte des Positionsnetzwerks anzupassen oder um weiter durch die restlichen Komponenten des Aufmerksamkeitsmodells propagiert zu werden.
Für die Klassifikationen beziehungsweise Lokalisierungen kennen wir zur Trainingszeit hingegen die richtigen Aktionen. So ist für die Trainingsdaten die korrekte Klassifizierung oder das Objekt beziehungsweise die Region, die lokalisiert werden soll, bereits bekannt. In diesen Fällen brauch man nicht auf durch die Belohnungen induzierte Lernimpulse zurückgreifen, sondern kann die Strategie mit herkömmlichen Methoden in einer Instanz des überwachten Lernens optimieren. Mithilfe der vorgegebenen korrekten Aktion a_T^* am Ende einer Episode maximiert man die durch den zugehörigen Verlauf bedingte Wahrscheinlichkeit $\pi_\theta(a_t^* \mid H_T)$ der gewünschten Aktion, indem man eine Zielfunktion wie die Kreuzentropie optimiert. Diese Herangehensweise eignet sich, um das Aktionsnetzwerk N_a zu trainieren, wobei die Gradienten stets durch die restlichen Komponenten des Aufmerksamkeitsmodells, sprich durch das Kern- und Glimpse-Netzwerk, zurück propagiert werden.

Definition 4.3 (Hybrides überwachtes Lernen)**.** Beim *hybriden überwachten Lernen* kombinieren wir das Konzept des überwachten Lernens mit dem des bestärkenden Lernens. Komponenten, für die zur Trainingszeit korrekte Lösungen bekannt sind, werden durch die Optimierung einer Zielfunktion trainiert. Komponenten, für die keine optimalen Aktionen bekannt sind, erhalten Lernimpulse aus akkumulierten Belohnungen, welche einer Bewertung für die Performance des Modells entsprechen.

Dieses Konzept des hybriden überwachten Lernens verbindet die Vorteile des überwachten Lernens mit der Flexibilität des bestärkenden Lernens. Die eigentlich wichtigen Ausgaben des Aktionsnetzwerks werden wie gewohnt überwacht trainiert und können somit auf spezielle Problemstellungen angepasst werden. Die nötigen Merkmale aus den vorgegeben Datenmengen erschließt sich das Modell dann eigenständig. Das Positionsnetzwerk, was sozusagen nur ein Mittel zum Zweck ist, lernt wichtige Regionen des Bildes zu fokussieren, indem es Verläufe mit hohem Gewinn bevorzugt. Beim bestärkenden Lernen werden dem Agenten keinen Vorschriften gemacht. Er lernt, sich optimal zu verhalten, ganz einfach aus Erfahrung, indem er mit seiner Umgebung interagiert und kann sich somit an die verschiedensten Problemstellungen anpassen.

4.2.3 Bayessche Variationsmethoden

In diesem Abschnitt wollen wir einen Ansatz nutzen, der nicht aus dem bestärkenden Lernen stammt, um ein solches Lernverfahren herzuleiten. Bayessche Variationsmethoden beschreiben eine Familie von Techniken zur Approximation hartnäckiger Integrale, wie sie beispielsweise im maschinellen Lernen vorkommen. Sie werden typischerweise in komplexen statistischen Modellen eingesetzt, welche sich aus wahrnehmbaren, sowie verborgenen Variablen (und Parametern) zusammensetzen. Für gewöhnlich werden Bayessche Variationsmethoden für folgende zwei Ziele eingesetzt:

- Man sucht eine analytische Abschätzung der A-posteriori-Wahrscheinlichkeit der verborgenen Variablen, gegeben der wahrnehmbaren Variablen, um statistische Schlussfolgerungen ziehen zu können.
- Man sucht eine untere Schranke für die marginalisierte Wahrscheinlichkeit der wahrgenommenen Variablen (Daten), um bessere Modelle zu finden. Die Idee ist, dass die Daten besser zu einem gegebenen Modell passen, wenn die zugehörige marginalisierte Wahrscheinlichkeit höher ist als bei anderen Modellen.

Wir werden zur Herleitung des Verfahrens, welche sich nach [4, Kapitel 10] richtet, eine allgemeine mathematische Notation verwenden und dieses erst danach auf unser Aufmerksamkeitsmodell übertragen.
Wir behandeln den diskreten Fall und bezeichnen zunächst die Menge aller verborgenen Variablen und Parameter mit Z und die Menge aller wahrgenommenen Variablen beziehungsweise Daten mit X. Das stochastische Modell spezifiziert eine gemeinsame Verteilung $p(X, Z)$ und das Ziel ist es, die A-posteriori-Verteilung $p(Z \mid X)$ der verborgenen Variablen und die *Evidenz* $p(X)$ des Modells abzuschätzen. Die Idee ist, die A-posteriori-Verteilung mit einer Variationsverteilung $q(Z)$ zu approximieren: $p(Z \mid X) \approx q(Z)$, wobei die Verteilung $q(Z)$ auf eine Familie von Verteilungen eingeschränkt wird, die eine einfachere Form als $p(Z \mid X)$ haben. Um eine gute Annäherung an die gewünschte Verteilung zu erhalten, misst man die Unterschiedlichkeit dieser beiden Verteilungen für gewöhnlich mit der sogenannten *Kullback-Leibler-Divergenz*.

Definition 4.4 (Kullback-Leibler-Divergenz)**.** Die *Kullback-Leibler-Divergenz* $D_{KL}(\cdot||\cdot)$ (KL-Divergenz) bezeichnet ein Maß für die Unterschiedlichkeit zweier Wahrscheinlichkeitsverteilungen q und p. Im Diskreten ist sie definiert als

$$D_{KL}(q||p) = \sum_{Z} q(Z) \log \frac{q(Z)}{p(Z)}. \tag{4.14}$$

Wir gehen nicht weiter darauf ein, warum sich diese Wahl zur Messung der Ähnlichkeit zweier Verteilungen anbietet, halten aber fest, dass stets $D_{KL}(q||p) \geq 0$ gilt und dass $D_{KL}(q||p) = 0$ genau dann gilt, wenn $q(Z) = p(Z)$. In unserem Fall kann die KL-Divergenz umgeschrieben werden zu

