Universität Stuttgart
Institut für Automatisierungstechnik
und Softwaresysteme

Untersuchung und prototypische Umsetzung eines Lifelong Deep Neural Network Algorithmus

Masterarbeit 3062

An der Universität Stuttgart vorgelegt von
Simon Kamm

Elektro- und Informationstechnik

Prüfer: Prof. Dr.-Ing. Michael Weyrich

Betreuer: Benjamin Maschler, M.Sc.

29.10.2019

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ART	**A**daptive **R**esonance **T**heory
CLS	**C**omplementary **L**earning **S**ystems
CNN	**C**onvolutional **N**eural **N**etworks
DBN	**D**eep **B**elief **N**ets
DGR	**D**eep **G**enerative **R**eplay
DNN	**D**eep **N**eural **N**etwork
EWC	**E**lastic **W**eight **C**onsolidation
FAM	**F**uzzy **A**RT**M**AP
FC	**F**ully **C**onnected
FCN	**F**ully **C**onnected **N**etwork
FLOP	**F**loating **P**oint **Op**eration
FPS	**F**rames **p**er **S**econd
GAN	**G**enerative **A**dversarial **N**etworks
GD	**G**radient **D**escent
GPU	**G**raphical **P**rocessing **U**nit
GUI	**G**raphical **U**ser **I**nterface
iCaRL	**I**ncremental **C**lassifier **a**nd **R**epresentation **L**earning
ILSVRC	**I**mageNet **L**arge **S**cale **V**isual **R**ecognition **C**hallenge
KI	**K**ünstliche **I**ntelligenz
kNN	**k**-**N**earest **N**eighbour
L DNN	**L**ifelong **D**eep **N**eural **N**etwork
LwF	**L**earning **w**ithout **F**orgetting
MLP	**M**ulti-**L**ayer **P**erceptron
MNIST	**M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology
PS	**P**arameter **S**erver

ReLU	**Re**ctifier **L**inear **U**nit
RNN	**R**ecurrent **N**eural **N**etworks
SGD	**S**tochastic **G**radient **D**escent
SVM	**S**upport **V**ector **M**achine
TP	**T**rue **P**ositive
VAE	**V**ariational **A**uto**E**ncoder
VGG	**V**isual **G**eometry **G**roup

Glossar

Backpropagation	Algorithmus, bei dem der Fehler des neuronalen Netzwerks rückwärts durch das Netzwerks propagiert wird, um den Gradienten-Vektor für das nächste Parameterupdate zu erhalten
Edge Device	Ein Endgerät, das hinsichtlich Speicher- und Rechenleistung begrenzt ist, z.B. Mikrocontroller oder Smartphone
Federated Learning	Spezielle Methode des verteilten Lernens, bei dem kein Austausch von Rohdaten notwendig ist
Inference	Einsatz eines DNN in der realen (Test-) Anwendung
Katastrophales Vergessen	Vergessen einer bereits erlernten Aufgabe durch das Erlernen einer neuen Aufgabe
Kontinuierliches Lernen	Bereich des maschinellen Lernens, in dem Modelle kontinuierlich neue Aufgaben erlernen
One-Shot Learning	Fall des kontinuierlichen Lernens, bei dem mithilfe eines einzelnen Samples die Klasse erlernt werden kann
Overfitting	Der Fehler eines DNN auf Basis der Trainingsdaten ist wesentlich geringer im Vergleich zu dem Fehler des DNN auf Testdaten
Post-Training Methode	Methode des verteilten Lernens, bei dem die Zusammenführung der Modelle nach dem abgeschlossenen Training der einzelnen Modelle stattfindet
Verteiltes Lernen	Bereich des maschinellen Lernens, in dem Modelle auf mehr als einem Gerät trainiert werden
Vigilance Parameter	Schwellwert-Parameter innerhalb eines ART-Netzwerks

Zusammenfassung

Der Schutz von eigenen Daten und multitaskingfähige Machine Learning Algorithmen stehen bisher weitestgehend im Widerspruch. Dieser Widerspruch verhindert vielfach eine größere Nutzung von KI-Methoden in der Praxis. In dieser Arbeit wird ein sogenannter Lifelong Deep Neural Network (L DNN) Algorithmus prototypisch implementiert und untersucht, der das Potenzial haben soll diesen Widerspruch zu lösen. Dafür soll dieser Algorithmus kontinuierlich weiterlernen können und verteiltes Lernen ohne den Austausch von Rohdaten ermöglichen. Zudem soll er auch auf Edge Devices laufen, die mit wenig Speicher- und Rechenleistung ausgestattet sind. Dafür werden zunächst die Grundlagen und theoretischen Hintergründe für die relevanten Themen beleuchtet und der L DNN Algorithmus wird mit seinem grundlegenden Aufbau eingeführt und in Relation zu bereits bekannten Algorithmen gebracht. Auf dieser Basis werden ein spezifischer L DNN Algorithmus für die Anforderungen in dieser Arbeit konzipiert und passende Module ausgewählt. Dafür werden unterschiedliche Modelle aus der Literatur mithilfe von definierten Merkmalen verglichen. Mit dieser Auswahl und Konzeption findet die prototypische Umsetzung statt, die im weiteren Verlauf evaluiert wird. Als Daten für die Evaluation dienen die beiden Bilddatensätze MNIST und ImageNet. Kriterien für die Evaluation sind die Klassifikationsgenauigkeit und der Speicherbedarf des Algorithmus. Zunächst wird der Einfluss modellabhängiger Parameter auf die Genauigkeit untersucht. Zudem wird untersucht, welchen Einfluss die Anzahl an verfügbaren Trainingsbilder pro Klasse auf die finale Genauigkeit hat. Weiter werden auf den genannten Datensätzen finale Ergebnisse für das kontinuierliche und verteilte Lernen auf bisher unbekannten Testdaten generiert und mit Algorithmen des kontinuierlichen Lernens in Relation gesetzt. Mit den optimierten Parametern findet schließlich eine finale Auswertung auf Basis des gesamten ImageNet-Datensatzes statt. Diese Ergebnisse dienen zur finalen Analyse und Bewertung des L DNN Algorithmus. Zusätzlich wird im Rahmen dieser Arbeit ein Demonstrator für das inkrementelle Klassen Lernen in Echtzeit, auf einem speicher- und rechenbegrenztem Edge Device, aufgebaut.

Schlüsselwörter: *Künstliche Intelligenz, Maschinelles Lernen, Deep Learning, Deep Neural Networks, Kontinuierliches Lernen, Lebenslanges Lernen, Verteiltes Lernen, Inkrementelles Klassen Lernen, Live-Bild Klassifizierung*

Abstract

The protection of own data and multitasking-capable machine learning algorithms have been contradictory so far. This contradiction often prevents a greater usage of AI methods in practice. In this thesis, a so-called Lifelong Deep Neural Network (L DNN) algorithm is prototypically implemented and evaluated, which should have the potential to solve this contradiction. Therefore, this algorithm should be able to learn continuously and enable distributed learning without the exchange of raw data. In addition, it should also run on edge devices that are equipped with little memory and computing power. First, the basics and theoretical backgrounds for the relevant topics will be introduced and the L DNN algorithm with its basic structure will be described and brought into relation to already known algorithms. On this basis, a specific L DNN algorithm is designed for the requirements of this thesis and suitable modules are selected. Different possible models from the literature will be compared using defined features. With this selection and conception, the prototypical implementation takes place, which will be evaluated in the further course. The two image data sets MNIST and ImageNet serve as data for the evaluation. Criteria for the evaluation is the classification accuracy and the memory requirements of the algorithm. First, the influence of model-dependent parameters on accuracy will be investigated. In addition, the influence of the number of available training images per class on the final accuracy will be investigated. Furthermore, results for continual and distributed learning will be generated on these data sets on previously unknown test data and related to algorithms of continuous learning. With the previous optimized parameters, the final evaluation on the whole ImageNet-Dataset will take place. These results are used for the final analysis and evaluation of the L DNN algorithm. In addition, a demonstrator for incremental class learning in real time will be built on a memory and computational limited edge device.

Key Words: *Artificial Intelligence, Machine Learning, Deep Learning, Deep Neural Networks, Continual Learning, Lifelong Learning, Distributed Learning, Incremental Class Learning, Live-Image Classification*

1 Einleitung

Datenschutz ist heutzutage ein wichtiger Aspekt, der in neuen Anwendungen nicht missachtet werden kann. Sowohl bei persönlichen Daten, die beispielsweise auf einem Smartphone gesammelt werden, als auch bei industriellen Daten (z.B. von einer Produktionsanlage) muss der Schutz dieser Daten gewährleistet werden. Der Schutz dieser Daten steht bisher weitgehend in Widerspruch zu multitaskingfähigen Machine Learning Algorithmen. Durch diesen Widerspruch wird eine flächendeckende Nutzung von KI-Methoden häufig verhindert. Dennoch ist der Wunsch nach einem breiteren Einsatz von KI-Methoden vorhanden, da dadurch viele neue Anwendungen erschlossen werden können oder bestehende Anwendungen weiter verbessert werden können.

Beispielhaft kann die Anwendung „*Predictive Maintenance*“ gesehen werden. Dabei werden auch heute schon Machine Learning Algorithmen eingesetzt, um mögliche Ausfälle von Maschinen vorherzusagen und präventive Instandhaltungsarbeiten zu ermöglichen, die wiederum lange und teure Ausfallzeiten verhindern. Dafür werden vortrainierte neuronale Netze oder andere fixe Machine Learning Algorithmen genutzt. Durch kontinuierlich („*Continual*“) und verteilt („*Distributed*“) lernende Algorithmen könnte der Einsatz von diesen Algorithmen sowie deren Performanz weiter gesteigert werden. Diese Algorithmen sind in der Lage während dem Betrieb kontinuierlich weiter zu lernen und können so auf abweichende Ereignisse reagieren, die vorher nicht bekannt waren und somit nicht erlernt werden konnten. Durch verteiltes Lernen können sich gleiche Maschinen zudem austauschen, wodurch die Information über einen Vorfall 1, den Maschine A gesehen und erlernt hat, an Maschine B weitergegeben werden kann. Wie bereits beschrieben brauchen bisherige Ansätze dafür jedoch den Austausch von Daten sowie die Speicherung dieser Daten, was bei Echtzeit-Anwendungen eine erhebliche Speicher- und Rechenleistung erfordert. Zudem kann es auch schlicht verboten bzw. unerwünscht sein, gesammelte Daten von Maschine A an Maschine B weiterzugeben, wenn diese bei einem Wettbewerber im Einsatz ist. Dasselbe gilt für private Anwendungen wie beispielsweise medizinischen Anwendungen. Mithilfe von gesammelten Daten von Fitnessuhren könnten Netzwerke Krankheiten oder Symptome von Krankheiten frühzeitig erkennen. Jedoch ist es in der Regel vom Anwender nicht erwünscht, dass diese persönlichen Daten auf einem zentralen Server gespeichert werden, um dort ein neuronales Netzwerk zu trainieren.

Sogenannte Lifelong Deep Neural Network (L DNN) Algorithmen können das Potenzial haben, diesen Widerspruch aufzulösen, indem sie verteiltes und kontinuierliches Lernen ohne den Austausch von Rohdaten ermöglichen und dabei auch auf mit wenig Speicher und Rechenleistung ausgestatteten Edge Devices lern- und lauffähig sind.

Im Rahmen dieser Arbeit wird das Konzept „*Lifelong Deep Neural Network*“ (siehe [1]) hinsichtlich seiner Funktionalität und Anwendbarkeit auf andere Aufgabengebiete analysiert. Dazu wird eine prototypische Implementierung zur praktischen Evaluierung auf Basis der technischen Beschreibungen in [1], Behauptungen in [2] und genannten Implementierungen in [3] umgesetzt.

In Kapitel 2 werden die theoretischen Grundlagen eingeführt, in welchem zunächst auf generelle Punkte zu Deep Learning eingegangen wird. Auf Basis dieser allgemeinen Grundlagen werden detaillierter die Themen kontinuierliches und verteiltes Lernen erläutert, da diese die Hauptaspekte dieser Arbeit sind. Zudem gibt es ein Unterkapitel zu inkrementellen Klassifikatoren. In Kapitel 2.5 wird der L DNN A vorgestellt, mit einer anschaulichen Beschreibung und Darstellung des Ansatzes sowie dessen detaillierten Erläuterung und Aufteilung. Innerhalb von Kapitel 2.5.4 wird dieser Ansatz mit aktuellen Ansätzen des kontinuierlichen und verteilen Lernen verglichen und die Unterschiede zu gängigen Ansätzen herausgestellt. Kapitel 3 vergleicht unterschiedliche Modelle, die innerhalb des L DNN Algorithmus eingesetzt werden können. Auf Basis dieser Vergleiche wird schließlich eine finale Architektur konzipiert. Der Aufbau der Evaluierung der beschriebenen prototypischen Umsetzung des Algorithmus wird in Kapitel 4 vorgestellt. Dabei werden die verwendeten Datensätze und Metriken sowie die später durchgeführten Evaluierungsfälle vorgestellt. Die Ergebnisse sowie die Aus- und Bewertung der vorgestellten Evaluierungsfälle werden in Kapitel 5 beschrieben. Kapitel 6 beschreibt den aufgebauten Demonstrator sowie die dazu entwickelte GUI. In Kapitel 7 wird schließlich eine Zusammenfassung der Ergebnisse sowie ein Ausblick gegeben.

2 Theoretische Hintergründe

In diesem Kapitel wird eine Übersicht über die theoretischen Grundlagen gegeben, welche im weiteren Verlauf der Arbeit notwendig sind. Zunächst findet eine Einführung von Deep Learning mit dem Augenmerk auf die kritischen Punkte statt. Darauf folgt eine detailliertere Einführung in die Themen kontinuierliches und verteiltes Lernen sowie inkrementelle Klassifikatoren.

Für eine grobe Einordnung kann gesagt werden, dass Deep Learning, kontinuierliches Lernen (*Continual Learning*) und verteiltes Lernen (*Distributed Learning*) spezifische Themen aus dem Bereich des maschinellen Lernens (*Machine Learning*) sind. Abbildung 1 gibt eine graphische, beispielhafte Darstellung der Verhältnisse. Die einzelnen Bereiche haben einen hohen Überschneidungsgrad, da z.B. für das *Continual Learning* eine Vielzahl von Ansätzen des *Deep Learning* genutzt werden. Dennoch hat jeder Bereich seine eigenen spezifischen Probleme und unterschiedliche Methoden, um diese zu lösen. In dieser Arbeit werden Methoden und Komponenten aus allen Bereichen kombiniert eingesetzt und genutzt.

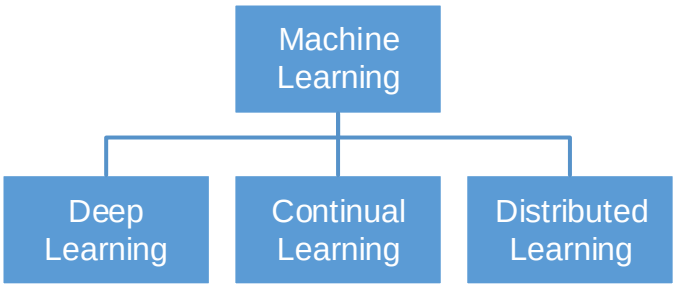

Abbildung 1: Verhältnis von verschiedenen Lernansätzen zu Machine Learning

2.1 Deep Learning

In diesem Abschnitt wird eine kurze Übersicht über Deep Learning gegeben. Es wird beschrieben wie *Deep Neural Networks* (DNN) funktionieren und wie diese trainiert werden können. Zudem wird der Zusammenhang zu maschinellem Lernen aufgezeigt. Danach wird die grundlegende Struktur und das Verhalten von neuronalen Netzen erklärt sowie die Algorithmik für das Training solcher Netze eingeführt. Zum Schluss werden mögliche Probleme beim Trainieren von diesen Netzen sowie dazugehörige Lösungsansätze aufgezeigt. Es werden die grundlegenden Punkte zu Deep Learning genannt und aufgeführt, jedoch wird in dieser Arbeit nicht auf jeden Punkt detailliert eingegangen, da das Hauptaugenmerk auf der später folgenden Untersuchung und Bewertung des L DNN Algorithmus liegt.

Deep Learning ist, wie in Abbildung 1 dargestellt, ein Gebiet des maschinellen Lernens. Unter maschinellem Lernen werden lernende und datenbasierte Ansätze verstanden welche eine gewisse Eingang-/Ausgangsrelation herstellen, beispielhaft dargestellt in Abbildung 2.

