Universität Rostock

Traditio et Innovatio

Masterarbeit

Intra-Life-Learning mittels parallelisierter Neuroevolution

Vorgelegt von:

Ole Fenske

ole.fenske@uni-rostock.de

Eingereicht am:

16. November 2020

Betreuer:

Prof.Dr.rer.nat.habil. Andreas Heuer

M.Sc. Robin Nicolay

Prof.Dr.rer.nat.habil. Alke Martens

Universität Rostock | Fakultät für Informatik und Elektrotechnik

Institut für Informatik | Albert-Einstein-Str. 22, 18059 Rostock

Tel: +49 381 498 7451

https://doi.org/10.18453/rosdok_id00002921

Kurzfassung

Im Bereich des maschinellen Lernens werden neuronale Netze dazu eingesetzt, um verschiedensten Aufgaben begegnen zu können. Dabei können diese Modelle in verschiedenen Szenarien (überwacht, unüberwacht, bestärkend) und mit unterschiedlichen Techniken (Backpropagation, Neuroevolution, Hebb-Lernen, usw.) trainiert werden. Das maschinelle Lernen ist dabei ein klassisches Trainings-Paradigma, bei dem das neuronale Netz von Grund auf neu trainiert wird, um das notwendige Wissen für eine spezifische Aufgabe zu generieren. Dieses Vorgehen funktioniert gut in Domänen, deren Aufgaben statischer Natur sind und deren unterliegende Struktur sich mit der Zeit nicht verändert. Problemstellungen aus der Praxis zeichnen jedoch oft ein anderes Bild, da die jeweiligen Domänen, in denen die neuronalen Netze agieren müssen, stetigen Veränderungen unterworfen sind. Aus diesem Grund entsteht die Anforderung der Adaptierbarkeit von Wissen, welches in neuronalen Netzen gespeichert ist.

Das maschinelle Lernen sieht dafür eine erneute Trainingsphase des jeweiligen neuronalen Netzes vor. Dabei muss das neuronale Netz jedoch von Grund auf neu trainiert werden und schon vorhandenes Wissen wird dabei komplett ignoriert, was in hohem Aufwand resultiert. Im Gegensatz dazu beschreibt das *Intra-Life-Learning* ein Lern-Paradigma, um neuronale Netze auf mehrere Aufgaben trainieren zu können und bereits vorhandenes Wissen so zu erweitern, dass sich das neuronale Netz an verändernde Umgebungen anpassen kann. Das setzt zum einen voraus, dass bisher gelerntes Wissen nicht vergessen wird und zum anderen, dass das Erlernen von neuen Problemen mit wachsendem Wissen effizienter werden soll. Die Zielstellung dieser Arbeit ist es, Techniken der sogenannten *Neuroevolution* darauf hin zu untersuchen, in welchem Ausmaß sie das Intra-Life-Learning im Zusammenhang mit neuronalen Netzen ermöglichen können.

Abstract

In the field of machine learning, neural networks are used in order to be able to deal with a wide variety of tasks. These models can be trained in different scenarios (supervised, unsupervised, reinforcement) and with different techniques (backpropagation, neuroevolution, Hebb learning, etc.). Machine learning is a classical training paradigm in which the neural network is trained from scratch to generate the necessary knowledge for a specific task. This approach works well in domains whose tasks are static and whose underlying structure does not change over time. However, practical problems often draw a different picture, because the domains in which the neural networks have to operate are subject to constant change. This leads to the requirement of adaptability of knowledge stored in neural networks.
The classical training paradigm suggests a new training phase of the respective neural network in order to learn a new task or to adapt to a changing environment. However, the neural network has to be trained again from scratch and already existing knowledge is completely ignored, which results in high costs. *Intra-Life-Learning* describes a learning paradigm to train neural networks for several tasks and to extend already existing knowledge so that the neural network can adapt its behaviour to the changing environment. This requires on the one hand that previously learned knowledge is not forgotten and on the other hand that the learning of new problems should become more efficient with increasing knowledge. The first goal of this thesis is to investigate techniques of the so-called *neuroevolution*. Based on this we want to analyze to what extent these techniques can be used for Intra-Life-Learning with neural networks.

Schlagworte: Neuronale Netze, Neuroevolution, Maschinelles Lernen, Meta-Lernen, Kontinuierliches Lernen

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel dient als Einstieg in die Arbeit. Zunächst wird eine kurze Motivation in Unterkapitel 1.1 gegeben, um zu beschreiben, wieso das hier behandelte Themengebiet relevant ist. Anschließend führt Unterkapitel 1.2 den Leser in den Hintergrund der Arbeit ein. Dabei wird die bestehende Thematik in vorhandene Forschungsgebiete eingeordnet und das Gesamt-Konzept der Arbeit näher erläutert. Das Unterkapitel 1.3 stellt dann die Ziele der Arbeit vor. Um die einzelnen Inhalte der folgenden Kapitel besser im Gesamt-Kontext verorten zu können, wird in Unterkapitel 1.4 abschließend der Aufbau dieser Arbeit geschildert.

1.1 Motivation

Das maschinelle Lernen und speziell das vertiefende Lernen haben in vielen Bereichen zu zahlreichen Durchbrüchen geführt. So ist beispielsweise einer künstlichen Intelligenz namens *AlphaGo* gelungen, einen Europa-Meister im Brettspiel *Go* zu schlagen und das allein auf Basis von erlerntem Wissen, welches aus einer Vielzahl an vorher beobachteten Spielen und „Ausprobieren“ generiert wurde (vgl. [Sil+16]). Dabei hatte die künstliche Intelligenz anfangs keinerlei Kenntnisse über das Regelwerk oder bestimmte Strategien bezüglich des Spiels. All diese Informationen wurden allein auf Basis von Beobachtungen erlernt. Das Modell, welches für das Speichern dieses Wissens genutzt wurde, ist ein sogenanntes neuronales Netz. Dabei handelt es sich um das künstliche Äquivalent zu biologischen neuronalen Netzen, welche in Form des Nervensystems oder Gehirns in biologischen Organismen gefunden werden können. Im mathematischen Sinne repräsentieren neuronale Netze sogenannte Funktionsapproximatoren. Das bedeutet, dass sie für eine Vielzahl an Aufgaben verwendet werden können und somit nahezu universell einsetzbar sind. Andere Modelle

des maschinellen Lernens, wie z. B. Support-Vector-Machines, Klassifikationsbäume oder auch Assoziationsregeln, sind im Gegenteil zu neuronalen Netzen in ihrem Anwendungsgebiet klassischerweise auf spezifische Aufgabenstellungen beschränkt. Das stellt einen klaren Vorteil von neuronalen Netzen gegenüber anderen Modellen dar, weshalb sie sich zu einem der dominierenden Modelle im Zusammenhang mit dem maschinellen Lernen entwickelt haben.
Die Komplexität von neuronalen Netzen hat zu einer Vielzahl an Lernalgorithmen geführt, mithilfe derer das jeweilige Wissen generiert werden kann. Dabei lassen sich die unterschiedlichen Techniken durch die verschiedenen Szenarien des maschinellen Lernens unterteilen. Der Backpropagation-Algorithmus wird z. B. überwiegend für das überwachte Lernen eingesetzt. Verschiedene Algorithmen der Neuroevolution werden wiederum für das bestärkende Lernen eingesetzt, während die Hebb-Regel ihre Anwendung im Kontext des unüberwachten Lernens findet. Im Fall von AlphaGo wurden z. B. Techniken des überwachten und bestärkenden Lernens genutzt. Das herkömmliche Vorgehen beim maschinellen Lernen sieht dabei vor, dass das jeweilige neuronale Netz in einer Trainingsphase trainiert wird, also mithilfe eines Lernalgorithmus Wissen generiert. Dieses erlernte Wissen wird dann verwendet, damit das neuronale Netz eine spezifische Aufgabe wie z. B. das Go spielen lösen kann.
Dieses Vorgehen funktioniert gut in Domänen, in denen sich die grundlegende Mechanismen bzw. Regeln mit der Zeit nicht verändern. Viele Aufgabenstellungen, die der realen Welt entspringen, wie z. B. das autonome Fahren, zeichnen sich jedoch durch stetig verändernde Umgebungen aus. Das wiederum führt dazu, dass neuronale Netze nach dem herkömmlichen Lernparadigma des maschinellen Lernens erneut die Trainingsphase durchlaufen und von Grund auf neu trainiert werden müssten, um sich an die verändernde Umgebung und ihre neuen Aufgaben anzupassen. Bisher erlerntes Wissen wird dabei komplett ignoriert und somit obsolet. Ferner sind herkömmliche Lernalgorithmen wie Backpropagation ebenfalls nicht dazu in der Lage, neuronale Netze auf weitere Aufgaben zu trainieren, da bei jedem neuen Lernprozess das bisher gesammelte Wissen bezüglich der vorher gelernten Aufgaben überschrieben wird. Dieses Phänomen ist unter dem Namen *katastrophales Vergessen* bekannt und wird schon seit Jahren im Zusammenhang mit dem maschinellen Lernen näher untersucht (vgl. [Fre99]).

Das *Intra-Life-Learning* beschreibt ein Lernparadigma, bei dem neuronale Netze resistent gegenüber den oben genannten Problemen sein sollen. So soll es möglich sein, dass in einem neuronalen Netz gespeicherte Wissen dynamisch an sich verändernde Umgebungen anzupassen, ohne es dazu von Grund auf neu trainieren zu müssen. Ebenso soll das neuronale Netz auch um Wissen bezüglich einer neuen Aufgabe erweitert werden können, ohne das bisher gesammelte Wissen zu vergessen. Um dies zu bewerkstelligen, können unter anderem Algorithmen der Neuroevolution verwendet werden. Diese Art von Lernalgorithmen ist dabei von der natürlichen Evolution von biologischen neuronalen Netzen inspiriert, welche zu menschlicher Intelligenz geführt haben. Das nächste Kapitel wird die Thematik dieser Arbeit in bestehende Forschungsgebiete einordnen und somit das notwendige Hintergrund-Wissen vermitteln, welches für die weiteren Kapitel notwendig ist.

1.2 Hintergrund

Diese Arbeit untersucht verschiedene Algorithmen aus den Gebieten des maschinellen Lernens und der evolutionären Algorithmen im Kontext des Intra-Life-Learning. Dabei wird versucht, die verschiedenen Verfahren miteinander zu kombinieren bzw. im Kontext von anderen Forschungsgebieten zu untersuchen, um sich daraus ergebende Vorteile darlegen zu können. In diesem Zusammenhang werden die nächsten beiden Abschnitte 1.2.1 und 1.2.2 jeweils die Teilgebiete des maschinellen Lernens und der evolutionären Algorithmen vorstellen. Die Zielstellung dieses Unterkapitels besteht also darin, dem Leser einen ersten Einstieg in die Thematik zu ermöglichen, sodass spätere Inhalte besser eingeordnet werden können.

1.2.1 Maschinelles Lernen

Maschinelles Lernen (ML) als Forschungsgebiet beschäftigt sich damit, Maschinen in die Lage zu versetzen, automatisch aus Daten Wissen zu erlernen, mithilfe dessen die Lösung einer bestimmten Aufgabe generiert werden kann (vgl. [Alp19, S. 2]). Dabei wird eine ML-Technik durch zwei Komponenten charakterisiert: das *Modell* und den *Lernalgorithmus*. Das erlernte Wissen wird mithilfe von Modellen repräsentiert, sodass der Computer auf Grundlage dieser Modelle entweder Vorhersagen

über die Zukunft treffen (*prädiktives* Modell) oder die den Daten zugrunde liegende Struktur erklären kann (*deskriptives* Modell). In diesem Zusammenhang bedeutet Lernen, dass mithilfe von Lernalgorithmen jene Modelle generiert werden.
Demnach kann man beim maschinellen Lernen zwischen verschiedenen Modellen und auch Lernalgorithmen unterscheiden, wobei nicht jeder Lernalgorithmus für jedes Modell und nicht jedes Modell für jede Art von Datensatz anwendbar ist. Daraus resultieren auch die Teilgebiete des maschinellen Lernens: *überwachtes*, *unüberwachtes* und *bestärkendes* Lernen. Dabei beschreibt jedes dieser Teilgebiete ein bestimmtes Lernszenario, welches jeweils spezifische Eigenschaften aufweist. Der Rest dieses Abschnittes wird diese drei Teilgebiete des maschinellen Lernens und ihre Techniken näher erläutern. Dabei wird jedoch nicht vertieft auf konkrete Algorithmen oder Modelle eingegangen, sondern nur ein Überblick über sie gegeben. Zusätzlich wird auch das *Transfer-Lernen* kurz erläutert, da es im weiteren Verlauf der Arbeit nochmals von Bedeutung sein wird.

Überwachtes Lernen
Das überwachte Lernen kann als *Lernen aus Beobachtungen* interpretiert werden. Es steht also ein Datensatz an Beobachtungen für eine spezifische Aufgabe zur Verfügung, aus dem dann Wissen generiert werden kann, um die jeweilige Aufgabe lösen zu können. Dabei kann zwischen zwei verschiedenen Szenarien des überwachten Lernens unterschieden werden:

- **Klassifikation:** Feststellen, zu welcher Kategorie/Klasse ein unbekanntes Beispiel gehört, auf Basis eines Trainingsdatensatzes, der bekannte Beispiele enthält, bei denen klar ist, zu welcher Kategorie/Klasse sie gehören.
- **Regression:** Vorhersage von Ereignissen (wie z. B. der Absatzmenge eines Supermarktes am morgigen Tag) auf Basis eines Trainingsdatensatzes, welcher unterschiedliche Kennzahlen zu Ereignissen der Vergangenheit (vergangene Absatzmengen eines Supermarktes, Wetter, Jahreszeit usw.) enthält.

Ein Beispiel für Klassifikation könnte die folgende Aufgabe sein (vgl. [Alp19, S. 5]):

Beispiel 1.0.1 *Eine Bank möchte Kredite vergeben. Dafür muss sie jedoch die Kreditwürdigkeit ihrer Kunden abschätzen können. Die Bank besitzt Daten über die Vergabe von vergangenen Krediten. Diese Daten enthalten unter anderem Informationen zum Alter, Beruf und Einkommen des jeweiligen Kunden. Zusätzlich verfügt die Bank auch über die Information, ob der jeweilige Kunde seinen Kredit vollständig zurückgezahlt hat oder nicht. Auf Basis dieser Daten möchte die Bank nun ein ML-Modell generieren, mithilfe dessen sie über die Vergabe zukünftiger Kredite entscheiden kann.*

Die Informationen bezüglich der Kunden stellen also den Datensatz dar, wobei ein einzelner Kunde mit seinen *Attributen* (Alter, Beruf, Einkommen) einen einzelnen *Datenpunkt* darstellt. Die Information, ob ein Kunde seinen Kredit zurückgezahlt hat oder nicht, wird dann als *Annotation* des jeweiligen Datenpunktes bezeichnet.
Das Ziel bei dem überwachten Lernen besteht dann darin, Zusammenhänge zwischen den Attributen und Annotationen eines Datensatzes zu finden, welche durch Regeln ausgedrückt werden. Diese Regeln werden dann mithilfe des jeweiligen ML-Modells repräsentiert, wodurch Vorhersagen über bisher nicht gesehene bzw. zukünftige Ereignisse getroffen werden können (prädiktives Modell). Bildet man das Ganze wieder auf Beispiel 1.0.1 ab, kann man also mithilfe des erlernten Modells, Aussagen über die Kreditwürdigkeit neuer Kunden auf Basis ihres Alters, Berufs und Einkommens treffen.
Diese Art von überwachtem Lernen ist auf eine Vielzahl von realen Aufgaben anwendbar. Beispiele dafür sind die *Mustererkennung*, *Gesichtserkennung*, *medizinische Diagnostik* und *Spracherkennung* (vgl. [Alp19, S7-8]).

Unüberwachtes Lernen

Das unüberwachte Lernen kann man als *Lernen ohne Feedback* bezeichnen. Es stehen somit zwar Daten zur Verfügung, jedoch keine Annotationen zu den einzelnen Datenpunkten. Mit dieser Art des Lernens sollen also Informationen über die den Daten zugrunde liegende Struktur generiert werden (deskriptives Modell). Mithilfe dieses Wissens lassen sich zwar keine Vorhersagen treffen, stattdessen aber Aussagen

über die Struktur des Datensatzes und häufig auftretende Muster formulieren (vgl. [Alp19, S. 12]). Das unüberwachte Lernen lässt sich ebenfalls in unterschiedliche Szenarien unterteilen:

- **Clustering:** Einteilung von einzelnen Datenpunkten in Cluster (Gruppen) auf Basis ihrer Attribute und der Annahme, dass Datenpunkte desselben Clusters sich eher ähneln als solche aus unterschiedlichen Clustern.
- **Assoziationsanalyse:** regelbasiertes Verfahren zum Finden von interessanten Zusammenhängen zwischen den einzelnen Attributen der Datenpunkte in einem Datensatz.

Beim Clustering werden die Datenpunkte eines Datensatzes also in Cluster unterteilt und geben somit Aufschluss darüber, wie das Mengenverhältnis der einzelnen Datenpunkte zu einander ist bzw. wie sehr sich bestimmte Datenpunkte ähneln. Unterteilt man einen Datensatz in n verschiedene Cluster, können diese Cluster als n unterschiedliche Klassen angenommen und somit wieder für die Klassifikation verwendet werden. Jeder Datenpunkt kann dann entsprechend seines Clusters mit der jeweiligen Information annotiert werden, welche anzeigt, zu welcher Klasse er gehört. Bleibt man bei Beispiel 1.0.1, so könnte man die bestehenden Kunden in Cluster unterteilen und somit der Bank eine natürliche Gruppierung ihrer Kunden zur Verfügung stellen. Dieses Verfahren wird dann als *Kundensegmentierung* bezeichnet und stellt eine der vielen Anwendungen von Clusteranalysen dar (vgl. [Alp19, S. 12]). Weitere Anwendungsfelder der Clusteranalyse sind in diesem Zusammenhang z. B. die *Bildkompression*, das *Dokument-Clustering* oder auch die *Sequenzanalyse* von DNA/RNA in der Bioinformatik (vgl. [Alp19, S. 13-14]).
Bei der Assoziationsanalyse versucht man hingegen sogenannte *Assoziationsregeln* zu finden, welche interessante Zusammenhänge zwischen den einzelnen Attributen eines Datenpunktes liefern. Ein Beispiel könnte die *Warenkorbanalyse* sein:

Beispiel 1.0.2 *Ein Supermarkt sammelt über jeden Kunden die Daten bezüglich seiner Einkäufe, die er im jeweiligen Supermarkt getätigt hat. Zu den Daten gehören alle Produkte, die der jeweilige Kunde während eines Einkaufes an der Kasse bezahlt hat. Der Supermarkt möchte auf Basis dieser Daten das Einkaufserlebnis des Kunden optimieren.*

Bei diesem Beispiel stellt der Einkauf eines Kunden einen einzelnen Datenpunkt dar. Die jeweiligen Produkte, die der Kunde gekauft hat, sind dann die Attribute dieses Datenpunktes. Auf dieser Basis können dann Assoziationen zwischen den einzelnen Attributen, also den Produkten, hergestellt werden. Dies können dann z. B. Assoziationsregeln in folgender Form sein: „*Kauft ein Kunde Produkt X, dann kauft er zu 90% auch Produkt Y*“. Anhand solcher Regeln, könnte der Supermarkt also z. B. die Waren im Regal umsortieren, sodass der Kunde dazu angehalten wird, mehr einzukaufen als üblich. Jedoch ist diese Art des unüberwachten Lernens nicht nur auf Supermärkte und die Warenkorbanalyse anwendbar, sondern z. B. auch auf Web-Shops und andere Arten von Online-Portalen (vgl. [Alp19, S. 5]). Auch außerhalb von Shops kann das unüberwachte Lernen angewendet werden. Ein Beispiel ist die Auswertung von Symptomen bei Erkrankungen.

Bestärkendes Lernen

Das bestärkende Lernen kann als *Lernen durch Ausprobieren* übersetzt werden. Im Gegensatz zum überwachten und unüberwachten Lernen geht es also nicht darum, aus einem vorhandenen Datensatz neue Informationen zu generieren, sondern auf Grundlage einer Sequenz vergangener Aktionen eine Taktik abzuleiten, die für die Bewältigung einer spezifischen Aufgabe verwendet werden kann (vgl. [Alp19, S. 14]). Im Zusammenhang mit dem bestärkenden Lernen spricht man auch oft von sogenannten *Agenten*. Dabei handelt es sich um eine künstliche Intelligenz, die einen Roboter in einer simulierten oder realen Umgebung steuert und für eine bestimmte Aufgabe eingesetzt wird. Ein Beispiel dafür ist ein Staubsauger-Roboter:

Beispiel 1.0.3 *Ein Roboter soll dafür eingesetzt werden, die Wohnung autonom saugen zu können. Dabei hat der Roboter anfangs keinerlei Informationen über den Aufbau der Wohnung und wo welche Zimmer liegen. Jedoch verfügt der Roboter über Sensoren, die anzeigen, ob sich Gegenstände in seiner Nähe befinden (vorne, hinten, links, rechts) und ob die Fläche, über die er gerade fährt, schmutzig oder sauber ist. Der Roboter kann außerdem zwischen den Aktionen „Bewegen nach links“, „Bewegen nach rechts“, „Bewegen nach vorne“ und „Saugen/nicht saugen“ wählen. Zusätzlich verfügt der Roboter über ein Belohnungssystem, welches anzeigt, ob das von ihm gezeigte Verhalten (= die gewählte Aktion) erwünscht ist oder nicht.*

Der Agent in diesem Beispiel versucht durch Ausprobieren herauszufinden, welche Aktion zu welchem Zeitpunkt angebracht ist (= sein Verhalten). Als Hilfestellung dient hierbei das Belohnungssystem, mithilfe dessen die korrekte Abbildung von Input (Sensordaten) auf Output (mögliche Aktionen des Agenten) gefunden werden soll. Die Sensordaten stellen dann den jeweils aktuellen Zustand der Welt dar, in der sich der Agent bewegt. Im Falle des Beispiels 1.0.3 soll der Agent also z. B. Lernen, dass er saugen soll (Aktion), wenn der Boden schmutzig ist (Zustand der Welt). Diese Abbildung vom Zustand der Welt auf eine Aktion stellt dann das Verhalten des Agenten dar.
Bei dieser Art des maschinellen Lernens kann jedoch auch zwischen verschiedenen Szenarien unterschieden werden. So ist z. B. die Beobachtbarkeit des Zustandes der jeweiligen Welt von Bedeutung. So gibt es Umgebungen, in denen der aktuelle Zustand vollständig beobachtbar ist und welche, in denen dieser nur teilweise beobachtet werden kann (vgl. [Alp19, S. 15]). In letzterem Fall spricht man dann auch von *Schließen unter Unsicherheit*. Zusätzlich ist auch ein Szenario denkbar, bei dem mehrere Agenten miteinander interagieren und/oder kooperieren müssen (vgl. [Alp19, S. 15]). Bei diesem Szenario spricht man dann auch von *Multi-Agenten-Systemen*.

Transfer-Lernen

Das Transfer-Lernen versteht sich weder als Lernen durch Ausprobieren, noch als Lernen durch Daten. Bei dieser Art des maschinellen Lernens wird vorhandenes Wissen über eine bestimmte Ausgangsaufgabe genutzt, um daraus neues Wissen für die Lösung einer neuen Zielaufgabe abzuleiten. Es wird also ein trainiertes Modell verwendet, welches eine spezifische Aufgabe (z. B. das Unterscheiden von Hunden und Katzen) lösen kann, um es anschließend auf eine andere Aufgabe (z. B. das Unterscheiden von Äpfeln und Birnen) anzuwenden. Somit kann dieses Vorgehen als Ergänzung zu den bisher vorgestellten Techniken des maschinellen Lernens verstanden werden.
Jedoch sind mit dem Transfer-Lernen auch verschiedene Problematiken verbunden. Zum einen bedarf es einer Metrik, mit welcher man die Ähnlichkeit von Ausgangs- und Zielaufgabe berechnen kann. Denn das Transfer-Lernen kann nur gelingen, wenn

sich beide Aufgaben ähneln. Als Vergleich könnte herangezogen werden, dass einem das Wissen über Fahrradfahren nicht dabei hilft, Hunde von Katzen zu unterscheiden. Dieses Phänomen wird in der Literatur auch als *negativer Transfer* bezeichnet (vgl. [TS10]).
Das Transfer-Lernen selbst kann zudem auch noch in verschiedene Techniken unterschieden werden. Eingangs wurde erwähnt, dass das Transfer-Lernen als Ergänzung bzw. Erweiterung herkömmlicher ML-Techniken verstanden werden kann. Daraus setzt sich dann auch die Unterscheidung verschiedener Techniken in diesem Gebiet zusammen (vgl. [TS10]): *Transfer-Lernen im Kontext von überwachten Lernen* und *Transfer-Lernen im Kontext von bestärkenden Lernen*. An dieser Stelle werden die einzelnen Techniken jedoch nicht weiter erläutert. Stattdessen wird der interessierte Leser auf die Arbeit [TS10] von Torrey und Shavlik verwiesen.
Zusammengefasst versucht man beim Transfer-Lernen Wissen zur Lösung einer Aufgabe auf eine andere (verwandte) Aufgabe zu übertragen, um dieses Wissen dann für die effizientere Lösung der neuen Aufgabe verwenden zu können. Auch der Mensch bedient sich häufig diesem Mechanismus, ohne dass es ihm bewusst ist. Wird er mit einer neuen Aufgabe konfrontiert, so wird oftmals bereits vorhandenes Wissen aus ähnlichen Situationen herangezogen, um abschätzen zu können, welches Vorgehen zu einem Erfolg führen könnte.

1.2.2 Evolutionäre Algorithmen

Evolutionäre Algorithmen (EAs) sind im Allgemeinen metaheuristische, populationsbasierte Optimierungsverfahren, welche durch die biologische Evolution inspiriert sind (vgl. [Kru+15, S. 157]). Die biologische Evolution hat mannigfaltige und komplexe Lebensformen auf der Erde generiert, die sich teils an schwierige äußere Bedingungen anpassen können (vgl. [Kru+15, S. 163]). Dieser Fakt legt nahe, dass man dieselben Prinzipien auch in der Informatik nutzen kann, um schwierige Optimierungsprobleme lösen zu können. Bei dieser Art von Optimierungsverfahren wird dementsprechend versucht, bestimmte Prinzipien der biologischen Evolution, wie z. B. Mutation und Selektion, auf Populationen von Lösungskandidaten anzuwenden, um somit eine hinreichende Lösung für ein Optimierungsproblem finden zu können. Der Rest dieses Abschnittes wird zunächst die Kernidee von evolutionären Algo-

rithmen erläutern und sich anschließend mit bestimmten EA-Verfahren (genetische Algorithmen, evolutionäre Strategien und genetische Programmierung) auseinandersetzen.

Kernidee von evolutionären Algorithmen

Auch wenn sich die einzelnen Varianten von evolutionären Algorithmen stark von einander unterscheiden können, verfolgen sie dennoch immer die gleiche Kernidee: die Verwaltung und Optimierung von Populationen von Lösungskandidaten. Dafür wird meist immer das gleiche Vorgehen genutzt:

1. Generieren einer initialen Population von Kandidaten.
2. Durchführen einer (Fitness)-Evaluation, welche den einzelnen Kandidaten einen Fitness-Wert (= Überlebensfähigkeit) zuordnet.
3. Selektion der fittesten (besten) Kandidaten.
4. Anwenden von evolutionären Operatoren auf den verbliebenen Kandidaten, um dadurch eine neue Generation (Population von neuen Kandidaten) zu erschaffen.

Die einzelnen EA-Techniken unterscheiden sich dann unter anderem durch unterschiedliche Selektion- und Evaluation-Mechanismen. Aber auch die Datenstruktur, auf der gearbeitet wird, kann unterschiedlicher Natur sein. Ein weiteres Differenzierungskriterium sind die verwendeten evolutionären Operatoren. So verwenden bestimmte EA-Techniken nur die Mutation, während andere zusätzlich noch die Rekombination verwenden. Auch die Art und Weise, wie der jeweilige Operator umgesetzt wird, kann sich dabei stark unterscheiden. (vgl. [Kru+15; CL18b])
Es ist noch zu erwähnen, dass in der Literatur keine einheitlichen Definitionen für die unterschiedlichen EA-Techniken existieren. Die hier gelieferte Unterscheidung ist zum Teil auf die historische Entwicklung des Forschungsgebietes, die Arbeit [CL18b] und das Wissen des Verfassers zurückzuführen.

Genetische Algorithmen

Bei Genetischen Algorithmen (GAs) handelt es sich um eine der ältesten Formen von EAs, welche auch aktuell stetig verwendet werden (vgl.[CL18b]). Es gibt viele verschiedene Varianten von genetischen Algorithmen in der Literatur, die unter anderem auch von ihrem Anwendungsgebiet abhängig sind. Jedoch haben alle den folgenden Kernalgorithmus gemein:

1. **Initialisierung:** Die Kandidaten der ersten Generation werden zufällig generiert
2. **Evolution:**
   a) **Evaluation:** Die Kandidaten der aktuellen Generation bekommen einen Fitness-Wert zugeordnet, je nachdem wie gut sie eine Aufgabe lösen bzw. wie gut sie sich in einer gegebenen Umgebung verhalten.
   b) **Selektion:** Die stärksten Kandidaten überleben, während die Schwächsten aussortiert werden.
   c) **Rekombination:** Die verbliebenen Kandidaten werden miteinander gekreuzt, um daraus eine neue Population zu erzeugen.
   d) **Mutation:** Eine zufällige Eigenschaft eines jeden Kandidaten der neuen Population wird verändert.

Was die einzelnen Varianten eines genetischen Algorithmus unterscheidet, sind meistens die für die Rekombination und Mutation verwendeten Metriken bzw. Operatoren. So muss z. B. für die Rekombination eine Distanzmetrik existieren, welche auf die verwendete Datenstruktur angewendet werden kann. Mithilfe dieser Distanzmetrik können dann die Unterschiede und Gemeinsamkeiten zwischen zwei Individuen berechnet werden, was notwendig ist, um diese miteinander zu kreuzen. (vgl. [CL18b])
Für die Mutation werden bei genetischen Algorithmen in der Regel unterschiedliche Wahrscheinlichkeitsverteilungen verwendet. Klassische Beispiele wären hier eine Gleich- oder Gauß-Verteilung, wobei die Letztere in der Praxis oft bevorzugt wird, da sie im Durchschnitt zu weniger disruptiven Veränderungen führt. (vgl. [CL18b])

Evolutionäre Strategien

Bei Evolutionären Strategien (ESs) handelt es sich um eine Klasse von Algorithmen, welche den genetischen Algorithmen recht ähnlich ist. Während frühere Varianten von evolutionären Strategien nur einzelne Kandidaten untersuchen und keine Rekombination innerhalb des Evolutionsvorganges verwenden, konvergieren moderne ES-Verfahren immer mehr gegen die Normen von genetischen Algorithmen (vgl. [CL18b]).

Jedoch existiert auch für die neueren ES-Techniken eine Eigenschaft, mit welcher sie sich von den genetischen Algorithmen differenzieren können. Im Gegensatz zu genetischen Algorithmen geschieht die Mutation bei ES-Verfahren nicht zufällig. Es wird eine Art *Strategie* verwendet, um die Mutation zu lenken. Damit soll sichergestellt werden, dass die Mutation bei ES-Verfahren zu einer besseren Lösung führt als die des vorigen Kandidaten. Ein weiterer Unterschied zu genetischen Algorithmen liegt darin, dass nicht nur eine Eigenschaft des Kandidaten verändert wird, sondern alle Eigenschaften auf Grundlage von bestimmten Eigenschaftsparametern. Diese Parameter haben Einfluss auf das Ausmaß der Veränderung und werden häufig während der Ausführung einer evolutionären Strategie mehrmals adaptiert. Dazu existieren in der Literatur verschiedene Verfahren, wie z. B. die 1/5-Regel, welche das Ausmaß der Veränderung, basierend auf der Anzahl erfolgreicher Mutationen, verringert oder erhöht. Es existieren auch Methoden, welche den Gradientenabstieg (welcher in Unterkapitel 3 noch näher erläutert wird) als Grundlage für die Adaption der Entscheidungsparameter verwendet. (vgl. [CL18b])

Die neueren ES-Verfahren nutzen auch andere Rekombinationsmechanismen als genetische Algorithmen und zusätzlich werden auch oftmals mehr als zwei Eltern verwendet. Ein Beispiel für die Rekombination innerhalb von ES-Verfahren ist, dass der neu generierte Kandidat die durchschnittlichen Werte aller Eigenschaften der Eltern-Kandidaten bekommt. Bei genetischen Algorithmen werden hingegen in der Regel nur zwei Eltern-Kandidaten verwendet und entweder nicht alle Eigenschaften der Eltern gekreuzt oder die Werte der Eigenschaften werden einer Wahrscheinlichkeitsverteilung entnommen. (vgl. [CL18b])

Genetische Programmierung

Die Genetische Programmierung (GP) ist im Gegensatz zu genetischen Algorithmen und evolutionären Strategien ein komplett anderer Ansatz. Dabei kann die genetische Programmierung allerdings auch als Variante von den GAs-Verfahren verstanden werden, welche sich dennoch stark unterscheidet.
Der Grundgedanke bei der genetischen Programmierung ist, dass man mithilfe einer Baumstruktur Programme oder mathematische Ausdrücke optimieren möchte. Dieses Verfahren lässt sich jedoch auch auf jegliche Aufgaben übertragen, welche ebenfalls als Baumstruktur modelliert werden können. Eine typische Anwendungsdomäne ist die *symbolische Regression*, bei welcher ein mathematischer Ausdruck gefunden werden soll, der die Daten eines bestimmten Datensatzes gut approximieren kann. Dabei macht die GP relativ wenig Annahmen über die mathematische Funktion, welche die Daten generiert hat und ermöglicht somit eine weitreichende Erkundung des Suchraumes von möglichen Lösungen. (vgl. [Kru+15, S. 243])
Der generelle Ablauf ist ungefähr der gleiche wie für evolutionäre Verfahren oder genetische Algorithmen. Es existiert ebenfalls eine Fitness-Metrik, ein Selektionsprozess wie auch bestimmte evolutionäre Operatoren (Mutation, Rekombination), welche auf die einzelnen Kandidaten einer Population angewendet werden.

Es wurde schon erwähnt, dass sich die einzelnen EA-Techniken in der verwendeten Datenstruktur unterscheiden können. So ist die *Neuroevolution* ebenfalls ein Forschungsgebiet, welches den evolutionären Algorithmen entspringt. Dabei werden unterschiedliche EA-Techniken verwendet und so angepasst, dass sie in Kombination mit neuronalen Netzen verwendet werden können, um deren Parameter, wie z. B. Verbindungsgewichte, Aktivierungsfunktionen oder Verbindungsmuster, generieren zu können. An dieser Stelle wird jedoch nicht weiter auf neuronale Netze und die Neuroevolution eingegangen, da beide Themengebiete in den späteren Kapitel 2 und 4 erläutert werden.

1.3 Ziele der Arbeit

Das Ziel der Arbeit besteht darin, zu untersuchen, welche Techniken sich für das Intra-Life-Learning eignen. Dabei war die Auswahl auf Algorithmen der Neuroevolution beschränkt. Im Verlauf dieser Arbeit haben sich jedoch neue Einblicke ergeben, weswegen das betrachtete Forschungsfeld erweitert wurde. Dadurch wurden auch die Zielstellungen dieser Arbeit leicht angepasst. Dabei ist zu erwähnen, dass keine der alten Zielstellungen verloren gegangen ist, sondern nur um neue Aspekte erweitert wurde.

Das erste Ziel dieser Arbeit besteht darin, zunächst die Grundlagen im Bereich der neuronalen Netze darzustellen, um somit ein erstes Verständnis für diese Art von Modellen beim Leser zu erreichen. Anschließend soll der *State-of-the-Art* im Bereich der Lernalgorithmen für neuronale Netze erläutert werden. An dieser Stelle wurde der Umfang der Arbeit erweitert. Anstatt nur Algorithmen der Neuroevolution zu untersuchen, werden nun zusätzlich auch Algorithmen des maschinellen Lernens betrachtet.

Aufbauend darauf besteht ein weiteres Ziel darin, die einzelnen Lernalgorithmen und Techniken der Neuroevolution im Kontext des Intra-Life-Learning miteinander zu vergleichen, um ein Fazit ziehen zu können, welche Ansätze für das Intra-Life-Learning geeignet sind. Im Verlauf der Bearbeitung dieses Themas hat sich jedoch ergeben, dass herkömmliche Algorithmen aus dem Bereich der Neuroevolution oder des maschinellen Lernens nicht für das Intra-Life-Learning geeignet sind. Deswegen wurde der Rahmen dieser Arbeit an dieser Stelle abermals erweitert, sodass der eigentliche Vergleich nun verschiedene Techniken des Meta-Lernens, der synaptischen Plastizität und der Neuromodulation umfasst. Diese unterschiedlichen Techniken sind dann ebenfalls genauer zu analysieren und strukturieren, sodass sie voneinander abgegrenzt und kategorisiert werden können. Die Kategorisierung dient dann letztendlich als Grundlage für den eigentlichen Vergleich, welcher die Stärken und Schwächen der einzelnen Algorithmen in Bezug auf das Intra-Life-Learning darlegen soll.

Eine letzte Zielstellung besteht dann darin, die aus dem Vergleich resultierenden Erkenntnisse zu nutzen, um eine Empfehlung für das Intra-Life-Learning geben zu können. Auch im Zusammenhang mit diesem Ziel wurde der Rahmen dieser Arbeit

erweitert: Anstatt nur eine Empfehlung zu geben, wird ein eigenes Konzept für die Realisierung eines Algorithmus für das Intra-Life-Learning vorgestellt. Als Grundlage dafür werden dann ebenso die Erkenntnisse aus dem vorher durchgeführten Vergleich verwendet, da dieser eine solide Basis darstellt, mithilfe derer die Potenziale der einzelnen Ansätze für das Intra-Life-Learning identifiziert werden können.

1.4 Aufbau der Arbeit

Der Aufbau dieser Arbeit folgt im Grunde genommen den in Unterkapitel 1.3 genannten Zielen. Somit werden in Kapitel 2 die Grundlagen im Bereich der neuronalen Netze gelegt. Dabei wird auf den Aufbau und die Funktionsweise einzelner Neuronen wie auch neuronaler Netze insgesamt eingegangen. In diesem Zusammenhang führt der Autor eine eigene Notation und Definitionen ein, um dem Leser den Einstieg in diese Thematik zu erleichtern und gegebenenfalls auf Wissen aus diesem Kapitel zurückgreifen zu können. Zusätzlich werden auch noch häufig verwendete Aktivierungsfunktionen und spezielle Typen von neuronalen Netzen vorgestellt.
Das Kapitel 3 wird dann auf spezielle Lernalgorithmen für neuronale Netze eingehen. Dafür werden zunächst einige Grundlagen im Bezug auf Fehlerfunktionen und Analysis gelegt. Darauf aufbauend wird dann der Backpropagation-Algorithmus anhand eines Beispiels erläutert, um auch hier dem Leser ein tiefes Verständnis dieser Thematik vermitteln zu können.
In Kapitel 4 wird dann der *State-of-the-Art* im Bereich der Neuroevolution vorgestellt. Dazu wird zunächst ein bekannter Algorithmus detailliert erläutert und die Skalierbarkeit der Neuroevolution im Allgemeinen diskutiert. Anschließend werden in diesem Kapitel die unterschiedlichen Forschungsgebiete der Neuroevolution näher betrachtet und bestimmte Techniken ebenfalls im Ansatz erklärt.
Das Kapitel 5 stellt dann das Herzstück dieser Arbeit dar. Während die vorherigen Kapitel die Grundlage für das Verständnis legen und in die verschiedenen Bereiche einführen, wird dieses Kapitel all dieses Wissen zusammenbringen. Dabei definiert der Autor zunächst, worum genau es sich beim Intra-Life-Learning handelt und aus welchen unterschiedlichen Lernszenarien es sich zusammensetzt. Dabei wird auch das Meta-Lernen durch den Autor formalisiert, um dem Leser das Verständnis in

den folgenden Unterkapiteln zu vereinfachen. Anschließend werden dann die einzelnen Techniken des klassischen Meta-Lernens, der synaptischen Plastizität und der Neuromodulation diskutiert, miteinander verglichen und in Bezug auf das Intra-Life-Learning evaluiert. Dazu wird jeder Abschnitt wie auch jedes Unterkapitel einen Fazit-Teil enthalten, der den jeweiligen Ansatz bzw. die jeweilige Technik beurteilt. Abschließend wird das letzte Unterkapitel die Haupterkenntnisse dieses Kapitels zusammenfassen.
Das Kapitel 6 enthält dann die Konzeption eines vom Autor selbst entwickelten Algorithmus für das Intra-Life-Learning. Dabei wird zunächst erläutert, welche Anforderungen bei der Konzeption zu beachten waren. Anschließend wird beschrieben, wie die Erkenntnisse aus Kapitel 5 ihre Anwendung bei dem vorgestellten Algorithmus finden. Dann wird die Kernidee des Algorithmus erläutert, bevor darauf eingegangen wird, wie der Algorithmus parallelisiert werden kann.
Abschließend werden in Kapitel 7 zunächst die Erkenntnisse dieser Arbeit zusammengefasst und die eigene Forschungsleistungen aufgezeigt. Aufbauend darauf kann ein Ausblick für das gesamte Thema gegeben werden, welcher dem Leser weitere mögliche Arbeiten in diesem Gebiet der künstlichen Intelligenz aufzeigen soll.

2 Grundlagen von neuronalen Netzen

Ein neuronales Netz (NN), auch oft als *künstliches neuronales Netz* bezeichnet, ist ein informationsverarbeitendes System, welches aus einer Vielzahl einfacher Einheiten (den Neuronen) besteht, welche sich über gewichtete Verbindungen (den Synapsen) Informationen über ihre Aktivierungszustände zuschicken (vgl. [Kru+15, S. 7]). Es ist auf die Arbeit „A logical calculus of the ideas immanent in nervous activity“ [MP43]] von *Warren McCulloch* und *Walter Pitts* aus dem Jahr 1943 zurückzuführen, welche die Grundlage für das heutige Forschungsgebiet der neuronalen Netze gelegt hat. Dabei beschreiben die beiden Mathematiker auf Basis des sogenannten *McCulloch-Pitts-Neurons* das erste Mal ein neurologisches Netzwerk und zeigen in diesem Zusammenhang, dass auch einfache neuronale Netze rein theoretisch jegliche arithmetische oder logische Funktion berechnen könnten.
Die Motivation zum Forschungsgebiet der neuronalen Netze ist auch teilweise durch die allgemeine Analogie zum Nervensystem und speziell dem Gehirn von Mensch und Tier motiviert (vgl. [Kru+15, S. 9]). Denn auch im Nervensystem findet die Informationsverarbeitung durch eine Vielzahl an Nervenzellen statt, welche den Grad ihrer Erregung (= Aktivierungszustand) über Nervenfasern an andere Zellen weitergeben. Die einzelnen Zellen sind dabei eher simpler Natur. Erst durch die Verknüpfung der einzelnen Zellen entsteht eine komplexe Struktur (also das Nervensystem), welche letztendlich zu intelligentem Verhalten führen kann.
Heutzutage werden neuronale Netze für viele verschiedene Aufgaben wie z. B. Klassifikation oder Clustering eingesetzt und sind somit ein mögliches Modell im Zusammenhang mit dem maschinellen Lernen. Somit können sie für die Speicherung von Wissen verwendet werden. Ihre Besonderheit gegenüber anderen ML-Modellen besteht dabei in ihrer universellen Anwendbarkeit und Ausdrucksstärke. Neuronale Netze können für jegliche Teilgebiete des ML verwendet werden (überwachtes, un-

überwachtes, bestärkendes und Transfer-Lernen). Andere Modelle sind meist eher auf eines der genannten Teilgebiete limitiert.
Da die Arbeit das Intra-Life-Learning im Zusammenhang mit neuronalen Netzen untersucht, wird sich das Kapitel den einzelnen Bestandteilen eines neuronalen Netzes widmen und diese jeweils im Detail erläutern. Denn tiefgreifendes Verständnis über neuronale Netze ist grundlegend für den Rest dieser Arbeit. In diesem Zusammenhang wird zunächst der Aufbau und die Funktionsweise eines einzelnen Neurons behandelt. Anschließend erfolgt die Erläuterung zu Aufbau und Funktionsweise eines neuronalen Netzes. Die restlichen Unterkapitel beschäftigen sich dann mit einzelnen Aspekten, wie z. B. häufig verwendeten Aktivierungsfunktionen. Abgerundet wird das Kapitel durch einen Überblick über die gängigen Typen von neuronalen Netzen.
Für die Notationen und Berechnungen wird überwiegend die lineare Algebra verwendet und sich an der Arbeit [LRU20] orientiert. Auch moderne Deep-Learning-Frameworks, wie z. B. TensorFlow, PyTorch oder auch Caffe nutzen Operatoren der linearen Algebra, um mittels Grafikprozessoren (GPUs) die Berechnungen, welche für neuronale Netze notwendig sind, drastisch zu beschleunigen (vgl. [LRU20, S. 530]). Deswegen und wegen ihrer Kompaktheit wird sie auch in diesem Kapitel verwendet.
An dieser Stelle sei auch erwähnt, dass es sich bei allen Vektoren (ob in Definitionen oder im Fließtext) um Spaltenvektoren handelt. An manchen Stellen (wie z. B. im Fließtext, teilweise aber auch in Definitionen) werden diese Vektoren aus Platzgründen als Zeilenvektoren geschrieben. Trotz der anderen Darstellungsweise handelt es sich bei diesen Vektoren dennoch um Spaltenvektoren.
Auf bestimmte Teilgebiete von neuronalen Netzen kann nur überblicksartig eingegangen werden, da eine detaillierte Betrachtung aller Facetten von neuronalen Netzen den Rahmen dieser Arbeit sprengen würde.

2.1 Aufbau und Funktionsweise von Neuronen

Das Neuron ist der fundamentale Bestandteil von Neuronalen Netzen (vgl. [Kru+15, S. 9]). Dabei besteht sein biologisches Vorbild im Groben aus drei Bestandteilen: dem *Zellkörper*, den *Dendriten* und dem *Axon* (vgl. [Kru+15, S. 10]). Exemplarisch ist ein solches biologisches Neuron in Abbildung 2.1 zu sehen.

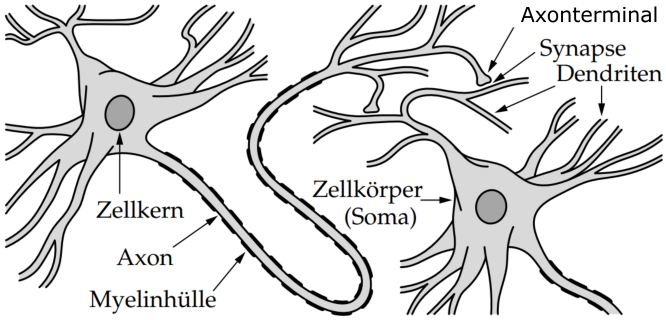


Abbildung 2.1: Ein biologisches Neuron mit seinen Bestandteilen (in Anlehnung an [Kru+15]).

Die Dendriten sind dabei für das Sammeln der eingehenden Signale verantwortlich. Der Zellkörper verarbeitet diese Signale intern und gibt den Output über das Axon an andere Neuronen weiter. Dabei verzweigt sich das Axon in mehrere *Axonterminale*, welche dann mit den Dendriten anderer Zellen über *Synapsen* in Verbindung treten. (vgl. [Kru+15, S. 10])
Die exakte Arbeitsweise eines biologischen Neurons wird an dieser Stelle nicht weiter erläutert, da diese Arbeit sich thematisch auf die künstlichen Äquivalente bezieht. Diese sind in ihrer Struktur wie auch Funktionsweise an das biologische Neuron angelehnt.
In Abbildung 2.2 ist das Modell eines (künstlichen) Neurons zu sehen. Dabei stellen die grünen Elemente das Äquivalent zum Axon dar. Eingehende Signale $x_0, x_1, ..., x_n$ werden durch Verbindungen aufgenommen, wobei jede Verbindung ein Gewicht $w_0, w_1, ..., w_n$ besitzt. Anschließend werden diese Signale durch die Übertragungsfunktion gesammelt und zu einem einzelnen Signal *net* gebündelt. Die blaue Komponente stellt dann die Analogie zum Zellkörper dar. Die Aktivierungsfunktion f_{act} eines Neurons ist somit für seine interne Verarbeitung zuständig, wodurch das jeweilige Neuron seinen internen Zustand a bekommt. Die roten Komponenten stellen

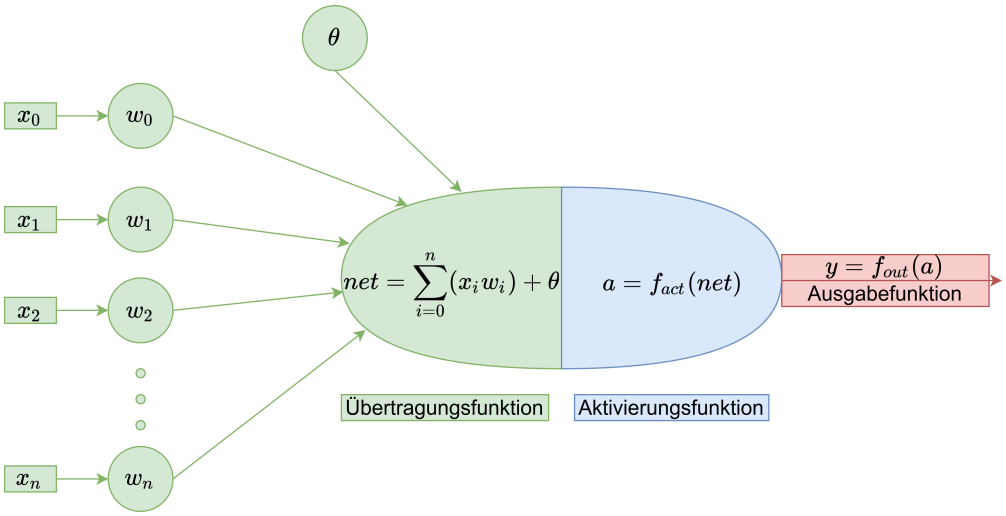


Abbildung 2.2: Modell eines Künstlichen Neurons.

dann das Axon dar. Über die Ausgabefunktion f_{out} berechnet das Neuron seine Ausgabe y, welche über die roten Kanten weitergegeben wird. Dementsprechend lässt sich daraus die Definition eines Neurons herleiten:

Definition 2.1 (Neuron)
Ein Neuron ist ein 5-Tupel $(\mathbf{w}, \theta, f_{net}, f_{act}, f_{out})$ *mit:*

1. *Gewichts-Vektor* $\mathbf{w} = [w_1, w_2, ..., w_n]$*: Das Gewicht* $w_i \in \mathbb{R}$ *beschreibt, ob ein Input* x_i *einen hemmenden* $(w_i < 0)$ *oder anregenden Einfluss* $(w_i > 0)$ *auf das jeweilige Neuron besitzt. Für eine nicht existente Verbindung zwischen dem Input* x_i *und dem jeweiligen Neuron ist* $w_i = 0$*.*

2. *Bias* $\theta \in \mathbb{R}$*: Der Bias stellt den Schwellenwert eines Neurons dar. Er ist unter anderem mitentscheidend für den Aktivierungszustand eines Neurons.*

3. *Übertragungsfunktion* f_{net}*: Diese Funktion berechnet den Netz-Input net eines Neurons.*

4. *Aktivierungsfunktion* f_{act}*: Diese Funktion berechnet den Aktivierungszustand* a *eines Neurons.*

5. *Ausgabefunktion* f_{out}*: Diese Funktion berechnet den Output* y *eines Neurons.*

Der Input (also die eingehenden Signale) eines Neurons besteht aus einem Vektor von numerischen Werten. Dieser Input kann zum einen aus der Beobachtung der Umwelt resultieren oder aber auch aus dem Output anderer Neuronen.

Definition 2.2 (Input-Vektor eines Neuron)
Der Input eines Neurons ist ein Vektor $\mathbf{x} = [x_1, x_2, ..., x_n]$*, dessen einzelne Elemente* $x_i \in \mathbb{R}$ *die Input-Werte des Neurons darstellen, wobei* n *die Anzahl der Inputs und* $1 \leq i \leq n$ *ist.*

Die Informationsverarbeitung eines Neurons erfolgt in Abbildung 2.2 von links nach rechts. Dabei wird zunächst der Netz-Input des Neurons berechnet. Dafür wird meist die *gewichtete Summe* als Übertragungsfunktion genutzt, sodass sich für den Netz-Input eines Neurons folgende Definition ergibt:

Definition 2.3 (Netz-Input eines Neurons)
Sei $\mathbf{x} = [x_1, x_2, ..., x_n]$ *der Input-Vektor,* $\mathbf{w} = [w_1, w_2, ..., w_n]$ *der Gewichtsvektor und* θ *der Bias eines Neurons. Dann wird der Netz-Input dieses Neurons wie folgt berechnet:*

$$net = f_{net}(\mathbf{x}, \mathbf{w}, \theta) = \sum_{i=1}^{n}(x_i w_i) + \theta$$

Es ist ersichtlich, dass der Neuronen-Input durch die gewichtete Summe des Input-Vektors mit dem Gewichts-Vektor und der zusätzlichen Addition des Bias berechnet wird. Der Bias repräsentiert dabei den sogenannten *Schwellenwert* eines Neurons. Dieser Wert hat die Aufgabe festzulegen, ab wann ein Neuron *aktiv* wird oder *inaktiv* bleibt bzw. ein negatives Signal weitergibt. Der Schwellenwert wird in der Praxis oft mit einem negativen numerischen Wert versehen, sodass das Neuron erst ab einer bestimmten Stärke der eingehenden Signale *feuert*, also aktiviert wird. Der daraus resultierende gebündelte Netz-Input *net* kann dann verwendet werden, um den internen Zustand eines Neurons zu berechnen. Dieser Zustand wird auch als *Aktivierungszustand* eines Neurons bezeichnet und durch die *Aktivierungsfunktion* berechnet:

Definition 2.4 (Aktivierungszustand)
Sei net der Netz-Input. Dann wird der Aktivierungszustand a eines Neurons wie folgt berechnet:

$$a = f_{act}(net)$$

Als Aktivierungsfunktion kann generell jegliche mathematische Funktion verwendet werden. In der Praxis ergeben sich jedoch öfter Beschränkungen in der Wahl der jeweiligen Aktivierungsfunktion. Auf diesen Sachverhalt wird allerdings in Unterkapitel 2.3 näher eingegangen. An dieser Stelle ist es zunächst ausreichend zu wissen, dass sich der Aktivierungszustand aus einer Funktion f_{act} ergibt, welche als Eingabe den jeweiligen Neuronen-Input bekommt. Aus diesem Aktivierungszustand kann anschließend mithilfe der Ausgabefunktion der endgültige Output-Wert eines Neurons berechnet werden:

Definition 2.5 (Neuronen-Output)
Sei a Aktivierungszustand eines Neurons. Dann wird der Output y eines Neurons wie folgt berechnet:

$$y = f_{out}(a)$$

Üblicherweise wird die Identitätsfunktion als Ausgabefunktion verwendet, sodass der Aktivierungszustand eines Neurons auch gleichzeitig seinen Output darstellt ($y = a$). Allerdings kann auch hier eine andere Funktion definiert werden, welche den Aktivierungszustand eines Neurons anderweitig manipuliert und so in den Output umwandelt.
Zusammengefasst erfolgt die Informationsaufnahme eines künstlichen Neurons durch die Summierung der gewichteten Eingaben und der zusätzlichen Addition des Bias. Die eigentliche Informationsverarbeitung geschieht über die Aktivierungsfunktion, welche den Aktivierungszustand eines Neurons bestimmt. Der finale Output wird dann durch die jeweilige Ausgabefunktion berechnet. Diese Darstellung eines künstlichen Neurons ist auf die *McCulloch-Pitts-Zelle* zurückzuführen (die allerdings in ihrer Struktur wesentlich simpler als die hier gewählte Darstellungsweise ist), welche ebenfalls alle eingehenden Signale summiert und anschließend einen Schwellenwert anwendet (vgl. [MP43]).
Das nächste Unterkapitel wird den Aufbau und die Funktionsweise von neuronalen Netzen erläutern. Dabei wird auf dem Wissen aus diesem Unterkapitel aufgebaut, sodass dem Leser eine ganzheitliche Sicht über neuronale Netze im Allgemeinen vermittelt werden kann.

2.2 Aufbau und Funktionsweise Neuronaler Netze

Ein neuronales Netz besteht aus einer Menge von simplen Einheiten (den Neuronen) und gerichteten, gewichteten Verbindungen (den Synapsen bzw. Kanten), welche die einzelnen Neuronen miteinander verbinden. Die Stärke der Verbindung zweier Neuronen wird durch ihr jeweiliges Gewicht repräsentiert. Allgemein sind neuronale Netze in sogenannten *Neuronen-Schichten* (engl. *layer*) organisiert. In Abbildung 2.3 ist ein solches Netzwerk mit seinen unterschiedlichen Schichten zu sehen. Dabei repräsentiert die *Input-Schicht* den jeweiligen *Input-Vektor* und die *Output-Schicht* den *Output-Vektor* des neuronalen Netzes. Dazwischen liegt die sogenannte *verdeckte Schicht* eines neuronalen Netzes, welche wiederum aus mehreren einzelnen Neuronen-Schichten bestehen kann. Während die Anzahl der Neuronen in Input- und Output-Schicht durch den jeweiligen In- und Output festgelegt ist, können die restlichen Schichten eines neuronalen Netzes aus beliebig vielen Neuronen bestehen. (vgl. [LRU20, S. 525-526])

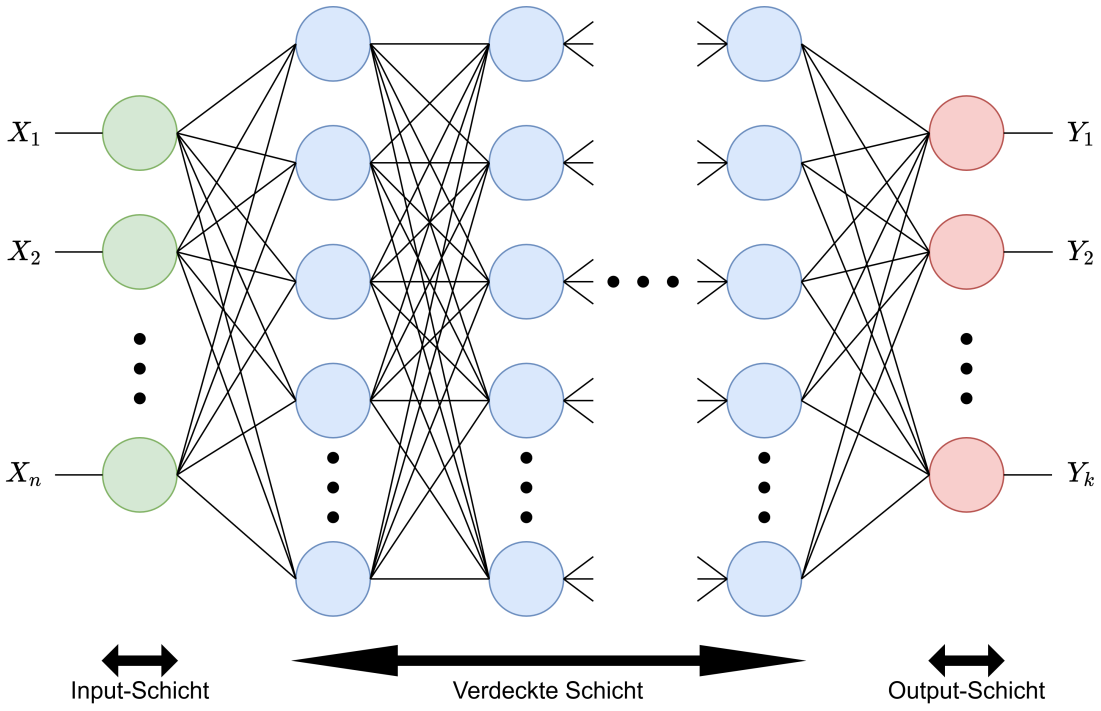


Abbildung 2.3: Ein neuronales Netz mit mehreren Schichten.