$$D_{KL}(q||p) = \sum_Z q(Z) \log \frac{q(Z)}{p(Z \mid X)} \tag{4.15}$$

$$= \sum_Z q(Z) \log q(Z) - \sum_Z q(Z) \log p(Z \mid X) \tag{4.16}$$

$$= \sum_Z q(Z) \log q(Z) - \sum_Z q(Z) \log \frac{p(X,Z)}{p(X)} \tag{4.17}$$

$$= \sum_Z q(Z) \log q(Z) - \sum_Z q(Z) \log p(X,Z) + \sum_Z q(Z) \log p(X) \tag{4.18}$$

$$= \sum_Z q(Z) \log \frac{q(Z)}{p(X,Z)} + \log p(X). \tag{4.19}$$

Dabei haben wir einerseits den Fakt genutzt, dass $\sum_Z q(Z) = 1$ gilt, da q eine Wahrscheinlichkeitsverteilung ist und andererseits haben wir den Multiplikationssatz für Wahrscheinlichkeiten benutzt: $p(X,Z) = p(Z \mid X)p(X)$. Die Gleichung (4.19) können wir nun nach $\log p(X)$ umstellen und erhalten

$$\log p(X) = D_{KL}(q||p) - \sum_Z q(Z) \log \frac{q(Z)}{p(X,Z)} = D_{KL}(q||p) + \mathcal{L}(q), \tag{4.20}$$

wobei $\mathcal{L}(q) = \sum_Z q(Z) \log \frac{p(X,Z)}{q(Z)}$ als *freie Variationsenergie* bezeichnet wird, da sie als *Energie* $\mathbb{E}_q[\log p(X,Z)]$ plus Entropie $H[q]$ von q dargestellt werden kann:

$$\mathcal{L}(q) = \sum_Z q(Z) \log \frac{p(X,Z)}{q(Z)} \tag{4.21}$$

$$= \sum_Z q(Z) \log p(X,Z) - \sum_Z q(Z) \log q(Z) \tag{4.22}$$

$$= \mathbb{E}_q[\log p(X,Z)] + H[q]. \tag{4.23}$$

In (4.20) ist $p(X)$ im Bezug auf q fixiert, das heißt wenn man $\mathcal{L}(q)$ maximiert, wird $D_{KL}(q||p)$ minimiert und somit erhält man einerseits eine analytische Abschätzung q der A-posteriori-Verteilung $p(Z \mid X)$ und andererseits eine untere Schranke $\log p(X) \geq \mathcal{L}(q)$ der Evidenz, da $D_{KL}(q||p) \geq 0$ gilt. Für uns von Interesse ist vor allem die untere Schranke $\mathcal{L}(q)$, da eine Optimierung dieser gleichzeitig die Evidenz erhöht. Im Folgenden nutzen wir die freie Variationsenergie, um ein Lernverfahren für das Aufmerksamkeitsmodell abzuleiten.
Wir betrachten ein Klassifikationsproblem für Eingabebilder I. Wir formulieren das Training wie zuvor als überwachtes Lernen. Das Modell implementiert die Kreuzentropie als Zielfunktion und versucht, die zum Bild gehörige Klasse t vorherzusagen. Die Ausgaben des Netzes sind durch die Positionen l von jedem Glimpse bedingt. Diese entsprechen den verborgenen Variablen, da sie aus der vom Modell parametrisierten Verteilung gezogen werden und somit nicht direkt wahrnehmbar sind. Um die Wahrscheinlichkeit der korrekten Klassifikation $p(t \mid \theta)$ zu erhöhen, können wir über die

Glimpse-Position marginalisieren (und logarithmieren):

$$\log p(t \mid \theta) = \log \sum_l p(t \mid l, \theta) p(l \mid \theta). \tag{4.24}$$

Wir können die marginalisierte Zielfunktion von unten mit der freien Variationsenergie $\mathcal{L}$ abschätzen und den Multiplikationssatz für Wahrscheinlichkeiten anwenden:

$$\log p(t \mid \theta) \geq \mathcal{L} \tag{4.25}$$

$$\begin{aligned} \log \sum_l p(t \mid l, \theta) p(l \mid \theta) &\geq \sum_l p(l \mid \theta) \log \frac{p(t, l \mid \theta)}{p(l \mid \theta)} \\ &= \sum_l p(l \mid \theta) \log p(t \mid l, \theta). \end{aligned} \tag{4.26}$$

Die Zielfunktion kann nun trainiert werden, indem wir diese freie Energie bezüglich den Modellparametern θ optimieren und ein Gradientenverfahren durchführen. Unter Verwendung des Likelihood-Ratio-Tricks erhält man für den Gradienten $\nabla_\theta \mathcal{L}$ der freien Energie:

$$\begin{aligned} \nabla_\theta \mathcal{L} &= \sum_l p(l \mid \theta) \nabla_\theta \log p(t \mid l, \theta) + \sum_l \log p(t \mid l, \theta) \nabla_\theta p(l \mid \theta) \\ &= \sum_l p(l \mid \theta) [\nabla_\theta \log p(t \mid l, \theta) + \log p(t \mid l, \theta) \nabla_\theta \log p(l \mid \theta)]. \end{aligned} \tag{4.27}$$

Die Summation in (4.27) kann man dann mit einer Monte-Carlo-Simulation approximieren, indem man K Beispielpositionen l^k zieht und das arithmetische Mittel bildet:

$$l^k \sim p(l \mid \theta) \tag{4.28}$$

$$\nabla_\theta \mathcal{L} \approx \frac{1}{K} \sum_{k=1}^{K} [\nabla_\theta \log p(t \mid l^k, \theta) + \log p(t \mid l^k, \theta) \nabla_\theta \log p(l^k \mid \theta)]. \tag{4.29}$$

