



hochschule mannheim

Klassifizierung von Bildern mithilfe von unüberwachtem Lernen mit Kohonen-Netzen

Jonathan Lieske

Bachelor-Thesis
zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)
Studiengang Informatik

Fakultät für Informatik
Hochschule Mannheim

31.01.2023

Betreuer
Prof. Dr. Jörn Fischer, Hochschule Mannheim
Prof. Dr. Thomas Ihme, Hochschule Mannheim

Lieske, Jonathan:

Klassifizierung von Bildern mithilfe von unüberwachtem Lernen mit Kohonen-Netzen /
Jonathan Lieske. –
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2023. 52 Seiten.

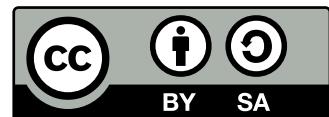
Lieske, Jonathan:

Classification of images using unsupervised learning with self-organizing maps / Jonathan
Lieske. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2023. 52 pages.

Erklärung

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

Dieses Werk ist lizenziert unter einer Creative Commons „Namensnennung – Weitergabe unter gleichen Bedingungen 4.0 International“ Lizenz.



Mannheim, 31.01.2023

A handwritten signature in blue ink that reads "Jonathan Lieske".

Jonathan Lieske

Abstract

Klassifizierung von Bildern mithilfe von unüberwachtem Lernen mit Kohonen-Netzen

Der Hippocampus des Gehirns von diversen Tieren, wie beispielsweise Ratten, beinhaltet sogenannte Place Cells, deren Aktivität Aufschluss über die Position des Lebewesens in seiner Umgebung gibt. Diese Arbeit beschäftigt sich mit der Simulation dieser Place Cells mit Methoden des maschinellen Lernens und stellt dabei ein System vor, welches die wahrgenommenen Bilder eines virtuellen Agenten in einem Labyrinth auf unüberwachte Weise klassifiziert, um so die wesentlichen Verhaltensweisen dieser Neuronen nachzustellen. Dabei werden sowohl ein vortrainiertes Convolution Neural Network zur Feature Extraction als auch eine Kohonen-Karte für die Klassifizierung verwendet.

Classification of images using unsupervised learning with self-organizing maps

The hippocampal region in the brain of several species, as for example rats, contain so called place cells, which have been shown to hold spatial information about the animal, such as its current position in the environment. This paper proposes a way of simulating these place cells using machine learning methods. The images perceived by a virtual agent in a maze are classified in an unsupervised manner in order to recreate the behavior of the neurons. The proposed architecture makes use of a Convolutional Neural Network for feature extraction and a Self Organizing Map for classification purposes.

Inhaltsverzeichnis

1 Einleitung	1
2 Grundlagen	3
2.1 Maschinelles Lernen	3
2.1.1 Überwachtes und unüberwachtes Lernen	4
2.2 Neuronale Netze	5
2.2.1 Das Neuron	5
2.2.2 Mehrschichtige neuronale Netze	7
2.3 Convolutional Neural Networks	9
2.3.1 Faltung	10
2.3.2 Pooling Layers	13
2.3.3 Fully-Connected-Schicht	14
2.4 Kohonen-Netze	15
2.4.1 Competitive Learning	15
2.4.2 SOM-Algorithmus	17
3 Verwandte Arbeiten	20
3.1 Feature Extraction mit CNN für überwachte Bildklassifizierung	20
3.2 Unüberwachte pixelweise Klassifizierung von Bildern mit SOM	21
3.3 Simulation von Place Cells mithilfe eines neuronalen Netzes	22
4 Implementierung	23
4.1 Zielsetzung	23
4.2 Architektur	26
4.3 Unity-Frontend	28
4.3.1 Unity Game Engine	29
4.3.2 Simulation	29
4.3.3 Bildgenerierung	31
4.4 WebSocket als Schnittstelle	31
4.4.1 Client im Frontend	32
4.4.2 Server im Backend	33
4.5 Python-Backend	34
4.5.1 TensorFlow und Keras	35
4.5.2 CNN	35

4.5.3 SOM	38
5 Ergebnisse	41
5.1 Ergebnisse	41
5.1.1 SOM-Fehler	42
5.1.2 Place Units	43
6 Diskussion und Weiterentwicklung	45
6.1 Nutzen der Place-Cell-Simulation	45
6.2 Verwendung der Architektur als Klassifizierer	46
6.2.1 Versuch mit zwei unterschiedlichen Datensets	46
6.2.2 Weiterentwicklung des Systems als Klassifizierer	49
7 Fazit und Ausblick	51
Abkürzungsverzeichnis	v
Abbildungsverzeichnis	vi
Quellcodeverzeichnis	xii
Literatur	xiii

Kapitel 1

Einleitung

Anwendungen der künstlichen Intelligenz finden sich heutzutage in zahlreichen Bereichen der Industrie, der Dienstleistung aber auch des täglichen Lebens wieder und haben somit an zentraler Bedeutung im modernen Leben gewonnen. Besonders im letzten Jahrzehnt wurden im Bereich des maschinellen Lernens zahlreiche Erfolge erzielt, sodass künstliche Intelligenzen, wie beispielsweise Google's AlphaGo, den Menschen in einigen Disziplinen bereits übertreffen [1]. Wirft man allerdings einen Blick auf die Anfänge der Forschung im Bereich der künstlichen Intelligenz, so stellt man fest, dass dieses Gebiet vergleichsweise jung ist und der Begriff der *Artificial Intelligence* erst im Jahre 1956 durch McCarthy im Zuge eines Forschungsprojektes am Dartmouth College Einzug in die Forschung fand [2]. Die Motivation hinter besagtem Projekt war die Vermutung, dass jegliche Aspekte der menschlichen Intelligenz auch von Maschinen realisiert werden können [3]. Noch bis heute ist das Ziel, den menschlichen Verstand mithilfe von Computern nachzuempfinden, zentraler Bestandteil der Forschung auf dem Gebiet der künstlichen Intelligenz [4].

Mit dem Ziel mehr über das menschliche Gehirn herauszufinden werden häufig Experimente mit anderen Säugetieren durchgeführt, da diese dem Menschen neuroanatomisch ähnlich sind. Besonders Ratten werden bei solchen Versuchen häufig eingesetzt, da die Entwicklung ihres Gehirns mit der des Menschen vergleichbar ist und daher Schlüsse zum menschlichen Gehirn getroffen werden können [5]. In einer Reihe von Experimenten mit Ratten konnte nachgewiesen werden, dass der Hippocampus der Tiere, ein Bereich des Gehirns welcher zu erheblichem Teil für Erinnerungen zuständig ist [6], auch Aufschluss über die aktuelle Position der Ratte in ihrer Umgebung liefert [7]. In einem späteren Versuch wurden Ratten einen labyrinthartigen erhöhten Laufsteg entlang geschickt. Dabei konnte das Vorhanden-

sein von sogenannten Place Units im Hippocampus der Ratten gezeigt werden. Bei Place Units handelt es sich um Zellen, die ausschließlich dann besonders stark aktiviert werden, wenn das Tier sich an einem bestimmten Ort und in einer bestimmten Ausrichtung im Labyrinth befindet [8]. Die Aktivität dieser Place Units liefert also umgekehrt auch Aufschluss über die Position der Ratte im Labyrinth.

Das Ziel dieser Arbeit ist es, ein ähnliches Verhalten wie das der Place Units von Ratten mithilfe von maschinellem Lernen in einem Computersystem zu simulieren. Dabei wird unter Verwendung einer Game Engine, also eines Frameworks mit primärer Verwendung zur Entwicklung von Computerspielen, ein virtueller Pfad entworfen, welcher vom Benutzer frei abgelaufen werden kann. Gleichzeitig wird das von der Kamera wahrgenommene Bild zur Laufzeit ausgewertet und mithilfe von Algorithmen des maschinellen Lernens in eine von neun Kategorien eingeteilt. Diese Kategorien werden zuvor im unüberwachten Lernprozess vom System ermittelt und zielen darauf ab, zwischen den Bildern einer Kategorie eine möglichst hohe Ähnlichkeit zu erzielen. Jede Kategorie ist dabei vergleichbar mit einer Place Unit im Gehirn der Ratte, wobei jedes Bild immer nur genau einer Kategorie zugeordnet wird.

Im folgenden Kapitel werden zunächst die zugrundeliegenden theoretischen Konzepte erklärt, welche zum Verständnis der Arbeit nötig sind. Im Anschluss werden ähnliche, bereits existierende Projekte vorgestellt und die wesentlichen Unterschiede zu dieser Arbeit hervorgehoben. Das darauffolgende Kapitel beschäftigt sich dann mit der Umsetzung des Projekts. Dabei werden die Systemarchitektur sowie die zur Implementierung verwendeten Technologien und Algorithmen genauer beleuchtet. Das Kapitel unterteilt sich dabei in drei Unterkapitel, welche sich jeweils mit einer der drei zentralen Bestandteile des Systems auseinandersetzen. Im Anschluss daran werden die im Versuch erzielten Ergebnisse präsentiert und ausgewertet. Abschließend wird die im Projekt beschriebene Architektur bewertet, auf ihren potenziellen Nutzen für maschinelles Lernen überprüft und, aufbauend darauf, Schritte zur sinnvollen Weiterentwicklung vorgeschlagen.

Kapitel 2

Grundlagen

Dieses Kapitel liefert einen Einblick in die theoretischen Grundlagen des Projektes. Dazu werden die in der Implementierung verwendeten Konzepte und Algorithmen des maschinellen Lernens zunächst ausführlich erläutert.

2.1 Maschinelles Lernen

Falls nicht anders angegeben beruft sich dieses Unterkapitel auf Informationen aus Kapitel 1 und 2 des Buches „Artificial Intelligence Technology“ von Huawei Technologies Co., Ltd. [9].

Das Konzept von *Künstlicher Intelligenz* ist seit einigen Jahrzehnten bereits fester Bestandteil der Industrie und Forschung, fand seinen Weg aber durch Film und Fernsehen sowie intelligente Haushaltsgeräte oder Automobile auch in das tägliche Leben der meisten Menschen. Trotzdem gibt es bis heute keine eindeutige Abgrenzung des Begriffs. Eine der ersten und bis heute weitgehend akzeptierten Definitionen beschreibt das Gebiet der künstlichen Intelligenz mit dem Ziel, die menschliche Intelligenz mit Maschinen nachzuempfinden. Inzwischen werden dafür Prinzipien aus den verschiedensten Disziplinen verwendet, was Künstliche Intelligenz (KI) zu einem Forschungsfeld macht, welches fächerübergreifend Relevanz besitzt.

Einen der Unterbereiche von KI stellt das *maschinelle Lernen* dar. Dieses Gebiet beschäftigt sich dabei primär mit der Simulation von Lernprozessen. Man spricht immer dann von maschinellem Lernen, wenn ein System eine bestimmte Aufgabe, in Abhängigkeit von dazugewonnener Erfahrung, besser absolviert. Das Ziel dieses Projektes ist es, Bilder in Kategorien zu sortieren, wobei die Bilder innerhalb

einer Kategorie möglichst hohe Ähnlichkeit aufweisen sollen. In späteren Kapiteln wird sich zeigen, dass die Performance des Systems dabei mit wachsender Erfahrung, also je mehr Bilder es präsentiert bekommt, ebenfalls steigt. Es liegt also ein maschinellem Lernprozess vor.

Maschinelles Lernen eignet sich vor allem dann, wenn die zugrunde liegenden Daten oder die Zusammenhänge zwischen den Daten und dem gewünschten Output besonders komplex sind. In diesen Fällen sind klassische Algorithmen oder statistische Lösungsansätze wie beispielsweise Regression häufig mit hohem Aufwand verbunden oder schlichtweg nicht anwendbar. Oft kann es dann helfen das System die Zusammenhänge selbstständig erlernen zu lassen.

2.1.1 Überwachtes und unüberwachtes Lernen

Beim maschinellen Lernen unterscheidet man in der Regel zwischen zwei grundlegenden Lernansätzen, zwischen welchen man in Abhängigkeit vom konkreten Szenario, dem bekannten Wissen über die Daten und dem gewünschten Verhalten des Systems wählt.

Beim *überwachten Lernen* erhält das System während der Trainingsphase nicht nur die Input-Daten, sondern zusätzlich dazu auch den erwarteten Output. In der Regel vergleicht das System den generierten Output mit dem erwarteten und passt interne Parameter an, um den Fehler, also den Unterschied zwischen den beiden Werten, in zukünftigen Iterationen zu verringern. Handelt es sich bei dem Output um eine diskrete Größe, so spricht man auch von Klassifikation. Die Daten werden dabei einer der diskreten, im Lernprozess in der Form des erwarteten Outputs erhaltenen Klassen zugeordnet. Ein Beispiel für eine solche Klassifikation wäre ein System, welches Umfang, Gewicht und Farbe einer konkreten Frucht erhält und diese als Output in eine der diskreten Gruppen Apfel oder Birne einteilt.

Überwachtes Lernen erfordert, dass den Trainingsdaten im Vorhinein vom Menschen der gewünschte Output, oft auch Label genannt, zugeordnet wurde. Falls das Labeling der Daten zu aufwändig ist oder die passenden Label schlichtweg nicht bekannt sind, eignet sich überwachtes Lernen dementsprechend nicht. Stattdessen können in diesen Fällen *unüberwachte Lernprozesse* verwendet werden. Dabei erhält das System in der Lernphase lediglich die Daten und nicht den erwarteten Output. Daraus folgt, dass das System den Fehler zwischen Output und Label nicht

minimieren kann, wie es beim überwachten Lernen der Fall wäre. Stattdessen zielt unüberwachtes Lernen darauf ab, innerhalb der erhaltenen Daten möglichst ähnliche Einträge zu finden, welche also mit hoher Wahrscheinlichkeit derselben Kategorie angehören. Vermutet man also in einer Menge von Äpfeln beispielsweise zwei verschiedene Sorten, weiß aber nicht um welche Sorten es sich handelt oder was diese ausmacht, so kann man ein unüberwachtes System die Früchte anhand von z. B. Umfang, Gewicht und Farbe in zwei Gruppen unterteilen lassen.

Das in der Arbeit behandelte Projekt bedient sich sowohl überwachter als auch unüberwachter Lernverfahren.

2.2 Neuronale Netze

Das menschliche Gehirn besitzt eine Rechenkraft und -effizienz, welche die von Computern um ein Vielfaches übertrifft [10]. Ein zentraler Ansatz in der Forschung zum maschinellen Lernen ist es daher, die Funktionen des Gehirns mit Computern nachzuempfinden. Bereits 1943 veröffentlichten Warren McCulloch und Walter Pitts einen Vorschlag für ein Modell, welches die im Gehirn des Menschen vorhandenen Neuronen-Zellen imitieren soll [11]. Dabei zeigten sie, dass mit Vernetzungen dieser beschriebenen Neuronenmodelle ein Großteil aller logischen Funktionen umsetzbar ist. Aufbauend auf dem beschriebenen McCulloch-Pitts-Neuron wurden im Laufe der Jahre viele weitere verschiedene Modelle entwickelt, die man mit dem Sammelbegriff der *künstlichen neuronalen Netze* bezeichnet.

In diesem Kapitel werden die Grundlagen zu sowohl künstlichen neuronalen Netzen als auch ihrer biologischen Vorlage behandelt. Außerdem werden mit den Convolutional Neural Networks und den Kohonen-Netzen zwei konkrete Modelle vorgestellt, welche in der Implementierung dieser Arbeit Verwendung finden.

2.2.1 Das Neuron

Das menschliche Gehirn besitzt ungefähr 90 Milliarden Nervenzellen [12], welche auch Neuronen genannt werden und die Grundlage für alle Denkprozesse bilden. Obwohl es verschiedene Typen von Neuronen gibt, welche sich im Aufbau unterscheiden, teilen die meisten die in Abbildung 2.1 dargestellte grundlegende Struktur. Die Dendriten dienen als Eingänge, über die elektrische Signale von benach-

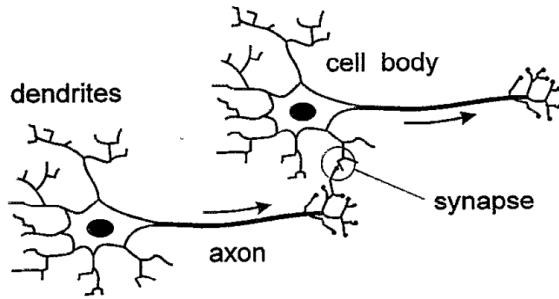


Abbildung 2.1: Typischer Aufbau eines Neurons im menschlichen Gehirn [10]. Die Dendriten übermitteln das Aktivierungssignal an das Neuron. Ist der Input stark genug, wird das Ausgangssignal entlang des Axons geführt, wo es über die Synapsen an benachbarte Neuronen weitergeleitet und dort wiederum als Eingangssignal erfasst wird.

barten Neuronen empfangen und an den Zellkörper weitergeleitet werden können. Ist die Erregung des Zellkerns durch die eingehenden Spannungen stark genug, so sendet der Zellkern selbst ein elektrisches Signal entlang des Axons aus. Die Axonenden, auch Synapsen genannt, stellen dieses Ausgangssignal mittels Neurotransmittern den zahlreichen benachbarten Neuronen zur Verfügung. Abhängig vom Typ der Synapse verstärkt oder vermindert das Signal so die Chance, dass das Nachbarneuron aktiviert wird. Wie stark sich dieser Effekt auswirkt, wird dabei durch den Stärkegrad der Synapse festgelegt [10].

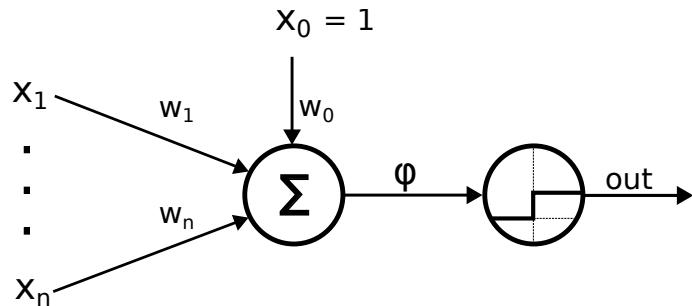


Abbildung 2.2: Aufbau eines einlagigen Perzeptron. Die Eingänge x_0 bis x_n werden jeweils mit dem zugehörigen Gewicht w_i multipliziert und aufaddiert. x_0 stellt dabei den Bias-Eingang dar, an welchem immer 1 anliegt. Überschreitet die gewichtete Summe der Eingänge φ den Wert 0, so wird das Perzeptron aktiviert und gibt den Wert 1 aus. Ist sie allerdings negativ, so wird der Output zu 0. Da am Eingang x_0 immer eine 1 anliegt, stellt das negative Biasgewicht $-w_0$ den Schwellwert dar, den die Summe aller anderen gewichteten Eingänge $\sum_{i=1}^n x_i w_i$ überschreiten muss, damit das Perzeptron aktiviert wird.

Nach dem Vorbild dieser biologischen Neuronen stellte Frank Rosenblatt 1958 ein Computersystem mit dem Namen Perzeptron vor [13]. Abbildung 2.2 zeigt ein solches Perzeptron in seiner einfachsten Form mit lediglich einer Schicht. Dieses einlagige Perzeptron ist vergleichbar mit einem einzelnen biologischen Neuron. Die Eingänge x_0 bis x_n können den Wert 0 oder 1 annehmen und entsprechen den Dendriten

im Neuron. Um den Stärkegrad der Synapsen zu simulieren wird jedem Eingang ein variables Gewicht w zugeteilt, welches mit dem Eingang multipliziert wird. Ähnlich wie im Zellkern des Neurons werden die eingehenden gewichteten Aktivitäten aufsummiert. Negative Gewichte an den Eingängen hemmen das Perzeptron also. Es ergibt sich folgende Formel:

$$\varphi = \sum_{i=0}^n x_i w_i \quad (2.1)$$

Überschreitet die gewichtete Summe φ den Wert 0, so wird das Perzeptron aktiviert und gibt als Output den Wert 1 zurück. Im anderen Fall bleibt die Ausgangsaktivität 0 [14]. Das Gewicht des Bias-Eingangs x_0 , an dem immer der Eingangswert 1 anliegt, legt also den Schwellwert fest, welchen die gewichtete Summe der anderen Eingänge überschreiten muss, um das Perzeptron zu aktivieren [9].