Wenn jedes Neuron eine simple (mathematische) Funktion implementiert und diese Neuronen über Verbindungen in einem NN kombiniert werden, kann man ein neuronales Netz auch als einen Funktionsapproximator interpretieren. Dies wird vor allem später im Zusammenhang mit dem Lernen, welches in Kapitel 3 näher beschrieben wird, eine wichtige Rolle spielen.
Die Verbindungen der einzelnen Neuronen der unterschiedlichen Schichten können auf unterschiedliche Weisen realisiert werden. In Abbildung 2.1 sind die Neuronen der verschiedenen Schichten z. B. voll verbunden (fully-conncected) und die Verbindungen sind ausschließlich *vorwärts* (in Abbildung 2.3 von links nach rechts) gerichtet. Diese Art von neuronalen Netzen wird *vorwärtsgerichetes neuronales Netz* oder auch Feed-Forward-Netz (FFN) genannt. Auf weitere Verbindungsarten für Neuronen-Schichten wird an dieser Stelle nicht weiter eingegangen. Allerdings ist es wichtig zu wissen, dass es neben voll-verbundenen Neuronen-Schichten auch noch andere Verbindungsmuster gibt. Dies wird im späteren Unterkapitel 2.4 noch eine wichtige Rolle spielen.
Ein neuronales Netz kann also wie folgt definiert werden:

Definition 2.6 (Neuronales Netz)
Ein neuronales Netz ist ein geordnetes 3-Tupel $(\mathbf{X}, H, \mathbf{Y})$ *mit:*

- *Input-Vektor* $\mathbf{X} = [X_1, X_2, ..., X_n]$*, wobei* $X_i \in \mathbb{R}$*.*
- *Menge von Neuronen-Schichten* $H = \{H_1, H_2, ..., H_m\}$*.*
- *Output-Vektor* $\mathbf{Y} = [Y_1, Y_2, ..., Y_k]$*, wobei* $k \in \mathbb{N}$ *die Anzahl der Outputs ist und* $Y_j \in \mathbb{R}$*.*

Somit besteht ein neuronales Netz also aus einem Input-Vektor, einer Menge von Neuronen-Schichten und einem Output-Vektor. Dabei bekommt jedes Neuron der Input-Schicht (in Abbildung 2.3 die grünen Neuronen) jeweils einen Input-Wert des Input-Vektors zugewiesen. Die Input-Neuronen besitzen dabei keine interne Verarbeitung. Sie geben ihren Input unmanipuliert über die gewichteten Verbindungen an die Neuronen der ersten verdeckten Schicht (in Abbildung 2.3 die erste blaue Schicht) weiter. Die Neuronen der verdeckten und der Output-Schicht besitzen jedoch eine interne Verarbeitung. Die Funktionsweise der einzelnen Neuronen wurde

bereits in Unterkapitel 2.1 erläutert und soll hier auch nicht weiter behandelt werden. Stattdessen wird sich der Rest dieses Unterkapitels der Funktionsweise bzw. Informationsverarbeitung eines kompletten neuronalen Netzes und somit seiner einzelnen Neuronen-Schichten widmen. Dazu wird zunächst definiert, was genau unter einer Neuronen-Schicht zu verstehen ist:

Definition 2.7 (Neuronen-Schicht)
Die Neuronen-Schicht eines neuronalen Netzes ist ein 5-Tupel $(\mathbf{b}, \mathbf{W}, f_{NET}, f_{ACT}, f_{OUT})$ *mit:*

1. *Bias-Vektor* $\mathbf{b} = [b_1, b_2, ..., b_m]$*: Dieser Vektor enthält den Bias aller Neuronen der Neuronen-Schicht.*
2. *Gewichts-Matrix* $\mathbf{W}$*: Enthält die Gewichte der eingehenden Verbindungen aller Neuronen der Neuronen-Schicht.*
3. *Schicht-Übertragungsfunktion* f_{NET}*: Funktion, welche die Netz-Inputs aller Neuronen der Neuronen-Schicht berechnet.*
4. *Schicht-Aktivierungsfunktion* f_{ACT}*: Funktion, welche die Aktivierungszustände aller Neuronen der Neuronen-Schicht berechnet.*
5. *Schicht-Ausgabefunktion* f_{OUT}*: Funktion, welche die Outputs aller Neuronen der Neuronen-Schicht berechnet.*

Die Definition einer Neuronen-Schicht ist somit ähnlich zu Definition 2.1 eines Neurons. Jedoch sind die einzelnen Elemente einer Neuronen-Schicht unterschiedlich zu denen eines Neurons. Im Folgenden werden die einzelnen Elemente detailliert erklärt. Das Erste, was dementsprechend zu definieren wäre, ist der Input einer Neuronen-Schicht. In Abbildung 2.3 ist zu sehen, dass die Schichten eines FFN sequenziell verschaltet sind. Dadurch bezieht jede Schicht (ausgenommen der Input-Schicht, welche nicht als herkömmliche Neuronen-Schicht behandelt wird) eines neuronalen Netzes seinen Input aus mindestens einer vorgeschaltete Schicht, sodass sich für den jeweiligen Input-Vektor einer Neuronen-Schicht folgende Definition ergibt:

Definition 2.8 (Input-Vektor einer Neuronen-Schicht)
Sei H_l eine Neuronen-Schicht und $\mathbf{Y}_{H_{l-1}}$ der Output-Vektor der vorgeschalteten Neuronen-Schicht H_{l-1}. Dann gilt für den Input-Vektor $\mathbf{X}_{H_l}$, dass $\mathbf{X}_{H_l} = \mathbf{Y}_{H_{l-1}}$, sodass $\mathbf{X}_{H_l} = [X_1, X_2, ..., X_n]$, wobei $n \in \mathbb{N}$ die Anzahl der Neuronen in Schicht H_{l-1} ist.

Das bedeutet also, dass der Output der Neuronen der vorgeschalteten Schicht H_{i-1} gleichzeitig der Input für die Schicht H_i darstellt. Für die Definition der Gewichts-Matrix wird nochmals auf Definition 2.1 verwiesen. Dort ist festgelegt, dass jedes Neuron einen Gewichts-Vektor besitzt. Die Gewichts-Matrix einer Neuronen-Schicht kann nun also als Kombination der Gewichts-Vektoren der Neuronen der jeweiligen Schicht aufgefasst werden:

Definition 2.9 (Gewichts-Matrix einer Neuronen-Schicht)
Sei H_l eine Neuronen-Schicht, $\mathbf{X}_{H_l} = [X_1, ..., X_n]$ ihr Input-Vektor. Dann ist die Gewichtsmatrix $\mathbf{W}_{H_l}$ definiert als $n \times m$-Matrix:

$$\mathbf{W}_{H_l} = \begin{bmatrix} w_{1,1} & \cdots & w_{1,m} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,m} \end{bmatrix}$$

wobei $w_{i,j} \in \mathbb{R}$ das Gewicht der Verbindung von Neuron i zu Neuron j darstellt, mit:

- $n \in \mathbb{N}$ = *Anzahl der Neuronen der vorgeschalteten Schicht von H_{l-1}*
- $m \in \mathbb{N}$ = *Anzahl der Neuronen der Schicht H_l*
- $1 \leq i \leq n$
- $1 \leq j \leq m$

Die j-te Spalte einer Gewichtsmatrix entspricht somit dem Gewichtsvektor $\mathbf{w}_j$ des j-ten Neurons einer Neuronen-Schicht.

Laut Definition 2.3 wird der Netz-Input eines Neurons über die gewichtete Summe und der zusätzlichen Addition des Bias berechnet. Allerdings lässt sich auch mittels linearer Algebra-Operationen der Netz-Input eines Neurons j berechnen:

$$(\mathbf{w}_j^T * \mathbf{x}_j) + b_j = (w_1, ..., w_n)^T * \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + b_j = w_1 x_1 + ... + w_n x_n + b_j$$

Dieses Vorgehen kann nun auch für die Berechnung des Netz-Inputs einer Neuronen-Schicht verwendet werden:

Definition 2.10 (Netz-Input einer Neuronen-Schicht)
Sei H_l eine Neuronen-Schicht mit einem Input-Vektor $\mathbf{X}_{H_l} = [X_1, X_2, ..., X_n]$, einer $n \times m$ Gewichtsmatrix $\mathbf{W}_{H_l}$ und einem Bias-Vektor $\mathbf{b}_{H_l} = [b_1, b_2, ..., b_m]$. Dann ist der Netz-Input ein Vektor $\mathbf{NET}_{H_l} = [NET_1, NET_2, ..., NET_m]$ der wie folgt berechnet wird:

$$\mathbf{NET}_{H_l} = f_{NET}(\mathbf{W}_{H_l}, \mathbf{X}_{H_l}, \mathbf{b}_{H_l}) = \mathbf{W}_{H_l}^T \mathbf{X}_{H_l} + \mathbf{b}_{H_l}$$

Somit ist der Netz-Input einer Neuronen-Schicht ebenfalls ein Vektor mit m Elementen, wobei m hierbei der Anzahl der Neuronen der jeweiligen Schicht entspricht. Folgende Definition beinhaltet dann die Berechnung des Aktivierungszustands einer Schicht:

Definition 2.11 (Aktivierungszustands-Vektor)
Sei H_l eine Neuronen-Schicht und $\mathbf{NET}_{H_l} = [NET_1, NET_2, ..., NET_m]$ der dazugehörige Netz-Input. Dann ist der Aktivierungszustand ein Vektor $\mathbf{A}_{H_l} = [A_1, A_2, ..., A_m]$ der wie folgt berechnet wird:

$$\mathbf{A}_{H_l} = f_{ACT}(\mathbf{NET}_{H_l})$$

Der Output einer Schicht wird dann durch die Anwendung ihrer Ausgabefunktion auf den jeweiligen Aktivierungszustands-Vektor berechnet:

Definition 2.12 (Neuronen-Schicht-Output)
Sei H_l eine Neuronen-Schicht und $\mathbf{A}_{H_l} = [A_1, A_2, ..., A_m]$ der dazugehörige Aktivierungszustandsvektor. Dann ist der Output ein Vektor $\mathbf{Y}_{H_l} = [Y_1, Y_2, ..., Y_m]$ der wie folgt berechnet wird:

$$\mathbf{Y}_{H_l} = f_{OUT}(\mathbf{A}_{H_l})$$

Ein Sonderfall ist, wenn die jeweilige Schicht H_l nur aus einem einzelnen Neuron besteht. So erhält man für $\mathbf{W}_{H_l}$ einen Spaltenvektor. Dadurch behält Definition 2.10 ihre Gültigkeit, da auch hier die Gewichtsmatrix transponiert wird. Dabei wird das Produkt eines Zeilen- mit einem Spalten-Vektor gebildet, was letztendlich in einem Skalar resultiert. Dadurch erhält man dementsprechend für Netz-Input IN_{H_l}, Aktivierungszustand A_{H_l} und Output Y_{H_l} jeweils einen Skalar, anstatt eines Vektors.
Diese Art der Berechnung kann auf neuronale Netze jeglicher Art übertragen werden, unabhängig davon, welche Aktivierungsfunktionen verwendet werden, wie viele Schichten das jeweilige Netz hat, wie viele Neuronen in einer Schicht vorhanden sind oder wie groß die jeweiligen Input- und Output-Vektoren des neuronalen Netzes sein sollten.
Zusammengefasst kann man sagen, dass die Funktionsweise einer Neuronen-Schicht und somit eines NN äquivalent zu der eines Neurons ist, bis auf den Unterschied, dass bei Neuronen-Schichten mit Vektoren und Matrizen gearbeitet wird, statt mit einzelnen Skalaren und Vektoren.

2.3 Aktivierungsfunktionen

In Unterkapitel 2.1 wurde beschrieben, dass Neuronen mit unterschiedlichen Aktivierungsfunktionen belegt werden können. Aus Unterkapitel 2.2 ist bekannt, dass es in der Praxis geläufig ist, die Neuronen eines neuronalen Netzes in Schichten zu organisieren. Neuronen derselben Schicht verwenden dabei oftmals dieselbe Aktivierungsfunktion, weswegen die Aktivierungsfunktionen meist für die Schichten und nicht für einzelne Neuronen gewählt werden. Das spielt insofern eine Rolle, dass Aktivierungsfunktionen somit nicht nur auf dem Netz-Input einzelner Neuronen (Skalare), sondern auch auf dem Netz-Input ganzer Schichten (Vektoren) angewendet werden. Da auch schon im letzten Unterkapitel 2.2 auf der Abstraktionsebene der

Neuronen-Schichten gearbeitet wurde, werden auch hier die Aktivierungsfunktionen bezüglich ganzer Neuronen-Schichten betrachtet. In der Schlussfolgerung bedeutet dies, dass der Input einer Aktivierungsfunktion vektorieller Natur ist (siehe Definition 2.11). Generell sei aber gesagt, dass alle hier vorgestellten Funktionen ebenso für einzelne Neuronen und somit auch auf Basis von Skalaren als Input funktionieren. In Unterkapitel 2.1 wurde schon erwähnt, dass die Wahl der Aktivierungsfunktion generell freisteht, sich in der Praxis jedoch einige Beschränkungen ergeben können. Dies resultiert in folgenden Anforderungen (vgl. [LRU20, S. 531]):

1. *Differenzierbarkeit*: Ein Funktion sollte an jeder Stelle differenzierbar sein.
2. *Stetigkeit*: Eine Funktion sollte stetig sein. Das bedeutet, dass eine geringe Änderung im Input auch eine ungefähr äquivalente Änderung im Output nach sich zieht. Ist eine Funktion differenzierbar, so ist sie auch stetig.
3. *Beschränktheit*: Die Ableitung einer Funktion darf nicht über dem erwarteten Input-Bereich sättigen (z. B. sehr klein werden, gegen null tendieren) oder explodieren (z. B. sehr groß werden, gegen unendlich tendieren), da dies zu dem Problem der numerischen Instabilität führen würde.

Die Kriterien ergeben sich aus dem Sachverhalt, dass neuronale Netze bevorzugt mit Verfahren trainiert werden, welche den *Gradientenabstieg* verwenden. Auf diese Verfahren wird jedoch nicht an dieser Stelle, sondern in Kapitel 3 näher eingegangen. Im Folgenden werden einige Aktivierungsfunktionen vorgestellt. Dabei wird sich an der Arbeit [LRU20] von *Leskovec et al.* orientiert, welche eine Zusammenfassung der geläufigsten Aktivierungsfunktionen beinhaltet.

2.3.1 Sigmoid und Tangens hyperbolicus

Die Klasse der *Sigmoid*-Funktionen sind nach ihrem S-förmigen Kurvenverlauf benannt. Die Bekannteste ist dabei die *logistische Sigmoid*-Funktion:

Definition 2.13 (Logistische Sigmoid-Funktion, nach [LRU20])

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$$

Auffällig ist, dass der logistische Sigmoid den Wert 1/2 an der Stelle $x = 0$ annimmt. Für $x > 0$ nähert die Funktion sich dem Wert 1 und für $x < 0$ dem Wert 0 an. Somit liegen die Funktionswerte des logistischen Sigmoids zwischen 0 und 1 und lassen sich dementsprechend auch als Wahrscheinlichkeit interpretieren. (vgl. [LRU20, S. 532]) Der *Tangens hyperbolicus* ist eine skalierte und verschobene Version des logistischen Sigmoids, welcher wie folgt definiert ist (vgl. [LRU20, S. 532]):

Definition 2.14 (Tangens hyperbolicus, nach [LRU20])

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Durch eine einfache algebraische Manipulation wird die Ähnlichkeit zum Sigmoid noch deutlicher (vgl. [LRU20, S. 532]):

$$tanh(x) = 2\sigma(2x) - 1$$

Der Unterschied des Tangens hyperbolicus zum logistischen Sigmoid liegt in seinem Output, welcher einen Wertebereich von -1 bis 1 aufspannt. Die Gemeinsamkeiten sind, dass beide Funktionen stetig und differenzierbar sind und sich somit theoretisch in Kombination mit dem *Gradientenabstieg* (wird in Kapitel 3 näher erläutert) verwenden lassen. Allerdings haben beide Funktionen das Problem, dass sie gegen unterschiedliche Werte konvergieren, umso weiter sich die Argumente von dem Bereich um $x = 0,5$ (logistischer Sigmoid) bzw. $x = 0$ (Tangens hyperbolicus) entfernen. Das führt dazu, dass der Lernprozess durch Algorithmen, die den Gradientenabstieg nutzen, zunehmend ausgebremst wird und somit das Gradientenabstiegsverfahren stagniert. Zu erwähnen ist hier auch noch, dass beide Funktionen komponentenweise arbeiten. Das bedeutet, dass, wenn man einen Input-Vektor $\mathbf{x} = [x_1, x_2, ..., x_n]$ hat, beide Funktion auf den einzelnen Komponenten des jeweiligen Input-Vektors arbeiten und die einzelnen Ergebnis-Skalare dann wieder in einem Vektor $\mathbf{y} = [y_1, y_2, ..., y_n]$ zusammengefasst werden. Diese vektorielle Verarbeitung führt zu einer wesentlich effizienteren Berechnungsweise, wie am Anfang dieses Kapitels schon beschrieben wurde.

2.3.2 Softmax

Die *Softmax*-Funktion wird oftmals als Aktivierungsfunktion für die Output-Schicht eines neuronalen Netzes verwendet. Der gravierendste Unterschied zu den bereits vorgestellten Aktivierungsfunktionen ergibt sich jedoch aus dem Fakt, dass die Softmax-Funktion auf einem ganzen Vektor arbeitet, anstatt auf seinen einzelnen Komponenten. Angenommen man hat, wie auch bei Sigmoid oder Tangens hyperbolicus, einen Input-Vektor $\mathbf{x} = [x_1, x_2, ..., x_n]$, dann ist die Berechnung von x_i wie folgt durch die Softmax-Funktion definiert:

Definition 2.15 (Softmax, nach [LRU20])

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Der Output von $softmax(\mathbf{x}) = [softmax(x_1), softmax(x_2), ..., softmax(x_n)]$ kann dadurch als Wahrscheinlichkeitsverteilung interpretiert werden, da alle Werte zwischen 0 und 1 liegen und aufsummiert ebenfalls 1 ergeben. Dabei wird die größte Komponente des Input-Vektors x gegen 1 verschoben, während alle anderen Komponenten gegen 0 tendieren. Dieser Fakt ist der Grund, weswegen diese Funktion oftmals für die Neuronen der Output-Schicht verwendet wird. (vgl. [LRU20, S. 533])

2.3.3 ReLU

Eine weitere verbreitete Klasse von Aktivierungsfunktionen ist Rectified Linear Unit (ReLU). Dabei ähnelt die klassische ReLU-Funktion der Identitätsfunktion jedoch mit dem Unterschied, dass nur die Identität von positiven Eingaben abgebildet wird und negative Werte den Wert 0 zugewiesen bekommen:

Definition 2.16 (Rectified Linear Unit, nach [LRU20])

$$\varphi(x) = max(0, x)$$

Dabei besitzt die Funktion jedoch das Problem, dass sie an der Stelle $x = 0$ nicht differenzierbar ist, was gegen die erste Anforderung verstößt. In der Praxis wird

die Ableitung an dieser Stelle deswegen auch einfach 0 oder 1 gesetzt. Oftmals ersetzt ReLU in neuronalen Netzen die herkömmliche Sigmoid-Funktion als Standard-Aktivierungsfunktion. Dies ist auf zwei Eigenschaften von ReLU zurückzuführen: Zum einen bleibt ihr Gradient konstant und ist für positive x niemals gesättigt. Das beschleunigt den Lernprozess, da so die optimalen Parameter für ein neuronales Netz effizienter gelernt werden können. Zum anderen können die Ableitungen der Funktion mit effizienten und elementaren mathematischen Operationen berechnet werden. So ist z. B. bei ReLU im Gegensatz zum Sigmoid keine Potenzierung vorhanden. (vgl. [LRU20, S. 534])
Allerdings besitzt ReLU das Problem der Sättigung seiner Ableitung für $x < 0$. Das hat zufolge, dass wenn die Input-Werte eines Neurons einmal negativ werden, der Output dieses Neurons für den Rest des Trainings bei 0 „stecken“ bleibt. Dieses Phänomen wird in der Literatur auch oft als *Dying ReLU* bezeichnet. (vgl. [LRU20, S. 535])
Dieses Problem löst eine Variante der ReLU-Funktion, indem sie der Funktion einen zusätzlichen Parameter α hinzufügt. Diese Funktion wird dann als *Leaky-ReLU* bezeichnet:

Definition 2.17 (Leaky-ReLU, nach [LRU20])

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

Dem Parameter α wird in den meisten Fällen ein kleiner positiver Wert wie z. B. $0,01$ zugeordnet. Jedoch kann α auch als weiterer Parameter während des Trainingsprozesses optimiert werden, was dann auch als *PReLU* bezeichnet wird. (vgl. [LRU20, S. 532])
Das nächste Unterkapitel wird auf verschiedene Typen von neuronalen Netzen eingehen. Neben dem vorwärtsgerichteten neuronalen Netz gibt es auch noch weitere Versionen, welche z. B. andere Verbindungsmuster zwischen den Neuronen-Schichten nutzen. Auch die Varianten aus denen das FFN hervorgegangen ist, werden Bestandteil des nächsten Unterkapitels sein.

2.4 Typen von Neuronalen Netzen

Dieses Unterkapitel stellt den Abschluss des gesamten Kapitels dar. In vorigen Unterkapiteln wurden die Funktionsweise und der Aufbau von Neuron und neuronalen Netzen im Allgemeinen besprochen. Dieses Unterkapitel erläutert nun einzelne Typen von neuronalen Netzen, um ein grundlegendes Verständnis in Bezug auf deren Begrifflichkeiten zu schaffen. Dabei wird an dieser Stelle kein Anspruch auf Vollständigkeit erhoben. Das bedeutet, dass in diesem Unterkapitel nur eine Auswahl an Architekturen vorgestellt wird, die im Zusammenhang mit den restlichen Kapiteln dieser Arbeit nochmals von Interesse sein werden.

2.4.1 Single-Layer-Perzeptron

Das Perzeptron wurde im Jahr 1958 durch *Frank Rosenblatt* in seiner Arbeit „The perceptron: a probabilistic model for information storage and organization in the brain“ [Ros58] eingeführt. Es stellt ein vereinfachtes neuronales Netz dar, welches in seiner Grundversion (auch *einfaches Perzeptron* genannt) aus einem einzelnen künstlichen Neuron mit editierbaren Gewichten und einem Bias besteht. Man könnte also sagen, dass es sich bei dem einfachen Perzeptron um einen *binären Klassifizierer* handelt (vgl. [LRU20]).
Ein Single-Layer-Perzeptron (SLP), oder auch *einlagiges Perzeptron* genannt, ist dementsprechend ein neuronales Netz mit nur einer Schicht von Neuronen, welche gleichzeitig auch den Ausgabevektor repräsentiert. Dabei erhält jedes Neuron als Input den kompletten Eingabevektor des neuronalen Netzes und verwendet als Aktivierungsfunktion die sogenannte *Schritt-Funktion*, wodurch der Output des jeweiligen Neurons binärer Natur ist. In Abbildung 2.4 ist ein Beispiel für ein SLP zu sehen, welches für die Berechnung der logischen OR-Funktion verwendet werden kann.
Ein SLP kann zur Lösung *linear separierbarer* Probleme verwendet werden (vgl. [Ros58]). Das bedeutet, wenn ein Problem nicht linearer Natur ist, kann das SLP dieses Problem nicht lösen. *Marvin Minsky* wies 1969 mit dem bekannten XOR-Problem diese Eigenschaft von einlagigen Perzeptronen nach (vgl. [MPB17]).

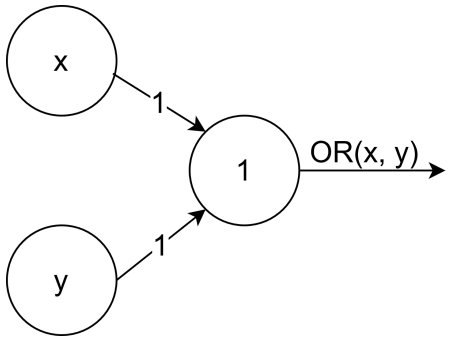


Abbildung 2.4: Ein SLP zur Berechnung der logischen OR-Funktion.

Trainiert werden kann ein SLP durch den sogenannten *Perzeptron-Lernalgorithmus*. Dieser wird für SLPs angewendet, welche die binäre Schritt-Funktion als Aktivierungsfunktion verwenden. Dass der Algorithmus in endlicher Zeit konvergiert, wies *Frank Rosenblatt* 1962 in seinem Buch „Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms“ nach (vgl. [Ros62]). Als Alternative zu dem Perzeptron-Lernalgorithmus kann auch die sogenannte *Delta-Regel*, auch *Widrow-Hoff-Algorithmus* genannt, verwendet werden. Diese Regel ist auf die Arbeit [WH60] von *Bernard Widrow* und *Marcian Edward Hoff* zurückzuführen. Sie ermöglicht als Gradienten-basiertes Verfahren sogar das Training von SLPs, welche als Aktivierungsfunktion nicht nur die binäre Schritt-Funktion verwenden. Somit kann sie als Spezialisierung des Backpropagation-Algorithmus aufgefasst werden, der im Kapitel 3 näher erläutert wird.

2.4.2 Multi-Layer-Perzeptron

Ein Multi-Layer-Perzeptron (MLP), auch mehrlagiges Perzeptron genannt, besteht im Gegensatz zum SLP aus mehreren Neuronen-Schichten. Daher werden FFNs oft mit dem MLPs assoziiert. Der Unterschied besteht jedoch darin, dass das MLP in seiner ursprünglichen Version als Aktivierungsfunktion seiner Neuronen ebenfalls nur die Schritt-Funktion verwendet und die Ausgabe der Neuronen somit nur binärer Natur wäre. Jedoch besteht in der Literatur keine Einigung über die genaue Definition von MLPs, weswegen an dieser Stelle ebenfalls die Vereinbarung getroffen wird, dass der Begriff des MLP synonym zum FFN verwendet werden kann.

Im Gegensatz zum SLP kann ein MLP auch für die Lösung *nicht-linear* separierbarer Probleme verwendet werden. Ein Beispiel dafür ist in Abbildung 2.5 zu sehen. Das dort illustrierte MLP berechnet eine Funktion, welche zur Lösung des nicht-linear separierbaren XOR-Problems verwendet werden kann. Dabei wird als Aktivierungsfunktion des MLP ebenfalls nur die binäre Schritt-Funktion verwendet. Wie allerdings schon erwähnt, kann diese jedoch allgemein für MLP durch andere Aktivierungsfunktionen ersetzt werden.

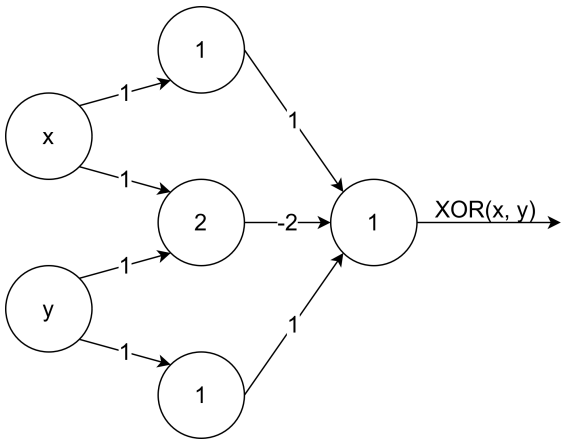


Abbildung 2.5: Ein MLP zur Berechnung der logischen XOR-Funktion.

Gewöhnlich wird ein MLP durch den Backpropagation-Algorithmus trainiert. Dieser wird überwiegend für das überwachte Lernen verwendet, kann allerdings auch im Rahmen des bestärkenden Lernens eingesetzt werden. Dabei basiert der Algorithmus auf dem sogenannten *Gradientenabstiegsverfahren*, welches im Zusammenhang mit der Optimierung von Funktionen verwendet wird. Der Backpropagation-Algorithmus (und auch der Gradientenabstieg) wird jedoch nicht weiterer Bestandteil dieses Abschnittes sein, stattdessen in Kapitel 3 näher beschrieben.
Ein weiteres Lern-Verfahren beschreibt die sogenannte Hebb-Regel [Heb49], welche auf den Psychologen *Donald Olding Hebb* zurückzuführen ist. Es gibt unterschiedliche Varianten dieser Lernregel, welche die Veränderungen der Gewichte von Verbindungen zwischen jeweils zwei Neuronen auf Grundlage der prä- und postsynaptischen Aktivierungszustände beschreibt. Aber auch diese Regel wird in einem späteren Kapitel genauer erläutert.

2.4.3 Convolutional Neural Networks

Bei Convolutional Neural Networks (CNNs) handelt es sich um eine spezielle Version eines FFN. Ihre Besonderheiten liegen in den verwendeten Verbindungsmustern, welche für die jeweiligen Neuronen-Schichten verwendet werden. In Unterkapitel 2.2 wurde schon erwähnt, dass es neben voll-verbundenen Neuronen-Schichten auch noch Schichten gibt, die andere Verbindungsmuster nutzen.
Die nachfolgende Liste erläutert unterschiedliche Neuronen-Schichten, die jeweils ein spezielles Verbindungsschema nutzen, welches im Zusammenhang mit CNNs relevant ist (vgl. [LRU20, S. 527]):

1. **Voll-verbundene Schicht:** Alle Neuronen einer Schicht sind mit jeweils allen Neuronen der nachfolgenden Schicht verbunden.

2. **Poolingschicht:** Alle Neuronen einer Schicht werden in eine bestimmte Anzahl von Clustern partitioniert. In der nächsten Schicht gibt es ein Neuron für jedes Cluster und dieses Neuron bekommt als Input dann nur den Output der Neuronen des jeweiligen Clusters der Vorgängerschicht.

3. **Faltungsschicht:** Dieser Ansatz von Verbindungen zwischen Neuronen behandelt die Neuronen einer jeden Schicht als Grid (typischerweise zweidimensional). In einer Faltungsschicht bekommt ein zu den Koordinaten (i, j) gehörendes Neuron als Input den Output des Neurons der vorigen Schicht, dessen Koordinaten in einer kleinen Region um (i, j) liegen. Als Beispiel: das Neuron (i, j) in einer Faltungsschicht bekommt als Input den Output der Neuronen der vorigen Schicht, welche zu den Koordinaten (p, q) gehören, wobei $i - 1 \leq p \leq i + 1$ und $j - 1 \leq q \leq j + 1$ sind.

Diese speziellen Neuronen-Schichten, welche im CNN verwendet werden, sind motiviert durch den Aufbau und die Funktionsweise des visuellen Kortex (vgl. [Wen+18]). Einzelne Studien konnten in der Tat feststellen, dass CNNs mit ähnlichen Codierungs- und Organisationsprinzipien wie der visuelle Kortex aufgebaut und trainiert werden (vgl. [DZR12; YD16]). Dadurch ist es naheliegend, dass CNNs häufig für die Klassifikation von Bildern eingesetzt werden. Dieses Szenario wird in diesem Abschnitt dazu verwendet, um die Funktionsweise eines CNN anschaulich erklären zu können.

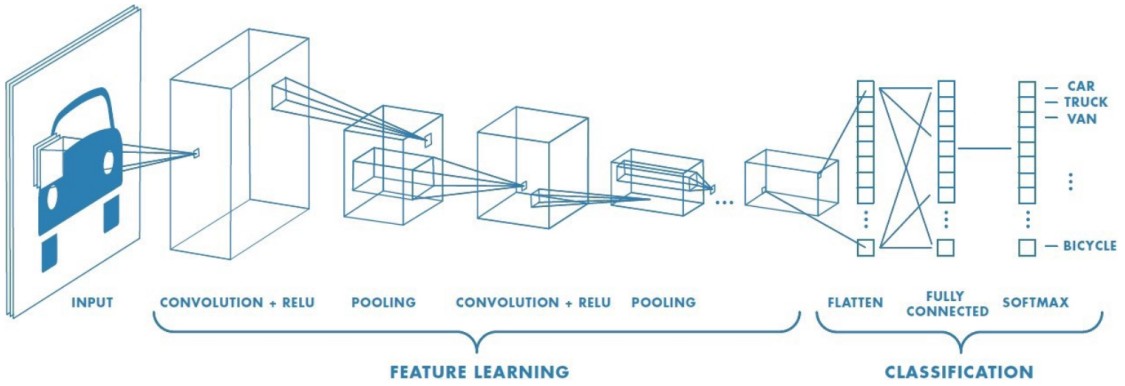


Abbildung 2.6: Die Architektur eines CNN für die Bildverarbeitung (Quelle: [Sum18]).

In Abbildung 2.6 ist zu sehen, dass es sich bei der ersten Schicht eines CNN um eine Faltungsschicht handelt. Diese ist gefolgt von einer Poolingschicht. Diese Kombination der beiden Schichten kann beliebig oft hintereinander angeordnet werden. Die letzten Schichten eines CNN sind dann allerdings wieder voll-verbunden. Die Faltungs- und Poolingschichten sind generell für das Lernen/Extrahieren von sogenannten *Features* des jeweiligen Datensatzes zuständig und die voll-verbundenen Schichten dann für die eigentliche Aufgabe des CNN. Im Folgenden wird die Arbeitsweise der drei verwendeten Schichten anhand des Beispiels einer Bild-Klassifikation noch etwas näher vorgestellt. Von einer ausführlichen mathematischen Betrachtung wird an dieser Stelle jedoch abgesehen, da es dem Zweck dieser Arbeit nicht weiter zuträglich wäre. Für den interessierten Leser wird an dieser Stelle auf die Arbeit [GBC16] von *Goodfellow et al.* und [LRU20] von *Leskovec et al.* verwiesen.

Faltungsschicht

Die Faltungsschichten eines CNN sind für das Extrahieren von *Features*, also bestimmten Eigenschaften bzw. Artefakten eines Bildes verantwortlich. Solche Features können z. B. einfache Kanten oder sonstige geometrische Formen sein. Setzt man diese einfachen Features jedoch in Beziehung, können daraus wiederum Features komplexerer Natur entstehen, wie z. B. ganze Gesichter oder andere Formen. (vgl. [LRU20, S. 549])

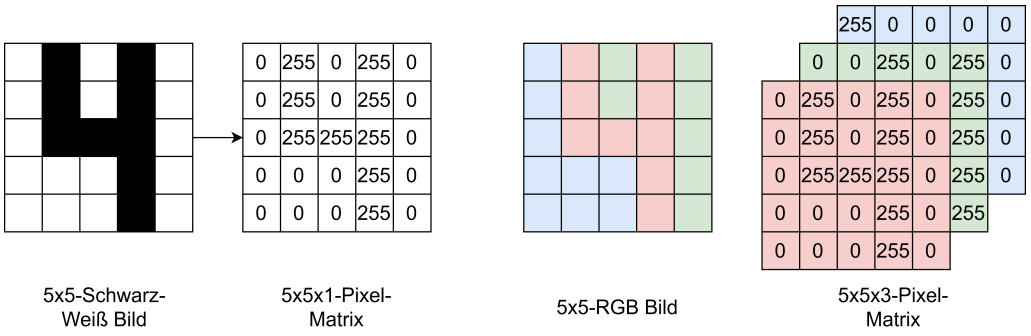


Abbildung 2.7: Ein Schwarz-Weiß- und RGB-Bild mit ihren jeweiligen Matrizen.

Zum Erkennen solcher Features in einem Bild werden sogenannte *Kernel-Filter* verwendet, welche auf alle Regionen eines Bildes angewendet werden. Um jedoch zu verstehen, wie dieses Vorgehen funktionieren kann, muss man zunächst wissen, dass Bilder in Form von Matrizen repräsentiert werden können, welche Farbwerte für jedes einzelne Pixel eines Bildes besitzen. In Abbildung 2.7 sieht man zwei 5×5-Pixel große Bilder: eines in Schwarz-Weiß und eines im RGB-Farbspektrum. Das Schwarz-Weiß-Bild resultiert also in einer $5 \times 5 \times 1$-Matrix, da hier nur die Farben Schwarz und Weiß verwendet werden. Der Wert 0 bedeutet demnach ein Pixel in komplett Weiß und der Wert 255 ein Pixel in komplett Schwarz. Alle Werte dazwischen sind verschiedene Graustufen. Das RGB-Bild setzt sich hingegen aus drei Farben zusammen. Jede Farbe hat ihre eigene Matrix, welche anzeigt, ob die jeweilige Farbe für das jeweilige Pixel vorhanden ist und in welcher Farbstufe, wodurch für ein RGB-Bild eine $5 \times 5 \times 3$-Matrix entsteht.
Eine Faltunsschicht bekommt dann diese Art von Matrizen als Input. Aus der Input-Matrix resultiert dann auch die Anordnung der einzelnen Neuronen in der Schicht. Die Aktivität eines jeden Neurons wird dann über die *Faltung* (engl. *convolution*) berechnet. Dazu werden die oben schon erwähnten Filter-Kernel verwendet, welche dann Schritt für Schritt über die Input-Matrix bewegt werden. Ein Filter-Kernel ist dabei selbst eine im Vergleich zur Input-Matrix recht kleine Matrix. In Abbildung 2.8 sieht man ein Beispiel für die Faltung einer Matrix mit einem Filter-Kernel. (vgl. [LRU20, S. 549])
Dabei wird der Filter-Kernel über die Input-Matrix gelegt, angefangen in der linken oberen Ecke (siehe Abbildung). Jedes Element des betrachteten Ausschnittes der

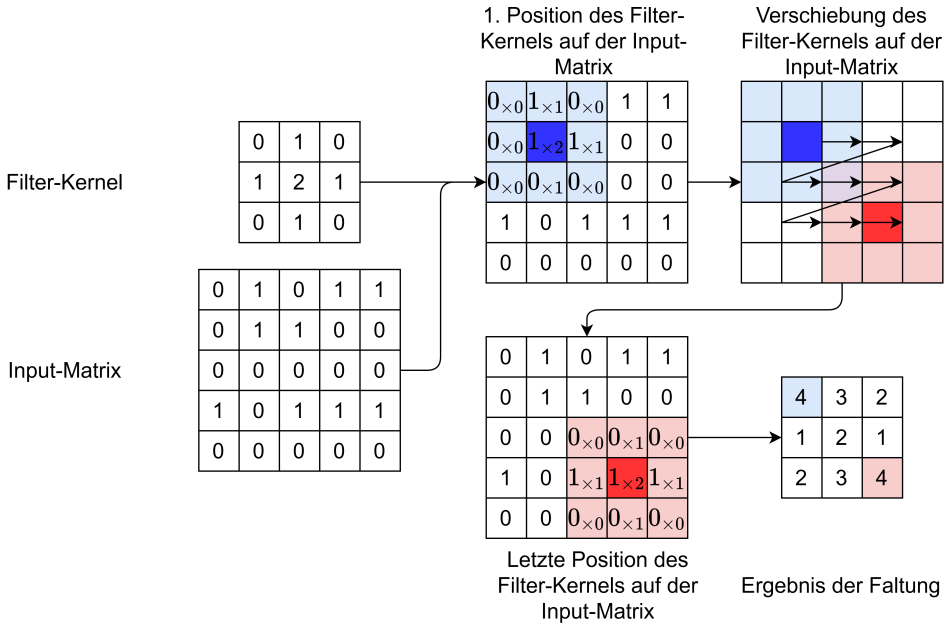


Abbildung 2.8: Die Faltung einer Input-Matrix mit einem Filter-Kernel.

Input-Matrix wird mit den dazugehörigen Elementen des Filter-Kernels multipliziert. Anschließend werden alle Elemente addiert und das Ergebnis bildet dann den Wert, welcher in der ersten Zelle der Ergebnis-Matrix vermerkt wird. (vgl. [LRU20, S. 550])
In Abbildung 2.8 ist dieser Schritt für die erste Position des Filter-Kernels zu sehen (der blaue Bereich). Der Filter-Kernel wird also auf diesen gekennzeichneten Bereich angewendet und der daraus resultierende Wert ist in der blauen Zelle der Ergebnis-Matrix zu sehen. Anschließend wird das zentrale Element (jeweils die dunkle Zelle in der Abbildung) des Filter-Kernels um eine Zelle nach rechts verschoben, sodass alle umliegenden Elemente des Filter-Kernels ebenfalls um diesen Schritt verschoben werden. Daraufhin wird der Wert der neuen Position des Filter-Kernels berechnet. Ist der Filter-Kernel am rechten Rand der Input-Matrix angelangt, verschiebt man ihn jeweils eine Zeile nach unten und an die ganz linke Position. Dieses Vorgehen wird fortgesetzt, bis man an der unteren rechten Ecke der Input-Matrix angekommen ist. (vgl. [S. 550][LRU20])
Jede Zelle der Ergebnis-Matrix entspricht dann einem einzelnen Neuron in der jeweiligen Faltungsschicht. Die Anordnung der Neuronen erfolgt dann auch entsprechend

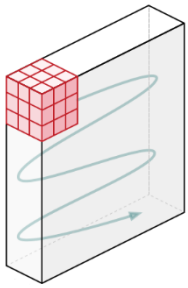


Abbildung 2.9: Verschiebung eines dreidimensionalen Filter-Kernels in einer dreidimensionalen Input-Matrix (in Anlehnung an [Sum18]).

der Ergebnis-Matrix (z. B. als Grid). Somit ist jedes Neuron mit einem Bereich der vorhergehenden Schicht über seine gewichteten Kanten verbunden. Diese Verbindungen zwischen den Neuronen werden dann im Kontext von CNN auch als *Filter-Bank* bezeichnet (vgl. [LBH15]).
In dem Beispiel aus Abbildung 2.8 wird eine zweidimensionale Matrix als Input betrachtet. Es kann aber auch der Fall sein, dass man als Input eine dreidimensionale Matrix bekommt (z. B. im Falle eines RGB-Bildes). Die Faltung einer dreidimensionalen Matrix funktioniert äquivalent zu der Faltung von zweidimensionalen Matrizen mit dem Unterschied, dass der verwendete Filter-Kernel dann ebenfalls dreidimensional wäre. (vgl. [LRU20, S. 550])
In Abbildung 2.9 ist die Verschiebung eines dreidimensionalen Filter-Kernels zu sehen. Dabei fällt auf, dass die Verschiebung weiterhin nur auf Grundlage der Spalten und der Zeilen erfolgt, nicht aber auf der dritten Dimension. Dieses Vorgehen lässt sich jedoch auch anders darstellen. Man kann die jeweilige dreidimensionale Input-Matrix und den Filter-Kernel auch in drei separate zweidimensionale Matrizen zerteilen und anschließend die Faltung eines dreidimensionalen Bereichs auf den drei Matrizen einzeln berechnen. In Abbildung 2.10 ist dieses Vorgehen exemplarisch zu sehen.
Generell ist noch zu erwähnen, dass die Größe der Filter-Kernel nicht auf eine 3×3-Matrix festgelegt ist, sodass z. B. auch eine 5×5-Matrix verwendet werden kann. Auch die Schrittgröße, um die der Filter-Kernel-Mittelpunkt verschoben wird, kann variiert werden, sodass z. B. nicht nur um jeweils eine Position, sondern um zwei

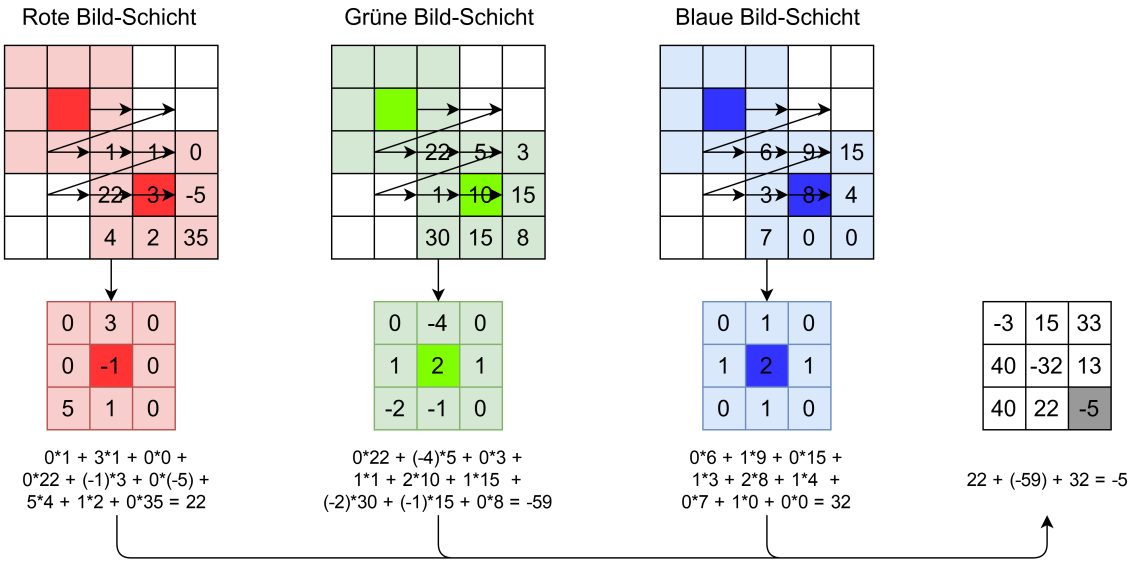


Abbildung 2.10: Die Faltung einer dreidimensionalen Pixel-Matrix eines RGB-Bildes mit einem dreidimensionalen Filter-Kernel.

verschoben wird. (vgl. [LRU20, S. 550-551])
Generell gilt auch je tiefer eine Faltungsschicht in einem CNN lokalisiert ist, umso komplexer werden die erkannten Muster. Während zum Beispiel in der ersten Faltungsschicht eines CNN nur Kanten betrachtet werden, könnten in der Siebten schon ganze Gesichter in einem Bild lokalisiert werden. Ein weiterer Punkt, der im Zusammenhang mit der Faltung nicht unerwähnt bleiben sollte, ist das sogenannte *Padding* (vgl. [LRU20, S. 551]). Da sich die bei einer Faltung betrachteten Bildbereiche überlappen können, haben insbesondere die Pixel im mittleren Bild-Bereich einen erhöhten Einfluss, da sie sich öfter im Bereich des Filter-Kernels befinden, als Pixel, die am äußeren Bildrand lokalisiert sind. Um diesem Umstand entgegenzuwirken, werden Padding-Methoden eingesetzt. Dabei wird das Bild künstlich um eine bestimmte Anzahl an Pixeln vergrößert, indem für die Faltung eine oder mehrere Reihen und Spalten am Rand einer Matrix hinzugefügt werden. Ein Beispiel für das Padding ist in Abbildung 2.11 zu sehen.
Die neu hinzugefügten Zellen werden dabei jeweils mit dem Wert 0 belegt. Die Faltung verläuft dann wie gehabt, bloß dass an dem künstlich hinzugefügten Bildrand

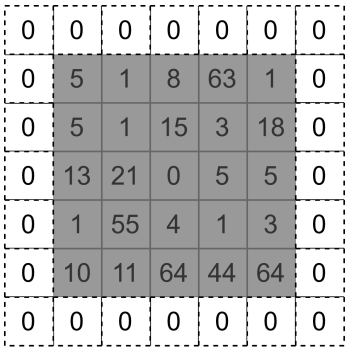


Abbildung 2.11: Padding einer 5×5-Matrix auf eine 7×7-Matrix.

begonnen wird. Dadurch haben die Pixel des wirklichen Bildrandes bei der Faltung der jeweiligen Matrix eine zentralere Position und somit einen höheren Einfluss.

Poolingschicht

Die Poolingschicht ist dafür zuständig, die Anzahl der gefundenen Bild-Features zu reduzieren. Als Input erhält die Poolingschicht das Ergebnis der vorgeschalteten Faltungsschicht. Ähnlich zur Faltung bewegt sich beim Pooling wieder ein Fenster über die Input-Matrix. Dadurch wird eine kleinere Ergebnis-Matrix berechnet. Dabei wird beim Pooling allerdings kein Filter genutzt. Stattdessen werden nur die originalen Pixel-Werte des Matrix-Ausschnittes betrachtet. Bei der Berechnung des Endergebnisses eines Ausschnittes kann dann zwischen zwei Varianten unterschieden werden: *Max-Pooling* und *Average-Pooling*. Das Max-Pooling übernimmt den maximalen Pixel-Wert des jeweiligen Matrix-Bereichs in die Ergebnis-Matrix. Die restlichen Werte des Ausschnittes werden bei diesem Verfahren verworfen und somit in den folgenden Neuronen-Schichten nicht mehr berücksichtigt. Beim Average-Pooling wird der Durchschnitt über alle betrachteten Pixel-Werte berechnet und als Wert in die Ergebnis-Matrix übernommen. In Abbildung 2.12 sind Beispiele für beide Verfahren zu sehen.

Das in der Abbildung verwendete Pooling-Fenster hat dabei die Größe 2×2. Generell kann die Größe des Fensters variiert werden, wobei gilt: je größer das Fenster,

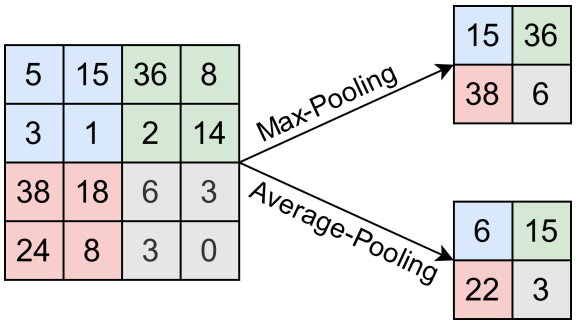


Abbildung 2.12: Max- und Average-Pooling einer Matrix.

umso kleiner die aus dem Pooling resultierende Matrix. Auch die Schrittweite, um die das Fenster jeweils verschoben wird, kann variabel gewählt werden. In Abbildung 2.12 wird als Schrittweite z. B. der Wert 2 gewählt. Dadurch überlappen sich die betrachteten Bereiche im Beispiel nicht.
An dieser Stelle wird nochmals daran erinnert, dass die Matrizen an dieser Stelle nur zur Visualisierung verwendet werden. Die einzelnen Zellen einer Input-Matrix für das Pooling entsprechen nach wie vor einzelnen Neuronen der vorherigen Faltungsschicht. Das eigentliche Pooling erfolgt dann über die Kanten zwischen den jeweiligen Neuronen der Faltungs- und Poolingschicht. Das bedeutet im Umkehrschluss, dass nicht alle Neuronen der Faltungsschicht mit allen Neuronen der Poolingschicht verbunden sind, sondern jedes Neuron der Poolingschicht mit einer bestimmten Gruppe von Neuronen der Faltungsschicht verbunden ist. Mithilfe der Aktivierungsfunktionen der Pooling-Neuronen kann dann der Durchschnitt oder Maximalwert des jeweiligen Inputs berechnet werden.
Generell erfüllt das Pooling die Aufgabe, nur die dominantesten Bild-Artefakte für die weitere Verarbeitung des CNN auszuwählen. Ansonsten würde die Anzahl der betrachteten Eigenschaften eines Bildes explodieren und somit das Training eines CNN verlangsamen.
Dadurch, dass beim Max-Pooling komplette Werte einer Matrix eliminiert werden, kann man das Max-Pooling auch als eine Art *Rausch-Filter* betrachten. Nur die am stärksten vertretenen Artefakte sind von Interesse, weshalb das Rauschen, was in einem Bild vorhanden sein kann, in diesem Schritt herausgefiltert wird. Dieses Selektieren von betrachteten Artefakten eines Bildes hat ebenfalls den Effekt einer

Dimensionsreduktion, da viele Parameter beim Pooling aussortiert (Max-Pooling) bzw. zu einem Wert gemittelt (Average-Pooling) werden. Der Kerngedanke bei diesem Vorgehen ist, dass das CNN wirklich nur mit den relevantesten Informationen gefüttert werden soll, um so seine Performance zu erhöhen.

Voll-verbundene Schicht

Bei der letzten Komponente eines CNN handelt es sich meistens um ein MLP. Je nachdem, worin die Aufgabe des CNN besteht, wird für das jeweilige MLP eine bestimmte Architektur gewählt. Um den Output der letzten Poolingschicht für das MLP aufzubereiten, wird ein Verfahren verwendet, welches als *Flattening* bezeichnet wird. Dabei werden die jeweiligen Zellen einer Matrix in einen zusammenhängenden Vektor überführt, indem die Zeilen der Matrix einfach hintereinander zu einem Vektor zusammengefügt werden. Dieser Vektor kann dann als Input für das MLP verwendet werden.

Das MLP selber kann ebenfalls aus mehreren Schichten bestehen, die jeweils voll verbunden sind. Für die Bild-Klassifikation wird für die Output-Schicht meistens die Softmax-Funktion aus Definition 2.15 verwendet, um eine Wahrscheinlichkeitsverteilung über alle möglichen Klassen zu erhalten.

Es kann also festgehalten werden, dass ein CNN aus mehreren aufeinander folgenden Faltungs- und Poolingschichten besteht, dessen Ergebnisse dann letztendlich den Input für ein MLP darstellen, welches für die eigentliche Aufgabe des CNN zuständig ist. Die Faltungs- und Poolingschicht sind also dafür zuständig, die Eigenschaften der einzelnen Datenpunkte (wie z. B. Bilder) zu extrahieren, damit das MLP diese dann für die zu erledigende Aufgabe verwenden kann.

Das Training eines CNN erfolgt, ähnlich zu dem eines herkömmlichen MLP, meistens überwacht und mit dem Backpropagation-Algorithmus. Um das Training noch mehr zu beschleunigen, wird für die Faltungsschichten auch bevorzugt ReLU als Aktivierungs-Funktion für die einzelnen Neuronen verwendet. Dadurch, dass z. B. alle Parameter der Faltung über die Kantengewichte zwischen den jeweiligen Neuronen modelliert werden, können durch das Trainieren eines CNN sogar neue Filter-Kernel entdeckt werden (vgl. [LBH15]).

Ein CNN kann neben der Verarbeitung von Bildern ebenso für anderen Datenformate verwendet werden. Auch Sprach-, Audio- und Video-Daten lassen sich mit CNNs verarbeiten (vgl. [LBH15]). Ferner muss die zentrale Aufgabe eines CNN nicht immer in der Klassifikation bestehen. Andere Szenarien aus dem Bereich des maschinellen Lernens sind ebenfalls realisierbar.

2.4.4 Rekurrente neuronale Netze

Rekurrente Neuronale Netze (RNNs) unterscheiden sich von herkömmlichen neuronalen Netzen darin, dass die Verbindungen zwischen den Neuronen nicht ausschließlich vorwärts gerichtet sein müssen. Diese Beschränkung, die für alle bisher vorgestellten Varianten von neuronalen Netzen gilt, wird bei dem RNN vollständig aufgehoben. Dadurch kann es in einem RNN zu sogenannten Rückkopplungen kommen. Dabei kann in verschiedene Arten von Rückkopplungen unterschieden werden:

1. **Direkt:** Der Output eines Neurons wird als ein weiterer eigener Input verwendet.

2. **Indirekt:** Der Output eines Neurons wird als Input von Neuronen vorheriger Schichten verwendet.

3. **Seitwärts:** Der Output eines Neurons wird als Input von Neuronen der selben Schichten verwendet.

4. **Vollständig:** Der Output eines Neurons wird als Input aller anderen Neuronen verwendet.

Dadurch dass der Output eines Neurons nicht mehr ausschließlich vorwärts weitergegeben wird, ist der eigene Aktivierungszustand nun nicht mehr nur vom aktuellen Input des neuronalen Netzes abhängig, sondern auch von den vorherigen. Ein Neuron kann somit eine Art *Historie* über vergangene Aktivierungszustände speichern und diese als weiteren Input verwenden. Dadurch kann ein RNN eine Reihe interner Zustände berechnen, welche von den aktuellen und vergangenen Zuständen der einzelnen Neuronen abhängig ist.

RNNs werden deswegen für die Verarbeitung von sequenziellen Daten verwendet. Dabei können viele Datensätze als eine Sequenz aufgefasst werden: Ein Satz ist eine Abfolge von Wörtern, ein Video ist eine Abfolge von Bildern und ein Lied eine Abfolge von Tönen. Zwischen den einzelnen Elementen einer Sequenz bestehen dabei oft zeitabhängige Relationen. So kann sich ein bestimmtes Wort in einem Satz auf ein nachfolgendes oder vorheriges Wort beziehen. Mithilfe seiner internen Zustände kann das RNN solche Relationen finden und somit auch bei der Berechnung berücksichtigen. (vgl. [LRU20, S. 557])

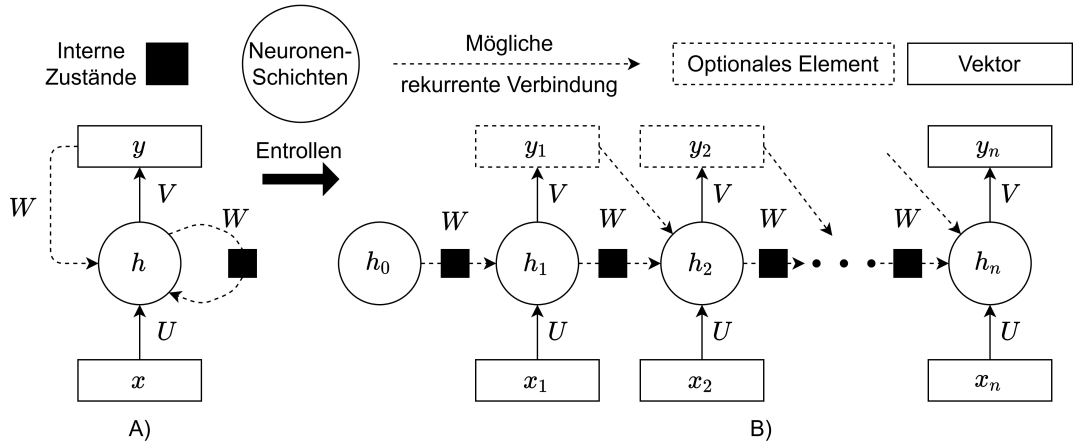


Abbildung 2.13: A) Simples RNN mit Input x, Neuronen-Schichten h und Output y, B) Ausgerolltes RNN mit n Zeitschritten.

Das Modell eines einfachen RNN kann man in Abbildung 2.13 sehen. Dabei stellt x jeweils den Input und y den Output des RNN dar. Die verdeckte Schicht wird durch h repräsentiert. Dabei besitzt das RNN rekurrente Verbindungen, welche über die Gewichtsmatrix $\mathbf{W}$ ihre Gewichte erhalten. Die Matrizen $\mathbf{U}$ und $\mathbf{V}$ stellen dabei die Gewichtsmatrizen der Input- und Output-Schicht dar. Aus der Abbildung 2.13 ist aber auch zu erkennen, dass es unterschiedliche Architekturen eines RNN geben kann. Man stelle sich vor, dass t der aktuelle Zeitschritt ist und n die maximale Anzahl an möglichen Zeitschritten. Dann ergeben sich daraus folgende Varianten eines RNN (vgl. [GBC16, S. 379]):

- Ein RNN, welches einen Output y_t mit jedem Zeitschritt t produziert und seine internen Zustände als Input für den nächsten Zeitschritt $t + 1$ weitergibt.

- Ein RNN, welches einen Output y_t mit jedem Zeitschritt t produziert und diesen Output als (zusätzlichen) Input für den nächsten Zeitschritt $t + 1$ weitergibt.
- Ein RNN, welches nur einen finalen Output y_n produziert und seine internen Zustände als Input für den nächsten Zeitschritt $t + 1$ weitergibt.

Neben diesen drei geläufigen Varianten gibt es jedoch auch noch andere, die vor allem auch mit dem in Zeitschritt t produzierten Fehler des RNN zusammenhängen. Für den interessierten Leser wird in diesem Zusammenhang auf das Werk [GBC16] von *Goodfellow et al.* verwiesen.
Das Training eines RNN erfolgt über die sogenannten Backpropagation Through Time (BPTT), zu deutsch *Backpropagation über die Zeit*, welche eine Variante des Backpropagation-Algorithmus ist (vgl. [LRU20, S. 561]). Das BPTT-Verfahren wird nicht weiter Bestandteil dieses Unterkapitels sein. Für weiterführende Informationen wird an dieser Stelle auf das Buch [LRU20] verwiesen.
Ein Problem mit den herkömmlichen RNNs ist, dass es beim Training durch BPTT zum *Verschwinden* oder *Explodieren* von Gradienten kommen kann (vgl. [LRU20, S. 561]). Ein RNN kann nur Verbindungen zwischen Inputs lernen, zwischen denen eine geringe Anzahl an Zeitschritten liegt. *Goodfellow et al.* beschreiben das *Verschwinden* eines Gradienten als das Problem, dass der Gradient in Zeitschritt t komplett von den kürzlich zurückliegenden Zeitschritten abhängig ist und Zeitschritte, die länger zurückliegen, kaum Einfluss haben (vgl. [GBC16, S. 418]). Nutzt man alternative Aktivierungsfunktionen, wie z. B. ReLU, so resultiert daraus das *Explodieren* des Gradienten, da die zur Berechnung verwendeten Matrizen zu große Werte enthalten (vgl. [GBC16, S. 418]).
Diese Probleme können mithilfe einer speziellen Art von RNN gelöst werden: den sogenannten Long-Short-Term-Memory (LSTM)-Netzwerken. Diese Art von Netzwerken wurden 1977 von *Sepp Hochreiter* und *Jürgen Schmidhuber* in ihrer Veröffentlichung „Long short-term memory“ [HS97] eingeführt. Im Gegensatz zu einem herkömmlichen RNN kann ein LSTM auch Relationen über einen längeren Zeithorizont erlernen und dadurch besser durch Verfahren des Gradientenabstiegs trainiert werden.