Die Gleichung (4.29) liefert uns eine in der Praxis anwendbare Abschätzung des gewünschten Gradienten. Während des Trainings kann man nach jedem Glimpse die Beispielpositionen ziehen und den Gradienten berechnen. Damit kann ein Gradientenanstieg durchgeführt werden, um unser Aufmerksamkeitsmodell zu trainieren, indem wir die Fehler standardmäßig durch die Netze zurück propagieren. Ein Problem stellt der unbeschränkte Term $\log p(t \mid l^k, \theta)$ dar, da er eine hohe Varianz in die Gradientenschätzung integriert. Ist die Beispielposition l^k zufällig weit vom gesuchten Objekt entfernt, wird die Wahrscheinlichkeit $p(t \mid l^k, \theta)$ der korrekten Klassifikation vermutlich gering sein und somit nach Logarithmierung einen unerwünscht großen Gradienten durchs Modell propagieren (vergleiche [1]). Um die Varianz zu reduzieren, kann man diesen Term durch eine diskrete Indikatorfunktion R ersetzen, welche nur die Werte 0 und 1 annimmt:

$$R = \begin{cases} 1 & t = \arg\max_{\hat{t}} \log p(\hat{t} \mid l^k, \theta) \\ 0 & \text{sonst} \end{cases} \tag{4.30}$$

und genau dann 1 ist, wenn die korrekte Klasse als wahrscheinlichste ausgegeben wurde. Ebenso kann man wie bei REINFORCE eine Baseline b integrieren, um die Varianz weiter zu reduzieren. Mit diesen beiden Anpassungen erhält man die folgende Gradientenschätzung:

$$\nabla_\theta \mathcal{L} \approx \frac{1}{K} \sum_{k=1}^{K} [\nabla_\theta \log p(t \mid l^k, \theta) + \lambda(R - b) \nabla_\theta \log p(l^k \mid \theta)], \tag{4.31}$$

wobei der Hyperparameter λ die Skalierung der beiden Komponenten kontrolliert. Betrachtet man diese Anpassung im RL-Framework, so entspricht der zweite Term in (4.31) einer erwartungstreuen Schätzung des Gradienten $\nabla_\theta G$ des erwarteten Gewinns G. Der erste Term hingegen entspricht dem Anteil des Aktionsnetzwerks im hybriden überwachten Lernen. Dieses Verfahren kann also mit der Optimierung der freien Variationsenergie motiviert werden.

5 Experimente

In diesem Abschnitt wollen wir auf die einzelnen Problemstellungen eingehen, an denen wir das Aufmerksamkeitsmodell getestet haben. Zunächst beschreiben wir einige einheitliche Konfigurationen bestimmter Hyperparameter. Parameter, die direkten Einfluss auf die Architektur des Modells haben, wie beispielsweise die Anzahl verwendeter Neuronen oder die Wahl der Aktivierungsfunktionen, wurden bereits in Kapitel 4.1 unter den jeweiligen Komponenten angegeben.
Alle Methoden wurden mit Mini-Batch-Training und einer Batch-Größe von 20 Trainingspaaren trainiert. Die Eingabedaten wurden nicht gemäß Kapitel 2.5.3 normalisiert, unterliefen aber abhängig von der Problemstellung einer anderweitigen Vorverarbeitung, bevor sie zum Training oder Testen eingesetzt wurden. Zur Initialisierung der Gewichte wurde eine Gleichverteilung mit Mittelwert 0 und einer Standardabweichung von 0.05 gewählt. Der Mittelwert war stets fixiert. Mit der Standardabweichung wurde experimentiert und 0.05 hat sich als ein guter Kompromiss von nicht zu kleinen oder zu großen Gewichten herausgestellt. Die Performance des Aufmerksamkeitsmodells wurde während des Trainings stets an der Testmenge überprüft. Eine Validierungsmenge und frühes Stoppen wurden nicht implementiert. Stattdessen haben wir eine feste Anzahl an Epochen vorgegeben. Als Obergrenze wurden 250 Epochen festgelegt, wobei die Performance meist schon vorher stagnierte. Das Training eines Aufmerksamkeitsmodells hat mehrere Tage in Anspruch genommen, obwohl parallel auf mehreren Kernen gearbeitet wurde. Dies ist vor allem der insgesamt hohen Parameteranzahl der verschiedenen Netze und dem zeitaufwendigen Laden der Trainingsdaten geschuldet. Für die Lernrate haben wir ein globales, exponentiell abnehmendes Modell gewählt, da dies in der Praxis am schnellsten die besten Ergebnisse lieferte. Für die erste Epoche wurde $\eta = 0.01$ gewählt und der Wert wurde in jeder weiteren Epoche mit 0.975 multipliziert, sodass in der n-ten Epoche $\eta = 0.975^n * 0.01$ gilt. Weiterhin wurde mit einer L_2-Regularisierung der Gewichte und unterschiedlichen Regularisierungsparametern experimentiert. In der Praxis konnten wir allerdings keine verbesserte Performance des Aufmerksamkeitsmodells feststellen, sodass die Regularisierungen später entfernt wurden. Um potentiell bessere Minima im Fehlergebirge zu finden, wurde dem Training ein Momentum mit statischem Momentumsparameter $\alpha = 0.9$ hinzugefügt.
Das Aufmerksamkeitsmodell machte seine Klassifizierungs- beziehungsweise Lokalisierungsentscheidung nur im letzten Zeitschritt $t = T$. Die Belohnung r_T des Agenten im letzten Zeitschritt hängt von der Problemstellung ab. Alle vorigen Belohnungen r_t $(t < T)$ waren 0. Zur Verringerung der Varianz in der Gradientenschätzung wurde eine Baseline b integriert. Da das Baseline-Netzwerk keine sichtliche Verbesserung gegenüber der exponentiellen Glättung erzielte, wurde meist letztere verwendet, um die Baseline zu bestimmen, da diese Herangehensweise billiger in der Berechnung ist. Die besten Ergebnisse lieferte der Baselineparameter $\gamma = 0.2$. Der Anteil des bestärkenden Ler-

nens am Gradienten durch das Positionsnetzwerk (λ in (4.31)) wurde abhängig von der Problemstellung entweder gar nicht skaliert oder verringert. In der Praxis hat sich auch herausgestellt, dass man nicht pauschal sagen kann, ob es besser ist, den Gradienten des Positionsnetzwerk nur zum Training desselben zu benutzen oder ob der Gradient durch alle restlichen Komponenten des Modells propagiert werden soll. Wir konnten mit beiden Varianten gute Ergebnisse erzielen. Für die zweite war aber eine Skalierung des Gradienten erforderlich, da die Performance sonst zu stark oszillierte. Der Gradient des Aktionsnetzwerks wurde immer durch alle anderen Netze propagiert. Eine andere Idee war es, den Gradienten des Glimpse-Netzwerks eines Zeitschritts durch das Positionsnetzwerk im vorhergehenden Zeitschritt zu propagieren, da die Positionen in der Vorwärtsphase aus der parametrisierten Verteilung gezogen wurden. Damit wurde die Performance allerdings verschlechtert und die Idee somit verworfen. Es bestand die Möglichkeit, das stochastische Verhalten des Positionsnetzwerks abzuwandeln, indem man die Ausgaben nicht als Mittelwert der Verteilung interpretiert, sondern direkt als Vorhersage für die nächste zu fokussierende Position wählt. Wir haben das stochastische Verhalten bevorzugt, da somit die Erkundung des Bildes begünstigt wird (siehe auch Kapitel 3.1.3). Mit der Varianz der Positionsverteilung konnte man kontrollieren, inwieweit sich das Modell an die Vorhersage halten sollte. Bei uns erwies sich eine Varianz von 0.01 als guter Kompromiss zwischen Erkundung und Ausnutzung.