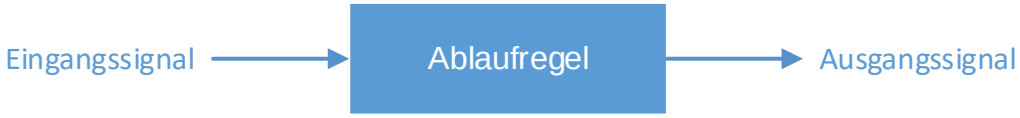

Abbildung 2: Generelle Problemstellung für maschinelles Lernen

Das Lernen dieser Ablaufregel geschieht mithilfe der Trainingsdaten. Dieser Zusammenhang zeigt, dass die Wahl der Trainingsdaten entscheidend ist, um eine gute und generalisierte Ablaufregel zu erlernen. Machine Learning und Deep Learning Algorithmen bekommen jeweils ein gewisses Eingangssignal, welches abhängig von der Anwendung vorverarbeitet wird. Der Unterschied zwischen diesen Algorithmen ist, dass in konventionellen Machine Learning Algorithmen die Features mithilfe einer vordefinierten Regel extrahiert werden [4]. Für diese Aufgabe existiert keine Theorie, weshalb Erfahrungen von Experten notwendig sind, um gute und relevante Features für die folgende Aufgabe, z.B. die Klassifikation, zu extrahieren. Die folgende Klassifikation wird von einem separaten Klassifikator durchgeführt, beispielsweise *kNN* (k-nearest Neighbour) oder *SVM* (Support Vector Machine). Innerhalb des Deep Learning existiert nur ein sogenanntes *Deep Neural Network* (DNN) für die Aufgaben der Feature Extraktion und Klassifikation. Das DNN lernt und adaptiert seine Netzwerkparameter mithilfe einer passenden *Loss*-Funktion. Das effiziente Anpassen der Parameter kann mithilfe des *Backpropagation*-Algorithmus umgesetzt werden ([4], [5], [6], [7]).

Die Lernstrategie von DNNs basiert grundlegend auf der Art wie Menschen lernen zu sprechen, zu laufen oder zu rechnen. Es wird anhand von Beispielen das Verhalten soweit angepasst, bis das gewünschte Ergebnis erzielt werden kann. Obwohl Deep Learning häufig als neue Technologie gesehen wird, gab es die ersten Untersuchungen und Erscheinungen in dem Themengebiet bereits in den 1940ern. Nach Ian Goodfellow ([4]) kann man die Geschichte des Deep Learning in drei Stufen unterteilen. Im Zeitraum von 1940 bis 1960 war es als *Cybernetics* bekannt. Zwischen 1980 und 1990 wurde es als *Connectionism* bezeichnet und beim Wiederaufleben seit 2006 mit dem aktuellen Namen *Deep Learning*. Die dritte Welle der Entwicklung, in der wir uns aktuell befinden, begann mit einem Durchbruch von Geoffrey Hinton. Er konnte zeigen, dass ein spezielles neuronales Netzwerk, das sogenannte „*Deep Belief Network*“, effizient mithilfe der Strategie „*Greedy Layer-Wise Pretraining*“ trainiert werden kann [8]. Seit diesem Durchbruch stieg und steigt auch weiterhin die Anzahl der Anwendungen von DNNs deutlich an. Beispielhafte Anwendungen für DNNs heutzutage sind Empfehlungssysteme (z.B. bei Amazon), automatische Spracherkennung, Text zu Sprache Übersetzung, Objekterkennung/-klassifizierung, Bildersegmentierung und viele weitere [7]. Abhängig von der speziellen Aufgabe wird das Netzwerk und die Architektur angepasst. Aufgrund der Vielzahl an unterschiedlichen Anwendungen gibt es auch eine Vielzahl an unterschiedlichen DNN-Architekturen, beispielsweise *Convolutional Neural Networks* (CNN), *Recurrent Neural Networks* (RNN) oder *Deep Belief Nets* (DBN) [6]. Im Folgenden wird die Architektur eines DNN beispielhaft anhand eines *Feedforward Neural Network* gezeigt. Diese Netzwerke, auch als *Multilayer Perceptron* (MLP) bekannt, werden als Basis Modul

innerhalb des Deep Learning bezeichnet [4]. Der Name *Feedforward* kommt von der Eigenschaft des Netzwerks, dass Information nur vorwärts (*Forward*), vom Eingangssignal durch das Netz zum Ausgangssignal, durch das Netzwerk fließt (siehe auch Abbildung 2). Diese Netzwerke besitzen keine Feedback Verbindungen. *Feedforward* Netzwerke bestehen aus mehreren Schichten (*Layer*), welche aneinandergereiht das Netzwerk bilden. Jede Schicht besteht wiederum aus mehreren Neuronen. Abbildung 3 stellt ein solches einzelnes Neuron in einem *Feedforward Neural Network* dar.

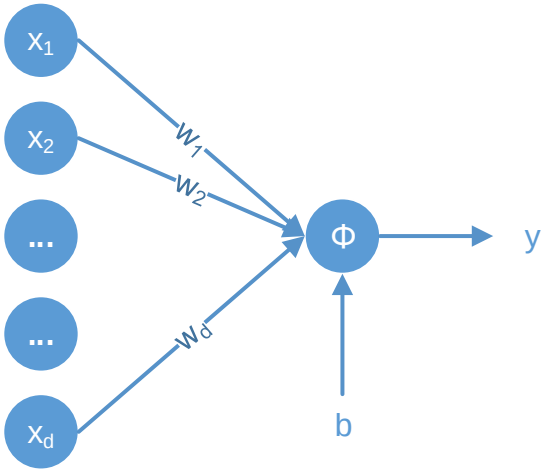

Abbildung 3: Einzelnes Neuron in einem *Feedforward Neural Network*

Diese graphische Darstellung kann auch mathematisch formuliert werden. Die mathematische Gleichung des Eingang-/Ausgangsverhaltens eines einzelnen Neurons ist in Formel **(1)** und Formel **(2)** gegeben.

$$a = \underline{w}^T \underline{x} + b \tag{1}$$

$$y = \phi(a) \tag{2}$$

Der Ausgang y eines einzelnen Neurons wird durch die (typischerweise nicht-lineare) Aktivierungsfunktion ϕ der Aktivierung a beschrieben. Die Aktivierung a wiederum ist eine affine Funktion des Eingangs $\underline{x}$. Das Eingangssignal kann ein Vektor, ein zweidimensionales Bild oder ein drei- oder höherdimensionaler Tensor sein. Die trainierbaren und adaptierbaren Parameter des Neurons sind die Gewichte $\underline{w}$ und der Bias b.

Eine Schicht (*Layer*) des Netzwerks, dargestellt mit einzelnen Neuronen in Abbildung 4 und in komprimierter Darstellungsform in Abbildung 5, besteht allgemein aus c Neuronen, welche mit dem Eingang und Ausgang verbunden sind.

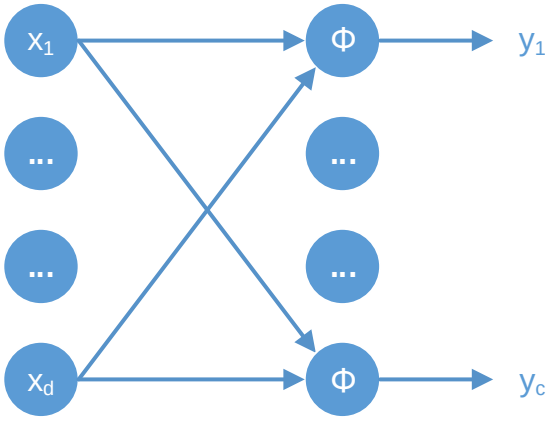

Abbildung 4: Neuronen-*Layer*

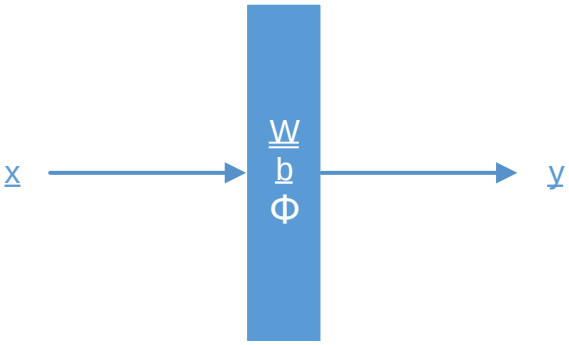

Abbildung 5: Neuronen-*Layer* in komprimierter Darstellung

Neuronen in derselben Schicht sind nicht miteinanderverbunden. Abhängig von der Netzwerkarchitektur variieren die Verbindungen der Neuronen zum Ein- und Ausgang. Zwei der meist genutzten *Layer*-Architekturen sind *Dense* (auch *Fully Connected* genannt) und *Convolutional Layer*. Bei einem *Dense Layer* sind alle Neuronen i mit jedem Eingang x_j verbunden. Ein Netzwerk, dass nur aus solchen Schichten besteht wird F*ully Connected Network* (FCN) genannt. Der Nachteil dieser Netze ist die sehr große Anzahl an Paramatern, da jede Verbindung eine Gewichtung benötigt. Diese große Anzahl an Parametern resultiert in einer sehr hohen Komplexität bei der Berechnung und einem hohen Speicherbedarf. Zudem wird aufgrund der Verbindungen in diesen Netzwerken kein lokales Verhalten/Feature des Eingangs gelernt, da alle Neuronen voll mit dem Eingang verbunden sind und den gesamten Eingang sehen. Diese Probleme können mithilfe eines *Convolutional Neural Network* (CNN) gelöst werden. Ein CNN besteht hauptsächlich aus *convolutional Layer*. Aufgrund der Eigenschaften von diesen Schichten mit *sparsen* Verbindungen und geteilten Parametern kann der Speicherbedarf deutlich reduziert werden und das Netzwerk ist fähig, lokale Verhaltensmuster unabhängig von der Position zu erkennen. Vereinfacht gesagt fokussiert sich ein CNN auf lokale Eingangsmuster [9].

Nachdem eine Schicht definiert ist können verschiedene Schichten zusammengeführt werden. Die resultierende Architektur stellt schließlich das Neuronale Netzwerk dar. Die finale Länge der Kette von Schichten ist die Tiefe des Modells. Von dieser Terminologie entstand der Name *Deep Learning*. Die erste und letzte Schicht wird als

Eingangs- (*Input Layer*) bzw. Ausgangsschicht (*Output Layer*) bezeichnet. Diese Schichten haben definierte Größen, da der Eingang und der Ausgang des Netzwerks definierte Größen haben und damit die Dimensionen dieser Schichten bestimmen. Alle Schichten zwischen diesen beiden sind die versteckten Schichten (*Hidden Layer*), da sie weder von der Eingangs- noch von der Ausgangsseite ersichtlich sind. Im Gegensatz zur Ein- und Ausgangsschicht ist die Größe dieser Schichten nicht fest vorgegeben und kann unabhängig von den Eingängen und Ausgängen gewählt werden. Typischerweise formen die Schichten einen Flaschenhals, welcher das Netzwerk zwingt, ein einfaches Modell des Systems zu erstellen. Dieses erlernte Modell soll in der Lage sein generelle Muster der Daten zu erlernen, um auch auf neuen, bisher unbekannten Daten (Test Daten) die gewünschte Aufgabe durchzuführen [10].

Mit nicht-linearen Aktivierungsfunktionen wie beispielsweise Softmax (Formel **(3)**) oder *Rectifier Linear Unit* (ReLU) (Formel **(4)**) kann gezeigt werden, dass bereits ein simples MLP eine willkürliche Ein-/Ausgangs-Beziehung beliebig genau annähern kann, wenn die Anzahl von versteckten Knoten nicht begrenzt ist [10]. Diese Eigenschaft von neuronalen Netzen ist auch bekannt als Universelle Funktionsapproximation (*Universal Function Approximation*).

$$\phi_i(\underline{a}) = \frac{e^{a_i}}{\sum_{j=1}^{c} e^{a_j}} \tag{3}$$

$$\phi_i(a_i) = \begin{cases} a_i, & a_i > 0 \\ 0, & a_i \leq 0 \end{cases} \tag{4}$$

Mit linearen Aktivierungsfunktionen ist das gesamte Netzwerk, unabhängig von der Tiefe des Netzes, nur eine lineare Transformation der Eingangsdaten. Damit können lediglich linear-lösbare Probleme gelöst werden. Das zeigt die Bedeutung von nichtlinearen Aktivierungsfunktionen.

Um ein neuronales Netzwerk mit Bezug auf das gewünschte Verhalten zu trainieren muss eine passende Verlust- (*Loss*-) Funktion definiert werden. Diese Funktion wird auf Basis der Trainingsdaten bezüglich den Netzwerkparametern minimiert. Allgemein ist das Ziel des Trainings die Kostenfunktion $L(\underline{\theta})$ auf Basis der verfügbaren Trainingsdaten zu minimieren, wie beschrieben in Formel **(5)**.

$$\min_{\underline{\theta}} L(\underline{\theta}) = \min_{\underline{\theta}} \frac{1}{N} \sum_{n=1}^{N} \iota(\underline{x}(n), \underline{y}(n); \underline{\theta}) \tag{5}$$

Dabei kann $\iota(\underline{x}(n), \underline{y}(n); \underline{\theta})$ eine willkürliche *Loss*-Funktion mit der gewünschten Ausgabe des Netzwerks (Label) $\underline{y}(n)$ und dem Eingang des Netzwerks $\underline{x}(n)$ für das Sample n sein. Die Kostenfunktion ist der gemittelte Wert der *Loss*-Funktion über alle N Trainingsdaten. Die Wahl der *Loss*-Funktion ist abhängig von der zu lösenden

Problemstellung. Für Regressionsprobleme ist der l_2-*Loss* eine typische Funktion, bei dem die l_2 Norm des Fehlers als Optimierungskriterium genutzt wird. Der Fehler (in der Literatur *Error* genannt) ist definiert als Unterschied zwischen der gewünschten Ausgabe $\underline{y}$ und der tatsächlichen Ausgabe des neuronalen Netzwerks, welcher mit der nicht-linearen Funktion $f(\underline{x};\underline{\theta})$ beschrieben werden kann. Diese Funktion stellt den willkürlichen Zusammenhang zwischen Eingang und Ausgang dar. Der genannte l_2-*Loss* ist in Gleichung **(6)** formuliert.

$$\iota\left(\underline{x}(n),\underline{y}(n);\underline{\theta}\right) = \left|\left|\underline{y} - f\left(\underline{x};\underline{\theta}\right)\right|\right|_2^2 \qquad (6)$$

Eine typische *Loss*-Funktion für die Klassifikation ist der kategorische *Loss*. Die Funktion ist in Gleichung **(7)** gegeben. Die letzte Schicht bei einer Klassifikationsaufgabe besitzt in der Regel eine Softmax-Aktivierungsfunktion, weshalb die finale Ausgabe des neuronalen Netzwerks ($f(\underline{x};\underline{\theta})$) die Wahrscheinlichkeiten für die Klassenzugehörigkeit darstellt.

$$\iota\left(\underline{x}(n),\underline{y}(n);\underline{\theta}\right) = -\underline{y}^T \ln f(\underline{x};\underline{\theta}) \qquad (7)$$

Das Ziel des gesamten Trainingsprozess ist es das Minimum $\min_{\underline{\theta}} L(\underline{\theta})$ zu finden, welches im Allgemeinen keine geschlossene Lösung besitzt. Aufgrund dessen werden numerische Optimierungsverfahren im Bereich Deep Learning eingesetzt. Der meist verwendete Algorithmus zur Minimierung der Kostenfunktion ist der *Gradient Descent* (GD) Algorithmus. In diesem Algorithmus werden die Netzwerkparameter so angepasst, dass ein kleiner Schritt in Richtung des negativen Gradienten gegangen wird [5]. Der Anpassungsschritt des GD-Algorithmus kann mit Formel **(8)** beschrieben werden.

$$\underline{\theta}^{t+1} = \underline{\theta}^t - \eta \underline{\nabla} L\left(\underline{\theta}\right)|_{\underline{\theta}=\underline{\theta}^t} \qquad (8)$$

Dabei ist $t \in \mathbb{Z} \geq 0$ der Iterationsindex und η die Schrittweite (oder auch Lernrate genannt), welche die Länge des Schrittes in Richtung des negativen Gradienten Vektors $-\underline{\nabla} L\left(\underline{\theta}\right)$ bestimmt. Zu Beginn ($t = 0$) müssen die Netzwerkparameter $\underline{\theta}$ initialisiert werden, was eine initiale Schätzung $\underline{\theta}^0$ darstellt.

Aufgrund der Größe der Trainingsdaten für neuronale Netze (z.B. 55000 Bilder für MNIST [11]) kann nicht der gesamte Datensatz im Speicher gehalten werden. Deshalb wird in der Praxis der *Stochastic Gradient Descent* (SGD) Algorithmus genutzt, welcher die Berechnungsaufwände für jede Iteration reduziert, da der Gradienten-Vektor $\underline{\nabla} L\left(\underline{\theta}\right)$ und das Update $\underline{\theta}^{t+1}$ lediglich für einen sogenannten Minibatch i berechnet werden. Der stochastische Gradienten-Vektor $\underline{\nabla} L_i$ ist die Schätzung des

Gradienten-Vektors $\underline{\nabla L}$. Diese Schätzung führt zu einem verzerrten Gradienten, was wiederum zu dem Namen **stochastischer** GD führt [7].