Ein solches einlagiges Perzeptron kann, durch Anpassung der Gewichte, z. B. die simplen Operatoren AND und OR realisieren. Einige Jahre später zeigten Minsky und Papert allerdings, dass nicht linear separierbare Mengen, welche zum Beispiel durch den XOR-Operator entstehen, nicht mithilfe eines einlagigen Perzeptron unterscheiden werden können [15]. Die Anwendungsfälle für einlagige neuronale Netze sind also entsprechend begrenzt.

2.2.2 Mehrschichtige neuronale Netze

Das Jahr 1986 wird auch als der Beginn der „Renaissance der Neuronalen Netze“ bezeichnet [2]. Nach fast zwei Jahrzehnten ohne weitere große Durchbrüche auf dem Gebiet der neuronalen Netze veröffentlichten Rumelhart, Hinton und Williams in diesem Jahr den ersten Vorschlag für ein lernfähiges mehrschichtiges neuronales Netz, welches die Einschränkungen des einlagigen Perzeptron lösen konnte [16]. Das sogenannte mehrlagige Perzeptron, wie in Abbildung 2.3 dargestellt, besitzt nicht nur Eingabe- und Ausgabeneuronen, sondern zusätzlich beliebig viele Zwischenschichten. Auf diese Weise können wesentlich komplexere Zusammenhänge vom Netz dargestellt werden. Das Besondere beim mehrlagigen Perzeptron ist, dass jedes Neuron einer Schicht mit jedem Neuron der Folgeschicht verbunden ist. Das bedeutet der Output der einen Schicht dient gleichzeitig als Input für alle Neuro-

nen der nächsten Schicht. Bei dieser Form von Netzen spricht man heutzutage von Feed-Forward-Netzwerken [9].

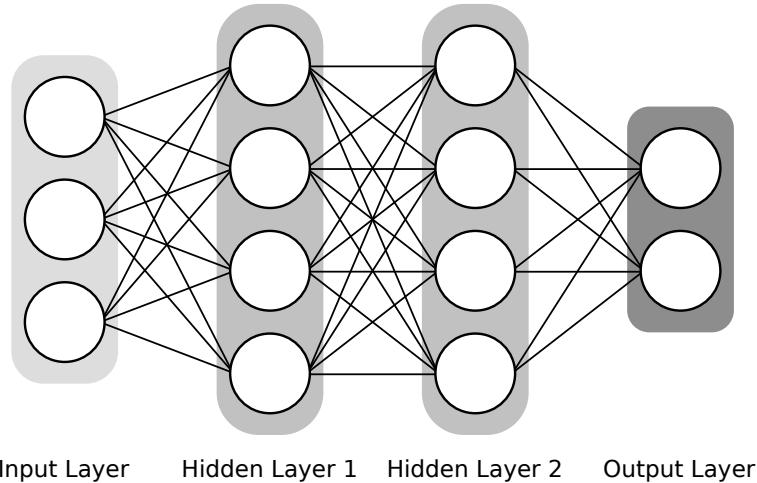


Abbildung 2.3: Aufbau eines von Rumelhart et al. beschriebenen mehrlagigen Perzeptron. Das dargestellte Netz besteht aus insgesamt vier Schichten. Dabei handelt es sich um den Input Layer, zwei Hidden Layer und den Output Layer. Die Neuronen des Input-Layers führen dabei keine Berechnungen durch, sondern symbolisieren lediglich die Eingangswerte des Netzes. Die Ausgänge eines Neurons dienen jeweils als Eingang für Neuronen der folgenden Schicht. Diese Form von voll vernetzten Netzwerken wird auch Feed-Forward-Netz genannt.

Insbesondere umfasst der Vorschlag mit der Backpropagation einen Lernalgorithmus zum Trainieren von mehrschichtigen Netzen. Dabei wird der Fehler mithilfe von Differenzialrechnung rückwärts über die Schichten geführt, sodass in jeder Schicht die Eingangsgewichte angepasst werden können. Dieses Verfahren ist allerdings nicht bei Neuronen mit Schwellwert als Aktivierungsfunktion, wie dem zuvor beschriebenen Perzeptron, einsetzbar. Übliche alternative Aktivierungsfunktionen stellen zum Beispiel die Sigmoid- oder die Rectifier-Funktion dar [10].

Sowohl Backpropagation als auch Feed-Forward-Netze erfreuen sich heutzutage weiterhin großer Beliebtheit bei der Umsetzung von Systemen des maschinellen Lernens. Feed-Forward-Netze sind aber nicht die einzige Form von neuronalen Netzen, die sich aus dem mehrlagigen Perzeptron entwickelt haben. Die folgenden zwei Kapitel beschäftigen sich mit zwei weiteren Formen von neuronalen Netzen, welche in der Implementierung dieser Arbeit Verwendung finden.

2.3 Convolutional Neural Networks

Die im vorherigen Unterkapitel beschriebenen Feed-Forward-Netzwerke sind flexibel und für diverse Anwendungsfälle einsetzbar. Es gibt allerdings Fälle, wo sie an ihre Grenzen stoßen.

Deep Learning stellt eine Subkategorie von maschinellem Lernen dar, bei der statt direkt von den Eingangsdaten abzuleiten, immer abstraktere Muster gebildet werden. Das bedeutet konkret, dass aus den Eingangsdaten zunächst einfache Muster extrahiert werden, aus welchen im Anschluss komplexere High-Level-Muster ermittelt werden können [17]. Im Kontext von neuronalen Netzen entspricht das einem Netzwerk mit einer großen Menge an Neuronenschichten. Dadurch können zwar deutlich komplexere Zusammenhänge gelernt werden, allerdings benötigen viele Schichten, besonders bei Feed-Forward-Netzwerken, auch eine höhere Menge an anpassbaren Parametern, also Gewichten [18].

Vor allem in der Verarbeitung von Bilddaten sind klassische Feed-Forward-Netze unbrauchbar. Für den Fall, dass ein Netzwerk beispielsweise die Pixelwerte eines 32x32 Farbbildes als Input erhält, so gäbe es $32 * 32 * 3 = 3072$ Inputneuronen im Netz. Durch die vollständige Vernetzung zweier benachbarter Schichten bei Feed-Forward-Netzwerken bräuchte also jedes Neuron in der Hiddenschicht 3072 Gewichte. Um die komplexen Muster in Bildern zu erkennen, benötigt das Netz außerdem eine große Anzahl an Neuronen im Netzwerk, was die Anzahl der Gewichte sehr schnell in unrealistische Höhen treibt [19].

Um dieses Problem zu umgehen wurde das *Convolutional Neural Network (CNN)* eingeführt. Der Name stammt von Convolution, dem englischen Wort für Faltung, also einer Matrixoperation, welche der Funktionsweise von CNNs zugrunde liegt. Faltungen sind besonders nützlich, wenn es darum geht Muster in Eingabedaten zu erkennen, was häufig in der Verarbeitung von Bildern oder natürlicher Sprache notwendig ist [19]. Convolutional Neural Networks können auf Inputdaten von beliebigen Dimensionen angewandt werden. Im Folgenden werden allerdings CNNs für zweidimensionale Daten, also Bilder, beschrieben.

Es gibt viele verschiedene Varianten in der Umsetzung von CNNs, allerdings teilen sich die meisten den grundlegenden Aufbau, der in Abbildung 2.4 dargestellt ist. In der Regel bestehen CNNs aus zwei eindeutig voneinander abgrenzbaren Subnetzen, dem Feature-Extraction-Netz und dem Feed-Forward-Netz. Die Convolutional Layers des Feature-Extraction-Netzes extrahieren dabei durch Faltung für den An-

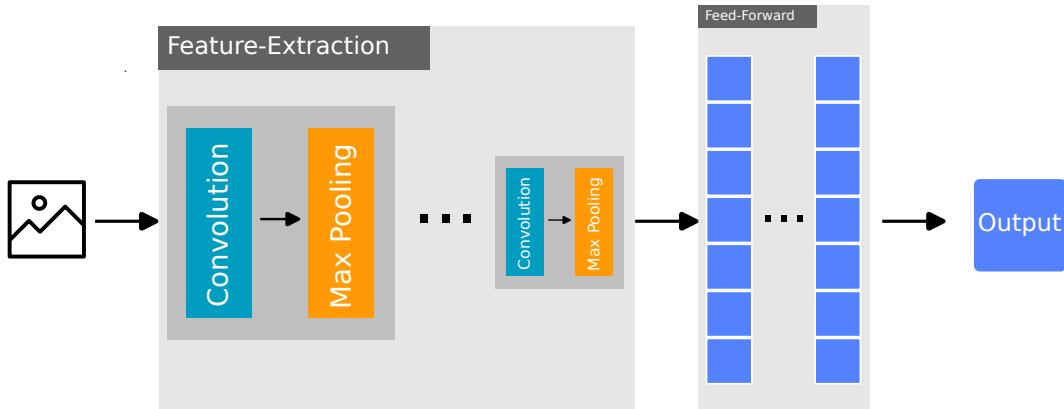


Abbildung 2.4: Grundlegender Aufbau eines Convolutional Neural Networks. Ein CNN ist in der Regel in zwei Teile, das Feature Extraction Network und ein Feed-Forward-Netzwerk abzugrenzen. Das Feature-Extraction-Netz besteht in seiner einfachsten Form wiederum aus einer alternierenden Folge von Faltungs- und Pooling-Schichten. In den Faltungsschichten oder auch Convolutional Layers werden aus dem Inputbild sogenannte Feature Maps extrahiert, welche im darauffolgenden Pooling Layer in ihrer Größe reduziert werden. Die letzten so entstandenen Feature Maps dienen als Input für das Feed-Forward-Netz, welches aus den extrahierten Merkmalen den gewünschten Output ableitet. Handelt es sich zum Beispiel um einen Klassifizierer für Bilder, könnte der Outputvektor die Wahrscheinlichkeiten dafür darstellen, dass das im Eingangsbild dargestellte Objekt der jeweiligen Klasse zugehört.

wendungsfall relevante Bildmerkmale, auch Features genannt. Zwischen zwei Convolutional Layers befindet sich in der Regel eine Pooling-Schicht, welche die entstandenen Feature Maps verkleinert. Beim Feed-Forward-Netz handelt es sich um ein klassisches, voll vernetztes, neuronales Netz, welches die Pixeldaten der letzten Feature Maps als Input erhält. Das Ergebnis des Feed-Forward-Netzes stellt den Output des CNNs dar [20]. Je nach Anwendungsfall kann die Struktur des Netzwerkes variieren und sogar andere Arten von Schichten enthalten. Die folgenden Abschnitte beschäftigen sich genauer mit den einzelnen Teilen und Operationen in einem CNN.

2.3.1 Faltung

Die Faltung oder Convolution ist die namensgebende und zentrale Operation im Convolutional Neural Network. Die Motivation dahinter ist, die bei Anwendungsfällen wie Bilderkennung häufig notwendige Mustererkennung in den Eingangsdaten zu erleichtern. Das Ziel von Faltungen ist es dementsprechend, über das gesamte Bild hinweg jeweils lokal in einem kleinen Bereich nach den gewünschten Mustern zu suchen. Dazu wird ein sogenannter Faltungskern oder Kernel, also üblicherweise eine 3×3 Maske, eingesetzt, welcher über das gesamte Inputbild geschoben wird.

An jeder Stelle werden die mithilfe der Maske ermittelten Werte aufaddiert und in ein Ergebnisbild, die Feature Map, geschrieben [20]. In Abbildung 2.5 ist dieser Prozess grafisch dargestellt. Die entstandenen Feature Maps geben also Aufschluss darüber, wo im Bild sich die detektierten Merkmale befinden. Abhängig von den Werten in einem Faltungskern können verschiedene Features, wie beispielsweise Kanten, erkannt werden. Abbildung 2.6 zeigt Beispiele für so entstehende Merkmalsbilder.

Die Werte der Faltungskerne werden im Lernprozess vom CNN angepasst, sodass eigenständig für den Output relevante Merkmale gelernt werden. Für jeden Punkt $x'_{n,m}$ im Feature Bild ergibt sich folgende Gleichung:

$$x'_{n,m} = \sum_{i=0}^2 \sum_{j=0}^2 x_{n+i,m+j} * k_{i,j} \quad (2.2)$$

Dabei steht $x_{n,m}$ für den Punkt an der Stelle n und m im Eingangsbild und $k_{i,j}$ für den Punkt im Faltungskern an der Stelle i und j . Bei genauerer Betrachtung der Gleichung erkennt man, dass sie der Summenformel des klassischen künstlichen Neurons entspricht, wobei der aktuell betrachtete Pixel in der Feature Map das Neuron darstellt. Die Pixelwerte des Eingangsbildes entsprechen somit den Werten der Eingangsneuronen und der Faltungskern definiert die Gewichte für jeden Eingang [19].

Auf diese Weise kann man die Anzahl der Parameter, also Gewichte, in einem Feed-Forward-Netz und in einem CNN vergleichen. Betrachtet man das in Abbildung 2.5 dargestellte Beispiel mit einem 5x5 Eingangsbild und einer 3x3 Faltungsmaske, so erkennt man, dass lediglich $3^2 = 9$, statt wie bei einem Feed-Forward-Netz $5^2 * 3^2 = 225$ lernbare Parameter benötigt werden. Während diese Zahl bei Feed-Forward-Netzen mit wachsenden Dimensionen der Inputbilder exponentiell ansteigt, bleibt sie bei CNNs konstant. CNNs lösen also weitestgehend das angesprochene Problem von zu vielen Gewichten in tiefen neuronalen Netzen.

Jeder Convolutional Layer im CNN besitzt in der Regel mehrere Faltungskerne, so dass in jeder Schicht viele Feature Maps entstehen. Häufig werden diese noch durch nichtlineare Funktionen, z. B. eine Rectifier-Funktion, angepasst. Danach werden die entstandenen Bilder in der Regel durch beispielsweise Max Pooling in ihrer Größe reduziert und dann wiederum als Input für die Faltung im nächsten Convolutional Layer verwendet [20]. Die nächste Schicht lernt also Features in den

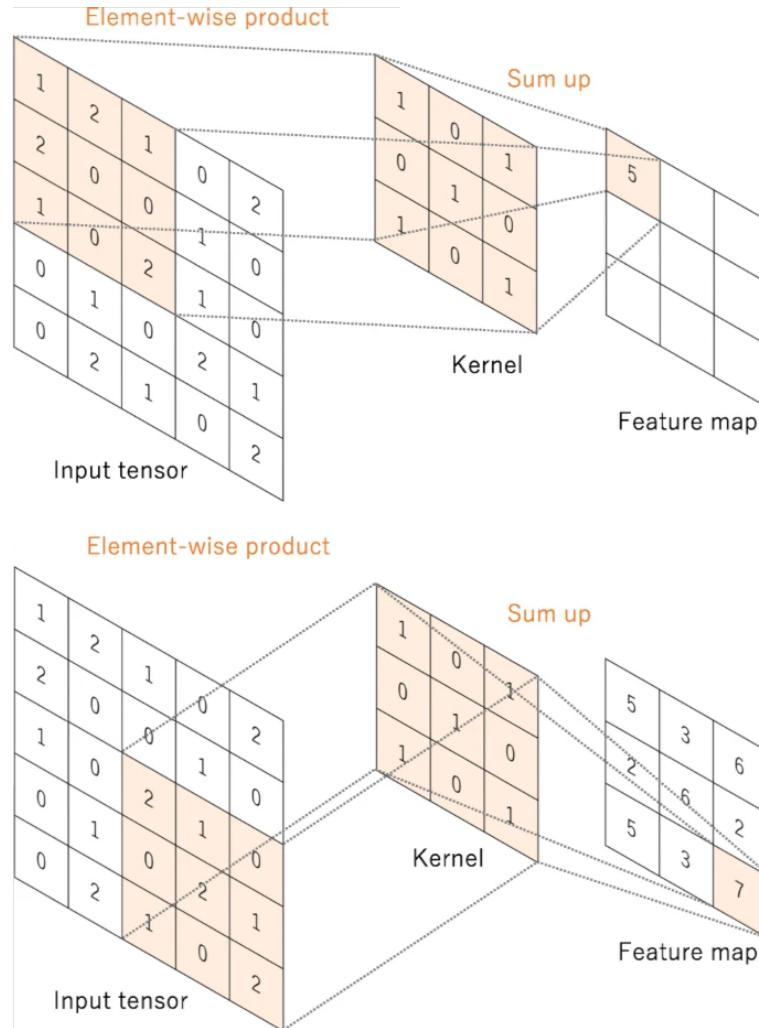


Abbildung 2.5: Grafische Darstellung der zweidimensionalen Faltungsoperation auf einem 5x5 Eingangsbild [20]. Der Faltungskern, in diesem Fall eine 3x3 Matrix, wird, beginnend in der oberen linken Ecke, über das Eingangsbild gelegt. Die Werte unter dem Faltungskern werden jeweils mit dem darüberliegenden Wert der Faltungsmaske multipliziert. Diese Produkte werden dann miteinander aufaddiert und der erhaltene Wert in die obere linke Ecke des Ergebnisbildes geschrieben. In der folgenden Iteration werden sowohl die Faltungsmaske über dem Eingangsbild als auch der zu beschreibende Pixel im Ergebnisbild um eine Koordinate verschoben. Dabei darf die Faltungsmaske niemals über den Rand des Bildes hinausragen. Dieser Vorgang wiederholt sich, bis die Faltungsmaske an jeder möglichen Position war und das Ergebnisbild gefüllt wurde. Besitzt das Eingangsbild die Dimensionen $n \times m$, so liefert die Faltung ein Ergebnisbild der Größe $(n-2) \times (m-2)$. Möchte man eine Änderung der Größe zwischen Input und Output vermeiden, so verwendet man ein Padding, wobei ein zusätzlicher Rahmen an Pixeln um das Ergebnisbild herum generiert wird.

extrahierten Feature Maps, was bei vielen Schichten dazu führt, dass die ermittelten Merkmale immer abstrakter werden, wie Abbildung 2.6 beispielhaft zeigt.



Abbildung 2.6: Stark vereinfachtes Beispiel für Faltungsmasken, welche in der Lernphase eines Klassifizierers gelernt werden könnten [17]. Die rechte Seite zeigt den abstrakten Aufbau eines CNN und die linke Seite das angelegte Eingangsbild. Die unterste Schicht des Netzes stellt die Inputdaten, also Pixelwerte aus dem Bild dar. Die folgenden drei Schichten zeigen die Faltungsmasken, welche in den Schichten des Netzes für die Faltung verwendet werden. Die Bilder repräsentieren dabei das Muster, welches bei der Anwendung der Faltungsmaske auf die Inputbilder aus den vorherigen Schichten detektiert wird. Während die unteren Schichten simple Muster wie Kanten erkennen, können höher liegende Schichten, aufbauend auf den detektierten Merkmalen der vorhergegangenen Schichten, komplexe Muster wie Räder oder Menschen erkennen.

2.3.2 Pooling Layers

In den Pooling Layers der Feature-Extraction-Schicht eines CNN werden die durch Faltung ermittelten Feature Maps in ihrer Größe verringert. Bei zweidimensionalen Bildern bedeutet das eine Verringerung der Auflösung der Merkmalsbilder. Dabei wird das Ausgangsbild in kleinere Bereiche unterteilt, die dann jeweils zu einem Wert zusammengefasst werden [20]. Wie groß die Bereiche sind und welche Schrittweite man zwischen den Bereichen wählt, beeinflusst, wie stark die Größenveränderung im Ergebnisbild gegenüber dem Ausgangsbild ausfällt. Bei Bereichen der Größe 2×2 und einer Schrittweite von zwei würde sich die Größe des Bildes in x- sowie y-Richtung auf die Hälfte verringern, wodurch das Bild insgesamt auf ein Viertel der ursprünglichen Größe schrumpft.

Wie genau die Pixel der jeweiligen Unterbereiche zusammengefasst werden hängt vom verwendeten Pooling-Verfahren ab. Besonders üblich ist dabei das Max Pooling,

aber auch z. B. Mean Pooling findet Verwendung [21]. Beim Max Pooling wird jeweils das Maximum des Subbereiches in das Ergebnisbild übernommen, während beim Mean Pooling der Mittelwert gebildet wird. Ein Beispiel für sowohl Max als auch Mean Pooling ist in Abbildung 2.7 dargestellt.