5.1 Bildklassifizierungen

Als erstes wollten wir testen, ob das Aufmerksamkeitsmodell mit unseren Trainingsmethoden in der Lage ist, erfolgreiche Glimpse-Strategien zu lernen und gleichzeitig Informationen über das Bild in seinem internen Zustand zu speichern. Dazu sollte es Ziffern klassifizieren, indem es auf der MNIST-Datenmenge (Mixed National Institute of Standards and Technology database) trainiert wurde. Die MNIST-Datenmenge enthält eine große Menge an handgeschriebenen Ziffern und wird häufig zum Training von Bildverarbeitungssystemen und im maschinellen Lernen eingesetzt. Sie umfasst 60000 Trainingspaare und 10000 Testpaare. Erzeugt wurde die MNIST-Datenmenge, indem Elemente aus den Trainings- und Testmengen der größeren NIST-Datenmenge miteinander gemischt wurden. Die Schwarz-Weiß-Bilder aus NIST wurden normalisiert und auf eine Größe von 20×20 Pixel skaliert. Die entstandenen Graustufenbilder wurden dann in 28×28 Ausschnitten derart eingebettet, dass die Ziffern zentriert sind. In der Abbildung 5.1 sind einige Beispielbilder der MNIST-Datenmenge zu sehen.

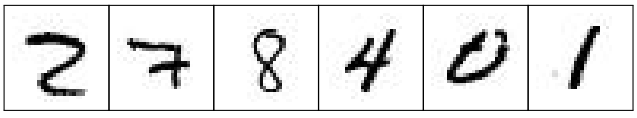


Abbildung 5.1: Beispielbilder der MNIST-Datenmenge.

Der Glimpse-Sensor wurde auf einen einzelnen Bildausschnitt der Größe 8×8 beschränkt, sodass der Agent immer nur einen kleinen Teil des eigentlichen Bildes sah

und gezwungen war, sich durch intelligente Wahl verschiedener Glimpse-Positionen eine interne Repräsentation des Bildes und der Ziffer zusammenzusetzen. In Anlehnung an die Ergebnisse von [24] haben wir uns auf 6 Glimpses pro Episode festgelegt. Mehr Glimpses erreichten keine definitive Verbesserung und vereinzelte Tests mit weniger Glimpses verliefen zwar schneller, konnten aber nie gleichwertige Ergebnisse liefern. Zur Approximation des Gradienten während des Trainings wurde ein einzelner Beispielverlauf benutzt ($K = 1$ in (4.11) beziehungsweise (4.31)), um die Laufzeit zu verringern. Klassifizierte der Agent das Bild im letzten Zeitschritt korrekt, erhielt er die Belohnung 1. Klassifizierte er falsch, war die Belohnung 0. Der Anteil des bestärkenden Lernens am Gradienten wurde nicht skaliert, sodass $\lambda = 1$ in (4.31) galt.
Wir erreichten mit dieser Konfiguration eine Klassifizierungsrate von 98.62 %. Mit der zugehörigen Fehlerrate von 1.38 % konnten wir somit fast die besten Modelle aus [24] bestätigen, welche eine Fehlerrate von 1.29 % aufwiesen. Die kleine Diskrepanz ist höchstwahrscheinlich auf ein Feintuning der vielen Hyperparameter und möglichen Optimierungen der Trainingsmethoden zurückzuführen. Unsere Ergebnisse beweisen, dass das hier implementierte Aufmerksamkeitsmodell in der Lage ist, aus unabhängigen Glimpses extrahierte Informationen miteinander zu kombinieren und gleichzeitig markante Merkmale für die Bildklassifizierung zu lernen. Einige Beispiele der gelernten Glimpse-Strategie sind in Abbildung 5.2 zu sehen. Es ist erkenntlich, dass das Modell leere Bereiche meidet und das Gebiet um die gesuchte Ziffer erkundet.

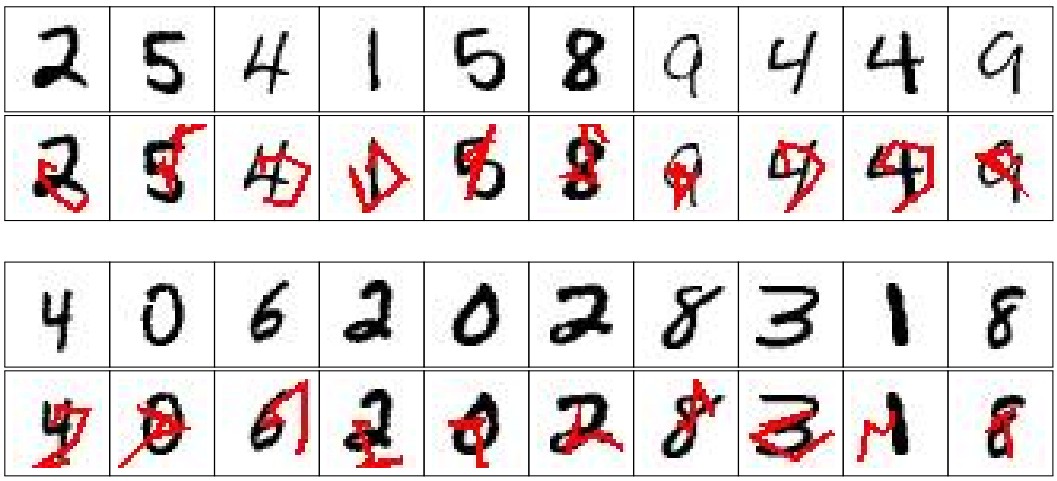


Abbildung 5.2: Beispiele der gelernten Glimpse-Strategie an der MNIST-Datenmenge. Zeile 1 und 3 enthalten zufällig ausgewählte Eingabebilder der MNIST-Datenmenge. Zeile 2 und 4 visualisieren die vom Aufmerksamkeitsmodell zugehörige Glimpse-Strategie mit 6 Schritten anhand des roten Pfades.