Die Berechnung des Gradienten-Vektors $\underline{\nabla L}$ ist elementar, um die Parameter anpassen zu können. Für die Berechnung des Gradienten-Vektors müssen die partiellen Ableitungen $\frac{\partial L}{\partial \underline{W}}$ und $\frac{\partial L}{\partial \underline{b}}$ der Kostenfunktion nach den Netzwerkparametern $\underline{W}$ und $\underline{b}$ berechnet werden. Dies kann mithilfe des *Error Backpropagation* Algorithmus effizient umgesetzt werden (häufig auch nur *Backpropagation* oder *Backprop* genannt). Wie der Name bereits sagt, werden dabei die Error-Vektoren durch das Netzwerk rückwärts fortgepflanzt. Dabei wird das Netzwerk rückwärts durchpropagiert, beginnend bei Ausgangsschicht L. Der beispielhafte Error-Vektor der Ausgangsschicht abgeleitet nach den Gewichten $w_{L_{ij}}$ ist in Formel **(9)** gegeben.

$$\underline{\delta_L^T} = \frac{\partial L(\underline{\theta})}{\partial w_{L_{ij}}} = \frac{\partial L}{\partial \underline{x_L}} \frac{\partial \underline{x_L}}{\partial \underline{a_L}} \frac{\partial \underline{a_L}}{\partial w_{L_{ij}}} \qquad (9)$$

Dabei ist $\underline{x_L}$ der Ausgang und $\underline{a_L}$ die Aktivierung von Schicht L. Die partiellen Ableitungen nach b folgen denselben Gleichungen, bis auf dem Austausch von $\partial w_{L_{ij}}$ durch ∂b_{L_i}.

Für die restlichen Schichten $l \in [1, L-1]$ kann der Error-Vektor mithilfe der Kettenregel für die Differenzierung berechnet werden. Der Error-Vektor der Schicht l kann mithilfe von Gleichung **(10)** berechnet werden [9].

$$\underline{\delta_l^T} = \underline{\delta_{l+1}^T} \frac{\partial \underline{a_{l+1}}}{\partial \underline{a_l}} \frac{\partial \underline{a_l}}{\partial w_{l_{ij}}} \qquad (10)$$

In den folgenden Abbildungen wird der Vorwärtspfad, sowie der Pfad der *Backpropagation* graphisch dargestellt. Abbildung 6 stellt dabei den Vorwärtspfad dar, bei dem die einzelnen Aktivierungen der Schichten berechnet werden, welche wiederum notwendig sind, um die Error-Vektoren rückwärts durch das Netzwerk propagieren zu können.

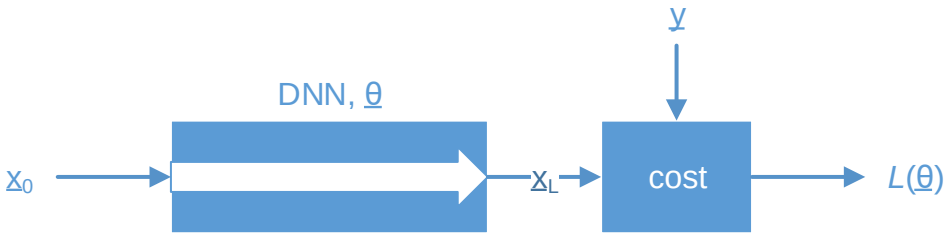

Abbildung 6: Vorwärts-Pfad durch ein Netzwerk

In Abbildung 7 wird die *Backpropagation* durch das Netzwerk vereinfacht graphisch dargestellt. Dafür werden die Aktivierungen, die durch den Vorwärtspfad errechnet wurden, genutzt, um die Error-Vektoren (partiellen Ableitungen) zu berechnen.

Abbildung 7: Error Backpropagation durch ein Netzwerk

Abbildung 6 und Abbildung 7 stellen somit die gegensätzlichen Pfade durch ein neuronales Netzwerk dar. Der Rückwärtspfad (Backpropagation) in Abbildung 7 wird dabei nur während des Trainings eines DNN verwendet, um die Parameter anzupassen. Der Vorwärtspfad aus Abbildung 6 ist sowohl während des Trainings, als auch während der späteren Anwendung aktiv. In der Trainingsphase eines DNN folgt der Rückwärtspfad immer einem vorhergehenden Vorwärtspfad durch das Netzwerk.

Backpropagation durch ein Netzwerk, wie in Abbildung 7, erhält als Eingangssignal den Error-Vektor der Ausgangsschicht L und gibt final den Error-Vektor des gesamten Netzwerks aus. Mit der gewonnenen Information können, wie bereits beschrieben, die Netzwerkparameter entsprechend einer gewählten numerischen Optimierungsmethode (z.B. SGD) angepasst werden.

In [5] wird der Algorithmus der *Error Backpropagation* in den folgenden vier Schritten einfach zusammengefasst:

- Gebe die Eingangsdaten $\underline{x}_0$ in das Netzwerk ein und propagiere diese wie in Abbildung 6 vorwärts durch das Netzwerk, um alle Aktivierungen der versteckten und Ausgangs Neuronen zu berechnen
- Berechne den Error-Vektor $\underline{\delta}_L^T$ für alle Ausgangsneuronen mithilfe Formel **(9)**
- Propagiere die Error-Vektoren $\underline{\delta}_l^T$ rückwärts durch das Netzwerk (Formel **(10)**), um die Error-Vektoren für alle versteckten Neuronen im Netzwerk zu erhalten (siehe Abbildung 7)
- Wende eine numerische Optimierungsmethode (z.B. SGD) an, um die Netzwerkparameter anzupassen

Die genannten Schritte können beliebig häufig wiederholt werden. Wenn alle Trainingsdaten einmal durch diesen Prozess durchgegangen sind, wird in der Literatur von einer Epoche gesprochen. Das Trainieren (Optimieren) eines tiefen neuronalen Netzwerks ist eine schwierige Aufgabe und kann unter einer Vielzahl von Optimierungsproblemen leiden (z.B. lokale Minima oder *vanishing* Gradient) [9].

Wie in [7] beschrieben bietet die Optimierung eine Möglichkeit die Kostenfunktion zu minimieren. Generell sind die Ziele der Optimierung und von Deep Learning jedoch unterschiedliche. In der reinen Optimierung ist es das Ziel das Minimum einer (Trainings-) Kosten-Funktion zu finden. Im Gegensatz dazu liegt der Fokus in Deep Learning darauf, den Generalisierung-Fehler zu minimieren, welcher der Wert der Kostenfunktion auf neuen, bisher unbekannten Eingangsdaten ist. Deshalb ist es während dem Training wichtig, regelmäßig Validationsdaten einzuspielen, um zu überprüfen ob es eine Überanpassung (*Overfitting*) des Netzwerks auf die

Trainingsdaten gibt. Overfitting im Kontext von Deep Learning bedeutet, dass der Fehler auf Basis der Trainingsdaten im Vergleich zu dem Generalisierungs-/Test-Error sehr gering ist. Es gibt wiederum einige verschiedene Methoden, um Overfitting vorzubeugen, wie z.B. Dropout oder Regularisierung ([4], [5], [6], [7]).

2.2 Kontinuierliches Lernen

Menschen wie auch Tiere haben die Fähigkeit sich kontinuierlich neues Wissen anzueignen und neue Fähigkeiten zu erlernen. Diese Fähigkeit wird in der Literatur als lebenslanges (*Lifelong*) oder kontinuierliches (C*ontinual*) Lernen bezeichnet.

In diesem Kapitel werden zunächst Schwierigkeiten beim kontinuierlichen Lernen aufgeführt. Darauffolgend werden unterschiedliche Methoden vorgestellt, welche die Probleme verringern bzw. lösen sollen. Schließlich wird der Überbegriff des kontinuierlichen Lernens in unterschiedliche konkrete Anwendungsbereiche unterteilt.

Kontinuierliches Lernen kann generell als eine besondere Form des Machine Learning gesehen werden, bei der meistens dieselben Architekturen (DNNs) wie im Bereich Deep Learning genutzt werden, jedoch aufgrund spezieller Probleme teilweise andere Algorithmen im Einsatz sind. Der entscheidende Punkt beim kontinuierlichen Lernen ist, dass durch das Erlernen neuen Wissens das alte, bereits erlernte Wissen nicht verloren gehen darf. Dieses Vergessen durch das Hinzufügen neuen Wissens wird im Bereich des maschinellen Lernens als katastrophales Vergessen (*Catastrophic Forgetting*) bezeichnet. Im Kontext von Deep Learning kann *Catastrophic Forgetting* als das Vergessen wichtiger Parameter von einer zuvor erlernten Aufgabe beim Trainieren einer neuen Aufgabe bezeichnet werden. Dieses Verhalten soll in Echtzeit-Systemen, die eine typische Anwendung von kontinuierlichem Lernen sind, vermieden werden.

Systeme, die mit der Umgebung interagieren oder bei denen sich die Umgebungsbedingungen ändern können, benötigen für eine durchgehend korrekte Funktionsweise die Fähigkeit, neue Informationen verarbeiten zu können und daraus neue Verhaltensmuster oder Entscheidungen ableiten zu können. Für das kontinuierliche Weiterlernen von DNNs gibt es verschiedene Probleme und Ansätze zum Lösen dieser, die in diesem Kapitel beleuchtet werden.

Aktuelle DNNs, welche für viele Anwendungen genutzt werden, benutzen Gradienten-basierte Methoden (z.B. SGD, siehe Kapitel 2.1). Wenn ein DNN mit solch einer Methodik inkrementell angepasst wird, erliegen diese Netze dem Problem des katastrophalen Vergessens ([12], [13], [14]). Das liegt daran, dass diese Gradienten-basierten Methoden die Netzwerkparameter entsprechend den aktuellen Error-Vektoren anpassen, welche lediglich von den Eingangsdaten des aktuellen Minibatch abhängig sind. Der Grund für *Catastrophic Forgetting* ist bekannt als Stabilität-Plastizität Dilemma [15]. Das Modell benötigt zunächst ausreichend Plastizität (Verformbarkeit) um neue Aufgaben zu erlernen. Große Parameteränderungen bewirken jedoch das Vergessen vorher erlernter Aufgaben. Wenn die Netzwerk-

Parameter stabil gehalten werden, werden vorher erlernte Aufgaben nicht vergessen. Eine zu große Stabilität verhindert jedoch das Erlernen neuer Aufgaben und verringert damit die Plastizität. Beispielsweise wird ein Netzwerk zunächst für eine Objektklasse „Hund“ trainiert. Im weiteren Verlauf werden nur noch „Katzen“ trainiert und die Objektklasse „Katze“ wird erlernt, während die bereits erlernte Klasse „Hund“ höchstwahrscheinlich verlernt wird.

Intuitiv kann als Lösung für dieses Problem gefordert werden, dass die Anpassungsregel so begrenzt wird, dass zuvor erlernte Informationen erhalten bleiben während Parameter gesucht werden, um eine neue Aufgabe zu lösen. Wie in [12] beschrieben kann das katastrophale Vergessen bildlich im Parameterraum illustriert werden. Abbildung 8 zeigt mögliche Verläufe im Parameterraum beim Erlernen einer neuen Aufgabe B, nachdem Aufgabe A erlernt wurde.

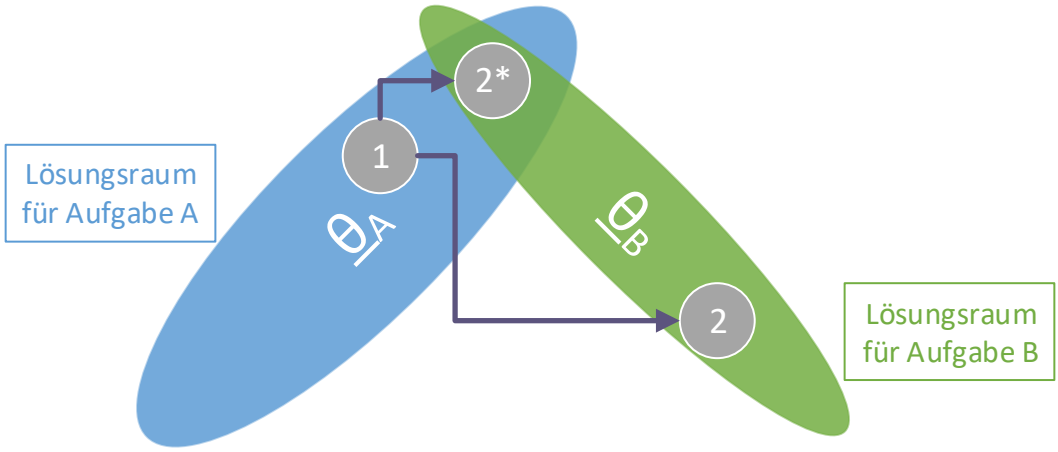

Abbildung 8: Erlernen neuer Aufgabe B und mögliche Folgen

Dabei stellt die blaue Ellipsoide $\underline{\theta_A}$ einen Lösungsraum mit geringem Fehler für die Aufgabe A dar mit der erlernten und gefundenen Lösung in Punkt 1. Wenn nun Aufgabe B zusätzlich erlernt werden soll, kann es im besten Fall vorkommen, dass die Parameter so angepasst werden, dass am Ende Punkt 2* erreicht wird. Dieser Punkt ist in der Schnittmenge zwischen $\underline{\theta_A}$ und $\underline{\theta_B}$ und kann somit beide Aufgaben mit geringem Fehler lösen. Wenn jedoch Punkt 2 erreicht wird, kann lediglich Aufgabe B ausreichend gelöst werden. Dies ist lediglich eine einfache Beschreibung der Thematik des *Catastrophic Forgetting*. In realen Anwendungen ist die Aufgabenstellung deutlich komplexer, wodurch es keine oder nur gering überlappende Bereiche zwischen verschiedenen Aufgaben geben kann.

Um *Catastrophic Forgetting* zu vermeiden oder den Einfluss zu minimieren gibt es unterschiedliche Ansätze, die genutzt werden können. Eine der ersten Ansätze von 1993 [16] sieht den Grund für das Vergessen beim Backpropagation Algorithmus. Dafür wurde eine Variation des Backpropagation Algorithmus entwickelt. Die Idee dahinter ist, dass nur die aktiven Neuronen während dem Training angepasst werden, die für den Fehler (Error) des gesamten Netzwerks verantwortlich sind. Dadurch soll der Einfluss auf andere, bereits erlernte Muster reduziert werden. Ein weiterer früher Ansatz zur Reduzierung des *Catastrophic Forgetting* ist die Reduzierung der internen

überlappenden Verteilungen, da die Überlappung der einzelnen internen Verteilung für verschiedene Muster als Grund für das *Catastrophic Forgetting* gesehen wurden [12]. Als Lösung dafür werden Semi-Distributed Repräsentationen eingeführt. Die Reduzierung der repräsentativen Überlappung wird durch die Einführung von *sparsen* Vektoren erzielt. *Sparse* Vektoren bedeuten, dass nur einige wenige Neuronen für die Repräsentation eines speziellen Musters aktiv sind, was automatisch die Überlappung zu anderen Mustern reduziert. Für diese Methode wurde ein Extraschritt im normalen Backpropagation Algorithmus eingeführt, bei dem die Aktivierungsmuster für die verdeckten Schichten „geschärft“ werden. Dabei werden die Aktivierungen der „aktivsten„ Neuronen erhöht, während gleichzeitig die Aktivierungen der weniger aktiven Neuronen reduziert werden. Diese Methode konnte *Catastrophic Forgetting* signifikant reduzieren, so lange nicht zu viele Muster gelernt werden müssen.

Aus diesen frühen Ansätzen wird bereits deutlich, dass die Lernalgorithmen einen großen Einfluss auf die Eigenschaft des Vergessens haben, weshalb diese besonders im Fokus der unterschiedlichen Ansätze stehen. Nach [17] kann zwischen fünf unterschiedlichen grundlegenden Methoden zur Vermeidung des C*atastrophic Forgetting* unterschieden werden. Diese fünf Ansätze werden im Folgenden vorgestellt.

Regularisierungsmethoden

Regularisierungsmethoden fügen Beschränkungen zu den Parameterupdates hinzu. Beispielhaft ist eine l_2-Regularisierung, bei der alle Gewichte dieselbe Regularisierung erfahren, in dem Fall durch die l_2-Norm der Gewichte. Die bekannteste und aktuell meist genutzte Methode aus dieser Kategorie ist die *Elastic Weight Consolidation* (EWC) [18]. Es wird eine Bedingung zur der *Loss*-Funktion hinzugefügt, welche Verformbarkeit (Plastizität) von den Parametern nimmt, die am relevantesten für die zuvor gelernte Aufgabe sind. Das Verhalten des EWC-Algorithmus ist graphisch in Abbildung 9 dargestellt.