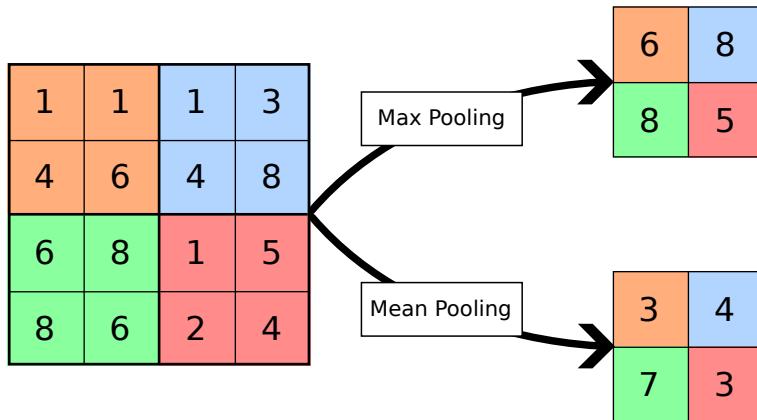


Abbildung 2.7: Grafische Darstellung der Pooling-Operation. Die linke Seite zeigt das Eingangsbild und die rechte Seite die beiden Outputbilder, die jeweils durch Max und durch Mean Pooling des Eingangsbildes entstehen. Beim Max Pooling wird jeweils das Maximum des betrachteten Bildbereiches ermittelt und in das Ergebnisbild übernommen. Beim Mean Pooling wird stattdessen der Mittelwert aus allen betrachteten Pixelwerten berechnet.

2.3.3 Fully-Connected-Schicht

Auf die Feature-Extraction-Schicht eines CNN folgt in der Regel ein klassisches Feed-Forward-Netzwerk, welches man oft als Fully-Connected-Schicht bezeichnet. Wie bei Feed-Forward-Netzen üblich dient dort der Output jedes Neurons als Input für alle Neuronen der Folgeschicht. Als Input erhält die Fully-Connected-Schicht die Pixelwerte der Feature Maps, welche in der letzten Schicht des Feature-Extraction-Netzes gewonnen wurden. Das Netz verwendet also die vorher gewonnenen Merkmale im Eingangsbild, um daraus den für den Anwendungsfall relevanten Output zu generieren. Damit die Feature Maps als Input für die Fully-Connected-Schicht verwendet werden können, werden sie in der Regel in einen eindimensionalen Vektor transformiert [20].

Die letzte Schicht des Fully-Connected-Netzes und damit auch der Output des CNN kann, abhängig vom Anwendungsfall, unterschiedliche Formen annehmen. Üblich ist z. B. für Klassifizierung in n Klassen eine Schicht mit n Output-Neuronen, wobei der Wert jedes Neurons für die Wahrscheinlichkeit steht, dass das im Eingangsbild dargestellte Objekt zu der dazugehörigen Klasse zählt [20].

2.4 Kohonen-Netze

Die bisher vorgestellten Arten von neuronalen Netzen stellen alle überwachte Lernverfahren dar, da der zur Anpassung der Gewichte notwendige Fehler des Outputs nur mit einem Zielwert für den Output berechnet werden kann, welcher dem Netzwerk mitgeliefert werden muss. Es gibt allerdings auch unüberwachte Lernverfahren, die der Gruppe der neuronalen Netze zugeordnet werden, auch wenn die Funktion der Neuronen dabei nicht unbedingt der des McCulloch-Pitts-Neurons entspricht. Ein Beispiel für ein solches unüberwachtes neuronales Netz stellen die Kohonen-Netze dar.

Kohonen-Netze oder auch Kohonen-Karten verdanken ihren Namen dem Ingenieur Teuvo Kohonen, welcher diese erstmals 1981 unter dem Namen *Self-Organizing Map (SOM)* oder zu Deutsch *Selbstorganisierende Karte* vorstellte [22]. Im Rest der Arbeit wird diese Art von Netz mit der üblichen Abkürzung SOM bezeichnet. Falls nicht anders angegeben bezieht sich der Inhalt dieses Abschnitts auf das Paper „The Self-Organizing Map“ [23], in welchem Kohonen eine Zusammenfassung der Grundlagen seiner Architektur bereitstellt.

SOMs legen nicht den für natürliche Neuronen typischen Aufbau von der Berechnung eines Outputs mithilfe von gewichteten Eingangssignalen zugrunde. Stattdessen realisieren sie eine andere zentrale Eigenschaft von Neuronen im menschlichen Gehirn, nämlich deren räumliche Abhängigkeit. Studien am menschlichen Nervensystem haben gezeigt, dass dieses kartenähnliche Neuronenstrukturen beinhaltet, bei denen dicht beieinanderliegende Neuronen Abhängigkeiten in ihrer Funktion aufweisen [24]. Um dieses Verhalten zu simulieren wird zunächst eine solche, in der Regel zweidimensionale, Neuronenkarte angelegt, welche daraufhin unüberwacht trainiert wird und sich dabei an die Eingangsdaten so anpasst, dass Neuronen, die in der Karte benachbart sind, auf ähnliche Eingangsdaten reagieren. Der dabei zugrundeliegende Algorithmus wird im Folgenden erläutert.

2.4.1 Competitive Learning

Competitive Learning stellt eine Untergruppe von Prinzipien des maschinellen Lernens dar, zu welcher auch die SOM gezählt wird. Beim Competitive Learning konkurrieren in der Regel zufällig initialisierte Einheiten miteinander, wobei der Ge-

winner auf den eingehenden Input reagieren kann und auf diese Weise den Output bzw. den Folgezustand des Systems beeinflusst [25].

Ein simpler kompetitiver Lernansatz wäre zum Beispiel der Folgende. Zunächst werden i n -dimensionale Referenzpunkt m_i gewählt, wobei n der Anzahl der Dimensionen der Eingangsdaten entspricht. Die Initialisierung dieser Referenzpunkt kann dabei zufällig erfolgen. Während des Lernprozesses wird in jeder Iteration ein Eingangspunkt x betrachtet. Man berechnet nun einen Ähnlichkeitswert s_i für jedes m_i , welcher beschreibt, wie ähnlich der Referenzpunkt zum Eingangspunkt ist. Wie sich dieser Ähnlichkeitsscore berechnet kann je nach Anwendungsfall individuell entschieden werden. Der Referenzpunkt mit der größten Ähnlichkeit wird im Sinne des kompetitiven Lernansatzes als Gewinner bezeichnet. Dieser Punkt wird nun so bewegt, dass s_i sich erhöht, er also ähnlicher zum Eingangspunkt wird. Alle anderen Referenzpunkt bleiben in diesem Schritt unverändert. Eine solche Lerniteration ist in Abbildung 2.8 bildlich dargestellt.

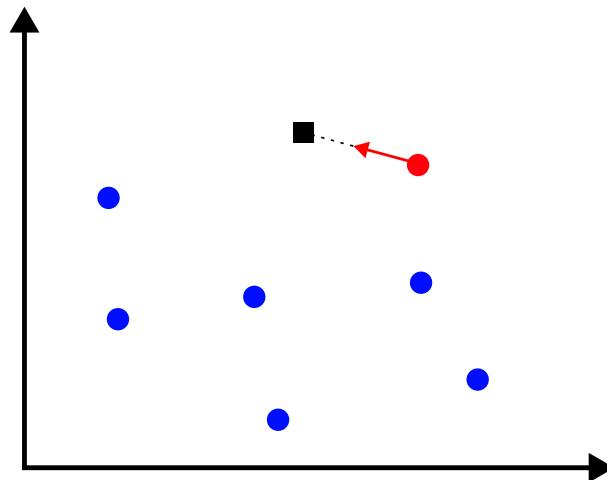


Abbildung 2.8: Darstellung eines Lernschrittes eines kompetitiven Lernalgorithmus. Die Punkte stellen die zweidimensionalen Referenzpunkte dar. Das Viereck beschreibt den Eingangspunkt im aktuellen Lernschritt. Der rote Referenzpunkt besitzt die größte Ähnlichkeit zum Eingangspunkt, in diesem Fall bestimmt als Inverses der Distanz zwischen den Punkten, und ist daher der Gewinner. Somit wird die Ähnlichkeit zwischen diesem Punkt und dem Eingangspunkt vergrößert, also in diesem Fall die Distanz verringert, indem der Referenzpunkt auf den Eingangspunkt zu bewegt wird.

Nach einer beliebigen Anzahl von Lernschritten wird der Lernprozess als beendet erklärt. Es wird weiterhin auf dieselbe Weise der ähnlichste Referenzpunkt bestimmt, allerdings wird dieser nicht mehr verändert, sondern stellt den Output des Systems dar. Im einfachsten Fall handelt es sich zum Beispiel um einen Klassifizierer, welcher durch Datenpunkte beschriebene Objekte in i Klassen einteilt, wobei jeder Punkt m_i eine solche Klasse beschreibt.

2.4.2 SOM-Algorithmus

Der Lernalgorithmus für Self-Organizing Maps baut auf dem soeben beschriebenen klassischen kompetitiven Lernverfahren auf, allerdings wird außerdem eine räumliche Abhängigkeit der Referenzpunkt zugrunde gelegt. Dazu wird zunächst eine räumliche Karte angelegt, indem beliebig viele Referenzpunkte m_i , auch Prototypen genannt initialisiert werden. Auch hier kann die Initialisierung der Vektoren zufällig geschehen. Im Anschluss legt man für jeden Prototyp eine Nachbarschaft fest. In Darstellungen solcher Karten wird diese Nachbarschaft oft durch eine Linie zwischen den beiden Einheiten symbolisiert. Abbildung 2.10 zeigt eine auf dieses Weise erstellte räumliche Karte mit neun Prototypen und einer klassischen Vierer-Nachbarschaft.

Stellt man diese Punkte in einem Koordinatensystem dar, wobei die von den Neuronen beschriebenen Vektoren als Ortsvektoren dienen, so ist zunächst keine räumliche Abhängigkeit in Bezug auf die Nachbarschaften zu erkennen, da die Initialisierung der Prototypen zufällig vorgenommen wird. Diese wird erst durch den Lernprozess sichergestellt. Ähnlich wie beim klassischen kompetitiven Lernen wird zunächst für einen eingehenden Input-Punkt x der ähnlichste Prototyp oder auch die *Best Matching Unit (BMU)* ermittelt. Dieser wird im Folgenden als m_c bezeichnet.

Als Ähnlichkeitsmaß wird bei SOMs in der Regel die inverse euklidische Distanz verwendet. Das bedeutet zwei Punkte sind sich ähnlich, wenn der Abstand zwischen ihren Ortsvektoren besonders gering ist. Die euklidische Distanz wird wie folgt berechnet:

$$d = \sqrt{\sum_{j=1}^n (x_j - m_j)^2} \quad (2.3)$$

Dabei stellt d die euklidische Distanz zwischen dem Eingangspunkt und einem Prototypen dar. n steht für die Anzahl der Dimensionen in den Ortsvektoren der beiden Punkte und x_j und m_j bezeichnen jeweils die Koordinate an Stelle j in den Ortsvektoren des Eingangspunktes x und des Prototyps m .

Statt nun allerdings lediglich die BMU an den Input-Punkt anzupassen, also in dessen Richtung zu bewegen, wird zusätzlich auch die (erweiterte) Nachbarschaft der BMU bewegt. Die Nachbarschaft N_c der BMU m_c wird dabei als die Menge der Punkte definiert, welche von m_c aus über r direkte Nachbarschaften erreicht werden können. r bezeichnet dabei den Nachbarschaftsradius. Dieser kann sich häufig auch im Laufe des Lernprozesses verringern, sodass zunächst größere Nachbarschaften

in die richtige Richtung bewegt werden, um der zufälligen Initialisierung möglichst drastisch entgegenzuwirken, in späteren Lernschritten aber trotzdem feinere Anpassungen getroffen werden können. In den letzten Lernschritten ist sogar $N_c = \{m_c\}$ denkbar, sodass lediglich die BMU bewegt wird.

Da als Ähnlichkeitsmaß die inverse euklidische Distanz gewählt wurde, muss d nun für die ermittelten Punkte verringert werden. Dafür ist die folgende Vorgehensweise gebräuchlich:

$$m_i(t+1) = \begin{cases} m_i(t) + \alpha(t)(x(t) - m_i(t)), & \text{falls } m_i \in N_c(t) \\ m_i(t), & \text{falls } m_i \notin N_c(t) \end{cases}$$

Hier bezeichnet $m_i(t)$ den Vektor des Neurons m_i im Lernschritt t . Es werden also die Referenzneuronen für den nächsten Lernschritt in Abhängigkeit der aktuellen Referenzpunkte sowie des in diesem Schritt anliegenden Inputs angepasst. Dabei wird der Richtungsvektor von $m_i(t)$ nach $x(t)$ als Differenz der beiden Ortsvektoren bestimmt und $m_i(t)$ in Abhängigkeit von α in diese Richtung bewegt, falls es sich um ein Element der aktuell betrachteten Nachbarschaft um $m_c(t)$ handelt. α beschreibt dabei den Anpassungsgrad der SOM. Es gilt $0 < \alpha < 1$. Üblicherweise sinkt α im Laufe des Trainingsprozesses, um in späteren Iterationen feinere Anpassungen zu erlauben. Für die Punkte in N_c verringert sich also die Distanz d zum Eingangspunkt. Alle Punkte außerhalb von N_c werden nicht verändert. Es ist außerdem üblich, den Anpassungsgrad für Neuronen, welche in der Neuronenkarte weiter von der BMU entfernt sind, zu verringern, um zu verhindern, dass zwei benachbarte Neuronen dasselbe Muster lernen. In Abbildung 2.10 ist ein solcher Lernschritt grafisch dargestellt.

Indem in jedem Lernschritt zusätzlich auch die Nachbarneuronen der BMU in Richtung des Inputs bewegt werden, wird sichergestellt, dass benachbarte Neuronen im durch ihre Vektoren beschriebenen Raum ebenfalls räumlich dicht beieinander liegen und somit auf ähnliche Eingangsmuster reagieren. Auf diese Weise kann die gewünschte räumliche Abhängigkeit der Neuronen in der Neuronenkarte erzielt werden. Betrachtet man den Fortschritt einer SOM über eine Vielzahl von Lernschritten, so ergibt sich bei erfolgreichem Lernen das in Abbildung 2.9 dargestellte Phänomen. Die zunächst zufällig initialisierte Neuronenkarte faltet sich über mehrere Iterationen hinweg auf und deckt dabei den durch den Input beschriebenen Datensatz möglichst optimal ab. Da benachbarte Neuronen auf ähnliche Muster im

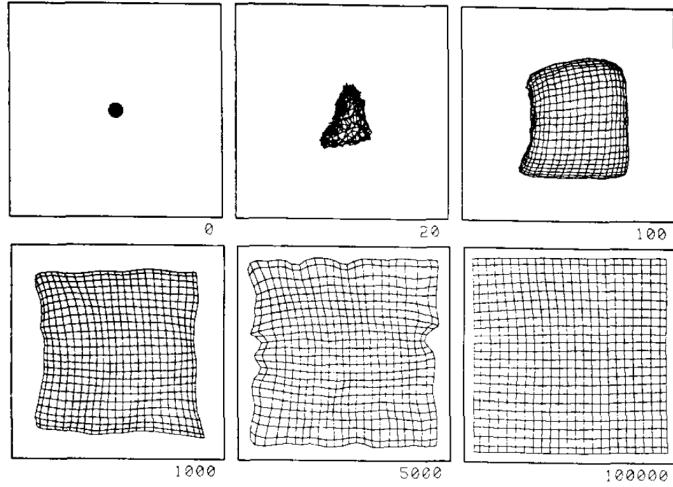


Abbildung 2.9: Eine über viele Lernschritte hinweg betrachtete Self Organizing Map im Vektorraum [23]. Die Inputwerte sind zufällig innerhalb des gezeigten Quadrates initialisiert worden. Es zeigt sich, dass die Neuronenkarte sich im Vektorraum entfaltet, um die über den gesamten Bereich verteilten Inputdaten so gut wie möglich abdecken zu können.

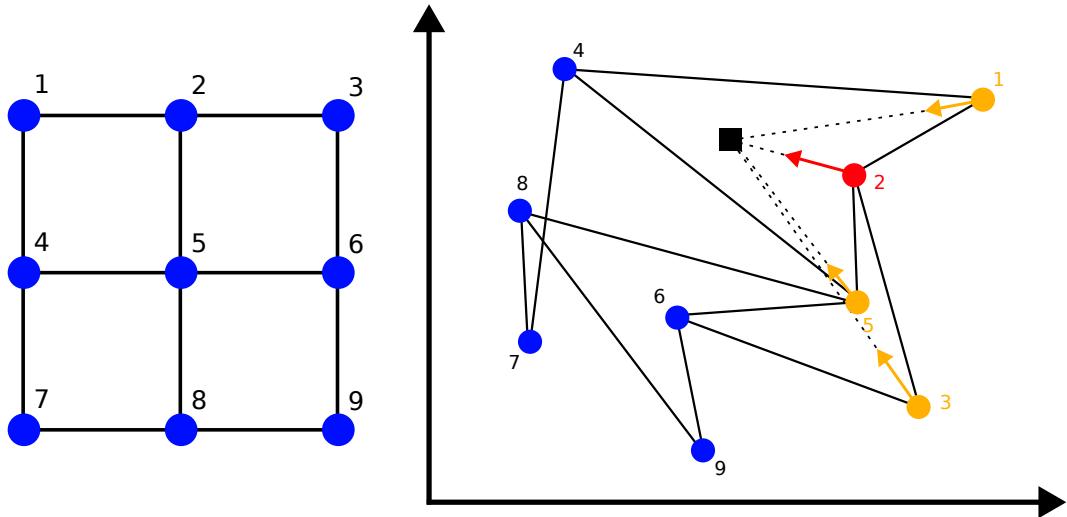


Abbildung 2.10: Darstellung einer Self Organizing Map. Das linke Bild zeigt dabei die Karte in Form eines Graphen, in welcher die Neuronen als Knoten und die Nachbarschaften zwischen zwei Neuronen als Kanten zwischen den beiden Knoten dargestellt werden. Um die Neuronen dabei eindeutig voneinander unterscheiden zu können wurden die Knoten nummeriert. Im linken Bild wurde der Graph in ein 2D-Koordinatensystem übertragen, wobei die Position der Neuronen, also der Prototypen, durch deren Vektor gegeben ist. Das Viereck symbolisiert den vom Eingangsvektor definierten Punkt $x(t)$ in einem Lernschritt t . Das rot gekennzeichnete Neuron stellt die BMU dar, während die gelb markierten Punkte die Neuronen darstellen, welche in der Nachbarschaft N_c liegen. In diesem Fall ist der Radius der Nachbarschaft 1, da wir nur diejenigen Neuronen betrachten, welche von der BMU über maximal eine Kante erreicht werden können. Zusätzlich ist die Änderung der Ortsvektoren der Neuronen in Richtung des Eingangspunkts in Form von Pfeilen gekennzeichnet.

Vektorraum reagieren, lässt sich die Struktur der Neuronenkarte in den späteren Schritten immer besser erkennen.

Kapitel 3

Verwandte Arbeiten

Im Mittelpunkt dieser Arbeit steht die Umsetzung einer unüberwachten Kategorisierung von Bildern in eine vorgegebene Anzahl von Kategorien. Die Kategorisierung findet dabei mithilfe einer SOM statt und die dafür benötigten Eingangsvektoren werden in Form von Bildmerkmalen mithilfe eines modifizierten, vortrainierten CNN aus den Bildern generiert. Dieses Kapitel beschäftigt sich mit ähnlichen Arbeiten anderer Autoren, welchen eine vergleichbare Zielsetzung zugrunde liegt. Dabei werden drei solcher Arbeiten vorgestellt, zentrale Ideen dieser kurz zusammengefasst und im Anschluss Gemeinsamkeiten und Unterschiede zu dieser Arbeit hervorgehoben.

3.1 Feature Extraction mit CNN für überwachte Bildklassifizierung

Dimpy Varshni et al. schlagen in ihrer Arbeit [26] ein Modell zur überwachten Kategorisierung von visuellen Patientendaten vor. Dabei werden Röntgenaufnahmen der Lunge der Patienten abhängig davon unterteilt, ob eine Lungenentzündung in der betrachteten Lunge festgestellt wurde oder nicht. Um dieses Ziel zu erreichen werden zunächst mithilfe eines auf dem ImageNet-Datenset [27] vortrainierten CNN für die Klassifikation möglicherweise relevante Features extrahiert. Dafür wird die Klassifikationsschicht des Netzes entfernt und der in der davorliegenden Schicht erzeugte Output als 50176-elementiger Featurevektor interpretiert, welcher daraufhin als Input für einen überwachten Klassifizierer verwendet wird. Experimente ergeben, dass eine Kombination des DenseNet-0169-Netzes für die Featureextraktion

und eine Support Vector Machine (SVM) als Klassifizierer bestmögliche Ergebnisse liefert.