2.5 Zusammenfassung

Dieses Kapitel hat in die Grundlagen von neuronalen Netzen eingeführt. Dazu wurde zunächst der Aufbau und die Funktionsweise eines einzelnen Neurons erläutert, da es der zentrale Bestandteil von neuronalen Netzen ist. Aufbauend darauf konnten dann der Aufbau und die Funktionsweise eines neuronalen Netzes erklärt werden. Dabei wurde ihre Schichten-Architektur ausgenutzt, indem Operatoren der linearen Algebra genutzt wurden, um die jeweiligen Aktivierungszustände der Neuronen-Schichten effizienter berechnen zu können. In Unterkapitel 2.3 wurden dann einige der geläufigsten Aktivierungsfunktionen vorgestellt und ihre Vor- und Nachteile betrachtet. Das Unterkapitel 2.4 hat dann einige Typen von neuronalen Netzen näher erläutert.
Generell ist noch zu erwähnen, dass neuronale Netze als ML-Modell nahezu universell einsetzbar sind, was einer ihrer stärksten Vorteile gegenüber anderen Modellen ist. Auch die Komplexität der Funktionen, die durch ein neuronales Netz modelliert werden können, stellen eine weitere Stärke dar. Das geht jedoch auf Kosten ihrer Interpretierbarkeit. Während andere Modelle wie z. B. Support-Vector-Machines oder Entscheidungsbäume recht gut nachvollziehbar sind, haben neuronale Netze das Problem, dass man das in ihnen gespeicherte Wissen nicht interpretieren kann. Man kann also die Korrektheit des gespeicherten Wissens nur durch den Test-Datensatz überprüfen. Deckt dieser nicht 100% aller möglichen Szenarien ab, so gibt es keine Garantie dafür, dass das jeweilige neuronale Netz in allen Fällen die richtige Ausgabe produziert. Dennoch werden neuronale Netze in vielen Systemen eingesetzt, da sie durch ihre Komplexität Leistungen erzielen können, die mit anderen ML-Modellen bisher nicht möglich sind.
Im nächsten Kapitel 3 wird der State-of-the-Art im Bereich des maschinellen Lernens dargestellt. Dabei wird ein populärer Lernalgorithmus für neuronale Netze vorgestellt, der im weiteren Verlauf dieser Arbeit eine wichtige Funktion einnehmen wird.

3 Lernen in neuronalen Netzen

In diesem Kapitel wird ein populärer Lernalgorithmus aus dem ML-Bereich beschrieben, welcher für das Training neuronaler Netze verwendet wird. Dazu müssen jedoch in Unterkapitel 3.1 zunächst die Grundlagen im Bereich der Fehlerfunktion von neuronalen Netzen gelegt werden. Das Unterkapitel 3.2 wird in die Gradienten-basierte Optimierung von Funktionen einführen und verschiedene Aspekte aus dem Bereich der Analysis definieren und näher erklären. Anschließend wird dann in Unterkapitel 3.3 der Lernalgorithmus erläutert.

3.1 Fehlerfunktionen

Dieses Unterkapitel behandelt die sogenannten Fehlerfunktionen, welche im Zusammenhang mit neuronalen Netzen und dem überwachten Lernen genutzt werden. Der Fehler eines neuronalen Netzes wie folgt definiert:

Definition 3.1 (Fehler eines neuronalen Netzes)
Sei N ein neuronales Netz, y der von N vorhergesagte Output und $\hat{y}$ der in der realen Welt beobachtete Output. Dann ist der Fehler L von N definiert als der Unterschied zwischen y und $\hat{y}$.

Der Fehler ist im Zusammenhang mit dem überwachten Trainieren bzw. Lernen eines neuronalen Netzes von Interesse, da er anzeigt, ob ein neuronales Netz den richtigen oder falschen Output produziert hat. Für die Berechnung des Fehlers können unterschiedliche Verfahren verwendet werden. Dabei muss jedoch unterschieden werden, um welches Lernszenario es sich handelt:

- **Regressions-Lernen:** Der Output y eines neuronalen Netzes ist skalarer Natur. Es soll also ein einzelner Wert vorausgesagt werden.

- **Klassifikations-Lernen:** Der Output y eines neuronalen Netzes ist vektorieller Natur. Es soll also ein Vektor vorausgesagt werden, dessen Elemente die Zugehörigkeit zur jeweiligen Klasse repräsentieren.

In beiden Szenarien wird für die Berechnung des Verlustes ein *annotierter* Datensatz benötigt. Das bedeutet, dass zu jedem Datenpunkt x_i auch einen entsprechenden (beobachteten) Output $\hat{y}$ existiert. Diese Art von Datensatz wird auch als Trainingsdatensatz bezeichnet.

Definition 3.2 (Trainingsdatensatz)
Ein Trainingsdatensatz $T = \{d_1, d_2, ..., d_n\}$ ist eine Menge von Tupeln $d_i = (x_i, \hat{y}_i)$, wobei x_i den jeweiligen Input für das neuronale Netz N darstellt und $\hat{y}_i$ den annotierten beobachteten Output zum jeweiligen Datenpunkt x_i. Die Anzahl aller Trainingsbeispiele in T wird durch n repräsentiert.

Ferner lassen sich die unterschiedlichen Verlustfunktionen weiter durch die Art des Lernens unterteilen. In dessen Kontext kommt die jeweilige Verlustfunktion zum Einsatz:

- **Online-Lernen:** Der Fehler eines NN wird für jedes Trainingsbeispiel aus T einzeln ermittelt.
- **Batch-Lernen:** Der Trainingsdatensatz T wird in sogenannte *Batches* unterteilt. Ein Batch ist eine Teilmenge von Trainingsbeispielen aus dem Datensatz T. Der Fehler eines NN berechnet sich hier also über alle Trainingsbeispiele in dem jeweiligen Batch.
- **Offline-Lernen:** Der Fehler eines NN wird über alle Trainingsbeispiele aus T ermittelt.

Im Rest dieses Unterkapitels werden unterschiedliche Funktionen zur Berechnung des Fehlers eingeführt. Wir orientieren uns dabei an der Arbeit [LRU20] von Leskovec, Rajaraman und Ullman .

3.1.1 Regressions-Fehler

Da beim Regressions-Lernen der Output eines neuronalen Netzes aus einem einzelnen Output-Wert y besteht, kann man intuitiv den Fehler bestimmen, indem man das Quadrat der Differenz aus produziertem und beobachteten Output zu berechnet:

Definition 3.3 (Squared-Error, nach [LRU20])

$$l(y, \hat{y}) = (y - \hat{y})^2$$

Diese Art von Fehlerberechnung für die Regression kann dem Online-Lernen zugeordnet werden, da hier nur der Output bezüglich eines einzelnen Trainingsbeispieles x berücksichtigt wird. In der Regel möchte man jedoch auch die Fehler für das Batch- bzw. Offline-Lernen berechnen können. Dazu kann die Formel aus Definition 3.3 abgewandelt werden, um den Fehler über mehrere Trainingsbeispiele zu berechnen:

Definition 3.4 (Mean-Squared-Error, nach [LRU20])
Sei $T = \{(x_1, \hat{y_1}), (x_2, \hat{y_2}), ..., (x_n, \hat{y_n})\}$ *der Trainingsdatensatz. Dann repräsentiert* $P = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ *die dazugehörigen Input-Output-Paare des neuronalen Netzes* N*. Der Fehler zwischen vorhergesagten und beobachteten Output ist dann folglich definiert durch:*

$$L(P, T) = \sum_{i=1}^{n} \frac{(y_i - \hat{y}_i)^2}{n} = MSE$$

Diese Art von Fehler wird Mean-Squared-Error (MSE), zu deutsch *mittlerer quadratischer Fehler*, genannt. Das liegt darin begründet, dass das Quadrat der Differenz der einzelnen Werte beider Outputs gebildet und anschließend über der Länge des Output-Vektors gemittelt wird.
Es existiert auch eine Modifikation des MSE, welche auch Root-Mean-Squared-Error (RMSE) genannt wird. Dabei ist MSE allerdings nur das Quadrat von RMSE:

Definition 3.5 (RMSE, nach [LRU20])
Sei $T = \{(x_1, \hat{y_1}), (x_2, \hat{y_2}), ..., (x_n, \hat{y_n})\}$ *der Trainingsdatensatz. Dann repräsentiert* $P = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ *die dazugehörigen Input-Output-Paare des neuronalen Netzes* N*. Der Fehler zwischen vorhergesagten und beobachteten Output ist dann folglich definiert durch:*

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(y_i - \hat{y}_i)^2}{n}} = \sqrt{MSE}$$

Es ist jedoch zweckmäßig die Quadratwurzel wegzulassen, um die Ableitung der Funktion zu vereinfachen, die während des Trainings benötigt wird. Wenn wir den MSE minimieren, wird der RMSE in jedem Fall ebenso minimiert. (vgl. [LRU20, S. 536])
Ein Problem mit dem MSE ist aber, dass er sehr sensitiv gegenüber Ausreißern ist, da die einzelnen Fehler quadriert werden. Einige wenige Ausreißer können sehr stark zum Fehler beitragen und die Wirkung anderer Punkte überlagern, was den Trainingsprozess anfällig für wilde Schwankungen macht. Ein Weg, damit umzugehen, ist es, den *Huber*-Verlust anzuwenden. (vgl. [LRU20, S. 536])

Definition 3.6 (Huber-Verlust, nach [LRU20])
Man nehme an das $z = y - \hat{y}$ *und* δ *eine Konstante ist. Der Huber-Verlust ist dann wie folgt definiert:*

$$L_\delta(z) = \begin{cases} z^2, wenn\ |z| \leq \delta \\ 2\delta(|z| - \frac{1}{2}\delta), sonst \end{cases}$$

Die Konstante δ stellt dabei zum einen den Schwellenwert für Ausreißer dar und zum anderen auch einen Faktor, um den der jeweilige Ausreißer abgeschwächt wird, um den Gesamtfehler nicht zu verfälschen. Dabei wird der Huber-Verlust auf die Fehler der einzelnen Trainingsbeispiele d_i angewendet, um den Gesamtfehler über dem jeweiligen Batch bzw. dem gesamten Trainingsdatensatz realistischer berechnen zu können.

3.1.2 Klassifikations-Fehler

Bei der Klassifikation soll ein Input einer oder mehreren Klassen zugeordnet werden. Dabei besteht der Output eines NN nicht in einem einzelnen Wert, sondern aus einem Vektor, dessen einzelne Komponenten die Zugehörigkeit zur jeweiligen Klasse anzeigen.
Man nehme beispielsweise ein Multiklassen-Problem mit den Zielklassen $C_1, C_2, ..., C_n$ an. Jeder Punkt im Trainingsdatensatz wird dann durch ein Tupel $(\mathbf{x}, \mathbf{p})$ dargestellt, wobei $\mathbf{x}$ der Input und $\mathbf{p} = [p_1, p_2, ..., p_n]$ der dazugehörige Output ist. Dabei gibt p_i die Wahrscheinlichkeit an, dass $\mathbf{x}$ zu einer bestimmten Klasse C_i gehört. Somit kann $\mathbf{p}$ als Wahrscheinlichkeitsverteilung interpretiert werden ($\sum_i p_i = 1$). Äquivalent verhält es sich bei den produzierten Outputs des neuronalen Netzes, welche jeweils durch ein Tupel $(\mathbf{x}, \mathbf{q})$ repräsentiert werden, wobei $\mathbf{x}$ hier auch wieder der Input und $\mathbf{q}$ der dazugehörige Output ist. Das jeweilige NN ist also so konstruiert, dass es als Output ebenfalls einen Vektor $\mathbf{q} = [q_1, q_2, ..., q_n]$ produziert, welcher ebenfalls als Wahrscheinlichkeitsverteilung interpretiert werden kann ($\sum_i q_i = 1$). In Abschnitt 2.3 wurde schon erläutert, dass die Softmax-Funktion verwendet werden kann, um einen solchen Vektor zu produzieren. (vgl. [LRU20, S. 537])
Da nun als Output zwei Wahrscheinlichkeitsverteilungen verglichen werden müssen, sollte für den Fehler eine Funktion verwendet werden, mit welcher sich die Distanz zwischen den jeweiligen Verteilungen ermitteln lässt. Um dies zu bewerkstelligen, wird zunächst die Entropie für diskrete Wahrscheinlichkeitsverteilungen definiert. (vgl. [LRU20, S. 537])

Definition 3.7 (Entropie, nach [LRU20])
Die Entropie einer Wahrscheinlichkeitsverteilungen p ist definiert als:

$$H(\mathbf{p}) = -\sum_{i=1}^{n} p_i \ log \ p_i$$

Die Entropie zeigt den mittleren Informationsgehalt pro Variable (in unserem Fall die Klassen) einer Wahrscheinlichkeitsverteilung an. Dabei stammt dieses Maß ursprünglich aus der Informationstheorie und wird deshalb auch oft als mittlerer Informationsgehalt einer Nachricht bezeichnet.

Mithilfe der Entropie kann nun die sogenannte Kreuz-Entropie definiert werden.

Definition 3.8 (Kreuz-Entropie, nach [LRU20])
Sei X die Menge der Zufallsvariablen, welche gemäß $\mathbf{p}$ verteilt ist und $\mathbf{q}$ eine weitere Verteilung auf dem Ereignisraum von X. Dann ist die Kreuz-Entropie bezüglich $\mathbf{p}$ und $\mathbf{q}$ definiert als:

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{n} p_i \ log \ q_i$$

Die Kreuz-Entropie lässt sich jedoch auch folgendermaßen alternativ berechnen:

$$H(\mathbf{p}, \mathbf{q}) = H(\mathbf{p}) + D(\mathbf{p}||\mathbf{q}) \text{ mit } H(\mathbf{p}) = H(\mathbf{p}, \mathbf{p}) \tag{3.1}$$

Dabei wird $D(\mathbf{p}||\mathbf{q})$ als Kullback-Leibler-Divergenz (KL-Divergenz) bezeichnet und lässt sich schließlich durch einfaches Umstellen der Formel 3.1 folgendermaßen definieren:

Definition 3.9 (Kullback-Leibler-Divergenz, nach [LRU20])

$$D(\mathbf{p}||\mathbf{q}) = H(\mathbf{p}, \mathbf{q}) - H(\mathbf{p}) = \sum_{i=1}^{n} p_i \ log\frac{p_i}{q_i}$$

Die KL-Divergenz kann somit als Maß der Unterschiedlichkeit zweier Wahrscheinlichkeitsverteilungen interpretiert werden. Deswegen wird sie häufig als *Distanz zwischen zwei Wahrscheinlichkeitsverteilungen* beschrieben, obwohl sie nicht kommutativ ist ($D(\mathbf{p}||\mathbf{q}) \neq D(\mathbf{q}||\mathbf{p})$). Trotzdem kann sie als Metrik für den Fehler verwendet werden, da es eine gewisse Asymmetrie gibt. Denn $\mathbf{p}$ kann als Grundwahrheit angenommen werden und $\mathbf{q}$ ist der vorhergesagte Output. Somit ist $H(\mathbf{p})$ unabhängig vom gelernten Modell und beruht somit alleine auf dem Input, weswegen das Minimieren der KL-Divergenz äquivalent zum Minimieren der Kreuz-Entropie ist. (vgl. [LRU20, S. 538])
Vertauscht man jedoch die Rolle von $\mathbf{p}$ und $\mathbf{q}$ so kommt ein anderes Ergebnis heraus, da dann der vorhergesagte Output vom neuronalen Netz als Grundwahrheit angenommen würde.

3.2 Gradienten-basierte Optimierung

Der Backpropagation-Algorithmus verwendet für die Optimierung eines neuronalen Netzes den sogenannten Gradientenabstieg. Daher müssen zunächst einige Grundlagen im Bereich der Analysis gelegt werden. Dazu wird der nächste Abschnitt 3.2.1 erläutern, was man unter einem *Gradienten* versteht und in diesem Zusammenhang erklären, wie das *Gradientenabstiegsverfahren* funktioniert. Außerdem wird definiert, was unter der *Jacobi-Matrix* zu verstehen ist und was die *Kettenregel* besagt. Anschließend werden in Abschnitt 3.2.2 einige Varianten des herkömmlichen Gradientenabstiegs erläutert, welche ebenfalls ihre Anwendung im Zusammenhang mit dem Backpropagation-Algorithmus finden.

3.2.1 Gradienten, Jacobi-Matrix und Kettenregel

Die meisten Optimierungsverfahren versuchen eine Funktion zu optimieren, indem sie ihre Argumente (also den Input der Funktion) so modifizieren, dass die jeweilige Funktion entweder ein Minimum oder Maximum annimmt (vgl. [GBC16, S. 82]). Im Zusammenhang mit dem Backpropagation-Algorithmus wird in diesem Abschnitt das Minimieren einer Funktion im Mittelpunkt stehen.
Angenommen man hat eine Funktion $y = f(x)$ mit $x, y \in \mathbb{R}$. Die Ableitung $f'(x)$ (auch als $\frac{dy}{dx}$ bezeichnet) gibt dann den Anstieg der Funktion $f(x)$ an dem Punkt x an. Das bedeutet dann folglich, dass die erste Ableitung einer Funktion aussagt, wie sich eine Änderung des Argumentes x auf den jeweiligen Funktionswert y auswirkt. Die Ableitung einer Funktion kann also dazu verwendet werden, um ihren Funktionswert zu minimieren, da sie verrät, was man an x ändern muss, um ein besseres y zu erhalten. Möchte man nun also eine Funktion $f(x)$ minimieren, macht man kleine Schritte in Richtung des jeweiligen Minimums, indem man x in die entgegengesetzte Richtung von $f'(x)$ verändert. Dieses Vorgehen wird dann auch als *Gradientenabstieg* (vgl. [Cau47]) bezeichnet und wurde von *Augustin Cauchy*, einem französischen Mathematiker, im Jahr 1847 eingeführt. (vgl. [GBC16, S. 83])
Oftmals müssen aber auch Funktionen abgeleitet werden, die mehr als einen Input-Parameter aufweisen: $f : \mathbb{R}^n \to \mathbb{R}$. Diese Funktion hat dann mehrere Argumente, aber dennoch nur einen skalaren Funktionswert. Die Ableitungen mehrdimensionaler

Funktionen werden dann als *partielle* Ableitungen $\frac{\partial}{\partial x_i} f(\mathbf{x})$ bezeichnet. Diese geben an, wie f sich verändert, wenn nur die Variable x_i im Punkt $\mathbf{x}$ ansteigt. Der *Gradient* einer Funktion f ist dann ein Vektor, dessen Elemente die partiellen Ableitungen der jeweiligen Funktion bezüglich ihrer Argumente repräsentieren. Der Gradient kann also auch als Verallgemeinerung der Ableitung für mehrdimensionale Funktionen verstanden werden. Der Operator ∇ wird auch als *Nabla-Operator* bezeichnet. Im folgenden Fließtext wird der Gradient auch durch g repräsentiert. (vgl. [GBC16, S. 84-85])

Definition 3.10 (Gradient, nach [LRU20])
Gegeben eine Funktion $f : \mathbb{R}^n \to \mathbb{R}$, welche einen Vektor $\mathbf{x} = [x_1, x_2, ..., x_n]$ auf einen Skalar $y = f(\mathbf{x})$ abbildet. Der Gradient von y im Punkt $\mathbf{x}$, repräsentiert durch $\nabla_{\mathbf{x}}(y)$, ist dann gegeben durch

$$\nabla_{\mathbf{x}}(y) = [\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, ..., \frac{\partial y}{\partial x_n}]$$

Nach dieser Definition ist der Gradient also ein n-komponentiger Vektor, der für jeden Punkt x_i einer differenzierbaren Funktion f die partielle Ableitung enthält. Dabei zeigt der Gradient g einer Funktion direkt nach oben (repräsentiert also den Anstieg der mehrdimensionalen Funktion an dem jeweiligen Punkt). Als Gradientenabstieg wird dann bezeichnet, sich von einem beliebigen Startpunkt aus schrittweise entgegen dem Gradienten zu bewegen, also in Richtung $-g$ zu gehen. Der Betrag $|g|$ gibt die Größe des jeweiligen Schrittes an, um die man sich vom aktuellen Punkt wegbewegt. (vgl. [GBC16, S. 85])
Der Gradientenabstieg gibt also als Ergebnis einen neuen Punkt zurück (vgl. [GBC16, S. 85]):

$$\mathbf{x}' = \mathbf{x} - \eta \nabla_{\mathbf{x}} f(\mathbf{x})$$

Die Variable η wird dabei als *Lernrate* bezeichnet und nimmt Einfluss auf die Schrittgröße des Abstiegs. Wählt man eine sehr geringe Lernrate (z. B. $\eta = 0,0000001$), so werden nur sehr kleine Schritte vollführt. Wählt man die Lernrate jedoch höher, so ist auch die daraus resultierende Schrittweite ebenfalls größer.

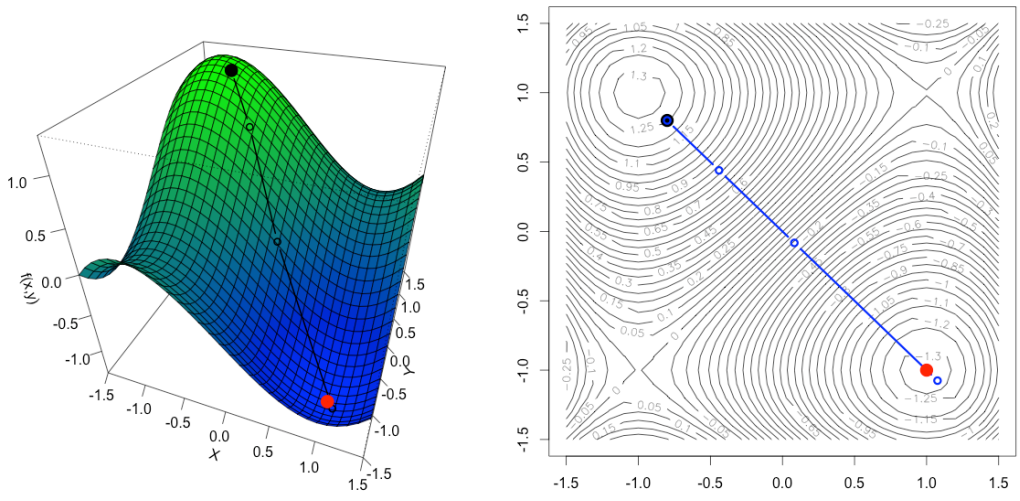


Abbildung 3.1: Der Gradientenabstieg für die Funktion $f(x, y) = \frac{x^3}{3} - x - (\frac{y^3}{3} - y)$ mit Höhenlinien-Diagramm.

In Abbildung 3.1 ist exemplarisch der Gradientenabstieg für eine Funktion $f(x, y)$ zu sehen. Man sieht, dass die Schrittweite des Abstiegs mit zunehmendem Gefälle größer wird. Im Höhenlinien-Diagramm ist ebenfalls zu sehen, dass ein zu großer Schritt das globale Minimum einer Funktion (roter Punkt im Diagramm) auch überspringen kann. So ist der Gradient beim dritten Schritt so groß, dass das globale Minimum der Funktion knapp übersprungen wird. Generell kann es im Zusammenhang mit dem Gradientenabstieg zu unterschiedlichen Problemen kommen (siehe Abbildung 3.2).

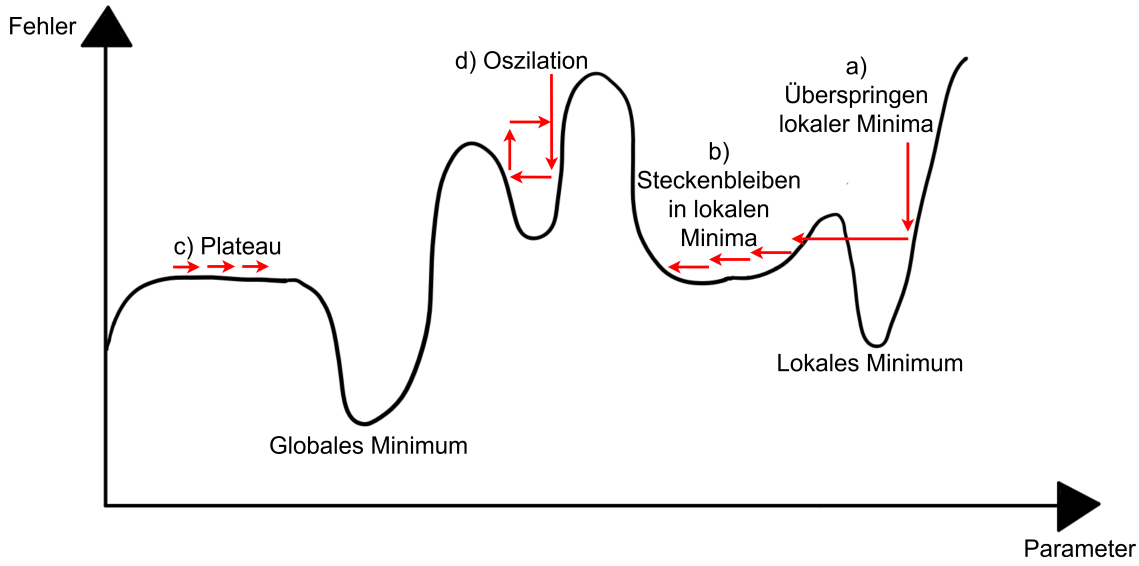


Abbildung 3.2: Probleme des Gradientenabstiegsverfahren.

So ist es möglich, dass man ein lokales Minimum wieder verlässt, weil man es durch zu große Schritte komplett überspringt. Dieses Problem tritt vor allem auf, wenn man eine zu hohe Lernrate wählt. Ein weiteres Problem ist das „Steckenbleiben“ in lokalen Minima, wenn die Schrittgröße zu klein ist, sodass man diese Minima nicht mehr verlassen kann. Dieses Phänomen tritt dann in Kombination mit geringen Lernraten auf, wobei auch die Kovergenzgeschwindigkeit unter einer zu geringen Lernrate leiden kann. Ein drittes Problem in diesem Zusammenhang ist das Durchlaufen von sogenannten *Plateaus*, da diese keinerlei Anstieg bzw. Gefälle besitzen. Dementsprechend stagniert der Gradient auf solchen Plateaus, wodurch sie dann

nicht verlassen werden können. Ein letztes Problem besteht in der *Oszillation* von Gradienten. Dies kann auftreten, wenn nach einem steilen Gefälle ein steiler Anstieg folgt.
Oftmals betrachtet man im Zusammenhang mit dem Gradientenabstieg auch Funktionen, welche als Argument (Input) wie auch Funktionswert (Output) einen Vektor haben: $f : \mathbb{R}^m \to \mathbb{R}^n$. Dadurch benötigt man dementsprechend alle partiellen Ableitungen bezügliche der Funktionswerte, wie auch Argumente der Funktion. Die Matrix, welche all diese partiellen Ableitungen enthält, wird als *Jacobi*-Matrix bezeichnet. Somit ist sie die Generalisierung des Gradienten für Funktionen, welche als Werte Vektoren verwenden. (vgl. [GBC16, S. 86])

Definition 3.11 (Jacobi-Matrix, nach [LRU20])
Gegeben sei eine Funktion $f : \mathbb{R}^m \to \mathbb{R}^n$ und $y = f(\mathbf{x})$. Die Jacobi-Matrix ist gegeben durch:

$$\mathbf{J}_{\mathbf{x}}(y) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_n}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_m} & \cdots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

In der Literatur wird die Jacobi-Matrix auch manchmal als die Transponierte der Definition 3.11 definiert. Da aber im Verlauf dieser Arbeit durchgängig angenommen wird, dass es sich bei allen Vektoren um Spalten-Vektoren handelt, passt die obige Definition besser, da die Jacobi-Matrix in diesem Fall nicht transponiert werden muss.
Für den Gradientenabstieg ist es notwendig, die Ableitung einer Funktion bilden zu können. Es kann jedoch auch der Fall auftreten, dass man die Ableitung mehrerer, miteinander verketteter Funktionen im Zusammenhang mit dem Gradientenabstieg bilden muss. Das ist dann der Fall, wenn z. B. eine Funktion $z = f(y)$ das Ergebnis einer anderen $y = g(x)$ als Argument enthält. Dabei muss dann zunächst die Ableitung von $g(x)$ bezüglich y gebildet werden, bevor die Ableitung von $f(y)$ gebildet werden kann. Dieses Vorgehen ist in der sogenannten *Kettenregel* festgehalten, welche wie folgt definiert ist:

Definition 3.12 (Kettenregel, nach [LRU20])
Wenn $y = g(x)$ und $z = f(y) = f(g(x))$, dann ist die Ableitung von z gegeben durch:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Wenn jedoch $z = f(u, v)$ mit $u = g(x)$ und $v = h(x)$, so ergibt sich aus der Kettenregel die Ableitung von z wie folgt:

$$\frac{dz}{dx} = \frac{\partial z}{\partial u}\frac{du}{dx} + \frac{\partial z}{\partial v}\frac{dv}{dx}$$

Diese Regel lässt sich dann in Kombination mit dem Gradientenabstieg verwenden, um die Ableitungen verketteter Funktionen berechnen zu können. Das wiederum führt dazu, dass der Gradientenabstieg dann auch für verkettete Funktionen möglich ist. Dieser Umstand wird vor allem im Kontext des Backpropagation-Algorithmus in Unterkapitel 3.3 eine wichtige Rolle einnehmen.

3.2.2 Varianten des Gradientenabstiegs

In Abschnitt 3.2.1 wurde schon das Gradientenabstiegsverfahren erläutert. Daher ist auch bekannt, zu welchen Problemen es in diesem Zusammenhang kommen kann. Aus Unterkapitel 3.1 ist ebenfalls bekannt, dass man zwischen mehreren Varianten des Lernens unterscheiden kann. Dementsprechend existieren auch für den Gradientenabstieg unterschiedliche Varianten mit bestimmten Stärken und Schwächen. Im Folgenden werden einige dieser Varianten kurz vorgestellt. Allerdings ist dieser Abschnitt nicht als vollständige Auflistung aller möglichen Varianten zu verstehen. Es finden nur jene Erwähnung, welche im späteren Verlauf der Arbeit nochmals thematisiert werden.
Der herkömmliche Gradientenabstieg wird für den kompletten Trainingsdatensatz durchgeführt. Das bedeutet, dass der Gradient über alle Beispiele des Datensatzes berechnet wird. Dieses Vorgehen wird dann als *Batch-Gradientenabstieg* bezeichnet und ist dem Offline-Lernen zuzuordnen. Dabei besteht das Problem dieser Variante darin, dass sie sehr langsam sein kann und auch nicht für Datensätze geeignet ist, die nicht komplett in den Hauptspeicher passen (vgl. [Rud17]).

Eine andere Variante wird hingegen für die einzelnen Datenpunkte eines Datensatzes berechnet. Dabei wird der tatsächliche Gradient (berechnet über den kompletten Datensatz) durch eine Approximation/Näherung (berechnet über den einzelnen Datenpunkten eines Datensatzes) ersetzt. Dieses Verfahren wird dann als stochastischer Gradientenabstieg oder Stochastic Gradient Descent (SGD) bezeichnet. SGD arbeitet im Gegensatz zum herkömmlichen (Batch-)Gradientenabstieg wesentlich schneller und kann für das Online-Lernen verwendet werden (vgl. [Rud17]). Der Nachteil besteht dabei jedoch darin, dass der Gradient häufiger berechnet werden muss, was den Gesamt-Rechenaufwand wesentlich erhöht.
Eine letzte Variante ist der sogenannte *Mini-Batch*-Gradientenabstieg. Dabei wird der Datensatz in mehrere kleinere Datensätze aufgeteilt, sodass der Gradient dann für diese Teilmengen berechnet wird. Das hat dann folgende Vorteile (vgl. [Rud17]):

1. Es reduziert die Varianz und den Rechenaufwand für die Parameter-Updates, was wiederum zu einer stabileren Konvergenz führen kann.

2. Es können optimierte Matrix-Operationen für die Berechnungen verwendet werden.

Diese Variante kann somit als ein Hybrid vom Batch-Gradientenabstieg und SGD interpretiert werden, da es die Kernideen von beiden Ansätzen nutzt und somit versucht, die Vorteile beider Verfahren zu kombinieren.
Ein Problem, was jedoch weiterhin besteht, ist die Wahl einer geeigneten Lernrate η. Im vorigen Unterkapitel 3.2.1 wurden schon bestimmte Probleme im Zusammenhang mit der Lernrate erläutert (Verlassen oder Überspringen lokaler Minima, Kovergenzgeschwindigkeit). Daher ist die Wahl der richtigen Lernrate bei dem Gradientenabstiegsverfahren von hoher Bedeutung.
Eine Reihe von Weiterentwicklungen wie der *Adagrad*- [DHS11] oder *Adam*-Optimierer [KB15] legen z. B. die Lernrate für jeden optimierbaren Parameter einzeln fest. Dabei ist der Kerngedanke, dass Parameter, die seltener auftreten, einen höheren Einfluss haben sollten als Parameter, die häufiger auftreten. Für einen Überblick zu weiteren Varianten und Weiterentwicklungen im Zusammenhang mit dem Gradientenabstieg wird der interessierte Leser auf die Arbeit [Rud17] von Ruder verwiesen.

3.3 Backpropagation

Lernen bedeutet im Zusammenhang mit neuronalen Netzen bestimmte Parameter des Netzwerkes zu optimieren. Dementsprechend kann ein neuronales Netz theoretisch Lernen, indem es folgende Parameter modifiziert:

1. Gewichte der Verbindungen zwischen den Neuronen
2. Bias/Schwellenwert der Neuronen
3. Aktivierungs-, Übertragungs- oder Ausgabefunktionen der Neuronen
4. Anzahl der Neuronen-Schichten
5. Anzahl der Neuronen pro Schicht
6. Art der Verbindungen zwischen den Schichten

Allerdings kann nicht jeder Lernalgorithmus auch alle aufgezählten Parameter beeinflussen und somit optimieren. Die Parameter, welche nicht vom Algorithmus modifiziert werden können, nennt man auch *Hyperparameter* und müssen vom Programmierer bzw. Entwickler des neuronalen Netzes selber vorgegeben werden.
In diesem Zusammenhang existieren verschiedene Lernalgorithmen, die unterschiedliche Parameter eines neuronalen Netzes beeinflussen bzw. optimieren können. In diesem Unterkapitel wird *Backpropagation* als ein solcher Lernalgorithmus erläutert, welcher überwiegend für das *überwachte Lernen* verwendet wird. Dabei kann dieser Algorithmus die Schwellenwerte und Verbindungsgewichte der Neuronen als beeinflussbare Parameter optimieren. Als Hyperparameter werden dementsprechend die Aktivierungs-, Übertragungs- oder Ausgabefunktionen der Neuronen, die Anzahl der Neuronen-Schichten und Neuronen pro Schicht und die Art der Verbindungen zwischen den Schichten angenommen.
Der nächste Abschnitt wird die Funktionsweise des Backpropagation-Algorithmus im Detail an einem Beispiel erläutern. Dabei wird ebenfalls erklärt, in welcher Weise Gradienten, Jacobi-Matrix und Kettenregel ihre Anwendung dabei finden. Zu erwähnen ist, dass das verwendete Beispiel sich auf den Batch-Gradientenabstieg bezieht. Generell kann der Backpropagation-Algorithmus jedoch auch in Kombination mit den anderen Varianten des Gradientenabstiegs verwendet werden.

3.3.1 Funktionsweise des Algorithmus

Der Backpropagation-Algorithmus lässt sich generell in zwei Phasen unterteilen:

1. **Forward-Pass:** Dem neuronalen Netz wird eine Eingabe präsentiert, welche es verarbeitet und die dementsprechende Ausgabe produziert.

2. **Backward-Pass:** Auf Basis des ermittelten Fehlers zwischen dem vom neuronalen Netz vorhergesagten und dem in der Realität beobachteten Output, wird die Fehlerrückführung durchgeführt. Das bedeutet, der Fehlerwert wird zurück in das neuronale Netz gegeben und es werden von hinten nach vorne die jeweiligen Parameter des Netzes auf Basis des Fehlers angepasst.

Dieses Vorgehen ist exemplarisch in Abbildung 3.3 zu sehen. Während die Informationsverarbeitung bzw. Funktionsweise eines neuronalen Netzes (also der Forward-Pass) in Unterkapitel 2.2 schon näher erläutert wurde, wird sich dieser Abschnitt nun auf die Fehlerrückführung, also den Backward-Pass, beziehen.

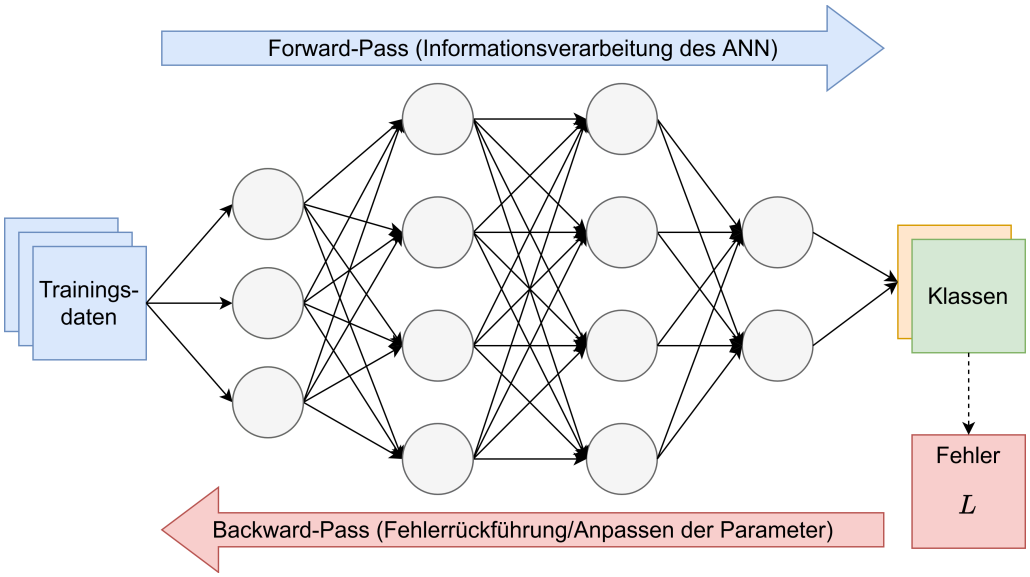


Abbildung 3.3: Der Backpropagation-Algorithmus.

In Unterkapitel 3.1 wurden die verschiedenen Fehlerfunktionen vorgestellt, welche im Zusammenhang mit neuronalen Netzen verwendet werden und deren Fehler berechnen können. Das Unterkapitel 3.2 hat in die Gradienten-basierte Optimierung

eingeführt und daher ist bekannt, dass der Gradient einer Funktion verwendet werden kann, um diese zu minimieren. In dem Unterkapitel 2.2 wurde ebenfalls schon beschrieben, dass neuronale Netze als Funktionsapproximatoren aufgefasst werden können. Die Kernidee des Backpropagation-Algorithmus ist nun, die Parameter der vom Netzwerk approximierten Funktion mithilfe des Gradientenabstiegs zu optimieren, sodass der Fehler des neuronalen Netzes minimiert wird.
Der Backpropagation-Algorithmus berechnet zunächst den Gradienten der Fehlerfunktion eines neuronalen Netzes, um anschließend auf dieser Basis die Gradienten der einzelnen Parameter des Netzwerkes ermitteln zu können. Mithilfe dieser Gradienten werden dann die Gewichte der Verbindungen und der Bias der Neuronen so verändert, dass der Fehler des jeweiligen neuronalen Netzes Schritt für Schritt geringer wird. Dieses Vorgehen wird solange wiederholt, bis das neuronale Netz eine gewisse Qualität oder die maximale Iterationsanzahl erreicht hat.
Dabei ist die Berechnung der einzelnen Gradienten (und somit des Anteils eines Kantengewichts oder Bias am Gesamtfehler des NN) nicht trivial. In diesem Zusammenhang kommt die in Abschnitt 3.2.1 vorgestellte Kettenregel zum Einsatz. Denn die einzelnen Neuronen berechnen unterschiedliche Funktionen. Diese sind jedoch über die Verbindungen des neuronalen Netzes miteinander verkettet. Mittels der Kettenregel lassen sich dann die einzelnen Gradienten dieser verketteten Funktionen ermitteln.
Um das ganze Vorgehen einmal zu verdeutlichen, wird es im Folgenden an einem einfachen Beispiel erklärt. Dazu werden das Beispiel und die einzelnen Erläuterungen sinngemäß aus der Arbeit [LRU20] übernommen. Dafür wird jedoch zunächst erläutert, was man unter einem Berechnungsgraphen versteht.
In Abbildung 3.4 ist ein solcher Berechnungsgraph zu sehen, welcher den Datenfluss bzw. die Informationsverarbeitung eines neuronalen Netzes modelliert. Formal gesehen handelt es sich bei einem Berechnungsgraphen um einen gerichteten azyklischen Graphen. Die einzelnen Knoten besitzen einen Operanden und (optional) einen Operator. Ein Operand kann dabei ein Skalar, Vektor oder eine Matrix sein und ein Operator ist eine Operation der linearen Algebra (wie z. B. + oder auch $\times$), eine Aktivierungsfunktion oder eine Fehlerfunktion. Wenn ein Knoten beides besitzt (Operator und Operand), so steht der Operator stets über dem Operanden. Besitzt

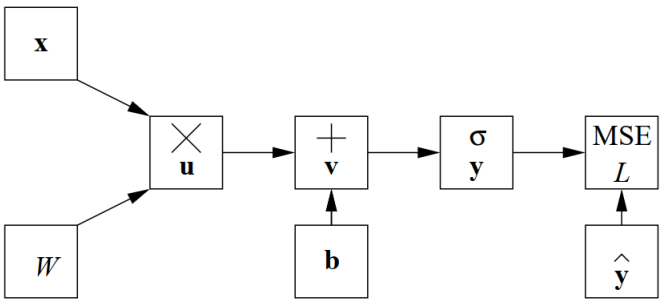


Abbildung 3.4: Der Berechnungsgraph eines einfachen neuronalen Netzes (Quelle: [LRU20]).

ein Knoten nur einen Operanden, so entspricht der Output des Knotens dem Wert, welcher mit seinem jeweiligen Operanden assoziiert ist. Der Output eines Knotens mit Operand und Operator entspricht dem Resultat des Anwendens des Operators auf die Vorgänger im Graph und der anschließenden Assoziation des Ergebnisses mit dem jeweiligen Operanden. (vgl. [LRU20, S. 540])
Für das Beispiel aus der Abbildung 3.4 würden sich also folgende Zuweisungen für die Operanden ergeben (vgl. [LRU20, S. 541]):

$$\mathbf{u} = \mathbf{W}^{\mathbf{T}}\mathbf{x}$$
$$\mathbf{v} = \mathbf{u} + \mathbf{b}$$
$$\mathbf{y} = \sigma(\mathbf{v})$$
$$L = \text{MSE}(\mathbf{y}, \hat{\mathbf{y}})$$

Die Variable $\mathbf{x}$ repräsentiert dabei den Input des NN und $\mathbf{W}$ die Gewichtsmatrix. Somit steht $\mathbf{u}$ also für den Input der Neuronen-Schicht ohne den dazugehörigen Bias $\mathbf{b}$. Dieser wird in $\mathbf{v}$ dazu addiert, sodass $\mathbf{v}$ dann den Input einer Neuronen-Schicht gemäß Definition 2.10 modelliert. Der Aktivierungszustand der Neuronen-Schicht (siehe Definition 2.11) ist dann folglich mit $\mathbf{y}$ assoziiert, was auch gleichzeitig den Output des NN darstellt (bei der Output-Funktion der einzelnen Neuronen handelt es sich um die Identitätsfunktion). Die Variable L ist dann der Fehler des neuronalen Netzes.
Wie in Abbildung 3.3 dargestellt, erfolgt die Fehlerrückführung in einem neuronalen Netz von hinten nach vorne. Das bedeutet, dass zunächst der Gradient für die

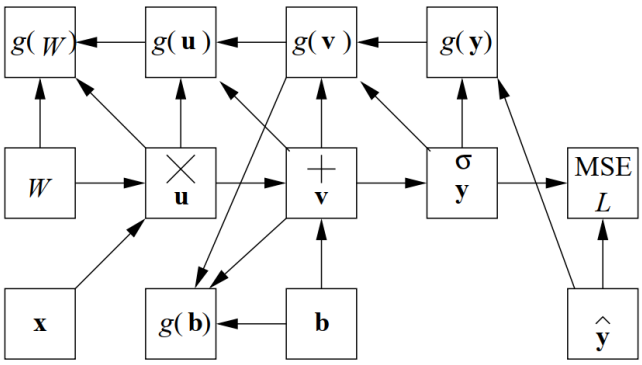


Abbildung 3.5: Der Berechnungsgraph mit Gradienten-Knoten (Quelle: [LRU20]).

Fehlerfunktion berechnet werden muss. In dem verwendeten Beispiel handelt es sich dabei um den MSE (vgl. [LRU20, S. 542]):

$$g(\mathbf{y}) = \nabla_{\mathbf{y}}(L) = 2(\mathbf{y} - \hat{\mathbf{y}})$$

Somit ist der Gradient $g(\mathbf{y})$ bezüglich des Fehlers L berechnet. Um die Gradienten der verbleibenden Parameter des neuronalen Netzes zu berechnen, geht man ebenfalls von hinten nach vorne im Berechnungsgraph vor. Man wählt einen Knoten, für dessen Nachfolger im Berechnungsgraph die jeweiligen Gradienten schon berechnet wurden und wendet die Kettenregel an, um den eigenen Gradienten des jeweiligen Knotens berechnen zu können. Dazu wird der Berechnungsgraph aus Abbildung 3.4 um jeweils einen Knoten für jeden Gradienten erweitert. Der daraus resultierende Berechnungsgraph ist in Abbildung 3.5 zu sehen. (vgl. [LRU20, S. 542])
Daraus ergibt sich für den Gradienten von $\mathbf{v}$ bezüglich L folgender Ausdruck (vgl. [LRU20, S. 542]):

$$g(\mathbf{v}) = \nabla_{\mathbf{v}}(L) = \mathbf{J}_{\mathbf{v}}(\mathbf{y})\nabla_{\mathbf{y}}(L) = \mathbf{J}_{\mathbf{v}}(\mathbf{y})g(\mathbf{y})$$

Da die Aktivierungsfunktion σ auf einer ganzen Neuronen-Schicht (und nicht nur auf einem einzelnen Neuron) berechnet wird, ergibt sich für die Berechnung von $\mathbf{y}$ eine Funktion mit vektorieller Eingabe, wie auch Ausgabe: $f : \mathbb{R}^m \to \mathbb{R}^n$. Das bedeutet, dass für die Berechnung des Gradienten von $\mathbf{v}$ die Jacobi-Matrix $\mathbf{J}_{\mathbf{v}}(\mathbf{y})$ verwendet werden muss. (vgl. [LRU20, S. 543])

Für die Berechnung dieser Matrix muss jedoch zunächst die Ableitung für die logistische Sigmoid-Funktion (siehe Definition 2.13) berechnet werden (vgl. [LRU20, S. 543]):

$$\frac{\partial y_i}{\partial v_j} = \begin{cases} y_i(1-y_i) & \text{wenn } i = j \\ 0 & \text{sonst} \end{cases}$$

Die sich daraus ergebene Jacobi-Matrix ist somit eine Diagonalmatrix (vgl. [LRU20, S. 543]):

$$\mathbf{J_v}(\mathbf{y}) = \begin{bmatrix} y_1(1-y_1) & 0 & \cdots & 0 \\ 0 & y_2(1-y_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & y_n(1-y_n) \end{bmatrix}$$

Somit ist auch der Gradient $g(\mathbf{v})$ berechnet, sodass nun $g(\mathbf{b})$ und $g(\mathbf{u})$ ebenfalls berechnet werden können. Für diese beiden Gradienten ergeben sich folgende Ausdrücke:

$$g(\mathbf{b}) = \nabla_{\mathbf{b}}(L) = \mathbf{J_b}(\mathbf{v})\nabla_{\mathbf{v}}(L) = \mathbf{J_b}(\mathbf{v})g(\mathbf{v})$$
$$g(\mathbf{u}) = \nabla_{\mathbf{u}}(L) = \mathbf{J_u}(\mathbf{v})\nabla_{\mathbf{v}}(L) = \mathbf{J_u}(\mathbf{v})g(\mathbf{v})$$

Da $\mathbf{v} = \mathbf{u} + \mathbf{b}$ gilt, ergibt sich folgender Sachverhalt für die Berechnung der jeweiligen Jacobi-Matrix (vgl. [LRU20, S. 544]):

$$\mathbf{J_b}(\mathbf{v}) = \mathbf{J_u}(\mathbf{v}) = \mathbf{I_n}$$

wobei $\mathbf{I_n}$ die $n \times n$ Einheitsmatrix ist. Das bedeutet letztendlich, dass die Gradienten von $\mathbf{u}$ und $\mathbf{b}$ dem Gradienten von $\mathbf{v}$ entsprechen (vgl. [LRU20, S. 544]):

$$g(\mathbf{b}) = g(\mathbf{u}) = g(\mathbf{v})$$

Der letzte Schritt wäre nun den Gradienten für $\mathbf{W}$ bezüglich L zu berechnen. An dieser Stelle ist daran zu erinnern, dass $\mathbf{u} = \mathbf{W^T x}$ ist. Bei $\mathbf{W}$ handelt es sich allerdings um die Gewichtsmatrix der Neuronen-Schicht, sodass das bisherige Vorgehen

nicht einfach wieder angewendet werden kann, da es sich bei $\mathbf{W}$ um keinen Vektor handelt, so wie es bisher der Fall war. Wenn man jedoch die Definition 2.9 einer Gewichtsmatrix betrachtet, wird klar, dass man die Matrix in einzelne Komponenten für die Verbindungen der einzelnen Neuronen der Schicht zerteilen kann. So repräsentiert die Spalte $\mathbf{w_j}$ der Matrix $\mathbf{W}$ den Gewichtsvektor des j-ten Neurons. (vgl. [LRU20, S. 544])
Dadurch können die Gewichtsvektoren einer Schicht einzeln betrachtet und dementsprechend auch ihre Gradienten einzeln berechnet werden (vgl. [LRU20, S. 544]):

$$g(\mathbf{w_j}) = \nabla_{\mathbf{w_j}}(L) = \mathbf{J}_{\mathbf{w_j}}(\mathbf{u})\nabla_{\mathbf{u}}(L) = \mathbf{J}_{\mathbf{w_j}}(\mathbf{u})g(\mathbf{u})$$

Man weiß, dass $u_j = \mathbf{w_j^T x}$ gilt und dass die anderen u_k keine Abhängigkeiten von $\mathbf{w_j}$ für $j \neq k$ besitzen (vgl. [LRU20, S. 544]). Die partielle Ableitung von u_j nach $\mathbf{w_j}$ wäre dann:

$$\frac{\partial u_j}{\partial \mathbf{w_j}} = \frac{\partial \mathbf{w_j^T x}}{\partial \mathbf{w_j}} = \mathbf{x}$$

Dadurch ist die Jacobi-Matrix $\mathbf{J}_{\mathbf{w_j}}(\mathbf{u})$ an allen Stellen null, bis auf die j-te Spalte, welche dann gleich $\mathbf{x}$ ist (vgl. [LRU20, S. 544]). Dadurch kann die obige Berechnung von $g(\mathbf{w_j})$ vereinfacht werden (vgl. [LRU20, S. 544]):

$$g(\mathbf{w_j}) = g(u_j)\mathbf{x}$$

Als Output bekommen wir nun also den Gradienten eines Gewichtsvektors des Fehlers L bezüglich des Gewichtsvektors $\mathbf{w_j}$. Dadurch, dass nun alle notwendigen Gradienten berechnet wurden, können die Parameter des neuronalen Netzes, also die Bias von Neuronen wie auch die Gewichte der einzelnen Verbindungen, so angepasst werden, dass der Gesamtfehler des NN reduziert wird. Dieses Vorgehen wird dann so oft für die Datenpunkte eines Trainingsdatensatzes wiederholt, bis der jeweilige Gradient keine weitere Verbesserung des Gesamtfehlers mehr zulässt.

3.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Fehlerfunktion bezüglich neuronaler Netze vorgestellt. Diese stellen die Basis für das Gradienten-basierte Lernen dar. Dabei wird versucht, den Fehlerwert eines neuronalen Netzes zu minimieren, indem die einzelnen Parameter durch das Gradientenabstiegsverfahren optimiert werden. In diesem Zusammenhang existieren unterschiedliche Varianten dieses Vorgehens, welche jeweils ihre eigenen Vor- und Nachteile mitbringen. Mit dem Backpropagation-Algorithmus wurde ein Lernalgorithmus vorgestellt, der den Gradientenabstieg für das überwachte Lernen verwendet.
Im nächsten Kapitel 4 werden weitere Optimierungsverfahren aus dem Gebiet der Neuroevolution vorgestellt. Dabei unterliegen diese Algorithmen einer gänzlich anderen Herangehensweise, was sie zu einer interessanten Alternative zu dem Gradienten-basierten Lernen macht.

4 Neuroevolution

Dieses Kapitel soll einen Überblick in dem Themengebiet der Neuroevolution (NE) geben. Bei NE-Techniken handelt es sich um eine Variante von evolutionären Algorithmen. Diese Art von Algorithmen arbeiten mit einer Population von neuronalen Netzen, welche im Folgenden auch als Kandidaten bezeichnet werden. Im Vordergrund steht also die Entwicklung von neuronalen Netzen mittels evolutionärer Verfahren. Dabei kann die Neuroevolution aus zwei verschiedenen Blickwinkeln betrachtet werden:

1. Alternative zu herkömmlichen Lernverfahren: In diesem Zusammenhang sollen NE-Verfahren andere Lernalgorithmen wie z. B. Backpropagation vollständig ersetzen. Das bedeutet, dass sie als Optimierungsalgorithmus verwendet werden, welcher die einzelnen Parameter eines neuronalen Netzes so bestimmen soll, dass eine möglichst zufriedenstellende Lösung gefunden wird.

2. Ergänzung zu herkömmlichen Lernverfahren: Hierbei dient die Neuroevolution nicht als Alternative, sondern als Ergänzung zu anderen Lernalgorithmen. Dabei kann mithilfe von Neuroevolution z. B. die Topologie eines neuronalen Netzes angepasst werden, während die Verbindungsgewichte weiterhin über herkömmliche Algorithmen wie Backpropagation trainiert werden.

Im Kontext der Neuroevolution wird weiterhin zwischen zwei verschiedenen Darstellungsweisen unterschieden: dem *Genotyp* und dem *Phänotyp*. Dabei handelt es sich im Grunde genommen um zwei verschiedene Repräsentationen eines Kandidaten (bzw. eines neuronalen Netzes). So gibt es NE-Algorithmen, bei denen der Genotyp dem Phänotyp entspricht. Das bedeutet in diesem Zusammenhang, dass jedes Element des Phänotyp (also des neuronalen Netzes) explizit im Genotyp kodiert ist. Bei anderen NE-Algorithmen hingegen unterscheidet sich der Genotyp vom Phänotyp.

Diese Art der Kodierung wird auch als *indirekt* bezeichnet. Der Abbildungsprozess zwischen dem Geno- und Phänotyp wird auch als *Genotyp-Phänotyp-Mapping* bezeichnet.
In Unterkapitel 3.3 wurden die verschiedenen Parameter eines neuronalen Netzes vorgestellt und die Aussage getroffen, dass nicht alle Lernverfahren auch alle Parameter optimieren können. Die NE-Techniken sind im Allgemeinen in der Lage, alle Parameter eines neuronalen Netzes zu optimieren. Jedoch beschränken sich einzelne NE-Techniken auch nur auf die Optimierung der Verbindungsgewichte (äquivalent zum Backpropagation-Verfahren). Diese Teilmenge von NE-Techniken wird auch als *klassische* Neuroevolution bezeichnet (vgl. [Sta+19]).
Das Gebiet der Neuroevolution lässt sich zudem in verschiedene Teilgebiete unterteilen. Dieses Kapitel wird sich diesen Teilgebieten widmen und ihre neusten Entwicklungen vorstellen. Als Grundlage wird der Artikel [Sta+19] von Stanley u. a. aus dem Jahr 2019 verwendet, welcher einen Überblick über die Neuroevolution als Forschungsgebiet liefert.
Zunächst wird dazu in Unterkapitel 4.1 eine weitverbreitete NE-Technik vorgestellt, die für den weiteren Verlauf von Relevanz ist. Zusätzlich wird aufgezeigt, ob sich die Neuroevolution im Allgemeinen auch für die Entwicklung von tiefen neuronalen Netzen, auch Deep Neural Network (DNN)[1] genannt, verwenden lässt. In diesem Zusammenhang wird auch die Parallelisierbarkeit von NE-Techniken thematisiert werden. Anschließend wird in Unterkapitel 4.2 das Teilgebiet der *Diversität* und *Neuheit* im Zusammenhang mit der Neuroevolution untersucht, wobei diesbezüglich ebenfalls unterschiedliche Algorithmen vorgestellt werden. In Unterkapitel 4.3 werden Techniken zur Komprimierung bzw. indirekten Kodierung von neuronalen Netzen vorgestellt, die gerade im Zusammenhang mit der Entwicklung von DNNs durch die Neuroevolution eine große Rolle spielen. Auch diese werden im weiteren Verlauf dieser Arbeit eine wichtige Rolle einnehmen und dementsprechend in den folgenden Kapiteln wieder aufgegriffen.

[1] Bei DNN handelt es sich um neuronale Netze, die eine Vielzahl an verdeckten Schichten besitzen. Die Anzahl der Schichten kann sich dabei in die Hunderte oder auch Tausende erstrecken.

4.1 Entwicklung der Neuroevolution

Wie schon erwähnt können NE-Techniken als Alternative zu anderen Lernalgorithmen verstanden werden. In Unterkapitel 3.3 wurde schon erläutert, dass der Backpropagation-Algorithmus eine oft verwendete Technik ist, um die Gewichte eines neuronalen Netzes trainieren zu können. Erste Versuche im Zusammenhang mit der klassischen Neuroevolution hatten dementsprechend das Ziel, Verfahren hervorzubringen, welche ein adäquater Ersatz zum Backpropagation-Algorithmus sein können (vgl. [Sta+19]). Dabei bleibt jedoch eine Fragestellung ungeklärt: Wie entscheidet man überhaupt, welche Neuronen wie miteinander verbunden sind bzw. wie findet man die beste Topologie[2] eines neuronalen Netzes? Schnell verschob sich der Fokus der NE-Forschung im Allgemeinen auf genau diese Fragestellung. Anstatt NE-Algorithmen nur für das Training der Verbindungsgewichte zu verwenden (= klassische Neuroevolution), sollte nun die komplette Topologie gleichzeitig mitentwickelt werden (vgl. [Sta+19]).
Mit diesem Vorhaben sind jedoch unterschiedliche Probleme verknüpft. Zum einen haben die einzelnen Kandidaten (also die neuronalen Netze) einer Population nicht mehr dieselbe Topologie. Während sich bei der Neurevolution zuerst nur die Verbindungsgewichte der einzelnen Kandidaten unterschieden haben, können sie sich nun zusätzlich in ihren Verbindungsmustern, Neuronen und auch Aktivierungsfunktionen von einander abgrenzen. Das verursacht im Zusammenhang mit der Rekombination das Problem, dass festgestellt werden muss, welche Teile topologisch unterschiedlicher neuronaler Netze miteinander übereinstimmen und welche nicht. Ein weiteres Problem ist, dass wenn man durch Mutation z. B. ein neues Neuron zu einem Netz hinzufügt, sich diese Veränderung oftmals zunächst negativ auf dessen Performance auswirkt (vgl. [SM02]). Somit würde das resultierende neuronale Netz im nächsten Selektionsschritt sofort wieder aussortiert werden.

[2] Bei der Topologie eines neuronalen Netzes handelt es sich um seine Architektur. Die Topologie umfasst dabei die Organisation der Neuronen-Schichten, wie sie miteinander verbunden sind und wie viele Neuronen pro Schicht verwendet werden. Auch die Aktivierungsfunktionen der einzelnen Neuronen(-Schichten) können als Bestandteil der Topologie aufgefasst werden.

4.1.1 NeuroEvolution of Augmented Topologies

In der Arbeit [SM02] liefern *Stanley* und *Miikkulainen* mit dem NEAT-Algorithmus (**N**euro**E**volution of **A**ugmented **T**opologies) erstmals einen Ansatz, welcher den eben genannten Problemen begegnen kann. Das grundlegende Prinzip von NEAT ähnelt dabei dem von genetischen Algorithmen:

1. *Initialisierung*: Die Kandidaten der ersten Generation werden zufällig generiert.

2. *Evolution*:

   a) *Evaluation*: Die Kandidaten der aktuellen Generation bekommen einen Fitness-Wert zugeordnet, je nachdem wie gut sie eine Aufgabe lösen bzw. wie gut sie sich in einer gegebenen Umgebung verhalten.

   b) *Selektion*: Die stärksten Kandidaten überleben, während die Schwächsten aussortiert werden.

   c) *Rekombination*: Die überlebenden Kandidaten werden miteinander gekreuzt, um daraus eine neue Generation zu erzeugen.

   d) *Mutation*: Eine zufällige Eigenschaft eines jeden Kandidaten der neuen Generation wird verändert.