Als nächster Schritt war ursprünglich geplant, zu überprüfen, ob unser Aufmerksamkeitsmodell auch Klassifizierungen auf größeren Bildern durchführen kann. Weiterhin sollte es eine Art Translationsinvarianz implementieren, wenn die vorhandenen Objekte nicht zentriert waren. Für dieses Problem konnten wir Daten zur Laufzeit generieren, indem wir ein Bild der MNIST-Datenmenge nahmen und dieses zufällig in einem grö-

ßeren leeren Bild platzierten. Aufgrund der unerwarteten Zeitkomplexität der vielen Trainingsdurchläufe konnten wir diesen Abschnitt der Experimente leider nicht weiter ausbauen, um uns dem Problem der Objektlokalisierung in reellen Szenarien widmen zu können.

5.2 Objektlokalisierungen

Mit den Experimenten an der MNIST-Datenmenge wurde verifiziert, dass unser Modell in der Lage ist, aufmerksamkeitsbasierte Strategien zu lernen und Informationen aus mehreren Zeitschritten miteinander zu kombinieren. Als einen alternativen Anwendungsbereich wollten wir testen, wie das Aufmerksamkeitsmodell mit Objektlokalisierungen in reellen Szenarien umgeht. Dies schien ein natürlicher Schritt zu sein, da Experimente an künstlich erstellten Datenmengen nur eine eingeschränkte Auskunft über die Anwendbarkeit eines Systems in der realen Welt liefern. Um den Anforderungen so einer Problemstellung gerecht zu werden, muss das Modell diverse Probleme bewältigen.
Einerseits können die Bilder wesentlich größer sein als künstlich kreierte 28×28 Bilder und außerdem können die Bilder ein weites Spektrum an unerwünschten Stördaten enthalten, welche es erschweren, die relevanten Informationen zu extrahieren. Insbesondere Systeme, die auf dem gesamten Bild gleichzeitig arbeiten, sind anfällig für solche Stördaten, da sie zu jedem Zeitpunkt mitverarbeitet werden. Das Aufmerksamkeitsmodell könnte den Vorteil haben, dass es den Glimpse-Sensor auf ausgewählte Bildausschnitte fokussiert und somit möglicherweise unwichtige Informationen direkt ausblendet. Im Idealfall lernt das Aufmerksamkeitsmodell, invariant gegenüber all den nebensächlichen Objekten und Informationen zu werden.
Ausgangspunkt für das Problem der Objektlokalisierung bildete ein Datensatz von 33000 Blitzerfotos, für die die eingrenzenden Koordinaten des zugehörigen Nummernschilds bekannt waren. Eingeteilt wurden die Daten in eine Trainingsmenge mit 30000 Elementen und eine Testmenge mit den restlichen 3000 Elementen. Diese Fotos umfassen diverse variable Komponenten, die eine Objektlokalisierung verkomplizieren. Durch die relativ großen Dimensionen der Bilder sind neben dem jeweiligen Fahrzeug viele irrelevante Nebeninformationen enthalten: Im Hintergrund sind verschiedenste Landschaften zu sehen. Es befinden sich Bäume, Gebäude und Straßenschilder am Straßenrand. Die Straße selbst weist Unregelmäßigkeiten und Fahrbahnmarkieren auf. Bei der Auswertung von Blitzerfotos ist man aber nur am Nummernschild interessiert. Dieses kann sich theoretisch an jeder beliebigen Position im Bild befinden und zusätzlich in seiner Orientierung variieren. Außerdem kann sich das Nummernschild in unterschiedlicher Entfernung zum Blitzer befinden, sodass sich die Maße relativ zur Bildgröße verändern. Einen weiteren wichtigen Faktor stellen die Beleuchtungsverhältnisse zum Zeitpunkt der Aufnahme dar. Sowohl die Tageszeit und Wetterverhältnisse als auch der Blitz selbst beeinflussen das endgültige Bildergebnis. Aus Datenschutzgründen können wir den genutzten Datensatz nicht illustrieren. Stattdessen wurden uns von der Planet artificial intelligence GmbH einige Aufnahmen von der Einfahrt ihrer firmeneigenen Tiefgarage zur Verfügung gestellt (Abbildung 5.3).

Abbildung 5.3: Beispielaufnahmen von PKWs mit KFZ-Kennzeichen an der Einfahrt der firmeneigenen Tiefgarage von Planet.

Um die Daten zu vereinheitlichen, erzeugten wir die eigentlichen Trainingsbilder zur Laufzeit. Die originalen Graustufenbilder wurden zunächst normalisiert, indem der Mittelwert der Pixeldaten auf 0 verschoben wurde und die Varianz auf 1 skaliert wurde. Aus einem normalisierten Bild der Größe 1280×1096 wurde dann eine 300×100 Region zufällig ausgeschnitten, unter der Bedingung, dass das Nummernschild im neuen Bildausschnitt enthalten ist. Diese Trainingsbilder wurden dem Aufmerksamkeitsmodell dann als Eingaben präsentiert. Ausgeben sollte es den Mittelpunkt $(x, y) \in [-1, 1]^2$ des Nummernschilds, wobei der Punkt $(-1, -1)$ der oberen linken Ecke des Bildes entsprach und der Punkt $(0, 0)$ die Mitte des Bildes angab.
Der Glimpse-Sensor machte eine initiale Aufnahme der Größe 15×15 Pixel. Die Anzahl der geringer auflösenden, aber größeren Folgeaufnahmen wurde variiert, wobei 5 Bildausschnitte gute Ergebnisse erzielten. Zur Approximation des Gradienten wurde wieder ein ($K = 1$) Beispielverlauf benutzt. Beim Lokalisierungsproblem entschieden wir uns für eine kontinuierliche Belohnung zwischen 0 und 1. Befand sich der vorhergesagte Mittelpunkt des Nummernschilds in einer inneren Region um den korrekten Mittelpunkt, deren Seitenmaße halb so groß waren wie die des Nummernschilds, dann erhielt der Agent die Belohnung 1. Lag die Vorhersage außerhalb des Nummernschilds, erhielt er die Belohnung 0. In dem Bereich zwischen der inneren Region und der äußeren Abgrenzung wurde die Belohnung von 1 auf 0 linear skaliert. In unseren Experimenten stellte sich heraus, dass eine Abschwächung des Gradienten des Positionsnetzwerks zu einem stabileren Training führte ($\lambda < 1$ in (4.31)). Als optimal erwiesen sich Werte in der Größenordnung 10^{-2}.
Als Regressionsaufgabe optimiert man den mittleren quadratischen Fehler zwischen dem korrekten Mittelpunkt des Nummernschilds und der Vorhersage des Modells. Die Koordinaten sind im Bild auf $[-1, 1]^2$ skaliert. Die besten Modelle lieferten MSEs zwischen 0.056 und 0.059 über der Testmenge. Zur visuellen Vorstellung: Das entspricht beispielsweise einer Abweichung von rund $35,4$ Pixeln in reiner x-Richtung, einer Abweichung von rund $11,8$ Pixeln in reiner y-Richtung (da unsere Bilder 300×100 Pixel groß sind, entspricht eine Einheit dreimal so vielen Pixeln in x-Richtung wie in y-Richtung) oder gemischten Werten dazwischen. Zur Visualisierung der Glimpse-Strategie nutzen wir die Beispielaufnahmen von Planet. Diese wurden der gleichen Vorverarbeitung wie der Datensatz der Blitzerfotos unterzogen: Nach einer Normalisierung der Graustufenbilder wurden 300×100 Regionen ausgeschnitten, die das Nummernschild enthalten. Einige Beispiele der gelernten Glimpse-Strategie sind in Abbildung 5.4 zu sehen.