Dieser Vorschlag weist deutliche Ähnlichkeiten zu dem in dieser Arbeit beschriebenen Modell auf und zeigt insbesondere, dass die Merkmalsextraktion mit einem vortrainierten CNN erfolgreich auf andere Anwendungsbereiche übertragen werden kann. Aufgrund des Vorhandenseins von Ground Truth in dem zugrundeliegenden Datenset kann in dem beschriebenen Fall allerdings auf eine SVM als ein überwachtes Kategorisierungsverfahren zurückgegriffen werden, während in dieser Arbeit die Kategorisierung der Bilder unüberwacht mithilfe einer SOM durchgeführt wird.

3.2 Unüberwachte pixelweise Klassifizierung von Bildern mit SOM

In dem von Jie Zhang und John Kerekes in [28] vorgeschlagenen Vorgehen werden Satellitenbilder von urbanen Regionen pixelweise mithilfe einer SOM unüberwacht klassifiziert. Dabei werden zunächst für jeden Bildpixel insgesamt 176 verschiedene Features ermittelt, welche unter Verwendung von Grauwertmatrizen direkt auf den Bilddaten berechnet werden. Im nächsten Schritt werden die Features aller Pixel eines Bildes an eine untrainierte SOM überreicht, welche diese pro Pixel als Inputvektor betrachtet und die BMU sowie deren Nachbarn im Verlauf entsprechend anpasst. Auf die Prototypenkarte der SOM wird im Anschluss eine Wasserscheidentransformation angewandt, wodurch die Prototypen in Klassen zusammengefasst werden. Die Höhe eines Prototyps wird dabei dadurch bestimmt, wie häufig dieser im Lernprozess als BMU ermittelt wird. Ein Bildpixel wird nun jener Klasse zugeordnet, der auch seine BMU nach der Wasserscheidentransformation angehört.

Im Vergleich zu dieser Arbeit können drei zentrale Unterschiede zwischen den Verfahren gefunden werden. Einerseits werden die Bilder hier pixelweise klassifiziert, sodass eine Klasse für jeden Bildpixel, statt lediglich für das gesamte Bild, ermittelt wird. Des Weiteren werden die Klassen nicht direkt durch die SOM-Neuronen abgebildet. Es wird stattdessen eine größere Neuronenkarte gewählt, in der die Neuronen nachträglich zusammengefasst werden. Außerdem ist hervorzuheben, dass es sich bei dem beschriebenen Ansatz um ein rein unüberwachtes Lernverfahren handelt. Anders als in dieser Arbeit wird kein auf überwachte Weise vortrainiertes CNN verwendet.

3.3 Simulation von Place Cells mithilfe eines neuronalen Netzes

Die Simulation von Place Cells in Gehirnen von Ratten mithilfe von neuronalen Netzen ist keine Neuheit, allerdings werden dabei häufig keine Daten in Form der von der Ratte wahrgenommenen Bilder verwendet. Ein solcher Versuch wurde bereits im Jahr 1991 von Patricia E. Sharp vorgeschlagen [29]. Hier wird sich an einer anderen Studie zum Verhalten der Place Cells bei Ratten orientiert, in welchem die Tiere in runde, abgeschlossene Umgebung platziert wurden. Eine ähnliche Umgebung wird mithilfe eines Computers nachempfunden und die Bewegung einer virtuellen Ratte in diesem Raum simuliert. Das Netz, welches die Aktivierung der jeweiligen Place Cell bestimmt, basiert auf einem simplen, unüberwachten kompetitiven Lernverfahren und beinhaltet 3 Schichten. Die Eingangsschicht beinhaltet Neuronen, die den Zellen im Neokortex der Tiere nachempfunden sind, und die von den Tieren wahrgenommenen Reize in das Netz geben. Die Aktivität dieser Zellen hängt dabei von der Position der Ratte im Verhältnis zu am Rand des Raumes positionierten Stimulanten ab. Die Eingangsneuronen beeinflussen nun die Aktivität der Neuronen der Zwischenschicht, welche wiederum die Place Cells in der letzten Schicht anregen. Nach dem „Winner takes all“-Prinzip feuert zu jedem betrachteten Zeitpunkt nun genau eine der Place Cells. Auf diese Weise kann ein Verhalten simuliert werden, welches der in echten Tieren beobachteten Neuronenaktivität ähnelt.

Neben dem alternativen Aufbau von Simulation, Eingangsdaten und Bestimmung der aktiven Place Unit ist dabei ein weiterer wesentlicher Unterschied zu dieser Arbeit, dass verschiedene Ratten mit verschiedenen zuvor zufällig initialisierten Startparametern simuliert werden. Dabei werden dem System die Reize, die die Ratte wahrnimmt, außerdem in einer realistisch erzeugbaren Reihenfolge präsentiert, während in dieser Arbeit die Eingangsmuster in Form der Bilder in der Trainingsphase zufällig aus den verschiedenen Gebieten entnommen werden.

Kapitel 4

Implementierung

In diesem Kapitel wird die Implementierung des Projektes genauer erläutert. Dabei wird unter anderem auf die Architektur, die Algorithmen und die Technologien eingegangen, die zur Umsetzung der in Kapitel 1 beschriebenen Zielsetzung verwendet werden. Dazu wird zuerst die Gesamtarchitektur des Systems betrachtet, um dann im Anschluss genauer auf die einzelnen Komponenten einzugehen.

Dieses Kapitel beschäftigt sich dabei lediglich mit dem Grundaufbau des Projektes, welcher zunächst ausführlich definiert wird. Anpassungen für Experimente mit alternativen Szenarien oder Lösungsansätzen werden im nächsten Kapitel spezifisch für jeden Sonderfall erläutert. Der hier behandelte Code ist vollständig im zugehörigen GitHub-Repository¹ verfügbar.

4.1 Zielsetzung

Das Ziel der Arbeit ist es, das Verhalten von im Gehirn von einigen Säugetieren vorhandenen Place Units, welche abhängig von der Position und Ausrichtung des Organismus in seiner Umgebung aktiviert werden [8], mit Methoden des maschinellen Lernens nachzuempfinden. Dabei soll es nicht in erster Linie darum gehen ein exaktes Modell der Place Units zu entwickeln, sondern stattdessen wesentliche Verhaltensweisen der Place Units zu extrahieren und darauf basierend praxisrelevante Algorithmen und Lösungen aufzubauen.

¹<https://github.com/j-lieske/place-cell-simulation>

Der Aufbau der Experimente orientiert sich am in [30] beschriebenen Versuch von Davidson et al. Dabei läuft eine Ratte einen Laufsteg oder allgemeiner ein Labyrinth entlang, während gleichzeitig die Aktivitäten der Place Units gemessen werden. Das zu entwickelnde System benötigt dementsprechend ein solches Labyrinth, also eine Umgebung, auf welche die Place Units reagieren können. Zusätzlich gibt es einen Agenten, welcher das Äquivalent zur Ratte darstellt. Dieser bewegt sich durch das Labyrinth und sammelt dabei Eindrücke aus seiner Umgebung, welche auf Position und Rotation schließen lassen und aufgrund derer die Place Units aktiviert werden können. Die Place Units selbst sollen mithilfe von Methoden des maschinellen Lernens realisiert werden.

Beim Labyrinth soll es sich um eine 3D-Umgebung handeln, welche möglichst dynamisch durchlaufen werden kann. Da ein physischer Aufbau einer solchen Umgebung mit einem hohen Zeit- sowie eventuellem Kostenaufwand verbunden ist, wird dieses Labyrinth stattdessen simuliert. Dafür bietet sich die Verwendung einer Game Engine an. Hierbei handelt es sich um ein Framework, welches primär zur Entwicklung von Videospielen gedacht ist und somit Möglichkeiten bietet, einen virtuellen Agenten durch eine 3D-Umgebung navigieren zu lassen. Für dieses Projekt wird die Game Engine Unity² verwendet, welche kostenfrei zur Verfügung steht und 3D-Spielentwicklung unterstützt. Mehr Informationen zu Unity finden sich im Unterabschnitt 4.3.1.

Place Cells haben die Eigenschaft selektiv genau dann besonders stark aktiviert zu werden, wenn das Lebewesen sich an einer bestimmten Stelle in seiner Umgebung befindet. Dabei sind oft auch mehrere dieser Zellen gleichzeitig aktiv. Außerdem kann es Orte geben, an denen gar keine Aktivität von Place Units erkennbar ist [8]. Um das Problem allerdings zu vereinfachen und die Ergebnisse besser veranschaulichen und transferieren zu können, wird in diesem Projekt vorausgesetzt, dass zu jedem Zeitpunkt genau eine Place Cell aktiviert wird. Um ein solches Verhalten zu realisieren, bietet sich ein Verfahren des kompetitiven Lernens an. Hierbei gibt es immer lediglich eine Gewinnereinheit, welche den Output, also in diesem Fall die aktive Place Cell, bestimmt.

Die Daten, welche der Agent über seine Position innerhalb des Labyrinths sammelt, werden in Form von Bildern seiner unmittelbaren Umgebung gespeichert. Dabei handelt es sich um das Äquivalent dessen, was ein Versuchstier im Labor mit seinen Augen wahrnehmen könnte. Hierbei ist anzumerken, dass gezeigt wurde, dass

²<https://unity.com>

die Aktivität der Place Units bei Tieren nicht nur von einem visuellen Stimulus abhängig ist [8]. Da das Ziel der Arbeit allerdings nicht in erster Linie der Entwurf eines möglichst biologienahen Modells ist, wird hier die Vereinfachung vorgenommen, die Place Units lediglich durch die visuelle Wahrnehmung des Agenten zu stimulieren.

Aus den beiden beschriebenen Annahmen ergibt sich, dass es sich bei der Aktivierung der Place Units in der Implementierung dieses Projektes effektiv um eine Klassifikation von Bildern handelt. Bei der Kategorie, welche für ein konkretes Bild ermittelt werden soll, handelt es sich dabei um die Place Unit, welche bei Betrachtung des im Bild dargestellten Ausschnittes des Labyrinths durch den Agenten aktiviert wird. Es bietet sich an, für den Lernprozess eine Kohonen-Karte anzulegen und diese mit den Inputdaten, welche der Agent produziert, zu trainieren. Dabei werden die Bilder zunächst mithilfe der Feature-Extraction-Schicht eines vortrainierten Convolutional Neural Networks in Vektoren umgewandelt, welche zur Bildkategorisierung hilfreiche Features in den Bildern kodieren. Mithilfe dieser Vektoren kann im Anschluss die SOM trainiert werden. Der Vorteil bei diesem Vorgehen ist, dass die trainierten Place Units, also die Neuronen in der Kohonen-Karte, lokal voneinander abhängig sind. Das bedeutet benachbarte Neuronen erkennen ähnliche Muster und reagieren damit auf ähnliche Umgebungen im Labyrinth. Überträgt man dieses Modell auf andere Anwendungen, so kann das System also nicht nur eigenständig Kategorien in einem Bilddatensatz finden, sondern auch zusätzlich Abhängigkeiten bzw. Ähnlichkeiten zwischen den Kategorien erkennen. Mehr zu den Vorteilen dieses Ansatzes und anderen möglichen Anwendungsbereichen für die Architektur findet sich in Kapitel 6.

Bei der Implementierung der im Projekt verwendeten Konzepte des maschinellen Lernens wird Python³ verwendet. Bei Python handelt es sich um eine in Bereichen des maschinellen Lernens etablierte Programmiersprache, welche diverse Bibliotheken zur Verfügung stellt, die die Implementierung von KI-Systemen vereinfachen. Die Machine Learning API Keras⁴ ermöglicht es dabei unter anderem bereits vortrainierte CNN-Modelle in ein Projekt einzubinden und auf die notwendigen Bedürfnisse anzupassen. Auf diese Weise wird die Feature Extraction aus den Inputbildern realisiert. Die darauffolgende Kohonen-Karte und die dafür notwendigen

³<https://www.python.org>

⁴<https://keras.io/about/>

Funktionalitäten werden hingegen von Grund auf implementiert. Mehr zu den verwendeten Bibliotheken und Algorithmen findet sich im Abschnitt 4.5.

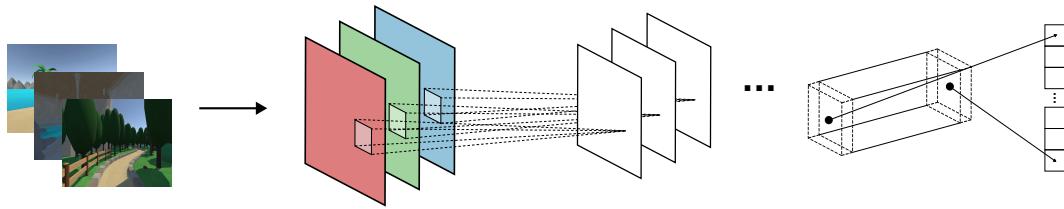
4.2 Architektur

Dieser Abschnitt verschafft einen Überblick über die Architektur des Systems und die Beziehungen zwischen den zentralen Bestandteilen darin. Im Anschluss daran gehen die folgenden Abschnitte detaillierter auf die Umsetzung der einzelnen Komponenten ein.

Wie in der Zielsetzung bereits beschrieben gibt es zwei zentrale Komponenten innerhalb des Systems. Dabei handelt es sich einerseits um das Unity-Projekt, mithilfe dessen das Labyrinth simuliert und der Agent gesteuert wird. Außerdem ist diese Komponente, im Folgenden auch als Frontend bezeichnet, dafür zuständig, die Inputdaten für die Place Units zu sammeln. Diese gesammelten Daten werden in der zweiten Komponente, dem Python-Backend, weiter verarbeitet. Dabei werden zunächst die Features mithilfe eines vorge trainierten CNN aus den Bildern extrahiert, damit diese daraufhin von einer SOM, welche die Aktivierung der Place Units simuliert, als Inputvektor verwendet werden können. Der Ablauf innerhalb des Backends ist in Abbildung 4.1 grafisch dargestellt.

Auch im Ablauf kann das System in zwei Abschnitte unterteilt werden. Dabei handelt es sich einerseits um die Trainingsphase und andererseits um die Inferenzphase. Zu Beginn der Trainingsphase ist die SOM im Backend frisch initialisiert und komplett untrainiert. Während dieser Phase trainiert das System die Kohonen-Karte also auf Inputdaten aus dem Frontend. Dabei ist es allerdings nicht notwendig, dass die Daten zeitgleich zum Lernprozess der SOM gesammelt werden. Es ist von Vorteil stattdessen vorher einen Datensatz mithilfe des Frontends anzulegen, welcher daraufhin im Backend für Trainingszwecke verwendet werden kann. Üblicherweise verwendet man zusätzlich einen Testdatensatz, um nach Abschluss der Trainingsphase überprüfen zu können, ob das Training erfolgreich war. Ist die Trainingsphase abgeschlossen, so kann das System zur Gewinnung von neuen Informationen verwendet werden. Dieser Prozess wird in Bezug auf Systeme maschinellen Lernens oft als Inferenzieren bezeichnet. Das Ziel ist, dass das System zur Laufzeit diejenige Place Unit ermitteln kann, die durch den aktuell im Frontend betrachteten Ausschnitt seiner Umgebung aktiviert wird, während der Agent sich durch das La-

CNN



SOM

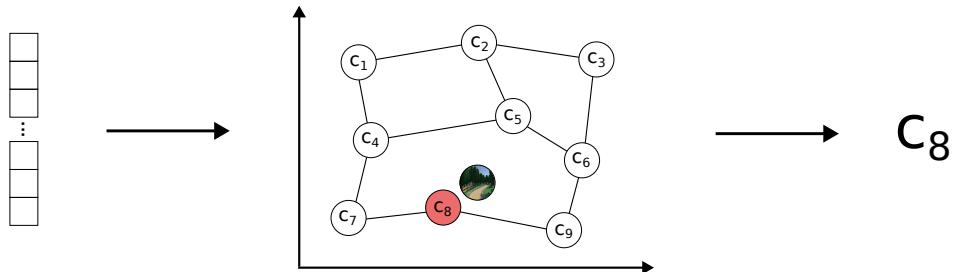


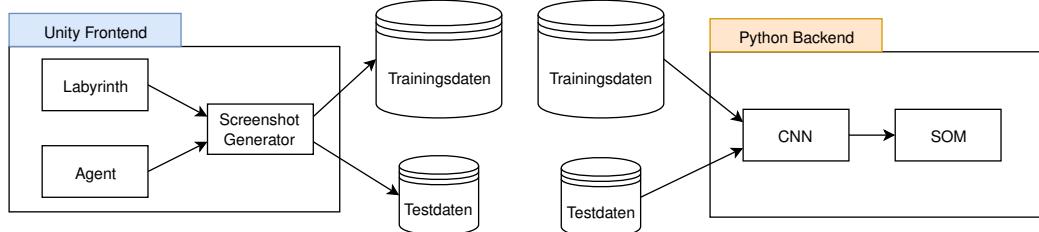
Abbildung 4.1: Vereinfachte Darstellung des Ablaufes innerhalb des Backends. Die aus der Simulation gewonnenen Bilddaten werden zunächst jeweils einzeln als Input in das vortrainierte CNN gespeist. Hier werden durch Faltungsoperationen über eine Vielzahl von Schichten hinweg Merkmale aus dem Bild extrahiert und in Form eines Vektors ausgegeben. Dabei handelt es sich um den Merkmalsvektor, welcher bei einem nicht modifizierten CNN den Input für die Fully-Connected-Schichten darstellen würde. Stattdessen wird dieser allerdings in die SOM gegeben, wo er als Eingangspunkt im hochdimensionalen Merkmalsraum interpretiert wird. Wie bei der Funktionsweise einer SOM üblich, wird im letzten Schritt derjenige Prototyp ermittelt, der den geringsten Abstand zu dem Punkt aufweist, welcher das Eingangsbild beschreibt. Je nach aktuell betrachteter Phase des Systems wird dieses Ergebnis entweder für den Lernprozess verwendet oder stellt den Output des Backends in Form der aktivierten Place Cell dar.

byrinh bewegt. Dazu ist es notwendig, dass Backend und Frontend gleichzeitig aktiv sind und in regelmäßigen Abständen miteinander Daten austauschen können. Da es keine Möglichkeit gibt innerhalb von Unity zur Laufzeit des Frontends Python Code auszuführen, muss eine Schnittstelle zwischen den beiden Subsystemen geschaffen werden. Dazu wird ein lokaler Server als Web Socket mit Python angelegt, dessen Aufgabe es ist, Anfragen des Frontend-Clients, welche den aktuell wahrgenommenen Bildausschnitt enthalten, an das Backend weiterzugeben. Nachdem das Backend die Daten verarbeitet und die aktivierte Place Unit bestimmt hat, wird diese Information in Form der Server Response an das Frontend zurückgegeben. Auf diese Weise kann das Frontend die aktuell aktive Place Unit zur Laufzeit dynamisch visualisieren. Für die Implementierung des Web Sockets wird die Bibliothek ZeroMQ⁵ verwendet, welche sowohl Client- als auch Server-Funktionalitäten in verschiedenen Programmiersprachen bereitstellt. Darunter befinden sich

⁵<https://zeromq.org>

unter anderem auch Python und C#⁶, die Programmiersprache, welche Unity für Skripte verwendet. Die Beziehungen zwischen Frontend und Backend sowie deren Teilkomponenten in den beiden Phasen sind in Abbildung 4.2 dargestellt.