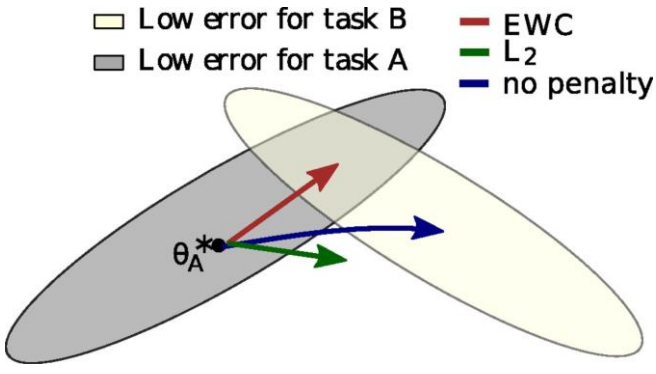

Abbildung 9: Einfluss von EWC auf Parameteranpassungen [18]

Dabei wird wieder eine einfache Darstellung des Parameterraums gewählt. Wenn keine Regularisierung gewählt wird, erzeugt das Erlernen von Aufgabe B das Verlernen der alten Aufgabe A (blauer Pfeil). Wenn alle Parameter gleich und zu stark gewichtet werden, kann die neue Aufgabe B aufgrund der geringen Anpassbarkeit der Parameter (grüner Pfeil) nicht korrekt gelernt werden. Mithilfe von EWC kann

schließlich eine Lösung für die Aufgabe B ohne Vergessen von Aufgabe A gefunden werden (roter Pfeil). Die Funktion $\mathcal{L}$, welche im EWC-Algorithmus minimiert werden soll, ist in Gleichung **(11)** gegeben.

$$\mathcal{L}(\underline{\theta}) = \mathcal{L}_B(\underline{\theta}) + \sum_i \frac{\lambda}{2} F_i \left(\underline{\theta}_i - \underline{\theta}^*_{A,i}\right)^2 \qquad (11)$$

Dabei ist $\mathcal{L}_B(\underline{\theta})$ die Kostenfunktion für Aufgabe B und λ die Gewichtung der Regularisierung. Über diese Gewichtung wird angegeben, wie wichtig die alte Aufgabe im Vergleich zur neuen ist. i ist der Index der Parameter und F_i ist die Fisher Information für jeden Parameter. Diese Information gibt an, wie relevant dieser Parameter zur Darstellung von Aufgabe A ist. $\underline{\theta}^*_A$ sind schließlich die erlernten Parameter der Lösung für Aufgabe A und der Startpunkt der Parameteranpassungen. Wenn eine weitere Aufgabe C hinzukommt, können die Parameter von A und B gesondert in die Formel eingehen und gewichtet werden oder die Aufgaben A und B werden als gemeinsame Aufgabe in der Gleichung gebündelt und erhalten dieselbe Gewichtung [17], [18].

Ensemble Methoden

Ensemble Methoden trainieren verschiedene Klassifikatoren und kombinieren diese unterschiedlichen Klassifikatoren, um eine finale Schätzung abzugeben. Besonders frühe Ansätze dieser Methode zeigten einen klaren Nachteil bezüglich des Speicherbedarfs, da mit steigender Anzahl an Aufgaben der Speicherbedarf stark ansteigt. Neuere Ansätze begrenzen die Modellgröße mithilfe verschiedener Ansätze, um den Speicherbedarf zu limitieren. Der bekannteste Algorithmus dieser Methoden ist der sogenannte Pathnet-Ansatz [19]. Bei diesem Ansatz werden Agenten in einem neuronalen Netzwerk eingesetzt, welche die Teile des Netzwerks identifizieren, die für eine neue Aufgabe wiederverwendet werden können. Die relevanten Pfade für die vorherige Aufgabe werden eingefroren, um *Catastrophic Forgetting* zu vermeiden [14].

Rehearsal Methoden

Rehearsal Methoden nutzen Daten von vorhergehenden Aufgaben und fügen diese dem Trainingsprozess der neuen Aufgabe zu. Dadurch entsteht ein hoher Speicherbedarf, um die Trainingsdaten vorheriger Aufgaben zur Verfügung zu stellen. Diese Methoden wurden bereits bei frühen Ansätzen genutzt und es lassen sich gute Ergebnisse erzielen. Neuere Ansätze nutzen verschiedene Methoden, um eine sinnvolle Auswahl oder Komprimierung der „alten“ Trainingsdaten zu ermöglichen, damit nur wenige relevante Daten gespeichert werden müssen. Dafür können generative Modelle wie ein Variational Autoencoder (VAE) oder Generative Adversarial Networks (GAN) genutzt werden, welche aus komprimierten Darstellungen pseudo-reale Eingangsdaten erstellen können [20], [21]. Zusammengefasst werden diese Methoden unter dem Namen *Deep Generative Replay* (DGR) [22].

Dual-Memory Methoden

Die Grundlagen für die Dual-Memory Methoden liegen in *Complementary Learning Systems* (CLS) [23]. Die CLS-Theorie baut auf den biologischen Prinzipien des Gehirnes von Säugetieren auf. In diesem werden Erinnerungen in unterschiedlichen Regionen des Gehirns abgespeichert. Frische Erinnerungen werden in einem Gebiet namens Hippocampus abgespeichert. Diese Erinnerungen werden dann langsam während des Schlafes zum Neocortex übertragen. Im Anwendungsfall ist der Neocortex für das Extrahieren von generellen Informationen zuständig, während der Hippocampus für die Erinnerung an spezifische Informationen eingesetzt wird. Eine schematische graphische Darstellung für die Dual-Memory Methode ist in Abbildung 10 zu sehen.

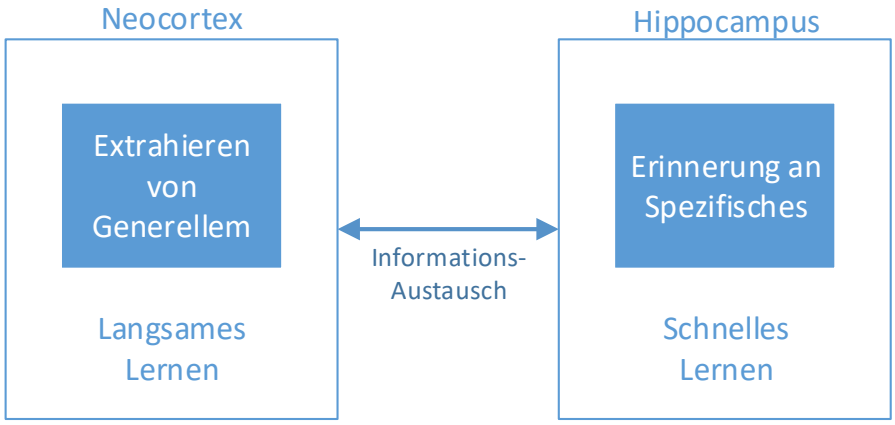

Abbildung 10: Schematische Darstellung der Dual-Memory Methode

Dieses Zusammenspiel eines langsam lernenden Netzes und eines schnell auffassenden Netzes wird in vielen Dual-Memory Methoden genutzt. Im Allgemeinen nutzen Dual-Memory Methoden zwei unterschiedliche Speicher und Netze, um unterschiedliche Informationen zu behalten. Die konkrete Umsetzung und Anwendung der CLS-Theorie auf die beiden zur Verfügung stehenden Netze variiert je nach Anwendungsfall und Netzwerkarchitektur [14], [17], [23].

Sparse-Coding Methoden

Bei diesen Methoden werden *sparse* Repräsentationen genutzt, welche die Wechselwirkung zwischen verschiedenen Repräsentationen (Aufgaben) reduziert. Die bereits eingeführte Methode aus [12] nutzt diese Methode, um effiziente *sparse* Repräsentationen einer Aufgabe zu erzeugen und damit ausreichend Parameter für das Erlernen einer neuen Aufgabe verfügbar zu haben.

Nachdem verschiedene Methoden vorgestellt wurden, die das *Catastrophic Forgetting* verhindern sollen, werden nachfolgend die Anwendungen des kontinuierlichen Lernens in unterschiedliche Kategorien unterteilt, um eine sinnvolle Vergleichbarkeit und Bewertung zu ermöglichen. Nach [13] können Anwendungen des kontinuierlichen Lernens in drei Gebiete unterteilt werden: *Incremental* ***Task*** *Learning, Incremental*

Domain *Learning* und *Incremental* ***Class*** *Learning*. Diese unterschiedlichen Szenarien werden im Folgenden definiert und der Unterschied zwischen den einzelnen Szenarien herausgestellt. Beispielhaft werden dafür zwei Aufgaben A und B angenommen, mit den Verteilungen der Eingangsdaten $P(X_A)$ und $P(X_B)$, den dazugehörigen Labels Y_A und Y_B und den jeweiligen Verteilungen $P(Y_A)$ und $P(Y_B)$. Abbildung 11 stellt die drei unterschiedlichen Szenarien am Beispiel des Split MNIST Datensatzes dar. In den gepunkteten Rechtecken wird der Eingang für das Training dargestellt, mit (x, y, t) für (Eingangsbild, Zielausgang, Aufgaben-ID).

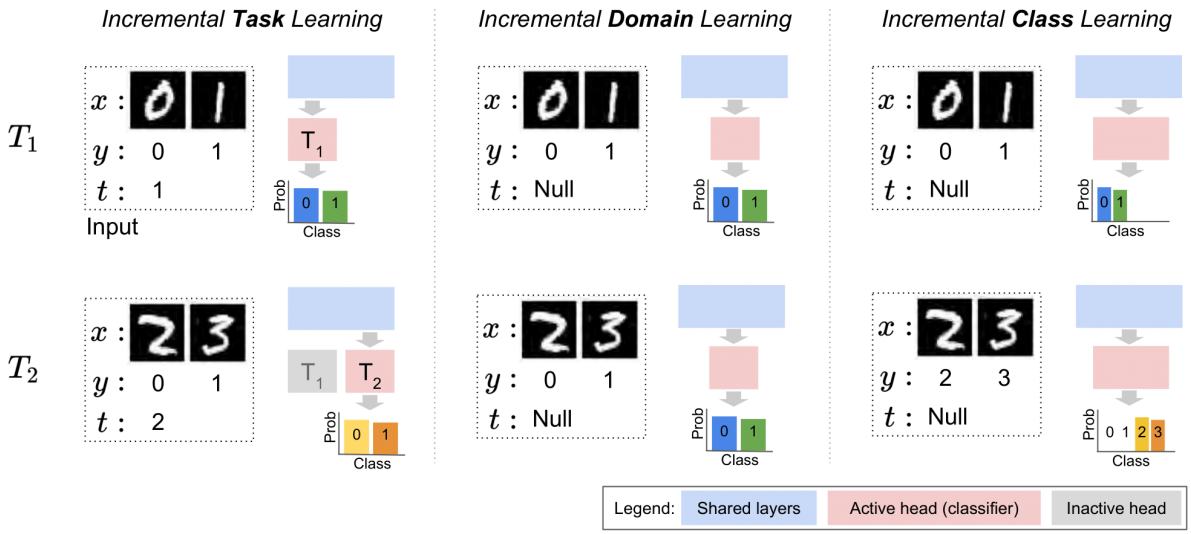

Abbildung 11: Darstellung der drei Continual Learning Szenarien am Beispiel von Split MNIST [13]

Incremental Task Learning

In diesen Szenarien haben die Aufgaben A und B unterschiedliche Ausgänge, $\{Y_A\} \neq \{Y_B\}$. Daraus ergibt sich eine unterschiedliche Verteilung der Ausgänge, $P(Y_A) \neq P(Y_B)$. Durch die Aufgabenstellung des kontinuierlichen Lernens gilt $P(X_A) \neq P(X_B)$, da eine neue Aufgabe inkrementell erlernt werden soll. Die Ausgänge unterscheiden sich in ihrer Dimension und semantischen Bedeutung. Beispielhaft kann die erste Aufgabe eine Klassifizierung zwischen fünf Klassen sein, während die zweite Aufgabe die Regression eines einzelnen Wertes ist. Dabei wird eine komplett neue Aufgabe erlernt (z.B. Regression statt Klassifikation). Um die korrekte Ausgabe zu ermöglichen, sind aufgabenabhängige Ausgangskomponenten (T_1 und T_2 in der linken unteren Graphik von Abbildung 11) notwendig, die abhängig von der Aufgaben-ID ausgewählt werden.

Deshalb ist bei diesen Szenarien zusätzlich die Aufgaben-ID t ein notwendiger Input für das Netzwerk [13].

Incremental Domain Learning

Beim inkrementellen Domain Learning variieren die Eingangsdaten und damit $P(X_A) \neq P(X_B)$. Das gesamte Netzwerk wird nicht angepasst, weshalb die Ausgabe des Netzwerks identisch bleibt mit $\{Y_A\} = \{Y_B\}$ und für ausgeglichene Datensätze auch $P(Y_A) = P(Y_B)$ gilt. Am Beispiel von Split MNIST mit der in Abbildung 11 dargestellten Ausgabe (binärer Klassifikator) können damit gerade von ungeraden Zahlen unterschieden werden. Bei diesen Anwendungen werden durch neue Aufgaben neue Bereiche (*Domains*) erlernt [13].

Incremental Class Learning

In diesen Szenarien werden inkrementell neue Klassen erlernt. Aufgrund der Vielzahl an Klassen gilt $P(Y_A) \neq P(Y_B)$ und aufgrund der grundlegenden Eigenschaften neuer Aufgaben auch $P(X_A) \neq P(X_B)$. In diesem Aufbau wird $\{Y_A\} = \{Y_B\}$ angenommen unter der Bedingung, dass die gesamte Anzahl an Klassen bekannt ist und zu Beginn die Dimension der Ausgabe auf die gesamte Anzahl an Klassen gesetzt wird. Auch in diesem Fall behält das Netzwerk über alle Aufgaben hinweg seine Architektur und besitzt keine aufgabenabhängigen Stellen [13]. Als Variation davon kann der Anwendungsfall gesehen werden, bei dem zu Beginn nicht die finale Anzahl an Klassen bekannt ist. Dort ändert sich bei einer neuen Aufgabe (neue Klasse) der Ausgang.

Im Rahmen dieser Arbeit liegt der Fokus auf der Aufgabe des *Incremental Class Learning*. Beispielhafte Anwendung ist ein Netzwerk, das zunächst auf gewisse Klassen eines Bilddatensatzes der Objekterkennung, z.B. Hunde und Katzen, trainiert wird. Zukünftig kommen nun Vögel hinzu, welche ebenfalls klassifiziert werden sollen. Dabei bleibt die Netzwerkstruktur erhalten und es soll für jede Klasse ein eindeutig identifizierbarer Ausgang vorhanden sein.

Mithilfe der in diesem Abschnitt definierten und unterteilten Methoden sowie Aufgabengebiete lassen sich unterschiedliche kontinuierliche Lernansätze miteinander vergleichen. Zudem wurde ein grundlegendes Verständnis über Schwierigkeiten sowie Lösungsansätze für *Lifelong/Continual* lernende Algorithmen vorgestellt.

2.3 Inkrementelle Klassifikatoren

Der untersuchte Anwendungsfall in dieser Arbeit liegt im Bereich des inkrementellen Klassen Lernens (*Incremental Class Learning*, siehe vorheriges Kapitel 2.2). Für diese Aufgabe ist es notwendig einen inkrementellen Klassifikator einzusetzen. Generell sind inkrementelle Klassifikatoren ein spezifischer Bereich des kontinuierlichen Lernens für den konkreten Anwendungsfall der Klassifikation. Deshalb gelten auch hier die

grundlegenden Probleme, welche in Kapitel 2.2 diskutiert wurden. In diesem Kapitel werden konkrete Beispiele für inkrementelle Klassifikatoren eingeführt, die später in der Konzeptionsphase verglichen und für den Anwendungsfall bewertet werden. Ein Klassifikator soll nach [24] folgende drei Punkte erfüllen, um inkrementell Klassen erlernen zu können:

- Er soll auf Basis eines Daten-Stream, in dem Sample der unterschiedlichen Klassen zu unterschiedlichen Zeitpunkten (zufällig) auftreten, trainierbar sein
- Zu jedem Zeitpunkt muss ein funktionierender Multi-Klassen Klassifikator für die bereits gesehenen und damit bekannten Klassen verfügbar sein
- Die Berechnungsanforderungen und der Speicherbedarf sollen beschränkt sein oder nur langsam mit Bezug auf die Anzahl an bekannten Klassen ansteigen

Im Folgenden werden nun beispielhaft zwei solcher inkrementellen Klassifikatoren vorgestellt:

Incremental Classifier and Representation Learning (iCaRL)

Der Incremental Classifier and Representation Learning (iCaRL) Algorithmus [24] besteht aus drei Komponenten. Die Klassifikation findet auf Basis einer *Nearest-Mean-of-Exemplars* Regel statt. Zudem gibt es eine priorisierte Exemplar-Auswahl und *Representation Learning* mithilfe von Wissens-Destillierung und prototypischen Samples. Bei diesem Algorithmus werden Prototypen/Repräsentationen für die unterschiedlichen Klassen angelegt. Auf Basis dieser exemplarischen Repräsentationen wird dann für ein neues Sample der Abstand zu den Repräsentationen (*Mean-of-Exemplars*) ermittelt. Schließlich folgt ein einfacher *Nearest-Mean* Klassifikator, beschrieben in Formel **(12)**.

$$y^* = \arg\min_{y=1,\dots,t} \left|\left|\varphi(x) - \mu_y\right|\right| \qquad (12)$$

Dabei ist y^* das prädizierte Label eines Inputs x auf Basis der Feature-Extraktion $\varphi(x)$. Der Prototypen-Vektor $\mu_1, \dots, \mu_t$ für jede bisher bekannte Klasse $y \in [1, t]$ ist durch Gleichung **(13)** definiert.

$$\mu_y = \frac{1}{|P_y|} \sum_{p \in P_y} \varphi(p) \qquad (13)$$

Dabei sind p die Prototypen der einzelnen Klassen, definiert in der Menge P_y. Die Anzahl an relevanten Prototypen $|P_y|$ kann dabei beliebig gewählt werden. Durch die direkte Verknüpfung der Feature-Extraktion in die Berechnung der exemplarischen Repräsentation μ_y ist sichergestellt, dass sich bei ändernder Feature-Extraktionsregel φ auch die Repräsentationen ändern und somit immer noch eine korrekte Klassifizierung der Eingangsdaten erfolgen kann.