Abbildung 5.4: Beispiele der gelernten Glimpse-Strategie an den PKW-Aufnahmen von Planet. Spalte 1 enthält einige Beispielaufnahmen. Spalte 2 zeigt die Bilder nach der Vorverarbeitung und der Anschaulichkeit halber vergrößert. Spalte 3 visualisiert die Glimpse-Strategie mit 6 Schritten anhand des roten Pfades und die Lokalisierung des Nummernschildes anhand des grünen Quadrates.

Die Lokalisierung der Nummernschilder wird durch das Aufmerksamkeitsmodell gut bewältigt. Es ist allerdings auffällig, dass das Modell oft dazu neigt, den Glimpse-Sensor auf die Ränder des Bildes auszurichten und dort stecken bleibt. Die Ausgaben des Positionsnetzwerks scheinen also durch saturierte Neuronen bestimmt zu sein. Idealerweise sollte das Modell die Region um das Nummernschild erkunden, um die relevanten Informationen extrahieren zu können. Ein Ansatz für weitere Experimente wären hier Regularisierungen an den Gewichten oder dynamisch gesteuerte Lernraten, um den Gradienten besser kontrollieren zu können und somit die Saturierung der Neuronen einzuschränken.

6 Fazit und Ausblick

6.1 Fazit

Als etablierter Zweig der künstlichen Intelligenz bieten künstliche neuronale Netzwerke zahlreiche Anwendungsmöglichkeiten in der Bild- und Mustererkennung. Die aktuell erfolgreichsten Systeme sind Faltungsnetzwerke, welche trotz ihrer sehr guten Performance leider unter hohen Berechnungskosten leiden, gerade wenn es um die Verarbeitung hochauflösender Bilder geht.
Das Ziel der Masterarbeit war es, ein aufmerksamkeitsbasiertes System zur zielgerichteten visuellen Bildverarbeitung aufzubauen. Auf der Grundlage eines rekurrenten neuronalen Netzes werden verschiedene Komponenten zur Vorverarbeitung, Aufmerksamkeit und Auswertung kombiniert. Das Modell lernt, verschiedene Bildausschnitte sequentiell zu durchlaufen und die extrahierten Informationen zu kombinieren, um eine dynamische interne Repräsentation der Szene aufzubauen. Die Bildausschnitte werden von einem visuellen Sensor erzeugt und umfassen nur eine vorgegebene Größe. Die Aufnahme und Verarbeitung der Bildausschnitte verläuft somit unabhängig von der Größe der Eingabebilder und das Modell kontrolliert, welcher Punkt des Bildes als nächster fokussiert wird. Das Training wird durch eine Art hybrides überwachtes Lernen realisiert. Die Komponenten, die für die Aufmerksamkeit des Modells zuständig sind, führen nicht differenzierbare Anteile ein, sodass ein reines überwachtes Training nicht möglich ist. Um dennoch Lernimpulse generieren zu können, nutzt man Methoden aus dem bestärkenden Lernen. Durch skalare Belohnungen wird dem Modell indiziert, welche Strategien auf lange Sicht positive Ergebnisse erzielen und es lernt, solche zu bevorzugen.
Die Experimente wurden in zwei Abschnitte gegliedert. Im ersten Teil sollte die grundlegende Funktionalität des Aufmerksamkeitsmodells sichergestellt werden. Es muss in der Lage sein, aufmerksamkeitsbasierte Strategien zu lernen, sodass es Bildausschnitte fokussiert, die für die Problemstellung relevante Informationen enthalten. Weiterhin muss es die extrahierten Informationen aus verschiedenen Zeitschritten miteinander kombinieren und in einer Art internem Gedächtnis speichern, sodass es bessere zukünftige Entscheidungen treffen kann. Die Erfüllung beider Anforderungen konnten wir mit unseren Experimenten zur Klassifikation von Ziffern der MNIST-Datenmenge nachweisen.
Im zweiten Teil wollten wir die Komplexität der Aufgabenstellung erhöhen. Wir entschieden uns für ein praxisrelevantes Lokalisierungsproblem, bei dem das Nummernschild in Blitzerfotos erfasst werden sollte. Das Modell muss hier mit wesentlich größeren Bildern arbeiten. Zusätzlich enthalten diese eine Menge an irrelevanten Nebeninformationen und die Bildqualität kann aufgrund äußerer Einflüsse erheblich variieren. Mit einigen Anpassungen am Modell, unter anderem der Einführung einer kontinuierlichen Belohnungsfunktion, der Kombination mehrerer Bildausschnitte pro Glimpse und einer

Skalierung des Anteils des bestärkenden Lernens an der Gradientenschätzung, konnten wir auch hier gute Ergebnisse erzielen.
Zusammenfassend haben wir gezeigt, dass das Aufmerksamkeitsmodell in der Lage ist, Objekte in reellen Szenarien zu erkennen und zu lokalisieren. Dabei kombiniert es die Stabilität und Vorteile des überwachten Lernens mit der Flexibilität des bestärkenden Lernens, um ein globales Training eines nicht differenzierbaren Modells zu ermöglichen. Gegenüber herkömmlichen Systemen hat es den Vorteil, unabhängig von der Größe der Eingabebilder zu arbeiten, da einzig und allein die Spezifikation des visuellen Sensors die zu verarbeitenden Bilddaten bestimmt. Beim Training fiel auf, dass das Modell kaum unter einer Überanpassung leidet. Selbst bei sehr langen Trainingszyklen verschlechterte sich die Performance auf der Testmenge kaum, sondern schwankte eher um ein Minimum. Dieser Effekt ist womöglich durch das stochastische Verhalten der Glimpse-Strategie zu begründen. Außerdem wird durch das aufmerksamkeitsbasierte Verhalten des Modells automatisch eine Translationsinvarianz implementiert.