Trainingsphase



Inferenzphase

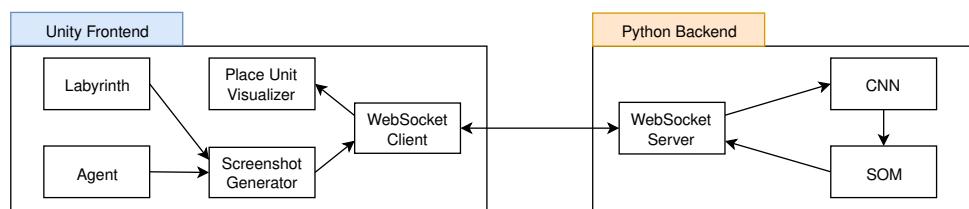


Abbildung 4.2: Architekturdiagramm für das in dieser Arbeit beschriebene System mit Unterteilung in die zwei vorgesehenen Abläufe Training und Inferenzieren. Die Trainingsphase beginnt damit, dass das Frontend einen Trainings- und Testdatensatz für den Lernprozess der SOM im Backend zur Verfügung stellt. Dafür generiert der Screenshot Generator eine Vielzahl an Bildern der Simulation des Labyrinths aus Perspektive des Agenten. Diese Bilder werden dann in zwei disjunkte Datensätze, die Trainings- und die Testdaten eingeteilt, wobei die Trainingsdaten üblicherweise deutlich mehr Bilder enthalten. Das Backend kann nun zu einem beliebigen Zeitpunkt danach auf die generierten Daten zugreifen, um diese zunächst mithilfe des vortrainierten CNN in Featurevektoren umzuwandeln. Die Featurevektoren dienen also als Trainingsinputs, mithilfe welcher die Prototypen der SOM angepasst werden können, bis die Testdaten zufriedenstellende Ergebnisse liefern und die Trainingsphase beendet werden kann. Während der Inferenzphase laufen beide Teilsysteme gleichzeitig, um eine Aktivierung der Place Units in der SOM zur Laufzeit des Frontends zu ermöglichen. Dabei wird auf die gleiche Weise in regelmäßigen Zeitabständen ein Bild aus der Perspektive des Agenten generiert, nun aber über den Web Socket Client mithilfe von TCP zum lokalen Web Socket Server des Backends übertragen. Dort wird wieder zunächst der Featurevektor des Bildes mithilfe des CNN extrahiert und daraus durch die SOM die zu aktivierende Place Unit ermittelt. Die Information über die Aktivierung der Place Units wird dann an den Web Socket Server übergeben, welcher diese dem Client als Antwort auf seine Anfrage zurücksendet. Nun kann die aktivierte Place Unit durch den Place Unit Visualizer im Frontend angezeigt werden.

4.3 Unity-Frontend

Beim Frontend handelt es sich um ein mit der Unity Game Engine entworfenes Subsystem, welches ein Labryinth im Sinne der in Abschnitt 4.1 beschriebenen Zielsetzung simuliert. Dabei kann der Benutzer einen Agenten in Egoperspektive

⁶<https://learn.microsoft.com/de-de/dotnet/csharp/tour-of-csharp/>

entlang eines vorgeschriebenen Pfades durch die Umgebung steuern. Während der Simulation wird in der Inferenzphase außerdem die bei der Betrachtung des aktuell dargestellten Ausschnitts der Umgebung aktivierte Place Unit angezeigt. In diesem Abschnitt werden die zugrundeliegenden Konzepte erläutert, welche im Frontend Verwendung finden. Davor wird allerdings zunächst ein grundlegender Einblick in die relevanten Funktionen von Unity gegeben.

4.3.1 Unity Game Engine

Unity ist eine Game Engine zur Entwicklung von 2D- und 3D-Spielen und gleichzeitig die meistverwendete Echtzeit-3D-Grafik-Engine. Dabei überzeugt sie besonders durch einfache Bedienbarkeit und hohe Flexibilität, weshalb sie für dieses Projekt anstelle der ebenfalls verbreiteten Unreal Engine⁷ verwendet wird.

Unitys grafische Oberfläche ermöglicht es, spielbare Umgebungen, sogenannte Szenen, zu erstellen und nach Belieben anzupassen. Eine Szene besteht dabei immer aus einer Menge von Game Objects, welche vergleichbar mit Objekten der reellen Welt sind. Game Objects können zum Beispiel der Spieler, aber auch die Kamera oder Teile des Levels sein. Ein Game Object hat diverse vorbestimmte Komponenten mit Eigenschaften, die vom Entwickler angepasst werden können, um das gewünschte Verhalten zu realisieren. Zusätzlich können eigene Komponenten in Form von Skripts programmiert werden, um individuelle Anforderungen zu erfüllen. Entwickelt werden diese Skripts dabei mit C#, einer typsicheren und objektorientierten Programmiersprache von Microsoft. Unity bietet mit dem Unity Asset Store zusätzlich die Möglichkeit, diverse von anderen Entwicklern und Designern angebotene Modelle, Game Objects, Skripte und Plug-ins in den eigenen Spielen zu verwenden.

4.3.2 Simulation

Für die Umsetzung der Simulation des Labyrinths ist zunächst einmal wichtig, welche Form von Labyrinth verwendet werden soll. Dabei können verschiedene Design-Entscheidungen getroffen werden. Im Vordergrund steht dabei die gewünschte Komplexität des Labyrinths. Das im als Vorbild dienenden Versuch verwendete Labyrinth besteht lediglich aus einem linearen, erhöhten Laufsteg, ohne besonde-

⁷<https://www.unrealengine.com/de>

re Reizeeinflüsse [30]. Für diesen Versuch ist es allerdings von Vorteil, komplexere visuelle Reize zu schaffen, da die Place Units sich hier lediglich auf das wahrgenommene Bild stützen. Da das Ziel ist, dass die Place Units jeweils unterschiedliche Bereiche abdecken, ist es also notwendig eine hohe visuelle Varianz zwischen den Bereichen zu schaffen, sodass diese lediglich durch eine Momentaufnahme aus Blickwinkel des Agenten voneinander unterschieden werden können.

Für diesen Versuch wird als Labyrinth eine 3D-modellierte Insel verwendet, über welche ein linearer Rundweg durch vier visuell deutlich unterscheidbare Bereiche führt. Die Bereiche sind dabei einem Wald, einem Strand, einer Winterlandschaft und einem Höhlensystem nachempfunden. Die Insel wird mithilfe des modularen Assetpacks „Modular Terrain 2.0“⁸ und des dazugehörigen „MAST“-Plug-Ins⁹ für Unity von Fertile Soil Productions erstellt. Abbildung 4.3 zeigt das auf diese Weise entstandene Labyrinth.



Abbildung 4.3: Ansicht des Labyrinths in Form einer Insel aus der Vogelperspektive. Drei der vier Gebiete; der Strand, der Wald und die Winterlandschaft; sind deutlich differenzierbar.

Beim Agenten handelt es sich um ein Game Object, welches als Elternobjekt für die Hauptkamera dient. Auf diese Weise bewegt sich die Kamera gemeinsam mit dem Agenten, nimmt also zu jedem Zeitpunkt seine Perspektive auf. Der Agent hat die Möglichkeit sich entlang eines vordefinierten Pfades vorwärts oder rückwärts zu bewegen und sich entlang seiner y-Achse zu drehen. Der Pfad wird dabei mithilfe des „Bézier Path Creator“¹⁰ Tools von Sebastian Lague manuell definiert. Dieses

⁸<https://fertile-soil-productions.itch.io/modular-terrain-2>

⁹<https://www.fertilesoilproductions.com/mast>

¹⁰<https://assetstore.unity.com/packages/tools/utilities/b-zier-path-creator-136082>

Tool ermöglicht außerdem die Bewegung des Agenten entlang des Pfads, indem ein Skript lediglich vor jedem Frame die Position des Agenten entlang des Pfades berechnet und diese vom Tool in die korrekten Weltkoordinaten umrechnen lässt.

4.3.3 Bildgenerierung

Die Logik zur Erzeugung von Bilddaten im Frontend ist in zwei wesentliche Funktionalitäten unterteilt. Einerseits kann der ScreenshotGenerator auf Knopfdruck eine beliebige Menge an Bildern für den Trainings- und Testprozess des Backends generieren und diese im entsprechenden Ordner speichern. Dazu muss zunächst angegeben werden, wie viele Bilder erzeugt werden sollen. Gute Ergebnisse werden zum Beispiel mit 3000 Trainings- und 1000 Testbildern erzielt. Um die Bilder zu erzeugen wird der Agent in jeder Iteration automatisch an eine zufällige Position entlang des Pfades mit einer zufälligen Rotation zwischen 0 und 360 Grad positioniert. Im Anschluss wird die Sicht der Hauptkamera in ein Bild gerendert und das Resultat im passenden Ordner gespeichert. Dieser Vorgang wird so lange wiederholt, bis die gewünschte Anzahl an Bildern erreicht worden ist. Des Weiteren kann das System in der Inferenzphase Bilder mithilfe des SomManagers generieren, um diese dann zur Laufzeit durch das Backend kategorisieren zu lassen. Dafür wird alle 0,2 Sekunden die aktuelle Sicht der Hauptkamera gerendert und mithilfe des Web Socket Clients an den Server übertragen. Das Ergebnis wird daraufhin durch den SomManager in der grafischen Benutzeroberfläche des Frontends dargestellt. Abbildung 4.4 zeigt die Simulation aus Perspektive des Agenten sowie die Neuroenkarke mit der aktuell aktiven Place Cell.

4.4 Websocket als Schnittstelle

Wie in Abschnitt 4.2 beschrieben, besteht das System in einer High-Level-Ansicht aus einer Frontend- und einer Backend-Komponente. Da das Frontend in Unity betrieben wird, welches keine Möglichkeit bietet, zur Laufzeit der Simulation Python Code auszuführen, müssen die beiden Komponenten als eigenständige Subsysteme realisiert werden. Um in der Inferenzphase trotzdem eine Kommunikation zwischen den beiden Komponenten zur Laufzeit zu ermöglichen, muss eine entsprechende Infrastruktur geschaffen werden. In diesem Fall wurde dafür ein simples Client-



Abbildung 4.4: Simulation des Labyrinths aus Perspektive des Agenten, aus welcher auch die Bilddaten für Training und Inferenz generiert werden. Die linke obere Ecke zeigt die Neuronenkarte, welche auf der im Backend laufenden SOM basiert. Das hervorgehobene Neuron stellt die aktuell aktive Place Cell dar.

Server-Prinzip gewählt, welches mithilfe des Web-Socket-Protokolls realisiert wird. Um eine Web-Socket-Lösung innerhalb eines verteilten Systems mit verschiedenen Programmiersprachen zu implementieren, wird hier die Open-Source-Bibliothek ZeroMQ verwendet. Diese liefert Client- und Server-Funktionalitäten für diverse Programmiersprachen. Auf diese Weise kann im Frontend ein C#-Client erzeugt werden, welcher den Python-Server im Backend ansprechen kann. Die Folgenden beiden Unterkapitel beschreiben die Implementierung des Clients und des Servers innerhalb ihrer jeweiligen Komponente.

4.4.1 Client im Frontend

Um einen Web-Socket-Client in C# zu entwickeln, stellt ZeroMQ die Klasse *RequestSocket* bereit. Diese beinhaltet diverse Methoden, um Serveraufrufe zu ermöglichen. Der grundlegende Ablauf, wie er in der Dokumentation von ZeroMQ¹¹ zu finden ist, ist dabei in Listing 4.1 dargestellt. Hier wird ein neuer RequestSocket erstellt und mit dem lokal laufenden Server verbunden, an welchen im Anschluss eine Nachricht geschickt wird. Die Antwort des Servers wird daraufhin empfangen und auf der Konsole ausgegeben.

```
using (var client = new RequestSocket())
```

¹¹<https://zeromq.org/languages/csharp/>

```
{
    client.Connect("tcp://127.0.0.1:5556");
    client.SendFrame("Hello");
    var msg = client.ReceiveFrameString();
    Console.WriteLine("From Server: {0}", msg);
}
```

Listing 4.1: WebSocket Client

Um einen solchen Client im Projekt verwenden zu können, werden diverse Anpassungen getroffen. Die Client-Funktionalität befindet sich in der Klasse *BackendRequester*, welche die Klasse *RunnableThread* erweitert. Auf diese Weise kann der Client auf die Antwort des Servers warten, während die Simulation parallel weiterläuft, ohne stoppen zu müssen.

Wie bereits im Abschnitt 4.3 beschrieben generiert die Klasse *SomManager* zur Laufzeit die Bilder, die vom Backend verarbeitet werden sollen. Dafür wird jedes dieser Bilder im *BackendRequester* in ein Byte-Array umgewandelt, welches dann mithilfe der Methode *SendFrame* des erstellten *RequestSocket*-Objektes an das Backend gesendet werden kann. Bei Antwort des Backends wird ein Listener benachrichtigt, welcher die übergebene, aktivierte Place Unit durch den *SomManager* visualisieren lässt. Da die genaue Umsetzung der Schnittstelle für diese Arbeit eher unerheblich ist, wird für das tiefere Verständnis auf die zur Verfügung gestellte Codebasis verwiesen.

4.4.2 Server im Backend

Auch der Web-Socket-Server wird mithilfe von ZeroMQ realisiert. Dafür werden zunächst die notwendigen Backendressourcen, also das verwendete CNN sowie die trainierte SOM geladen, um diese für die Inferenz zu verwenden. Im Anschluss daran wird der Server initialisiert und gestartet. Da sowohl Frontend als auch Backend auf derselben Maschine laufen können die Anfragen über die localhost-Adresse gesendet werden, welche dementsprechend vom Server abgehört werden muss.

Nachdem der Initialisierung des Web Sockets beginnt eine Endlosschleife in welcher der Server Anfragen des Clients entgegennimmt und diese entsprechend der zuvor beschriebenen Inferenzphase verarbeitet. Dabei werden drei zentrale Schritte durchlaufen. Wurde eine Anfrage des Clients erhalten, so wird diese zunächst aus dem gesendeten Byte-Buffer in das zu verarbeitende Bild umgewandelt. Im Anschluss daran wird das Bild als Input für das modifizierte CNN verwendet, um den

Featurevektor zu extrahieren, welcher im Anschluss mithilfe der SOM auf die ihm ähnlichste Place Unit abgebildet wird. Die ermittelte BMU in Form ihrer Indizes in der Kohonen-Karte wird nun in ein Byte-Array umgewandelt, welches daraufhin als Antwort zurück an den Client gesendet werden kann. Listing 4.2 zeigt den für die Implementierung dieser Funktionalität verwendeten Code. Die Implementierung und Funktionsweise der erwähnten Backend-Komponenten wird im folgenden Abschnitt ausführlich behandelt.

```
#initialize nnet
model, im_size = cnn.mobilenet()

# initialize som
som = sm.load('som_trained.npy')

# initialize socket
context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

while True:
    bytes_received = socket.recv()
    img = img_from_buff(bytes_received)
    img = prep_img(img)
    prediction = model.predict(img) # feature extraction
    features = prediction.reshape((1,1024))
    bmus, _ = sm.test(som, features) # inferencing on SOM using features
    byte_ans = bmus.tobytes()
    socket.send(byte_ans)
```

Listing 4.2: WebSocket Server

4.5 Python-Backend

Dieser Abschnitt befasst sich mit dem Design und der Umsetzung der Funktionen des maschinellen Lernens, welche die Funktionalität des Backends realisieren. Dieses besteht aus zwei zentralen Bestandteilen, dem vortrainierten und modifizierten Convolutional Neural Network und der Self-Organizing Map. Als Programmiersprache wird Python gewählt, welche in Bereichen des maschinellen Lernens weit verbreitet ist und daher viele nützliche Funktionalitäten und Unterstützung für die Implementierung Systeme dieser Art bietet. Bei der Implementierung des CNN wird das Deep-Learning-Framework TensorFlow¹² in Verbindung mit Keras verwendet. Die SOM wird ohne Verwendung von externen Bibliotheken, mit Aus-

¹²<https://www.tensorflow.org>

nahme von NumPy¹³ welches lediglich zur Optimierung der Rechenschritte dient, implementiert.

Die beiden Komponenten werden nach einer kurzen Einführung in TensorFlow und Keras jeweils genauer erläutert.

4.5.1 TensorFlow und Keras

Bei TensorFlow und Keras handelt es sich um zwei Bibliotheken zur Entwicklung von Anwendungen des maschinellen Lernens mit verschiedenen Schwerpunkten. TensorFlow arbeitete dabei lange Zeit auf einem niedrigen Level und hatte in erster Linie das Ziel, für maschinelles Lernen notwendige komplexe Operationen möglichst effizient auf der Hardware durchzuführen. Keras hingegen liefert eine High-Level-API um damit möglichst schnell und einfach neuronale Netze zu implementieren. Als solche bietet Keras auch die Möglichkeit, vorgebildete Modelle zu verwenden und auf unkomplizierte Weise an die umzusetzenden Anforderungen anzupassen. In den aktuellen Versionen von TensorFlow 2 ist Keras aus diesem Grund eingebunden, was die Vorteile beider Bibliotheken vereint. Auf diese Weise ist es möglich, skalierbare Modelle schnell und einfach zu implementieren und dabei zum Beispiel die Möglichkeit zu haben Berechnungen auf die GPU zu verlagern oder Anwendungen für Mobilgeräte oder Browser zu exportieren.

4.5.2 CNN

Die Aufgabe des CNN ist es, aus den Bildern, welche zum Training bzw. zum Inferenzieren verwendet werden sollen, für die SOM verwendbare Featurevektoren zu extrahieren. Alternativ wäre es auch möglich, die Bilder direkt als Inputvektoren für die SOM zu verwenden. Ein Nachteil dieses Ansatzes wäre allerdings, dass bei der verwendeten Bildgröße von 224x224 Pixeln jeder Featurevektor aus $224 * 224 * 3 = 150528$ Einträgen bestehen würde, da jeder Farbwert jedes Pixels durch einen Zahlenwert dargestellt wird. Mithilfe eines CNN kann dieser Featurevektor auf lediglich 1024 Werte verkleinert und bei Bedarf sogar noch weiter komprimiert werden, was deutliche Vorteile in der Performance des Systems zur Folge hat. Des Weiteren hat die Art der Inputdaten Einfluss auf das Verhalten der SOM.

¹³<https://numpy.org>

Wie in Abschnitt 2.4 beschrieben, zielt diese nämlich darauf ab den Abstand zwischen den Inputvektoren und dem jeweils nächsten Referenzvektor zu minimieren. Das bedeutet, dass die Bilder innerhalb eines durch einen Referenzvektor abgebildeten Clusters lediglich ähnliche Farbwerte an denselben Pixelpositionen aufweisen. Stattdessen ist das Ziel, die im Bild dargestellten Objekte und Muster so gut wie möglich in dem Featurevektor zu kodieren. Je nach dargestelltem Bildinhalt ist dabei die Position sowie die Farbe der Objekte möglicherweise komplett irrelevant. Das Waldgebiet zeichnet sich beispielsweise vorwiegend durch das Vorhandensein von Laubbäumen aus. Dabei ist es allerdings egal, ob die Bäume links oder rechts im Bild zu sehen sind.

Welche Features ein CNN aus einem Bild extrahiert hängt in erster Linie von dessen Architektur sowie den für das Training verwendeten Bildern ab. Abbildung 2.4 zeigt den typischen Aufbau eines simplen CNN. Die Schichten des Feature Extraction Network finden dabei Muster in dem Eingangsbild und reduzieren sie auf einen Vektor, welcher diese sogenannten Features in Form von Zahlenwerten beinhaltet und daraufhin vom Classifier Network für die Klassifizierung verwendet wird. Der Ansatz, welcher in dieser Arbeit verfolgt wird, ist, diesen Featurevektor, welcher bei der Feature Extraction erzeugt wird, als Input für die SOM zu verwenden. Dadurch, dass die extrahierten Merkmale durch mehrere, aufeinander aufbauende Faltungsschichten bestimmt werden, kann es sein, dass der entstandene Vektor sehr komplexe Features beinhaltet, die nicht mehr vom Menschen nachvollzogen werden können [20]. Da die dafür zuständigen Faltungsmasken im Trainingsprozess so angepasst werden, dass der Fehler in der Kategorisierung der Trainingsdaten minimiert wird, werden bei erfolgreichem Training eben jene Features extrahiert, die für die Klassifizierung dieser Bilder nützlich sind. Bei der Wahl eines vortrainierten CNN ist also wichtig, dass die im Featurevektor kodierten Merkmale für den vorliegenden Anwendungsfall relevant sind.