3. Wiederhole die Schritte a) bis d) solange bis eine Obergrenze für die Anzahl der Generationen erreicht ist oder ein Kandidat eine zufriedenstellende Performance aufweist.

Dabei beginnt NEAT mit der kleinstmöglichen Topologie eines neuronalen Netzes, was bedeutet, dass die Kandidaten der ersten Generation nur aus Input- und Output-Schicht bestehen und sich lediglich in ihren zufällig generierten Verbindungen und den jeweiligen Gewichten unterscheiden. Dadurch soll NEAT das kleinstmögliche neuronale Netz für eine spezifische Aufgabe generieren können, um somit die Dimensionalität zu reduzieren. Die nächsten Absätze erläutern die einzelnen Teile des NEAT-Algorithmus im Detail.

Kodierung

NEAT arbeitet mit einer direkten Kodierung. Das bedeutet, dass jedes Element des Phänotypen eines Kandidaten explizit in seinem Genotypen kodiert ist. In Abbildung 4.1 ist ein Beispiel dafür zu sehen, wie NEAT den jeweiligen Genotypen eines neuronalen Netzes darstellt.

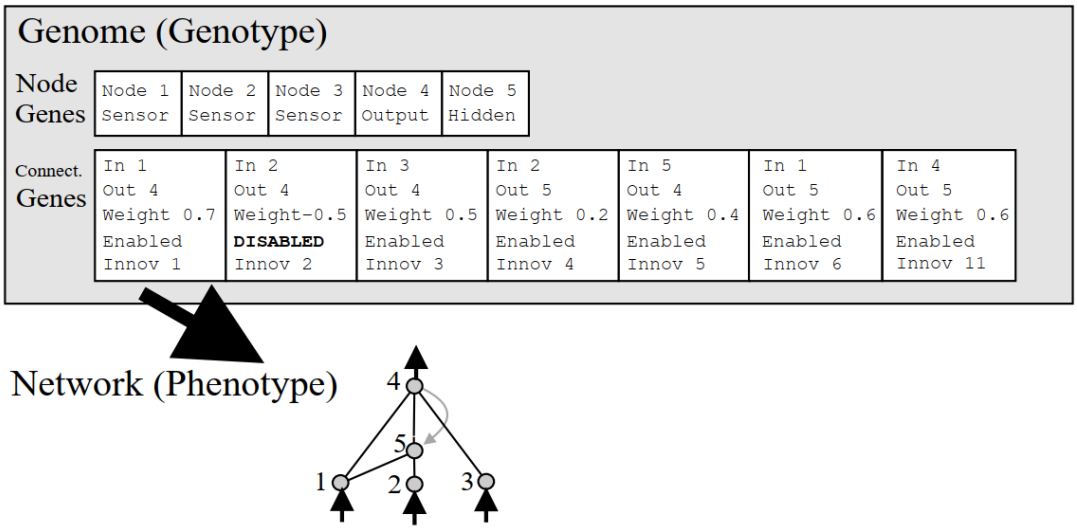


Abbildung 4.1: Darstellung des Genotypen eines neuronalen Netzes in NEAT (Quelle: [SM02]).

Dabei fällt auf, dass der Genotyp sich aus den *Knoten-* und *Verbindungsgenen* zusammensetzt (in der Abbildung als *Node* und *Connection Genes* bezeichnet). Jedes Knotengen enthält dabei die Identifikationsnummer (ID) und den Typ des Knotens (Input-, Output- oder verdeckter/innerer Knoten). Ein Verbindungsgen enthält die ID des Eingabe- und Ausgabe-Knotens, das Gewicht und eine Innovationsnummer (welche an späterer Stelle noch genauer erläutert wird). Zusätzlich kann für jede Verbindung festgelegt werden, ob diese im Phänotyp aktiviert ist oder nicht. Wie schon erwähnt handelt es sich bei dieser Art der Darstellung um eine direkte Kodierung, bei der jedes Element des neuronalen Netzes explizit repräsentiert wird. Folglich findet an dieser Stelle keine Kompression statt, weshalb der komplette Parameter-Suchraum über alle Elemente des neuronalen Netzes betrachtet werden muss.

Niching-Mechanismus

Bei der Selektion macht sich der Algorithmus den sogenannten *Niching*-Mechanismus zunutze. Dabei werden die einzelnen Kandidaten für die Selektion in Spezies unterteilt, sodass neuartige Kandidaten, welche sich von den Bisherigen stark unterscheiden, genügend Zeit bekommen, sich zu entwickeln und nicht schon im nächsten Iterationsschritt wieder aussortiert werden. Die Selektion wird dann nur lokal in den jeweiligen Spezies ausgeführt, sodass sichergestellt wird, dass immer mindestens einer der Kandidaten einer jeden Spezies überlebt. Dadurch bekommt ein Kandidat, welcher durch Mutation entstanden ist und so neuartig ist, dass er in eine eigene Spezies eingeteilt wird, genügend Zeit, sich zu entwickeln und somit sein wahres Potenzial zu zeigen. Damit wird dem Problem begegnet, dass manche Mutationen sich zunächst negativ auf die Performance eines Kandidaten auswirken und jene dementsprechend schnell aussortiert werden würden.

Mutation

Die Mutation umfasst zwei Varianten: So wird entweder ein Verbindungsgewicht oder die Struktur des neuronalen Netzes verändert. Die Mutation der Verbindungen umfasst dabei ausschließlich das Gewicht einer Verbindung. Strukturelle Mutationen hingegen können in zwei unterschiedlichen Weisen auftreten. Die erste Variante fügt eine neue Verbindung mit zufälligem Gewicht zwischen zwei bisher nicht verbundenen Knoten hinzu. Bei der anderen Variante wird eine vorhandene Verbindung $n_1 \rightarrow n_2$ zerteilt, sodass ein neuer Knoten n_3 an der Stelle hinzugefügt wird, wo vorher die Verbindung war. Die alte Verbindung wird dabei *deaktiviert* und zwei neue Verbindungen ($n_1 \rightarrow n_3$ und $n_3 \rightarrow n_2$) werden hinzugefügt. Dabei bekommt die Verbindung $n_1 \rightarrow n_3$ ein Gewicht von 1 zugewiesen und die Verbindung $n_3 \rightarrow n_2$ das gleiche Gewicht wie $n_1 \rightarrow n_2$. Dadurch wirkt sich das Hinzufügen eines Knotens nicht allzu stark auf das Verhalten des neuronalen Netzes aus, als wenn man die Gewichte der neuen Verbindungen komplett zufällig generieren würde. In Kombination mit dem Niching-Mechanismus hat ein neu entstandener Kandidat dadurch genügend Zeit, sich so zu entwickeln, um von seiner neuen Topologie bzw. Struktur Gebrauch machen zu können.

Rekombination durch historische Markierungen

Das Problem der Rekombination neuronaler Netze mit verschiedenen Topologien löst der Algorithmus durch *historische Markierungen*. Dazu werden die vorher schon erwähnten Innovationsnummern der Verbindungsgene im Genotypen verwendet (siehe Abb. 4.1). Haben zwei Verbindungsgene unterschiedlicher Kandidaten die gleiche Innovationsnummer, so werden sie als identisch angenommen. Dadurch kann für zwei topologisch unterschiedliche neuronale Netze ermittelt werden, welche Teile identisch sind und welche nicht. In Abbildung 4.2 ist die Rekombination zweier neuronaler Netze exemplarisch dargestellt.

Bei gleichen Verbindungsgenen wird eines der Elternteile zufällig ausgewählt, sodass auch das Verbindungsgewicht des jeweiligen Elternteils für das Kind übernommen wird. Unterschiedliche Verbindungsgene werden hingegen alle aus den jeweiligen Elternteilen übernommen.

Zusammengefasst generiert NEAT pro Generation eine Population neuronaler Netze. Dabei verhindert das Niching, dass neuartige/innovative Kandidaten in der nächsten Generation sofort wieder wegen zu schlechter Fitness aussortiert werden, was es ihnen erlaubt, ihr volles Potenzial auszuschöpfen. Die Mutation kann bei NEAT entweder die Verbindungsgewichte oder Struktur eines neuronalen Netzes modifizieren. Bei der Rekombination werden die einzelnen neuronalen Netze bzw. Kandidaten einer Generation miteinander gekreuzt. Dabei werden die historischen Markierungen bzw. Innovationsnummern der Verbindungen genutzt, um topologisch identische und unterschiedliche Bereiche der neuronalen Netze identifizieren zu können. Dadurch ist es möglich, die entsprechenden Netze miteinander zu kreuzen.

Die Kombination all dieser Mechanismen erlauben es NEAT, die Struktur eines neuronalen Netzes mit jeder Generation komplexer werden zu lassen. Dadurch muss die Topologie nun nicht mehr vom Entwickler selbst vorgegeben werden, sondern wird implizit durch den NEAT-Algorithmus generiert.

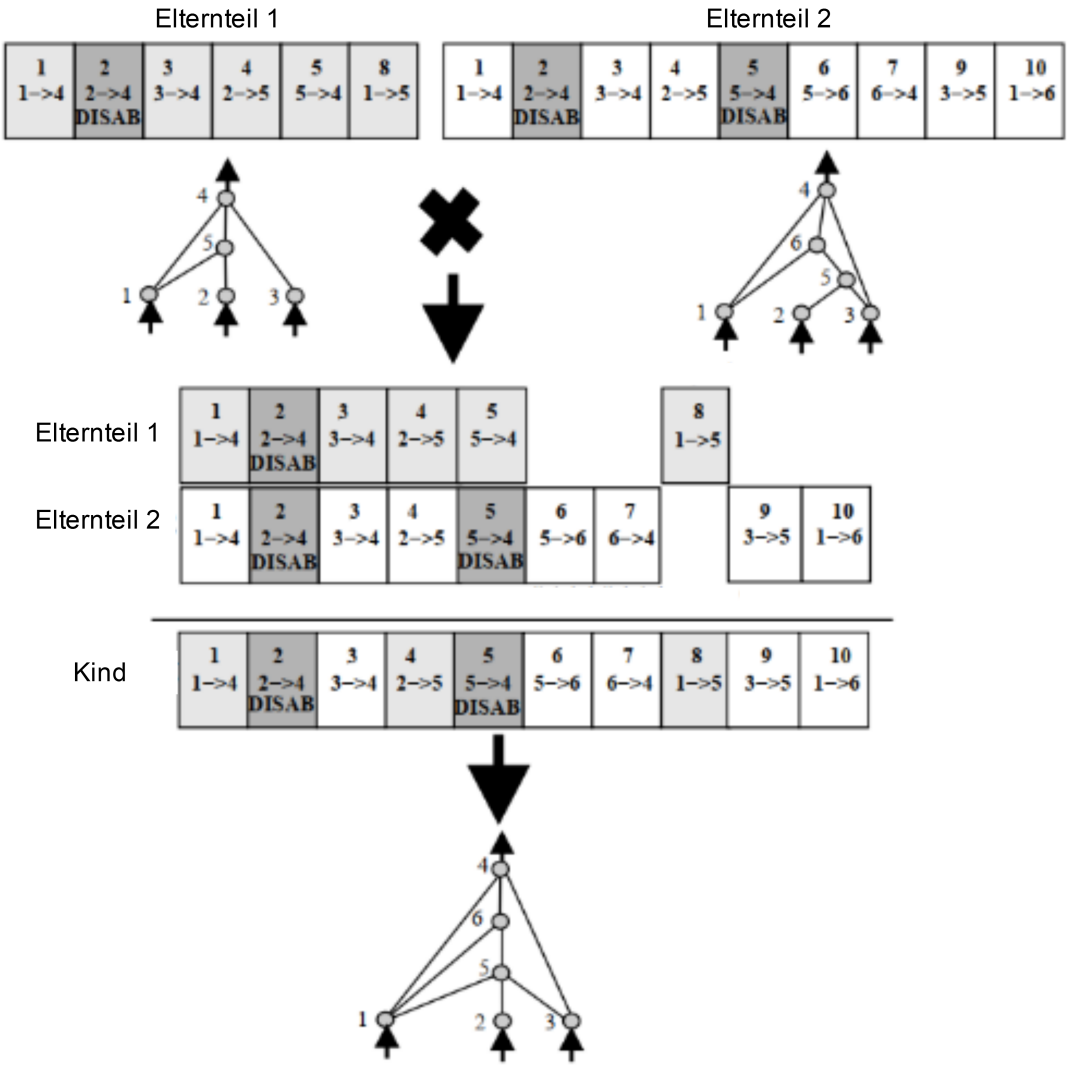


Abbildung 4.2: Rekombination in NEAT (in Anlehnung an [SM02]).

4.1.2 Skalierbarkeit

Die ersten Erfolge in der klassischen Neuroevolution werden oft auf die Entwicklung von NN-Controllern für die Robotik zurückgeführt (vgl. [Sta+19]). Ein Beispiel dafür war die Generierung von Gattern durch Neuroevolution, mittels dessen der Sony Aibo Roboter laufen lernte (vgl. [Hor+00]). Ein anderes Beispiel für die erfolgreiche Anwendung ist die Entwicklung von neuronalen Netzen und Morphologien für Roboter, welche anschließend 3-D-gedruckt wurden und sich in der realen Welt bewegen konnten (vgl. [LP00]). Aber auch außerhalb der Robotik hat die Neuroevolution erfolgreich Anwendung gefunden. So wurde z. B. durch den NEAT-Algorithmus die bisher genauste Schätzung der Masse des Top-Quarks berechnet (vgl. [Aal+09]). Neuroevolution wurde ebenso dazu genutzt, um die Ursprünge von Regularien, Modulen und Hierarchien in biologischen Netzwerken, wie z. B. den Hirnen von Tieren, zu untersuchen (vgl. [CML13; HCM14]).
Obwohl beeindruckend, vor allem zu ihrer Zeit, handelte es sich bei all diesen erfolgreichen Anwendungen um winzige neuronale Netze nach modernen Standards, die aus Hunderten oder Tausenden von Verbindungen bestehen, anstatt der Millionen, die man üblicherweise in heutigen tiefen neuronalen Netzen vorfindet. (vgl. [Sta+19])
Neuere Forschung hat jedoch ergeben, dass NE-Techniken auch für die erfolgreiche Generierung von DNNs verwendet werden können. Vor allem im Bereich des bestärkenden Lernens haben zahlreiche Studien gezeigt, dass klassische NE-Techniken ein adäquater Ersatz für das Gradienten-basierte Lernen sein können, wenn sie auf die gestiegenen technischen Ressourcen angepasst werden (vgl. [Sta+19]). So haben z. B. *Salimans et al.* gezeigt, dass eine aktuelle Variante einer evolutionären Strategie als Alternative zu herkömmlichen Algorithmen für das bestärkende Lernen (z. B. A3C, DQN) angesehen werden können (vgl. [Sal+17]). Dabei wurden jedoch nur die Gewichte eines ansonsten topologisch fixierten neuronalen Netzes entwickelt, welches ebenfalls für die herkömmlichen Lernverfahren verwendet wurde. Da jedoch bei dieser Variante der evolutionären Strategie auch Gradienten für den Parameter-Raum berechnet wurden, kann dieses Verfahren nicht als komplett gradientenfrei eingestuft werden. Ein simpler genetischer Algorithmus hat jedoch gezeigt, dass auch gradientenfreie Verfahren kompetitiv zu den herkömmlichen Tech-

niken des bestärkenden Lernens sein können (vgl. [Suc+17]). Ebenfalls haben beide Studien [Sal+17; Suc+17] ergeben, dass NE-Techniken schneller als die originalen A3C- oder DQN-Technik ausgeführt werden können, da sie besser parallelisierbar sind (vgl. [Sta+19]). Sie benötigen in manchen Fällen zwar mehr Durchläufe, allerdings können diese wegen der hohen Parallelisierbarkeit von NE-Techniken schneller in Bezug auf die benötigte Zeit ausgeführt werden.
Was an den oben beschriebenen Erfolgen beeindruckend ist, ist die Tatsache, dass sie durch recht „simple“ (klassische) NE-Techniken zustande gekommen sind. Es wurden also keine komplexeren Algorithmen, wie z. B. NEAT verwendet. Viele unskalierte NE-Techniken komplexerer Natur bergen somit das Potenzial, einen weiteren Fortschritt erzielen zu können, wenn sie auf die modernen Rechenressourcen angepasst werden. (vgl. [Sta+19])

4.2 Diversität und Neuheit

Eines der Merkmale, welche die natürliche Evolution auszeichnet, ist ihre Mannigfaltigkeit. Die hoch parallele Exploration von unterschiedlichen Organismen in der Natur führte dazu, dass auf unserem Planeten so viele verschiedene Lebewesen existieren. Letztendlich könnte eben diese Diversität in der Evolution ein kritischer Faktor auf dem Weg zur menschlichen Intelligenz gewesen sein. (vgl. [Sta+19])
Somit kann die Diversität ebenfalls eine kritische Rolle spielen, wenn man Neuroevolution als einen möglichen Weg zur menschenähnlichen künstlichen Intelligenz betrachtet. Aus diesem Grund lag der Fokus auch im Zusammenhang mit Neuroevolution schon länger auf dem Aspekt der Diversität. Denn, wie in Abschnitt 4.1.2 schon erwähnt, sind NE-Algorithmen dazu geeignet, massiv parallel ausgeführt werden zu können (wie auch ihr biologisches Vorbild). (vgl. [Sta+19])

4.2.1 Genetische und Verhaltens-Diversität

Die initiale NE-Forschung zum Thema Diversität bezog sich zunächst darauf, Vielfalt im *genetischen Suchraum* hervorzubringen. Das bedeutet, dass versucht wurde Vielfalt im Parameter-Raum der neuronalen Netze zu erzeugen. Die Kernidee dabei ist, dass, wenn die Suche zu einem lokalen Optimum konvergiert, die Ermutigung

zur Erkundung weg von diesem Optimum ausreichen könnte, um eine neue bessere Lösung für die jeweilige Aufgabe zu finden. Repräsentative Ansätze sind in diesem Zusammenhang das sogenannte *Crowding* [Jon75], bei dem ein neuer Kandidat den genetisch (oder phänotypisch) Ähnlichsten ersetzt, und das *explizite Fitness-Sharing* [GR87], bei dem die einzelnen Kandidaten auf Grundlage ihrer genetischen Distanz in Cluster eingeteilt und dafür bestraft werden, wie viele andere Kandidaten sich im jeweils gleichen Cluster befinden. (vgl. [Sta+19])
Bei einem neuronalen Netz gibt es jedoch unendlich viele Varianten, die Verbindungsgewichte so zu setzen, dass sie letztendlich immer wieder dasselbe Verhalten hervorbringen (vgl. [Sta+19]). Deswegen führt Diversität im Parameter-Suchraum nicht zwingend zu vielfältigem Verhalten (vgl. [LS11]). Durch dieses Problem fokussieren sich neuere Ansätze eher darauf, Diversität im Verhalten zu untersuchen (vgl. [LS11; MD12]), bzw. neuronale Netze gezielt nach bestimmten Kriterien, wie z. B. Neugier [SC18], Entwicklungsfähigkeit [MLC16] oder das Erzeugen von Überraschung [GLY16] zu generieren. Auch hybride Ansätze aus zielgerichteter Suche und Diversität werden verwendet. Der NSGA-II-Algorithmus [Deb+02] belohnt die einzelnen Kandidaten z. B. für Beides: ihre Fitness zu erhöhen (zielgerichtet) und sich von anderen Kandidaten abzuheben (Diversität). Dadurch kann der Algorithmus die verschiedenen Kandidaten in der Population zu neuem Verhalten antreiben, ohne dabei ihre Performance zu vernachlässigen.

4.2.2 Verhaltensneuheit

Einen völlig anderen Ansatz verfolgt die *Novelty-Search* [LS11]. Die Kernidee dabei ist, dass die Neuroevolution nicht länger als Optimierungsalgorithmus angesehen wird. Das bedeutet, dass der Algorithmus nicht länger gegen eine spezifische Lösung konvergieren möchte, sondern das Ziel verfolgt, diverse Lösungen zu generieren. So werden die einzelnen Kandidaten bei der Novelty-Search dafür belohnt, wenn sie sich in ihrem Verhalten von vorher generierten Kandidaten unterscheiden. Dadurch verläuft die Suche nach einer Lösung nicht mehr zielorientiert, wodurch kein Druck mehr auf die einzelnen Kandidaten ausgeübt wird, gegen eine bessere Lösung zu konvergieren. Anders ausgedrückt versucht die Novelty-Search die Population als Ganzes so zu entwickeln, sodass sie letztendlich ein breites Spektrum an Verhalten

aufweisen kann. Wie vorher schon erwähnt, kann der Gradient bezüglich des genetischen Raumes oft uninformativ sein, da viele verschiedene Parameterinstanzen dasselbe uninteressante Verhalten erzeugen können. Der Gradient der Verhaltensneuheit hingegen enthält oft nützliche Informationen bezüglich der Domäne. Diese können dann dazu genutzt werden, um Kandidaten zu produzieren, die letztendlich auch eine bessere Performance bezüglich der jeweiligen Domäne besitzen. Dadurch kann die Novelty-Search die zielgerichtete Suche in manchen Szenarien des bestärkenden Lernens sogar übertreffen (vgl. [LS11; Con+18]). (vgl. [Sta+19])

4.2.3 Qualitätsvielfalt

Aufbauend auf den Ergebnissen der Forschung zu Verhaltensneuheit erforscht ein weiterer Bereich der Neuroevolution Algorithmen bezüglich der „Qualitätsvielfalt“. Dabei soll ein Algorithmus die Vielfalt an möglichen guten Lösungen untersuchen. Dadurch generieren Algorithmen der Qualitätsvielfalt keine einzelne spezifische Lösung für eine Aufgabe, sondern eine Menge an unterschiedlichen Lösungskandidaten. Diese Menge wird auch oft als *Repertoire* bezeichnet. Ist das Qualitätsmerkmal z. B. die Geschwindigkeit einer Kreatur, so würde das Generieren eines schnellen Pumas nicht die Generierung einer schnellen Ameise ausschließen, da beide in ihrer jeweiligen Nische die schnellsten Kandidaten sind. Bei der natürlichen Evolution läuft ein ähnlicher Vorgang ab, wodurch ebenfalls gut angepasste Organismen in unterschiedlichen Nischen der Umwelt hervorgebracht wurden. (vgl. [Sta+19])
Beispiele für solche Algorithmen sind NSLC [LS11] und MAP-Elites [MC15]. Der NSLC-Ansatz ist in der Lage, eine Population für Diversität und lokal beste Lösungen zu optimieren. Dabei nutzt der Ansatz einen modifizierten multi-objektiven EA. Bei MAP-Elites hingegen wird die Population in diskrete Nischen unterteilt, wobei jede Nische einen Champion enthält. Die Unterteilung erfolgt dabei auf Grundlage des Verhaltens der einzelnen Kandidaten, sodass ähnliches Verhalten in die gleiche Nische eingeteilt wird. Die Selektion wird dann nur lokal in der jeweiligen Nische ausgeführt. Die Rekombination und Mutation hingegen erfolgt global. Denn wenn ein Kandidat der Champion in einer Nische wird, kann dies in einem weiteren Schritt resultieren, der dann einen Champion in einer anderen Nische hervorbringt. (vgl. [Sta+19])

Ein Beispiel, um dieses Vorgehen zu verdeutlichen, ist das Erlernen von Fahrradfahren. Meistert man jeweils die Fähigkeiten das eigene Gleichgewicht zu halten und mit Stützrädern zu fahren, kann das letztendlich zum Erlernen des normalen Fahrradfahrens führen. Das Gebiet der Qualitätsvielfalt wächst auch weiterhin, sodass stets neue Forschungsarbeiten hinzukommen (vgl. [BS17; HSM16; HMC16; MM17; NYC16]).

4.3 Indirekte Kodierung

Das Teilgebiet der indirekten Kodierung beschäftigt sich mit der Repräsentation eines neuronalen Netzes durch seinen Genotyp. Dabei ist das Ziel, die Struktur des neuronalen Netzes möglichst kompakt durch Mehrfachverwendung von bestimmten Motiven bzw. Mustern im Genotyp zu kodieren. So existieren z. B. im menschlichen Gehirn etwa 100 Billionen Verbindungen und 100 Milliarden Neuronen (vgl. [Her09]). Der DNA-basierte genetische Code eines Menschen besteht jedoch nur aus rund 30.000 Genen bzw. 3 Milliarden Basenpaaren (vgl. [Ven+01]). Auch hier muss dementsprechend eine Art indirekte Kodierung existieren, welche es erlaubt, eine solch große Struktur wie das menschliche Gehirn (= Phänotyp) durch 30.000 Gene (= Genotyp) abzubilden.
Die Kompression von komplexen Strukturen wie einem neuronalen Netz ist auch in Bezug auf die Skalierbarkeit von NE-Algorithmen wichtig. Beispielsweise arbeitet der NEAT-Algorithmus mit einer direkten Kodierung - kodiert somit explizit jedes Element des neuronalen Netzes im jeweiligen Genotypen. Für herkömmliche neuronale Netze ist diese Repräsentationsform ausreichend (vgl. [Sta+19]). Ist man jedoch darauf angewiesen DNNs mit einem NE-Ansatz zu generieren, so benötigt man eine Repräsentationsform, die ein DNN kompakter darstellen kann, da ansonsten zu viele Parameter durch den NE-Algorithmus optimiert werden müssten.
Die Verwendung von wiederkehrenden Mustern stellt in diesem Zusammenhang einen intuitiven Ansatz für die indirekte Kodierung dar. Dass bestimmte Muster in neuronalen Netzen durchaus mehrfach verwendet werden können, sieht man am Beispiel eines CNN (siehe Abschnitt 2.4.3), bei dem ein Filter-Kernel mehrfach auf die unterschiedlichen Regionen des CNN angewendet wird.

4.3.1 Compositional-pattern-producing-networks

Eine aktuelle Variante von indirekter Kodierung sind die sogenannten Compositional-pattern-producing-networks (CPPNs) [Sta07]. Dabei handelt es sich um ein Modell, welches in seiner Struktur und Funktionsweise äquivalent zu einem neuronalen Netz ist. Die Motivation kommt dabei jedoch aus der Entwicklungsbiologie, in der bestimmte Strukturen eines biologischen Organismus innerhalb eines geometrischen Raumes lokalisiert und ausgebildet werden. So helfen z. B. bei der Entwicklung eines Embryos chemische Gradienten bei der Definition der Achsen von Kopf bis Fuß, von hinten nach vorne und von links nach rechts (vgl. [Mei82]). Dadurch können Strukturen, wie Arme und Beine, in ihre korrekten Positionen platziert werden. Dabei kann jede Struktur selbst auch wieder einen geometrischen Raum aufspannen, in dem dann die jeweiligen Sub-Strukturen platziert werden können. So kann z. B. die Hand eines Menschen als System interpretiert werden, in dem die einzelnen Finger korrekt platziert werden. (vgl. [Sta+19])
Auch die Strukturen an sich, wie z. B. Finger, Arm, Bein usw., können sich (in manchen Fällen mit leichten Abweichungen) wiederholen. An Hand dieser Beispiele ist zu erkennen, dass die Wiederverwendung und Kombination von verschiedenen Mustern durchaus zur kompakten Darstellung von ansonsten komplexen Strukturen verwendet werden können.
CPPNs abstrahieren genau diesen Prozess mithilfe der Kombination simpler mathematischer Funktionen in einem Netzwerk, welches als gerichteter Graph repräsentiert werden kann. In Abbildung 4.3 ist ein Beispiel dafür zu sehen. Als Eingabe der Input-Schicht dienen jeweils die Koordinaten eines Punktes (z. B. x und y für 2-dimensionale Strukturen), welche dann als Basis für die weiteren Berechnungen dienen. Die folgenden Schichten beinhalten dann eine Menge von Knoten mit unterschiedlichen Funktionen, welche miteinander kombiniert werden, um komplexere Strukturen generieren zu können. So kann z. B. die Gauß-Funktion als Äquivalent zum chemischen Gradienten für Symmetrie oder die Sigmoid-Funktion für Asymmetrie gesehen werden. Die Kombination solcher simplen Funktionen kann dann letztendlich in komplexen Strukturen resultieren. Dabei stellt das jeweilige CPPN den Genotypen und die generierte Struktur den Phänotypen dar. Daraus ergibt sich dann, dass ein CPPN z. B. im zweidimensionalen Raum eine Funktion $f(x, y) = w_{xy}$

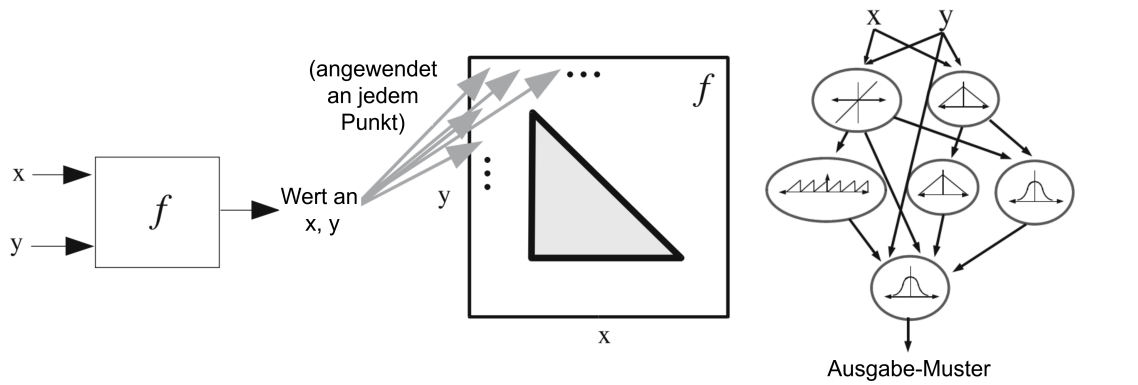


Abbildung 4.3: Generieren einer Struktur mit einem CPPN (in Anlehnung an [SDG09]).

ist, welche die Koordinaten x und y eines Punktes auf einen Wert w_{xy} abbildet, welcher die Intensität des jeweiligen Punktes anzeigt. (vgl. [Sta07])
Dadurch reichen die Anwendungsgebiete der CPPNs von der Generierung von Bildern (vgl. [Sec+11]) bis hin zur Erschaffung dreidimensionalen Objekte (vgl. [CL11]). Dabei wird ein CPPN mithilfe einer Abwandlung des NEAT-Algorithmus generiert (vgl. [Sta07]).
Ein weiteres Anwendungsgebiet im Zusammenhang mit der Neuroevolution ist die Generierung von Verbindungsgewichten eines neuronalen Netzes. Der HyperNEAT-Ansatz nutzt dazu eine leichte Abwandlung von CPPNs, genannt connective-CPPNs. Dieser Algorithmus wird jedoch Bestandteil des folgenden Kapitels 5 sein.

4.3.2 Kartesisch genetische Programmierung

Ein weiterer Ansatz, um den Phänotypen eines neuronalen Netzes durch seinen Genotypen darzustellen, ist die sogenannte *kartesisch genetische Programmierung*, welche eine graph-basierte Form von genetischen Programmen ist. Dabei werden die genetischen Programme in einem gerichteten, oftmals azyklischen Graphen organisiert, welcher zusätzlich um kartesische Koordinaten ergänzt wird. Dabei bekommt jeder Knoten seine Eingabe von vorigen Knoten oder der Programm-Eingabe. Es

können jedoch auch rekurrente Graphen dargestellt werden. Die Knoten werden dann in einem $r \times c$-Grid organisiert, wobei Knoten in derselben Spalte nicht miteinander verbunden sein dürfen (äquivalent zu einem MLP). Die Parameter r und c werden dabei vom Nutzer selbst vorgegeben, wodurch das kartesisch genetische Programm seine Maximalgröße nicht übersteigen kann. (vgl. [TM13])
Neuronale Netze sind eine natürliche Anwendung für kartesisch genetische Programme, da sie sich ebenfalls als gerichtete (azyklische) Graphen darstellen lassen. Wenn man also ein kartesisch genetisches Programm verwendet, um ein neuronales Netz darzustellen, kann das neuronale Netz soweit unverändert in den Genotyp übernommen werden, bis auf den Unterschied, dass für jede Verbindung ein extra Knoten modelliert werden muss, um das jeweilige Gewicht der Verbindung kodieren zu können. Die Gewichte liegen dabei jeweils immer in einem Wertebereich, der vom Nutzer ebenfalls selbst spezifiziert werden muss. (vgl. [TM13])
Üblicherweise verwenden kartesisch genetische Programme als Änderungsoperator die Mutation. Dabei wird vom Nutzer selbst spezifiziert, wie viele Knoten pro Mutation geändert werden. Dieser Wert wird meistens als Prozentzahl der Gesamtanzahl der Knoten im jeweiligen Graphen angegeben. (vgl. [Mah+13])

4.4 Zusammenfassung

Dieses Kapitel hat in das Forschungsgebiet der Neuroevolution eingeführt. Dabei wurde am Anfang aufgezeigt, dass NE-Techniken im Allgemeinen aus zwei verschiedenen Blickwinkeln betrachtet werden können: als Alternative oder als Ergänzung zu anderen Lernalgorithmen.
Das Unterkapitel 4.1 hat zunächst kurz die Entwicklung der Neuroevolution als Forschungsgebiet beschrieben, bevor mit NEAT einer der wichtigsten NE-Algorithmen vorgestellt wurde. Ebenso wurde die Skalierbarkeit von (simplen) NE-Techniken im Vergleich zu anderen Lernalgorithmen beschrieben.
Unterkapitel 4.2 hat sich dem Aspekt der Diversität im Zusammenhang mit der Neuroevolution gewidmet. Dabei wurden einzelne Ansätze wie z. B. Qualitätsvielfalt und Verhaltensneuheit und ihre jeweiligen Algorithmen näher vorgestellt.

Das letzte Unterkapitel 4.3 ist dann auf Varianten für die indirekte Kodierung des Phänotypen im jeweiligen Genotypen eingegangen. Zwei prägnante Ansätze sind dabei die kartesisch genetische Programmierung und CPPNs, wobei Letzterer im weiteren Verlauf dieser Arbeit noch eine spezielle Rolle einnehmen wird.
Das nächste Kapitel wird sich dem Themengebiet des Intra-Life-Learning widmen und die unterschiedlichen Ansätze aufzeigen, welche dafür genutzt werden können. Dabei wird auf schon beschriebene Techniken aus den Kapiteln 3 und 4 zurückgegriffen und dargelegt, wie diese sich gewinnbringend min anderen Ansätzen kombinieren lassen können.

5 Intra-Life-Learning

Herkömmlich werden neuronale Netze mit einem bestimmten Lernalgorithmus für eine spezifische Aufgabe trainiert. In den vorherigen Kapiteln wurden diesbezüglich unterschiedliche Algorithmen wie Backpropagation oder NE-Techniken vorgestellt. In Kapitel 3 wurde schon erläutert, dass für diese Art von Training jeweils ein Datensatz bezüglich der jeweiligen Aufgabe verwendet wird, um daraus Wissen generieren zu können, mithilfe dessen die jeweilige Aufgabe gelöst werden kann. Diese Art von Lernen hat in vielen Gebieten wie z. B. Bildverarbeitung [He+16], bestärkendes Lernen [Sil+16] und Sprachverarbeitung [Dev+19] zu Durchbrüchen geführt.
Viele Anwendungen erfordern es jedoch, bereits bestehendes Wissen anzupassen oder neues Wissen hinzulernen zu können. Das Intra-Life-Learning (ILL) beschreibt ein Lernparadigma für neuronale Netze, das diesen Anforderungen gerecht werden kann. Dazu wird im nächsten Unterkapitel 5.1 die Kernidee des Intra-Life-Learning beschrieben und wie es realisiert werden kann. Die weiteren Unterkapitel 5.2, 5.3 und 5.4 erläutern dann jeweils konkrete Ansätze und evaluieren diese in Bezug auf das Intra-Life-Learning. Abschließend fasst Unterkapitel 5.5 die Erkenntnisse dieses Kapitels zusammen.

5.1 Kernidee und Begriffsklärung

Die Kernidee des Intra-Life-Learning ist es, mittels bestimmter Lernalgorithmen neuronale Netze in die Lage zu versetzen, bereits vorhandenes Wissen auf neue Umstände adaptieren zu können oder neues Wissen hinzuzulernen.
Eine intuitive Herangehensweise, um das Intra-Life-Learning zu realisieren, wäre, herkömmliche Lernalgorithmen wie Backpropagation oder verschiedene NE-Techniken zu verwenden, um das jeweilige Wissen für die verschiedenen Aufgabenstellungen zu

erlernen. Allerdings existieren in diesem Zusammenhang klare Limitationen (vgl. [Mar18]).
Beim Backpropagation-Algorithmus existiert z. B. das Problem des *katastrophalen Vergessens* [Fre99]. Dabei wird das vorhandene Wissen von Neuem überschrieben, wenn man das neuronale Netz auf eine neue Aufgabe trainiert. Das hat dann zufolge, dass das neue Wissen zwar gut auf die neue oder veränderte Aufgabe anwendbar ist, jedoch die alte Ausgangsaufgabe nicht mehr effizient gelöst werden kann. Dieses Phänomen verhindert, dass neuronale Netze neues Wissen hinzulernen können, ohne dabei das alte Wissen zu vergessen bzw. zu überschreiben.
Viele Lernalgorithmen [Hos+20] und auch NE-Techniken [Sta+19] erfordern auch eine große Datenmenge, um neuronale Netze zufriedenstellend trainieren zu können. Eine schnelle, effiziente Anpassung des neuronalen Netzes auf neue Zielstellungen oder sich verändernde Umstände in der jeweiligen Domäne ist somit nicht möglich. Das exkludiert sie für Anwendungen, in denen nur wenige Daten bzw. Trainingsbeispiele zur Verfügung stehen.
Ein dritter Punkt ist das statische Design der Algorithmen. So können sich Lernalgorithmen wie Backpropagation oder NE-Techniken nicht selber optimieren. Bereits vorhandenes Wissen kann also nicht verwendet werden, um die Lernstrategie zu optimieren. Hinzu kommt, dass der jeweilige Programmierer viel zusätzliches Wissen bezüglich einer Domäne benötigt, um die Hyperparameter des neuronalen Netzes und des jeweiligen Algorithmus bestimmen zu können.
Deswegen ist die Idee, das sogenannte Meta-Lernen für die Realisierung des Intra-Life-Learning zu verwenden. Dabei werden die herkömmlichen Lernalgorithmen wie Backpropagation oder NE-Techniken nicht mehr dafür verwendet, dass jeweilige neuronale Netz zu trainieren, sondern um sogenanntes *Meta-Wissen* zu generieren, welches dann eingesetzt werden kann, um das Lernverhalten des neuronalen Netzes positiv zu beeinflussen. Um dem Leser im weiteren Verlauf die Unterscheidung zwischen den verschiedenen Begrifflichkeiten des Meta-Lernens zu erleichtern, werden an dieser Stelle folgende Begriffe eingeführt:

- **Basis-Modell:** Dabei handelt es sich um das Modell bzw. neuronale Netz, welches für eine gegebene Aufgabe optimiert werden soll. Synonym kann im

weiteren Verlauf dieser Arbeit auch der Begriff Basis-NN verwendet werden. In diesem Zusammenhang sei auch noch erwähnt, dass das im Basis-Modell gespeicherte Wissen im weiteren Verlauf dieser Arbeit als Basis-Wissen bezeichnet wird.

- **Meta-Modell:** In diesem Modell wird das eben schon erwähnte Meta-Wissen gespeichert. Das Meta-Modell kann (anders, als das Basis-Modell) unterschiedliche Ausprägungen annehmen, welche im weiteren Verlauf dieses Kapitels näher erläutert werden.

- **Meta-Optimierer**: Damit wird der Algorithmus bezeichnet, welcher genutzt wird, um das Meta-Modell zu trainieren. Dabei kann es sich entweder um einen evolutionären Algorithmus, wie z. B. NE, GA oder ES, handeln oder um ein Lernverfahren wie den Backpropagation-Algorithmus.

- **Basis-Optimierer:** Hierbei handelt es sich um den Algorithmus, mit dem das Basis-Modell trainiert wird. Der Basis-Optimierer ist dabei immer abhängig von der Art des Meta-Modells bzw. des Meta-Wissens. Dies kann entweder herkömmliche Lernalgorithmen unterstützen oder komplett obsolet machen.

Das Meta-Lernen alleine reicht allerdings nicht aus, um den schon erwähnten Problemen (katastrophales Vergessen, Anpassen an sich verändernde Umstände, Effizienz usw.) begegnen zu können. Daher setzt sich das Intra-Life-Learning aus mehreren unterschiedlichen Lernszenarien zusammen, die sich gegenseitig bedingen können.
Im folgenden Abschnitt 5.1.1 wird zunächst der Prozess des Meta-Lernens formalisiert und genauer erläutert, damit beim Leser ein konsistentes und tiefgreifendes Verständnis für dieses Thema aufgebaut werden kann. Der anschließende Abschnitt 5.1.2 wird dann die eben schon erwähnten Lernszenarien, welche im Kontext des Intra-Life-Learnings relevant sind, erklären und voneinander abgrenzen. Aufbauend darauf werden dann in Abschnitt 5.1.3 der Aufbau der in diesem Kapitel durchgeführten Evaluation und die verwendeten Kriterien erläutert.

5.1.1 Formalisierung des Meta-Lernens

Das Meta-Lernen kann als „Lernen, wie man lernt“ [Sch87] verstanden werden. Dabei wird ein neuronales Netz nicht auf einer einzelnen Aufgabe (wie z. B. beim herkömmlichen maschinellen Lernen) trainiert, sondern über eine Menge von Aufgaben der gleichen oder unterschiedlicher Domänen. In Abbildung 5.1 ist dieser Unterschied zwischen dem Meta- und herkömmlichen Lernen dargestellt.

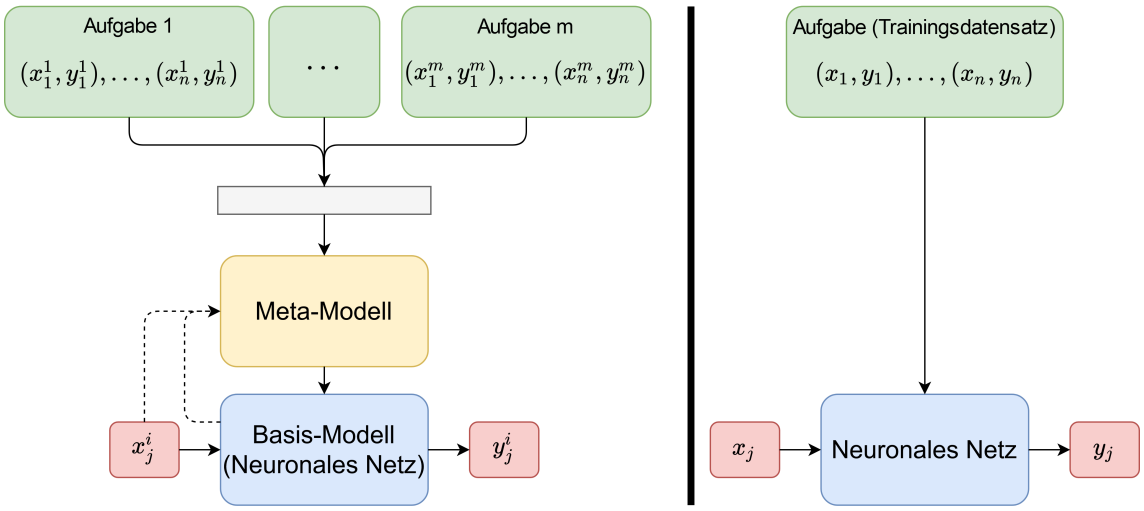


Abbildung 5.1: Das Meta-Lernen (links) im Vergleich mit dem klassischen Vorgehen (rechts) des maschinellen Lernens.

Die grünen Elemente stellen dabei den Trainingsdatensatz dar, der genutzt wird, um das jeweilige neuronale Netz zu trainieren. Die roten Elemente $x_j^{(i)}$ sind dann jeweils reale Daten, für die das trainierte neuronale Netz den jeweiligen Output $y_j^{(i)}$ produzieren soll. Beim Meta-Lernen wird, im Gegensatz zum klassischen Vorgehen, ein Meta-Modell parallel über mehrere Aufgaben trainiert, um das Meta-Wissen generieren zu können. Dieses Meta-Wissen kann dann entweder genutzt werden, um das Basis-Modell (= neuronales Netz) direkt zu trainieren oder sein (Lern-)Verhalten zu beeinflussen. Die gestrichelte Linie symbolisiert einen optionalen Datenfluss. Bei manchen Algorithmen des Meta-Lernens bekommt das Meta-Modell ebenfalls reale Daten nach dem eigentlichen Trainingsprozess oder auch Parameter des Basis-Modells als zusätzlichen Input.

Mathematisch gesehen handelt es sich beim klassischen maschinellen Lernen um ein Optimierungsproblem [Hos+20]:

$$\theta^* = \arg\min_{\theta} L(D, \theta, \omega) \tag{5.1}$$

Dabei definiert θ das untrainierte Modell, welches nach dem Training in θ^* resultiert. Die Fehlerfunktion und der jeweilige Datensatz werden durch L und D notiert. Der Parameter ω repräsentiert einen bestimmten Faktor des Trainingsprozesses, wie z. B. den gewählten Lernalgorithmus (nachfolgend auch als Optimierer bezeichnet). Die Performance bzw. Generalisierung von θ^* wird dann mithilfe eines Test-Datensatzes gemessen, welcher bisher nicht gesehene Beispiele für die jeweilige Aufgabe enthält. Es wird also angenommen, dass für jede Aufgabe D das jeweilige neuronale Netz θ von Grund auf neu trainiert wird, wobei ω vorher vom Programmierer selbst spezifiziert werden muss.
Das Meta-Lernen hingegen bezieht sich nicht darauf θ zu trainieren, sondern ω zu erlernen, welches dann genutzt werden kann, um θ auf eine oder mehrere Aufgaben zu trainieren. Bei θ handelt es sich somit um das Basis-Modell (in unserem Fall ein neuronales Netz) und bei ω um das Meta-Modell, welches genutzt wird, um das Basis-Modell trainieren zu können.
Der Parameter w spezifiziert also „wie man lernt“ und wird beim Meta-Lernen über eine Verteilung von Aufgaben $p(T)$ trainiert. Eine einzelne Aufgabe $T = \{D, L\}$ wird dabei als ein Datensatz D und eine Fehlerfunktion L definiert. Dadurch kann das Meta-Lernen dann folgendermaßen spezifiziert werden [Hos+20]:

$$\min_{w} \mathop{\mathbb{E}}_{T \sim p(T)} L(D; \omega) \tag{5.2}$$

Es soll also ω optimiert werden, indem der Erwartungswert $\mathbb{E}$ bezüglich der Fehlerfunktion L über der Verteilung $p(T)$ minimiert wird.
Um dieses Problem in der Praxis lösen zu können, nimmt man an, dass man Zugriff auf eine Menge von Trainings-Aufgaben hat, die aus $p(T)$ ausgewählt werden. Damit erfolgt dann das *Meta-Training* von ω. Formal wird diese Menge von M Quell-Aufgaben durch $\mathscr{D}_{source} = \{(D^{train}_{source}, D^{val}_{source})^{(i)}\}_{i=1}^{M}$ definiert, wobei jede Aufgabe jeweils einen Trainings- und Validierungsdatensatz besitzt. (vgl. [Hos+20])

Ein Meta-Trainings-Schritt bezüglich ω kann dann beschrieben werden durch [Hos+20]:

$$\omega^* = \arg\max_{\omega} \log p(\omega|\mathscr{D}_{source}) \tag{5.3}$$

Dabei wird ω bezüglich der Log-Likelihood-Funktion über der Verteilung $p(\omega|\mathscr{D}_{source})$ optimiert.
Für das Meta-Testen wird dann ein Datensatz $\mathscr{D}_{target} = \{(D_{target}^{train}, D_{target}^{test})^{(i)}\}_{i=1}^{Q}$ verwendet, welcher Q Ziel-Aufgaben enthält. Jede Ziel-Aufgabe besteht dabei aus einem Trainings- und einem Test-Datensatz. (vgl. [Hos+20])
Für das Meta-Testen wird dann das gelernte Meta-Modell ω^* verwendet, um das Basis-Modell auf den bisher nicht gesehenen Ziel-Aufgaben zu trainieren [Hos+20]:

$$\theta^{*(i)} = \arg\max_{\theta} \log p(\theta|\omega^*, D_{target}^{train\ (i)}) \tag{5.4}$$

Das Lernen einer bestimmten Aufgabe i profitiert in diesem Szenario vom Meta-Modell ω^*. Die Genauigkeit des Meta-Modells bezüglich dieser Aufgabe kann dann mithilfe der Performance von $\theta^{*(i)}$ auf dem Test-Datensatz der jeweiligen Ziel-Aufgabe $D_{target}^{test\ (i)}$ gemessen werden. (vgl. [Hos+20])
Dieses Vorgehen des Meta-Lernens lässt sich auch verallgemeinern, indem man den Meta-Trainings-Schritt aus Gleichung 5.3 als ein *Bilevel*-Optimierungsproblem modelliert. Dabei handelt es sich um ein hierarchisches Optimierungsproblem, bei dem die Meta-Optimierung die Basis-Optimierung als Bedingung enthält. (vgl. [Hos+20])
Aufbauend darauf kann das Meta-Training wie folgt formalisiert werden [Hos+20]:

$$\omega^* = \arg\min_{\omega} \sum_{i=1}^{M} L^{meta}(\theta^{*(i)}(\omega), \omega, D_{source}^{val\ (i)}) \tag{5.5}$$

$$\text{wobei } \theta^{*(i)}(\omega) = \arg\min_{\theta}\ L^{task}(\theta, \omega, D_{source}^{train\ (i)}) \tag{5.6}$$

Dabei stellen L^{meta} und L^{task} jeweils das äußere und innere Optimierungskriterium dar. Dafür können z. B. die schon in Unterkapitel 3.1 vorgestellten Fehlerfunktionen verwendet werden. Für den Basis-Optimierer wird in diesem Fall der Trainings-Datensatz bezüglich $\mathscr{D}_{source}$ verwendet und für das Training des Meta-Optimierers

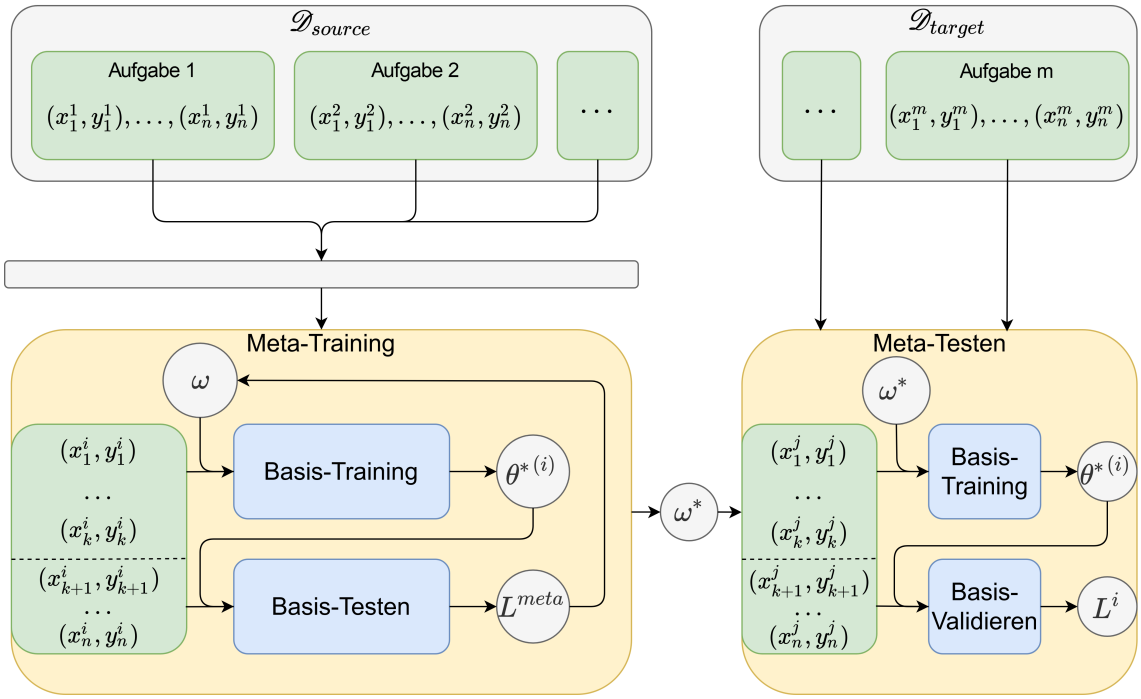


Abbildung 5.2: Der Prozess des Meta-Lernens.

der dazugehörige Validierungsdatensatz. In Abbildung 5.2 ist der komplette Prozess des Meta-Lernens zu sehen.

An dieser Stelle wird auch das wesentliche Merkmal des Bilevel-Paradigmas sichtbar. Dabei handelt es sich um die sogenannte *Leader-Follower-Asymmetrie* zwischen der äußeren und inneren Optimierung: die Basis-Optimierung aus Gleichung 5.6 ist an die von der Meta-Optimierung definierten Lernstrategie ω gebunden, kann aber ω während des Trainings von θ nicht verändern. (vgl. [Hos+20])

Das Meta-Modell ω wird also über alle Aufgaben (auch Episoden genannt) in $\mathscr{D}_{source}$ hinweg, mithilfe des Meta-Optimierers, verbessert. Dadurch, dass das Meta-Modell über alle Aufgaben in $\mathscr{D}_{source}$ trainiert wird, enthält es generalisiertes Wissen über diese Aufgaben, welches dann dazu verwendet werden kann, um das Basis-Modell θ für die einzelnen Aufgaben zu trainieren. Die Evaluation des Meta-Modells erfolgt dann ebenfalls über eine weitere Menge mit mehreren Aufgaben, wobei das jeweilige Basis-Modell $\theta^{(i)}$ mithilfe von ω^* auf einer bestimmten Aufgabe i trainiert und an-

schließend auf selbiger getestet wird. Die Performance des gelernten Meta-Modells ω^* wird also für jede Aufgabe in $\mathscr{D}_{target}$ einzeln berechnet, statt über alle Aufgaben hinweg.
Bei der Meta-Optimierung wird dadurch im sogenannten *Meta-Raum* nach den idealen Parametern für ω gesucht, während bei der Basis-Optimierung im *Basis-Raum* ebenfalls nach den passenden Parametern für θ gesucht wird. Das Ziel des Meta-Lernens besteht somit darin, Meta-Wissen zu generieren, welches über alle gegebenen Aufgaben *generalisiert*, damit es dabei helfen kann, das Basis-Modell auf ähnlichen, ungesehenen Aufgaben zu optimieren.

5.1.2 Lernszenarien

Wie schon erwähnt, lässt sich im Rahmen des Intra-Life-Learning zwischen unterschiedlichen Lernszenarien unterscheiden. Zum einen sollen neuronale Netze auf neue Aufgaben trainiert werden können, ohne dabei das vorhandene Wissen katastrophal zu vergessen. Das impliziert, dass es möglich sein muss, ein neuronales Netz auf eine Menge von Aufgaben in einer sequenziellen Abfolge zu trainieren. Damit steht es dem Meta-Lernen gegenüber, was herkömmlicherweise alle Aufgaben auf einmal und nicht hintereinander als Input bekommt.
Zum anderen soll das neuronale Netz sich auch an verändernde Umgebungen bezüglich einer Aufgabe anpassen können. Dabei ist es natürlich ebenso wichtig, dass das alte Wissen einer Aufgabe nicht komplett überschrieben wird.
Die nächsten beiden Absätze werden deswegen in die Lernszenarien des *kontinuierlichen* und *adaptiven* Lernens einführen und sie erläutern. Dabei bringt jedes Szenario seine eigenen Anforderungen mit, welche in dem jeweiligen Absatz ebenfalls erklärt werden.

Kontinuierliches Lernen

Das kontinuierliche Lernen (KL), auch als *lebenslanges Lernen*, *sequenzielles Lernen* oder *inkrementelles Lernen* bezeichnet, untersucht das Problem des Lernens aus einem unendlichen Strom von Daten, die aus sich ändernden Eingabedomänen stammen und mit verschiedenen Aufgaben verbunden sind, mit dem Ziel, das erworbene Wissen bei der Aufgabenlösung und beim zukünftigen Lernen nutzen zu

können (vgl. [Alj19, S. 3], [CL18a]). Dabei bringt diese Art des Lernens unterschiedliche Anforderungen mit:

1. **Online-Lernen:** Lernen findet zu jedem Zeitpunkt statt, ohne feste Aufgaben oder Datensätze und ohne klare Grenzen zwischen den Aufgaben.

2. **Wissenstransfer (vorwärts/rückwärts):** Das Modell sollte in der Lage sein, von zuvor gesehenen Aufgaben auf Neue zu übertragen (Transfer vorwärts) sowie neue Aufgaben zur Verbesserung der Leistung bei älteren Aufgaben verwenden zu können (Transfer rückwärts).

3. **Widerstand gegen katastrophales Vergessen:** Das Erlernen neuer Aufgaben zerstört nicht die Leistung auf zuvor gesehenen Aufgaben.

4. **Begrenzte Systemgröße:** Die Kapazität des Modells sollte festgelegt werden, um das System zu zwingen, seine Kapazität intelligent zu nutzen.

5. **Kein direkter Zugriff auf frühere Erfahrungen:** Während das Modell sich an eine begrenzte Menge an Erfahrungen erinnern kann, sollte ein Algorithmus für kontinuierliches Lernen keinen direkten Zugriff auf vergangene Aufgaben haben bzw. in der Lage sein, die Lern-Umgebung zurückzuspulen.

Bei dieser Liste wird kein Anspruch auf Vollständigkeit erhoben. Ferner wird sich dieses Kapitel und die hier vorgestellte Evaluation nur auf Anforderung 2 und 3 beschränken. Die erste Anforderung wird vernachlässigt, da die in Abschnitt 5.1.1 vorgestellte Formalisierung des Meta-Lernens die Abgrenzbarkeit zwischen den unterschiedlichen Aufgaben voraussetzt. Die vierte Anforderung hingegen steht in Widerspruch zur Dritten. Deswegen wird diese ebenso vernachlässigt. Die fünfte Anforderung steht ebenfalls im Gegensatz zu dem, was in Abschnitt 5.1.1 definiert wurde. Denn für das (herkömmliche) Meta-Lernen wird angenommen, dass der jeweilige Algorithmus während des Lernprozess Zugriff auf alle Aufgaben hat.
Ferner fällt bei der Anforderung 1 auf, dass beim kontinuierlichen Lernen zwischen zwei verschiedenen Szenarien unterschieden werden kann. Zum einen kann die Variante betrachtet werden, dass der jeweilige Lernalgorithmus als Input einen unendlichen Strom von Daten bekommt, ohne Zusatzinformationen, ob es sich dabei um

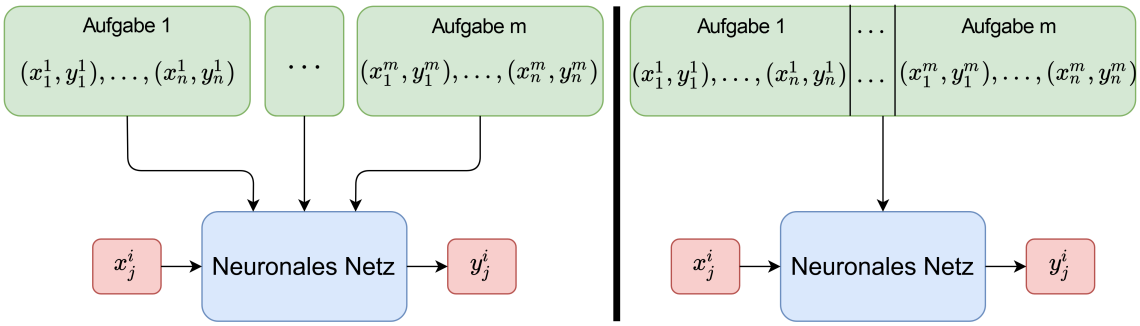


Abbildung 5.3: Die Batch- (links) und Online-Variante (rechts) des kontinuierlichen Lernens im Vergleich.

Daten bezüglich der gleichen, einer alten oder neuen Aufgabe handelt. Zum anderen kann der Input auch in unterschiedliche Aufgaben unterteilt werden, sodass der Algorithmus weiß, welcher Datenpunkt zu welcher Aufgabe gehört. In Abbildung 5.3 sind beide Varianten zu sehen.
Im Rahmen des Intra-Life-Learning wird sich zunächst auf die Variante des Batch-Learning (Datenstrom ist in verschiedene Aufgaben zerteilt) beschränkt. Die Online-Variante würde nochmal ganz eigene Probleme mit sich bringen, wie z. B. dass der jeweilige Algorithmus zusätzlich noch lernen muss, zwischen unterschiedlichen Aufgaben zu unterscheiden. Das würde voraussetzen, dass das jeweilige Modell eine interne Metrik erlernt, mit derer die Ähnlichkeit zwischen den einzelnen Inputs berechnet werden kann. Sollte es jedoch Algorithmen geben, welche diese Anforderungen schon leisten können, wird offen gelassen, ob die Definition des Intra-Life-Learning an dieser Stelle nicht erweitert wird.