Neue Daten, die für eine Klasse vorhanden sind, werden in die Berechnung der Klassenrepräsentation mit einbezogen. Dadurch kann eine kontinuierliche Weiterentwicklung und Verbesserung der Generalisierung des Klassifikators erreicht werden.

Wenn Daten für neue Klassen zur Verfügung stehen, wird mithilfe des iCaRL-Algorithmus nicht nur das exemplarische Prototypen-Set, sondern auch die Feature-Extraktion angepasst. Dafür werden die neuen Daten und die Repräsentationen der alten, bekannten Klassen als neuer Trainingsdatensatz genutzt. Nun findet ein Training nach typischen Deep Learning Algorithmen (z.B. Backpropagation) statt. Dabei soll für die neuen Daten das korrekte neue Klassenlabel ausgegeben werden, während die korrekte Klassifizierung der alten Klassen weiterhin bestehen soll. Dieser Schritt wird *Representation Learning* genannt, da auch der Feature-Extrahierer und damit die Repräsentation der Eingangsdaten im Netzwerk angepasst wird. Der Algorithmus kann auch ohne *Representation Learning* genutzt werden. In diesem Fall werden nur die Prototypen-Vektoren angepasst, beziehungsweise neue Vektoren hinzugefügt, wenn neue Klassen hinzukommen.

Mit diesem Ansatz werden auf Datensätze wie ImageNet gute Ergebnisse erzielt [24]. Grundlegend baut dieser Algorithmus auf dem Prinzip des *Rehearsal Replay* auf, da die gemittelten Prototypen-Vektor für das *Representation Learning* wieder in das System eingespeist werden.

Adaptive Resonance Theory (ART)

Die *Adaptive Resonance Theory* (ART) wurde auf Basis von unterschiedlichen Aspekten, wie das Gehirn Informationen verarbeitet, entwickelt. Die ART ist nicht ein konkretes, einzelnes Modell eines neuronalen Netzwerks, sondern es beschreibt eine Familie von unterschiedlichen Modellen. Das Ziel dieser Modelle ist es, die menschliche kognitive Informations-Verarbeitung nachzubilden [25], [26]. Der große Vorteil von ART-Netzwerken ist die Fähigkeit, das Stabilitäts-Plastizitäts-Dilemma [15] zu lösen. Mithilfe von ART-Netzwerken können neue Assoziationen erlernt werden, ohne bereits bekannte Assoziationen zu verlernen. Dadurch werden diese Netzwerke häufig als inkrementelle Klassifikatoren eingesetzt werden.

Die grundlegende Intuition hinter ART Modellen ist die Idee, dass Aufgaben wie Objekt-Klassifizierung/-Erkennung beim Menschen typischerweise als Ergebnis der Interaktion von „*Top-Down*“ Erwartungen zu „*Bottom-Up*“ Eingangsdaten ablaufen. Dieses Verhalten wird auch als *Match-Based Learning* beschrieben [25]. Bei diesem Verfahren wird ein externer Input (z.B. das Bild von einer Kamera) mit internen Erinnerungen/Erwartung eines aktiven Codes verglichen. Wenn beispielsweise ein Hund von einem Menschen gesehen wird, wird das Objekt mit den bekannten Repräsentationen eines Hundes verglichen. Wenn das Objekt die Erwartungen eines Hundes erfüllt, wird das Objekt von einem Menschen als Hund erkannt.

Für ART-Modelle führt das zu dem im Folgenden erläuterten Lernalgorithmus:

Zu Beginn hat das neuronale Netz eine gewisse Anzahl an frei verfügbaren Neuronen. Die freien Neuronen dienen in einem ART-Netzwerk als Knoten für die spätere Klassifikation der Eingangsdaten. Solange keine Eingangsdaten vorliegen befindet sich das neuronale Netzwerk in einem passiven Zustand. Eintreffende Eingangsdaten werden mit bisher bekannten Mustern verglichen (*gematched*). Wenn das Ergebnis des sogenannten Matching zwischen den Eingangsdaten und den bekannten Repräsentationen einen Schwellwert übersteigt, werden diese Eingangsdaten der Klasse der Repräsentation zugeordnet. Das eingehende Muster ist somit bereits bekannt. Der Schwellwert wird in einem ART-Netzwerk als *Vigilance Parameter* bezeichnet. Nachdem die Eingangsdaten einer Klasse zugeordnet werden kann, kann entweder die Netzwerkrepräsentation dieser Klasse weiter angepasst werden oder die Repräsentation wird nicht verändert. Das ist relevant, wenn eine neue Klasse auf Basis von wenigen Samples erlernt wurde. Diese Klasse kann auch nach der „Repräsentation“ im Netzwerk weiter inkrementell trainiert und damit verbessert werden. Falls der Schwellwert nicht erreicht wurde, wird ein neuer Knoten (eine neue Repräsentation) im Netzwerk erstellt und mit den Eingangsdaten initialisiert. Durch dieses Verhalten können neue Klassen erkannt und erlernt werden. Falls keine freien Knoten/Neuronen mehr verfügbar sind, können lediglich bestehende Knoten angepasst werden. Dadurch ist das Netzwerk nicht mehr in der Lage neue Klassen zu erlernen. Das muss bei der Architektur eines ART-Netzwerk berücksichtigt werden [25], [26]. Eine graphische Darstellung des beschriebenen Ablaufs ist in Abbildung 12 gegeben.

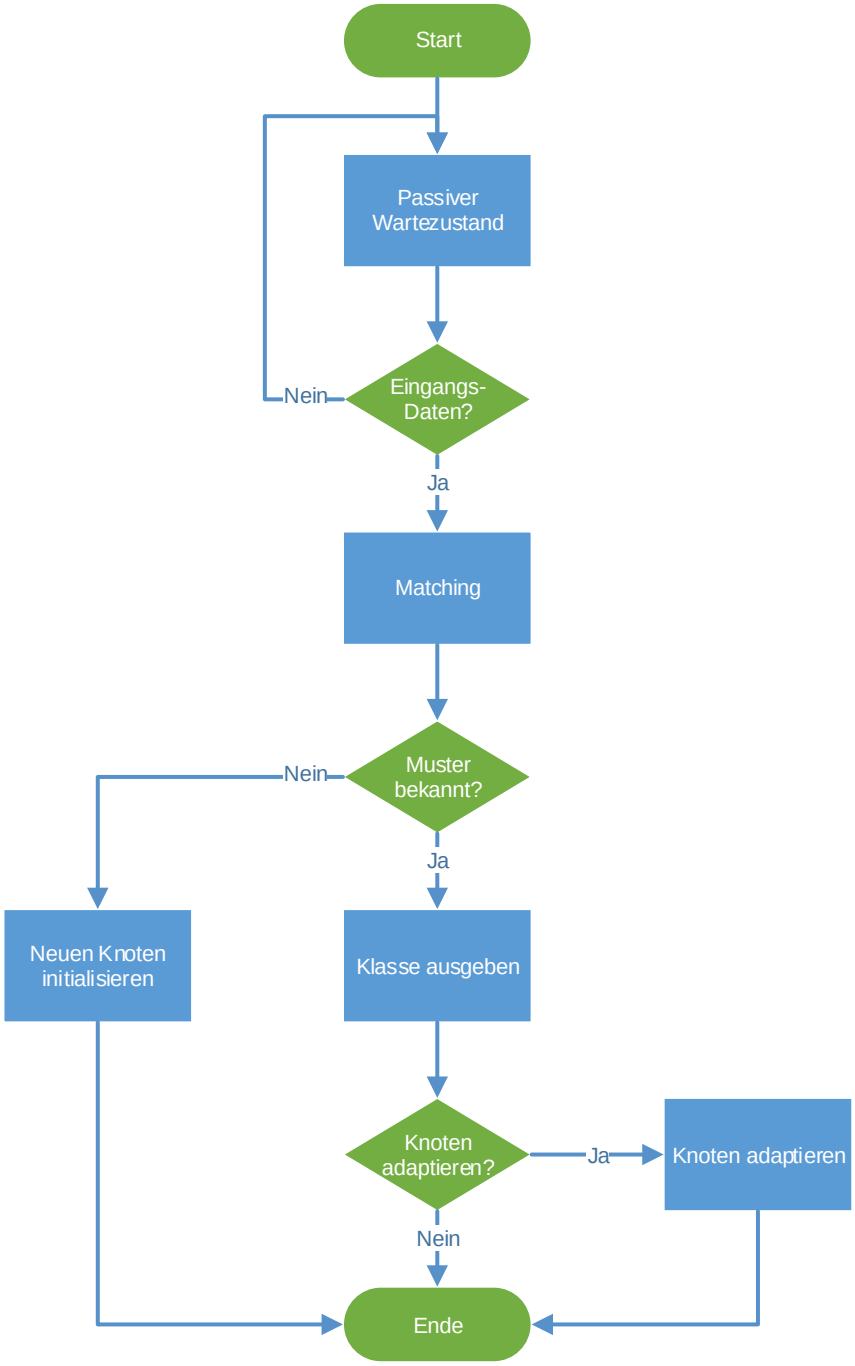

Abbildung 12: Ablaufdiagramm des Betriebs eines ART-Netzwerk

Zum Trainieren eines ART-Netzwerks gibt es zwei unterschiedliche Methoden, das langsame und das schnelle Training/Lernen. Das langsame Training ähnelt dabei mehr dem biologischen Prozess und nutzt Differenzialgleichungen zur kontinuierlichen und iterativen, aber langsamen Anpassung der Gewichte. Dies wird typischerweise bei kontinuierlichen Datenströmen eingesetzt, da dort die Eingangsdaten über einen längeren Zeitraum anliegen und somit das System die Möglichkeit hat sich langsam an den gewünschten Grenzwert anzupassen. Das schnelle Lernen (*Fast Learning*) ermöglicht dem System sich schnell auf seltene Eingangsdaten anzupassen. Dabei werden die Netzwerkparameter auf die berechneten asymptotischen Parameterwerte gesetzt [25], [26].

Wie bereits beschrieben gibt es unterschiedliche ART-Architekturen, die je nach Anwendungsfall variieren. Diese werden hier nicht einzeln detailliert aufgelistet.

In diesem Abschnitt wurden zwei beispielhafte inkrementelle Klassifikatoren vorgestellt und deren prinzipielle Betriebsweise erläutert. Auf Basis dieser eingeführten Klassifikatoren kann für den in dieser Arbeit relevanten Anwendungsfall ein passender inkrementeller Klassifikator ausgewählt werden.

2.4 Verteiltes Lernen

Verteiltes Lernen wird in der Literatur unter den Namen *Distributed* oder *Parallel Learning* beschrieben. Es gibt viele unterschiedliche Gründe warum verteiltes Lernen eingesetzt wird. Beispielsweise kann es aufgrund der Größe des Netzwerkes notwendig sein das Modell auf mehrere Prozessoren zu verteilen. Auch kann es wegen der langen Trainingszeiten von DNNs bei großen Datenmengen gewünscht sein, paralleles und verteiltes Training mit aufgeteilten Datensätzen durchzuführen und die verschiedenen trainierten Modelle nach dem Training (in der Literatur *post-training* genannt) zusammenzuführen. Ein anderer Anwendungsfall kann schließlich das Sammeln von riesigen Datenmengen auf lokalen verteilten Geräten (z.B. Smartphones) sein. Da nur begrenzter Speicher zur Verfügung steht und Datenschutzrichtlinien eingehalten werden müssen, können die lokal gesammelten Daten häufig nicht auf einen zentralen Server geladen werden, wo ein zentralisiertes Training stattfinden könnte. Deshalb kann es notwendig sein auf den jeweiligen Endgeräten verteilt und lokal zu trainieren (lernen) und lediglich Parameteränderungen auf einem Server zu sammeln, um am Ende ein besseres globales Netzwerk zu erhalten. Dadurch liegen sicherheitskritische Daten nicht zentral gesammelt auf einem Server, sondern verteilt auf den Endgeräten, wodurch das Sicherheitsrisiko verringert werden kann. In diesem Kapitel werden nachfolgend unterschiedliche Anwendungsgebiete, sowie die dazugehörigen Ansätze und Methoden des verteilten und parallelen Lernens eingeführt. Aufgrund der Vielzahl an unterschiedlichen Methoden werden nur ausgewählte Methoden vorgestellt.

Ursprünglich entstand der Wunsch nach verteiltem und parallelem Lernen durch die langen Trainingszeiten von komplexen neuronalen Netzwerken. Komplexe Netzwerke, die auf großen Datensätzen trainiert werden, können Tage bis Wochen auf einzelnen Prozessoren benötigen, um die Parametrisierung zu erlernen. Durch die Weiterentwicklung und Nutzung von *Graphical Processing Units* (GPUs) kann das Training von DNNs bereits deutlich beschleunigt werden. Dennoch kann durch paralleles, verteiltes Training diese Rechenzeit weiter reduziert werden. Zudem kann es auch vorkommen, dass Datensätze oder Modelle zu groß sind, um auf einem einzigen Gerät gespeichert zu werden.

Zur Einordnung kann vereinfacht zwischen lokalem und verteiltem Training unterschieden werden. Bei lokalem Training werden die Daten und das Modell auf einem einzelnen Gerät gespeichert. Es können mehrere Kerne dieses Geräts zur Parallelisierung genutzt werden. Zum Beispiel können unterschiedliche Kerne genutzt

werden um verschiedene Inputs parallel zu bearbeiten oder die unterschiedlichen Kerne können genutzt werden um mehrere Minibatches parallel zu prozessieren. Beim verteilten Training ist es nicht möglich und/oder nicht erwünscht den gesamten Datensatz oder das gesamte Modell auf einem einzelnen Gerät zu speichern. Hier wird das Modell auf mehrere Geräte verteilt.

Für eine feinere Aufteilung wird zwischen der Parallelisierung in Netzwerken allgemein und der Parallelisierung im Training von diesen Netzwerken unterschieden. Für die Parallelisierung in Netzwerken kann weiter zwischen Daten-Parallelisierung und Modell-Parallelisierung unterschieden werden. Bei der Daten-Parallelisierung werden die Daten auf verschiedene Geräte verteilt, wenn die Datenmenge zu groß ist oder ein schnelleres Training erwünscht ist. Modell-Parallelisierung wird angewendet, wenn das Modell speichertechnisch nicht auf einem Gerät gespeichert werden kann. Dabei werden dann unterschiedliche Schichten des neuronalen Netzwerkes auf unterschiedliche Geräte verteilt werden. Dies erfordert eine Kommunikation der Geräte, um die Daten durch das Netzwerk zu propagieren und auch um Backpropagation durchführen zu können [27].

In [28] wird genauer die Parallelisierung des Trainings von Netzwerken beschrieben. Im Rahmen dieser Arbeit sind besonders diese Methoden und Ansätze interessant, weshalb auf diese nun detaillierter eingegangen wird. In den bisher eingeführten Parallelisierungen/Verteilungen der Modelle oder Daten existiert lediglich eine Version der Parameter $\underline{\theta}$ des Netzes. In den nachfolgenden Methoden existiert jedoch generell mehr als eine Instanz der Netzwerk-Parameter. Die Ansätze können dabei in drei Kategorien unterteilt werden: Modell-Übereinstimmung, Parameter-Verteilung und Training-Verteilung. Zu jeder Kategorie gibt es verschiedene Methoden, von denen einige beispielhaft in Tabelle 1 zusammengefasst werden [28].