Keras bietet eine Auswahl von vortrainierten CNN-Modellen¹⁴, mit verschiedenen Vor- und Nachteilen. Durch den simplen Stil der zur Modellierung des Labyrinths verwendeten 3D-Modelle und die deutliche Unterscheidbarkeit der dargestellten Gebiete wird keine besonders aufwendige Architektur benötigt. Um eine möglichst zeitnahe Rückmeldung des Backends zur Laufzeit des Frontends zu erreichen, sollte zudem die Laufzeit des Backends möglichst optimiert werden. Aus diesen beiden

¹⁴<https://keras.io/api/applications/>

Gründen wird in dieser Arbeit die in [31] beschriebene MobileNet-Architektur¹⁵ verwendet, welche in erster Linie für mobile Anwendung und Embedded Systems konzipiert wurde und somit deutliche Vorteile in Performance und Größe gegenüber anderen verfügbaren Modellen aufweist. Die simple Architektur dieses Modells ermöglicht zudem eine unproblematische Anpassung des Netzes auf die Feature-Extraction-Schicht, indem lediglich das Classifier-Netz entfernt wird. Die von Keras zur Verfügung gestellte voreingestellte Variante des Netzwerkes wurde mit Bildern aus dem etablierten ImageNet-Datensatz [27] trainiert, welches Objekte aus diversen unterschiedlichen Kategorien beinhaltet und auf welchem die Architektur eine Accuracy von 70,4 % in der Objekterkennung erreichen konnte. Es wurde bereits gezeigt, dass auf ImageNet trainierte CNNs auch erfolgreich für Anwendungen in anderen Gebieten verwendet werden und dort zufriedenstellende Ergebnisse erzielen können [26]. Aus diesen Gründen ist davon auszugehen, dass die erkannten Muster und der daraus resultierende Featurevektor für diese Anwendung geeignet ist. Um sicherzugehen, wird überprüft, ob das gesamte Netz inklusive Fully-Connected-Schicht für Bilder aus manuell bestimmten Kategorien unterschiedliche Ausgaben liefert. Die in Abbildung 4.5 dargestellten Ergebnisse lassen darauf schließen, dass die Feature-Extraction-Schichten für Bilder von verschiedenen Gebieten zur Genüge unterscheidbare Featurevektoren liefern.

Das CNN erfüllt in der Trainings- sowie in der Inferenzphase denselben Zweck. Die zu verarbeitenden Bilder werden lediglich als Input übergeben und der von der Feature-Extraction-Schicht generierte Output von der SOM weiterverwendet. Dazu müssen die Bilder allerdings für die Verwendung mit MobileNet angepasst werden. Keras bietet dafür die Methode *preprocess_input* aus dem MobileNet-Package, die zuvor auf jedes der zu verwendenden Bilder aufgerufen werden sollte. Dabei werden die Bildpixel auf Werte zwischen -1 und 1 normalisiert, um dem von MobileNet erwarteten Format zu entsprechen. Des Weiteren werden die Bilder auf die von der Inputschicht des Netzwerkes erwartete Größe von 224x224 Pixeln skaliert. Standardmäßig geben Keras-Modelle bei der Vorhersage nur den Output der letzten Schicht des Netzes, also in diesem Fall des Classifier-Netzes, zurück. Um stattdessen den bei der Feature Extraction ermittelten Merkmalsvektor zu erhalten, muss das Modell angepasst werden. Hierzu wird ein neues Modell angelegt, welches als Output die Ausgabe der letzten Schicht des Feature-Extraction-Netzes liefert. Um diese Schicht zu ermitteln, kann die Methode *summary* verwendet werden, welche

¹⁵<https://keras.io/api/applications/mobilenet/>

	seashore 0.2490	umbrella 0.0992	lakeside 0.0960
	mosquito_net 0.2564	wing 0.0618	patio 0.0318
	bathing_cap 0.1203	lakeside 0.0477	maze 0.0453
	bobsled 0.1383	cloak 0.1095	missile 0.0620

Abbildung 4.5: Die Top-3 vom vortrainierten MobileNet vorhergesagten Kategorien des ImageNet-Datensatzes für die ausgewählten Kategorien. Jede Kategorie beinhaltet dabei jeweils 5 Bilder, welche manuell nach den vier Gebieten im Labyrinth bestimmt werden. Die erste Spalte der Tabelle zeigt dabei einen Repräsentanten der Bilder aus der Kategorie, während die rechte Spalte die drei mit durchschnittlich höchster Sicherheit vorhergesagten Kategorien mit jeweils dem zugehörigen durchschnittlichen Confidence-Score zeigt. Es sind deutliche Unterschiede in den ermittelten Labels zwischen den manuell gewählten Kategorien erkennbar. Dieses Ergebnis lässt vermuten, dass der für die Kategorisierung im Classifier-Netzwerk des CNN zugrundeliegende Merkmalsvektor zwischen den Gebieten genug differenzierbar ist, sodass die darauf zu trainierende SOM sinnvolle Cluster finden kann.

einen Überblick über die Schichten eines Netzes liefert. Das Modul *cnn.py* im Codeverzeichnis dieser Arbeit liefert Methoden um sowohl eine vollständige, als auch eine entsprechend angepasste Instanz von MobileNet zu erhalten.

4.5.3 SOM

Dieser Abschnitt beschäftigt sich mit der Implementierung der SOM im Backend. Dabei wird zwischen der Trainings- und der Inferenzphase unterschieden, da das Netz sich je nach Phase unterschiedlich verhält.

Eine SOM besteht aus zwei typischerweise mehrdimensionalen Räumen. Einer davon ist die Karte, in der die Nachbarschaften der Neuronen festgelegt sind, der andere derjenige, in welchem sich die Vektoren der Neuronen bewegen. Die Neuroenkarte wird zweidimensional mit einer Größe von 3x3 und einer Vierernachbarschaft angelegt. Wir erhalten also 9 Place Units, die zwischen 2 und 4 Nachbarn, abhängig von ihrer Position auf der Karte, besitzen. Die entstandene Karte entspricht der Darstellung im linken Teil von Abbildung 2.10. Die Dimensionalität der Referenzvektoren der Neuronen entspricht der Dimensionalität der Inputvektoren. Diese ergibt sich aus der Größe des Outputs der Feature-Schicht des CNN, welcher

1024 Werte beinhaltet. Bei der Initialisierung der Vektoren werden diese zunächst mit zufälligen Werten zwischen 0 und 2 gefüllt.

Damit die aktivierte Place Units in der Inferenzphase ermittelt werden können, muss diese zufällig initialisierte SOM zunächst trainiert werden. Da das CNN bereits vorgenommen ist und es sich bei einer SOM um ein unüberwachtes Lernverfahren handelt, benötigt das System in der Trainingsphase lediglich Bilder und keine entsprechenden Labels. Diejenige Place Unit, welche bei einem konkreten Inputbild feuert, wird also beim Lernen vom System selbstständig ermittelt, indem die Referenzvektoren so angepasst werden, dass sie Cluster in den Eingangsdaten möglichst gut abdecken. Als Inputdaten wird dabei das im Frontend generierte Trainingsdatenset verwendet, welches 3000 an zufälligen validen Positionen des Agenten im Labyrinth generierte Bilder beinhaltet. Für jedes dieser Bilder, welche in zufälliger Reihenfolge ausgewählt werden, läuft dann der folgende Prozess. Zunächst wird das Bild als Input für das modifizierte CNN verwendet, um den Featurevektor zu generieren, der das Bild beschreibt. Dieser Featurevektor wird dann als Input für das Lernverfahren der SOM verwendet, welches in Abschnitt 2.4 beschrieben ist. Dabei wird die euklidische Distanz zwischen dem Inputvektor und dem Referenzvektor jeder Place Unit berechnet und das Neuron mit der geringsten Distanz zum Input in diesem Lernschritt als BMU bestimmt. In Python kann die euklidische Distanz zwischen zwei Punkten berechnet werden, indem die Funktion `numpy.linalg.norm` der Bibliothek NumPy auf die Differenz ihrer Ortsvektoren angewandt wird. Die ermittelte BMU und ihre im betrachteten Umkreis liegenden Nachbarn werden daraufhin ein Stück in Richtung des Eingangsvektors bewegt. Der Radius für die betrachtete Nachbarschaft N_c um die BMU wird als $r = 1$ gewählt, es werden also nur direkte Nachbarn von m_c bewegt. Der Anpassungsgrad für sowohl die BMU als auch die Nachbarneuronen verringert sich im Laufe des Lernprozesses. Dies erlaubt eine höhere Lernrate in den frühen Lernschritten, was zur Folge hat, dass die Karte sich schneller entfaltet und die Neuronen sich stärker voneinander abgrenzen. Gleichzeitig werden die Neuronen aber in den späteren Lernschritten, in denen die Cluster weitestgehend erlernt wurden, nicht mehr so stark angepasst, sodass feinere Anpassungen vorgenommen werden können. Der optimale Wert für die Anpassungsrate wird experimentell ermittelt und beläuft sich für m_c auf $\alpha(t) = 0,02 * (2 - t/t_{max})$ und für die Neuronen aus N_c auf $\alpha(t) = 0,01 * (1 - t/t_{max})$, wobei t_{max} der Anzahl an Lernschritten insgesamt entspricht. Die Nachbarneuronen der BMU werden also in den späteren Lernschritten fast gar nicht mehr bewegt, um nicht zum Ende

des Lernprozesses noch ruckartig in Richtung eines Nachbarn von ihren Clustern entfernt zu werden. Für die Änderung der Referenzvektoren ergibt sich folgende Gleichung:

$$m_i(t+1) = \begin{cases} m_i(t) + 0,02 * (2 - t/t_{max}) * (x(t) - m_i(t)), & \text{falls } m_i = m_c \\ m_i(t) + 0,01 * (1 - t/t_{max}) * (x(t) - m_i(t)), & \text{falls } m_i \in N_c \\ m_i(t), & \text{falls } m_i \notin N_c \end{cases}$$

Um den Trainingserfolg zu bestimmen wird dem System im Anschluss das generierte Testdatenset mit 1000 Bildern übergeben. Dieses durchläuft denselben Prozess wie das Trainingsdatenset, allerdings werden die Gewichtsvektoren in der SOM im Testprozess nicht mehr angepasst. Stattdessen wird lediglich die BMU für jedes Bild bestimmt. Um das Ergebnis zu veranschaulichen wird jedes Bild in einem dem aktivierte Neuron entsprechenden Ordner abgespeichert. Da es sich um einen unüberwachten Lernprozess handelt, ist der Erfolg des Trainings schwierig zu messen. Eine Möglichkeit ist die manuelle Überprüfung der kategorisierten Bilder darauf, ob die gefundenen Kategorien sinnvoll und nachvollziehbar sind. Mehr zu möglichen Ansätzen zur Bestimmung des Lernerfolges findet sich in Kapitel 5.

Sind die Testergebnisse zufriedenstellend, so kann das System nun in der Inferenzphase verwendet werden. Dabei wird zu jedem Zeitpunkt jeweils nur genau ein Bild vom Frontend-Client übermittelt und vom CNN verarbeitet. Das Neuron der SOM mit dem geringsten euklidischen Abstand zum Featurevektor des Eingangs, also die BMU, stellt die aktivierte Place Unit dar. Diese wird nun zurück an das Frontend übermittelt, um dort visualisiert zu werden.

Jegliche beschriebenen Funktionalitäten zum Betrieb der SOM finden sich in der Codebasis dieser Arbeit im Modul *som.py*. Dieses liefert Methoden zum Anlegen einer SOM, Trainieren dieser auf einer Menge von Inputvektoren, Durchführen von Test- oder Inferenzschritten und zum Speichern in Form einer *.npy*-Datei sowie Laden einer gespeicherten SOM. Des Weiteren finden sich dort die zwei Skripte *train.py* und *test.py*. Diese stellen ausführbare Dateien dar, die den kompletten Trainings- bzw. Testablauf für das gesamte Backendsystem inklusive CNN realisieren.

Kapitel 5

Ergebnisse

Dieses Kapitel stellt die durch den in Kapitel 4 beschriebenen Projektaufbau erzielten Ergebnisse dar. Diese werden im Anschluss analysiert und diskutiert, um mögliche Schlussfolgerungen zu ziehen.

5.1 Ergebnisse

Die zentrale Forschungsfrage dieser Arbeit behandelt die Replikation von Place Units im Gehirn von Säugetieren. Diese werden als Neuronen einer SOM implementiert, welche auf die aus Bildern der Umgebung des betrachteten Agenten extrahierten Featurevektoren trainiert wurde. Dasjenige Neuron, welches beim Eingeben eines Bildausschnittes aktiviert wird, symbolisiert die Place Unit, welche im Sinne der beschriebenen Zielsetzung feuert. Die Ergebnisse des Projektes basieren also zu großen Teilen auf der Qualität der Ermittlung der Place Units. Um die Ergebnisse also analysieren zu können, muss zunächst ein Qualitätsmaß für den Erfolg der Place-Unit-Aktivierung definiert werden. Da die SOM als einzige lernfähige Komponente auf den durch das CNN ermittelten Featurevektoren lernt, sind die Ergebnisse stark von der Art der ermittelten Features abhängig. Das bedeutet, dass vom Menschen nicht nachvollziehbare Ergebnisse nicht direkt auf eine Fehlfunktion des Systems schließen lassen, da die verwendeten Features möglicherweise sehr abstrakt sind. Es biete sich also an, zunächst die Qualität der Ergebnisse der SOM zu betrachten.

5.1.1 SOM-Fehler

Der Lernalgorithmus einer SOM hat das Ziel, die Menge der Eingangsvektoren möglichst optimal mit den veränderbaren Prototypen abzudecken. Ein Lernerfolg ist also dann zu verzeichnen, wenn der euklidische Abstand der Eingangsvektoren zum jeweils dichtesten Prototypen möglichst gering ist. Abbildung 5.1 zeigt diesen Abstand zur BMU im Verlauf des Trainings der für die Inferenzphase verwendeten SOM. Dabei wird der Fehler, also die Distanz zwischen beiden Vektoren, jeweils über 100 Bilder gemittelt.

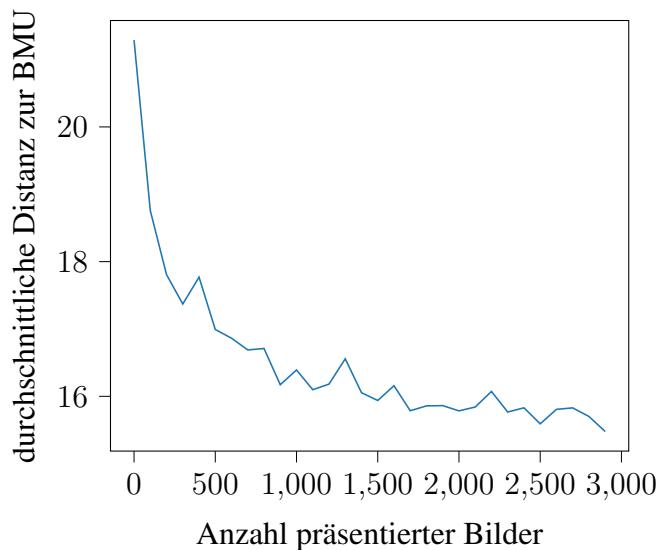


Abbildung 5.1: SOM-Fehler als durchschnittliche Distanz der Eingangsvektoren zu ihrer BMU über jeweils 100 Bilder im Verlauf des Trainingsprozesses der SOM. Obwohl die x-Achse eine diskrete Größe beschreibt, wird für die Darstellung des Graphen zur besseren Visualisierung eine Linie verwendet. Es ist erkennbar, dass der Fehler mit wachsender Anzahl präsentierter Bildern sinkt. Das System lernt also eine Verteilung der Prototypen, die den Raum, in dem sich die Eingangsdaten befinden, besonders gut abdeckt.

Es ist erkennbar, dass der Fehler mit steigender Anzahl präsentierter Bilder immer weiter sinkt, bis er schließlich relativ konstant bleibt. Spontane Anstiege im Fehler während der Trainingsphase lassen nicht unbedingt darauf schließen, dass das System Rückschritte macht. Aufgrund der geringen Größe der für jeden Datenpunkt betrachteten Teilmenge der Eingangsdaten ist eine Varianz im Fehler zu erwarten.

Bei der Auswahl der Lernparameter, wie zum Beispiel der Anpassungsrate α , kann der durchschnittliche Fehler auf der Testmenge betrachtet werden. Ein geringer Fehler spricht dabei für ein Netz, welches die Bilder aus der Testmenge besonders gut abdeckt. Es ist wichtig dabei möglichst mehrere mit einem konkreten Parametersatz trainierte Netze zu testen, da die zufällige Initialisierung der Prototypen in

der Kohonen-Karte beziehungsweise die Reihenfolge der präsentierten Daten der Lernmenge unter Umständen eine ungünstige Anordnung der Neuronen hervorrufen kann, obwohl die gewählten Lernparameter im Schnitt erfolgreich sind. Auf diese Weise werden die in Abschnitt 2.4 beschriebenen Parameter ermittelt.

Der betrachtete Fehler alleine besitzt noch keine Aussagekraft darüber, ob der Lernprozess erfolgreich war. Obwohl der Fehler zu sinken scheint, kann die Vermutung angestellt werden, dass ein Abfall von gerade einmal circa 25 % zwischen der ersten und letzten betrachteten Teilmenge nicht ausreichend ist, um gute Ergebnisse zu erzielen. In anderen Anwendungsfällen kann es sinnvoll sein, weitere Qualitätsmerkmale für die Performance der SOM zu betrachten. Diverse Maße, die dafür in Frage kommen könnten, sind in [32] von Forest et al. zusammengefasst worden. Für diese Arbeit ist dies allerdings nicht notwendig, da der Erfolg des Systems nur bedingt an die Performance der SOM gebunden ist. Auch bei einer optimalen Abdeckung der Eingangsdaten kann es passieren, dass die Place Units keine nachvollziehbaren oder zusammenhängenden Bereiche im Labyrinth beschreiben.

5.1.2 Place Units

Um einen Eindruck dafür zu bekommen, welche Bereiche des Labyrinths letztendlich in der Inferenzphase des Systems von den Place Units beschrieben werden, wird der Agent manuell durch das Labyrinth bewegt und gleichzeitig die Aktivität der Place Units aufgenommen. Durch diesen Prozess entsteht die in Abbildung 5.2 dargestellte Karte der Aktivitäten der Place Units. Bei näherer Betrachtung stellt man fest, dass das Verhalten der Neuronen in vier Bereiche unterteilbar ist, welche jeweils eine charakteristische Aktivität von ein oder zwei in erster Linie dort auftretenden Place Units vorweisen. Da in der in dieser Arbeit beschriebenen Simulation der Place Cells lediglich visuelle Stimuli betrachtet werden, bezeichnen diese Bereiche Gebiete im Labyrinth, die visuelle Ähnlichkeiten aufweisen. Vergleicht man die Neuronen mit der in Abbildung 4.4 dargestellten Nachbarschaft der Zellen in der Kohonen-Karte, so erkennt man, dass Neuronen die innerhalb eines der vier Bereiche feuern, zueinander benachbart sind. Eines der Neuronen, welches vor allem an den Grenzen zwischen zwei Bereichen aktiviert wird, ist das in der Karte zentral gelegene Neuron. Dieses Verhalten ist zu erwarten und ist durch die lokale Abhängigkeit der Neuronen zu ihren Nachbarn im Lernprozess der SOM begründet.