Adaptives Lernen

Dieses Lernszenario zeichnet sich durch die Anpassung von vorhandenem Wissen an Veränderungen in einer Domäne oder Aufgabe aus. Ändert sich also z. B. die einer spezifischen Aufgabe zugrundeliegende Struktur, so soll das in einem neuronalen Netz gespeicherte Wissen bezüglich dieser Aufgabe an diese Veränderung angepasst werden können, ohne den kompletten Lernprozess von Grund auf neu durchlaufen zu müssen. Das inkludiert, dass das vorhandene Wissen bei dem Trainingsprozess

mitgenutzt werden soll. Im Gegensatz zum kontinuierlichen Lernen, soll also kein neues Wissen generiert werden. Stattdessen wird vorhandenes Wissen bezüglich einer schon erlernten Aufgabe angepasst. Dadurch kann das neuronale Netz angemessen auf die Veränderungen in jener Aufgabe reagieren.
Dabei sind allerdings, wie auch beim kontinuierlichen Lernen, bestimmte Anforderungen zu beachten:

1. **Widerstand gegen Interferenz:** Wenn das neuronale Netz sein Wissen bezüglich einer bestimmten Aufgabe anpasst, dann soll das Wissen hinsichtlich einer anderen Aufgabe davon nicht betroffen sein.

2. **Schnelligkeit:** Das schon vorhandene Wissen soll möglichst schnell angepasst werden können. Es sollen also nicht so viele Inputs wie für das herkömmliche Lernen benötigt werden, um das neuronale Netz auf sich verändernde Aufgaben anpassen zu können.

Diese zusätzlichen Anforderungen sind vor allem für das 6. Kapitel relevant. Für den weiteren Verlauf dieses Kapitels werden diese beiden Anforderungen jedoch ignoriert. Stattdessen wird nur untersucht, ob die vorgestellten Algorithmen für das adaptive Lernen im Allgemeinen geeignet sind. Dadurch soll die im nächsten Abschnitt 5.1.3 vorgestellte Evaluation übersichtlicher gehalten werden.

5.1.3 Aufbau der Evaluation und Kriterien

Die Evaluation besteht aus mehreren Vergleichen, die jeweils auf unterschiedlichen Ebenen durchgeführt werden. So werden in diesem Kapitel drei verschiedene *Ansätze* im Zusammenhang mit dem Intra-Life-Learning vorgestellt: Meta-Lernen (Unterkapitel 5.2), synaptische Plastizität (Unterkapitel 5.3)und Neuromodulation (Unterkapitel 5.4). Dabei bringt jeder Ansatz wiederum seine eigene Menge von *Techniken* mit, welche in den jeweiligen Abschnitten detailliert vorgestellt werden. Jede Technik kann dabei durch mehrere (verschiedene) *Algorithmen* umgesetzt werden.
So ergibt sich für jede Technik ein Vergleich zwischen seinen Algorithmen, der innerhalb des jeweiligen Abschnittes zu finden sein wird. Um die verschiedenen Techniken eines Ansatzes vergleichen zu können, existiert ein „Fazit“-Abschnitt am Ende eines jeden Unterkapitels, welcher diesen Vergleich der Ansätze beinhaltet. In der

Zusammenfassung dieses Kapitels werden dann anschließend die Ansätze miteinander verglichen, was dementsprechend der obersten Ebene der kompletten Evaluation entspricht.
Alle Vergleiche unterliegen denselben objektiven Kriterien. Diese ergeben sich zum Teil aus den vorher schon erläuterten Lernszenarien (Meta-Lernen, kontinuierliches Lernen, adaptives Lernen):

- **Adaptierbarkeit**: Dieses Kriterium umfasst die Anpassbarkeit des jeweiligen Modells auf sich verändernde Umstände innerhalb einer spezifischen Aufgabe. Damit steht dieses Kriterium stellvertretend für die Fähigkeit des *adaptiven Lernens*.

- **Stabilität**: Dieses Kriterium beschreibt die Fähigkeit der Erweiterbarkeit des Wissens um Wissen bezüglich einer neuen Aufgabe, ohne dabei schon vorhandenes/älteres Wissen zu überschreiben. Damit steht dieses Kriterium stellvertretend für den *Widerstand gegen katastrophales Vergessen* im Kontext des *kontinuierlichen Lernens*.

- **Effizienz**: Dieses Kriterium beschreibt, ob das bisher gesammelte Meta-Wissen dazu eingesetzt werden kann, um eine neue Aufgabe effektiver zu lösen. Damit ist es der Anforderung des *Wissenstransfers* aus dem *kontinuierlichen Lernen* zuzuordnen.

- **Generalisierbarkeit**: Dieses Kriterium spiegelt die Fähigkeit eines Algorithmus wider, Meta-Wissen generieren und auf unterschiedliche Aufgaben anwenden zu können. Es repräsentiert somit die Fähigkeit des *Meta-Lernens*.

- **Parallelisierbarkeit**: Ist es möglich, den genutzten (Meta-)Optimierer zu parallelisieren? Dieses Kriterium ist davon abhängig, welche Art von Algorithmus für den Optimierer verwendet wird.

Aus den hier eingeführten Kriterien ergibt sich dann die Tabelle 5.1 als Basis für einen objektiven Vergleich.

Da die Idee ist, dass Intra-Life-Learning mittels Meta-Lernen umzusetzen, ist nur schlüssig, dass bei der Evaluation auch zusätzlich zwischen dem Meta- und Basis-Raum unterschieden werden sollte, da beide jeweils einen separaten Parameter-Suchraum aufspannen (siehe Abschnitt 5.1.1). Daraus ergibt sich folgende Tabellenstruktur für die Evaluation:

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[BSP20]	✓/O/X	✓/O/X	✓/O/X	✓/O/X	✓/O/X	✓/O/X	✓/O/X	✓/O/X	✓/O/X

Tabelle 5.1: Beispieltabelle für die Vergleiche.

Die Zeichen ✓, O und X bedeuten jeweils „erfüllt“, „möglich/naheliegend“ und „nicht erfüllt“ und gelten für alle Spalten der Tabelle. Die Parallelisierbarkeit bezieht sich nur auf den verwendeten Optimierer, weshalb für dieses Kriterium nicht weiter zwischen Meta- und Basis-Raum unterschieden wird.
Im folgenden Unterkapitel 5.2 wird zunächst eine kurze Einführung in das (klassische) Meta-Lernen im Allgemeinen gegeben, bevor die unterschiedlichen Techniken näher vorgestellt und deren Algorithmen im einzelnen miteinander verglichen werden.

5.2 Klassisches Meta-Lernen

Erste Forschung im Bereich des Meta-Lernens erschien 1987 in Form von zwei voneinander unabhängigen Arbeit von *Jürgen Schmidhuber* [Sch87] und *Geoffrey Hinton* [HN87]. Schmidhuber [Sch87] hat sich dabei mit der Idee des *selbst-referenzierenden* Lernens auseinandergesetzt. Dabei bekommt ein neuronales Netz seine eigenen Verbindungsgewichte als zusätzlichen Input und kann diese dann selber manipulieren bzw. verbessern. Das setzte die theoretische Grundlage für neuere Arbeiten, welche diese Idee aufgreifen. Ebenso argumentiert Schmidhuber in seiner Arbeit, dass diese Art von neuronalen Netzen mittels Neuroevolution gelernt werden könnte. Dieser Gedanke ist naheliegend, da auch die natürliche Evolution als Meta-Lernen interpretiert werden kann (vgl. [Sta+19]). Sie kann als eine Art externe Optimierung

verstanden werden, die Organismen mit eigenen Lernmechanismen, also einer internen Optimierung, hervorgebracht hat.
Hinton und Nowlan [HN87] hingegen verfolgen einen anderen Ansatz, bei dem für jede Verbindung zwischen Neuronen zwei statt einem Gewicht vorgesehen sind. Dabei handelt es sich beim ersten Gewicht um ein „langsames“ Gewicht, welches standardmäßig über den Optimierer trainiert wird. Das zweite Gewicht bzw. das „schnelle“ Gewicht generiert Wissen während der Inferenz[1]. Dabei soll das schnelle Gewicht das Wissen des langsamen Gewichtes wiederherstellen, welches wegen der Optimierer-Updates verloren gegangen ist.
Beide Ansätze legten unter anderem die Grundlage für die heutige Forschung im Bereich des Meta-Lernens (vgl. [Hos+20]). Und auch beide Ansätze zeigen, dass es unterschiedliche Strategien gibt, nach welcher Logik das Meta-Lernen erfolgen sollte. So merkt Schmidhuber an, dass z. B. NE-Techniken für das Meta-Lernen verwendet werden könnten. Hinton und Nowlan hingegen verwenden weiterhin den Gradientenabstieg als Grundlage für das Lernen.

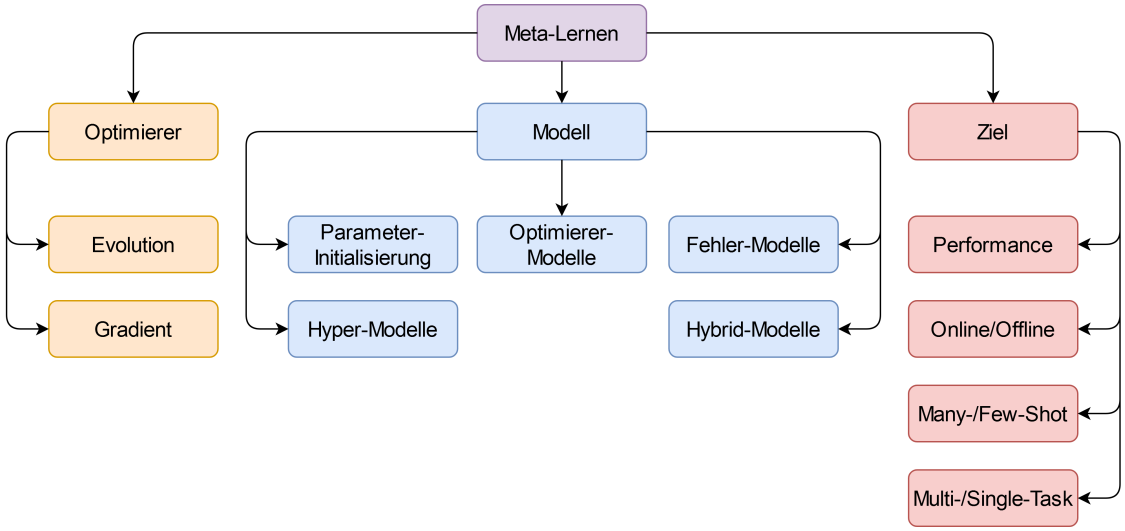


Abbildung 5.4: Übersicht über die Unterscheidungsmerkmale im Gebiet des Meta-Lernens.

[1] In diesem Zusammenhang bezeichnet dieser Begriff die Phase der Anwendung eines neuronalen Netzes in einer bestimmten Domäne.

Dieses Unterkapitel stellt einen Überblick über die neuesten Ansätze für das Meta-Lernen dar. Dabei können die Arbeiten zu den einzelnen Algorithmen anhand ihrer Zielstellung, dem genutzten Optimierer und dem Meta-Modell unterschieden werden. In Abbildung 5.4 ist ein Überblick über die Ausprägungen der einzelnen Unterscheidungsmerkmale zu sehen.
In diesem Kapitel werden die einzelnen Algorithmen anhand der gewählten Modelle in verschiedene Techniken unterteilt und im jeweiligen Abschnitt vorgestellt. Dabei wird zunächst der Kerngedanken der jeweiligen Technik erläutert, bevor die einzelnen Algorithmen näher beschrieben werden. Zu erwähnen ist noch, dass es auch weitere Modelle, Optimierer und Zielstellungen im Kontext des Meta-Lernens gibt. Da sich diese Arbeit allerdings auf die Neuroevolution und den Gradientenabstieg als Optimierungsverfahren beschränkt und der Rahmen dieses Unterkapitels nicht gesprengt werden soll, wird nur eine bestimmte Auswahl aller Algorithmen für das Meta-Lernen vorgestellt. Für einen kompletten Überblick über die Thematik wird der interessierte Leser auf die Arbeit [Hos+20] verwiesen.

5.2.1 Parameter-Initialisierung

Als erste Technik ist die *Parameter-Initialisierung* zu nennen, welche die initialen Parameter eines neuronalen Netzes zu erlernen versucht. Dabei wird das neuronale Netz weiterhin mit dem Gradientenabstieg trainiert. Der Grundgedanke hinter diesem Vorgehen ist, dass eine gute Parameter-Initialisierung nur ein paar Gradientenschritte von einem lokalen Optimum entfernt ist. Diese Technik wird vor allem für das sogenannte *Few-Shot-Learning* eingesetzt, bei dem es darum geht, mit so wenig Daten wie möglich die Lösung für eine neue Aufgabe zu finden. Eine der Kern-Herausforderungen bei der Parameter-Initialisierung ist jedoch die Anzahl der Parameter, welche durch den Meta-Optimierer gefunden werden müssen. Dabei muss der jeweilige Meta-Optimierer meistens genauso viele Parameter optimieren wie der Basis-Optimierer. Das wiederum hat über die Jahre zu der Erforschung von Algorithmen geführt, welche das Meta-Lernen auf eine bestimmte Teilmenge der Parameter eines neuronalen Netzes reduzieren. Beispiele dafür sind Teilräumen [YS18; And+18], bestimmte Neuronen-Schichten [Qia+18; And+18; AA19] oder Skalierung und Verschiebung [Sun+19]. Diese Ideen könnten auch auf die Parameter-

Initialisierung angewendet werden, um auch hier die Anzahl der zu berücksichtigenden Parameter zu reduzieren.
Ein bekanntes Beispiel für die Technik der Parameter-Initialisierung ist der MAML-Algorithmus [FAL17] mit all seinen Varianten [FL18; Fin+19; Raj+19; Son+20]. Dabei werden die initialen Parameter des Basis-Modells als Konditionen der Basis-Optimierung angenommen. Besonders hervorzuheben ist dabei ES-MAML [Son+20]. Bei dieser Variante von MAML wird als Meta-Optimierer eine evolutionäre Strategie verwendet. Das Basis-Modell wird dabei weiterhin mit dem Gradientenabstieg als Basis-Optimierer trainiert. Das ist insofern besonders, da die restlichen Varianten von MAML für die Meta-Optimierung ebenfalls den Gradientenabstieg verwenden.

Fazit

Das bei Algorithmen dieser Technik nur die initialen Parameter eines (Basis-)Modells gelernt werden, macht sie vor allem für das Few-Shot-Lernen attraktiv, bei dem versucht wird, auf Basis weniger Trainingsbeispiele für die verschiedenen Aufgaben einer Domäne die bestmögliche Lern-Performance zu erzielen. Dabei wird Meta-Wissen über Gemeinsamkeiten der unterschiedlichen Aufgabenstrukturen generiert, welches dann dementsprechend möglichst über alle Aufgaben einer Domäne generalisieren soll. Damit entspricht es dem Kriterium der Generalisierbarkeit. Fast alle Algorithmen erfüllen dieses Kriterium, da sie auf unterschiedlichen Datensätzen für das Few-Shot-Lernen wie z. B. Omniglot [Lak+11] oder MiniImagenet [Vin+16] trainiert und anschließend auch getestet wurden. Eine Ausnahme bildet dabei der FTML-Algorithmus [Fin+19], welcher als Zielstellung die Steigerung der Effizienz des Basis-Optimierers mit wachsendem Meta-Wissen hat und eine Weiterentwicklung des MAML-Algorithmus für das kontinuierliche Lernen darstellt. Dadurch kann dieser Algorithmus nicht das Kriterium der Generalisierbarkeit erfüllen, da er nicht für das Multi-Task-Lernen (MTL) konzipiert ist. Dafür wird allerdings das Kriterium der Effizienz für diesen Algorithmus nachgewiesen.
Die restlichen Algorithmen können keines der Kriterien des kontinuierlichen Lernens erfüllen, da sie alle auf das Meta-Lernen ausgelegt sind und dadurch mit einer sequenziellen Abfolge von Aufgaben nicht umgehen können.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[FAL17]	X	X	X	X	X	X	✓	X	X
[FL18]	X	X	X	X	X	X	✓	X	X
[Fin+19]	X	X	X	X	✓	X	X	X	X
[Raj+19]	X	X	X	X	X	X	✓	X	X
[Son+20]	X	X	X	X	X	X	✓	X	✓

Tabelle 5.2: Algorithmen der Parameter-Initialisierung im Überblick.

Auch die Adaptierbarkeit ist bei keinem der Algorithmen gegeben, da eine Adaptierung von vorhandenem Wissen nur über ein erneutes Training bezüglich aller zu lernenden Probleme umgesetzt werden kann. Dieses Vorgehen jedoch steht im Widerspruch zu der Definition von adaptiven Lernen.
Weiterhin dürfte ein Großteil der Algorithmen nicht gut parallelisierbar sein, da als Meta-Optimierer das Backpropagation-Verfahren verwendet wird. Eine Ausnahme bildet hier ES-MAML [Son+20], welcher eine evolutionäre Strategie als Meta-Optimierer verwendet. In Abschnitt 4.1.2 wurde schon erläutert, dass evolutionäre Algorithmen im Allgemeinen und somit auch NE-Techniken durch ihren Aufbau wesentlich besser parallelisierbar sind als Lernalgorithmen wie Backpropagation.
Somit lässt sich zusammenfassend sagen, dass sich diese Technik des klassischen Meta-Lernens eher für das Meta-Lernen eignet als für das kontinuierliche oder adaptive Lernen. Allerdings zeigt die Arbeit, dass sich diese Technik auch auf das kontinuierliche Lernen übertragen lässt, was somit eine interessante Forschungsrichtung eröffnet.

5.2.2 Optimierer-Modelle

Eine weitere Technik sind *Optimierer*-Modelle. Dabei wird eine Funktion erlernt, welche als Input einen Zustand (z. B. den Output des Basis-Modells) und den jeweiligen Fehler bekommt. Diese Funktion dient als Optimierer und produziert den Optimierungsschritt des neuronalen Netzes mit jeder Iteration der Basis-Optimierung. Anstatt Techniken wie den Gradientenabstieg zu nutzen, wird der Basis-Optimierer

selber durch das Meta-Lernen generiert und im Meta-Modell gespeichert. Jener Lernmechanismus kann dann dafür genutzt werden, das Basis-Modell für eine spezifische oder mehrere Aufgaben einer Domäne zu trainieren.
Die Arbeit von Andrychowicz u. a. [And+16] reduziert sogar noch zusätzlich die Anzahl der bei der Meta-Optimierung zu berücksichtigenden Parameter, indem auf die jeweiligen Neuronen des Basis-Optimierers mithilfe von Koordinaten zugegriffen wird. Damit verwendet dieser Algorithmus eine ähnliche Idee wie der HyperNEAT-Algorithmus, welcher in Abschnitt 5.2.3 noch detaillierter vorgestellt wird.
Die Arbeit [Wic+17] stellt eine Weiterentwicklung bzw. Verbesserung des Algorithmus aus [And+16] dar. Dabei wurden z. B. Best-Practices bezüglich der Topologie eines neuronalen Netzes (z. B. Attention-Mechanismus usw.) bei der Entwicklung der Architektur des Meta-Modells berücksichtigt. Weiterhin wurde der Algorithmus so konzipiert, dass er nicht für das Few-Shot-Lernen verwendet werden kann, sondern auch für die herkömmliche Optimierung mit mehreren Optimierungsschritten bzw. Trainingsbeispielen. Weiterhin hat der Algorithmus aus [And+16] auch das Problem, dass er nicht auf neuronale Netze mit abweichenden Aktivierungsfunktionen generalisieren kann. Auch diese Schwäche wird in [Wic+17] aufgegriffen und mithilfe des dort vorgestellten Algorithmus behoben. Dadurch ist das jeweilige Meta-Modell nicht nur auf unterschiedliche Topologien des Basis-Modells, sondern zusätzlich auch auf verschiedene Aktivierungsfunktionen generalisierbar.
Bei der Art des Meta-Modells, welches das gelernte Meta-Wissen speichert, können sich weitere Unterschiede ergeben. Einige Algorithmen verwenden z. B. LSTMs [RL17] oder RNNs [And+16; Wic+17]. Andere hingegen benutzen eher simple Repräsentationen wie parametrisierte Funktionen [Li+17]. Im Fall der parametrisierten Funktion [Li+17] wird als Basis-Optimierer weiterhin der Gradientenabstieg bzw. Backpropagation-Algorithmus verwendet, dessen Parameter dann aber teilweise durch das Meta-Modell vorgegeben werden.
Es gibt auch Algorithmen [RL17; Li+17], die Optimierer-Modelle mit der Technik der Parameter-Initialisierung kombinieren, indem beides zusammen trainiert wird. Dabei werden mittels Parameter-Initialisierung die Anfangsbedingungen für das Basis-Modell erlernt, sodass dieses noch schneller bzw. mit weniger Optimierungsschritten gegen ein (lokales) Optimum konvergieren kann.

Ein weiterer Algorithmus [Vuo+18] hat als Zielstellung das Problem des katastrophalen Vergessens im Basis-Modell einzudämmen. Dabei wird der Algorithmus aus der Arbeit [And+16] auf das Szenario des kontinuierlichen Lernens angepasst. Dazu wird zum einen die Vorgehensweise des Meta-Trainings angepasst und zum anderen der für die Meta-Optimierung genutzte (Meta-)Fehler. Herkömmlich wird der Meta-Fehler L^{meta} auf Basis der aktuellen Aufgabe T_j optimiert. Die Arbeit [Vuo+18] definiert einen Meta-Optimierungsschritt jedoch wie folgt:

$$w^* = \text{Adam}(\nabla_w L(f_\theta(T_{j-1} \cup T_j))) \tag{5.7}$$

Der Meta-Fehler wird bei diesem Algorithmus also nicht nur über die aktuelle Aufgabe T_j, sondern auch zusätzlich über die vorherige Aufgabe T_{j-1} berechnet. Dadurch kann gemessen werden, ob das Basis-Modell katastrophal vergessen hat oder nicht. Diese Information kann dann auch bei der Meta-Optimierung berücksichtigt werden. Dadurch wird bei diesem Algorithmus also ein Meta-Modell generiert, welches für das kontinuierliche Lernen eines Basis-Modells verwendet werden kann, das nicht katastrophal vergisst. Dadurch kann das Basis-Modell fortlaufend auf weitere Aufgaben mithilfe des Meta-Modells trainiert werden, ohne das Wissen bezüglich vorheriger Aufgaben katastrophal zu vergessen.

Fazit

Die Zielstellung der meisten Algorithmen dieser Technik besteht darin, das Lernen über eine Menge an Aufgaben mit wenigen Trainingsbeispielen pro Aufgabe (Few-Shot-Lernen) zu optimieren. Dazu müssen die jeweiligen Algorithmen in der Lage sein, das gesammelte Meta-Wissen bezüglich der Basis-Optimierung über mehrere Aufgaben zu generalisieren. In Tabelle 5.3 ist zu sehen, dass alle der untersuchten Algorithmen für genau dieses Kriterium der Generalisierbarkeit konzipiert sind. Hervorzuheben ist hier die Arbeit [Wic+17], welche ihr trainiertes Meta-Modell sogar auf einen komplett neuen Datensatz und somit eine andere Domäne überträgt und dadurch ebenfalls die Generalisierbarkeit nachweist. Deswegen erfüllt dieser Algorithmus nicht nur die Generalisierbarkeit auf neue Aufgaben innerhalb einer Domäne, sondern auch auf Aufgaben unterschiedlicher Domänen.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[And+16]	X	X	X	X	X	X	✓	X	X
[Wic+17]	X	X	X	X	X	X	✓	X	X
[RL17]	X	X	X	X	X	X	✓	X	X
[Li+17]	X	X	X	X	X	X	✓	X	X
[Vuo+18]	X	X	X	O	X	X	✓	X	X

Tabelle 5.3: Algorithmen der Optimierer-Modelle im Überblick.

Nur einer der Algorithmen ist auch für das kontinuierliche Lernen geeignet. Allerdings kann dieser auch nicht alle Kriterien in diesem Zusammenhang (vollkommen) erfüllen. Nur die Eindämmung des katastrophalen Vergessens beim Basis-Modell (= Stabilität) kann nachgewiesen werden. Dabei evaluiert die Arbeit [Vuo+18] den Algorithmus jedoch auf Sequenzen, die nur aus 2 oder 3 Aufgaben bestehen. Aus Formel 5.7 wird ersichtlich, dass der Meta-Optimierer nur die aktuelle und vorherige berücksichtigt. Daher ist davon auszugehen, dass das Wissen bezüglich der ersten Aufgabe mit steigender Zahl an Aufgaben immer mehr verdrängt wird. Die Ergebnisse des Experimentes, in dem drei anstatt zwei Aufgaben gelernt werden, deutet dieses Verhalten des Algorithmus auch schon an.
Da sämtliche Algorithmen (bis auf [Vuo+18]) für das Meta-Lernen konzipiert sind, kann das Kriterium der Adaptierbarkeit von keinem dieser Algorithmen erfüllt werden. Auch in [Vuo+18] wurde diese Art des Lernens nicht berücksichtigt, da beim Lernen einer neuen Aufgabe (wie schon erwähnt) auf den Datensatz bezüglich der aktuellen und vorherigen Aufgabe zurückgegriffen wird. Allerdings ist es bei der Technik der Optimierer-Modelle generell denkbar, dass diese sich durchaus für das adaptive Lernen eignen könnten. Dazu muss „nur“ das schon gelernte Wissen in dem Prozess der Basis-Optimierung berücksichtigt werden. Das könnte dann im Zusammenhang mit dem Meta-Lernen zu einer neuen Art des Lernens führen.
Alle untersuchten Algorithmen verwenden als Meta-Optimierer den Backpropagation-Algorithmus. Deswegen ist davon auszugehen, dass die einzelnen Algorithmen eher schwierig zu parallelisieren sind.

Generell kann man sagen, dass die Optimierer-Modelle ebenfalls eher für das Meta-Lernen ausgelegt sind. Allerdings gibt es auch hier wieder eine Arbeit, welche diese Technik des (klassischen) Meta-Lernens auf das kontinuierliche Lernen überträgt, indem der Meta-Trainingsprozess so angepasst wird, dass das jeweilige Meta-Modell auch erlernt, wie man das Basis-Modell sequenziell trainieren kann, ohne dass dieses katastrophal vergisst.

5.2.3 Hyper-Modelle

Die *Hyper-Modelle* [HDL16] sind eine weitere Technik des Meta-Lernens. Gelernt wird ein Meta-Modell, welches als Input einen Kontext-Vektor bekommt und als Ausgabe Parameter des Basis-Modells produziert, anstatt es iterativ (z. B. durch Gradientenabstieg) zu trainieren. Dabei wird das Meta-Wissen bzw. Meta-Modell durch ein eigenes neuronales Netz repräsentiert.
In Abschnitt 4.3.1 wurde mit dem CPPN ein ähnliches Modell vorgestellt, welches ebenfalls als Hyper-Modell interpretiert werden kann. Der HyperNEAT-Algorithmus [SDG09] nutzt in diesem Zusammenhang den NEAT-Algorithmus, um das jeweilige CPPN für ein gegebenes Basis-Modell zu entwickeln. Die Kernidee des Algorithmus ist es, die Gewichte der Verbindungen zwischen den einzelnen Neuronen als geometrische Funktion durch ein CPPN zu berechnen. Dabei bekommt ein jedes Neuron z. B. kartesische Koordinaten in dem sogenannten *Substrat* zugewiesen, welches das jeweilige Koordinatensystem und die Anordnung der Neuronen des Basis-Modells vorgibt. Dabei kann das Substrat unterschiedliche Formen annehmen, wie in Abbildung 5.5 zu sehen ist.
So lassen sich Neuronen in einem Grid (a) oder auch in einem dreidimensionalen Raum (b) anordnen. Bei dem Zustandsraum-Sandwich (c) können die Neuronen einer Schicht nur Verbindungen zu Neuronen der jeweils anderen Schicht besitzen. Eine weitere Anordnungsvariante ist die Kreisförmige (d), bei der die Positionen der Neuronen mit Polar-Koordinaten angegeben werden.
Die Parameter des Basis-Modells werden dann durch das jeweilige CPPN generiert. In Abbildung 5.6 ist ein Beispiel dafür zu sehen. Nimmt man sich z. B. einen zweidimensionalen Raum (mit x- und y-Koordinaten) und möchte nun das Kantengewicht zwischen zwei Neuronen mittels des jeweiligen CPPNs ermitteln, so geschieht dies

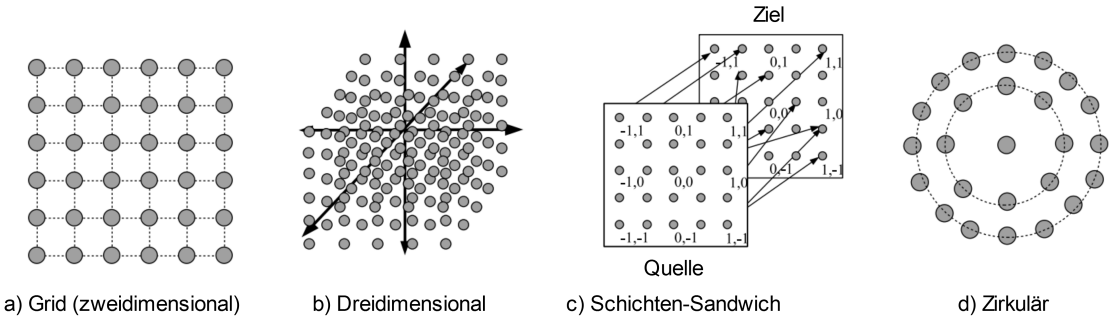


Abbildung 5.5: Mögliche Anordnungen von Neuronen in einem Substrat (in Anlehnung an [SDG09]).

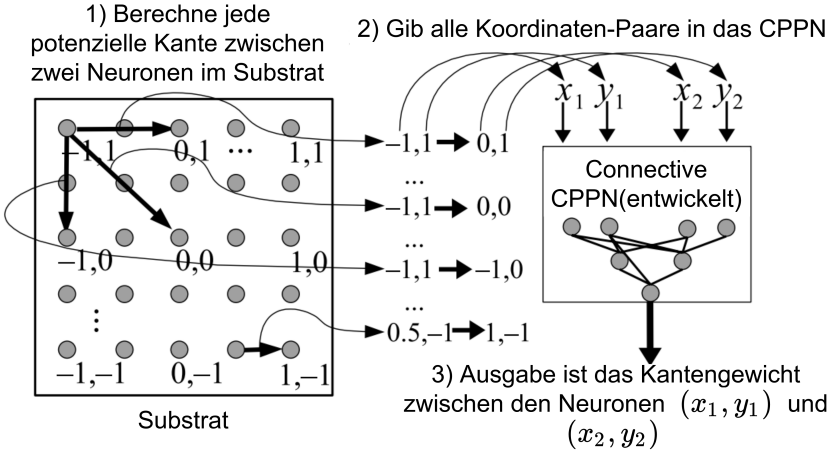


Abbildung 5.6: Berechnung von Kantengewichten mittels eines connective-CPPN (in Anlehnung an [SDG09]).

mithilfe einer Funktion $f(x_1, y_1, x_2, y_2) = w_{x_1,y_1 \to x_2,y_2}$. Das CPPN bekommt also als Eingabe die Koordinaten der jeweiligen Neuronen und bildet diese dann auf einen Skalar ab, welcher das Gewicht der Kante zwischen den jeweiligen Neuronen darstellt.
So wurde der HyperNEAT-Algorithmus schon für viele verschiedene Szenarien eingesetzt. Wie in Unterkapitel 4.3 schon erwähnt, kann die Faltung in CNNs ebenfalls als sich wiederholendes geometrisches Verbindungsmuster der neuronalen Struktur verstanden werden. So wurden mit dem HyperNEAT-Algorithmus in Kombination mit SGD ebenfalls Muster gefunden, welche Ähnlichkeiten zu den in der Faltung verwendeten Mustern haben (vgl. [Fer+16]).
In [HDL16] werden sogenannte *Hypernetworks* eingeführt, welche die indirekte Codierung des HyperNEAT-Algorithmus so weiterentwickeln, dass die Kantengewichte des Meta-Modells komplett durch SGD trainiert werden können. Dadurch ist es möglich, die Performance von LSTMs für die Sprach-Modellierung und andere Aufgaben zu steigern (vgl. [HDL16]).
Ein wiederum anderer Algorithmus [Bro+17] nutzt die Hyper-Modelle nicht nur für die bloße Generierung von Parametern des Basis-Modells, sondern versucht mit ihnen die Suche nach einer guten Topologie des Basis-Modells effizienter zu gestalten. Dabei werden zunächst zufällige Basis-Modelle generiert, deren Parameter dann durch das Hyper-Modell vorgegeben werden. Anschließend wird ein erster Performance-Test der Basis-Modelle durchgeführt. Danach werden die einzelnen Basis-Modelle mit dem Backpropagation-Algorithmus trainiert. Anschließend wird ein zweiter Performance-Test der trainierten Basis-Modelle durchgeführt. Die Korrelation zwischen den Ergebnissen der beiden Perfomance-Tests wird dann als Meta-Feedback verwendet, um das jeweilige Hyper-Modell verbessern zu können. Hat das Hyper-Modell eine zufriedenstellende Performance erreicht, so kann es dafür verwendet werden, die Parameter topologisch unterschiedlicher Basis-Modelle zu generieren, anstatt jedes Modell einzeln zu trainieren. Dadurch kann die Perfomance-Evaluation der einzelnen Basis-Modelle effizienter gestaltet werden, was wiederum die Suche nach einer guten Topologie des Basis-Modells schneller gestaltet.

Fazit

Die Hyper-Modelle sind ebenfalls eine Technik, die keine direkte Auswirkung auf das Lernverhalten des Basis-Modells hat, da die endgültigen Gewichte eines Basis-NN durch das Meta-Modell generiert werden sollen. Die Generalisierbarkeit ist dennoch gegeben, da sie ebenfalls parallel über mehrere Aufgaben trainiert werden, was das in ihnen gespeicherte Wissen „verallgemeinert“ und somit auch Auswirkungen auf die Generalisierbarkeit des Basis-NN hat. Ein Vorteil dieser Art des Meta-Lernens ist, dass der Trainingsaufwand für das Basis-Modell fast komplett entfällt, da das jeweilige Meta-Modell direkt die finalen Gewichte des Basis-Modells generiert und somit ein weiteres Training durch den Basis-Optimierer hinfällig wird.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[SDG09]	X	X	X	X	X	X	(✓)	(✓)	✓
[HDL16]	X	X	X	X	X	X	✓	✓	X
[Bro+17]	X	X	X	X	X	X	✓	✓	X

Tabelle 5.4: Algorithmen der Hyper-Modelle im Überblick.

Eine Besonderheit stellt jedoch der HyperNEAT-Algorithmus [SDG09] dar. Dieser trainiert das CPPN nicht nach dem Paradigma des Meta-Lernens auf mehreren Aufgaben einer Domäne, sondern nach der herkömmlichen Vorgehensweise des maschinellen Lernens auf einer Einzelnen. Es ist jedoch davon auszugehen, dass dieser Algorithmus ebenso gut generalisiert wie seine Äquivalente, da die HyperNets [HDL16] im Grunde genommen auch nur eine Weiterentwicklung des HyperNEAT-Algorithmus sind.

Es sei auch noch erwähnt, dass das Hauptziel der Hyper-Modelle auch eigentlich gar nicht in den hier betrachteten Vergleichskriterien besteht. Diese Art von Modellen bzw. Algorithmen wird hauptsächlich für die Komprimierung des Parameter-Suchraumes verwendet, in dem der jeweilige Optimierer arbeiten soll. Dadurch kann z. B. die Gewichtsoptimierung von tiefen neuronalen Netzen mit Millionen von Parametern durch eine einfache NE-Technik ermöglicht werden (vgl. [SDG09]). Das wiederum macht sie im Zusammenhang mit der Wahl der Repräsentation des Geno-

typen des Basis- oder auch Meta-Modells interessant. Der HyperNEAT-Algorithmus [SDG09] zeichnet sich zudem durch gute Parallelisierbarkeit aus, da er den NEAT-Algorithmus als Optimierer verwendet.
Was alle Algorithmen dieser Technik gemein haben, ist, dass keiner von ihnen auf das kontinuierliche oder adaptive Lernen ausgelegt ist. Jedoch existieren vor allem im Zusammenhang mit dem HyperNEAT-Algorithmus weitere Arbeiten, die diesem Fakt begegnen wollen. Diese Arbeiten werden jedoch Inhalt von späteren Abschnitten sein. Somit kann abschließend auch für diese Technik festgehalten werden, dass sie eher für das Meta-Lernen geeignet ist.

5.2.4 Hybride Modelle

Die Technik der *hybriden Modelle* zeichnet sich dadurch aus, dass Basis- und Meta-Modell aus verschiedenen Komponenten bestehen können. Der OML-Algorithmus [JW19] z. B. unterteilt ein neuronales Netz in zwei Komponenten, die dann als Meta- und Basis-Modell interpretiert werden können. In Abbildung 5.7 ist ein Beispiel für ein OML-Netz und seine zwei Komponenten zu sehen.

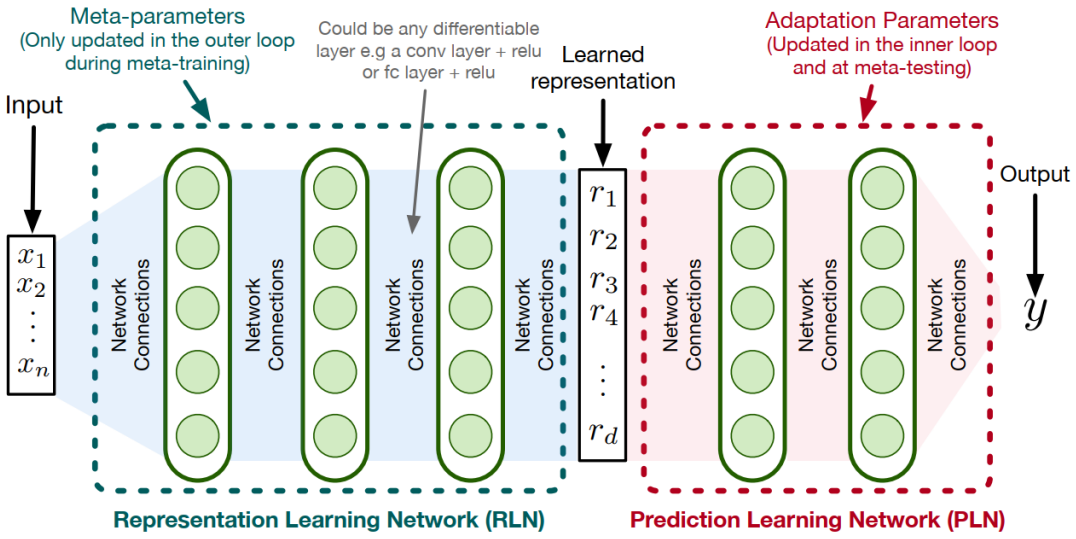


Abbildung 5.7: Die Architektur eines OML-NN (Quelle: [JW19]).

Die erste Komponente ist das Repräsentationsnetzwerk (RLN), welches generalisierte Repräsentationen über mehrere Aufgaben hinweg erlernen soll (Langzeitspeicher = Meta-Modell). Die zweite Komponente, das Prädiktionsnetzwerk (PLN), wird hingegen aufgabenspezifisch trainiert (Kurzzeitspeicher = Basis-Modell). Dadurch soll dem Effekt des katastrophalen Vergessens in Bezug auf das Basis-Wissen entgegengewirkt werden.
Eine weitere Variante ist es, sogenannte *neuronale Turing-Maschinen* [AGI14] zu verwenden. Dabei handelt es sich um neuronale Netze mit einem externen Speicher, welche es ermöglichen, ältere Daten jederzeit abzurufen. In [San+16] wird der externe Speicher einer neuronalen Turing-Maschine so modifiziert, dass ein assoziativer Zugriff (äquivalent zum Zugriffsmechanismus in neuronalen Netzen) auf dessen Inhalte möglich ist. Dabei dienen die Gewichte des Basis-Modells (welche mittels Gradientenabstieg trainiert werden) als Langzeitspeicher und der externe Speicher (in welchem die jeweiligen Informationen explizit abgespeichert werden) als Kurzzeitspeicher.

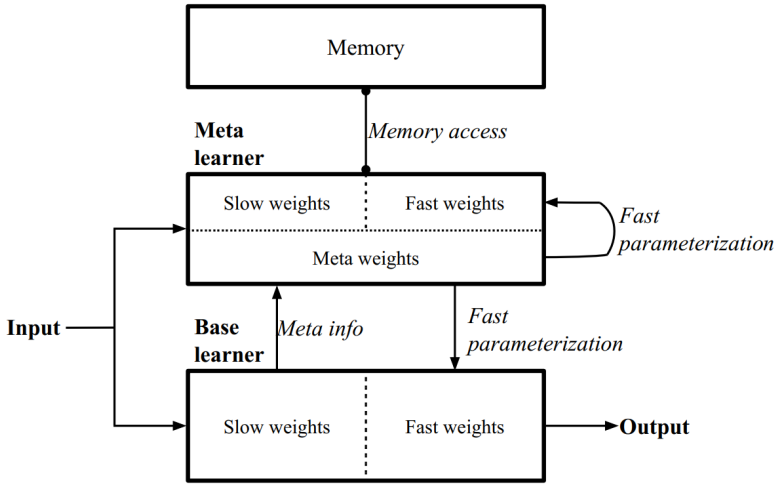


Abbildung 5.8: Die Architektur eines MetaNets (Quelle: [MY17]).

Mit ihren *MetaNets* [MY17] kombinieren Munkhdalai und Yu quasi die beiden obigen Ansätze miteinander. Dabei werden das Meta- und Basis-Modell wieder durch verschiedene neuronale Netze repräsentiert. Zusätzlich existiert ein externer Speicher (wie bei den neuronalen Turing-Maschinen), auf den das Meta-Modell zugreifen

kann. Beide Netzwerke haben dabei zwei Gewichte pro Verbindung (langsame und schnelle). In Abbildung 5.8 ist der Aufbau eines MetaNets zu sehen.
Die langsamen Gewichte von Basis- und Meta-Modell werden dabei durch einen Lernalgorithmus (z. B. SGD) trainiert. Die schnellen Gewichte hingegen werden mit jedem Trainingsbeispiel oder pro Aufgabe neu berechnet. Tiefer wird an dieser Stelle nicht auf diesen Algorithmus eingegangen, da alleine das Meta-Modell aus drei verschiedenen neuronalen Netzen besteht und der Trainingsprozess der kompletten Architektur sehr komplex ist. Die komplette Erläuterung würde den Rahmen dieses Abschnittes sprengen.

Fazit
Die hybriden Modelle verwenden verschiedene Komponenten für das langsame und schnelle Lernen. Die MANNs [San+16] verwenden z. B. zusätzlich zum Basis-Modell (langsame Komponente) noch einen Speicher (schnelle Komponente). Dadurch generalisieren sie zwar gut, allerdings werden die restlichen Kriterien nicht hinlänglich erfüllt.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[San+16]	X	X	X	X	X	X	✓	X	X
[MY17]	X	O	O	X	✓	X	✓	X	X
[JW19]	X	X	✓	X	X	X	✓	✓	X

Tabelle 5.5: Algorithmen der hybriden Modelle im Überblick.

Die MetaNets [MY17] hingegen können recht vielfältig eingesetzt werden. Durch die Kombination von langsamen und schnellen Gewichten, der Verwendung eines externen Speichers und Unterscheidung zwischen Basis- und Meta-Modell erfüllen sie mehrere der untersuchten Kriterien. So wurde für die MetaNets z. B. nachgewiesen, dass das Meta-Modell gegen das Phänomen des katastrophalen Vergessens resistent ist. Allerdings geschah die Evaluation diesbezüglich nur auf zwei Aufgaben. Dadurch kann keine Aussage darüber getroffen werden, ob sich diese Ergebnisse auf Domänen mit mehreren Aufgaben übertragen lassen. Jedoch wurde ein erfolgreicher

Wissenstransfer in der Arbeit zu den MetaNets nachgewiesen. Das bedeutet, dass sich die Performance des Meta- und somit auch des Basis-Modells bezüglich einer älteren Aufgabe erhöht hatte, nachdem es auf einer neuen Aufgabe trainiert wurde (Rückwärtstransfer). Dabei schlug die Trainingsperformance allerdings ins Negative um, wenn das Modell zu oft auf der neuen Aufgabe trainiert wurde (= negativer Wissenstransfer).
Die Adaptierbarkeit des Algorithmus [MY17] ist nur bedingt gegeben. Zwar generalisiert das Meta-Modell so gut, dass kleinere Veränderungen in der jeweiligen Domäne keine Auswirkungen haben. Jedoch wurde dieser Algorithmus, wie schon erwähnt, nur durch zwei unterschiedliche Aufgaben getestet, sodass eine endgültige Aussage an dieser Stelle ausbleibt.
Der OML-Algorithmus [JW19] hingegen ist so konzipiert, dass das Meta-Modell das Kriterium der Stabilität erfüllt. Dadurch kann auf bisher gelerntes Wissen bezüglich einer Aufgabe ohne ein erneutes Training zurückgegriffen werden. Allerdings kann bei diesem Algorithmus kein Wissenstransfer irgendeiner Art nachgewiesen werden, weswegen das Kriterium der Effizienz leider unerfüllt bleibt. Generell verwenden alle Algorithmen den Backpropagation-Algorithmus als Meta-Optimierer, der sich durch schlechte Parallelisierbarkeit auszeichnet.
Zu der Technik der hybriden Modelle kann das Fazit gezogen werden, dass sie im Gegensatz zu den bisher vorgestellten Techniken zusätzlich zum Meta-Lernen auch für das kontinuierliche Lernen verwendet werden können. Das liegt vermutlich daran, dass die unterschiedlichen Komponenten in verschiedenen Geschwindigkeiten trainiert werden und dadurch das Speichern von (generellen) Langzeitwissen und (aufgabenspezifischen) Kurzzeitwissen möglich machen. Somit stellt diese Technik einen interessanten Ansatz in Bezug auf das Intra-Life-Learning dar.

5.2.5 Fehler-Modelle

Die *Fehler*-Modelle gehören vielleicht zu den außergewöhnlichsten Ansätzen des Meta-Lernens. Die Kernidee ist, Modelle (meistens neuronale Netze) zu erlernen, welche auf Grundlage des Outputs des Basis-Modells den Fehlerwert berechnen, der dann z. B. vom Backpropagation-Algorithmus verwendet wird, um das Basis-NN zu trainieren. An dieser Stelle werden also keine Parameter des neuronalen Netzes

oder der Optimierer selber gelernt, sondern es wird eine künstliche Fehlerfunktion generiert. Das wiederum kann zu einem Fehler führen, der z. B. die Optimierung des Basis-Modells vereinfacht. In [Rei+18] wird eine evolutionäre Strategie verwendet, mithilfe derer eine differenzierbare Fehlerfunktion entwickelt wird. Als Meta-Modell wird hierbei ebenfalls eine bestimmte NN-Architektur verwendet, um die jeweilige Fehlerfunktion kodieren zu können. Durch die Verwendung von evolutionären Strategien als Meta-Optimierer soll die entwickelte Fehlerfunktion auch auf unterschiedliche Aufgaben generalisieren können, die nicht Bestandteil der Trainingsmenge waren.

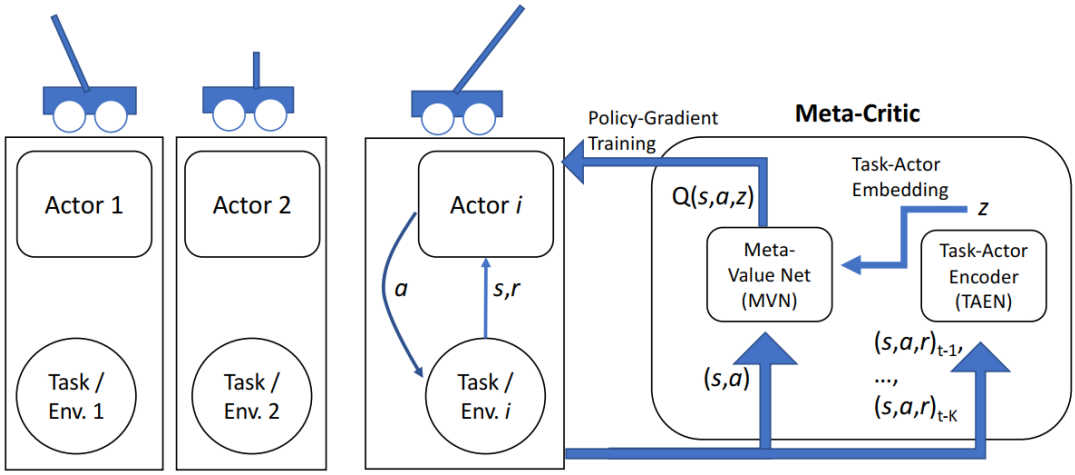


Abbildung 5.9: Das Task-Actor-Modell zum Lernen einer Fehler-Funktion (Quelle: [Sun+17]).

Im Gegensatz dazu stehen die Ansätze, die den Gradientenabstieg als Meta-Optimierer verwenden. Der *Meta-Critic*-Algorithmus [Sun+17] verwendet z. B. ein drittes, separates neuronales Netz, um ein sogenanntes *Task-Actor-Embedding* zu berechnen, welches als zusätzlicher Input für das eigentliche Meta-Modell (MVN) dient (siehe Abbildung 5.9). Die Arbeit [Zho+20] hingegen, versucht nicht die herkömmlich verwendete Fehlerfunktion vollständig zu ersetzen, sondern sie um eine gelernte Fehlerfunktion zu erweitern. Dabei werden dann also für den Optimierungsschritt des Basis-Modells die herkömmliche und künstliche Fehlerfunktion zusammen verwendet.

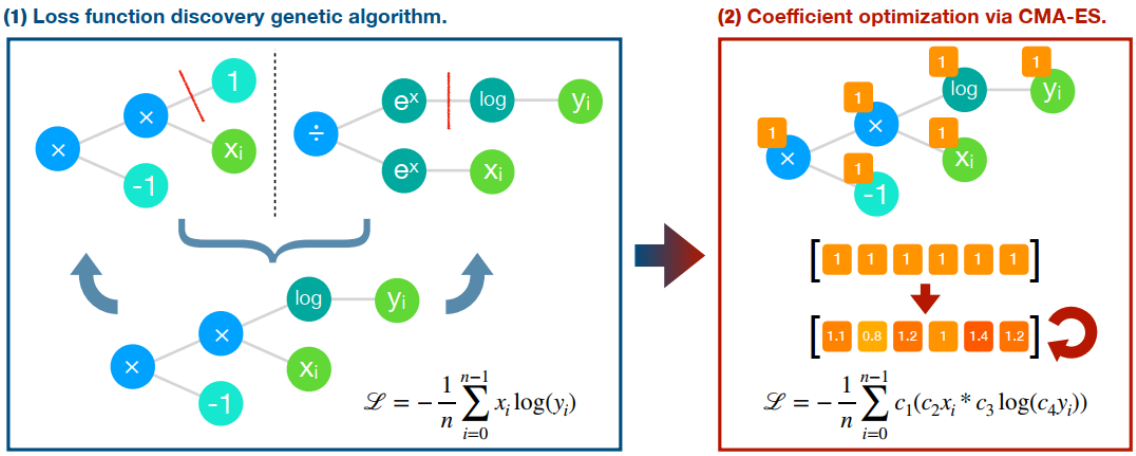


Abbildung 5.10: Genetischer Generierungsprozess einer Fehlerfunktion (Quelle: [GM19]).

Die Arbeit [GM19] stellt wiederum einen Algorithmus zum Generieren einer Fehlerfunktion vor, welche zu schnellerem Lernen mit verbesserter Generalisierung führt. Dabei wird die jeweilige Fehlerfunktion durch einen GP-Algorithmus generiert, wofür eine hierarchisch organisierte Baumstruktur als Meta-Modell verwendet wird. Zusätzlich wird der CMA-ES-Algorithmus [HO96](**C**ovariance **M**atrix **A**daption **E**volution **S**trategie) genutzt, um die generierte Fehlerfunktion mit Koeffizienten für die einzelnen Variablen zu verfeinern. Der Prozess ist exemplarisch in Abbildung 5.10 zu sehen.

Andere Algorithmen [BSC18; Li+19] generieren wiederum Fehlerfunktionen, dessen Minima in Basis-Modellen resultieren, die robust gegenüber einem Domänen-Wechsel sind. In [BSC18] wird dazu eine Art *Regularisierungs*-Funktion gelernt, welche die genutzte Fehlerfunktion auf ungesehene (ähnliche) Aufgaben generalisieren kann. Li u. a. [Li+19] hingegen nutzen einen Fehler, der entweder ausschließlich oder zusätzlich zur herkömmlichen Fehlerfunktion für das Training des Basis-Modells verwendet wird.

Ein anderer Algorithmus [GSS19] versucht eine vorhandene Fehlerfunktion mithilfe eines neuronalen Netzes (= Meta-Modell) zu approximieren. Dadurch können für das Training des Basis-Modells auch Fehlerfunktionen verwendet werden, welche nicht differenzierbar sind. Da bei vielen Ansätzen Algorithmen als Basis-Optimierer

genutzt werden, welche den Gradientenabstieg verwenden, ist die Differenzierbarkeit der Fehlerfunktion eigentlich eine zwingende Voraussetzung. Dieser Algorithmus besitzt das Potenzial, diese Einschränkung obsolet zu machen.

Fazit

In Tabelle 5.6 ist zu sehen, dass die Algorithmen überwiegend nur das Kriterium der Generalisierbarkeit erfüllen. Allerdings gibt es in diesem Kontext auch einige „Ausreißer“, was daran liegt, dass einige der Algorithmen nur für das Erlernen einer Fehlerfunktion bezüglich einer einzelnen Aufgabe konzipiert sind.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[Sun+17]	X	X	X	X	X	X	✓	X	X
[Rei+18]	X	X	X	X	X	X	✓	X	✓
[BSC18]	X	X	X	X	X	X	✓	X	X
[GM19]	X	X	X	X	X	X	✓	X	✓
[Li+19]	X	X	X	X	X	X	✓	X	X
[GSS19]	X	X	X	X	X	X	X	X	X
[Zho+20]	X	X	X	X	X	X	X	X	X

Tabelle 5.6: Algorithmen der Fehler-Modelle im Überblick.

Zwei Algorithmen erfüllen zusätzlich noch das Kriterium der Parallelisierbarkeit, da bei ihnen ein evolutionärer Algorithmus als Meta-Optimierer verwendet wird.
Die Kriterien des kontinuierlichen Lernens können von keinem der Algorithmen erfüllt werden. Die Begründung dafür liegt in dem Fakt, dass keine der Arbeiten den jeweiligen Algorithmus für ein sequenzielles Lernszenario evaluiert hat, sondern nur für das Meta-Lernen, bei dem die Daten alle auf einmal zur Verfügung stehen.
Ebenso betrachtet keine Arbeit die Adaptierbarkeit des Meta- oder Basis-Modells, sodass auch dieses Kriterium von keinem der aufgelisteten Algorithmen erfüllt werden kann.
Generell sei zu der Technik der Fehler-Modelle gesagt, dass es sich dabei wohl eher um eine Ergänzung zu anderen Techniken handelt. So könnte man z. B. diese Art

von Algorithmen gut mit den Optimierer-Modellen kombinieren, um somit ein noch effizienteres Training ermöglichen zu können. Ansonsten ist diese Technik des klassischen Meta-Lernens in Bezug auf die hier betrachteten Lernszenarien nicht sehr vielseitig einsetzbar.

5.2.6 Fazit

Wie in Tabelle 5.7 zu sehen, sind die Techniken des (klassischen) Meta-Lernens eher auf Generalisierbarkeit als auf die anderen Kriterien ausgelegt. Eine Ausnahme bilden dabei jedoch die hybriden Modelle aus Abschnitt 5.2.4. Die dort vorgestellten Algorithmen lassen sich zwar allesamt nicht wirklich gut parallelisieren, allerdings scheinen sie im Gegensatz zu den Algorithmen der restlichen Techniken besser für das kontinuierliche Lernen geeignet zu sein, was eines der Kernelemente des Intra-Life-Learning darstellt. Auch erfüllt einer der Algorithmen dieser Technik (wenn auch nur eingeschränkt) die Adaptierbarkeit. Das ist bei sonst keiner anderen Technik der Fall.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
Technik 5.2.1	0/5	0/5	0/5	0/5	1/5	0/5	4/5	0/5	1/5
Technik 5.2.2	0/5	0/5	0/5	(1)/5	0/5	0/5	5/5	0/5	0/5
Technik 5.2.3	0/3	0/3	0/3	0/3	0/3	0/3	2/3	2/3	1/3
Technik 5.2.4	0/3	(1)/3	(2)/3	0/3	1/3	0/3	3/3	1/3	0/3
Technik 5.2.5	0/7	0/7	0/7	0/7	0/7	0/7	5/7	0/7	2/7
$\sum$	**0/23**	**(1)/23**	**(2)/23**	**(1)/23**	**2/23**	**0/23**	**19/23**	**3/23**	**4/23**

Tabelle 5.7: Techniken des Meta-Lernens im Überblick.

Auch eine Arbeit zu den Optimierer-Netzen aus Abschnitt 5.2.2 bezieht sich auf das Szenario des kontinuierlichen Lernens. Dabei erfüllt dieser eine Algorithmus jedoch das Kriterium der Stabilität ebenfalls nur mit Einschränkungen. Die Arbeit liefert trotzdem einen guten Ansatz, der in späteren Arbeiten zu diesem Thema durchaus aufgegriffen werden kann.

Zusammenfassend kann man festhalten, dass die Ansätze in der Tat überwiegend (nur) für das Meta-Lernen ausgelegt sind. Dabei wird jedoch davon ausgegangen, dass die jeweiligen Trainingsdaten von Anfang an komplett zur Verfügung stehen und dementsprechend auch nur ein initialer (Meta-)Trainingsprozess vorgesehen ist. Möchte man die jeweiligen Modelle dann auf neue Trainingsdaten anpassen, so muss das komplette Training für die neuen Daten wiederholt werden. Dieses Vorgehen steht jedoch im Gegensatz zum kontinuierlichen und adaptiven Lernen. Es gibt allerdings schon einzelne Arbeiten, die diesem Umstand begegnen und bestehende Meta-Lernalgorithmen so weiterentwickeln, dass ein sequenzielles Training im Sinne des kontinuierlichen Lernens möglich ist. Wie in Abschnitt 5.1.2 schon erläutert, bringt dieses Lernszenario allerdings eine Reihe von Anforderungen mit sich, von denen kein hier untersuchter Algorithmus alle erfüllen kann. Dass es aber Algorithmen gibt, die Meta-Lern-Techniken erfolgreich auf einzelne Anforderungen des kontinuierlichen Lernens anpassen können, zeigt, dass in diese Richtung eindeutig Forschungspotenzial besteht.

5.3 Synaptische Plastizität

Ein anderer Ansatz, um das Meta-Lernen zu betrachten, ist die sogenannte *synaptische Plastizität*, welche einer der Kernmechanismen hinter dem neuronalen Lernen in der Natur ist (vgl. [SSR18]). Dabei geht es darum, Lernregeln zu entdecken, welche vorschreiben, wie sich die Gewichte von Verbindungen ändern sollten, auf Basis der Aktivierungszustände ihrer Quell- und Ziel-Neuronen (vgl. [Sta+19]).
Viele Algorithmen nutzen Varianten der sogenannten *Hebb*-Regel, um die Plastizitäten der Verbindungen eines neuronalen Netzes zu kodieren. Während für das Training des Basis-Modells Varianten der Hebb-Regel verwendet werden, nutzt der Meta-Optimierer dann verschiedene Techniken, um die jeweiligen Komponenten der Hebb-Regel und somit des Basis-Optimierers zu entwickeln.
Im nächsten Unterkapitel 5.3.1 wird die Hebb-Regel und einige ihrer Varianten genauer vorgestellt. Anschließend werden in den restlichen Abschnitten 5.3.2, 5.3.3, 5.3.4 und 5.3.5 die verschiedenen Techniken erläutert, welche diese Regeln verwenden, um die synaptische Plastizität in neuronalen Netzen zu ermöglichen. Der letzte

Abschnitt 5.3.6 wird dann ein Fazit zu diesem Ansatz des Meta-Lernens in Bezug auf seine Anwendbarkeit in den unterschiedlichen Lernszenarien ziehen.