6.2 Ausblick

Aufgrund des zeitlichen Rahmens dieser Arbeit konnten einige Punkte und mögliche Erweiterungen nicht weiter erforscht werden. Die modulare Architektur und Flexibilität der Trainingsmethoden eröffnen Möglichkeiten, das Aufmerksamkeitsmodell weiter auszubauen.
In beiden Experimenten musste das Modell nur je ein Objekt klassifizieren beziehungsweise lokalisieren. Es sind weitere Experimente notwendig, um unser Aufmerksamkeitsmodell auf Problemstellungen mit mehreren Objekten zu erweitern. Vom Aufbau her sollte es möglich sein, die verschiedenen Objekte in eine Sequenz einzuordnen, sodass das Modell jedes einzelne sequentiell abarbeitet. In [1] wurden bereits ähnliche Experimente mit guten Ergebnissen durchgeführt.
Die Tests der Parameteroptimierungen wurden insgesamt in einem relativ kleinen Rahmen vorgenommen. Einige strukturelle Konstellationen haben wir direkt aus [24] und [1] übernommen, da diese sich bereits als gute Ausgangspunkte bewährt hatten. Hyperparameter wie die Lernrate, Mini-Batch-Größe und diverse weitere Parameter wurden experimentell bestimmt. Allerdings war es uns aufgrund der zeitintensiven Trainingsvorgänge nicht möglich, perfekte Feinjustierungen vorzunehmen, sondern wir versuchten eher die optimalen Größenordnungen zu bestimmen. Vereinzelt wurden auch intuitive Werte verwendet, wie zum Beispiel bei der Größe der initialen Glimpse-Aufnahme und der Anzahl der zugehörigen Bildausschnitte.
Ein mögliches Anwendungsgebiet, das in dieser Arbeit gar nicht betrachtet wurde, bilden dynamische Umgebungen. Das Aufmerksamkeitsmodell könnte auf Videos oder interaktive Probleme angewandt werden. In diesen Fällen hätten die Aktionen mitunter Auswirkungen auf den Zustand der Umgebung, wie beim Spielen eines Videospiels. Weiterhin würde der Agent Belohnungen in jedem Zeitschritt erhalten, anstatt nur eine Belohnung im finalen Schritt bei statischen Problemen. Für solche Aufgaben und Anwendungen in der Praxis sind extensive Experimente nötig.
Ausgehend von unser aktuellen Implementierung des Aufmerksamkeitsmodells ergeben

sich auch diverse Erweiterungsmöglichkeiten oder alternative Ansätze in der Architektur. Beispielsweise kann man die Ausgabe des Positionsnetzwerks als deterministische Vorhersage interpretieren, sodass das stochastische Verhalten verloren geht. In diesem Fall würde sich das Modell wie ein normales FNN verhalten. Das Aktionsnetzwerk ließe sich durch Zusatzfunktionen erweitern. Beispielsweise könnte man eine Aktion hinzufügen, die dem Modell erlaubt, zu jedem beliebigen Zeitpunkt zu stoppen, um seine finale Ausgabe abzuliefern. Das Modell könnte somit stoppen, sobald es ausreichend eindeutige Informationen gesammelt hat, um eine endgültige Entscheidung zu treffen. Damit würde man Zeit- und Berechnungskosten sparen. Mit einer anderen Aktion könnte man dem Modell erlauben, die Skalierungen, mit denen der Glimpse-Sensor Aufnahmen macht, selbst einzustellen, um sich somit besser an die Größen der gesuchten Objekte anzupassen.
Einen relativ großen Ausbaupunkt stellt der interne Zustand des Aufmerksamkeitsmodells dar. Zurzeit ist dieser durch die einfachen rekurrenten Verbindungen des Kernnetzwerks realisiert. Alle Informationen, die der Agent kombiniert, werden in den verborgenen Einheiten gespeichert. Aufgrund des Problems des verschwindenen Gradienten (siehe Kapitel 2.3.2) eignen sich normale RNNs in der Praxis allerdings nicht für langfristige Abhängigkeiten. Dieser Nachteil spielte bei unseren Experimenten keine schwerwiegende Rolle, da maximal eine Sequenz von 6 Glimpses erzeugt wurde. Für komplexere Problemstellungen bietet es sich an, speziell konstruierte rekurrente Architekturen zu integrieren. Weit verbreitet sind die *Netze mit langem Kurzzeitgedächtnis* (Long Short-Term Memory, LSTM, [12]), welche in der Lage sind, Informationen beliebig lange zu speichern. LSTM-Netzwerke bestehen aus mehreren *Speicherzellen*, welche verschiedene Aktivierungen speichern. Der Zugriff auf sie wird durch spezielle Einheiten gesteuert, die lernen, sich je nach Kontext zu öffnen oder zu schließen und somit kontrollieren, welche Informationen in welchem Maße verarbeitet werden. Aus jüngerer Entwicklung stammen die *Netze mit gesteuerten rekurrenten Einheiten* (Gated Recurrent Unit, GRU, [5]). Diese sind etwas simpler als LSTMs aufgebaut, bestehen aber auch aus Speicherzellen und Einheiten, die den Informationsfluss kontrollieren. Weitere Varianten zum Lösen des Problems langfristiger Abhängigkeiten in neuronalen Netzwerken sind *tiefen-gesteuerte rekurrente neuronale Netze* (Depth-gated Recurrent Neural Networks, [37]) und sogenannte *rekurrente neuronale Uhrwerk-Netzwerke* (Clockwork Rekurrent Neural Networks, [17]).

Literaturverzeichnis

[1] **J. L. Ba, V. Mnih und K. Kavukcuoglu:** *Multiple object recognition with visual attention.* ICLR (2015).

[2] **J. Baxter und P. Bartlett:** *Infinite-horizon policy-gradient estimation.* JAIR 15 (2001), 319–350.