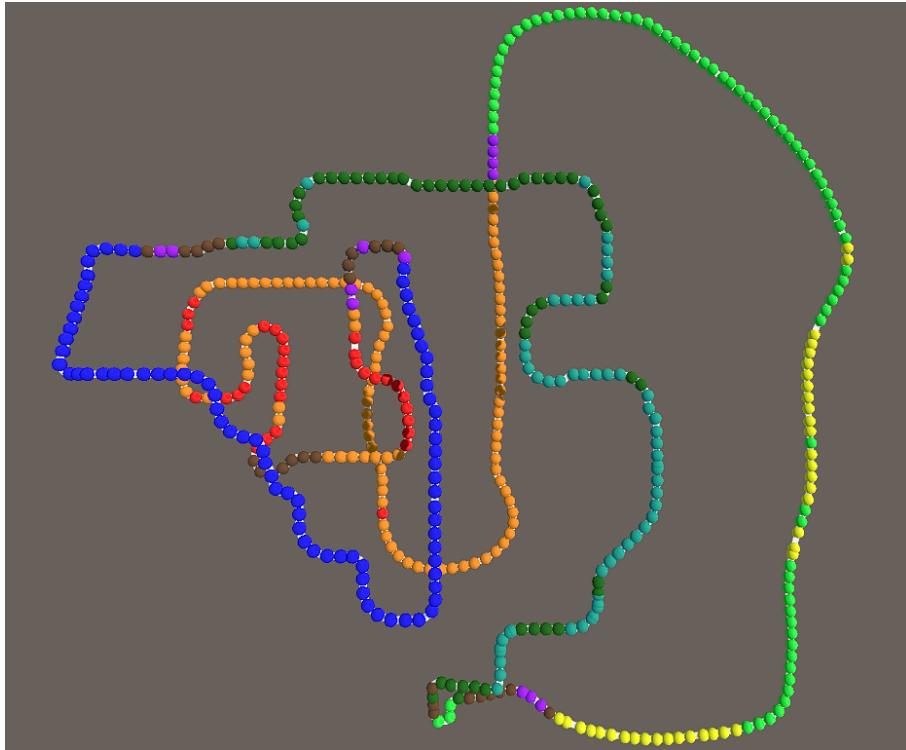


Abbildung 5.2: Repräsentation des Laufpfades im Labyrinth in Top-Down-Perspektive. Die farbigen Kreise entlang des Pfades repräsentieren die Place Units, welche beim manuellen Durchlaufen des Labyrinthes an dieser Stelle feuern. Jede Farbe steht dabei für eine der Place Units. Die Farben korrespondieren zu der in Abbildung 4.4 gewählten Farbgebung der Neuronen in der Kohonen-Karte. Es sind klar vier eindeutig voneinander abgegrenzte Bereiche zu erkennen, in welchen jeweils nur ein bis zwei Place Units feuern. Es ist außerdem erkennbar, dass zwei der Place Units vor allem dann aktiviert werden, wenn der Agent sich am Übergang zwischen zwei dieser Regionen befindet.

Das Projekt basiert im Aufbau auf dem in [30] beschriebenen Experiment, in welchem die Aktivität der Place Cells im Gehirn von Ratten in einem vergleichbaren Labyrinth bestimmt und ausgewertet wird. Vergleicht man Abbildung 5.2 mit den dort beschriebenen Ergebnissen, so erkennt man klare Ähnlichkeiten im Verhalten der Neuronen. In beiden Versuchen werden die Place Cells jeweils überwiegend in charakteristischen Bereichen, im Laborexperiment als Place Fields bezeichnet, aktiviert. Die Aktivität der Place Cells gibt demnach Aufschluss über die Position des Agenten bzw. der Ratte im Labyrinth. Es handelt sich also um eine, im Sinne der in Abschnitt 4.1 formulierten Zielsetzung vereinfachte, erfolgreiche Simulation des Verhaltens von Place Cells unter Verwendung von unüberwachten Verfahren des maschinellen Lernens.

Kapitel 6

Diskussion und Weiterentwicklung

In diesem Kapitel wird die zur Realisierung der Zielstellung verwendete Architektur bewertet und ihr potenzieller Nutzen für das maschinelle Lernen herausgearbeitet. Des Weiteren werden sinnvolle Schritte zur Weiterentwicklung des Aufbaus vorgeschlagen.

6.1 Nutzen der Place-Cell-Simulation

Die Entdeckung von Place Cells im Hippocampus des Gehirns hat einen merklichen Einfluss auf die Entwicklung von KI-Systemen genommen. Es existieren eine Vielzahl an Abhandlungen, welche sich mit der Simulation von Place Cells mithilfe von neuronalen Netzen beschäftigen. Anders als diese Arbeit behandeln viele dabei allerdings vorrangig die Wechselwirkung zwischen Place Cells und den sogenannten Grid Cells, welche den primären Stimulus für die Aktivierung der Place Cells erzeugen [33]–[35].

Gleichzeitig ist ein Modell für die Kodierung von Informationen zur Position eines Agenten in seiner Umgebung natürlich von starker Relevanz für den Bereich der Robotik. Vor allem hier wäre der Einsatz eines Systems zur Simulation von Place Cells, wie in dieser Arbeit beschrieben, denkbar. In vielen Fällen haben Roboter Kameras verbaut, welche die als Input erforderlichen Bilddaten liefern können. Einige der Umsetzungen von Place Units in Robotern nutzen dafür eine ähnliche Abstraktionsebene wie das Modell in dieser Arbeit, bei welcher die Grid Cells nicht explizit umgesetzt werden und die Place Units stattdessen direkt auf die Bilddaten oder indirekt auf daraus extrahierte Features reagieren [36], [37].

Weitere Versuche mit dieser Architektur könnten dementsprechend in Verbindung mit einem simulierten oder physischen Roboter durchgeführt werden. Interessant wäre zum Beispiel, inwiefern sich die Position des Roboters in seiner Umgebung aus der Aktivität der Place Cells rekonstruieren lässt. Dafür wäre es möglicherweise sinnvoll, die Aktivierung mehrerer Place Units mit unterschiedlichem Stärkegrad zu erlauben, da auf diese Weise mehr Informationen kodiert und tendenziell genauere Aussagen getroffen werden können.

6.2 Verwendung der Architektur als Klassifizierer

Das System wird in erster Linie zur Simulation von Place Cells entwickelt. Dabei ist das Ziel, gegebene Bilder aus einem Labyrinth unüberwacht aufgrund ihrer Ähnlichkeit zu gruppieren. Die dabei entstandenen Klassen von Bildern stehen dabei für die Place Units, die jeweils bei der Betrachtung einer Szene im Labyrinth durch den Agenten aktiviert werden. Da zu jeder Zeit immer nur genau eine Place Cell aktiv sein soll wird für jedes anliegende Bild genau eine Kategorie gefunden. Es handelt sich bei der Architektur also um einen unüberwachten Klassifizierer für Bilddaten. Als solcher ist dieser nicht nur auf die Simulation von Place Cells beschränkt, sondern könnte auch für beliebige andere Klassifizierungszwecke verwendet werden. In diesem Abschnitt wird die Tauglichkeit des Systems für die Klassifizierung von anderen Bilddaten experimentell untersucht und bewertet.

6.2.1 Versuch mit zwei unterschiedlichen Datensets

Das Backend des Systems besteht in erster Linie aus dem CNN und der SOM. Lediglich die SOM wird dabei im Trainingsprozess verändert. Das Ziel der Anpassung der SOM-Prototypen im Training ist, die eingehenden Feature-Vektoren, welche vom CNN extrahiert werden und das jeweils aktuell anliegende Bild beschreiben, möglichst optimal abzudecken. Es werden also nicht die Bilder klassifiziert, sondern tatsächlich die Feature-Vektoren, welche die Bilder beschreiben. Aufgrund dessen ist das Ergebnis der Klassifizierung stark abhängig von der Art der extrahierten Features und den darin enthaltenen Informationen. Diese wiederum werden im Training des CNN auf das verwendete Datenset zugeschnitten, um auf diesem möglichst optimale Ergebnisse zu erzielen. Dabei werden diejenigen Features erlernt, anhand de-

rer die benötigten Klassen optimal voneinander unterschieden werden können. Das in dieser Arbeit verwendete MobileNet wurde auf dem ImageNet-Dataset trainiert, welche eine Vielzahl an unterschiedlichen Gegenständen und Lebewesen abbildet. Es ist daher davon auszugehen, dass die ermittelten Features eher generelle Formen und Texturen repräsentieren.

Zur Untersuchung des Einflusses des Trainingsdatensets des CNN auf den Erfolg der Klassifizierung des Systems in unterschiedlichen Disziplinen wird der folgende experimentelle Aufbau durchgeführt. Zunächst werden zwei zu vergleichende Datensets für Klassifizierungszwecke inklusive Labels ausgewählt. Dabei handelt es sich einerseits um das in [38] vorgestellte natürliche Datenset für die Klassifizierung von Objekten, welches im Folgenden als D_1 bezeichnet wird. Dabei ist hervorzuheben, dass D_1 ImageNet in der Art der enthaltenen Bilder ähnelt und diese in Kategorien unterteilt, welche auch in ImageNet vorkommen. Bei dem zweiten Datenset D_2 handelt es sich um die in [39] beschriebene Sammlung von Bildern von verschiedenen Wetterbedingungen. Da die Bilder in ImageNet nach dargestelltem Objekt unterteilt werden und nur ein bedingter Zusammenhang zwischen dem Wetter im Bild und den dargestellten Objekten besteht, ist zu vermuten, dass die vom CNN ermittelten Features für die Klassifizierung der Bilddaten aus D_2 nach den erwarteten Labels weniger brauchbar sind.

Die Bilder der zwei gewählten Datensets werden nun mithilfe des für die Place-Cell-Simulation verwendeten Systems klassifiziert. Dabei wird pro Datenset eine neu initialisierte 3x3-SOM zunächst auf einer Teilmenge der Bilddaten trainiert. Im Anschluss daran werden die Bilder der Testmenge mithilfe der trainierten SOM klassifiziert, wobei die für das aktuell anliegende Bild ermittelte Klasse durch die aktivierte Zelle beschrieben wird. Die Bilder werden also in jeweils eine von neun möglichen Klassen eingeteilt. Um für beide Datensets gleiche Bedingungen zu schaffen werden die Trainingsmengen aus jeweils 200 Bildern und die Testmengen aus jeweils 50 Bildern aus acht der definierten Klassen zusammengesetzt.

Um den Erfolg der Klassifizierung für beiden Datensets vergleichen zu können wird jeweils die Genauigkeit oder auch Accuracy ermittelt. Bei überwachten Lernverfahren berechnet sich diese als:

$$Accuracy = \frac{\text{Anzahl korrekter Klassifizierungen}}{\text{Anzahl Klassifizierungen insgesamt}} \quad (6.1)$$

Bei unüberwachten Lernprozessen, wie der Klassifizierung mithilfe einer SOM, ist allerdings keine klare Aussage darüber möglich, welche Klassifizierungen richtig und welche falsch sind, da keine Zuordnung der gefundenen Klassen zu den Labels des Datensets stattfindet. Stattdessen ist das Ziel, möglichst viele Bilder mit demselben Label in dieselbe Klasse zu sortieren. Aus diesem Grund wird die Accuracy über alle Bilder mit Label l in diesem Fall wie folgt berechnet:

$$Accuracy_l = \frac{\text{Anzahl Bilder mit Label } l \text{ in } c_l}{\text{Anzahl Bilder mit Label } l \text{ insgesamt}} \quad (6.2)$$

wobei c_l diejenige Klasse beschreibt, in welche die meisten Bilder mit Label l eingesortiert wurden. Die Accuracy des gesamten Modells auf einem der Datensets ergibt sich dann als Durchschnitt der Genauigkeitswerte für jedes der Labels.

Das beschriebene Experiment wird für beide Datensets jeweils zehnmal durchgeführt, um Ausreißer in der Performance, welche beispielsweise durch ungünstige Initialisierung der SOM zustande kommen können, möglichst auszugleichen. Die durchschnittliche Genauigkeit der Klassifizierung für D_1 über alle zehn Durchläufe liegt bei 0,964. Die gefundenen Kategorien spiegeln also sehr genau die Unterteilung der Daten nach den Labels wider. Die Klassifizierung von D_2 hingegen weist nur eine durchschnittliche Accuracy von 0,689 auf. Es werden also Klassen gefunden, die nicht den erwarteten Labels entsprechen. Gleichzeitig ist allerdings der Fehler der SOM, also die durchschnittliche Distanz der Eingangsvektoren zu den Prototypen, für D_2 geringer. Daraus lässt sich schließen, dass die auf D_2 ermittelten Featurevektoren nicht etwa weiter verteilt oder schwieriger durch die SOM abzudecken sind. Stattdessen ist die schlechtere Performance desselben Systems auf D_2 vor allem dadurch zu begründen, dass die vom CNN ermittelten Features für die Klassifizierung von Objekten optimiert wurden und somit keine für die Unterscheidung von Wetterbedingungen notwendigen Informationen beinhalten. Im Umkehrschluss ist die Performance auf D_1 besonders gut, da die erwarteten Klassen denen von ImageNet ähneln und die Featurevektoren diese somit deutlich besser beschreiben.

Aus den Ergebnissen des Experiments lässt sich schließen, dass das System in dieser Form nicht für beliebige Klassifizierungsaufgaben von Nutzen ist. Stattdessen kann es in erster Linie dabei unterstützen, in einer Menge von Bildern, für welche keine Klassen bekannt sind, Gruppen von ähnlichen Bildern zu finden. Da die vom CNN ermittelten Features für die Unterscheidung einer Vielzahl von Labels optimiert wurden, kann davon ausgegangen werden, dass die Klassifizierung in vielen

Fällen sinnvolle Ergebnisse liefert. Erwartet man allerdings bestimmte Klassen, so sollte darauf geachtet werden, dass das zu klassifizierende Datenset Ähnlichkeiten zu ImageNet aufweist. Auch in diesen Fällen hat eine Klassifizierung mithilfe einer SOM einige Vorteile gegenüber der Verwendung eines klassischen CNN mit Klassifizierungsschichten. Durch die freie Wahl des Aufbaus der SOM-Karte kann beispielsweise eine beliebige Menge an Klassen gefunden werden, statt nur diejenigen, welche dem CNN im Trainingsprozess bekannt gemacht wurden. Auf diese Weise können Klassen unterteilt oder ähnliche Klassen zusammengefasst werden. Des Weiteren kann durch die Nachbarschaft der Prototypen in der SOM möglicherweise auf Ähnlichkeiten zwischen Klassen geschlossen werden. Alternativ kann das CNN auch mit anderen Trainingsdaten vorgenommen werden, welche mehr Ähnlichkeiten zum gewünschten Anwendungszweck aufweisen.

6.2.2 Weiterentwicklung des Systems als Klassifizierer

Wie im vorherigen Abschnitt beschrieben ist das vorgestellte System in dieser Ausführung nur unter bestimmten Bedingungen für Klassifizierungszwecke einsetzbar. Dieser Abschnitt befasst sich mit Möglichkeiten der Erweiterung der Architektur, um damit eine größere Menge an Klassifizierungsaufgaben lösen zu können.

Anstatt die SOM als letzte Instanz in der Klassifizierung einzusetzen, könnte man diese alternativ auch lediglich für eine Dimensionalitätsreduzierung der Daten als Zwischenschritt verwenden. Man kann die Neuronenkarte der SOM dabei zum Beispiel deutlich größer wählen. Auf diese Weise werden die Bilder in Form ihrer Featurevektoren nicht direkt in komprimierte Klassen unterteilt, sondern stattdessen in Unterklassen, wobei mehr Informationen erhalten bleiben und besser zwischen den Bildern differenziert werden kann. Es handelt sich also um eine Reduzierung der Dimensionalität der Bildfeatures. Bei der Verwendung einer größeren Neuronenkarte sollte für die Nachbarschaftsfunktion auf eine Mexican-Hat-Funktion zurückgegriffen werden [40]. Diese bewegt die unmittelbaren Nachbarn der BMU ebenfalls auf den Eingangspunkt zu, aber entfernt auch Nachbarn ab einer bestimmten Distanz zur BMU, was verhindert, dass sich die Karte in einigen Gebieten staucht.

Aufgrund der räumlichen Abhängigkeiten der Prototypen bilden die SOM-Zellen Nachbarschaften von zueinander ähnlichen Unterklassen. Diese Nachbarschaften können nun mit verschiedenen Verfahren, wie zum Beispiel einer Wasserscheidentransformation [28] in allgemeinere Klassen zusammengefasst werden. Auch die

6 Diskussion und Weiterentwicklung

Verwendung von überwachten Lernmethoden ist dafür denkbar. Sollten durch die Verwendung einer zweidimensionalen Neuronenkarte mit Vierernachbarschaft zu viele Informationen verloren gehen, sodass die gewünschte Klassifizierung nicht möglich ist, können auch höherdimensionale Karten oder andere Nachbarschaftsbeziehungen verwendet werden.

Bei diesen Vorschlägen handelt es sich lediglich um mögliche Ansätze der Weiterentwicklung, deren Nutzen zunächst untersucht werden muss. Diese Untersuchungen sind nicht Teil der Forschungsfrage dieser Arbeit und werden daher nicht weiter verfolgt.

Kapitel 7

Fazit und Ausblick

Diese Arbeit realisiert eine vereinfachte Simulation von im Hippocampus von Ratten und anderen Säugetieren vorkommende Place Cells, welche Informationen über die Position und Ausrichtung des Lebewesens in seiner Umgebung verarbeiten. Dazu wird ein virtueller Agent durch eine simulierte Umgebung gesteuert und gleichzeitig das Verhalten der Place Cells mithilfe von unüberwachten, maschinellen Lernprozessen nachempfunden. Hierbei werden die vom Agenten wahrgenommen Bilddaten mithilfe eines vortrainierten Convolutional Neural Network in Featurevektoren überführt, welche wesentliche Merkmale des Bildes beschreiben. Die Bilder werden daraufhin anhand ihrer Featurevektoren durch eine Self Organizing Map klassifiziert. Die ermittelte Klasse entspricht gleichzeitig der bei der Betrachtung des im Bild dargestellten Ausschnittes des Labyrinths durch den Agenten aktivierten Place Cell.

Durch diesen Aufbau gelingt es wesentliche Merkmale des Verhaltens von Place Cells bei echten Tieren zu simulieren. Dabei wird die Funktionsweise der jeweiligen Komponenten ausführlich erläutert und zentrale Designentscheidungen, welche den Erfolg des Systems beeinflussen können, verglichen. Die dadurch entstandene Kodierung von Positionsinformationen hat einen Nutzen in der Navigation von Robotern und in der Kartierung ihrer Umgebung. Gleichzeitig wird anhand der Simulation des Agenten eine Möglichkeit vorgestellt, synthetische Bilddaten für maschinelles Lernen zu generieren. Ein solcher Ansatz kann für das Training diverser KI-Systeme verwendet werden, welche große Mengen an Bildern erfordern, deren natürliche Gewinnung zu viel Aufwand erfordern würde. Des Weiteren stellt die Architektur zur Simulation der Place Cells auch eine Möglichkeit dar, Bilddaten

unüberwacht zu klassifizieren. Abhängig vom zugrundeliegenden Datenset können damit sehr zufriedenstellende Ergebnisse erzielt werden.

Das Ziel dieser Arbeit ist es lediglich, einen erfolgreichen Ansatz für die Simulation von Place Cells mithilfe von maschinellem Lernen vorzustellen. Dabei werden viele Eigenschaften der Place Units vereinfacht und das zentrale Verhalten dieser Zellen isoliert, um einerseits dem Rahmen der Arbeit gerecht zu werden, andererseits aber auch um so für andere Zwecke des maschinellen Lernens nützliche Vorgehensweisen erschließen zu können. Es gibt diverse andere Ausführungen, welche sich mit Lösungen dieses Problems beschäftigen, die den tatsächlichen Funktionsweisen von biologischen Place Cells gerechter werden. Trotzdem liefert das vorgestellte System diverse vielversprechende Ansätze für sowohl Weiterentwicklung als auch Verwendung zur Lösung anderer Problemstellungen.