Tabelle 1: Übersicht über Verteilte Deep Learning Methoden

Kategorie	Methode
Model-Übereinstimmung	
Synchronisation	Synchron Asynchron Nicht-deterministische Kommunikation
Parameter-Verteilung und Kommunikation	
Zentralisierung	Parameter Server (PS) Dezentralisiert
Training-Verteilung	
Modell-Konsolidierung	Ensemble Lernen Wissens-Destillierung

Nachfolgend wird auf einzelne Methoden spezifischer eingegangen, wobei in einer späteren Anwendung die unterschiedlichen Methoden miteinander genutzt werden können, da sie unterschiedliche Aspekte des verteilten Lernens behandeln:

Modell-Übereinstimmung

In den Methoden, welche in der Kategorie Modell-Übereinstimmung zusammengefasst sind, werden Berechnungen der Update-Schritte parallel auf unterschiedlichen Knoten ausgeführt. Diese Methoden können als eine spezielle Form der Daten-Parallelisierung angesehen werden. Aktuelle Parameter auf einem Master-Gerät werden als konsistentes Modell angesehen. Das Master-Gerät kann dabei durch einen zentralen Parameter Server (PS) oder dezentralisiert auf unterschiedlichen Knoten realisiert werden. Bei synchronen Methoden senden alle Knoten zum gleichen Zeitpunkt ihre entsprechend berechneten Parameteränderungen, welche zentral zu einem neuen konsistenten Modell zusammengefasst werden. Dieses Modell wird wieder verteilt und die unterschiedlichen Knoten können den nächsten Zyklus (Minibatch) durchführen. Bei asynchronen Methoden findet diese Synchronisation asynchron zu unterschiedlichen Zeitpunkten statt. Bei einer nicht-deterministischen Kommunikation kann beispielsweise event-getriggert die Synchronisation stattfinden, z.B. nach einer gewissen Anzahl an Trainingsschritten [27], [28].

Parameter-Verteilung und Kommunikation

Eine zentralisierte Netzwerkarchitektur beinhaltet in der Regel eine PS-Infrastruktur. Mit dieser Infrastruktur senden die einzelnen Knoten ihre Änderungen, was im Fall eines DNN der errechnete Gradient ist, zu einem zentralen PS. Die eintreffenden Gradienten werden von dem zentralen PS benutzt, um neue Parameterwerte zu berechnen. Es gibt in diesem Szenario somit nur einen zentralen Optimierer. Die neu optimierten Parameter werden als Antwort an die unterschiedlichen Knoten verteilt. Bei einem dezentralisierten Ansatz besitzt jeder Knoten einen eigenen Optimierer, wodurch jeder Knoten separate Parameteranpassungen berechnet und durchführt. Die unterschiedlichen Knoten können untereinander kommunizieren und Parameter austauschen.

Generell ist eine PS Infrastruktur für die Leistung und die Fehlertoleranz des Netzwerkes förderlich, da mithilfe eines zentralisierten PS Checkpoints zentral gespeichert werden können. Bei Erkennen eines möglichen Overfittings oder anderen unterwünschten Trainingseffekten kann auf einen zuvor gespeicherten Checkpoint zurückgegangen werden. Dennoch müssen bei diesem Ansatz auch die Kommunikationskosten abgewogen werden, die durch einen zentralen PS im Vergleich zu einem dezentralen Ansatz entstehen [27], [28].

Training-Verteilung

In diesen Ansätzen finden nur selten und unregelmäßig Parameterupdates, beziehungsweise der Austausch von Parametern, statt. Es werden auf unterschiedlichen Knoten Kopien der Parameter angelegt und die durch das Training erhaltenen Parameter nach dem Training (*Post-Training*) oder einige Male während dem Training kombiniert. Eine bekannte und häufig genutzte Kombinationsmöglichkeit nach dem Training ist das Ensemble Lernen. Beim Ensemble Lernen werden mehrere Instanzen des Netzwerks angelegt und parallel und unabhängig voneinander trainiert.

Es findet keine Kommunikation zwischen den einzelnen Knoten während des Trainings statt. Die finale Ausgabe des Ensembles ist die kombinierte Ausgabe der einzelnen Netzwerke. Die Gewichtung kann dabei gleichmäßig geschehen, was dem Mittelwert der unterschiedlichen Netzwerkausgänge entspricht. Alternativ können die Ausgänge einzelner Netzwerke, welche als vertrauenswürdiger eingestuft werden, stärker gewichtet werden. Eine weitere *Post-Training* Methode ist die Wissens-Destillierung (*Knowledge-Distillation*). Bei dieser Methode wird die Größe des DNNs reduziert, indem ein zweistufiges Training stattfindet. Zunächst wird ein großes Netzwerk oder ein Ensemble von mehreren Netzwerken trainiert. Im zweiten Schritt wird ein neuronales Netzwerk trainiert, das den Ausgang des großen Ensembles imitiert. Mit diesen kleineren Netzwerken können häufig dieselben Ergebnisse wie mit größeren Ensembles erzielt werden [28].

Als zusätzliche, spezifische Methode wird föderiertes Lernen (*Federated* Learning) aufgezeigt. Diese Methode wurde 2016 in [29] vorgestellt. Das Ziel dieser Methode ist es, ein hochqualitatives, zentralisiertes Modell auf Basis vieler verteilter Netzwerke zu trainieren. Dabei liegen die Daten ungleichmäßig verteilt über die unterschiedlichen Knoten vor. Die lokalen Knoten werden dabei als Rechnerknoten benutzt, die mithilfe der lokal verfügbaren Daten Optimierungen durchführen. Diese Daten müssen dabei nicht auf einem zentralen Server gespeichert werden, sondern liegen nur auf den lokalen Knoten vor. Mithilfe dieses Aufbaus müssen mögliche private, sicherheitskritische Daten nicht auf einen Server geladen werden, was die Reduzierung des Sicherheitsrisikos zur Folge hat. Zudem kann mit dieser Methode der Kommunikationsaufwand zwischen den einzelnen Knoten und einem zentralen Server (globales Modell) sowie die Kommunikation unter den Knoten minimiert werden. In dem Ansatz des föderierten Lernens werden lediglich Anpassungen an den zentralen Server geschickt (z.B. Gradienten-Vektor), wodurch die benötigte Kommunikationsbandbreite im Vergleich zum Senden der kompletten Trainingsdaten drastisch reduziert wird. Gleichzeitig ist die geschickte Information deutlich abstrahierter von den möglicherweise personalisierten Daten. *Federated Learning* kann mithilfe der folgenden Notationen definiert werden:

Es gibt N Besitzer von Knoten (z.B. Smartphones), mit den gesammelten persönlichen Daten $\{D_1, \dots D_N\}$. In klassischen Ansätzen würden die Daten zusammengelegt mit $D = D_1 \cup \dots \cup D_N$, um ein zentrales Modell $\mathcal{M}_{SUM}$ zu trainieren. Ein föderiertes System ist ein lernender Prozess, in dem die einzelnen Knoten jeweils ein eigenes Modell $\mathcal{M}_{FED}$ trainieren. Dabei werden die Daten D_i der einzelnen Knoten i nicht mit den anderen Knoten geteilt. Zusätzlich soll die Genauigkeit der einzelnen Modelle, beschrieben durch Υ_{FED}, annähernd die Genauigkeit des hypothetischen zentralen Modells $\mathcal{M}_{SUM}$, Υ_{SUM}, erreichen. Mathematisch kann das mithilfe der nicht-negativen reellen Zahl δ in Gleichung **(14)** beschrieben werden:

$$|\Upsilon_{FED} - \Upsilon_{SUM}| < \delta \tag{14}$$

Ein föderiertes System hat somit einen Genauigkeitsverlust δ. Föderierte Systeme können weiter in unterschiedliche Anwendungsfälle kategorisiert werden, die in Tabelle 2 dargestellt werden [30].

Tabelle 2: Kategorisierung von *Federated Learning*

Kategorie	Beschreibung
Horizontales Federated Learning	Unterschiedliche Samples, Gleiche Features
Vertikales Federated Learning	Gleiche Samples, Unterschiedliche Features

Für ein besseres Verständnis werden konkrete Beispiele für die unterschiedlichen Kategorien gegeben:

Horizontales Federated Learning

> Zwei regionale Banken mit unterschiedlichen Benutzergruppen aus ihren jeweiligen Regionen haben eine geringe (oder keine) Überschneidung der Kunden. Das Geschäft der beiden Banken ist jedoch sehr ähnlich, wodurch die Features sehr ähnlich sind. Für eine bessere Generalisierung können die Parameter der unabhängig trainierten Netze nach dem Training ausgetauscht werden.

Vertikales Federated Learning

> Zwei unterschiedliche Firmen in der gleichen Stadt, eine Bank und ein Internetshop, haben eine sehr große Überschneidung bei den Nutzern. Die Features der beiden Firmen sind jedoch sehr unterschiedlich. Die Bank speichert zum Beispiel das monatlich einkommende Gehalt und das Kreditranking, während der Internetshop Browserverläufe und Einkaufsverhalten abspeichert. Durch das Verbünden beider trainierten Netzwerke kann mithilfe der Kundendaten das Einkaufsverhalten einzelner Gruppen/Personen genauer vorhergesagt werden

Im Rahmen dieser Arbeit ist das horizontale *Federated Learning* der relevante Anwendungsfall. Der prototypische Anwendungsfall in dieser Arbeit für das verteilte Lernen ist die Objekterkennung in Bildern. Die verschiedenen Bilder haben dieselben grundlegen Features wie z.B. Kanten, Farbe oder Form der Objekte. Jedoch sehen die unterschiedlichen Netzwerke unterschiedliche Bilder. Beispielsweise sieht Netzwerk A nur Bilder von Hunden und Katzen während des Trainings. Netzwerk B sieht dafür Kamele und Frösche. Nach dem Training sollen durch den Austausch von Parametern der föderierten Netzwerke beide Netzwerke in der Lage sein alle vier Tiere klassifizieren zu können.

In diesem Abschnitt wurden unterschiedliche Gründe und Anwendungsgebiete für den Einsatz von verteiltem Lernen vorgestellt. Zudem wurden Schwierigkeiten und unterschiedliche Methoden dargestellt. Diese Methoden wurden weiter in Kategorien

eingeteilt. Zum Ende wurde ein aktueller Algorithmus des verteilten Lernens beschrieben, der Lösungen für den im Rahmen dieser Arbeit untersuchten Anwendungsfall bereithält.

2.5 Lifelong Deep Neural Network Algorithmus

In dieser Arbeit wird ein L DNN Algorithmus untersucht und prototypisch umgesetzt. Der Algorithmus ist in [1] beschrieben. In diesem Kapitel wird der Algorithmus, sowie dessen Vor- und Nachteile aufgezeigt. Auch werden Behauptungen nach [2] über das Potenzial des Algorithmus genannt und in Relation zu bereits bekannten Methoden gesetzt. Zum Schluss wird eine kurze Zusammenfassung gegeben.

2.5.1 Beschreibung

Der L DNN Algorithmus soll den Bereich von Deep Learning revolutionieren, indem er schnelles Lernen nach Auslieferung ohne ausführliches Training, vielen Rechenressourcen oder extremer Datenspeicherung ermöglicht [1]. Dafür wurden mehrere Punkte gefunden, die bei bisherigen Deep Learning Ansätzen (welche den Backpropagation Algorithmus nutzen) ein Problem darstellen, um die oben genannten Punkte zu erfüllen. Nach [1] können die Probleme in den folgenden fünf Punkten zusammengefasst werden, welche den bisherigen Einsatz von Deep Learning Algorithmen eingrenzen:

- Es ist unmöglich das System „on-the-fly“ mit neuem Wissen up-zu-daten
- Lernen während dem gesamten Einsatzzyklus eines Gerätes ist ohne regelmäßige Kommunikation mit Servern und ohne eine erhebliche Wartezeit für ein Wissensupdate unmöglich
- Das Erlernen neuer Informationen verbraucht Serverspeicher, Energie und lokalen Gerätespeicher um alle Eingangsdaten unendlich lange für das weitere Training abzuspeichern
- Es ist unmöglich auf einem kleinen Endgerät zu lernen (trainieren)
- Es ist nicht möglich Wissen über mehrere Endgeräte auszutauschen ohne ein langsames und teures Training auf einem Server und Neuverteilung der erlernten Parameter

Diese beschriebenen Probleme sind bekannte Probleme des Deep Learnings, und wurden bereits in den vorhergehenden Kapiteln teilweise beschrieben.

Mit dem L DNN Algorithmus sollen diese Probleme überwunden werden können. Die Grundlage für den L DNN Algorithmus liegen in der Dual-Memory Methode (siehe Kapitel 2.2). Er besteht aus einem langsam und einem schnell lernenden System, welche im Folgenden genauer definiert werden.

Als erste Stufe wird ein langsam lernendes neuronales Netzwerk (z.B. CNN) genutzt (Modul A). Dieses Netzwerk ist ein vortrainierter Feature-Extrahierer. Je nach Anwendungsfall kann dieses Netzwerk während der Laufzeit mit kleinen, langsamen Updateschritten weiter trainiert werden oder es wird nach dem Vortraining fixiert. Modul A wird mithilfe des Backpropagation-Algorithmus auf Basis von repräsentativen Datensätzen vortrainiert. Ein beispielhafter Datensatz, der für das Vortrainieren solcher Feature-Extrahierer für die Bildverarbeitung genutzt wird, ist ImageNet [31]. Dieser Datensatz besitzt 1000 verschiedenen Klassen, wodurch eine gute Generalisierbarkeit erreicht werden kann.

Von dem vortrainierten DNN wird die letzte Feature-Schicht als Eingang des zweiten Moduls, Modul B, genutzt. Typischerweise wird als Interface die letzte Schicht des DNN vor der DNN-eigenen Klassifikationsschicht genutzt. Abbildung 13 zeigt ein beispielhaftes Modul A sowie das Interface zu Modul B. Jedes Rechteck ist dabei eine willkürliche (*Convolutional*, *Fully Connected*) Schicht eines DNN. Die letzte grüne Schicht stellt die Schicht dar, welche für die Klassifikation verantwortlich ist. Innerhalb des L DNN Algorithmus wird diese Schicht nicht genutzt. Die Features der vorherigen Schicht dienen als Eingang für Modul B. Somit stellt Modul A eine modifizierte Version des vortrainierten DNN dar.

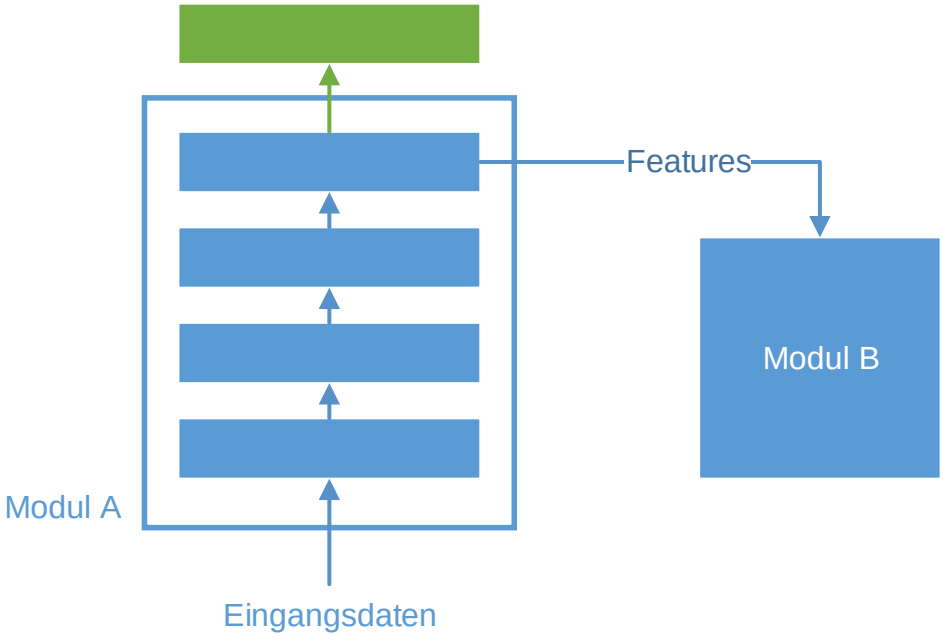

Abbildung 13: Modul A und Interface zu Modul B

Modul B wird durch einen inkrementellen Klassifikator realisiert, der Gewichte und Repräsentationen auf Basis von wenigen Trainingsdaten ändern kann. Modul B erhält die extrahierten Features von Modul A und kann mithilfe dieser z.B. eine Klassifikation durchführen. Falls die erhaltenen Repräsentationen nicht bekannt sind, bzw. zu keiner bekannten Klasse ausreichend ähnlich sind, kann Modul B durch Interaktion mit dem User das korrekte Label erhalten und aufgrund seiner schnell lernenden Eigenschaft diese neue Klasse mit einem (*One-Shot Learning*) oder wenigen Samples erlernen. In [1] genannte Beispiele für ein Modul B sind z.B. ein *Adaptive Resonance Theory* (ART)

Netzwerk oder als nicht-neuronale Methode die *Support Vector Machine* (SVM). Generell kann jeder schnell lernende, überwachte Klassifikator-Prozess für Modul B genutzt werden. Die grundsätzliche, allgemeine Architektur des L DNN Algorithmus ist in Abbildung 14 dargestellt.