5.3.1 Hebb-Regel und ihre Varianten

Die *Hebb-Regel* [Heb49] ist eine weitverbreitete Technik, um die synaptische Plastizität eines neuronalen Netzes zu steuern. Diese Regel legt fest, wie sich das Gewicht der Verbindung zwischen zwei Neuronen auf Basis ihrer Aktivierungszustände verändert. Eine simple Darstellungsweise der Hebb-Regel ist:

$$\Delta w_{i \to j} = \eta_w a_i a_j, \tag{5.8}$$

Diese beschreibt die Veränderung im Gewicht der Kante von Neuron i nach Neuron j, als eine Funktion ihrer Aktivierungszustände a_i und a_j. Der Koeffizient η entscheidet über den Grad der synaptischen Plastizität einer jeden Verbindung, welche wiederum durch verschiedene Optimierungsverfahren entwickelt werden kann. Je häufiger also zwei Neuronen zusammen aktiv werden, umso stärker ist die synaptische Verbindung zwischen ihnen („What fires together, wires together."). Dabei gehört diese Regel in die Familie der lokalen Mechanismen, welche die synaptische Plastizität einer Verbindung auf Basis der lokalen Aktivität steuert (vgl. [NR20]). Jedoch gibt es mehrere Varianten dieser Regel, die die Gleichung 5.8 um verschiedene Elemente erweitern. Ein Beispiel dafür ist das generalisierte Hebb-ABC-Modell, welches die Gewichte einer Verbindung zwischen zwei Neuronen folgendermaßen berechnet (vgl. [RS10]):

$$\Delta w_{i \to j} = \eta_w * (A_w a_i a_j + B_w a_i + C_w a_j) \tag{5.9}$$

Dabei ist A_w der Korrelationskoeffizient, B_w der Koeffizient für die präsynaptischen Aktivierung a_i und C_w der Koeffizient für die postsynaptische Aktivierung a_j. Zusätzlich kann zur Gleichung 5.9 noch ein weiterer Koeffizient D_w hinzugefügt werden (vgl. [NR20]):

$$\Delta w_{i \to j} = \eta_w * (A_w a_i a_j + B_w a_i + C_w a_j + D_w) \tag{5.10}$$

Diese Variante wird dann als Hebb-ABCD-Modell bezeichnet, wobei der zusätzliche Koeffizient D_w als Bias einer jeden synaptischen Verbindung zwischen den jeweils verbundenen Neuronen interpretiert werden kann.
Ein weiteres Beispiel für eine Modifikation der Hebb-Regel ist die Oja-Regel, welche das *Stabilisierungsproblem* der Hebb-Regel durch multiplikative Normalisierung löst (vgl. [Oja08]):

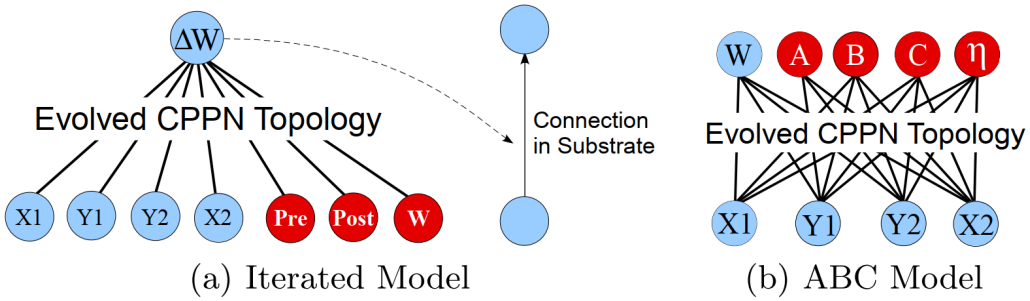
$$\Delta w_{i \to j} = \eta_w * (a_i a_j - a_j^2 w_{i \to j}) \tag{5.11}$$

Dabei ist der neue Wert $\Delta w_{i \to j}$ des Verbindungsgewichts abhängig von dem alten Gewichtswert $w_{i \to j}$. Es gibt noch viele weitere Varianten der Hebb-Regel (vgl. [Vas+11]). Jedoch wird an dieser Stelle auf die Erläuterung weiterer Regeln verzichtet, da dies nicht weiter zuträglich wäre.

5.3.2 NEAT-basierte Ansätze

Die Arbeiten [RS10; RS12; NR20] verwenden den NEAT-Algorithmus, um die unterschiedlichen Parameter der jeweiligen Hebb-Regel zu optimieren. Der *adaptive HyperNEAT*-Algorithmus [RS10] nutzt die in Abschnitt 4.3.1 erläuterten CPPNs, um somit die Parameter der Hebb-Regel für jede synaptische Verbindung einzeln zu optimieren. Dabei generiert der NEAT-Algorithmus dann das jeweilige CPPN.
Risi und Stanley unterscheiden zusätzlich noch zwischen verschiedenen Varianten des CPPN, welche in Abbildung 5.11 zu sehen sind. Dabei kodiert das iterative Modell (a) die einfache Hebb-Regel direkt für jede Verbindung des Basis-Modells. Als Grundlage werden dabei dann die Koordinaten der prä- und postsynaptischen Neuronen, deren jeweiliger Aktivierungszustand und das aktuelle Gewicht der entsprechenden Verbindung verwendet. Das CPPN produziert dann das neue Gewicht. Diese Variante nennt sich *iterativ*, da sie für jeden Input erneut die Gewichte des Basis-Modells berechnet. Das ABC-Modell (b) hingegen berechnet einmalig die einzelnen Parameter der Hebb-ABC-Regel und die initialen Gewichte der Verbindungen. Auch hier werden die Koordinaten der prä- und postsynaptischen Neuronen als Input für das CPPN verwendet. Die Gewichte des Basis-Modells werden bei diesem Vorgehen ebenfalls mit jedem Input neu berechnet, allerdings nicht direkt durch das

CPPN, sondern durch die Hebb-ABC-Regeln einer jeden Verbindung, deren Parameter durch das CPPN generiert werden.



(a) Iterated Model (b) ABC Model

Abbildung 5.11: Die Varianten eines CPPN (Quelle: [RS10]).

Der *adaptive ES-HyperNEAT*-Algorithmus [RS12] geht dann noch einen Schritt weiter. Hier werden nicht nur die Parameter einer Lernregel oder die Lernregel selbst durch den NEAT-Algorithmus generiert, sondern auch die Anordnung der Neuronen des Basis-Modells im Substrat, durch das die Neuronen ihre Koordinaten zugewiesen bekommen. Dadurch wird zwar nicht die Topologie des Basis-Modells verändert, aber seine *Topographie*[2].
Es existieren allerdings auch Arbeiten, welche direkt die jeweiligen Parameter einer Lernregel optimieren. Najarro und Risi [NR20] z. B. erzeugen die jeweiligen Parameter der Lernregel einer jeden Verbindung direkt mit einer evolutionären Strategie. Ein Nachteil bei dieser Vorgehensweise ist jedoch, dass der jeweilige Optimierer (in diesem Fall die evolutionäre Strategie) deutlich mehr Parameter zu optimieren hat, als wenn man z. B. ein CPPN dafür verwendet.

Fazit

Es ist recht eindeutig in Tabelle 5.8 zu erkennen, worauf die hier vorgestellten Algorithmen ausgelegt sind. Die einzelnen Arbeiten haben ihren jeweiligen Algorithmus meistens im Rahmen einer Aufgabe aus dem Bereich des bestärkenden Lernens evaluiert und untersucht, inwiefern der Algorithmus das jeweilige Basis-Modell auf sich verändernde Umstände innerhalb der gegebenen Umgebung anpassen kann. Deswe-

[2] Topographie bezeichnet in diesem Zusammenhang die Art der räumlichen Anordnung von den Neuronen eines neuronalen Netzes zueinander.

gen können auch alle hier aufgeführten Algorithmen das Kriterium der Adaptierbarkeit im Basis-Raum erfüllen.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[RS10]	?	✓	?	X	?	X	?	X	✓
[RS12]	?	✓	?	X	?	X	?	X	✓
[NR20]	?	✓	?	X	?	X	?	X	✓

Tabelle 5.8: NEAT-basierte Algorithmen der synaptischen Plastizität im Überblick.

Die Kriterien des kontinuierlichen Lernens hingegen können nicht erfüllt werden. Man könnte zwar argumentieren, dass beim bestärkenden Lernen die Aufgabe sequenziell bearbeitet wird, allerdings handelt es sich dabei immer um dieselbe Aufgabe. Deswegen ist auch nicht davon auszugehen, dass die einzelnen Algorithmen in diesem Kontext das Kriterium der Stabilität erfüllen können. Das Kriterium der Effizienz wurde ebenfalls weder für den Basis- noch für den Meta-Raum von irgendeiner Arbeit untersucht. Deswegen ist auch hier davon auszugehen, dass ein Wissenstransfer in keiner Weise stattfindet, da die einzelnen Algorithmen ganz offensichtlich nicht dafür konzipiert sind.
Auch die Generalisierbarkeit ist bei diesen Algorithmen nicht gegeben. Dies ist auf dieselben Gründe zurückzuführen, aus denen auch keines der Kriterien des kontinuierlichen Lernens erfüllt werden können. Allerdings haben alle drei Algorithmen den Vorteil, dass sie entweder den NEAT-Algorithmus oder eine evolutionäre Strategie als Meta-Optimierer verwenden und deswegen sehr gut parallelisierbar sein dürften. Insgesamt kann für diese Technik festgehalten werden, dass ihr Zweck eindeutig in dem Erlernen von effizienten Algorithmen für das adaptive Lernen liegt. Allerdings wird dabei nicht dem Trainingsprozess des Meta-Lernens gefolgt, sondern die jeweiligen Algorithmen arbeiten wie herkömmliche Verfahren des maschinellen Lernens auf einzelnen Aufgaben. Deswegen können die einzelnen Algorithmen auch nicht auf die Kriterien im Meta-Raum untersucht werden. Diese Technik jedoch in den Kontext des Meta-Lernens oder kontinuierlichen Lernens zu transferieren, könnte ein großes Potenzial für das Intra-Life-Learning bergen.

5.3.3 Feedback-Netzwerke

Eine weitere Arbeit [LL20] erweitert das herkömmliche neuronale Netz (also das Basis-Modell) um zusätzliche *Feedback*-Verbindungen, welche den Fehler mittels gewichteter Verbindungen nach jedem Forward-Pass an die einzelnen Neuronen übermitteln. Dieser Fehler wird dann als Grundlage verwendet, um den Aktivierungszustand eines jeden Neurons im Basis-Modell neu zu berechnen. In Abbildung 5.12 sind Beispiele für solche Feedback-Netzwerke zu sehen.

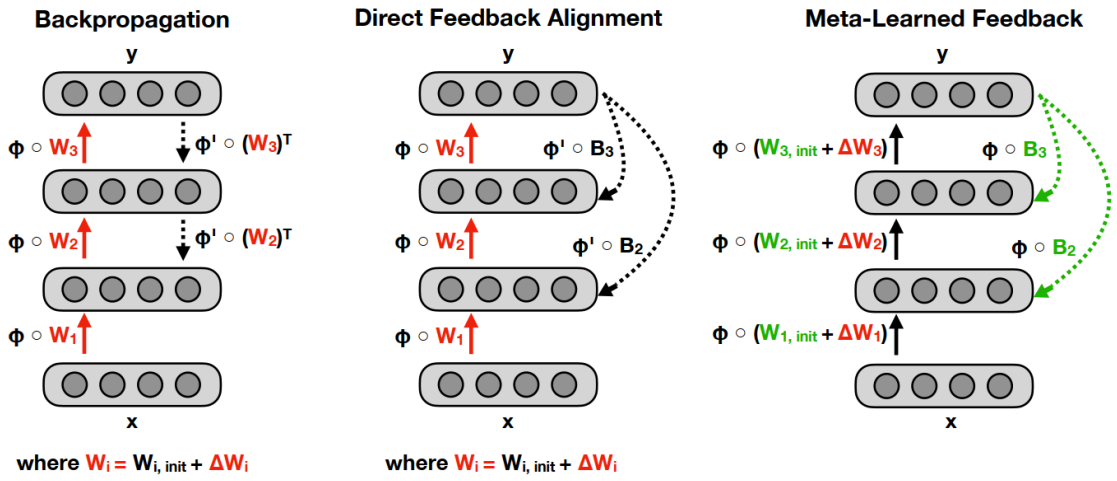


Abbildung 5.12: Die Varianten eines Feedback-Netzwerkes im Vergleich mit einem herkömmlichen neuronalen Netz (Quelle: [LL20]).

Dabei zeigt das Netzwerk auf der linken Seite ein herkömmliches neuronales Netz, welches über den Bakpropagation-Algorithmus trainiert wird. In der Mitte ist eine Variante für das sogenannte direkte Feedback zu sehen. Rechts ist schließlich das Feedback-Netzwerk zu sehen, welches durch Meta-Lernen generiert wird. Die roten Elemente repräsentieren dabei plastische Gewichte, die durch den Basis-Optimierer (z. B. Backpropagation oder Hebb-Regel) verändert werden können. Die grünen Elemente jedoch stehen für Gewichte, die durch das Meta-Lernen generiert und dementsprechend nicht durch den Basis-Optimierer verändert werden können.
Wie in Abbildung 5.12 zu sehen, werden die Gewichte der Feedback-Verbindungen gleichzeitig mit den initialen Gewichten des Basis-Modells durch den Meta-Optimierer

erlernt. Dafür wird bei diesem Algorithmus jedoch keine NE-Technik verwendet, sondern der Gradientenabstieg. Als Basis-Optimierer kommt dann eine Modifikation von Ojas-Regel aus Gleichung 5.11 zum Einsatz.

Fazit

Diese Technik der synaptischen Plastizität weist wesentliche bessere Merkmale auf, als die in Abschnitt 5.3.2 vorgestellten Algorithmen. Zunächst ist dabei festzuhalten, dass auch die Feedback-Netzwerke das Kriterium der Adaptierbarkeit erfüllen können. Dazu wurde z. B. innerhalb einer Regressionsaufgabe mehrere Male die zu erlernende Funktion verändert. Das Feedback-Netzwerk war immer in der Lage, sich mithilfe seiner Meta-gelernten Feedback-Verbindungen auf die neuen Umstände anzupassen. Dabei wird allerdings (im Gegensatz zu den Neat-basierten Algorithmen) das Wissen über vorherige Funktionen nicht vergessen, wodurch dieser Algorithmus auch das Kriterium der Stabilität im Basis-Raum erfüllen kann.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[LL20]	X	✓	X	✓	X	X	✓	X	X

Tabelle 5.9: Feedback-Netzwerke im Überblick.

Die Effizienz kann für die Feedback-Netzwerke jedoch nicht nachgewiesen werden, da es diesbezüglich keine weiteren Untersuchungen bzw. Evaluationen des Algorithmus gibt. Allerdings wurde die Generalisierbarkeit im Meta-Raum mithilfe des Omniglot- und MiniImagenet-Datensatzes nachgewiesen.
Da der Algorithmus Adam [KB15] als Meta-Optimierer verwendet, ist auch davon auszugehen, dass sich eine Parallelisierung des Algorithmus eher schwierig gestalten würde.
Abschließend ist zu dieser Technik zu sagen, dass sie viele der geforderten Kriterien erfüllen kann. Die Kombination von unterschiedlichen Gewichten/Verbindungen hat schon bei den hybriden Netzwerken aus Abschnitt 5.2.4 zu einer Vielzahl an erfüllten Kriterien geführt. Dieser Fakt könnte darauf hinweisen, dass bei diesem Ansatz ein hohes Potenzial in Bezug auf das Intra-Life-Learning vorhanden ist.

5.3.4 Hybride Modelle

Miconi, Stanley und Clune [MSC18] verfolgen eine nochmal andere Kernidee. Sie verwenden spezielle Gewichte, welche aus einer statischen und plastischen Komponente bestehen:

$$a_j(t) = \sigma(\sum_{i \in inputs} [(w_{i,j} + \alpha_{i,j}\text{Hebb}_{i,j}(t))a_i(t-1)]) \tag{5.12}$$

$$\text{Hebb}_{i,j}(t+1) = \text{Clip}(\text{Hebb}_{i,j}(t) + \eta a_i(t-1)a_j(t)) \tag{5.13}$$

Dabei werden $w_{i,j}$, $\alpha_{i,j}$ und η per Gradientenabstieg über mehrere Episoden optimiert. Die plastischen Gewichte hingegen werden innerhalb der jeweiligen Episode per Hebb-Regel gelernt. Damit wird also der Gradientenabstieg als Meta-Optimierer und das Hebb-Lernen als Basis-Optimierer verwendet. Meta- und Basis-Modell werden in diesem Fall durch das gleiche neuronale Netz repräsentiert, wobei die statischen Gewichte $w_{i,j}$ das Langzeit- und die Hebb-Gewichte $\text{Hebb}_{i,j}$ das Kurzzeitwissen modellieren. Die Funktion $\text{Clip}(x)$ stellt hierbei eine beliebige Funktion dar, welche den Wert von $\text{Hebb}_{i,j}$ auf einen Bereich $[-1, 1]$ beschränkt. Dadurch wird das Problem der numerischen Instabilität der herkömmlichen Hebb-Regel behoben.
Auch andere Arbeiten [Mic16; TTM19; TMT20] verwenden die Idee, statische mit dynamischen Komponenten zu kombinieren, um Kurzzeit- und Langzeitwissen innerhalb desselben neuronalen Netzes speichern zu können.

Fazit

Wie bei den anderen Techniken der synaptischen Plastizität, erfüllen auch diese Algorithmen das Kriterium der Adaptierbarkeit im Basis-Raum. Das liegt vor allem in der Idee begründet, schnelle aufgabenspezifische Gewichte und langsame generelle Gewichte zu verwenden.
Auch die Generalisierbarkeit im Meta-Raum ist bei allen Ansätzen gegeben. Eine Ausnahme bildet hier jedoch die Arbeit [Mic16] bei der diesbezüglich keine Evaluation durchgeführt wird. Zu der Generalisierbarkeit im Basis-Raum sei gesagt, dass die jeweiligen Ansätze diesen Aspekt nicht im herkömmlichen Sinne erfüllen. Es ist eher so, dass verschiedene Komponenten $(a_{i,j}, \eta)$ der schnellen synaptischen

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[Mic16]	X	✓	X	X	X	X	X	X	X
[MSC18]	X	✓	X	X	X	X	✓	O	X
[TTM19]	X	✓	X	✓	X	X	✓	O	X

Tabelle 5.10: Hybride plastische Modelle im Überblick.

Gewichte über mehrere Episoden meta-gelernt werden. Da die Ansätze allerdings jeweils im Meta-Raum die Generalisierbarkeit erfüllen, ist davon auszugehen, dass die meta-gelernten Komponenten der schnellen synaptischen Gewichte ebenfalls generalisieren. Damit stellen die synaptischen Gewichte eine Mischung aus aufgabenspezifischen Gewichten und generalisierten Koeffizienten dar, die das Lernen der schnellen Gewichte beeinflussen.
Der Algorithmus von Thangarasa, Taylor und Miconi [TTM19] kann zusätzlich noch das Kriterium der Stabilität im Basis-Raum erfüllen. Diesbezüglich wurde eine gesonderte Evaluation des Algorithmus vorgenommen. Das Kriterium der Parallelisierbarkeit kann jedoch von keinem der Algorithmen erfüllt werden.
Insgesamt scheint bei dieser Technik ebenfalls ein Potenzial zu bestehen, sie auf das kontinuierliche Lernen übertragen zu können. Das adaptive Lernen hingegen ist bei allen Algorithmen dieser Technik gegeben. Diese beiden Punkte machen die Technik im Zusammenhang mit dem Intra-Life-Learning ebenfalls zu einer interessanten Variante.

5.3.5 Plastizitäts-Netzwerke

Eine weitere Technik [WO19] verwendet zwei verschiedene neuronale Netze als Geno- und Phänotyp. Dabei wird der Genotyp durch ein sogenanntes *Plastizitäts-Netzwerk* repräsentiert, welches beispielhaft in Abbildung 5.13 zu sehen ist. Dieses Netzwerk bekommt als Input Gewicht und Bias der Verbindung, sowie prä- und postsynaptische Aktivierungszustände der jeweiligen Neuronen. Zusätzlich kann eine Synapsenspezifische Variable h_i verwendet werden, um die Plastizitätsfunktion zwischen den

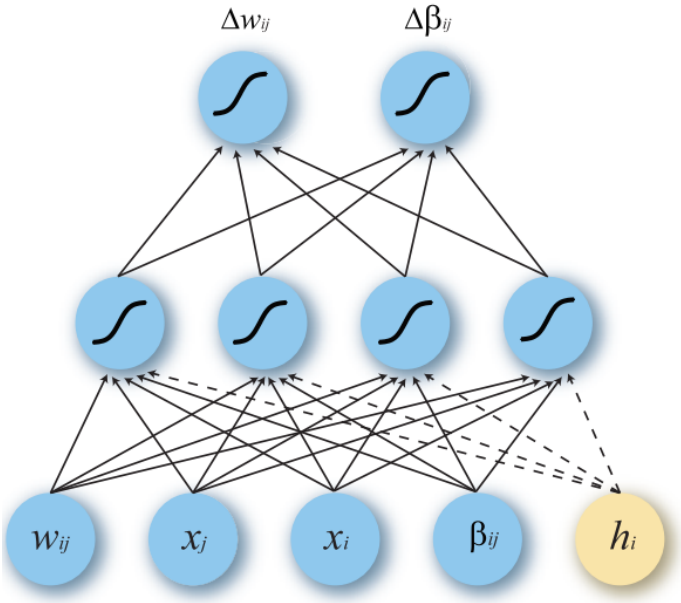


Abbildung 5.13: Das Plastizitäts-Netzwerk, welches als Meta-Modell verwendet wird (Quelle: [WO19]).

einzelnen Verbindungen variieren zu können. Als Optimierer wird ein genetischer Algorithmus verwendet. Für das Training des Phänotyps wird dann der Genotyp verwendet, welcher in Form des Plastizitäts-Netzwerkes direkt das neue Gewicht und den Bias einer Verbindung berechnet. Der Trainingsprozess des Phänotyps läuft dabei ohne eine Fehlerfunktion ab, sodass kein zusätzliches Feedback zur Verfügung steht. Für die Optimierung des Genotyps hingegen wird eine Fehler- bzw. Belohnungsfunktion verwendet, um das Training des Plastizitäts-Netzwerkes steuern zu können.

Dieser Algorithmus ähnelt dem *adaptive-HyperNEAT*-Algorithmus. Jedoch besteht der Unterschied in den gewählten Modellen für den Genotyp. Das CPPN verwendet zusätzlich noch die Koordinaten eines jeden Neurons, wodurch geometrische Informationen berücksichtigt werden können. Das Plastizitäts-Netzwerk hingegen verwendet nur die herkömmlichen Parameter, wie z. B. das aktuelle Verbindungsgewicht und die Aktivierungszustände, die auch für die Hebb-Regel verwendet werden. Dadurch ist es dem Plastizitäts-Netzwerk nicht möglich zwischen den verschiedenen Verbindungen eines neuronalen Netzes zu unterscheiden.

Fazit

In ihrer Arbeit [WO19] verfolgen Wang und Orchard das Ziel zu beweisen, dass biologisch plausibles Lernen (Hebb-Regeln) die gleichen Leistungen wie die klassischen Lernalgorithmen, wie beispielsweise Backpropagation, erzielen können. Dazu wird schließlich ein Plastizitäts-Netzwerk als Genotyp verwendet, um die Lernregeln der einzelnen Verbindungen des Phänotyps zu generieren. Allerdings folgt diese Technik nicht dem Paradigma des Meta-Lernens, weswegen hinsichtlich des Meta-Raumes keine Aussagen getroffen werden können. Allerdings lassen sich die Kriterien trotzdem noch für den Basis-Raum überprüfen.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[WO19]	?	✓	?	X	?	X	?	X	✓

Tabelle 5.11: Plastizitäts-Netzwerke im Überblick.

Dadurch, dass alle Kantengewichte des Phänotyps über diese Varianten der Hebb-Regel gesteuert werden, erfüllt die Technik das Kriterium der Adaptierbarkeit im Basis-Raum.

Alle restlichen Kriterien, die sich auf das Lernverhalten beziehen, bleiben jedoch unerfüllt. Das liegt zum einen daran, dass der Algorithmus nur auf einer einzelnen Aufgabe evaluiert wird. Zum anderen wird der Aspekt des Wissenstransfers nicht weiter untersucht, wobei auch nicht davon auszugehen ist, dass durch einen herkömmlichen evolutionären Algorithmus Wissenstransfer von einer Aufgabe zur nächsten durchgeführt werden kann.

Allerdings ist der Algorithmus gut parallelisierbar, da als Optimierer ein genetischer Algorithmus verwendet wird. Insgesamt ist zu diesem Algorithmus zu sagen, dass der Fokus eher darauf liegt eine biologisch plausible Alternative zu Backpropagation zu finden, als die konkreten Aspekte des adaptiven, kontinuierlichen oder Meta-Lernens zu adressieren. Würde die Technik jedoch auf das Paradigma des Meta-Lernens angepasst werden, könnte man eventuell auch den hier untersuchten Kriterien begegnen.

5.3.6 Fazit

In der vorliegenden Tabelle 5.12 ist eindeutig abzulesen, worin die eigentliche Stärke der synaptischen Plastizität in Bezug auf ihre Anwendbarkeit für das Intra-Life-Learning liegt. So sind alle Techniken auf die Adaptierbarkeit im Basis-Raum ausgelegt, was im Umkehrschluss bedeutet, dass das Basis-Modell auf sich verändernde Aufgaben oder Umgebungen angepasst werden kann.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
Technik 5.3.2	0/3	3/3	0/3	0/3	0/3	0/3	0/3	0/3	3/3
Technik 5.3.3	0/1	1/1	0/1	1/1	0/1	0/1	1/1	0/1	0/1
Technik 5.3.4	0/3	3/3	0/3	1/3	0/3	0/3	2/3	(2)/3	0/3
Technik 5.3.5	0/1	1/1	0/1	0/1	0/1	0/1	0/1	0/1	1/1
$\sum$	**0/8**	**8/8**	**0/8**	**2/8**	**0/8**	**0/8**	**3/8**	**(2)/8**	**4/8**

Tabelle 5.12: Die verschiedenen Techniken der synaptischen Plastizität im Überblick.

Auch fällt ins Auge, dass einige der Algorithmen das Kriterium der Stabilität im Basis-Raum erfüllen können. Auch in Bezug auf die Generalisierbarkeit im Meta- wie auch im Basis-Raum können bestimmte Algorithmen die einzelnen Kriterien erfüllen.
Besonders ist für die synaptische Plastizität die Technik der hybriden Modelle hervorzuheben. Sie erfüllt viele der geforderten Kriterien. Die Kombination aus schnellen und langsamen Gewichten stellt dabei eine interessante Idee dar, die auch im Zusammenhang mit der Konzeption eines eigenen Algorithmus für das Intra-Life-Learning Anwendung finden könnte.
Als positive Eigenschaft kann auch die Parallelisierbarkeit der einzelnen Algorithmen betrachtet werden. So zeichnet sich die Hälfte aller in diesem Unterkapitel betrachteten Algorithmen durch eine gute Parallelisierbarkeit aus, da für die Meta-Optimierung bestimmte Varianten von evolutionären Algorithmen verwendet werden.

Generell ist auch noch festzuhalten, dass es im Vergleich zu den klassischen Techniken des Meta-Lernens wenige Arbeiten gibt, welche die synaptische Plastizität mit dem Meta-Lernen kombinieren. Es wurden lediglich acht Arbeiten gefunden, die sich mit dieser Thematik auseinandersetzen, während das klassische Meta-Lernen mit 28 Arbeiten deutlich stärker vertreten ist. Das hat allerdings nicht zwangsläufig zu bedeuten, dass dieser Ansatz weniger Potenzial besitzt. Es ist eher davon auszugehen, dass auf diesem Gebiet noch viel Forschungspotenzial besteht, was es für zukünftige Arbeiten bezüglich des adaptiven oder kontinuierlichen Lernens interessant machen sollte.
Auch der Fakt, dass einige der Techniken noch nicht das Paradigma des Meta-Lernens nutzen, spricht dafür, dass in der weiteren Erforschung solcher Techniken noch viel Potenzial stecken könnte.

5.4 Neuromodulation

Die *Neuromodulation* geht im Vergleich zur synaptischen Plastizität noch einen Schritt weiter. Anstatt eine Menge von statischen Lernregeln für jede Verbindung eines neuronalen Netzes zu generieren, kann dessen synaptische Plastizität auch über die komplette Lebensspanne des Basis-Modells variiert werden. Da die Plastizität einer Verbindung kontrolliert, ob und wie sich ihr Gewicht verändern kann, kommt die Neuromodulation der Steuerung des Lernens gleich. Dazu wird ein spezielles neuromodulares Signal verwendet, um die Plastizität in bestimmten synaptischen Verbindungen erhöhen oder senken zu können.
Ein interessanter Vorteil, der sich durch die Neuromodulation ergibt, ist die Eindämmung des katastrophalen Vergessens. So kann die Plastizität z. B. nur in den Regionen eines neuronalen Netzes eingeschaltet werden (das Lernen wird für die jeweiligen Verbindungen aktiviert), die für das Bewältigen der neuen Aufgabe notwendig sind. Dadurch bleiben die Gewichte der restlichen Verbindungen (für die das Lernen ausgeschaltet ist) unberührt, sodass das in ihnen gespeicherte Wissen erhalten bleiben kann.

5.4.1 Modulare Netzwerke

Ellefsen, Mouret und Clune weisen in ihrer Arbeit [EMC15] nach, dass die Neuromodulation in Kombination mit Modularität das katastrophale Vergessen verhindern kann. Dazu wird das herkömmliche neuronale Netz um spezielle *neuromodulative* Neuronen erweitert, welche die synaptische Plastizität der Verbindungen beeinflussen können. Die Modularität im neuronalen Netz wird über zusätzliche „Kosten“ für das Hinzufügen und Verwalten einer Verbindung forciert. Als Optimierer wird bei dieser Technik ein multi-objektiver evolutionärer Algorithmus verwendet, welcher mehrere Optimierungskriterien und deren Relevanz berücksichtigen kann. Dabei ist die Minimierung der Kosten für jede Verbindung eines der Optimierungskriterien, welches Einfluss auf die Selektion nimmt. Dadurch entstehen neuronale Netze, welche ihre Funktionalitäten in einzelne Module auslagern. In Abbildung 5.14 sind Beispiele für unterschiedliche modulare Netzwerke zu sehen.

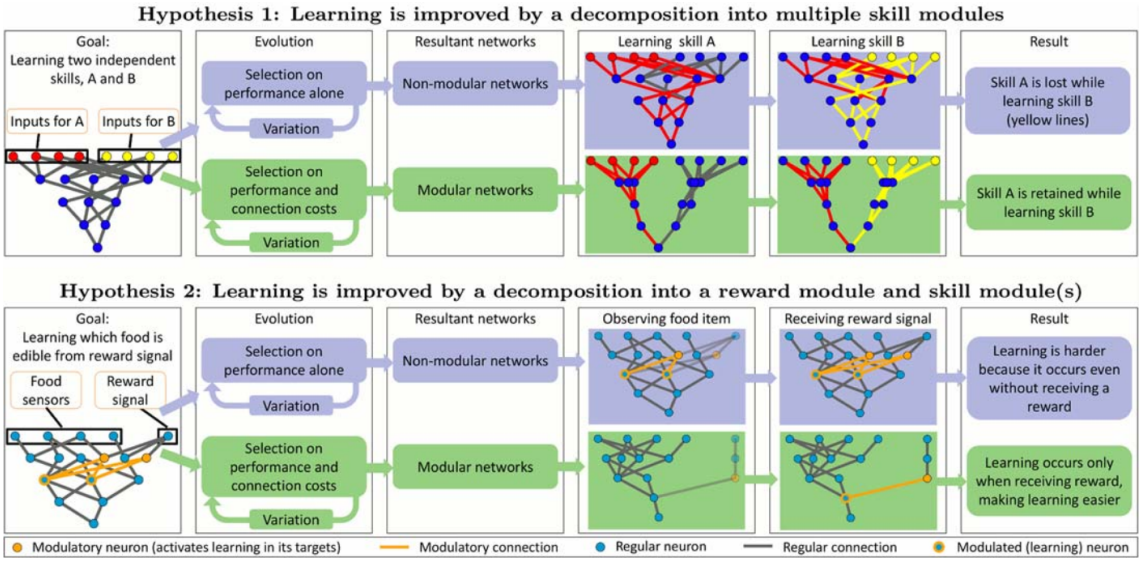


Abbildung 5.14: Die zwei Hypothesen die durch die Arbeit aufgestellt werden (Quelle: [EMC15]).

Es fällt auf, dass es sich bei einem Modul um eine *Clique*[3] im Sinne der Graphen-Theorie handelt. Es ist ebenfalls zu sehen, dass die Module untereinander nur mit einzelnen wenigen Kanten verbunden sind. In Kombination mit der Neuromodulation ermöglicht das dann, dass die synaptische Plastizität nur in jenen Regionen eines neuronalen Netzes aktiviert wird, welche für das Erlernen oder Bewältigen der jeweiligen Aufgabe zuständig sind. Die Plastizität in allen anderen Teilen des neuronalen Netzes wird dann verringert bzw. deaktiviert, um das in ihnen gespeicherte Wissen erhalten zu können.

Fazit

In Tabelle 5.13 ist zu sehen, dass die Kombination aus Modularität und einfacher Neuromodulation sehr gewinnbringend in Bezug auf die einzelnen Kriterien sein kann. Dazu sei allerdings auch erwähnt, dass die einzelnen Kriterien für relativ simple Aufgabenstellungen evaluiert werden und auch die entstandenen neuronalen Netze nicht die Komplexität von heutigen tiefen neuronalen Netzen mit mehreren Millionen Parametern aufweisen.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[EMC15]	?	✓	?	✓	?	X	?	X	✓

Tabelle 5.13: Modulare Netzwerke im Überblick.

Durch die Verwendung von Hebb-Regeln für die Aktualisierung der einzelnen Gewichte erfüllt dieser Algorithmus die Adaptierbarkeit im Basis-Raum. Denn die Hebb-Regel arbeitet unabhängig von einer Fehler- oder Belohnungsfunktion und erlaubt es somit, die Gewichte sogar während der Inferenz anzupassen. Dadurch muss keine Trainingsphase im herkömmlichen Sinne durchlaufen werden. Allerdings wird dabei das Gewicht einer jeden Verbindung durch eine statische Hebb-Regel aktualisiert, welche nur über das Modulationssignal gesteuert wird und ansonsten für jede Verbindung gleich ist. Hier könnte der Algorithmus aber mit einer komplexeren

[3] Eine Clique ist eine Menge von Knoten in einem Graphen, welche stark untereinander verbunden sind, aber nur wenige Verbindungen zu Knoten außerhalb ihrer Clique besitzen.

Variante wie z. B. dem adaptive HyperNEAT-Algorithmus oder den hybriden Netzwerken aus Unterkapitel 5.3 kombiniert werden, um für jede Verbindung spezifische Lernregeln zu generieren und somit die Performance des neuronalen Netzes weiter zu verbessern.
Da diese Technik der Neuromodulation nicht nach dem Paradigma des Meta-Lernens arbeitet, können nur Aussagen bezüglich der verschiedenen Kriterien im Basis-Raum getroffen werden. So ist das Kriterium der Stabilität für den Basis-Raum in der entsprechenden Arbeit nachgewiesen und auf die Kombination von Neuromodulation und Modularität zurückzuführen. Allerdings sei an dieser Stelle auch erwähnt, dass die verwendete Testumgebung den Agenten „nur“ mit zwei verschiedenen Szenarien konfrontiert hat. Es wäre also näher zu evaluieren, ob sich die Ergebnisse des Experiments auch auf längere Sequenzen von unterschiedlichen Aufgaben übertragen lassen.
Zu der Effizienz bzw. dem Wissenstransfer gibt es keine weiteren Hinweise oder Untersuchungen, weswegen das Kriterium im Basis-Raum als „nicht erfüllt“ gekennzeichnet wurde.
Da das Lernen im Basis-Modell nur aufgabenspezifisch abläuft, kann nicht davon ausgegangen werden, dass das darin gespeicherte Wissen generalisiert. Daher bleibt das Kriterium der Generalisierbarkeit an dieser Stelle unerfüllt.
Durch die Verwendung eines multi-objektiven evolutionären Algorithmus als Optimierer ist auch davon auszugehen, dass sich der Algorithmus gut parallelisieren lässt.
Abschließend bleibt zu sagen, dass diese Technik der Neuromodulation viele gute Ansätze miteinander vereint. Dabei besteht in der Übertragung auf das Meta-Lernen weiteres Potenzial, was im Zusammenhang mit dem Intra-Life-Learning relevant werden könnte.

5.4.2 Hybride Modelle

Der *Backpropamine*-Algorithmus [Mic+19] greift die Idee aus [MSC18] auf, welche statische und dynamische Gewichte im Basis-Modell miteinander kombiniert. Es gibt also dementsprechend kein gesondertes Meta-Modell. Das Meta-Wissen wird durch die statischen Gewichte repräsentiert, welche mit dem Backpropagation-Algorithmus

trainiert werden. Hinzu kommt bei diesem Algorithmus jedoch, dass auch die neuromodulativen Signale während des Meta-Trainings angepasst werden. Diese steuern dann wiederum die Plastizitäten der dynamischen Gewichte, die dann das Basis-Wissen darstellen und mit der Hebb-Regel trainiert werden. Dazu wird η aus Gleichung 5.13 einfach durch ein neuromodulatives Signal $M(t)$ ersetzt:

$$\text{Hebb}_{i,j}(t+1) = \text{Clip}(\text{Hebb}_{i,j}(t) + M(t)a_i(t-1)a_j(t)) \tag{5.14}$$

Der Koeffizient $M(t)$ kann dabei z. B. ein simpler skalarer Output-Wert des neuronalen Netzes sein, welcher direkt genutzt wird. Eine weitere Variante besteht darin, dass $M(t)$ für jede Verbindung einzeln berechnet wird, indem der Skalar mit einem Meta-gelernten Vektor multipliziert wird, wobei jedes Element dieses Vektors jeweils einer Verbindung $w_{i,j}$ in dem neuronalen Netz entspricht. Dadurch würde $M(t)$ für jedes $\text{Hebb}_{i,j}$ einen spezifischen Wert annehmen.

Fazit

Der Fokus der Arbeit [Mic+19] liegt nicht auf der Anwendung im Kontext des Meta- oder kontinuierlichen Lernens, sondern darauf die Idee aus der Arbeit [MSC18] für die Neuromodulation weiterzuentwickeln. Das bedeutet, dass der Backpropamine-Algorithmus den Prozess der Neuromodulation in den Backpropagation-Algorithmus einbettet. Deswegen wurde auch nur eine Evaluation in Bezug auf die Performance des Basis-Modells durchgeführt und andere Aspekte wie die Generalisierbarkeit oder Stabilität vernachlässigt.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[Mic+19]	X	✓	X	X	X	X	X	X	X

Tabelle 5.14: Neuromodulierte hybride Modelle im Überblick.

Wie aus Tabelle 5.14 zu entnehmen ist, kann Backpropamine nicht ohne zusätzlichen Aufwand für das Intra-Life-Learning verwendet werden. Jedoch stellt der Algorithmus dennoch eine interessante Variante dar, die wiederum in einen kom-

plexeren Algorithmus eingebettet werden könnte. So wäre es z. B. eine Variante für die Optimierung der statischen Komponenten des Algorithmus nicht länger den Backpropagation-Algorithmus zu verwenden, sondern ein Optimierer-Netzwerk. Das könnte die Trainingsgeschwindigkeit weiter verbessern und eventuell auch die Performance des eigentlichen Modells optimieren.

5.4.3 Neuromodulations-Netzwerke

Wang, Zheng und Orchard [WZO20] greifen ebenfalls ihre eigene Idee [WO19] aus dem Bereich der synaptischen Plastizität auf, bei der zwei verschiedene neuronale Netze für das Basis- und Meta-Modell verwendet werden. Die Verbindungen des Basis-Modells sind dabei alle plastisch, sodass dessen Gewichte in Abhängigkeit vom Input variieren können. Als Basis-Optimierer wird dann das Meta-Modell direkt verwendet. Dabei handelt es sich dann um ein Neuromodulations-Netzwerk, welches beispielhaft in Abbildung 5.15 zu sehen ist.

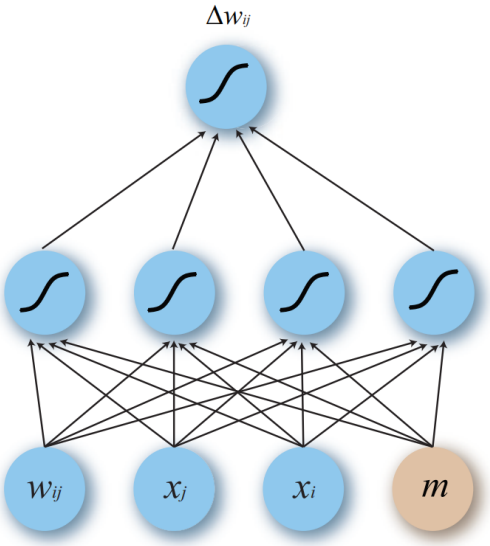


Abbildung 5.15: Das Neuromodulations-Netzwerk, welches als Meta-Modell verwendet wird (Quelle: [WZO20]).

Als Input werden für das Meta-Modell die prä- und postsynaptischen Aktivitäten x_i, x_j und das Gewicht w_{ij} der jeweiligen Verbindung verwendet. Das neuromodulative Signal m gilt als optionaler Input. Auf dieser Basis können dann die neuen

Werte der jeweiligen Gewichte des Basis-Modells berechnet werden. Wird Δw_{ij} ohne Hinzunahme von m berechnet, so ergibt sich der neue Wert einer Kante durch $\Delta w_{ij} m$. Bei Hinzunahme von m wird der Wert von Δw_{ij} direkt als neues Gewicht übernommen. Dieses Vorgehen soll dazu dienen, eine vielfältigere Interaktion zwischen Neuromodulation und synaptischer Plastizität zu ermöglichen.

Fazit

In Tabelle 5.15 ist zu sehen, dass die Plastizitäts-Netzwerke das Kriterium der Adaptierbarkeit für den Basis-Raum erfüllen. Das ist insofern keine Überraschung, da auch hier für die Basis-Optimierung Hebb-Lernen verwendet wird.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[WZO20]	X	✓	X	X	X	X	✓	X	✓

Tabelle 5.15: Neuromodulations-Netzwerke im Überblick.

Zusätzlich wird gezeigt, dass das Meta-Modell auch auf bisher ungesehene Aufgaben generalisieren kann, auf denen es nicht trainiert wurde. Dadurch ist also auch das Kriterium der Generalisierbarkeit im Meta-Raum erfüllt.
Für das Szenario des kontinuierlichen Lernens wurde der Algorithmus leider nicht weiter evaluiert. Deswegen können keinerlei Aussagen über die Kriterien der Stabilität oder Effizienz getroffen werden, weswegen sie als „nicht erfüllt“ gekennzeichnet sind.
Allerdings offenbart die Arbeit einen anderen Punkt, der im Zusammenhang mit dem Design eines eigenen Algorithmus Beachtung finden könnte. Und zwar haben die Experimente ergeben, dass die Netze, bei denen der Modulationskoeffizient m als zusätzliche Eingabe des Meta-Modells dient, wesentlich bessere Performance zeigen als die Netze, bei denen m einfach nur multiplikativ hinzugerechnet wird. Das legt nahe, dass eine komplexere Interaktion zwischen Neuromodulation und synaptischer Plastizität auch zu besserer Performance führen kann, als würde man beides einfach nur multiplikativ verknüpfen.

Die Neuromodulations-Netzwerke sind also eine passende Technik, um komplexere Formen der Interaktion zwischen Neuromodulation und synaptischer Plastizität meta-lernen zu können. Allerdings können sie nicht weiteren wichtigen Kriterien, wie z. B. dem Eindämmen des katastrophalen Vergessens begegnen. Das macht sie zwar im Zusammenhang mit dem kontinuierlichen Lernen unbrauchbar, allerdings kann diese Technik eine gute Ergänzung zu anderen Techniken darstellen, um auch das adaptive Lernen zu lernen und somit effizienter zu machen.

5.4.4 Zelluläre Neuromodulation

Bei den bisherigen Ansätzen wurde über die Neuromodulation jeweils direkt die Lernrate einer Verbindung gesteuert. Der ANML-Algorithmus [Bea+20] hingegen verwendet eine gänzlich andere Idee. Dabei steuert die Neuromodulation die jeweiligen Aktivierungen der Neuronen. Wie in Abbildung 5.16 zu sehen, wird auch bei diesem Algorithmus weiterhin zwischen Meta- (θ^{NM}) und Basis-Modell (θ^{P}) unterschieden.

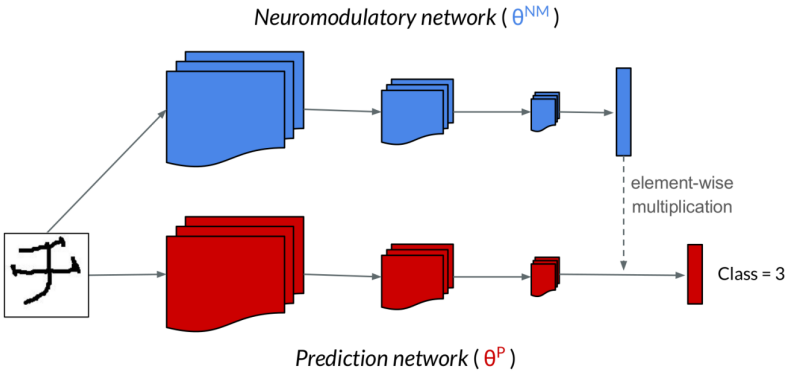


Abbildung 5.16: Die Architektur des ANML-Algorithmus (Quelle: [Bea+20]).

Dabei wird der finale Output der letzten Schicht von θ^{NM} mit dem Input der letzten Schicht von θ^{P} elementweise multipliziert, wobei beide Vektoren dieselbe Größe besitzen. Dadurch werden die Aktivierungszustände der Neuronen der letzten Schicht von θ^{P} durch das Meta-Modell gesteuert. Der Grundgedanke dabei ist, dass dadurch die Interferenz im Forward- wie auch im Backward-Pass minimiert werden soll. Denn

die Veränderung eines Gewichtes im Backward-Pass hängt von den Aktivierungszuständen der Neuronen ab, egal ob Backpropagation oder Hebb-Regeln verwendet werden. Haben bestimmte Neuronen beispielsweise einen Aktivierungszustand von 0, so verändern sich die Gewichte von deren eingehenden Verbindungen während des Backward-Pass nicht. Das resultiert darin, dass das jeweilige Wissen in diesen Verbindungen unverändert bleibt. Ebenso ist dieses Vorgehen beim Forward-Pass von Vorteil, da dadurch während der Ausführung das Wissen bezüglich einer Aufgabe nicht mit dem Wissen einer anderen vermischt wird (vgl. [Bea+20]).
Vecoven u. a. [Vec+19] gehen noch einen Schritt weiter und verwenden eine spezielle Aktivierungsfunktion für jedes Neuron des Basis-Modells (siehe Abbildung 5.17B).

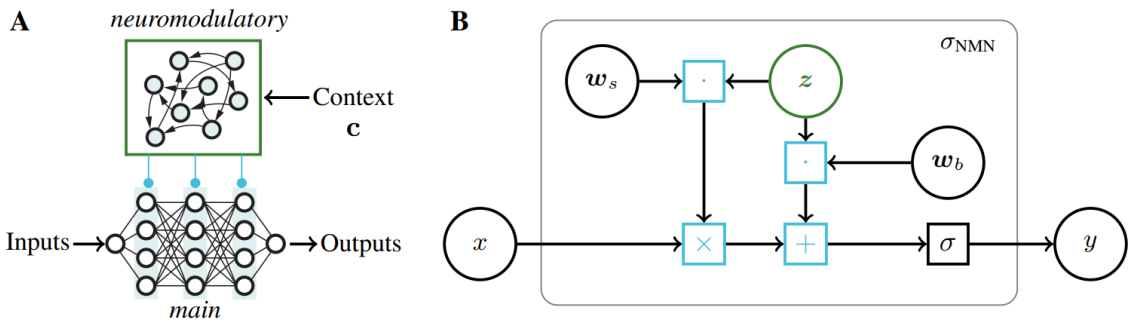


Abbildung 5.17: (A) Neuromodulations-Netzwerk, (B) Aktivierungsfunktion eines Neurons des Basis-Modells (Quelle: [Vec+19]).

Das Neuromodulations-Netzwerk bzw. Meta-Modell bekommt als Input hier einen Kontext-Vektor $\mathbf{c}$. Auf dieser Basis berechnet das Netzwerk dann einen vektoriellen Parameter $\mathbf{z} \in \mathbb{R}^k$ für jedes Neuron (siehe Abbildung 5.17A). Dieser Parameter $\mathbf{z}$ wird dann verwendet, um den Output eines jeden Neurons zu beeinflussen. Dadurch ergibt sich dann folgender Ausdruck zur Berechnung des Outputs eines Neurons:

$$\sigma_{NMN}(x, \mathbf{z}, \mathbf{w_s}, \mathbf{w_b}) = \sigma(\mathbf{z}^T(x\mathbf{w_s} + \mathbf{w_b})) \tag{5.15}$$

Dabei sind $\mathbf{w_s}, \mathbf{w_b} \in \mathbb{R}^k$ jeweils vektorielle Parameter eines Neurons. Der Parameter k ist als Hyperparameter definiert und muss eigenhändig vorgegeben werden. Bei diesem Algorithmus werden also nicht nur die Aktivierungszustände der Neuronen

der letzten Schicht, sondern von allen Neuronen im Basis-Modell per Neuromodulation gesteuert. Der Mehrwert dieses Algorithmus besteht somit in der Veränderung der Aktivierungsfunktionen, wodurch die Topologie bzw. der Typ des Meta-Modells wie auch dessen Input undefiniert bleiben und somit dem Entwickler überlassen werden.

Fazit

Die Algorithmen der zellulären Neuromodulation gehen im Vergleich zu den anderen Techniken noch einen Schritt weiter und modellieren nicht nur die Plastizität, sondern auch die Aktivierungszustände der einzelnen Neuronen. So ist es wenig überraschend, dass auch diese beiden Algorithmen das Kriterium der Adaptierbarkeit im Basis-Raum erfüllen.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[Bea+20]	X	✓	X	✓	X	X	✓	X	X
[Vec+19]	X	✓	X	X	X	X	✓	X	X

Tabelle 5.16: Algorithmen der zellulären Neuromodulation im Überblick.

Beaulieu u. a. [Bea+20] zeigen allerdings auch, dass ihr Algorithmus zusätzlich die Kriterien der Generalisierbarkeit im Meta-Raum und Stabilität im Basis-Raum erfüllen kann. Für beide Kriterien wurden eigene Experimente durchgeführt. Deren Daten ergeben, dass der Algorithmus sowohl das Meta-Wissen auf neue ungesehene Aufgaben generalisieren kann, als auch das katastrophale Vergessen des Basis-Wissens beim kontinuierlichen Lernen eindämmt.
Vecoven u. a. evaluieren in ihrer Arbeit [Vec+19] nur für die Generalisierbarkeit. Dabei wird nachgewiesen, dass das im Meta-Training generierte Wissen auf unterschiedliche Aufgaben anwendbar ist und sich die Basis-Optimierung dadurch schneller gestaltet, als wenn ein klassisches rekurrentes Netz als Meta-Modell verwendet wird.
Abschließend kann festgehalten werden, dass die Idee die Aktivierungszustände der einzelnen Neuronen anstatt der Lernraten zu steuern, einen weiteren Vorteil bei der

Neuromodulation bringen kann. Dieser Punkt ist im Nachhinein betrachtet auch recht naheliegend, da sich die Veränderung der Gewichte immer (egal ob Backpropagation oder Hebb-Regel) durch die Aktivierungszustände der jeweiligen Neuronen ergibt. Außerdem kann dadurch nicht nur die Interferenz während des Lernens, sondern auch während der Ausführung eingeschränkt werden. Das macht diese Technik in Kombination mit anderen Techniken des Meta-Lernens zu einem interessanten Ansatz in Bezug auf das Intra-Life-Learning.

5.4.5 Diffusions-basierte Neuromodulation

Was die bisher vorgestellten Ansätze alle gemein haben ist, dass sie die synaptische Plastizität für jede Verbindung bzw. jedes Neuron einzeln steuern. Dadurch entsteht das Problem, dass viele Parameter durch den Meta-Optimierer optimiert werden müssen. Somit ist es schwierig, mit diesen Ansätzen tiefe neuronale Netze zu entwickeln, welche Millionen von Verbindungen und Neuronen besitzen.

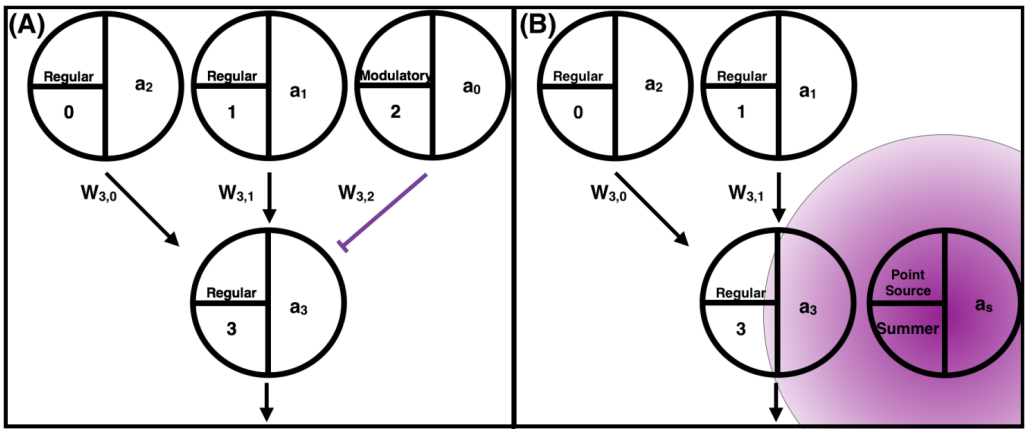


Abbildung 5.18: Ein diffundierendes Neuron in einem neuronalen Netz (Quelle: [VC17]).

Biologische Neuronen besitzen jedoch eine breite Auswahl an Kommunikationsmechanismen (vgl. [VC17]). Aus Kapitel 2 ist bekannt, dass sich biologische Neuronen über die an den Axonterminalen befindlichen Synapsen Signale zusenden können. Diese Art der Kommunikation wird auch als *Wire-Transmission* (zu deutsch *Drahtübertragung*) bezeichnet. Eine weitere Variante der Kommunikation besteht in der

sogenannten *Volume-Transmission*. Dabei setzt das Neuron Signalchemikalien wie Neutransmitter (welche das Lernen beeinflussen) frei, die Signale an andere Neuronen in einer bestimmten Umgebung diffundieren bzw. verteilen (vgl. [VC17]).
Auf dieser Basis führen Velez und Clune [VC17] die Technik der *Diffusions-basierten* Neuromodulation ein. Der Kerngedanke dabei ist, dass bestimmte Neuronen ein neuromodulatives Signal diffundieren, welches nur durch Neuronen in einem bestimmten Umkreis wahrgenommen werden kann. Abbildung 5.18(B) zeigt ein Beispiel für ein solches Neuron. In Abbildung 5.18(A) ist hingegen ein herkömmliches neuromodulatives Neuron zu sehen, das seine Signale über gerichtete Verbindungen an andere Neuronen weitergibt.
Bei dieser Technik sind jedem Neuron des Basis-Modells Koordinaten zugewiesen. Über eine Distanzfunktion können dann die Entfernungen zwischen den einzelnen Neuronen ermittelt werden. Das neue Gewicht einer Verbindung wird dann über folgende Formel berechnet:

$$\Delta w_{i,j} = \eta m_i a_i a_j \tag{5.16}$$

$$m_i = \phi(\sum_{k \in C_m} a_k g(d_{ik})) \tag{5.17}$$

$$g(x) = \begin{cases} \frac{e^{-2}}{\sqrt{2\sigma^2\phi}} e^{\frac{-x^2}{2\sigma^2}} & \text{wenn } x \leq 1,5 \\ 0 & \text{sonst} \end{cases} \tag{5.18}$$

Dabei ist m_i das neuromodulative Signal, a_k das diffundierende Neuron und $g(x)$ die Distanzfunktion. Zu erwähnen ist hier, dass theoretisch auch andere Funktionen zum Berechnen der Distanz verwendet werden können. Die in Gleichung 5.18 definierte Funktion soll als Beispiel dienen und wurde direkt aus der Arbeit [VC17] übernommen.
Als Optimierer wird der PNSGA-Algorithmus [CML13] verwendet. Dabei handelt es sich um einen multi-objektiven evolutionären Algorithmus, welcher eine Weiterentwicklung von NSGA-II [Deb+02] ist und beim Selektionsprozess die Minimierung der Kosten für Verbindungen berücksichtigt. Dadurch wird Modularität in den generierten neuronalen Netzen forciert . Durch diese Modularität lässt sich die Plastizität in den verschiedenen Teilen noch besser durch die Neuromodulation regulieren.

Fazit

Die diffusions-basierte Neuromodulation forciert das neuronale Netz dazu, sein Wissen bezüglich unterschiedlicher Aufgaben in voneinander separierten Modulen zu speichern. Dazu werden die Verbindungsmuster über einen evolutionären Algorithmus gelernt. Die jeweiligen Gewichte werden dann mithilfe einer Hebb-Regel generiert, wodurch der Algorithmus das Kriterium der Adaptierbarkeit erfüllen kann. Damit greift es im Grunde genommen die Idee der modularen Netzwerke auf. Insofern stellt diese Arbeit eine Weiterentwicklung der in [EMC15] vorgestellten Idee dar.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
[VC17]	?	✓	?	✓	?	X	?	X	✓

Tabelle 5.17: Algorithmen der diffusions-basierten Neuromodulation im Überblick.

Somit ist es nicht verwunderlich, dass auch dieser Algorithmus nicht nach dem Paradigma des Meta-Lernens abläuft. Dadurch kann über die einzelnen Kriterien im Meta-Raum keine Aussage getroffen werden. Dennoch ist der Algorithmus in der Lage das Kriterium der Stabilität zu erfüllen. Durch die Platzierung von diffundierenden Neuronen im Basis-Modell kann das katastrophale Vergessen eingedämmt werden. Diese Neuronen regen nämlich das Lernen in den umliegenden Synapsen nur an, wenn sie selber angeregt werden, indem bestimmte Voraussetzungen bzw. Eigenschaften in der jeweiligen Umgebung wahrgenommen werden.
Jedoch kann der Algorithmus den Faktor der Generalisierbarkeit nicht erfüllen, da kein generelles Wissen über mehrere Aufgaben hinweg generiert wird. Stattdessen wird das jeweilige Wissen aufgabenspezifisch gelernt und steht somit im Widerspruch zur Generalisierbarkeit und dem Meta-Lernen im Allgemeinen.
Dennoch wäre es interessant zu untersuchen, welche Synergien sich ergeben würden, wenn man diese Technik der Neuromodulation mit dem Meta-Lernen kombiniert. In diese Richtung könnte also durchaus Forschungspotenzial vorhanden sein.

5.4.6 Fazit

In Tabelle 5.18 ist deutlich zu sehen, dass alle Algorithmen das Kriterium der Adaptierbarkeit erfüllen. Dies liegt darin begründet, dass die Neuromodulation eine weitere Abstraktionsebene über der synaptischen Plastizität darstellt, indem sie diese in den jeweiligen neuronalen Netzen steuert. Deswegen lassen sich diese beiden Gebiete auch nicht komplett unabhängig voneinander betrachten, was darin resultiert, dass viele Algorithmen der Neuromodulation ihren Ursprung in Algorithmen der synaptischen Plastizität haben und somit eine Weiterentwicklung dieser sind.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
Technik 5.4.1	0/1	1/1	0/1	1/1	0/1	0/1	0/1	0/1	1/1
Technik 5.4.2	0/1	1/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Technik 5.4.3	0/1	1/1	0/1	0/1	0/1	0/1	1/1	0/1	1/1
Technik 5.4.4	0/2	2/2	0/2	1/2	0/2	0/2	2/2	0/2	0/2
Technik 5.4.5	0/1	1/1	0/1	1/1	0/1	0/1	0/1	0/1	1/1
$\sum$	**0/6**	**6/6**	**0/6**	**3/6**	**0/6**	**0/6**	**3/6**	**0/6**	**3/6**

Tabelle 5.18: Algorithmen der Neuromodulation im Vergleich.

Nicht alle Ansätze der Neuromodulation sind auf das Vermeiden von katastrophalem Vergessen und somit auf das Kriterium der Stabilität ausgelegt. Einige der Algorithmen [Vec+19; WZO20] haben etwa die Zielstellung, das adaptive Lernen eines neuronalen Netzes zu verbessern und wurden somit nicht für das kontinuierliche Lernen konzipiert.
Auch das Kriterium der Generalisierbarkeit kann nicht von allen Techniken erfüllt werden. Das liegt entweder daran, dass die entsprechenden Arbeiten den Algorithmus nicht dahingehend evaluiert haben oder der jeweilige Algorithmus nicht auf das Meta-Lernen ausgelegt ist.
Allgemein stellt die Neuromodulation einen guten Ansatz dar, um dem Problem des katastrophalen Vergessens begegnen zu können. Auch das adaptive Lernen wird durch die einzelnen Techniken gut abgedeckt, allerdings nur, weil die Neuromodulation die synaptische Plastizität steuert und diese wiederum das adaptive Lernen

ermöglicht. Alles in allem ist die Neuromodulation also eher mit dem Problem des katastrophalen Vergessens in Verbindung zu bringen. Sie stellt eine biologisch plausible Möglichkeit dar, diesem Problem in neuronalen Netzen begegnen zu können. Somit kann sie auch eine wichtige Rolle im Zusammenhang mit dem Intra-Life-Learning einnehmen.