Abkürzungsverzeichnis

BMU Best Matching Unit

CNN Convolutional Neural Network

KI Künstliche Intelligenz

SOM Self-Organizing Map

SVM Support Vector Machine

Abbildungsverzeichnis

2.1	Typischer Aufbau eines Neurons im menschlichen Gehirn [10]. Die Dendriten übermitteln das Aktivierungssignal an das Neuron. Ist der Input stark genug, um das Neuron zu aktivieren, wird das Ausgangssignal entlang des Axons geführt, wo es über die Synapsen an benachbarte Neuronen weitergeleitet und dort wiederum als Eingangssignal erfasst wird.	6
2.2	Aufbau eines einlagigen Perzeptron. Die Eingänge x_0 bis x_n werden jeweils mit dem zugehörigen Gewicht w_i multipliziert und aufaddiert. x_0 stellt dabei den Bias-Eingang dar, an welchem immer 1 anliegt. Überschreitet die gewichtete Summe der Eingänge φ den Wert 0, so wird das Perzeptron aktiviert und gibt den Wert 1 aus. Ist sie allerdings negativ, so wird der Output zu 0. Da am Eingang x_0 immer eine 1 anliegt, stellt das negative Biasgewicht $-w_0$ den Schwellwert dar, den die Summe aller anderen gewichteten Eingänge $\sum_{i=1}^n x_i w_i$ überschreiten muss, damit das Perzeptron aktiviert wird.	6
2.3	Aufbau eines von Rumelhart et al. beschriebenen mehrlagigen Perzeptron. Das dargestellte Netz besteht aus insgesamt vier Schichten. Dabei handelt es sich um den Input Layer, zwei Hidden Layer und den Output Layer. Die Neuronen des Input-Layers führen dabei keine Berechnungen durch, sondern symbolisieren lediglich die Eingangswerte des Netzes. Die Ausgänge eines Neurons dienen jeweils als Eingang für Neuronen der folgenden Schicht. Diese Form von voll vernetzten Netzwerken wird auch Feed-Forward-Netz genannt.	8

- | | |
|--|--|
| <p>2.4 Grundlegender Aufbau eines Convolutional Neural Networks. Ein CNN ist in der Regel in zwei Teile, das Feature Extraction Network und ein Feed-Forward-Netzwerk abzugrenzen. Das Feature-Extraction-Netz besteht in seiner einfachsten Form wiederum aus einer alternierenden Folge von Faltungs- und Pooling-Schichten. In den Faltungsschichten oder auch Convolutional Layers werden aus dem Inputbild sogenannte Feature Maps extrahiert, welche im dar-auffolgenden Pooling Layer in ihrer Größe reduziert werden. Die letzten so entstandenen Feature Maps dienen als Input für das Feed-Forward-Netz, welches aus den extrahierten Merkmalen den gewünschten Output ableitet. Handelt es sich zum Beispiel um einen Klassifizierer für Bilder, könnte der Outputvektor die Wahrscheinlichkeiten dafür darstellen, dass das im Eingangsbild dargestellte Objekt der jeweiligen Klasse zugehört.</p> <p>2.5 Grafische Darstellung der zweidimensionalen Faltungsoperation auf einem 5x5 Eingangsbild [20]. Der Faltungskern, in diesem Fall eine 3x3 Matrix, wird, beginnend in der oberen linken Ecke, über das Eingangsbild gelegt. Die Werte unter dem Faltungskern werden jeweils mit dem darüberliegenden Wert der Faltungsmaske multipliziert. Diese Produkte werden dann miteinander aufaddiert und der erhaltene Wert in die obere linke Ecke des Ergebnisbildes geschrieben. In der folgenden Iteration werden sowohl die Faltungsmaske über dem Eingangsbild als auch der zu beschreibende Pixel im Ergebnisbild um eine Koordinate verschoben. Dabei darf die Faltungsmaske niemals über den Rand des Bildes hinausragen. Dieser Vorgang wiederholt sich, bis die Faltungsmaske an jeder möglichen Position war und das Ergebnisbild gefüllt wurde. Besitzt das Eingangsbild die Dimensionen $n \times m$, so liefert die Faltung ein Ergebnisbild der Größe $(n-2) \times (m-2)$. Möchte man eine Änderung der Größe zwischen Input und Output vermeiden, so verwendet man ein Padding, wobei ein zusätzlicher Rahmen an Pixeln um das Ergebnisbild herum generiert wird.</p> | <p style="margin-top: 100px;">10</p> <p>12</p> |
|--|--|

- | | | |
|-----|---|----|
| 2.6 | Stark vereinfachtes Beispiel für Faltungsmasken, welche in der Lernphase eines Klassifizierers gelernt werden könnten [17]. Die rechte Seite zeigt den abstrakten Aufbau eines CNN und die linke Seite das angelegte Eingangsbild. Die unterste Schicht des Netzes stellt die Inputdaten, also Pixelwerte aus dem Bild dar. Die folgenden drei Schichten zeigen die Faltungsmasken, welche in den Schichten des Netzes für die Faltung verwendet werden. Die Bilder repräsentieren dabei das Muster, welches bei der Anwendung der Faltungsmaske auf die Inputbilder aus den vorherigen Schichten detektiert wird. Während die unteren Schichten simple Muster wie Kanten erkennen, können höher liegende Schichten, aufbauend auf den detektierten Merkmalen der vorhergegangenen Schichten, komplexere Muster wie Räder oder Menschen erkennen. | 13 |
| 2.7 | Grafische Darstellung der Pooling-Operation. Die linke Seite zeigt das Eingangsbild und die rechte Seite die beiden Outputbilder, die jeweils durch Max und durch Mean Pooling des Eingangsbildes entstehen. Beim Max Pooling wird jeweils das Maximum des betrachteten Bildbereiches ermittelt und in das Ergebnisbild übernommen. Beim Mean Pooling wird stattdessen der Mittelwert aus allen betrachteten Pixelwerten berechnet. | 14 |
| 2.8 | Darstellung eines Lernschrittes eines kompetitiven Lernalgorithmus. Die Punkte stellen die zweidimensionalen Referenzpunkte dar. Das Viereck beschreibt den Eingangspunkt im aktuellen Lernschritt. Der rote Referenzpunkt besitzt die größte Ähnlichkeit zum Eingangspunkt, in diesem Fall bestimmt als Inverses der Distanz zwischen den Punkten, und ist daher der Gewinner. Somit wird die Ähnlichkeit zwischen diesem Punkt und dem Eingangspunkt vergrößert, also in diesem Fall die Distanz verringert, indem der Referenzpunkt auf den Eingangspunkt zu bewegt wird. | 16 |
| 2.9 | Eine über viele Lernschritte hinweg betrachtete Self Organizing Map im Vektorraum [23]. Die Inputwerte sind zufällig innerhalb des gezeigten Quadrates initialisiert worden. Es zeigt sich, dass die Neuronenkarte sich im Vektorraum entfaltet, um die über den gesamten Bereich verteilten Inputdaten so gut wie möglich abdecken zu können. | 19 |

- 2.10 Darstellung einer Self Organizing Map. Das linke Bild zeigt dabei die Karte in Form eines Graphen, in welcher die Neuronen als Knoten und die Nachbarschaften zwischen zwei Neuronen als Kanten zwischen den beiden Knoten dargestellt werden. Um die Neuronen dabei eindeutig voneinander unterscheiden zu können wurden die Knoten nummeriert. Im linken Bild wurde der Graph in ein 2D-Koordinatensystem übertragen, wobei die Position der Neuronen, also der Prototypen, durch deren Vektor gegeben ist. Das Viereck symbolisiert den vom Eingangsvektor definierten Punkt $x(t)$ in einem Lernschritt t . Das rot gekennzeichnete Neuron stellt die BMU dar, während die gelb markierten Punkte die Neuronen darstellen, welche in der Nachbarschaft N_c liegen. In diesem Fall ist der Radius der Nachbarschaft 1, da wir nur diejenigen Neuronen betrachten, welche von der BMU über maximal eine Kante erreicht werden können. Zusätzlich ist die Änderung der Ortsvektoren der Neuronen in Richtung des Eingangspunkts in Form von Pfeilen gekennzeichnet. 19

4.1 Vereinfachte Darstellung des Ablaufes innerhalb des Backends. Die aus der Simulation gewonnenen Bilddaten werden zunächst jeweils einzeln als Input in das vortrainierte CNN gespeist. Hier werden durch Faltungsoperationen über eine Vielzahl von Schichten hinweg Merkmale aus dem Bild extrahiert und in Form eines Vektors ausgegeben. Dabei handelt es sich um den Merkmalsvektor, welcher bei einem nicht modifizierten CNN den Input für die Fully-Connected-Schichten darstellen würde. Stattdessen wird dieser allerdings in die SOM gegeben, wo er als Eingangspunkt im hochdimensionalen Merkmalsraum interpretiert wird. Wie bei der Funktionsweise einer SOM üblich, wird im letzten Schritt derjenige Prototyp ermittelt, der den geringsten Abstand zu dem Punkt aufweist, welcher das Eingangsbild beschreibt. Je nach aktuell betrachter Phase des Systems wird dieses Ergebnis entweder für den Lernprozess verwendet oder stellt den Output des Backends in Form der aktivierten Place Cell dar.

4.2	Architekturdiagramm für das in dieser Arbeit beschriebene System mit Unterteilung in die zwei vorgesehenen Abläufe Training und Inferenzieren. Die Trainingsphase beginnt damit, dass das Frontend einen Trainings- und Testdatensatz für den Lernprozess der SOM im Backend zur Verfügung stellt. Dafür generiert der Screenshot Generator eine Vielzahl an Bildern der Simulation des Labyrinths aus Perspektive des Agenten. Diese Bilder werden dann in zwei disjunkte Datensätze, die Trainings- und die Testdaten eingeteilt, wobei die Trainingsdaten üblicherweise deutlich mehr Bilder enthalten. Das Backend kann nun zu einem beliebigen Zeitpunkt danach auf die generierten Daten zugreifen, um diese zunächst mithilfe des vortrainierten CNN in Featurevektoren umzuwandeln. Die Featurevektoren dienen also als Trainingsinputs, mithilfe welcher die Prototypen der SOM angepasst werden können, bis die Testdaten zufriedenstellende Ergebnisse liefern und die Trainingsphase beendet werden kann. Während der Inferenzphase laufen beide Teilsysteme gleichzeitig, um eine Aktivierung der Place Units in der SOM zur Laufzeit des Frontends zu ermöglichen. Dabei wird auf die gleiche Weise in regelmäßigen Zeitabständen ein Bild aus der Perspektive des Agenten generiert, nun aber über den Web Socket Client mithilfe von TCP zum lokalen Web Socket Server des Backends übertragen. Dort wird wieder zunächst der Featurevektor des Bildes mithilfe des CNN extrahiert und daraus durch die SOM die zu aktivierende Place Unit ermittelt. Die Information über die Aktivierung der Place Units wird dann an den Web Socket Server übergeben, welcher diese dem Client als Antwort auf seine Anfrage zurücksendet. Nun kann die aktivierte Place Unit durch den Place Unit Visualizer im Frontend angezeigt werden.	28
4.3	Ansicht des Labyrinths in Form einer Insel aus der Vogelperspektive. Drei der vier Gebiete; der Strand, der Wald und die Winterlandschaft; sind deutlich differenzierbar.	30
4.4	Simulation des Labyrinths aus Perspektive des Agenten, aus welcher auch die Bilddaten für Training und Inferenz generiert werden. Die linke obere Ecke zeigt die Neuronenkarte, welche auf der im Backend laufenden SOM basiert. Das hervorgehobene Neuron stellt die aktuell aktive Place Cell dar.	32

Quellcodeverzeichnis

4.1	WebSocket Client	32
4.2	WebSocket Server	34

Literatur

- [1] David Silver, Aja Huang, Chris J. Maddison u. a., „Mastering the game of Go with deep neural networks and tree search“, *Nature*, Jg. 529, Nr. 7587, S. 484–489, Jan. 2016. DOI: 10.1038/nature16961.
- [2] Wolfgang Ertel, *Introduction to artificial intelligence*. Springer, 2018.
- [3] Stephanie Dick, „Artificial Intelligence“, *Issue 1*, Juni 2019. DOI: 10.1162/99608f92.92fe150c.
- [4] Wei Liu, Guangda Zhuang, Xin Liu, Shaobo Hu, Ruilin He und Yuhu Wang, „How do we move towards true artificial intelligence“, in *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, (Haikou, Hainan, China, 20.–22. Dez. 2022), IEEE, Dez. 2021, S. 2156–2158. DOI: 10.1109/hpcc-dss-smartcity-dependsys53884.2021.00321.
- [5] Barbara Clancy, Barbara L. Finlay, Richard B. Darlington und K.J.S. Anand, „Extrapolating brain development from experimental species to humans“, *NeuroToxicology*, Jg. 28, Nr. 5, S. 931–937, 2007, Twenty-Third International Neurotoxicology Conference: ”Neurotoxicity in Development and Aging“. DOI: <https://doi.org/10.1016/j.neuro.2007.01.014>.
- [6] Chris M. Bird und Neil Burgess, „The hippocampus and memory: insights from spatial processing“, *Nature Reviews Neuroscience*, Jg. 9, Nr. 3, S. 182–194, März 2008. DOI: 10.1038/nrn2335.
- [7] J. O’Keefe und J. Dostrovsky, „The hippocampus as a spatial map. Preliminary evidence from unit activity in the freely-moving rat“, *Brain Research*, Jg. 34, Nr. 1, S. 171–175, Nov. 1971. DOI: 10.1016/0006-8993(71)90358-1.

- [8] John O'Keefe, „Place units in the hippocampus of the freely moving rat“, *Experimental Neurology*, Jg. 51, Nr. 1, S. 78–109, Jan. 1976. DOI: 10.1016/0014-4886(76)90055-8.
- [9] Ltd. Huawei Technologies Co., *Artificial Intelligence Technology*. Springer Nature Singapore, 2023. DOI: 10.1007/978-981-19-2879-6.
- [10] Chris M. Bishop, „Neural networks and their applications“, *Review of Scientific Instruments*, Jg. 65, Nr. 6, S. 1803–1832, Juni 1994. DOI: 10.1063/1.1144830.
- [11] Warren S. McCulloch und Walter Pitts, „A logical calculus of the ideas immanent in nervous activity“, *The Bulletin of Mathematical Biophysics*, Jg. 5, Nr. 4, S. 115–133, Dez. 1943. DOI: 10.1007/bf02478259.
- [12] Frederico A.C. Azevedo, Ludmila R.B. Carvalho, Lea T. Grinberg u. a., „Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain“, *The Journal of Comparative Neurology*, Jg. 513, Nr. 5, S. 532–541, Apr. 2009. DOI: 10.1002/cne.21974.
- [13] F. Rosenblatt, „The perceptron: A probabilistic model for information storage and organization in the brain.“, *Psychological Review*, Jg. 65, Nr. 6, S. 386–408, 1958. DOI: 10.1037/h0042519.
- [14] Laveen N Kanal, „Perceptron“, in *Encyclopedia of Computer Science*, 2003, S. 1383–1385.
- [15] M. Minsky und S. Papert, *Perceptrons*. M.I.T. Press, 1969.
- [16] D Rumelhart, G Hinton und R Williams, „Learning Internal Representations by Error Propagation“, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D. E. Rumelhart und J. L. McClelland, Hrsg. Cambridge, MA: MIT Press, 1986, Bd. 1, S. 318–362.
- [17] Ian Goodfellow, Yoshua Bengio und Aaron Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [18] Hannes Schulz und Sven Behnke, „Deep Learning“, *KI - Künstliche Intelligenz*, Jg. 26, Nr. 4, S. 357–363, Mai 2012. DOI: 10.1007/s13218-012-0198-z.
- [19] Saad Albawi, Tareq Abed Mohammed und Saad Al-Zawi, „Understanding of a convolutional neural network“, in *2017 International Conference on Engineering and Technology (ICET)*, (Antalya, Turkey, 21.–23. Aug. 2017), 2017, S. 264–296. DOI: 10.1109/ICEngTechnol.2017.8308186.

- [20] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do und Kaori Togashi, „Convolutional neural networks: an overview and application in radiology“, *Insights into Imaging*, Jg. 9, Nr. 4, S. 611–629, Juni 2018. DOI: 10.1007/s13244-018-0639-9.
- [21] Phil Kim, „Convolutional Neural Network“, in *MATLAB Deep Learning*, Apress, 2017, S. 121–147. DOI: 10.1007/978-1-4842-2845-6_6.
- [22] Teuvo Kohonen, „Automatic formation of topological maps of patterns in a self-organizing system“, in *Proceedings of the 2nd scandinavian Conference on Image Analysis*, (Espoo, Finland), 1981, S. 214–220.
- [23] T. Kohonen, „The self-organizing map“, *Proceedings of the IEEE*, Jg. 78, Nr. 9, S. 1464–1480, 1990. DOI: 10.1109/5.58325.
- [24] E I Knudsen, S Lac und S D Esterly, „Computational Maps in the Brain“, *Annual Review of Neuroscience*, Jg. 10, Nr. 1, S. 41–65, März 1987. DOI: 10.1146/annurev.ne.10.030187.000353.
- [25] David E. Rumelhart und David Zipser, „Feature Discovery by Competitive Learning“, *Cognitive Science*, Jg. 9, Nr. 1, S. 75–112, Jan. 1985. DOI: 10.1207/s15516709cog0901_5.
- [26] Dimpy Varshni, Kartik Thakral, Lucky Agarwal, Rahul Nijhawan und Ankush Mittal, „Pneumonia Detection Using CNN based Feature Extraction“, in *2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, (Coimbatore, India, 20.–22. Feb. 2019), IEEE, Feb. 2019. DOI: 10.1109/icecct.2019.8869364.
- [27] Olga Russakovskiy, Jia Deng, Hao Su u. a., „ImageNet Large Scale Visual Recognition Challenge“, *International Journal of Computer Vision*, Jg. 115, Nr. 3, S. 211–252, Apr. 2015. DOI: 10.1007/s11263-015-0816-y.
- [28] Jie Zhang und John Kerekes, „Unsupervised urban land-cover classification using WorldView-2 data and self-organizing maps“, in *2011 IEEE International Geoscience and Remote Sensing Symposium*, (Vancouver, British Columbia, Canada, 24.–29. Juli 2011), IEEE, Juli 2011. DOI: 10.1109/igarss.2011.6048920.
- [29] Patricia E. Sharp, „Computer simulation of hippocampal place cells“, *Psychobiology*, Jg. 19, Nr. 2, S. 103–115, Juni 1991. DOI: 10.3758/bf03327179.
- [30] Thomas J. Davidson, Fabian Kloosterman und Matthew A. Wilson, „Hippocampal Replay of Extended Experience“, *Neuron*, Jg. 63, Nr. 4, S. 497–507, Aug. 2009. DOI: 10.1016/j.neuron.2009.07.027.

- [31] Andrew G. Howard, Menglong Zhu, Bo Chen u. a., *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, 2017. DOI: 10.48550/ARXIV.1704.04861.
- [32] Florent Forest, Mustapha Lebbah, Hanane Azzag und Jérôme Lacaille, *A Survey and Implementation of Performance Metrics for Self-Organized Maps*, 2020. DOI: 10.48550/ARXIV.2011.05847.
- [33] S. Saeidi und F. Towhidkhah, „From Grid Cells to Place Cells: A Radial Basis Function Network Model“, in *2008 Cairo International Biomedical Engineering Conference*, (Cairo, Egypt, 18.–20. Dez. 2008), IEEE, Dez. 2008. DOI: 10.1109/cibec.2008.4786111.
- [34] Naigong Yu, Hejie Yu, Lin Wang, Yishen Liao und Chunlei Yin, „A Model of Information Circulation and Transmission Network of Grid Cells in Entorhinal Cortex and Place Cells in the Hippocampus of Rat“, in *2020 IEEE 9th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, (Chongqing, China, 11.–13. Dez. 2020), IEEE, Dez. 2020. DOI: 10.1109/itaic49862.2020.9338829.
- [35] Gonzalo Tejera, Martin Llofriu, Alejandra Barrera und Alfredo Weitzenfeld, „A spatial cognition model integrating grid cells and place cells“, in *2015 International Joint Conference on Neural Networks (IJCNN)*, (Killarney, Ireland, 12.–17. Juli 2015), IEEE, Juli 2015. DOI: 10.1109/ijcnn.2015.7280557.
- [36] Xiaomo Zhou, Cornelius Weber und Stefan Wermter, „A Self-organizing Method for Robot Navigation based on Learned Place and Head-Direction Cells“, in *2018 International Joint Conference on Neural Networks (IJCNN)*, (Rio de Janeiro, Brazil, 13. Juli 2018), IEEE, Juli 2018. DOI: 10.1109/ijcnn.2018.8489348.
- [37] Hesam Omranpour und Saeed Shiry, „Representation of map model for mobile robot based on hippocampus place cell functionality“, in *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, (Tehran, Iran, 22. Dez. 2017), IEEE, Dez. 2017. DOI: 10.1109/kbei.2017.8324890.
- [38] Prasun Roy, Subhankar Ghosh, Saumik Bhattacharya und Umapada Pal, „Effects of Degradations on Deep Neural Network Architectures“, *arXiv preprint arXiv:1807.10108*, 2018.
- [39] Haixia Xiao, Feng Zhang, Zhongping Shen, Kun Wu und Jinglin Zhang, „Classification of Weather Phenomenon From Images by Using Deep Convolutional Neural Network“, *Earth and Space Science*, Jg. 8, Nr. 5, Mai 2021. DOI: 10.1029/2020ea001604.

- [40] Dubravko Miljkovic, „Brief review of self-organizing maps“, in *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, (Opatija, Croatia, 22.–26. Mai 2017), IEEE, Mai 2017. DOI: 10.23919/mipro.2017.7973581.