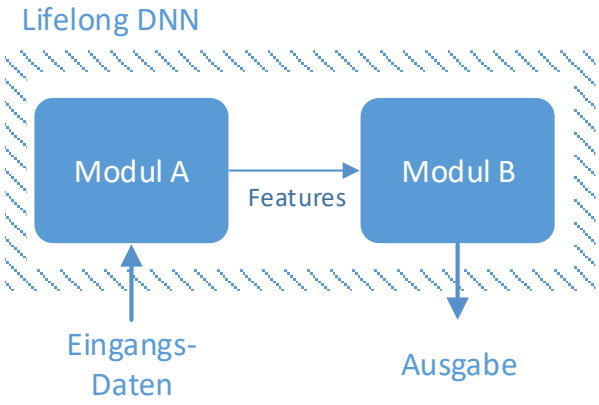

Abbildung 14: Graphische Darstellung des L DNN A

Je nach Anwendungsfall variieren die Eingangsdaten und die Ausgabe. Die konkreten Module A und B können nach Bedarf und auf der Basis der gegebenen Rahmenbedingungen gewählt und geändert werden.

Der Algorithmus ist so aufgebaut, dass auf mehreren Geräten parallel neues Wissen erlernt werden kann. Dieses Wissen kann durch die direkte Kommunikation zwischen einzelnen Geräten oder durch eine zentrale Steuerung über einen Server ausgetauscht werden. Dafür können beliebig viele parallele Geräte, die mit einem L DNN Algorithmus ausgestattet sind, verbunden werden. Ein beispielhaftes Szenario ist in Abbildung 15 dargestellt. In diesem Szenario sind einzelne Geräte (in der Graphik die beiden linken Geräte) direkt miteinander verbunden. Zusätzlich gibt es einen zentralen Server, der mit allen Geräten verbunden ist und von diesen Updates erhält. Dieser zentrale Server kann wiederum dann kombinierte Parameterupdates verteilen.

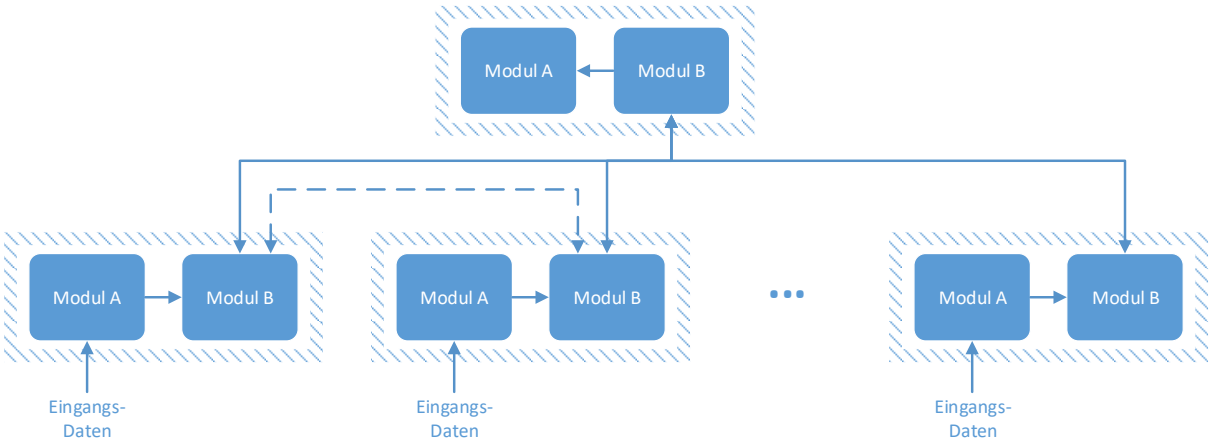

Abbildung 15: Beispielhaftes Szenario mit mehreren Endgeräten und einem zentralen Server

Wie zu sehen ist, tauschen sich die schnell lernenden Module B aus, da dort das sich ändernde Netzwerk implementiert ist. So können Objektklassen, die von einem Gerät gesehen und erlernt werden, mit anderen Geräten geteilt werden. Dadurch erhalten die anderen Geräte ebenfalls die Fähigkeit diese neue Objektklasse bestimmen zu können, ohne jemals ein Objekt dieser Klasse gesehen zu haben.

Der Aufteilung in Kapitel 2 folgend ist es bei diesem Algorithmus sinnvoll, die zwei Themen des kontinuierlichen und des verteilten Lernens separat zu behandeln. Diese beiden Themen werden mit Bezug auf den beschriebenen Algorithmus im Folgenden detaillierter untersucht.

Kontinuierliches Lernen

Der grundlegende Aufbau des L DNN Algorithmus in Bezug auf das kontinuierliche Lernen basiert auf der Dual-Memory Methode. Es werden zwei Submodule genutzt. Ein langsam lernendes oder fixes Modul A und ein schnell lernendes Modul B, das neue spezifische Informationen schnell erkennen und erlernen soll. Die grundlegende Idee dieser Architektur kommt von dem Gehirn der Säugetiere (Details in Kapitel 2.2). Für das kontinuierliche Lernen ist hauptsächlich Modul B relevant, da Modul A als Feature-Extrahierer fix ist oder sich nur sehr langsam ändert. Die Aufgabe, welche mit diesem Ansatz in dieser Arbeit gelöst werden soll ist das inkrementelle Klassen Lernen (*Incremental Class Learning*). Die grundlegende Aufgabe, z.B. Objekterkennung, bleibt während des Lebenszyklus des Algorithmus identisch. Es sollen jedoch inkrementell neue Klassen erlernt werden können. Modul B ist deshalb ein schnell lernender Klassifikator, der für manche Klassen bereits vortrainiert sein kann oder komplett untrainiert während des Betriebes „on-the-fly“ trainiert werden kann. Für das Training *on-the-fly* ist ein Feedback des Nutzers nötig, um korrekte Labels und Klassenbezeichnungen zu erhalten. Das gewünschte Verhalten von Modul B wird im Folgenden erläutert:

Wenn Feature-Vektoren von Modul A eintreffen, soll auf Basis der bereits bekannten Klassen die Klassenzugehörigkeit ermittelt werden. Wenn ein Label (z.B. User-Feedback im Anwendungsfall) für dieses Sample vorhanden ist, soll eine weitere Anpassung für die Klasse stattfinden, um eine bessere Generalisierbarkeit zu erreichen. Somit soll auch für bereits bekannte Klassen kontinuierlich weitergelernt werden. Für den Fall, dass die Klasse des Samples nicht bekannt ist, wird in dem L DNN Algorithmus das „*Nothing I know*“-Konzept vorgeschlagen. Dafür wird ein Schwellwert für die Klassenzugehörigkeit berechnet, der erreicht werden muss damit die Klasse dem Sample zugeordnet wird. Wenn dieser Schwellwert nicht erreicht wird, ordnet der Algorithmus dieses Sample der Klasse „*Nothing I know*“ zu. In diesem Fall wird der User aufgefordert, die Klasse des Sample zu benennen, damit die neue Klasse auf Basis des Samples erlernt werden kann (*One-Shot Learning*). In [1] wird für die Berechnung des Schwellwertes θ folgende Gleichung **(15)** vorgeschlagen:

$$\theta = s\,\frac{C}{N} \tag{15}$$

Dabei stellt C die Anzahl an bereits erlernten Kategorien (Klassen) dar und N die gesamte Anzahl an Kategorie-Knoten im Netzwerk. Der Skalierungsfaktor s wird auf Basis des genutzten DNN in Modul A angepasst und gesetzt. Wenn dieser Wert zu hoch gewählt wird, steigt die False Negative Rate, da potentiell zu viele neue Klassen erstellt werden. Ein zu niedriger Wert von s vergrößert die False Positive Rate, da Samples von neuen, bisher unbekannten Klassen eher zu bekannten Klassen zugeordnet werden. Ein Klassifikator-Netzwerk, welches diese Anforderungen erfüllen kann, wird schließlich als Modul B genutzt. Die konkrete Umsetzung für das Modul B wird in Kapitel 3.2 detailliert untersucht und beschrieben.

Verteiltes Lernen

Der L DNN Algorithmus nutzt von seinem Aufbau das Prinzip der Trainings-Verteilung. Auf mehreren Endgeräten liegen jeweils unabhängige Kopien der Parameter vor, welche dort lokal angepasst werden und gegebenenfalls direkt zwischen den Geräten oder über einen zentralen Server ausgetauscht werden können (siehe Kapitel 2.4). Wie in Abbildung 15 exemplarisch dargestellt, sind die unterschiedlichen Module B für den Austausch der Parameter relevant, da in diesen Modulen die Parameter während der Benutzung kontinuierlich angepasst werden. Das Grundprinzip dabei ähnelt dem *Federated Learning* [29], [30]. Es treten auf mehreren Geräten lokale Daten auf, welche jedoch nur genutzt werden, um das lokale Netzwerk zu trainieren. Diese geänderten und erlernten Parameter können wiederum ausgetauscht werden, um neues Wissen mit anderen Geräten zu teilen. Nach der Kategorisierung aus Tabelle 2 kann der hier genutzte Anwendungsfall in die Kategorie des horizontalen *Federated Learning* eingegliedert werden, da die genutzten Features von Modul B identisch sind (aufgrund desselben Moduls A), jedoch unterschiedliche Samples gesehen werden.

Der L DNN Algorithmus besitzt einen zusätzlichen Schritt zur effizienten Kommunikation neuen Wissens, die Konsolidierung. Dieser Schritt wird genutzt, um eine effiziente Struktur der einzelnen Klassifikatoren zu erzielen, wodurch der Speicherbedarf gesenkt werden kann und dadurch der Kommunikationsaufwand für die Parameterupdates verringert werden kann. Die Konsolidierung findet statt, nachdem eine oder mehrere Objekte *on-the-fly* gelernt wurden. Dabei werden die Repräsentationen der neuen Objekte komprimiert und diese neuen Objekte mit den Repräsentationen bereits bekannter Objekte integriert. Dadurch soll die Generalisierung des Netzwerks verbessert und der Speicherbedarf reduziert werden. Dieses konsolidierte Wissen der einzelnen Netzwerke kann dann kommuniziert werden. Dafür können zu willkürlichen Zeitpunkten (z.B. nachdem neue Klassen erlernt wurden, nach einem definierten Zeitraum etc.) die konsolidierten Parameter des schnell lernenden Moduls B ausgetauscht werden. Dies kann geschehen, indem alle Geräte ihre Parameter einem zentralen Server/Gerät senden oder einzelne Geräte sich „peer-to-peer“ untereinander austauschen. Wenn in einem zentralen Punkt (auf einem Server oder einem Mastergerät) alle Parameter zusammentreffen, kann die Fusion der einzelnen erlernten Parameter stattfinden. Beim Kombinieren der Parameter können Redundanzen zwischen einzelnen Geräten entfernt werden und der Speicherbedarf wird durch das Kombinieren mehrerer Parameter im Vergleich zu

einem naiven Ansatz, bei dem alle Repräsentationen aneinandergehängt werden, gesenkt. Zudem kann dabei die Generalisierung weiter verbessert werden, da Einflüsse von unterschiedlichen Daten die Parameter der einzelnen Geräte beeinflusst haben. Weiterhin soll die Genauigkeit des Systems bestehen bleiben und durch das Kombinieren im besten Fall weiter verbessert werden. Diese kombinierten Parameter können wiederum als Parameterupdates an alle Geräte versendet werden. Alternativ kann auch das zentrale Netzwerk für weitere Anwendungen genutzt werden. Durch das Verteilen der kombinierten Parameter kann das erlernte Wissen vieler verteilter Geräte konsolidiert werden und neues Wissen der einzelnen Geräte mit anderen Geräten ausgetauscht werden.

2.5.2 Vorteile

Der dargestellte Algorithmus bietet gegenüber klassischen DNN-Architekturen und Algorithmen einige Vorteile, die im Folgenden genannt und erläutert werden [1], [2]. Der große beschriebene Vorteil des L DNN Algorithmus ist seine Fähigkeit, kontinuierlich zu lernen ohne dabei dem Problem des *Catastrophic Forgetting* zu unterliegen. Maßgeblich dafür entscheidend ist, dass für das schnelllernende Modul B kein Backpropagation Algorithmus genutzt wird. Dies kann durch die zweistufige Architektur erzielt werden (Dual-Memory Methode). Mithilfe des Algorithmus können inkrementell neue Klassen schnell erlernt werden, ohne dafür alte Trainingsdaten zu benötigen. Deshalb ist keine Speicherung großer Datenmengen notwendig. Zudem soll es dadurch möglich sein auf einem lokalen Endgerät (z.B. Smartphone) ohne große Rechen- oder Speicherkapazität weiter zu lernen. Aufgrund der beschriebenen Architektur sollen deutlich weniger Instanzen pro Klasse notwendig sein, um die Klasse zu erlernen, wodurch die Trainingszeiten von einzelnen Klassen erheblich reduziert werden können. Aufgrund dessen soll es möglich sein diesen Algorithmus in Echtzeit zu trainieren. In Tabelle 3 sind die genannten Behauptungen im Vergleich zu einem traditionellen DNN nach [2] dargestellt.

Tabelle 3: Übersicht der behaupteten Vorteile des L DNN Algorithmus gegenüber traditionellen DNNs

	Traditionelles DNN	Lifelong DNN Algorithmus
Kontinuierliches Lernen	Nein	Ja
Lernen auf Endgeräten	Nein	Ja
Datenspeicherung notwendig nach Auslieferung	Ja	Nein
Datenanforderung	1000 – 10000 Instanzen pro Klasse Tausende von Präsentationen pro Instanz	10 – 100 Instanzen pro Klasse Eine Präsentation pro Instanz
Trainingszeit	Tage – Wochen – Monate	Sekunden

Ein weiterer beschriebener Vorteil der Architektur ist, dass auch bei mehreren verteilten Geräten kein Austausch der Daten notwendig ist. Es werden lediglich Parameter ausgetauscht, welche zwar in gewisser Weise eine abstrahierte Darstellungsform der Daten sind, jedoch keinen direkten Rückschluss auf die Daten zulassen. Damit kann auf Basis von sicherheitskritischen oder persönlichen Daten erlerntes Wissen ohne Versenden, kostspieliger Speicherung und/oder Sicherung dieser Daten ausgetauscht werden.

2.5.3 Nachteile

Da die vorhandene Literatur zu diesem Algorithmus von der verfassenden Firma (Neurala) geschrieben wurde, werden lediglich positive Aspekte ausdrücklich erwähnt und detailliert beschrieben. Veröffentlichungen, welche diesen Ansatz kritisch untersuchen, sind bisher nicht vorhanden. Deshalb kann hier noch nicht von konkreten Nachteilen gesprochen werden, da auf Basis der verfügbaren Literatur nur Vorteile gegenüber klassischen Ansätzen gegeben sind. Dennoch werden in diesem Kapitel Punkte genannt, die nicht zwingend ein Nachteil darstellen, jedoch für eine korrekte und faire Betrachtung des L DNN Algorithmus genannt werden müssen.

Dazu muss zunächst genannt werden, dass trotz dem schnell lernenden Modul B dennoch ein klassisches, zeit- und rechenaufwändiges DNN-Training stattfinden muss. Modul A, welches die Features für Modul B extrahiert, muss vortrainiert werden. Um eine gute Generalisierung zu erzielen sollten möglichst viele unterschiedliche Trainingsdaten (hier Bilder) gesehen werden. Dafür wird in der Regel der ImageNet-Datensatz genutzt, welcher ca. 1,3 Millionen Trainingsbilder umfasst. Abhängig von der gewünschten Netzwerkarchitektur kann das Training dieser Architektur auch hier einige Stunden bis Tage in Anspruch nehmen. Zudem kann durch das vortrainierte Modul A keine weitere Aufgabe durch den L DNN Algorithmus erlernt werden. Das DNN in Modul A ist beispielsweise für die Objektklassifizierung vortrainiert und erreicht dort sehr gute Performanz. Jedoch wird im weiteren Verlauf lediglich Modul B kontinuierlich angepasst, wodurch die Aufgabe fixiert ist. Dies muss jedoch kein Nachteil sein, da in der Regel der Anwendungsfall bereits vorher bekannt ist.

2.5.4 Zusammenfassung und Vergleich zu klassischen Ansätzen

Auf Basis der in Kapitel 2 dargestellten Grundlagen zu den einzelnen Gebieten (Deep Learning, Continual Learning, Distributed Learning sowie inkrementelle Klassifikatoren) kann gesagt werden, dass bekannte Methoden und Ansätze als Grundlage für den L DNN Algorithmus dienen und diese dort zu einem neuen Algorithmus kombiniert werden.