5.5 Zusammenfassung

Diese Kapitel hat mit dem Intra-Life-Learning, neben dem klassischen maschinellen Lernen und der Neuroevolution, ein weiteres Lernparadigma eingeführt, welches sich aus unterschiedlichen Lernszenarien zusammensetzt. In diesem Zusammenhang wurden unterschiedliche Ansätze daraufhin untersucht, wie sie sich für das Intra-Life-Learning eigenen und welche Stärken und Schwächen sie jeweils mitbringen. In Tabelle 5.19 ist die Zusammenfassung über alle Ansätze (Meta-Lernen, synaptische Plastizität, Neuromodulation) zu sehen.

	Adaptives Lernen		Kontinuierliches Lernen				Meta-Lernen		**Parallelisierbarkeit**
	Adaptierbarkeit		**Stabilität**		**Effizienz**		**Generalisierbarkeit**		
	Meta	Basis	Meta	Basis	Meta	Basis	Meta	Basis	
Ansatz 5.2	0/23	(1)/23	(2)/23	(1)/23	2/23	0/23	19/23	3/23	4/23
Ansatz 5.3	0/8	8/8	0/8	2/8	0/8	0/8	3/8	(2)/8	4/8
Ansatz 5.4	0/6	6/6	0/6	3/6	0/6	0/6	3/6	0/6	3/6
$\sum$	**0/37**	**15/37**	**(2)/37**	**6/37**	**2/37**	**0/37**	**25/37**	**5/37**	**11/37**

Tabelle 5.19: Ansätze für das Intra-Life-Learning im Überblick.

Was aus der kompletten Evaluation abzuleiten ist und sich auch in der Tabelle abzeichnet, ist, dass sich die unterschiedlichen Ansätze jeweils für ein bestimmtes Lernszenario des Intra-Life-Learning eignen. Die verschiedenen Techniken des Meta-Lernens eignen sich für das Multi-Task- bzw. Few-Shot-Lernen. Allerdings liegt der größte Beitrag nicht in den einzelnen Algorithmen des Meta-Lernens, sondern eher in dem Trainingsparadigma, welches auch gut auf andere Ansätze übertragbar ist und somit die Möglichkeit für eine Vielzahl an neuen Algorithmen eröffnet.

Auch der Fakt, dass kein Einziger der in diesem Kapitel untersuchten Algorithmen auf eine parallele Verarbeitung ausgelegt ist, stellt ein großes Potenzial für zukünftige Arbeiten in diesem Bereich dar. So ist aus Tabelle 5.19 ersichtlich, dass einige der Algorithmen durch ihren Aufbau durchaus für die parallele Verarbeitung geeignet sind. Somit wäre eine interessante Fragestellung, was noch alles möglich ist, wenn man diese Algorithmen auf heutige Rechenressourcen von Computer-Clustern anpassen würde.
Es ist ebenfalls in obiger Tabelle zu erkennen, dass die Ansätze der synaptischen Plastizität und Neuromodulation jeweils auf das adaptive und kontinuierliche Lernen ausgelegt sind. Vor allem die Neuromodulation stellt einen vielversprechenden Ansatz in Bezug auf das kontinuierliche Lernen und seine verschiedenen Facetten dar, welcher allerdings bisher noch nicht allzu gut erforscht wurde. Das ist zum einen an der Anzahl der Arbeiten in diesem Bereich zu sehen und zum anderen daran, dass dieser Ansatz bisher kaum in Kombination mit anderen Ansätzen des Lernens in neuronalen Netzen kombiniert wurde.
Für das Intra-Life-Learning als Lernparadigma kann das Fazit gezogen werden, dass es bisher keinen Algorithmus gibt, der den wichtigsten Anforderungen begegnen kann. Bei der Sichtung der Literatur war auffällig, dass sich die Algorithmen immer bestimmte Teilprobleme herausgreifen und diese versuchen zu lösen. Allerdings konnte keine Arbeit gefunden werden, welche versucht, die verschiedenen Lernszenarien miteinander in Beziehung zu setzen und in einen Zusammenhang zu bringen. Das nächste Kapitel soll genau diesem Punkt begegnen. Dabei wird ein eigener Algorithmus vorgestellt, welcher die Szenarien des adaptiven und kontinuierlichen Lernens in Beziehung mit dem Meta-Lernen setzt und unterschiedliche Techniken aus diesen Gebieten in einem Algorithmus vereint.

6 NeRO

Das vorherige Kapitel 5 hat aufgezeigt, dass es aktuell keine Technik gibt, die allen Anforderungen des Intra-Life-Learning begegnen kann. Dabei ist allerdings auch festzuhalten, dass das Intra-Life-Learning mehrere Paradigmen des maschinellen Lernens in sich vereint. Das spiegelt sich auch in der Erkenntnis wider, dass die in Kapitel 5 untersuchten Ansätze nur für bestimmte Teilprobleme des Intra-Life-Learnings geeignet sind.
Daher wird in diesem Kapitel mit NeRO (**Ne**uromodulated **R**euse of evolved **O**ptimizers) ein eigener Algorithmus für das Intra-Life-Learning vorgestellt. Dazu werden zunächst im folgenden Unterkapitel 6.1 die Anforderungen definiert, die bei der Konzeption von NeRO berücksichtigt werden sollten. Anschließend wird in Unterkapitel 6.2 thematisiert, welche Erkenntnisse aus Kapitel 5 genutzt wurden. Zusätzlich dazu werden auch zwei neue Ansätze vorgestellt. Das Unterkapitel 6.3 erläutert dann den Aufbau und die Funktionsweise von NeRO. Darauf aufbauend wird das Unterkapitel 6.4 die Parallelisierung des Algorithmus genauer erläutern. Abschließend fasst das letzte Unterkapitel 6.5 den Inhalt dieses Kapitels zusammen.

6.1 Anforderungsanalyse

In Unterkapitel 5.1 wurde schon die Kernidee des Intra-Life-Learnings vorgestellt und aufgezeigt, welche Lernszenarien in diesem Kontext ihre Anwendung finden. Daher ist bekannt, dass sich das Intra-Life-Learning grundsätzlich aus dem adaptiven und kontinuierlichen Lernen zusammensetzt. Wenn man beide Szenarien zusammennimmt und zusätzlich den Aspekt der Parallelisierbarkeit des Algorithmus berücksichtigt, lässt sich das Intra-Life-Learning auf folgende Anforderungen eingrenzen:

1. **Evolutionäre Optimierung:** der Algorithmus sollte möglichst auf evolutionären Algorithmen beruhen. Aus Abschnitt 4.1.2 ist bekannt, dass diese sich durch eine gute Parallelisierbarkeit auszeichnen, im Gegensatz zu herkömmlichen Lernverfahren wie Backpropagation.

2. **Wissenstransfer (vorwärts/rückwärts):** der Algorithmus soll es ermöglichen, bisher gesammeltes Wissen auf neue Aufgaben anwenden zu können oder neues Wissen zur besseren Lösung alter Aufgaben zu verwenden.

3. **Widerstand gegen katastrophales Vergessen**: beim Lernen jeglicher Art soll katastrophale Interferenz vermieden werden. Das bedeutet, dass sich das Wissen bezüglich unterschiedlicher Aufgaben nicht gegenseitig verdrängen bzw. überschreiben soll.

4. **Kein direkter Zugriff auf frühere Erfahrungen:** der Algorithmus kann die Lernumgebung bzw. Umwelt nicht zurückspulen und besitzt somit keinen direkten Zugriff auf vorherige Zustände der Welt.

5. **Schnelligkeit:** der Algorithmus soll dazu in der Lage sein, aus möglichst wenigen Trainingsbeispielen zu lernen, damit er auch auf Aufgaben anwendbar ist, bei denen nur wenige Trainingsdaten zur Verfügung stehen.

Es ist noch zu erwähnen, dass sich einige der Anforderungen gegenseitig positiv bedingen können. So kann z. B. der Wissenstransfer förderlich in Bezug auf die Schnelligkeit sein, wenn die neu zu erlernende Aufgabe einer älteren in ihrer Struktur gleicht. Andersherum ist es allerdings ebenso möglich, dass sich bestimmte Anforderungen gegenseitig hemmen. Verwendet man altes Wissen zur Lösung einer neuen Aufgabe und passt dieses dann entsprechend an, kann es zur Interferenz kommen, was entgegen der Anforderung 2 wäre.
Die oben definierten Anforderungen stellen somit das Fundament für den vorgestellten Algorithmus dar. Bei der Konzeption wurde versucht, für alle Anforderungen eine entsprechende Lösung zu finden.

6.2 Ansätze

Dieses Unterkapitel wird erläutern, auf welchen Ansätzen, Effekten oder auch Theorien NeRO beruht. Dazu werden in Abschnitt 6.2.1 die Erkenntnisse aus Kapitel 5 aufgegriffen und begründet, warum welche Ansätze bzw. Techniken verwendet werden. Abschnitt 6.2.2 setzt sich näher mit dem sogenannten Baldwin-Effekt auseinander und zeigt auf, wie sich dieser im Design des Algorithmus niederschlägt. In Abschnitt 6.2.3 wird schließlich die Complementary Learning Systems (CLS)-Theorie erläutert. Dabei werden auch bestimmte Begriffe eingeführt, die für das weitere Verständnis des Kapitels grundlegend sind.

6.2.1 Genutzte Erkenntnisse

In Kapitel 5 wurde gezeigt, dass sich die untersuchten Ansätze des (klassischen) Meta-Lernens, der synaptischen Plastizität und der Neuromodulation für jeweils unterschiedliche Aspekte des Intra-Life-Learnings eignen. Dabei kann jedoch kein einziger Ansatz, geschweige denn Algorithmus alle Anforderungen in diesem Zusammenhang erfüllen. Allerdings wurde in Kapitel 5 auch für die jeweiligen Techniken aufgezeigt, an welchen Stellen durchaus noch Potenzial für eine Weiterentwicklung bzw. weitere Forschungsarbeit besteht.
In diesem Zusammenhang wurde in Unterkapitel 5.1 die Idee eingeführt, das Meta-Lernen zu verwenden, um automatisch einen geeigneten Algorithmus für das Intra-Life-Learning zu generieren. Dieser generelle Ansatz wird auch bei NeRO verwendet. Anstatt also Algorithmen von Hand zu entwickeln, welche die unterschiedlichen Lernszenarien des Intra-Life-Learnings bedienen können, wird auch hier die Idee verfolgt, diese einzelnen Algorithmen automatisiert zu erlernen.
Daher wird bei NeRO ebenfalls zwischen Meta- und Basis-Modell unterschieden werden. Allerdings wird das Trainingsparadigma des Meta-Lernens auf das kontinuierliche Lernen übertragen. Der Kerngedanke richtet sich dabei nach dem Prinzip, dass das jeweilige Modell für den Trainingsdatensatz der jeweiligen Aufgabe optimiert wird, das Testen dieses Modells jedoch auf einem speziellen Meta-Datensatz erfolgt. Dieser Datensatz setzt sich aus den Testdatensatz der aktuellen Aufgabe und zufälligen Daten bezüglich älterer Aufgaben zusammen. Der aus dem Meta-

Datensatz resultierende Fehler (fortlaufend als Meta-Fehler bezeichnet) gibt einen Aufschluss darüber, ob die Optimierung bezüglich der neu zu erlernenden Aufgabe das alte, bisher gesammelte Wissen überschrieben hat oder nicht. Wie genau das allerdings durch den Algorithmus bewerkstelligt werden kann, wird Inhalt der folgenden Unterkapitel sein.
Im Zusammenhang mit Anforderung 5 (Schnelligkeit) haben sich in Kapitel 5 vor allem die unterschiedlichen Techniken des (klassischen) Meta-Lernens hervorgetan. So wurde eine Vielzahl dieser Techniken für das Few-Shot-Lernen konzipiert, welches sich durch ein effizientes Lernen im Zusammenhang mit der Anzahl der benötigten Trainingsbeispiele auszeichnet. Daher wurde sich bei der Konzeption von NeRO dafür entschieden, die Technik der Optimierer-Modelle zu verwenden. Die Gründe dafür, warum genau diese Technik anstatt anderer wie z. B. Parameter-Initialisierung oder Fehler-Modelle verwendet wird, werden jedoch Bestandteil eines späteren Unterkapitels sein.
Im Kontext von Anforderung 3 (katastrophales Vergessen) werden jeweils eine Technik der Neuromodulation und der synaptischen Plastizität miteinander kombiniert. Dabei richtet sich die Kernidee jedoch nach dem Paradigma der Neuromodulation, da das Ziel darin besteht, die synaptische Plastizität zu regulieren. Ferner handelt es sich bei den Techniken der Neuromodulation oftmals um Weiterentwicklungen von Algorithmen, die ihren Ursprung in der synaptischen Plastizität haben. Daher ist es nur sinnvoll, auf den Ansatz der Neuromodulation, statt der synaptischen Plastizität, zu setzen. An dieser Stelle wird nicht weiter erläutert, welche Techniken exakt verwendet werden. Diese Erklärung wird ebenfalls Bestandteil eines der folgenden Unterkapitel sein.

6.2.2 Baldwin-Effekt

Der Baldwin-Effekt [Bal96] beschreibt, wie Lernen den Prozess der Evolution beeinflussen kann. Schon Hinton und Plaut wiesen 1987 in ihrer Arbeit [HP87] nach, dass das Lernen innerhalb der Lebensspanne eines Organismus den Verlauf der Evolution positiv bedingt. In einer Computersimulation nutzten sie einen genetischen Algorithmus, um eine Population zu generieren, an der anschließend gezeigt wird, dass Lernen die Evolution in Szenarien unterstützen kann, in denen eine evolutionäre Su-

che allein ineffektiv ist. Folglich findet ein evolutionäres Verfahren bessere Lösungen für manche Probleme, wenn die einzelnen Organismen über eigene Lernmechanismen verfügen, statt ihr Wissen durch die Evolution vorgegeben zu bekommen.
In [Fer+18] wird der Baldwin-Effekt im Zusammenhang mit dem Meta-Lernen untersucht. Dabei wird ein genetischer Algorithmus (Evolution) in Kombination mit dem Gradientenabstieg (Lernen) genutzt, um zu zeigen, dass Algorithmen, die auf dem Baldwin-Effekt beruhen, konkurrenzfähig zu rein Gradienten-basierten Verfahren, wie z. B. MAML, und rein evolutionären Algorithmen sind.
Im Umkehrschluss bedeutet dies, dass das Wissen bezüglich einer Aufgabe oder mehrerer Aufgaben nicht direkt durch einen evolutionären Algorithmus generiert werden sollte, sondern durch einen Lernmechanismus, der wiederum aus der (simulierten) Evolution über mehrere Generationen hervorgeht.
In der Arbeit von Soltoggio, Stanley und Risi [SSR18] werden in diesem Zusammenhang auch unterschiedliche Dimensionen eingeführt, welche die unterschiedlichen Suchräume der einzelnen Optimierungsverfahren repräsentieren:

- **Phylogenetisch:** diese Dimension bezeichnet den Suchraum, in dem die Evolution arbeitet.
- **Ontogenetisch:** in dieser Dimension findet die Entwicklung des jeweiligen Organismus statt. Dabei bezeichnet der Begriff Entwicklung in dem hier gewählten Kontext von neuronalen Netzen jedoch nicht den Prozess der Morphologie- oder die Topologie-Entwicklung eines Organismus, sondern die Generierung bzw. das Erlernen von Wissen über mehrere Episoden bzw. Aufgaben hinweg.
- **Epigenetisch:** in dieser dritten Dimension findet das Lernen innerhalb einer Aufgabe bzw. Episode statt.

Die hier gewählten Definitionen der sogenannten POE-Dimensionen sind an die Arbeiten [Bal96] und [SSR18] angelehnt, wurden jedoch auf das Szenario des Intra-Life-Learning angepasst. Diese Dimensionen werden auch im weiteren Verlauf des Kapitels verwendet werden, um die unterschiedlichen Suchräume der einzelnen Optimierungsprozesse besser voneinander abgrenzen zu können.

6.2.3 Complementary Learning Systems

Bei der CLS-Theorie [MMO95] wird mit dem Neokortex und dem Hippocampus zwischen verschiedenen neuronalen Systemen unterschieden. Dabei besitzen diese Systeme unterschiedliche Lernmechanismen. So wird der Neokortex für das Lernen und Archivieren von Langzeitwissen verwendet, wodurch es das Langzeitgedächtnis repräsentiert. Der Hippocampus hingegen wird für das (schnelle) Lernen von aufgabenspezifischen Wissen benutzt und zeichnet sich dadurch aus, dass das erlernte Wissen nur für eine kurze Zeit „gespeichert“ wird. Somit stellt es das Kurzzeitgedächtnis dar. Der Kerngedanke der CLS-Theorie ist, dass sich diese beiden Systeme komplementär bzw. ergänzend zueinander verhalten.
In Abbildung 6.1 ist ein Beispiel für den (approximierten) Aufbau eines Gehirns zu sehen. Der von gestrichelten Linien umgebene Bereich stellt dabei den medialen Temporallappen (MTL) mit dem Hippocampus (dunkelgrau) und den umgebenen Regionen des MTL (hellgrau) dar.

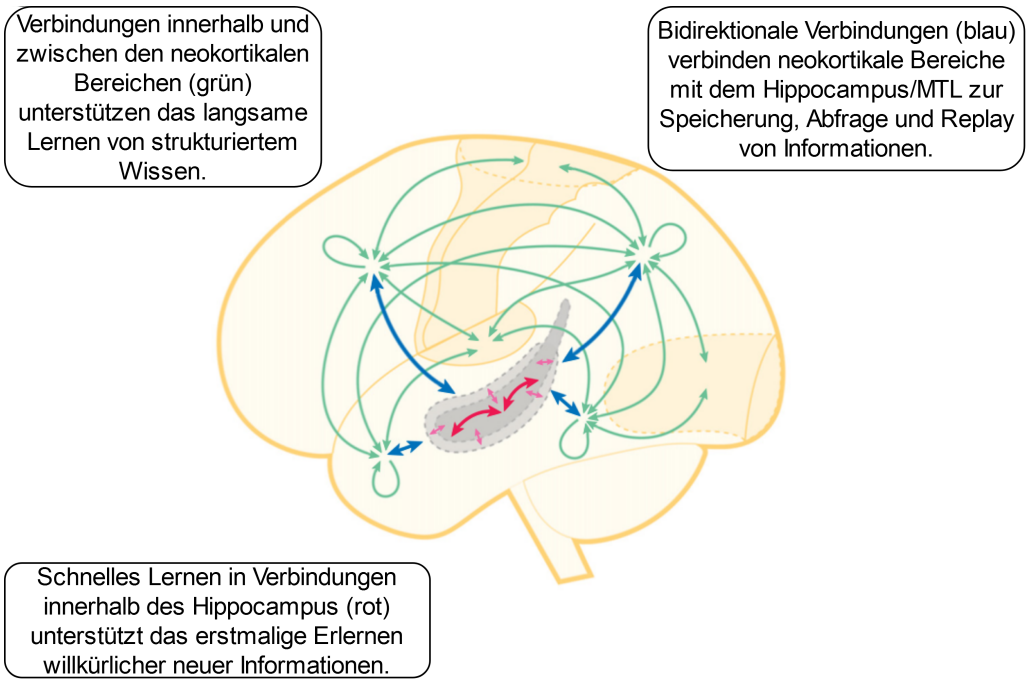


Abbildung 6.1: Seitliche Ansicht einer Gehirnhälfte (in Anlehnung an [KHM16]).

Neokortex

Eine der Kernannahmen der CLS-Theorie ist, dass der Neokortex Wissen auf eine strukturierte Art und Weise abspeichert. Dieser Grundsatz ergab sich aus der Beobachtung, dass mehrschichtige neuronale Netze nach und nach lernen Strukturen zu extrahieren, wenn sie durch Anpassung der Verbindungsgewichte trainiert werden. Ebenfalls wird davon ausgegangen, dass das Lernen dabei zwangsweise langsam vorangehen muss. Diese Annahme beruht auf zwei verschiedenen Gründen: (1) jedes Trainingsbeispiel repräsentiert eine reale Erfahrung aus der jeweiligen Umwelt bzw. Domäne. Eine geringe Lernrate erlaubt es somit die einer Aufgabe zugrundeliegenden Statistiken besser approximieren zu können, da die jeweiligen Informationen effektiver über mehrere Trainingsbeispiele aggregiert werden können. (2) Die optimale Anpassung eines jeden Verbindungsgewichts hängt von den Werten aller anderen Verbindungsgewichte ab. Ein initiales neuronales Netz enthält am Anfang viel Rauschen, da die Gewichte der unterschiedlichen Verbindungen meistens zufällig initialisiert werden. (vgl. [KHM16])

Diese beiden Punkte führen allerdings auch zu zwei Nachteilen. Zum einen muss es auch möglich sein, dass bestimmtes Verhalten auf Basis einzelner Erfahrungen/Trainingsbeispiele gelernt werden kann. Ein Beispiel, welches diesen Sachverhalt untermauert, ist, dass Menschen ebenfalls dazu in der Lage sind, aus individuellen/einzelnen Erfahrungen zu lernen. So ist es in der Regel der Fall, dass die meisten Personen eine lebensgefährliche Situation nach einmaligem Durchleben kategorisch umgehen können. Man könnte natürlich die Lernrate eines neuronalen Netzes entsprechend hoch wählen, um somit das Speichern der jeweiligen Erfahrung zu forcieren. Dies führt allerdings zum zweiten Nachteil. Schnelle bzw. große Änderungen in den jeweiligen Verbindungsgewichten eines neuronalen Netzes können zu katastrophaler Interferenz führen, was wiederum das Phänomen des katastrophalen Vergessens begünstigt. Da das Wissen von neuronalen Netzen in den jeweiligen Verbindungsgewichten gespeichert ist, kann sich das Wissen bezüglich eines einzelnen Sachverhalts über mehrere Verbindungen erstrecken. Diese werden jedoch an den einzelnen Neuronen kombiniert, was dann wiederum dazu führen kann, dass sich verschiedene Wissensrepräsentationen überlappen.

Hippocampus

Der Hippocampus hingegen speichert sein jeweiliges Wissen auf eine muster-separierten Art und Weise. Diese Speicherstruktur steht somit ebenfalls konträr zu der im Neokortex verwendeten, bei der Informationen in strukturierten und sich überlappenden Repräsentationen abgespeichert werden. Dies erlaubt allerdings das schnelle Lernen von Informationen, da bei dem Abspeichern keine Interferenz befürchtet werden muss.

Dadurch können der Hippocampus und verwandte Strukturen im medialen Temporallappen die anfängliche Speicherung erfahrungsspezifischer Informationen unterstützen, wie z. B. bestimmte Merkmale der Umgebung, in der man in eine lebensgefährliche Situation geraten ist. Allerdings hat auch diese Art von Speicherung Nachteile. So ist es z. B. ineffizient in Bezug auf die benötigten Kapazitäten dadurch, dass im Hippocampus keine „Ressourcen“ (Gewichte) geteilt werden. Ebenfalls kann keine Generalisierung stattfinden, da die einzelnen Trainingsbeispiele/Erfahrungen gespeichert werden und somit keine Aggregation über mehrere Beispiele hinweg möglich ist. (vgl. [KHM16])

Zusammenspiel beider Systeme

Nachdem in den vorherigen beiden Absätzen die einzelnen Systeme und ihre Funktionen erläutert wurden, ist deutlich geworden, in welchen Punkten sich Neokortex und Hippocampus unterscheiden. Allerdings bleibt die Frage bestehen, wie genau sich diese beiden Systeme nun ergänzen sollen. Für diese Fragestellung führt die CLS-Theorie den sogenannten *Replay* ein. Dabei handelt es sich um einen Mechanismus, bei dem das Wissen aus dem Hippocampus in den Neokortex integriert wird. Dies ist möglich, indem die neuen (schnell gelernten) Erfahrungen aus dem Hippocampus reaktiviert werden, um sie somit verschränkt mit älteren Erfahrungen in den Neokortex überspielen zu können. (vgl. [KHM16])

Bricht man das Ganze jetzt nochmal auf ein Minimum hinunter, bedeutet dies, dass der Replay-Mechanismus ein erneutes Durchleben von kürzlich generierten Erfahrungen simuliert, diese mit älteren Erinnerungen verschränkt und sie anschließend in den Neokortex überspielt. Dabei ist bisher jedoch ungeklärt, welche Erinnerungen genau für die erwähnte Verschränkung verwendet werden (vgl. [KHM16]). Letztend-

lich liefert die CLS-Theorie einen Ansatz, wie Informationen bzw. Wissen schnell gelernt und anschließend langsam in das Langzeitgedächtnis übertragen werden kann. Daher wird dieser Mechanismus, wie auch die Unterscheidung zwischen Neokortex und Hippocampus und ihren jeweiligen Lernmechanismen in den Algorithmus übernommen. An dieser Stelle soll jedoch nicht weiter auf diese Theorie oder ihre Anwendung innerhalb des Algorithmus eingegangen werden. Dies wird in den nächsten Unterkapiteln erfolgen. Den interessierten Leser verweisen wir für mehr Informationen bezüglich der CLS-Theorie ausdrücklich auf die Arbeit [KHM16] von Kumaran, Hassabis und McClelland. Diese enthält nicht nur detaillierter Erläuterungen bezüglich der unterschiedlichen Aspekte der CLS-Theorie, sondern untermauert diese z. B. zusätzlich noch mit empirischen Beweisen.

6.3 Aufbau und Funktionsweise

NeRO verwendet eine spezielle Modell-Architektur, welche sich aus zwei Komponenten zusammensetzt. Die einzelnen Komponenten werden dabei durch insgesamt 6 neuronale Netze repräsentiert. Der Aufbau der kompletten Architektur ist in Abbildung 6.2 zu sehen.

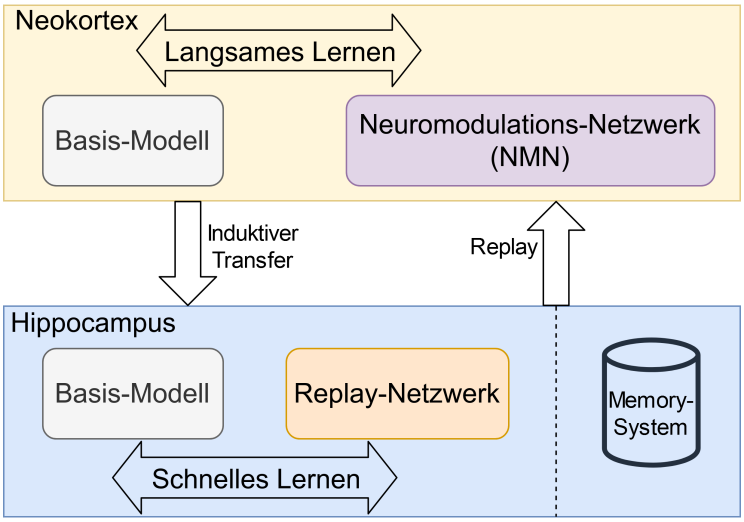


Abbildung 6.2: Die Architektur von NeRO mit ihren einzelnen Komponenten im Überblick.

Die einzelnen Komponenten sind für bestimmte Funktionalitäten verantwortlich. In Unterkapitel 6.2 wurde schon erläutert, dass NeRO unterschiedliche Ansätze nutzt, um die einzelnen Anforderungen und die sich daraus ergebenen Funktionalitäten umsetzen zu können. Diese verschiedenen Ansätze zeichnen sich auch in der präsentierten Architektur ab.
So wird gemäß der CLS-Theorie bei NeRO ebenfalls zwischen Hippocampus und Neokortex unterschieden. Dabei stellt diese Unterteilung der gesamten Architektur in diese zwei Komponenten die gröbste Granularität dar, welche für die weitere Struktur dieses Unterkapitels verwendet wird. In der Abbildung ist auch zu sehen, dass das sogenannte Memory-System ebenfalls Teil des Hippocampus ist, jedoch separat von seinen restlichen Komponenten existiert. Daher wird es auch in diesem Unterkapitel separat betrachtet werden.
Der Hippocampus lässt sich weiterhin unterteilen, da er mit dem Basis-Modell und Replay-Netzwerk zwei weitere Modelle enthält. Diese werden für das schnelle Lernen verwendet. Im Umkehrschluss bedeutet dies, dass der Hippocampus für das initiale Lernen und Speichern des Wissens der jeweiligen neuen Aufgabe zuständig ist. Das Replay-Netzwerk steuert dabei das Lernen im Hippocampus, wodurch der Hippocampus auf eine effiziente Art und Weise lernen können soll. Das Basis-Modell speichert dann das gelernte (aufgabenspezifische) Wissen ab und repräsentiert somit das Kurzzeitgedächtnis des Algorithmus.
Der Neokortex enthält ebenfalls mit dem Basis-Modell und dem Neuromodulations-Netzwerk (NMN) zwei weitere Modelle. Diese sind dann dementsprechend für das langsame Lernen zuständig. Damit ist der Neokortex für das Lernen und Speichern von generalisiertem Wissen über mehrere Aufgaben hinweg zuständig. Das NMN übernimmt dabei dann die Regulierung des Lernens, damit altes Wissen nicht von Neuem überschrieben wird. Das Basis-Modell ist für die Speicherung dieses Wissens zuständig und stellt somit das Langzeitgedächtnis des Algorithmus dar.
Das Memory-System dient zu guter Letzt dazu, die Verschränkung von altem und neuem Wissen gemäß der CLS-Theorie zu ermöglichen. Dadurch soll älteres Wissen in den Trainingsprozess des Neokortex integriert werden können, damit der Effekt des katastrophalen Vergessens eingedämmt werden kann.

Der nächste Abschnitt wird den Hippocampus mit seinen unterschiedlichen Bestandteilen und seiner Funktionsweise näher behandeln. Darauf aufbauend kann dann die Funktionsweise des Neokortex in Abschnitt 6.3.2 erläutert werden. Das Memory-System wird in Abschnitt 6.3.3 vorgestellt. Dabei wird ebenfalls näher auf seinen Aufbau und die Funktionsweise eingegangen. Im letzten Abschnitt 6.3.4 wird dann eine Zusammenfassung über den kompletten Lernprozess von NeRO gegeben, indem sein Ablauf in einer einzelnen Grafik zusammengefasst und dessen Kernelemente überblicksartig erläutert werden.

6.3.1 Hippocampus

Wie schon erwähnt, besteht der Hippocampus mit dem Replay-Netzwerk und dem Basis-Modell im Kern aus zwei verschiedenen Komponenten, welche jeweils für das Lernen und Speichern von aufgabenspezifischen Wissen zuständig sind. Das Basis-Modell wird durch ein herkömmliches neuronales Netz repräsentiert. Für das Replay-Netzwerk hingegen wird die Technik der Optimierer-Modelle verwendet, welche schon in Abschnitt 5.2.2 erläutert wurde. Daher ist bekannt, dass sie sich besonders gut für das Few-Shot-Lernen eignet. Diese Art des Lernens hat eine hohe Ähnlichkeit mit dem Lernmechanismus des Hippocampus, welcher sich nach der CLS-Theorie ebenfalls durch effizientes und schnelles Lernen auszeichnet. Daher ist es naheliegend, auch hier auf diese Technik zurückzugreifen.
Das Replay-Netzwerk wird durch ein LSTM repräsentiert. Als Input werden unter anderem die Koordinaten der prä- und postsynaptischen Neuronen einer Verbindung des Basis-Modells verwendet. Dafür werden die einzelnen Neuronen des Basis-Modells auf ein beliebiges Koordinatensystem abgebildet. Das ermöglicht es dem Replay-Netzwerk zwischen den einzelnen Verbindungen unterscheiden zu können, was den Trainingsprozess des Basis-Modells positiv beeinflussen sollte. Zusätzlich bekommt das Replay-Netzwerk aber auch noch den realen und letzten künstlichen Gradienten der jeweiligen Verbindung. Dadurch besteht die Möglichkeit, dass beim Training des Basis-Modells Momentum aufgebaut werden kann, äquivalent zu der Vorgehensweise von Optimierern wie Adam oder RMSProp. Der komplette Aufbau des Hippocampus und die Interaktionen zwischen den beiden Komponenten ist in Abbildung 6.3 zu sehen.

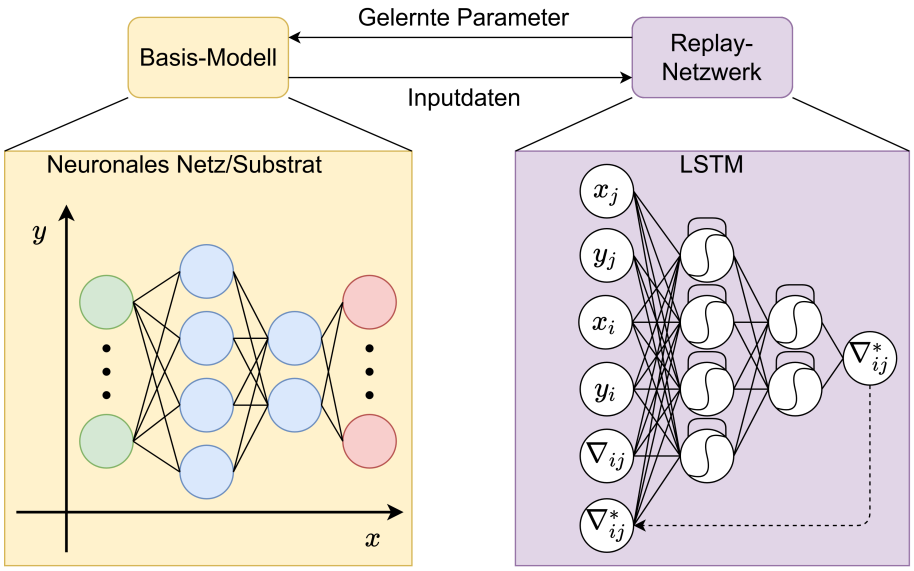


Abbildung 6.3: Der Hippocampus mit seinen einzelnen Komponenten.

Nachdem nun deutlich wurde, wie das Basis-Modell und Replay-Netzwerk aufgebaut sind, kann der Lernprozess im Hippocampus erläutert werden. Aus Algorithmus 1 ist ersichtlich, dass dieser aus zwei Phasen besteht: Basis- und Meta-Optimierung. Damit ist der Lernprozess des Hippocampus also an den des Meta-Lernens angelehnt. Daher ist bekannt, dass normalerweise zunächst mit der Basis-Optimierung des Basis-Modells begonnen wird. An dieser Stelle ergibt sich jedoch eine Besonderheit. Bevor damit begonnen wird das Basis-Modell zu trainieren, erfolgt der sogenannten *induktive Transfer*[1]. Hierbei werden die aktuellen Parameter des Basis-Modells des Neokortex mit dem Basis-Modell des Hippocampus geteilt. Dieses Prinzip soll es ermöglichen, schon gesammeltes Wissen für das Lernen einer neuen Aufgabe verwenden zu können, was dem Erfüllen von Anforderung 3 (Wissenstransfer) gleichkommt. Dadurch können bestimmte Wissensrepräsentationen, welche nützlich für das Bewältigen der neuen Aufgabe sind und durch vorheriges Lernen schon im Neokortex gespeichert sind, auch vom Hippocampus genutzt werden.

[1] Induktiver Transfer bezeichnet die Fähigkeit, Wissen oder Fähigkeiten bezüglich einer alten Aufgabe zu verwenden, um die aktuelle Aufgabe besser lösen oder ihren Lernprozess schneller durchführen zu können (vgl. [Utg+11]).

Algorithmus 1 : Lernen im Hippocampus

Input : $\theta, \beta, \sigma, D(T), \epsilon$
Output : θ^*, β^*
$\theta = \omega$; // Induktiver Transfer
while $L^{task} > \epsilon$ **do**
  /* Basis-Optimierung (schnelles Lernen) */
  $\theta^* = \theta$;
  for $(x, \hat{y}) \in T_{train}$ **do**
    $y = \text{predict}(\theta^*, x)$; // Forward-Pass
    $\nabla_{\theta^*} = \text{gradient}(\text{loss}(y, \hat{y}))$; // Gradientberechnung
    $\beta^*(\nabla_{\theta^*}) = \text{predict}(\beta^*, \nabla_{\theta^*})$; // Gradientenschritt
    $\theta^* = \text{update}(\theta^*, \beta^*(\nabla_{\theta^*}))$; // Parameter-Update
  end
  /* Meta-Optimierung */
  $L^{task} = 0$;
  for $(x, \hat{y}) \in T_{test}$ **do**
    $y = \text{predict}(\theta^*, x)$; // Forward-Pass
    $L^{task} = L^{task} + \text{loss}(y, \hat{y}))$; // Fehlerberechnung
  end
  $L^{task} = L^{task}/n$; // $T_{test} = \{(x, \hat{y})\}^n$
  if $(L^{task}) > \epsilon$ **then**
    $\beta^* = \text{BACKPROP}(\beta^*, L^{task})$; // Parameter-Update
  end
end
return θ^*, β^*;

Anschließend wird dann die Basis-Optimierung (= schnelles Lernen) durchgeführt. Gemäß der Definition des Meta-Lernens aus Abschnitt 5.1.1 wird dazu das Meta-Modell verwendet, welches bei NeRO durch das Replay-Netzwerk repräsentiert wird. Das Basis-Modell mit den Parametern θ^* bekommt zunächst das jeweilige Trainingsbeispiel $(x, \hat{y})$ aus dem Trainingsdatensatz T_{train} als Input und produziert den jeweiligen Output y. Anschließend können auf Basis des Fehlers zwischen y und $\hat{y}$ die Gradienten ∇_{θ^*} für das Basis-Modell berechnet werden. Diese Gradienten dienen dann wiederum dem Replay-Netzwerk als Input, um die Updates aller Verbindungsgewichte des Basis-Modells berechnen zu können. Dieses Vorgehen wird für alle Datenpunkte des Trainingsdatensatzes T_{train} mindestens einmal wiederholt.
In der zweiten Lernphase des Hippocampus wird die Optimierung des Replay-Netzwerkes durchgeführt. Dazu wird das trainierte Basis-Modell auf die einzelnen Beispiele $(x, \hat{y})$ aus dem Testdatensatz T_{test} angewendet. Die ermittelten Fehler werden aufsummiert und durch die Anzahl der Beispiele des Testdatensatzes geteilt, um somit den Fehler des Basis-Modells zu ermitteln. Dieser Fehler wird dann als Feedback verwendet, um festzustellen ob das Basis-Modell ausreichend gut für die jeweilige Aufgabe trainiert wurde. Mithilfe dieses Fehlers L^{task} kann dann das Replay-Netzwerk optimiert werden. Für diese (Meta-)Optimierung wird dann der Backpropagation-Algorithmus verwendet. Die exakte Wahl des Gradientenverfahrens wird an dieser Stelle offen gelassen und stellt somit einen Hyperparameter von NeRO dar.
Zusammengefasst kann man sagen, dass der Hippocampus für das Erlernen aufgabenspezifischer Informationen zuständig ist. Dabei wird das Paradigma des Meta-Lernens verwendet, um das Basis-Modell und Replay-Netzwerk für diese Aufgabe zu optimieren. Die Basis-Optimierung läuft dann in der epigenetischen Dimension ab, während die Meta-Optimierung in der phylogenetischen Dimension stattfindet. Durch den induktiven Transfer wird auf vorher schon erlerntes Wissen zurückgegriffen, was den Lernprozess schneller und effizienter gestalten kann. Diese Art des Lernens lässt sich auch mit der CLS-Theorie vereinbaren, da es ebenfalls aufgabenspezifisch abläuft und der Neokortex durch den induktiven Transfer ebenfalls Einfluss auf das Lernen nehmen kann.

6.3.2 Neokortex

Der Neokortex dient unter anderem dazu, das generalisierte Wissen bezüglich aller bisher gelernten Aufgaben abzuspeichern. Sein Basis-Modell repräsentiert in diesem Zusammenhang das Langzeitgedächtnis des Algorithmus. Dabei handelt es sich um ein herkömmliches neuronales Netz.
Allerdings ist bekannt, dass sich im Zusammenhang mit neuronalen Netzen mehrere Probleme (katastrophales Vergessen, Datenhunger usw.) ergeben, wenn sie sequenziell auf mehreren Aufgaben trainiert werden. Durch diese Probleme ist eine Anwendung als Langzeitspeicher im Kontext des Intra-Life-Learning ohne Weiteres nicht denkbar. Daher wird beim Neokortex der Ansatz der Neuromodulation verwendet. Dabei wird die Idee der zellulären Neuromodulation aus Abschnitt 5.4.4 übernommen. Das hat den Vorteil, dass das Neuromodulationssignal durch die Neuronen und nicht durch die Verbindungen des Basis-Modells modelliert wird. Dadurch sinkt wiederum die Anzahl der Parameter, welche durch die Neuromodulation berücksichtigt werden müssen. Anstatt also die synaptische Plastizität jeder Verbindung einzeln zu modellieren, werden diese an ihrem jeweiligen Zielneuron gebündelt und durch ein einzelnes Signal m gesteuert. Im Gegensatz zu den Arbeiten [Vec+19; Bea+20] kontrolliert die Neuromodulation hier jedoch nur die synaptische Plastizität der Verbindungen und nicht die Aktivierungszustände der Neuronen. Demnach ist nur das Lernen des neuronalen Netzes von der Neuromodulation betroffen, nicht jedoch seine Inferenz.
Die synaptische Plastizität des Basis-Modells soll im Neokortex dann von dem NMN gesteuert werden. Das NMN selbst besteht wiederum aus zwei Komponenten, wobei hier die Idee, zwischen Phäno- und Genotyp zu unterscheiden, genutzt wird. Der Phänotyp stellt dabei die Komponente dar, welche letztendlich die Neuromodulation steuert und kann durch ein beliebiges neuronales Netz repräsentiert werden. Für den Genotyp wird ein CPPN verwendet, welches die Gewichte des Phänotyp kodiert. Dadurch kann die Anzahl an Parametern reduziert werden, die durch den Meta-Optimierer berücksichtigt werden müssen. Das CPPN kann dann anhand der Koordinaten der prä- und postsynaptischen Neuronen das Gewicht einer Verbindung im Phänotyp berechnen. Der Phänotyp erhält wiederum dieselben Daten als Input, wie auch das Basis-Modell. Dadurch können die neuromodulativen Signale m_i aller

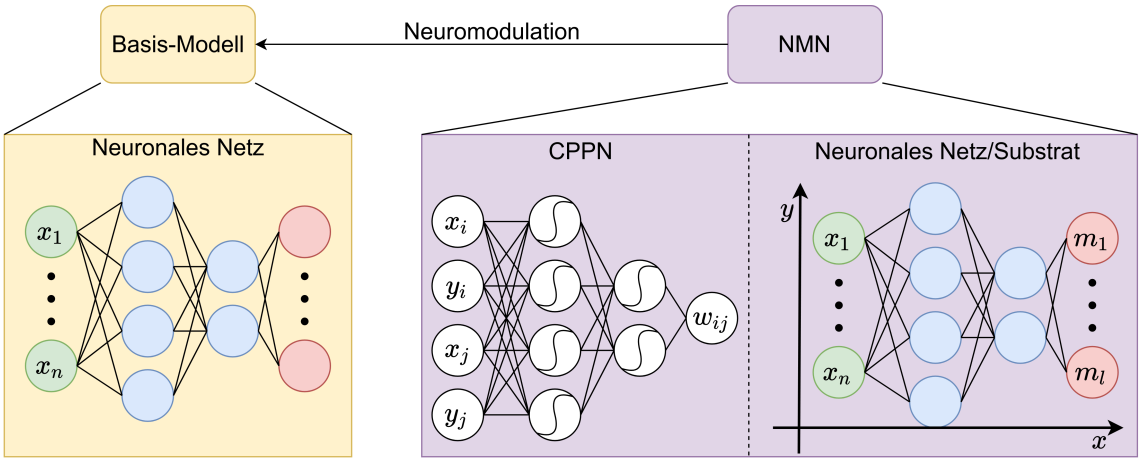


Abbildung 6.4: Der Neokortex mit seinen einzelnen Komponenten.

Neuronen des Basis-Modells abhängig vom Input berechnet werden. Das ermöglicht, dass neues Wissen intuitiv durch das NMN in den Regionen des Basis-Modells gespeichert wird, welche ähnliches Wissen enthalten. Zusätzlich kann dadurch neben dem Wissen des Basis-Modells auch das Wissen des NMN generalisiert werden.
Als Meta-Optimierer für das NMN wird an dieser Stelle das Vorgehen des *adaptive HyperNEAT*-Algorithmus übernommen. Dadurch können zeitgleich das Substrat des Phänotyps und das CPPN optimiert werden. Da dieser Ansatz NEAT als Optimierer verwendet, wird zusätzlich auch der Baldwin-Effekt aus Abschnitt 6.2.2 genutzt.
Nachdem erläutert wurde, wie die einzelnen Komponenten des Neokortex funktionieren, kann nun auf das Lernen insgesamt eingegangen werden. In Algorithmus 2 ist zu sehen, dass genauso wie beim Hippocampus der Lernprozess des Neokortex aus zwei Phasen besteht. Dabei wird zunächst wieder das Basis-Modell optimiert (= langsames Lernen). Da hier allerdings die Besonderheit besteht, dass die Neuromodulation die katastrophale Interferenz beim Lernen im Basis-Modell auf ein Minimum zu reduzieren versucht, ergeben sich an dieser Stelle einige Änderungen.
Zunächst bekommt das Basis-Modell mit den Parametern ω^* den Input x und berechnet auf dieser Basis den Output y. Anschließend erfolgt auch hier die Berechnung aller Gradienten ∇_{ω^*} des Basis-Modells durch den ermittelten Fehler zwischen y und $\hat{y}$. An dieser Stelle wird das trainierte Replay-Netzwerk mit den Parame-

Algorithmus 2 : Lernen im Neokortex

Input : $\omega, \mu, \beta^*, \sigma, D(T), \epsilon$
Output : ω^*, μ^*
$\mu^* = \mu$;
while $L^{meta} > \epsilon$ **do**
  $\omega^* = \omega$;
  /* Basis-Optimierung (langsames Lernen) */
  for $(x, \hat{y}) \in M_{train}$ **do**
    $y = \text{predict}(\omega^*, x)$; // Forward-Pass
    $\nabla_{\omega^*} = \text{gradient}(\text{loss}(y, \hat{y}))$; // Gradientenberechnung
    $\beta^*(\nabla_{\omega^*}) = \text{predict}(\beta^*, \nabla_{\omega^*})$; // Gradientenschritt
    $m_{\omega^*} = \text{predict}(\mu^*, x)$; // Neuromodulation
    $\omega^* = \text{update}(\omega^*, m_{\omega^*}, \beta^*(\nabla_{\omega^*}))$; // Parameter-Update
  end
  /* Meta-Optimierung */
  $L^{meta} = 0$;
  for $(x, \hat{y}) \in M_{test}$ **do**
    $y = \text{predict}(\omega^*, x)$; // Forward-Pass
    $L^{meta} = L^{meta} + \text{loss}(y, \hat{y}))$; // Fehlerberechnung
  end
  $L^{meta} = L^{meta}/n$; // $M_{test} = \{(x, \hat{y})\}^n$
  if $L^{meta} > \epsilon$ **then**
    $\omega^* = \text{NEAT}(\omega^*)$; // Parameter-Update
  end
end
return ω^*, θ^*;

tern β^* verwendet, um auf Basis von ∇_{ω^*} die künstlichen Gradienten $\beta^*(\nabla_{\omega^*})$ für das Basis-Modell zu ermitteln. Anschließend erfolgt die Neuromodulation durch das NMN mit den Parametern μ für das Basis-Modell. Die daraus resultierenden Neuromodulationssignale m_ω werden in Kombination mit den künstlichen Gradienten $\beta^*(\nabla_\omega)$ verwendet, um die Parameter ω^* vom Basis-Modell neu zu berechnen. Dieses Vorgehen wird für jedes Trainingsbeispiel in M_{train} mindestens einmal wiederholt. Dabei repräsentiert M_{train} einen speziellen Meta-Trainingsdatensatz. Dieser ergibt sich aus der Vereinigung von T_{task} und den vom Memory-System produzierten Daten $D(\sigma) = \{(x_\sigma, y_\omega)\}$, wobei x_σ vom Memory-System mit den Parametern σ und y_ω vom Basis-Modell mit den alten Parametern ω generiert wird. Das gleiche Prinzip wird auch für den Datensatz M_{test} angewendet, sodass sich für diese beiden Datensätze folgende Definition ergeben:

$$M_{train} = \{(x, y)|x,y \in \{T_{train} \cup D(\sigma)\}\} \tag{6.1}$$

$$M_{test} = \{(x, y)|x, y \in \{T_{test} \cup D(\sigma)\}\} \tag{6.2}$$

Diese Definition lässt offen, zu welchen Anteilen die jeweiligen Datensätze in M_{train} und M_{test} einfließen. So besteht die Möglichkeit, dass das Basis-Modell des Neokortex ausschließlich auf den Trainingsdaten T_{train} trainiert wird. Alternativ können zusätzlich Daten aus $D(\sigma)$ hinzugefügt werden.
Für die Meta-Optimierung besteht jedoch ein Zwang, die jeweiligen Datensätze für M_{test} zu verschränken, da ansonsten kein aussagekräftiges Feedback in Bezug auf das katastrophale Vergessen und die Performance des Neokortex auf der aktuellen Aufgabe gegeben werden kann. Das trainierte Basis-Modell mit seinen Parametern ω^* wird dann auf alle Beispiele aus M_{test} angewendet. Die daraus resultierenden Fehlerwerte werden aufsummiert und anschließend durch die Anzahl der Beispiele aus M_{test} dividiert, um somit den Meta-Fehler zu erhalten. Auf Basis dieses Meta-Fehlers kann dann das NMN optimiert werden. Als Meta-Optimierer wird an dieser Stelle der NEAT-Algorithmus verwendet. Dabei wird sich allerdings an dem Optimierungsvorgang des *adaptive ES-HyperNEAT*-Algorithmus [RS12] orientiert, da dieser nicht nur das CPPN entwickelt, sondern zeitgleich auch das dazugehörige Substrat.

Zusammengefasst kann man sagen, dass der Neokortex für das langsame Lernen von generalisiertem Wissen zuständig ist. Dabei wird auch hier das Paradigma des Meta-Lernens verwendet, um das Basis-Modell und NMN für diese Aufgabe zu optimieren. Der Prozess der Basis-Optimierung läuft dabei in der ontogenetischen Dimension ab, während die Meta-Optimierung (wie auch im Hippocampus) in der phylogenetischen Dimension stattfindet. Um das katastrophale Vergessen zu vermeiden, wird der Ansatz der Neuromodulation genutzt. Die zusätzlich Verwendung des Replay-Netzwerkes für die Basis-Optimierung repräsentiert an dieser Stelle den Replay aus der CLS-Theorie. Dadurch, dass das Replay-Netzwerk beim Lernen im Hippocampus für die Optimierung bezüglich der zu erlernenden Aufgabe optimiert wurde, kann es auch für die Optimierung des Basis-Modells vom Neokortex verwendet werden. Das NMN übernimmt dann die Rolle der Neuromodulation, wodurch in der Kombination von Optimierer-Modell und Neuromodulation die eigentliche Kernidee von NeRO liegt. Die zusätzliche Verschränkung alter und neuer Daten im Basis- und Meta-Optimierungsprozess ist ebenfalls im Sinne der CLS-Theorie, da bei dieser auch alte mit neuen Erfahrungen kombiniert werden, während der Replay ausgeführt wird. Durch die Kombination all dieser Mechanismen kann somit das Wissen aus dem Hippocampus in den Neokortex übertragen werden. Das wiederum resultiert in einem Algorithmus, welcher schnell neues Wissen bezüglich einer Aufgabe erlernen kann, um es anschließend langsam in den Langzeitspeicher zu übertragen, ohne dass schon vorhandenes Wissen dabei verdrängt wird.

6.3.3 Memory-System

In Abschnitt 6.2.3 wurde im Kontext der CLS-Theorie schon erläutert, dass beim Replay kürzlich gelernte Erfahrungen des Hippocampus „simuliert“ und dabei mit älteren Erfahrungen verschränkt werden, um das neue Wissen in den Neokortex übertragen zu können. Dabei muss es jedoch möglich sein, auf ältere Daten bzw. Erfahrungen zurückgreifen zu können. Da das Basis-Modell des Hippocampus bei diesem Algorithmus jedoch nur Informationen bezüglich der aktuellen Aufgabe lernt und speichert, reicht es alleine nicht aus, um diese Verschränkung vorzunehmen.
An dieser Stelle findet das Memory-System seine Anwendung. Es wird wie durch sogenannte Generative Adversial Networks (GANs) [Goo+14] repräsentiert. Dabei

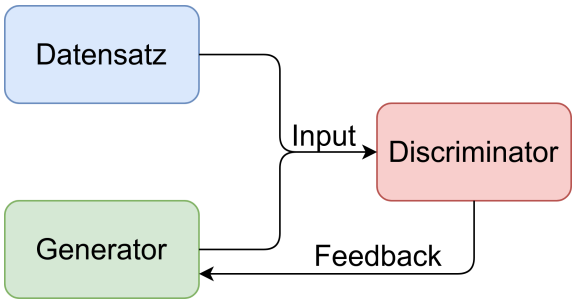


Abbildung 6.5: Der Trainingsprozess eines GANs vereinfacht dargestellt.

handelt es sich im Grunde genommen um zwei neuronale Netze, die „gegeneinander“ arbeiten und sich dadurch jeweils selber optimieren. In Abbildung 6.5 ist ein Beispiel für den Aufbau des Memory-Systems zu sehen.
Diese beiden neuronalen Netze werden auch als *Generator* und *Discriminator* bezeichnet. Dabei hat der Generator die Aufgabe, künstliche Daten zu generieren. Damit stellt er also den eigentlichen „Speicher“ innerhalb des Algorithmus dar, welcher die Daten bezüglich aller schon gelernten Aufgaben generieren können soll.
Der Discriminator hingegen vergleicht den künstlich generierten Input des Generators mit dem echten Input aus einem Datensatz und soll dann entscheiden, ob es sich jeweils um echte oder generierte Daten handelt. Er hat also die Aufgabe, dem Generator ein Feedback zu geben, welches anzeigt, ob die generierten Daten qualitativ nahe an ihren realen Äquivalenten liegen. Somit wird diese Komponente nicht direkt für den Algorithmus verwendet, ist aber dennoch notwendig, da ohne sie der Generator nicht trainiert werden kann. Der Trainingsprozess ist stark vereinfacht in Abbildung 6.5 dargestellt. Mathematisch gesehen handelt es bei diesem Prozess um eine Minmax-Funktion (vgl. [Goo+14]):

$$\min_G \max_D [\mathbb{E}_{x \sim p_{data}} \log D(x) + \mathbb{E}_{z \sim p_z} \log(1 - D(G(z)))] \tag{6.3}$$

Dabei bezeichnen G und D jeweils die Parameter des Generator und Discriminators. Es ist ersichtlich, dass der Discriminator versucht, die Funktion zu maximieren.

Daher wird für ihn der Gradientenaufstieg durchgeführt (vgl. [Goo+14]):

$$\nabla_D \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))] \tag{6.4}$$

Der Generator hingegen versucht die Funktion zu minimieren, weswegen hier der Gradientenabstieg verwendet wird (vgl. [Goo+14]):

$$\nabla_G \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))) \tag{6.5}$$

Durch den Wechsel beider Verfahren wird das Netzwerk trainiert. Mit dieser Vorgehensweise kann ein GAN für einen einzelnen Datensatz optimiert werden. Jedoch muss dieser Prozess noch auf das Szenario des Intra-Life-Learning übertragen werden, da es das Ziel ist, die Datensätze aller schon gelernten Aufgaben mit einem einzelnen GAN generieren zu können.
Dazu wird sich an der Arbeit [Shi+17] orientiert. Darin wird ein Prozess vorgeschlagen, welcher die GANs im Kontext des kontinuierlichen Lernens anwendet. Der Prozess ist schematisch in Abbildung 6.6 dargestellt.

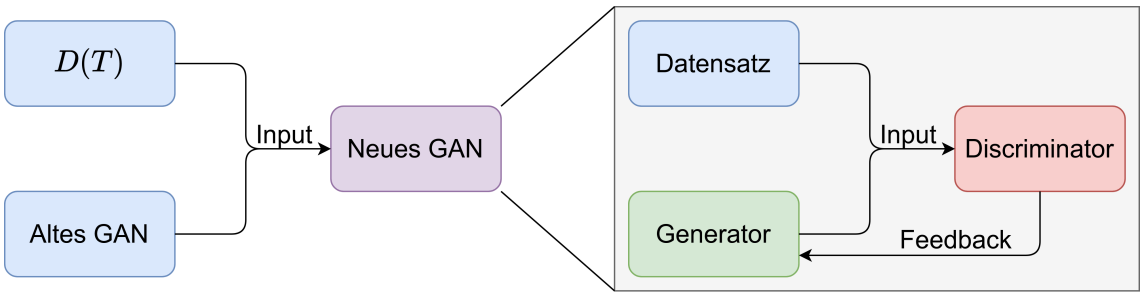


Abbildung 6.6: Der Intra-Life-Learning-Prozess eines GANs vereinfacht dargestellt.

Es ist ersichtlich, dass als Input für den Trainingsprozess des neuen GAN das alte GAN und der jeweilige Datensatz $D(T)$ bezüglich der aktuellen Aufgabe T verwendet wird. Dadurch werden wie auch schon beim Lernen im Neokortex alte mit neuen Daten verschränkt und somit als Grundlage für das Training des neuen GAN verwendet. Daraus resultierend lernt der Generator des neuen GAN wie er ähnliche Verteilung bezüglich des verschränkten Datensatzes generieren kann.

6.3.4 Zusammenfassung aller Prozesse

Um sich nicht in den Details dieses Unterkapitels zu verlieren, soll an dieser Stelle nochmal der gesamte Lernprozess von NeRO dargestellt werden. Dazu wurden die einzelnen Teilprozesse, welche im Hippocampus und Neokortex ablaufen, in Abbildung 6.7 zusammengefasst. Es ist ersichtlich, dass zunächst zwischen dem Lernen im Hippocampus und Neokortex unterschieden werden kann. Dabei ist der Hippocampus für das schnelle Lernen von initialem Wissen bezüglich einer neuen Aufgabe zuständig. Der Neokortex hingegen soll dieses Wissen langsam in sein Basis-Modell integrieren, ohne dabei schon vorhandenes Wissen zu überschreiben und somit katastrophal zu vergessen.

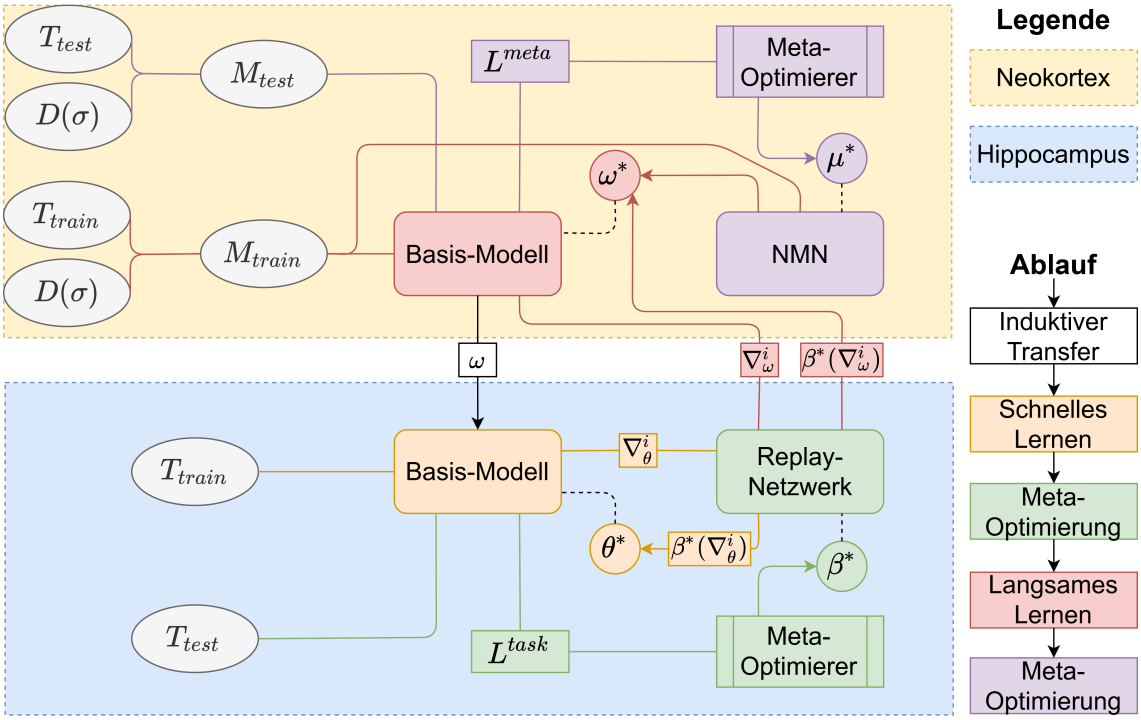


Abbildung 6.7: Der Ablauf des gesamten Prozesses für das Lernen einer Aufgabe.