[3] **C. M. Bishop:** *Neural networks for pattern recognition.* Oxford: Clarendon Press (1995).

[4] **C. M. Bishop:** *Pattern recognition and machine learning.* Springer (2006).

[5] **K. Cho, B. van Merriënboer, C. Gulcehre, F. Bougares, H. Schwenk und Y. Bengio:** *Learning phrase representations using RNN encoder-decoder for statistical machine translation.* EMNLP (2014).

[6] **R. Duda, P. Hart und D. Stork:** *Pattern classification.* Wiley-Interscience Publication, New York (2001).

[7] **A. Gibiansky:** *Convolutional neural networks.* Erhältlich auf `http://andrew.gibiansky.com/archive.html` (2014).

[8] **P. Glynn:** *Likelihood ratio gradient estimation: An overview.* Proceedings of the winter simulation conference (1987), 366–375.

[9] **I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud und V. Shet:** *Multi-digit number recognition from street view imagery using deep convolutinal neural networks.* ICLR (2014).

[10] **A. Graves:** *Supervised sequence labelling with recurrent neural networks.* Heidelberg: Springer (2012).

[11] **E. Greensmith, P. L. Bartlett und J. Baxter:** *Variance reduction techniques for gradient estimates in reinforcement learning.* Journal of Machine Learning Research 5 (2004), 1471–1530.

[12] **S. Hochreiter und J. Schmidhuber:** *Long short-term memory.* Neural Computation 9(8) (1997), 1735-1780.

[13] **K. Hornik, M. Stinchcombe und H. White:** *Multilayer feedforward networks are universal approximators.* Neural Networks 2(5) (1989), 359-366.

[14] **M. Jaderberg, K. Simonyan, A. Vedaldi und A. Zisserman:** *Synthetic data and artificial neural networks for natural scene text recognition.* NIPS workshop on deep learning (2014).

[15] **K. Jarrett, K. Kavukcuogl, M. Ranzato und Y. LeCun.** *What is the best multi-stage architecture for object recognition?* ICCV (2009), 2146–2153.

[16] **L. P. Kaelbling, M. L. Littman und A. W. Moore:** *Reinforcement learning: A survey.* Journal of Artificial Intelligence Research 4 (1996), 237-285.

[17] **J. Koutník, K. Greff, F. Gomez und J. Schmidhuber:** *A clockwork RNN.* ICML (2014), 1845-1853.

[18] **D. Kriesel:** *Ein kleiner Überblick über Neuronale Netze.* Erhältlich auf `http://www.dkriesel.com` (2007).

[19] **A. Krizhevsky, I. Sutskever und G. E. Hinton:** *ImageNet classification with deep convolutional neural networks.* NIPS (2012), 1106–1114.

[20] **H. Larochelle und G. E. Hinton:** *Learning to combine foveal glimpses with a third-order boltzmann machine.* NIPS (2010).

[21] **Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard und L. D. Jackel:** *Backpropagation applied to handwritten zip code recognition.* Neural Computation 1(4) (1989), 541–551.

[22] **Y. LeCun, L. Bottou, G. Orr und K. Muller:** *Efficient BackProp.* In G. Orr and M. K., editors, Neural Networks: Tricks of the trade. Springer (1998).

[23] **P. Lewis:** *Multilayer perceptrons and backpropagation.* Erhältlich auf `http://www.cs.bham.ac.uk/internal/courses/intro-nc/current/` (2012).

[24] **V. Mnih, N. Heess, A. Graves und K. Kavukcuoglu:** *Recurrent models of visual attention.* NIPS (2014).

[25] **M. A. Nielsen:** *Neural networks and deep learning.* Determination Press (2015).

[26] **J. Peters und S. Schaal:** *Reinforcement learning of motor skills with policy gradients.* Neural Networks 21(4) (2008), 682–697.

[27] **R. A. Rensink:** *The dynamic representation of scenes.* Visual Cognition 7 (2000), 17-42.

[28] **P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus und Y. LeCun:** *Overfeat: Integrated recognition, localization and detection using convolutional networks.* ICLR (2014).

[29] **S.P. Singh, T. Jaakkola und M.I. Jordan:** *Learning without state-estimation in partially observable markovian decision processes.* International Conference on Machine Learning (1994), 284–292.

[30] **J. C. Spall:** *Introduction to stochastic search and optimization: Estimation, simulation, and control.* Hoboken, NJ: Wiley (2003).

[31] **R. S. Sutton und A. G. Barto:** *Reinforcement learning: An introduction.* Cambridge, MA: MIT Press (2012).

[32] **R. S. Sutton, D. McAllester, S. Singh und Y. Mansour:** *Policy gradient methods for reinforcement learning with function approximation.* Advances in Neural Information Processing Systems 12 (2000), 1057-1063.

[33] **L. Weaver und N. Tao:** *The optimal reward baseline for gradient-based reinforcement learning.* In Processings of The Seventeeth Conference on Uncertainty in Artificial Intelligence (2001), 538–545.

[34] **D. Wierstra, A. Foerster, J. Peters und J. Schmidhuber:** *Solving deep memory POMDPs with recurrent policy gradients.* ICANN (2007), 697-706.

[35] **R. J. Williams:** *Simple statistical gradient-following algorithms for connectionist reinforcement learning.* Machine Learning 8 (1992), 229-256.

[36] **M. Wooldridge:** *Intelligent agents: The key concepts.* In Multi-Agent-Systems and Applications, volume 2322 of LNCS, 3-43. Springer Verlag (2002).

[37] **K. Yao, T. Cohn, K. Vylomova, K. Duh und C. Dyer:** *Depth-gated recurrent neural networks.* arXiv preprint arXiv:1508.03790v2 (2015).

[38] **H. G. Zimmermann, R. Grothmann, A. M. Schaefer und C. Tietz:** *Identification and forecasting of large dynamical systems by dynamical consistent neural networks.* In S. Haykin, J. Principe, T. Sejnowski und J. McWhirter, editors, New Directions in Statistical Signal Processing: From Systems to Brain, 203-242. MIT Press (2006).

[39] `http://deeplearning.net/tutorial/lenet.html` (2016).

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt und ohne fremde Hilfe verfasst habe. Dazu habe ich keine außer den von mir angegebenen Hilfsmitteln und Quellen verwendet und die den benutzten Werken inhaltlich und wörtlich entnommenen Stellen habe ich als solche kenntlich gemacht.

Rostock, den 16.09.2016

_____

Johannes Michael