Geht man mehr ins Detail, lässt sich das Lernen im Hippocampus in drei Prozesse unterteilen: Induktiver Transfer, schnelles Lernen (= Basis-Optimierung) und Meta-Optimierung. Der induktive Transfer ist dabei recht eingängig, da hier nur die Parameter von einem Basis-Modell (Neokortex) zum anderen (Hippocampus) transferiert werden. Das schnelle Lernen hingegen verwendet den Trainingsdatensatz T_{train} der Aufgabe T und optimiert mithilfe des Replay-Netzwerkes die Parameter θ^* des Basis-Modells (orange). Sobald dieser Prozess beendet ist, wird mithilfe des Testdatensatzes T_{train} und dem trainierten Basis-Modell der Fehler L^{task} ermittelt, welcher dann als Grundlage für die Optimierung der Parameter β^* des Replay-Netzwerkes verwendet werden kann. Diese drei Prozesse stellen dann in ihrer Summe das Lernen im Hippocampus dar.
Das Lernen im Neokortex lässt sich in zwei Teilprozesse untergliedern: Langsames Lernen (= Basis-Optimierung) und Meta-Lernen. Dabei stellt das langsame Lernen den wichtigsten Prozess des gesamten Algorithmus dar und markiert somit auch seine Kernidee. Dabei wird das Basis-Modell mit einem speziellen Datensatz M_{train} trainiert, welcher aus verschränkten Daten bestehen kann. Alternativ kann M_{train} auch nur Daten aus dem Trainingsdatensatz bezüglich der aktuellen Aufgabe T enthalten, sodass $M_{train} = T_{train}$ entspricht. Zusätzlich werden für das Training des Basis-Modells zwei neuronale Netze eingesetzt. Dabei produziert das Replay-Netzwerk das jeweilige Update der Parameter ω^* des Basis-Modells. Diese Updates werden jedoch durch das NMN mithilfe der Neuromodulation so skaliert, dass der Effekt des katastrophalen Vergessens möglichst auf ein Minimum reduziert wird. Mithilfe des so trainierten Basis-Modells und dem verschränkten Testdatensatz M_{test} kann letztendlich in Form des Fehlers L^{meta} kontrolliert werden, wie gut das katastrophale Vergessen eingedämmt werden konnte. Auf dieser Basis kann dann wiederum das NMN optimiert werden. Die Meta-Optimierung markiert dann auch den Endpunkt des Lernprozesses. Das Lernen des Memory-Systems ist als gesonderter Prozess konzipiert und besitzt somit keine Abhängigkeiten zum regulären Lernen. Damit fällt es aus der hier durchgeführten Betrachtung wie auch aus der Abbildung 6.7 heraus.

6.4 Parallelisierung

Damit der in Unterkapitel 6.3 erläuterte Lernprozess des Systems noch effizienter gestaltet werden kann, wird in diesem Unterkapitel seine Parallelisierung diskutiert. Dazu wird Abschnitt 6.4.1 kurz in die Parallelisierungsplattform *Apache Spark* einführen und ihre grundlegenden Mechanismen erläutern. Auf dieser Basis kann dann in Abschnitt 6.4.2 erläutert werden, wie der Lernalgorithmus von NeRO parallelisiert werden kann.

6.4.1 Apache Spark

Apache Spark bezeichnet ein Framework für das Cluster-Computing. Dieses ist 2009 im Laufe eines Forschungsprojektes vom *AMPLab* an der *University of California* entstanden und steht seit 2010 unter einer Open-Source-Lizenz. Im Jahr 2013 wurde das Projekt dann offiziell von der *Apache Foundation* weitergeführt und 2014 dann schließlich als Top-Level-Projekt eingestuft.
Spark setzt sich aus mehreren Teilen zusammen. Den Kern bilden dabei die RDD API und die darauf aufbauenden Bibliotheken. Spark kann auch an unterschiedliche Datenquellen angebunden und mit mehreren Cluster-Managern kombiniert werden, wodurch es vielfältig einsetzbar ist. Ein Überblick über die einzelnen Bestandteile von Spark ist in Abbildung 6.8 zu sehen.
Für den weiteren Verlauf dieses Unterkapitels ist vor allem die RDD API wichtig. Diese stellt mit dem Resilient Distributed Dataset (RDD) eine fehlertolerante, verteilte Datenstruktur bereit, um auch Berechnungen auf großen Datensätzen parallelisieren zu können. Dabei kann ein RDD auf zwei unterschiedliche Arten erstellt werden: entweder durch das Laden aus einer externen Datenquelle oder durch das „parallelisieren“ eines schon existierenden Datensatzes:

```
1    # Laden eines existierenden Datensatzes in ein RDD
2    data = [x_1, x_2, x_3]
3    rddData = sc.parallelize(data)
4    # Laden aus einer externen Datenquelle
5    rddFile = sc.textFile("/home/data.txt")
```

Das Ausführen einer Änderungsoperation auf einem vorhandenen RDD ist eine weitere Variante, um ein neues RDD zu erzeugen. Dafür stellt die API sogenannte

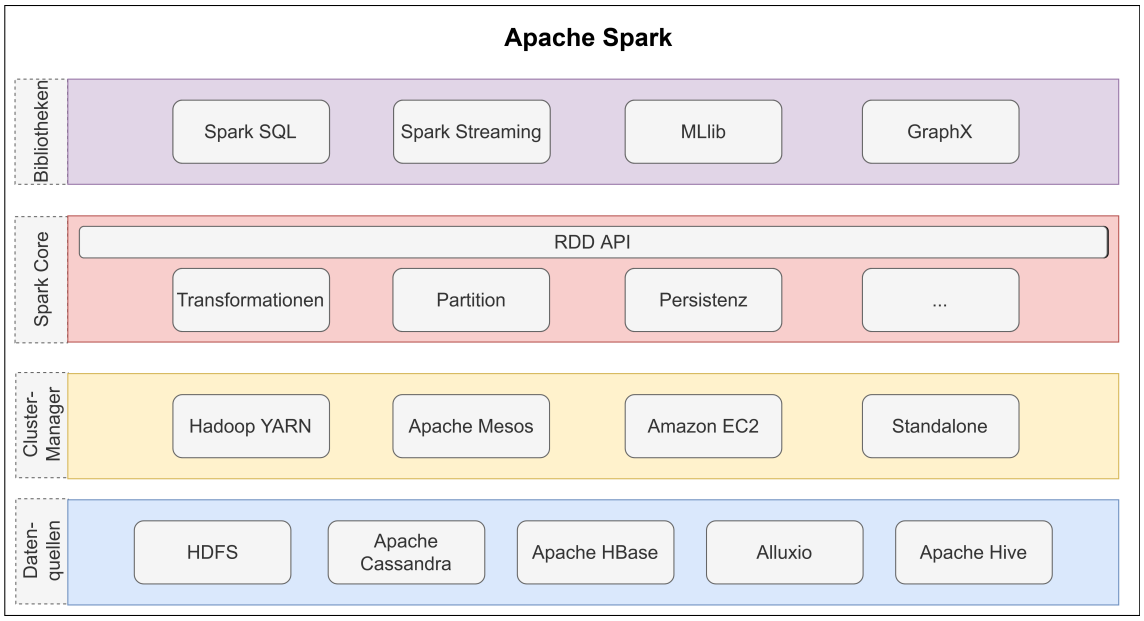


Abbildung 6.8: Der Aufbau von Apache Spark im Überblick.

Transformationen und *Aktionen* bereit. Ein Beispiel dafür ist, dass man die *map*-Funktion verwendet, welche eine Transformation darstellt. Damit kann dann z. B. eine selbst definierte Funktion auf ein bestehendes RDD angewendet werden:

```
1    if __name__ == "__main__":
2    def meineFunktion(data):
3        worte = s.split(" ")
4        return len(worte)
5
6    conf = SparkConf().setAppName(appName).setMaster(master)
7    sc = SparkContext(conf=conf)
8    worte = sc.textFile("data.txt").map(myFunc)
```

Neben dieser Funktion stellt die API natürlich noch weitere zur Verfügung. Von einer vollständigen Auflistung wird an dieser Stelle jedoch abgesehen, da es dem Zweck der Arbeit nicht weiter zuträglich wäre.
Durch die Speicherung der Daten in einem RDD wird zudem versucht, die Datenreplikation zu vermeiden. Dazu werden die verwendeten Operationen, welche das RDD generiert haben, in einem Log gespeichert. Im Falle eines Datenverlustes oder Fehlers auf einem der Cluster-Knoten kann dann die verloren gegangene Partition des

RDD mithilfe der anderen RDDs rekonstruiert werden, ohne zusätzlichen Speicher für die Datenreplikation verwenden zu müssen. Durch diesen Mechanismus stellen die RDDs ihre Fehlertoleranz sicher.
Weitere Aspekte, die vom Nutzer beeinflusst werden können, sind die Partitionierung und Persistenz von RDDs. Dabei kann die Partitionierung durch einen vom Nutzer festgelegten Schlüssel manuell bestimmt werden. Es ist ebenso möglich die Art der Speicherung eines RDD zu beeinflussen. Benötigt man bestimmte Daten z. B. auf sämtlichen Knoten des Computer-Clusters, so kann das entsprechende RDD per Broadcast auf alle Cluster-Knoten repliziert werden.
Ein letzter Punkt ist, dass Spark die RDD API für unterschiedliche Programmiersprachen wie Java, Python, Scala oder R anbietet. Vor allem die Schnittstelle für Python ist an dieser Stelle von hohem Interesse, da viele entsprechende Bibliotheken für das maschinelle Lernen und neuronale Netze für diese Programmiersprache zur Verfügung gestellt werden.

6.4.2 Umsetzung

Wie in Unterkapitel 6.3 schon erläutert, besteht der Lernalgorithmus von NeRO aus zwei Teilen: dem Lernen im Hippocampus und dem Lernen im Neokortex. Dabei setzen sich beide Teile wiederum aus mehreren Prozessen zusammen. Nachfolgend wird erläutert, welche dieser Prozesse sich parallelisieren lassen und welche Änderungen dafür vorgenommen werden müssten.
Die Basis-Optimierung in beiden Komponenten (Hippocampus und Neokortex) wird von jeweils einem oder mehreren neuronalen Netzen berechnet. Daher lässt sich diese auch auf den ersten Blick nicht parallelisieren. Jedoch ist dies jeweils nur bei dem ersten Durchlauf des Lernens in Hippocampus und Neokortex der Fall. Die anschließende Meta-Optimierung generiert z. B. für den Neokortex mehrere Varianten für das NMN, da es sich beim Meta-Optimierer um einen neuroevolutionären Algorithmus handelt. In Abschnitt 4.1.2 wurde schon erläutert, dass sich evolutionäre Algorithmen bzw. Verfahren der Neuroevolution besonders gut für eine Parallelisierung eignen, da diese Algorithmen naturgemäß über einer Population von Lösungen optimieren, anstatt einer einzelnen.

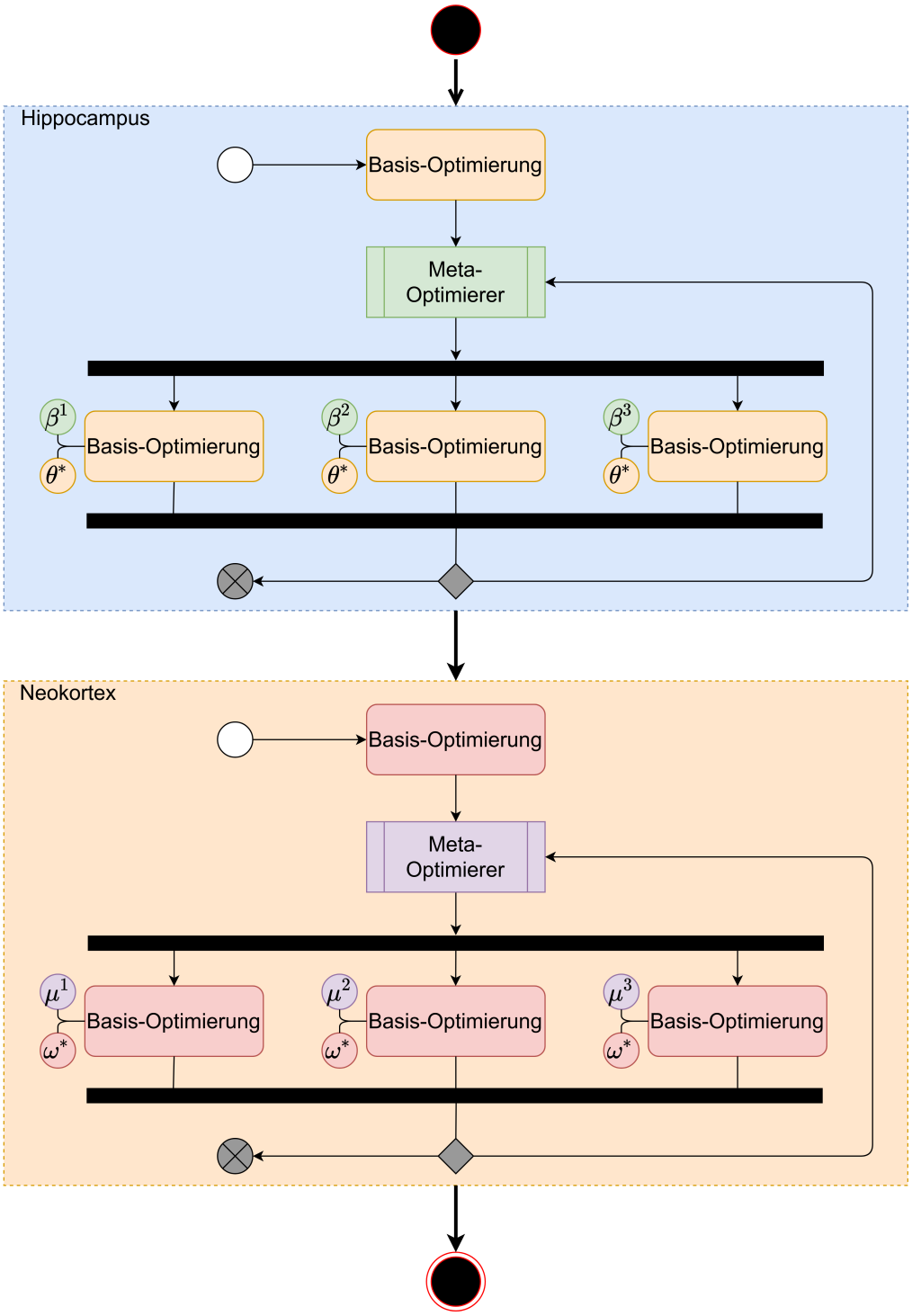


Abbildung 6.9: Die parallele Variante vom NeRO-Algorithmus.

Klassische Lernalgorithmen wie Backpropagation oder Hebb-Lernen hingegen lassen sich schwerer parallelisieren. Daher muss für eine komplett parallelisierte Variante von NeRO ein anderer Algorithmus als Meta-Optimierer für den Hippocampus verwendet werden. So wäre ein Vorschlag, den Backpropagation-Algorithmus durch NEAT zu ersetzen. Jedoch kann hier das Problem aufkommen, dass NEAT zu ineffizient wäre, wenn das Replay-Netzwerk aus mehreren Schichten bestehen würde. In diesem Fall wäre der Parameter-Suchraum einfach zu groß, um ein Optimum durch NEAT zu finden.
Um dies zu vermeiden, könnte jedoch der Gradientenabstieg weiterhin als Engine hinter der Mutation von NEAT verwendet werden. Also anstatt die Gewichte „zufällig“ bei der Mutation zu verändern, könnten diese weiterhin durch den Gradientenabstieg oder eine seiner Varianten angepasst werden. Eine weitere Alternative wäre es, sogenannte *sichere Mutationen* [Leh+18] zu verwenden. Dabei können zufällige Mutationen dahingehend bewertet werden, was für einen Einfluss sie auf das Verhalten des neuronalen Netzes haben. Zusätzlich besteht die Möglichkeit, zwischen einer Berechnung mit oder ohne Gradienten zu wählen, je nachdem, ob das jeweilige neuronale Netz differenzierbar ist oder nicht.
Verwendet man eine der eben genannten Alternativen als Meta-Optimierer, so können die Berechnungen für den Hippocampus und Neocortex jeweils parallelisiert werden. In Abbildung 6.9 ist zu sehen, welche Teilschritte parallelisierbar sind. Die schwarzen Kästen stellen dabei die jeweilige Verteilung der Berechnungen auf mehrere Cluster-Knoten und anschließende Zusammenführung der Ergebnisse dar. Die grauen Rauten repräsentieren eine Verzweigung. Je nachdem, ob die jeweiligen Ergebnisse zufriedenstellend sind oder nicht, terminiert das Lernen in der jeweiligen Komponente oder der Lernprozess wird ein weiteres Mal durchlaufen. Es ist auch zu sehen, dass jeder der Cluster-Knoten für die Basis-Optimierung die gleichen Parameter für das Basis-Modell erhält. Die Parameter des jeweiligen Meta-Optimierers (Replay-Netzwerk oder NMN) unterscheiden sich jedoch.
Mithilfe der RDD API lässt sich diese Parallelisierung auch einfach umsetzen. Dabei können die einzelnen Kandidaten, welche durch die Meta-Optimierung generiert werden, in einem RDD gespeichert werden. Die Methode für die jeweilige Basis-Optimierung kann dann als Parameter an die *map*-Funktion der RDD API über-

geben werden. Spark teilt dann automatisch die einzelnen Kandidaten aus dem entsprechenden RDD auf die verfügbaren Cluster-Knoten auf und führt dort lokal die Berechnung der übergebenen Methode (Basis-Optimierung) aus. Um die Performance zusätzlich zu erhöhen, können die Parameter des Basis-Modells und die benötigten Datensätze über eine von Spark bereitgestellte *broadcast*-Funktion auf die einzelnen Cluster-Knoten repliziert werden, sodass der Netzwerkverkehr zwischen den einzelnen Knoten reduziert werden kann.
Für die programmatische Umsetzung der neuronalen Netze stehen mit TensorFlow, Caffee, PyTorch oder Keras genügend Bibliotheken für Python zur Verfügung. Wie in Abschnitt 6.4.1 stellt auch Apache Spark eine Schnittstelle für diese Programmiersprache zur Verfügung, sodass die eben erwähnten Bibliotheken auch in Kombination mit der Parallelisierung über Spark verwendbar sein sollten.

6.5 Zusammenfassung

Dieses Kapitel hat mit NeRO einen Algorithmus vorgestellt, welcher für das Intra-Life-Learning verwendet werden kann und zudem auch noch auf parallele Datenverarbeitung ausgelegt ist. Dazu wurden in Abschnitt 6.1 Anforderungen festgelegt, die bei der Konzeption von NeRO zu berücksichtigen waren. Das Unterkapitel 6.2 hat dann die Ansätze erläutert, die bei der Konzeption von NeRO als Grundlage dienten. Dabei wurde aufgezeigt, wie die Erkenntnisse aus Kapitel 5 bei dem Design genutzt werden konnten. Zusätzlich wurden mit dem Baldwin-Effekt und der CLS-Theorie zwei weitere (biologisch) plausible Theorien erläutert, welche ebenfalls angewendet werden konnten. Das Unterkapitel 6.3 hat dann schließlich den Aufbau und die Funktionsweise von NeRO vorgestellt. Dabei ist vor allem die Unterteilung in Hippocampus und Neokortex und das Zusammenspiel ihrer jeweiligen Lernmechanismen hervorzuheben, welche sich gegenseitig ergänzen und es somit ermöglichen, verschiedene Aufgaben in einer sequenziellen Reihenfolge zu erlernen, ohne dabei vorher gesammeltes Wissen zu vergessen. Zusätzlich begünstigen Mechanismen wie der induktive Transfer den Lernprozess bezüglich neuer Aufgaben, indem einen Wissenstransfer forciert wird. Das Unterkapitel 6.4 hat schließlich aufgezeigt, wie NeRO mithilfe von Apache Spark parallelisiert werden kann.

7 Schlussbetrachtung

Die Zielstellung dieser Arbeit bestand darin, die Neuroevolution im Rahmen des Intra-Life-Learnings zu untersuchen und zu analysieren, inwiefern sie die damit verknüpften Problemstellungen erfüllen kann.
Dazu wurde in Kapitel 1 zunächst das Thema motiviert und in die angrenzenden Forschungsgebiete des maschinellen Lernens und der evolutionären Algorithmen eingeführt.
Das 2. Kapitel hat anschließend die Grundlagen der neuronalen Netze, in Bezug auf ihren Aufbau und ihre Funktionsweise, vermittelt. Dazu wurden vom Verfasser dieser Arbeit eigene Definitionen eingeführt, um bei dem Leser ein konsistentes Verständnis dieser Thematik sicherzustellen.
Darauf aufbauend wurde in Kapitel 3 das maschinelle Lernen erläutert. Dazu wurden weitere Grundlagen im Bereich der Fehlerfunktionen und Analysis gelegt, um auf dieser Basis mit dem Backpropagation-Algorithmus ein verbreitetes Lernverfahren für neuronale Netze erklären zu können.
Das Kapitel 4 dient als Einführung in die Neuroevolution. Dazu wurde vom Verfasser der Stand der Forschung in diesem Bereich erfasst und aufgearbeitet. Diesbezüglich hat das Kapitel zunächst die Entwicklung des Forschungsgebietes behandelt und in diesem Kontext den NEAT-Algorithmus eingeführt und die Skalierbarkeit von NE-Techniken diskutiert. Die Teilgebiete der Diversität und Neuheit und der indirekten Kodierung wurden im Kontext der Neuroevolution ebenfalls näher beschrieben, wobei einzelne Algorithmen bzw. Modelle näher erläutert wurden.
Das 5. Kapitel stellt das eigentliche Herzstück dieser Arbeit dar. Angefangen mit der Kernidee, wurde in das Intra-Life-Learning eingeführt. In diesem Zusammenhang hat der Verfasser bestimmte Begrifflichkeiten definiert und Probleme aufgezeigt, die im Kontext des Intra-Life-Learning von Relevanz sind. Daraus ergebend

wurde zunächst das Meta-Lernen formalisiert, da ein konsistentes Verständnis dieser Thematik sichergestellt werden musste. Anschließend hat der Verfasser das Intra-Life-Learning in verschiedene Lernszenarien unterteilt, um es in die bestehende Forschungslandschaft einordnen zu können. Dabei hat der Verfasser zusätzlich einzelne Anforderung im Kontext der Lernszenarien definiert. Basierend darauf konnte der Aufbau der Evaluation, welche Bestandteil des restlichen Kapitels ist, beschrieben werden. Diesbezüglich wurden eigene Vergleichskriterien definiert, die sich aus den Anforderungen der einzelnen Lernszenarien ergeben. Die anschließenden Unterkapitel haben die einzelnen Ansätze mit ihren Techniken beschrieben und miteinander verglichen. Dabei ist herausgekommen, dass die Ansätze jeweils für ein bestimmtes Lernszenario konzipiert sind und nur einzelne Teilprobleme des Intra-Life-Learning adressieren können. Zusätzlich hat der Autor für die einzelnen Techniken, wie auch Ansätze, Forschungspotenziale identifiziert und aufgezeigt.
In Kapitel 6 wurde mit NeRO schließlich ein Algorithmus vorgestellt, welcher vom Autor selbst entwickelt wurde, um den schon erwähnten Problemen bzw. Anforderungen des Intra-Life-Learnings begegnen zu können. Dabei wurden die in Kapitel 5 gesammelten Erkenntnisse aufgegriffen, um zum einen die Problemstellung des Intra-Life-Learnings weiter zu präzisieren und zum anderen die einzelnen Lernszenarien in einen Zusammenhang zu bringen. Zusätzlich wurden mit der CLS-Theorie und dem Baldwin-Effekt zwei weitere Ansätze eingeführt, die bei der Konzeption von NeRO berücksichtigt werden sollten. Schließlich wurden die bisherigen Erkenntnisse, Ansätze und verschiedenen Lernszenarien in einen Zusammenhang gebracht, woraus der NeRO-Algorithmus resultierte, der in Unterkapitel 6.3 beschrieben wird. Darüber hinaus wurde in Unterkapitel 6.4 beschrieben, wie der Algorithmus parallelisiert werden kann, um ihn somit auf die Ressourcen eines Computer-Clusters anpassen zu können.
In Bezug auf die eigentliche Zielstellung der Arbeit kann schlussendlich festgehalten werden, dass bisher kein Algorithmus existiert, der dem Intra-Life-Learning mit seinen unterschiedlichen Lernszenarien begegnen kann. Mit NeRO wurde jedoch ein Algorithmus entwickelt, der das Potenzial hat, diese Lücke zu schließen. Allerdings lässt sich auch in diesem Zusammenhang weiteres Forschungspotenzial identifizieren, welches in den nächsten Abschnitten abschließend erläutert werden soll.

Zunächst wäre eine Performance-Evaluation des NeRO-Algorithmus interessant. Dazu müsste der Algorithmus prototypisch implementiert werden. Im Zusammenhang mit der qualitativen Evaluation des Algorithmus ergibt sich ebenfalls Forschungspotenzial, welches ausgeschöpft werden könnte. So ist es der Fall, dass die bisherigen Testumgebungen, von z. B. Algorithmen des Meta-Lernens, unzulänglich für das Intra-Life-Learning sind. Eine geeignete Testumgebung für NeRO zu definieren, welche die Kriterien des kontinuierlichen und adaptiven Lernens erfüllt, stellt somit eine weitere Herausforderung beim Intra-Life-Learning dar.
Ein anderer Punkt, der im Zusammenhang mit NeRO offen bleibt, ist die Architektur seines Systems. Wenn ein intelligenter Agent der beispielsweise den NeRO-Algorithmus verwendet, in seiner jeweiligen Umgebung agiert, müsste genau spezifiziert sein, wann die Handlung des Agenten für ein Lernen unterbrochen werden würde. Zusätzlich müsste ebenfalls deklariert werden, zu welchem Zeitpunkt welche Art von Lernen stattfinden soll. An dieser Stelle stellt die CLS-Theorie einen Ansatz dar, welcher dahingehend näher analysiert werden sollte.
Weitere Potenziale für NeRO ergeben sich auch im Zusammenhang mit seinem Neokortex. So wäre eine Idee, die diffusions-basierte Neuromodulation mitzuverwenden. Bisher werden die neuromodulativen Signale für jedes Neuron einzeln durch das NMN gesteuert. Es könnten jedoch auch diffundierende Neuronen zum Basis-Modell hinzugefügt werden, welche ihr neuromodulatives Signal an die Neuronen im näheren Umkreis weitergeben. Dabei müsste allerdings eine Möglichkeit gefunden werden, ihre Position und Anzahl wie auch den Radius der Diffusion als Parameter zu modellieren, die vom Meta-Optimierer zusätzlich berücksichtigt werden können.
Auch im Zusammenhang mit dem Hippocampus ergeben sich Potenziale, die noch ausgeschöpft werden könnten. So ist die Erforschung von anderen Lern- und Speichermechanismen ein interessanter Punkt. Nach der CLS-Theorie speichert der Hippocampus sein Wissen auf eine muster-separierte Weise. Ein ähnliches Phänomen kann bei den *Self-Organizing-Maps* beobachtet werden. Diese Variante von neuronalen Netzen im Kontext des schnellen Lernens im Hippocampus zu verwenden, wäre mit Sicherheit ein interessanter Aspekt für die weitere Forschung auf diesem Gebiet.

Literatur

[AA19] Antreas Antoniou und Amos Storkey. *Learning to learn via Self-Critique*. 2019. URL: http://arxiv.org/pdf/1905.10295.

[Aal+09] T. Aaltonen u. a. "Measurement of the top-quark mass with dilepton events selected using neuroevolution at CDF". eng. In: *Physical review letters* 102.15 (2009). Journal Article, S. 152001. ISSN: 0031-9007.

[AGI14] Alex Graves, Greg Wayne und Ivo Danihelka. "Neural Turing Machines". In: *ArXiv* abs/1410.5401 (2014). URL: http://arxiv.org/pdf/1410.5401.

[Alj19] Rahaf Aljundi. *Continual Learning in Neural Networks*. PhD Thesis, Supervisor: Tinne Tuytelaars. 2019. URL: https://arxiv.org/pdf/1910.02718.

[Alp19] Ethem Alpaydin. *Maschinelles Lernen*. De Gruyter Oldenbourg, 2019. ISBN: 9783110617894. DOI: 10.1515/9783110617894.

[And+16] Marcin Andrychowicz u. a. *Learning to learn by gradient descent by gradient descent*. 2016. URL: http://arxiv.org/pdf/1606.04474.

[And+18] Andrei A. Rusu u. a. *Meta-Learning with Latent Embedding Optimization*. 2018. URL: http://arxiv.org/pdf/1807.05960.

[Bal96] James Mark Baldwin. "A New Factor in Evolution". In: *The American Naturalist* 30.354 (1896).

[Bea+20] Shawn Beaulieu u. a. "Learning to Continually Learn". In: *European Conference on Artificial Intelligence - ECAI* (2020). URL: https://arxiv.org/pdf/2002.09571.

[Bro+17] Andrew Brock u. a. *SMASH: One-Shot Model Architecture Search through HyperNetworks*. 2017. URL: http://arxiv.org/pdf/1708.05344.

[BS17] Jonathan C. Brant und Kenneth O. Stanley. "Minimal criterion coevolution". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17: Genetic and Evolutionary Computation Conference (Berlin Germany,). Hrsg. von Peter A. N. Bosman. New York, NY, USA: ACM, 2017, S. 67–74. ISBN: 9781450349208. DOI: 10.1145/3071178.3071186.

[BSC18] Yogesh Balaji, Swami Sankaranarayanan und Rama Chellappa. "MetaReg: Towards Domain Generalization using Meta-Regularization". In: *Advances in Neural Information Processing Systems*. Hrsg. von S. °Bengio u. a. 2018, S. 998–1008. URL: https://papers.nips.cc/paper/7378-metareg-towards-domain-generalization-using-meta-regularization.

[Cau47] Augustin Cauchy. "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comp. Rend. Sci. Paris* 25.1847 (1847), S. 536–538.

[CL11] Jeff Clune und Hod Lipson. "Evolving 3D objects with a generative encoding inspired by developmental biology". In: *ACM SIGEVOlution* 5.4 (2011), S. 2–12. ISSN: 1931-8499. DOI: 10.1145/2078245.2078246.

[CL18a] Zhiyuan Chen und Bing Liu. *Lifelong Machine Learning. Zhiyuan Chen (Google), Bing Liu (University of Illinois at Chicago)*. eng. Second edition. Bd. #38. Synthesis Lectures on Artificial Intelligence and Machine Learning. Chen, Zhiyuan (VerfasserIn) Liu, Bing (VerfasserIn) Chen, Zhiyuan (VerfasserIn) Liu, Bing (VerfasserIn). San Rafael: Morgan & Claypool Publishers, 2018. 209 S. ISBN: 9781681733036. URL: https://www.cs.uic.edu/~liub/lifelong-machine-learning-draft.pdf.

[CL18b] David W. Corne und Michael A. Lones. "Evolutionary Algorithms". In: 1.3 (2018). To appear in R. Marti, P. Pardalos, and M. Resende, eds., Handbook of Heuristics, Springer, S. 1–22. DOI: 10.1007/978-3-319-07153-4_27-1. URL: http://arxiv.org/pdf/1805.11014v1.

[CML13] Jeff Clune, Jean-Baptiste Mouret und Hod Lipson. "The evolutionary origins of modularity". eng. In: *Proceedings. Biological sciences* 280.1755 (2013). Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S., S. 20122863. DOI: 10.1098/rspb.2012.2863. eprint: 23363632.

[Con+18] Edoardo Conti u. a. "Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents". In: *Advances in neural information processing systems*. 2018, S. 5027–5038.

[Deb+02] Kalyanmoy Deb u. a. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), S. 182–197. ISSN: 1089-778X. DOI: 10.1109/4235.996017.

[Dev+19] Jacob Devlin u. a. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (2019), S. 4171–4186. DOI: 10.18653/v1/N19-1423.

[DHS11] John C. Duchi, Elad Hazan und Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12 (2011), S. 2121–2159. ISSN: 1532-4435.

[DZR12] James J. DiCarlo, Davide Zoccolan und Nicole C. Rust. "How does the brain solve visual object recognition?" eng. In: *Neuron* 73.3 (2012). Journal Article Research Support, N.I.H., Extramural Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S., S. 415–434. DOI: 10.1016/j.neuron.2012.01.010. eprint: 22325196.

[EMC15] Kai Olav Ellefsen, Jean-Baptiste Mouret und Jeff Clune. "Neural modularity helps organisms evolve to learn new skills without forgetting old skills". eng. In: *PLoS computational biology* 11.4 (2015). Journal Article Research Support, Non-U.S. Gov't Journal Article Research Support,

Non-U.S. Gov't, e1004128. DOI: 10.1371/journal.pcbi.1004128. eprint: 25837826.

[FAL17] Chelsea Finn, Pieter Abbeel und Sergey Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *ArXiv* abs/1703.03400 (2017). URL: http://arxiv.org/pdf/1703.03400v3.

[Fer+16] Chrisantha Fernando u. a. "Convolution by Evolution". In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16*. Genetic and Evolutionary Computation Conference - GECCO '16 (Denver, Colorado, USA,). Hrsg. von Frank Neumann, Tobias Friedrich und Andrew M. Sutton. New York, New York, USA: ACM Press, 2016, S. 109–116. ISBN: 9781450342063. DOI: 10.1145/2908812.2908890.

[Fer+18] Chrisantha Thomas Fernando u. a. *Meta-Learning by the Baldwin Effect*. 2018. DOI: 10.1145/3205651.3205763. URL: https://arxiv.org/pdf/1806.07917.

[Fin+19] Chelsea Finn u. a. *Online Meta-Learning*. 2019. URL: http://arxiv.org/pdf/1902.08438.

[FL18] Chelsea Finn und Sergey Levine. "Meta-Learning and Universality: Deep Representations and Gradient Descent can Approximate any Learning Algorithm". In: *ICLR 2018* (2018). URL: https://arxiv.org/pdf/1710.11622.

[Fre99] Robert French. "Catastrophic forgetting in connectionist networks". In: *Trends in Cognitive Sciences* 3.4 (1999). PII: S1364-6613(99)01294-2, S. 128–135. ISSN: 1364-6613. DOI: 10.1016/S1364-6613(99)01294-2. URL: http://www.sciencedirect.com/science/article/pii/S1364661399012942.

[GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep learning*. eng. Goodfellow, Ian (VerfasserIn) Bengio, Yoshua (VerfasserIn) Courville, Aaron (VerfasserIn). Cambridge, Massachusetts und London, England: MIT press, 2016. 785 S. ISBN: 9780262035613. URL: http://www.deeplearningbook.org/.

[GLY16] Daniele Gravina, Antonios Liapis und Georgios Yannakakis. “Surprise Search”. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO ’16*. Genetic and Evolutionary Computation Conference - GECCO ’16 (Denver, Colorado, USA,). Hrsg. von Frank Neumann, Tobias Friedrich und Andrew M. Sutton. New York, New York, USA: ACM Press, 2016, S. 677–684. ISBN: 9781450342063. DOI: 10.1145/2908812.2908817.

[GM19] Santiago Gonzalez und Risto Miikkulainen. “Improved Training Speed, Accuracy, and Data Utilization Through Loss Function Optimization”. In: *ArXiv* abs/1905.11528 (2019). URL: http://arxiv.org/pdf/1905.11528.

[Goo+14] Ian J. Goodfellow u. a. “Generative Adversarial Nets”. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS’14. Cambridge, MA, USA: MIT press, 2014, S. 2672–2680. URL: https://papers.nips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

[GR87] David E. Goldberg und Jon Richardson. “Genetic Algorithms with Sharing for Multimodal Function Optimization”. In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. USA: L. Erlbaum Associates Inc, 1987, S. 41–49. ISBN: 0805801588.

[GSS19] Josif Grabocka, Randolf Scholz und Lars Schmidt-Thieme. “Learning Surrogate Losses”. In: (2019). URL: https://openreview.net/forum?id=BkePHaVKwS.

[HCM14] Joost Huizinga, Jeff Clune und Jean-Baptiste Mouret. “Evolving neural networks that are both modular and regular”. In: *Proceedings of the 2014 conference on Genetic and evolutionary computation - GECCO ’14*. the 2014 conference (Vancouver, BC, Canada,). Hrsg. von Dirk V. Arnold. New York, New York, USA: ACM Press, 2014, S. 697–704. ISBN: 9781450326629. DOI: 10.1145/2576768.2598232.

[HDL16] David Ha, Andrew Dai und Quoc V. Le. "HyperNetworks". In: *arXiv preprint arXiv:1609.09106* (2016). URL: http://arxiv.org/pdf/1609.09106.

[He+16] Kaiming He u. a. "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, S. 770–778. ISBN: 1063-6919. DOI: 10.1109/CVPR.2016.90.

[Heb49] Donald Olding Hebb. *The organization of behavior: a neuropsychological theory*. J. Wiley und Chapman & Hall, 1949.

[Her09] Suzana Herculano-Houzel. "The human brain in numbers: a linearly scaled-up primate brain". eng. In: *Frontiers in human neuroscience* 3 (2009). Journal Article, S. 31. DOI: 10.3389/neuro.09.031.2009. eprint: 19915731.

[HMC16] Joost Huizinga, Jean-Baptiste Mouret und Jeff Clune. "Does Aligning Phenotypic and Genotypic Modularity Improve the Evolution of Neural Networks?" In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16*. Genetic and Evolutionary Computation Conference - GECCO '16 (Denver, Colorado, USA,). Hrsg. von Frank Neumann, Tobias Friedrich und Andrew M. Sutton. New York, New York, USA: ACM Press, 2016, S. 125–132. ISBN: 9781450342063. DOI: 10.1145/2908812.2908836.

[HN87] Geoffrey E. Hinton und Steven J. Nowlan. "How Learning Can Guide Evolution". In: *Complex Systems* 1 (1987), S. 495–502. URL: https://bit.ly/34FiChy.

[HO96] Nikolaus Hansen und Andreas Ostermeier. "Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation". In: *Proceedings of IEEE International Conference on Evolutionary Computation*. IEEE, 1996, S. 312–317. DOI: 10.1109/ICEC.1996.542381.

[Hor+00] G. S. Hornby u. a. "Evolving robust gaits with AIBO". In: *Robotics and Automation, 2000 IEEE International Conference*. 2000 ICRA. IEEE International Conference on Robotics and Automation (San Francisco,

CA, USA,). IEEE Robotics and Automation Society Staff. Piscataway: IEEE, 2000, S. 3040–3045. ISBN: 0-7803-5886-4. DOI: 10.1109/ROBOT.2000.846489.

[Hos+20] Timothy M. Hospedales u. a. "Meta-Learning in Neural Networks: A Survey". In: *ArXiv* abs/2004.05439 (2020). URL: http://arxiv.org/pdf/2004.05439v1.

[HP87] Geoffrey E. Hinton und David C. Plaut. "Using Fast Weights to Deblur Old Memories". In: *Conference Of The Cognitive Science Society* (1987).

[HS97] Sepp Hochreiter und Jürgen Schmidhuber. "Long short-term memory". eng. In: *Neural computation* 9.8 (1997). Journal Article Research Support, Non-U.S. Gov't, S. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735). eprint: 9377276.

[HSM16] Babak Hodjat, Hormoz Shahrzad und Risto Miikkulainen. "Distributed Age-Layered Novelty Search". In: *Proceedings of the Artificial Life Conference 2016*. Artificial Life Conference 2016 (Cancun, Mexico,). Cambridge, MA: MIT press, 2016, S. 131–138. ISBN: 978-0-262-33936-0. DOI: 10.7551/978-0-262-33936-0-ch027.

[Jon75] Kenneth Alan de Jong. "An Analysis of the Behavior of a Class of Genetic Adaptive Systems". AAI7609381. USA, 1975.

[JW19] Khurram Javed und Martha White. "Meta-Learning Representations for Continual Learning". In: *Advances in Neural Information Processing Systems 32*. Hrsg. von H. Wallach u. a. Vancouver, Canada: Curran Associates, Inc, 2019, S. 1820–1830. URL: http://papers.nips.cc/paper/8458-meta-learning-representations-for-continual-learning.pdf.

[KB15] Diederik P. Kingma und Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* 3 (2015). URL: https://arxiv.org/pdf/1412.6980.

[KHM16] Dharshan Kumaran, Demis Hassabis und James L. McClelland. “What Learning Systems do Intelligent Agents Need? Complementary Learning Systems Theory Updated”. eng. In: *Trends in Cognitive Sciences* 20.7 (2016). Journal Article Review, S. 512–534. ISSN: 1879-307X. DOI: 10.1016/j.tics.2016.05.004. eprint: 27315762. URL: https://www.cell.com/trends/cognitive-sciences/fulltext/S1364-6613(16)30043-2?_returnURL=https%3A%2F%2Flinkinghub.elsevier.com%2Fretrieve%2Fpii%2FS1364661316300432%3Fshowall%3Dtrue.

[Kru+15] Rudolf Kruse u. a. *Computational Intelligence*. Wiesbaden: Springer Fachmedien Wiesbaden, 2015. ISBN: 978-3-658-10903-5. DOI: 10.1007/978-3-658-10904-2.

[Lak+11] Brenden M. Lake u. a. *One shot learning of simple visual concepts*. 2011. URL: https://cims.nyu.edu/~brenden/LakeEtAl2011CogSci.pdf.

[LBH15] Yann LeCun, Yoshua Bengio und Geoffrey Hinton. “Deep learning”. eng. In: *Nature* 521.7553 (2015). Journal Article Research Support, Non-U.S. Gov’t Research Support, U.S. Gov’t, Non-P.H.S. Review, S. 436–444. DOI: 10.1038/nature14539. eprint: 26017442.

[Leh+18] Joel Lehman u. a. *Safe Mutations for Deep and Recurrent Neural Networks through Output Gradients*. 2018. URL: http://arxiv.org/pdf/1712.06563v3.

[Li+17] Zhenguo Li u. a. *Meta-SGD: Learning to Learn Quickly for Few-Shot Learning*. 2017. URL: http://arxiv.org/pdf/1707.09835.

[Li+19] Yiying Li u. a. “Feature-Critic Networks for Heterogeneous Domain Generalization”. In: *Proceedings of the 36th International Conference on Machine Learning* (Long Beach, California, USA). Hrsg. von Kamalika Chaudhuri und Ruslan Salakhutdinov. Bd. 97. Proceedings of Machine Learning Research. PMLR, 2019, S. 3915–3924. URL: http://arxiv.org/pdf/1901.11448.

[LL20] Jack Lindsey und Ashok Litwin-Kumar. "Learning to Learn with Feedback and Local Plasticity". In: *ArXiv* abs/2006.09549 (2020). URL: https://arxiv.org/pdf/2006.09549.pdf.

[LP00] Hod Lipson und Jordan B. Pollack. "Automatic design and manufacture of robotic lifeforms". eng. In: *Nature* 406.6799 (2000). Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S., S. 974–978. DOI: 10.1038/35023115. eprint: 10984047.

[LRU20] Jure Leskovec, Anand Rajaraman und Jeffrey David Ullman. *Mining of Massive Datasets*. 3. Aufl. Cambridge university press, 2020. ISBN: 9781108684163. DOI: 10.1017/9781108684163.

[LS11] Joel Lehman und Kenneth O. Stanley. "Abandoning objectives: evolution through the search for novelty alone". eng. In: *Evolutionary computation* 19.2 (2011). Journal Article, S. 189–223. ISSN: 1063-6560. DOI: 10.1162/EVCO_a_00025. eprint: 20868264.

[Mah+13] Maryam Mahsal Khan u. a. "Fast learning neural networks using Cartesian genetic programming". In: *Neurocomputing* 121 (2013), S. 274–289. ISSN: 09252312. DOI: 10.1016/j.neucom.2013.04.005.

[Mar18] Gary Marcus. "Deep Learning: A Critical Appraisal". In: *ArXiv* abs/1801.00631 (2018). URL: https://arxiv.org/pdf/1801.00631.

[MC15] Jean-Baptiste Mouret und Jeff Clune. "Illuminating search spaces by mapping elites". In: *arXiv preprint arXiv:1504.04909* (2015).

[MD12] J-B Mouret und S. Doncieux. "Encouraging behavioral diversity in evolutionary robotics: an empirical study". eng. In: *Evolutionary computation* 20.1 (2012). Journal Article, S. 91–133. ISSN: 1063-6560. DOI: 10.1162/EVCO_a_00048. eprint: 21838553.

[Mei82] Hans Meinhardt. *Models of Biological Pattern Formation (Academic Press, London, 1982)*. 1982.

[Mic+19] Thomas Miconi u. a. "Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity". In: *International Conference on Learning Representations*. 2019. URL: https://arxiv.org/pdf/2002.10585.pdf.

[Mic16] Thomas Miconi. "Learning to learn with backpropagation of Hebbian plasticity". In: *arXiv: Neural and Evolutionary Computing* (2016).

[MLC16] Henok Mengistu, Joel Lehman und Jeff Clune. "Evolvability Search". In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16*. Genetic and Evolutionary Computation Conference - GECCO '16 (Denver, Colorado, USA,). Hrsg. von Frank Neumann, Tobias Friedrich und Andrew M. Sutton. New York, New York, USA: ACM Press, 2016, S. 141–148. ISBN: 9781450342063. DOI: 10.1145/2908812.2908838.

[MM17] Elliot Meyerson und Risto Miikkulainen. "Discovering evolutionary stepping stones through behavior domination". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '17: Genetic and Evolutionary Computation Conference (Berlin Germany,). Hrsg. von Peter A. N. Bosman. New York, NY, USA: ACM, 2017, S. 139–146. ISBN: 9781450349208. DOI: 10.1145/3071178.3071315.

[MMO95] James L. McClelland, Bruce L. McNaughton und Randall C. O'Reilly. "Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory". eng. In: *Psychological review* 102.3 (1995). Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, Non-P.H.S. Research Support, U.S. Gov't, P.H.S. Review, S. 419–457. ISSN: 0033-295X. DOI: 10.1037/0033-295X.102.3.419. eprint: 7624455. URL: https://pubmed.ncbi.nlm.nih.gov/7624455/.

[MP43] Warren S. McCulloch und Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Bio-*

physics 5.4 (1943). PII: BF02478259, S. 115–133. ISSN: 0007-4985. DOI: 10.1007/BF02478259.

[MPB17] Marvin Minsky, Seymour A. Papert und Léon Bottou. *Perceptrons. An Introduction to Computational Geometry*. eng. The MIT Press Ser. Cambridge: MIT press, 2017. 317 S. ISBN: 9780262343930.

[MSC18] Thomas Miconi, Kenneth Stanley und Jeff Clune. "Differentiable plasticity: training plastic neural networks with backpropagation". In: Hrsg. von Jennifer Dy und Andreas Krause. Bd. 80th Proceedings of Machine Learning Research. 10–15 Jul. Stockholmsmässan, Stockholm Sweden: PMLR, 2018, S. 3559–3568. URL: https://arxiv.org/pdf/1804.02464.pdf.

[MY17] Tsendsuren Munkhdalai und Hong Yu. "Meta Networks". In: *Proceedings of machine learning research* 70 (2017), S. 2554–2563. URL: http://arxiv.org/pdf/1703.00837.

[NR20] Elias Najarro und Sebastian Risi. "Meta-Learning through Hebbian Plasticity in Random Networks". In: *ArXiv* abs/2007.02686 (2020). URL: https://arxiv.org/pdf/2007.02686.

[NYC16] A. Nguyen, J. Yosinski und J. Clune. "Understanding Innovation Engines: Automated Creativity and Improved Stochastic Optimization via Deep Learning". eng. In: *Evolutionary computation* 24.3 (2016). Journal Article, S. 545–572. ISSN: 1063-6560. DOI: 10.1162/EVCO_a_00189. eprint: 27367139.

[Oja08] Erkki Oja. "Oja learning rule". In: *Scholarpedia* 3.3 (2008). PII: 3612, S. 3612. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.3612. URL: http://www.scholarpedia.org/article/Oja_learning_rule.

[Qia+18] Siyuan Qiao u. a. "Few-Shot Image Recognition by Predicting Parameters from Activations". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018). Jun. DOI: 10.1109/cvpr.2018.00755. URL: http://arxiv.org/pdf/1706.03466.

[Raj+19] Aravind Rajeswaran u. a. "Meta-Learning with Implicit Gradients". In: *Advances in Neural Information Processing Systems 32*. Hrsg. von H. Wallach u. a. NeurIPS 2019. First two authors contributed equally. Curran Associates, Inc, 2019. URL: https://arxiv.org/pdf/1909.04630.

[Rei+18] Rein Houthooft u. a. "Evolved Policy Gradients". In: *ArXiv* abs/1802.04821 (2018). URL: http://arxiv.org/pdf/1802.04821.

[RL17] Sachin Ravi und Hugo Larochelle. "Optimization as a Model for Few-Shot Learning". In: *International Conference on Learning Representations*. 2017.

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain". eng. In: *Psychological review* 65.6 (1958). Journal Article, S. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. eprint: 13602029.

[Ros62] Frank Rosenblatt. "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms". In: (1962).

[RS10] Sebastian Risi und Kenneth O. Stanley. "Indirectly Encoding Neural Plasticity as a Pattern of Local Rules". In: *From Animals to Animats 11*. Hrsg. von David Hutchison u. a. Bd. 6226. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 533–543. ISBN: 978-3-642-15192-7. DOI: 10.1007/978-3-642-15193-4_50.

[RS12] Sebastian Risi und Kenneth O. Stanley. "A unified approach to evolving plasticity and neural geometry". In: *The 2012 International Joint Conference on Neural Networks (IJCNN)*. 2012 International Joint Conference on Neural Networks (IJCNN 2012 - Brisbane) (Brisbane, Australia,). IEEE, 2012, S. 1–8. ISBN: 978-1-4673-1490-9. DOI: 10.1109/IJCNN.2012.6252826.

[Rud17] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *ArXiv: 1609.04747* (2017). Added derivations of AdaMax and Nadam. URL: https://arxiv.org/pdf/1609.04747.

[Sal+17] Tim Salimans u. a. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. 2017. URL: http://arxiv.org/pdf/1703.03864v2.

[San+16] Adam Santoro u. a. "Meta-Learning with Memory-Augmented Neural Networks". In: *International Conference on Machine Learning*. 2016. URL: http://proceedings.mlr.press/v48/santoro16.pdf.

[SC18] Christopher Stanton und Jeff Clune. *Deep Curiosity Search: Intra-Life Exploration Can Improve Performance on Challenging Deep Reinforcement Learning Problems*. 2018. URL: http://arxiv.org/pdf/1806.00553v3.

[Sch87] Jürgen Schmidhuber. "Evolutionary Principles in Self-referential Learning, or on Learning How to Learn: The Meta-meta-...Hook." PhD thesis. Technische Universität München, 1987.

[SDG09] Kenneth O. Stanley, David B. D'Ambrosio und Jason Gauci. "A hypercube-based encoding for evolving large-scale neural networks". eng. In: *Artificial life* 15.2 (2009). Journal Article, S. 185–212. ISSN: 1064-5462. DOI: 10.1162/artl.2009.15.2.15202. eprint: 19199382.

[Sec+11] Jimmy Secretan u. a. "Picbreeder: a case study in collaborative evolutionary exploration of design space". eng. In: *Evolutionary computation* 19.3 (2011). Journal Article, S. 373–403. ISSN: 1063-6560. DOI: 10.1162/EVCO_a_00030. eprint: 20964537.

[Shi+17] Hanul Shin u. a. *Continual Learning with Deep Generative Replay*. NIPS 2017. 2017. URL: https://arxiv.org/pdf/1705.08690.

[Sil+16] David Silver u. a. "Mastering the game of Go with deep neural networks and tree search". eng. In: *Nature* 529.7587 (2016). Journal Article, S. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. eprint: 26819042.

[SM02] Kenneth O. Stanley und Risto Miikkulainen. "Evolving neural networks through augmenting topologies". eng. In: *Evolutionary computation* 10.2 (2002). Journal Article Research Support, Non-U.S. Gov't

Research Support, U.S. Gov't, Non-P.H.S., S. 99–127. ISSN: 1063-6560. DOI: 10.1162/106365602320169811. eprint: 12180173.

[Son+20] Xingyou Song u. a. "ES-MAML: Simple Hessian-Free Meta Learning". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/pdf?id=S1exA2NtDB.

[SSR18] Andrea Soltoggio, Kenneth O. Stanley und Sebastian Risi. "Born to Learn: the Inspiration, Progress, and Future of Evolved Plastic Artificial Neural Networks". In: *Neural Networks* 108 (2018). Neural Networks, 2018, S. 48–67. ISSN: 08936080. DOI: 10.1016/j.neunet.2018.07.013. URL: http://arxiv.org/pdf/1703.10371v3.

[Sta+19] Kenneth O. Stanley u. a. "Designing neural networks through neuroevolution". In: *Nature Machine Intelligence* 1.1 (2019). PII: 6, S. 24–35. DOI: 10.1038/s42256-018-0006-z.

[Sta07] Kenneth O. Stanley. "Compositional pattern producing networks: A novel abstraction of development". In: *Genetic Programming and Evolvable Machines* 8.2 (2007). PII: 9028, S. 131–162. ISSN: 1389-2576. DOI: 10.1007/s10710-007-9028-8.

[Suc+17] Felipe Petroski Such u. a. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*. 2017.

[Sum18] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Hrsg. von medium.com. 2018. URL: https://perma.cc/8XT7-K42V (besucht am 21. 07. 2020).

[Sun+17] Flood Sung u. a. "Learning to Learn: Meta-Critic Networks for Sample Efficient Learning". In: *ArXiv* abs/1706.09529 (2017). URL: http://arxiv.org/pdf/1706.09529.

[Sun+19] Qianru Sun u. a. "Meta-Transfer Learning for Few-Shot Learning". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2019). Jun. DOI: 10.1109/cvpr.2019.00049. URL: http://arxiv.org/pdf/1812.02391.

[TM13] Andrew James Turner und Julian Francis Miller. "Cartesian genetic programming encoded artificial neural networks". In: *Proceeding of the fifteenth annual conference on Genetic and Evolutionary Computation Conference*. Proceeding of the fifteenth annual conference (Amsterdam, The Netherlands,). Hrsg. von Enrique Alba und Christian Blum. ACM Special Interest Group on Genetic and Evolutionary Computation und GECCO 13. New York, NY: ACM, 2013, S. 1005. ISBN: 9781450319638. DOI: 10.1145/2463372.2463484.

[TMT20] Vithursan Thangarasa, Thomas Miconi und Graham W. Taylor. "Enabling Continual Learning with Differentiable Hebbian Plasticity". In: *ArXiv* abs/2006.16558 (2020). URL: https://arxiv.org/pdf/2006.16558.pdf.

[TS10] Lisa Torrey und Jude Shavlik. "Transfer Learning". In: *Handbook of Research on Machine Learning Applications and Trends*. Hrsg. von Emilio Soria Olivas u. a. IGI Global, 2010, S. 242–264. ISBN: 9781605667669. DOI: 10.4018/978-1-60566-766-9.ch011.

[TTM19] Vithursan Thangarasa, Graham W. Taylor und Thomas Miconi. "Differentiable Hebbian Plasticity for Continual Learning". In: *Proceedings of the 1st Adaptive & Multitask Learning Workshop* (Long Beach, California). 2019. URL: https://openreview.net/pdf?id=r1x-E5Ss34.

[Utg+11] Paul E. Utgoff u. a. "Inductive Transfer". In: *Encyclopedia of machine learning*. Hrsg. von Claude Sammut. Springer reference. New York, NY: Springer, 2011, S. 545–548. ISBN: 978-0-387-30768-8. DOI: 10.1007/978-0-387-30164-8_401.

[Vas+11] Zlatko Vasilkoski u. a. "Review of stability properties of neural plasticity rules for implementation on memristive neuromorphic hardware". In: *The International Joint Conference on Neural Networks*. 2011, S. 2563–2569. ISBN: 2161-4407. DOI: 10.1109/IJCNN.2011.6033553.

[VC17] Roby Velez und Jeff Clune. "Diffusion-based neuromodulation can eliminate catastrophic forgetting in simple neural networks". In: *PloS one*

12.11 (2017), e0187736. DOI: 10.1371/journal.pone.0187736. URL: http://arxiv.org/pdf/1705.07241v3.

[Vec+19] Nicolas Vecoven u. a. "Cellular neuromodulation in artificial networks". In: *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)* (Vancouver, Canada). 2019. URL: https://bit.ly/35JS4uF.

[Ven+01] J. C. Venter u. a. "The sequence of the human genome". eng. In: *Science* 291.5507 (2001). Journal Article Research Support, Non-U.S. Gov't Journal Article Research Support, Non-U.S. Gov't, S. 1304–1351. ISSN: 0036-8075. DOI: 10.1126/science.1058040. eprint: 11181995.

[Vin+16] Oriol Vinyals u. a. *Matching Networks for One Shot Learning*. 2016. URL: https://arxiv.org/pdf/1606.04080.

[Vuo+18] Risto Vuorio u. a. "Meta Continual Learning". In: *ArXiv* abs/1806.06928 (2018). URL: https://arxiv.org/pdf/1806.06928.

[Wen+18] Haiguang Wen u. a. "Neural Encoding and Decoding with Deep Learning for Dynamic Natural Vision". In: *Cerebral Cortex* 28.12 (2018). Cerebral Cortex. 2017 pp.1-25 27 pages, 10 figures, 1 table, S. 4136–4160. ISSN: 1047-3211. DOI: 10.1093/cercor/bhx268. URL: http://arxiv.org/pdf/1608.03425v2.

[WH60] Bernard Widrow und Marcian E. Hoff. "Adaptive switching circuits". In: *IRE WESCON Convention Record*. Bd. 4. 1960, S. 96–104.

[Wic+17] Olga Wichrowska u. a. *Learned Optimizers that Scale and Generalize*. 2017. URL: http://arxiv.org/pdf/1703.04813.

[WO19] Lin Wang und Jeff Orchard. "Investigating the Evolution of a Neuroplasticity Network for Learning". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49.10 (2019), S. 2131–2143. ISSN: 2168-2232. DOI: 10.1109/TSMC.2017.2755066.

[WZO20] Lin Wang, Junteng Zheng und Jeff Orchard. "Evolving Generalized Modulatory Learning: Unifying Neuromodulation and Synaptic Plasticity". In: *IEEE Transactions On Cognitive And Developmental Systems*

(2020). DOI: 10.1109/tcds.2019.2960766. URL: https://bit.ly/3e7PvGA.

[YD16] Daniel L. K. Yamins und James J. DiCarlo. "Using goal-driven deep learning models to understand sensory cortex". eng. In: *Nature neuroscience* 19.3 (2016). Journal Article Review, S. 356–365. DOI: 10.1038/nn.4244. eprint: 26906502.

[YS18] Yoonho Lee und Seungjin Choi. *Gradient-Based Meta-Learning with Learned Layerwise Metric and Subspace*. 2018. URL: http://arxiv.org/pdf/1801.05558.

[Zho+20] Wei Zhou u. a. "Online Meta-Critic Learning for Off-Policy Actor-Critic Methods". In: *ArXiv* abs/2003.05334 (2020). URL: http://arxiv.org/pdf/2003.05334.

Abbildungsverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Abkürzungsverzeichnis

NN Neuronales Netz

GPU Grafikprozessor

FFN Feed-Forward-Netz

DNN Deep Neural Network

MSE Mean-Squared-Error

RMSE Root-Mean-Squared-Error

KL-Divergenz Kullback-Leibler-Divergenz

SLP Single-Layer-Perzeptron

MLP Multi-Layer-Perzeptron

CNN Convolutional Neural Network

ReLU Rectified Linear Unit

RNN Rekurrentes Neuronales Netz

BPTT Backpropagation Through Time

LSTM Long-Short-Term-Memory

EA Evolutionärer Algorithmus

GA Genetischer Algorithmus

ES Evolutionäre Strategie

GP Genetische Programmierung

ML Maschinelles Lernen

NE Neuroevolution

CPPN Compositional-pattern-producing-network

SGD Stochastischer Gradientenabstieg

EPANN Evolved plastic neural network

ILL Intra-Life-Learning

NAS Neuronale Architektur-Suche

SGD Stochastic Gradient Descent

ILL Intra-Life-Learning

AL Adaptives Lernen

KL Kontinuierliches Lernen

MTL Multi-Task-Lernen

CLS Complementary Learning Systems

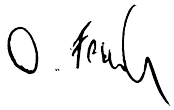
NMN Neuromodulations-Netzwerk

GAN Generative Adversial Network

RDD Resilient Distributed Dataset

Eidesstattliche Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt wurden. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Rostock	16. November 2020	
Ort	Datum	Unterschriften