Die grundlegende Architektur basiert auf bereits bekannten Dual-Memory Methoden, mit einem langsam (oder nicht) lernenden Modul A und einem schnell lernenden Modul B. Modul A ist dabei ein typisches DNN, welches genutzt wird, um sinnvolle Features für Modul B auf Basis der Eingangsdaten zu extrahieren. Dieses DNN wird mithilfe klassischer Deep Learning Algorithmen (z.B. Backpropagation) trainiert.

Modul B ist ein schnell lernendes Klassifikator-Netzwerk, welches die Fähigkeit besitzt sich anzupassen und neue Klassen mit aufnehmen zu können. Neue Klassen werden über ein sogenanntes „*Nothing I know*“-Konzept erkannt und benötigen ein User-Feedback, um diesen Klassen sinnvolle Labels zuweisen zu können.

Wenn mehrere Geräte denselben L DNN Algorithmus nutzen, können diese Geräte sich untereinander austauschen und ihr Wissen weitergeben. Die grundlegende Idee folgt dabei dem Ansatz des *Federated Learning*. Mit diesem Ansatz können sich einzelne Netzwerke auf Basis lokal verfügbarer Trainingsdaten separat trainieren. Die erlernten und geänderten Parameter werden wiederum mit einem zentralen Server oder direkt unter den einzelnen Geräten ausgetauscht. Dadurch kann das Wissen konsolidiert und über alle Geräte verteilt werden. Mithilfe dieses Ansatzes können Geräte unabhängig von den Daten ihr erlerntes Wissen austauschen und sicherheitskritische oder persönliche Daten müssen nicht langfristig auf einem Server hochgeladen und gespeichert werden.

Durch die Kombination dieser bereits bekannten Ansätze besitzt der L DNN Algorithmus das theoretische Potenzial eine Vielzahl von Anwendungen bedienen zu können. Dieses Potenzial soll nun im weiteren Verlauf der Arbeit mithilfe des beispielhaften Anwendungsfall der Objekterkennung untersucht und bewertet werden.

3 Konzeption

In diesem Kapitel wird die konkrete Konzeption des L DNN Algorithmus beschrieben. Dafür werden zunächst unterschiedliche Architekturen untersucht und verglichen. Schließlich wird pro Modul eine konkrete Architektur ausgewählt, welche im weiteren Verlauf der Arbeit implementiert wird. Im Folgenden werden nun unterschiedliche Architekturen separat für die einzelnen Module A und B bewertet.

3.1 Modul A

Modul A ist das langsame (oder nicht) lernende Modul innerhalb des L DNN Algorithmus (siehe Kapitel 2.5). Im Rahmen dieser Arbeit werden fixe vortrainierte DNN-Architekturen genutzt, da das Training solcher DNN-Architekturen sehr zeit- und rechenaufwändig ist und es eine Vielzahl an vortrainierten DNN frei verfügbar gibt.

Es gibt viele DNN-Architekturen, die heutzutage zur Feature-Extraktion eingesetzt werden. Jede Architektur erfüllt dabei unterschiedliche Anforderungen, beziehungsweise nutzt unterschiedliche Ansätze, die für den einen oder anderen Anwendungsfall besser geeignet sind. Für Modul A innerhalb des L DNN Algorithmus ist es wichtig, dass die extrahierten Features eine bestmögliche Klassifikation erlauben. Zudem soll das System auch in Echtzeitsystemen auf mobilen Endgeräten (z.B. Smartphone) funktionieren [1], [2]. Dafür muss Modul A Eingangsdaten schnell verarbeiten können und gleichzeitig darf der Speicher- und Rechenbedarf des Netzwerkes nicht zu groß sein, da sowohl Speicher als auch Rechenleistung auf mobilen Endgeräten begrenzt ist.

Im Folgenden werden bekannte Klassifikator-Netzwerke auf Basis von DNN-Architekturen vorgestellt und auf deren Einsetzbarkeit in dieser Arbeit untersucht. Als Grundlage für den späteren Vergleich und für die genannten Metriken wird der Bilddatensatz ImageNet [31] genutzt. Mit ca. 1,3 Millionen Trainingsbildern aus 1000 verschiedenen Klassen stellt ImageNet ein sehr komplexes Problem dar. Vortrainierte Netzwerke, welche zur Extraktion von Features aus Bildern genutzt werden, sind heutzutage auf diesem Datensatz trainiert, da er aufgrund seiner großen Anzahl an Trainingsbildern und Klassen eine gute Generalisierung der extrahierten Features gewährleistet.

3.1.1 AlexNet

AlexNet war eines der ersten DNNs welches die Klassifikationsgenauigkeit auf ImageNet signifikant erhöhte. Im Rahmen der ILSVRC 2012 (*ImageNet Large Scale Visual Recognition Competition*) erreichte AlexNet einen Top-5 Klassifikationsfehler von 15,3%. Der zweitplatzierte dieses Wettbewerbs erreichte lediglich einen Top-5 Klassifikationsfehler von 26%. Es konnte eine erhebliche Steigerung der Klassifikationsgenauigkeit erzielt werden. Dabei ist AlexNet im Vergleich zu aktuellen Architekturen sehr simpel, mit lediglich fünf Convolutional, verschiedenen Max-Pooling

sowie drei Fully Connected Schichten [32], [33]. Neuerungen im Vergleich zu damaligen DNNs war der Einsatz von ReLU-Aktivierungsfunktionen sowie die Regularisierung des Netzwerks mithilfe von Dropout [34]. Beide Methoden sind seitdem in nahezu allen DNN-Architekturen im Einsatz. Das Netzwerk besteht aus 60 Millionen Parametern und 650.000 Neuronen. In Abbildung 16 ist die Modellarchitektur des AlexNet graphisch dargestellt, bei welcher die fünf Convolutional und die drei Fully Connected (Dense) Schichten zu sehen sind.

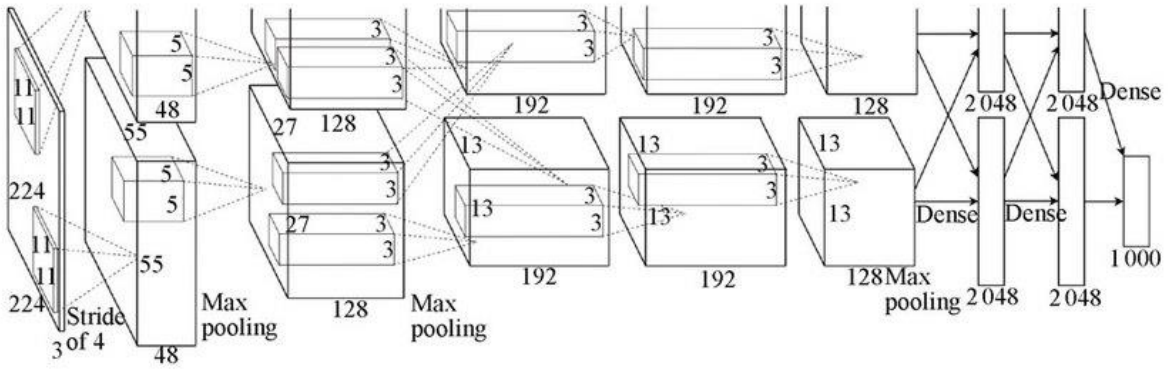

Abbildung 16: Modelarchitektur des AlexNet [33]

Die *Default*-Größe eines Parameters (Floating Point oder Integer) in TensorFlow ist mit 32-bit angegeben [35]. Unter der Annahme dieser Parametergröße benötigt AlexNet mit 60 Millionen Parametern 240 MB Speicher zur Sicherung des Modells. Um die Berechnungskomplexität vergleichen zu können wird die notwendige Anzahl an Operationen für ein einzelnes Eingangselement (z.B. einzelnes Bild) genutzt. Dabei wird lediglich der Vorwärtspfad in Betracht gezogen, wie es bei der späteren Anwendung (*Inference*) des Netzwerkes auch wäre. Diese Anzahl ist bekannt als Nummer von FLOPs (**F**loating **P**oint **OP**erations). AlexNet benötigt 727 Mega-FLOPs um alle notwendigen Berechnungen im Vorwärtspfad durchzuführen.

3.1.2 VGG

Die sogenannten VGG-Architekturen entstammen der *Visual Geometry Group* (VGG) der Universität Oxford. Dabei gibt es einige bekannte Architekturen. Die am häufigsten genutzten sind das VGG-16 sowie das VGG-19 Netzwerk. Die Zahlen stehen dabei für die Anzahl an Gewichts-/Parameterschichten innerhalb des DNN. Von der grundsätzlichen Architektur ähnelt es dem bereits beschriebenen AlexNet. Der große Unterschied ist der Einsatz kleinerer Kernel-Filter in den *Convolutional* Operationen (3x3 Kernel anstatt z.B. 11x11 oder 5x5 Kernel in AlexNet). Zudem werden mehrere dieser *Convolutional* Schichten hintereinander gesetzt, wodurch die finale Funktion des DNN diskriminativer werden soll. Ein großer Vorteil dieser Architektur ist die geringere Anzahl an Parametern. Unter der Annahme, dass der Eingang und Ausgang eines *Convolutional Stacks* mit drei 3x3 *Convolutional* Schichten C Kanäle besitzt, werden $3 * (3 * 3 * C * C) = 27C^2$ Parameter benötigt. Wird lediglich eine einzelne *Convolutional* Schicht mit 7x7 Kernel-Filtern genutzt, werden $7 * 7 * C * C = 49C^2$ Parameter benötigt [34]. Durch diese Parameterreduzierung können tiefere Netzwerke

mit mehr Schichten aufgebaut werden, ohne einen signifikant erhöhten Speicherbedarf zu haben. Durch den Anstieg der Tiefe des Netzwerks ist das Netzwerk fähig, komplexere Features zu erlernen [36]. Unterschiedliche Architekturen der VGG-Netzwerke für den ImageNet Bilddatensatz mit 1000 Klassen und RGB Bildern der Größe 224x224 sind in Tabelle 4 dargestellt.

Tabelle 4:VGG-Netzwerk Architekturen [36]

A 11 Gewichtsschichten	B 13 Gewichtsschichten	C 16 Gewichtsschichten	D 19 Gewichtsschichten
Eingangssignal (z.B. 224x224 RGB Bild)			
Conv3-64	Conv3-64 Conv3-64	Conv3-64 Conv3-64	Conv3-64 Conv3-64
Max-Pooling			
Conv3-128	Conv3-128 Conv3-128	Conv3-128 Conv3-128	Conv3-128 Conv3-128
Max-Pooling			
Conv3-256 Conv3-256	Conv3-256 Conv3-256	Conv3-256 Conv3-256 Conv3-256	Conv3-256 Conv3-256 Conv3-256 Conv3-256
Max-Pooling			
Conv3-512 Conv3-512	Conv3-512 Conv3-512	Conv3-512 Conv3-512 Conv3-512	Conv3-512 Conv3-512 Conv3-512 Conv3-512
Max-Pooling			
Conv3-512 Conv3-512	Conv3-512 Conv3-512	Conv3-512 Conv3-512 Conv3-512	Conv3-512 Conv3-512 Conv3-512 Conv3-512
Max-Pooling			
FC-4096			
FC-4096			
FC-1000			
Softmax			

Wie bereits geschrieben werden die Netzwerke anhand der Anzahl an Gewichtsschichten (*Convolutional* oder *Fully Connected* (FC) Schicht) benannt. So ist Netzwerk C als VVG-16 bekannt und Netzwerk D als VGG-19. Die häufig eingesetzten Feature-Extrahierer VGG-16 und VGG-19 werden im Folgenden bezüglich ihres Speicherbedarfes und der Klassifikationsgenauigkeit auf ImageNet untersucht. Dabei

sind in Tabelle 5 die Werte der unterschiedlichen Kriterien für die einzelnen Netzwerke gelistet [36].

Tabelle 5: Vergleich von VGG-16 und VGG-19

Netzwerk-Architektur	Anzahl Parameter	Speicher-bedarf Parameter	Anzahl FLOPs	Top-1 Klassifikations-fehler	Top-5 Klassifikations-fehler
VGG-16	$138x10^6$	552 MB	$16x10^9$	25,6 %	8,1 %
VGG-19	$144x10^6$	576 MB	$20x10^9$	25,5 %	8,0 %

Die Anzahl der Parameter und damit auch der Speicherbedarf steigt mit der Anzahl an Schichten. Hinsichtlich des Klassifikationsfehlers auf ImageNet ist zu beobachten, dass trotz drei zusätzlichen Schichten im VGG-19 Netzwerk ein nahezu vernachlässigbarer Genauigkeitsgewinn gegenüber VGG-16 erreicht werden kann. Es werden 6 Millionen zusätzliche Parameter (24MB zusätzlicher Speicher) sowie $4x10^9$ zusätzliche FLOPs benötigt, um den Top-1 und Top-3 Klassifikationsfehler um 0.1% zu verringern.

3.1.3 ResNet

Aufgrund der bisher beschriebenen Modelle kann die Aussage getroffen werden, dass eine größere Tiefe des Netzwerks eine bessere Klassifikationsgenauigkeit ermöglicht. Bei diesen Überlegungen muss jedoch berücksichtigt werden, dass bei der Backpropagation der Error-Vektoren durch das Netzwerk viele Schichten durchlaufen werden müssen. Dies führt dazu, dass der Error, der bei frühen Schichten ankommt, sehr klein oder gleich null ist und damit nur eine geringe oder gar keine Anpassung der Parameter stattfindet. Dieses Problem ist in der Literatur unter dem Namen *Vanishing Gradient* (Verschwindender Gradient) bekannt. Zusätzlich wird bei sehr tiefen neuronalen Netzwerken das Degradations-Problem beobachtet. Mit einer zunehmenden Netzwerk-Tiefe findet eine Sättigung in Bezug auf die Genauigkeit des Netzwerkes statt. Ab einem gewissen Punkt degradiert diese Genauigkeit schließlich. Dieses Verhalten lässt darauf schließen, dass nicht alle Netzwerke beliebiger Architektur gleich zu trainieren sind.

Um diese genannten Probleme zu lösen, wurden *Deep Residual Networks*, häufig nur als ResNet (Residual Network) bezeichnet, eingeführt [37]. Diese ResNets nutzen dieselben grundlegende Schichten wie VGG-Netze mit kleinen Kernel-Filter (3x3) und hauptsächlich *Convolutional* Schichten. ResNets setzen zusätzlich eine Vielzahl von *Residual* Blöcken ein. Ein solcher *Residual* Block ist beispielhaft in Abbildung 17 dargestellt.

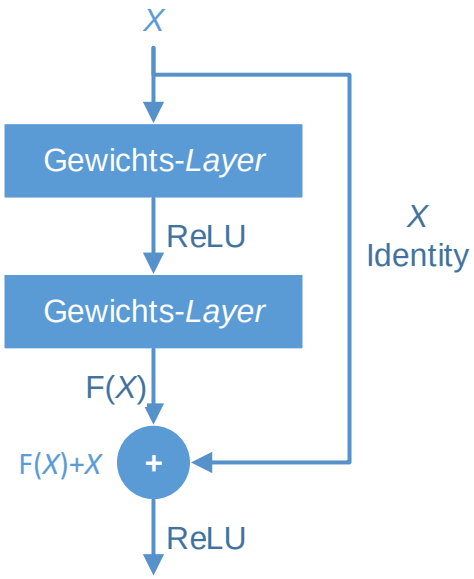

Abbildung 17: Beispielhafter Residual Block [37]

Diese *residual* Blöcke verwirklichen das gewünschte Verhalten $H(\boldsymbol{X})$ des Eingangssignals $\boldsymbol{X}$ mithilfe der nicht-linearen Verarbeitung $\mathcal{F}(\boldsymbol{X})$ und der Hinzunahme des ursprünglichen Eingangssignals $\boldsymbol{X}$, wodurch gilt: $H(\boldsymbol{X}) = \mathcal{F}(\boldsymbol{X}) + \boldsymbol{X}$. Die Hinzunahme des Eingangssignals wird mithilfe von sogenannten *Shortcut Connections* realisiert. In den meisten ResNets entspricht die *Shortcut Connection* dabei dem Identity Mapping. In der Theorie kann diese Verbindung jedoch auch einen zusätzlichen beliebigen Operator beinhalten. Mithilfe dieser Netzwerke können zum einen sehr tiefe Netzwerke (mehr als 100 Schichten) effizient und schnell trainiert werden, zum anderen kann die Genauigkeit dieser Netzwerke weiter erhöht werden, da das Potenzial der zusätzlichen tiefen Schichten genutzt wird und keine Degradation stattfindet. Damit kann sowohl das Problem des *Vanishing Gradient*, als auch das *Degradation Problem* gelöst werden [37].

Ein Ausschnitt einer beispielhaften ResNet-Architektur ist in Abbildung 18 gegeben. Es sind exemplarisch nur die ersten Schichten dargestellt. Links ist als Referenz die Architektur des VGG-19 Netzwerks zu sehen. In der Mitte ist ein „normales“ DNN mit 34 Schichten gezeigt und rechts das dazu passende ResNet, ebenfalls mit 34 Schichten aber bestehend aus *residual* Blöcken.

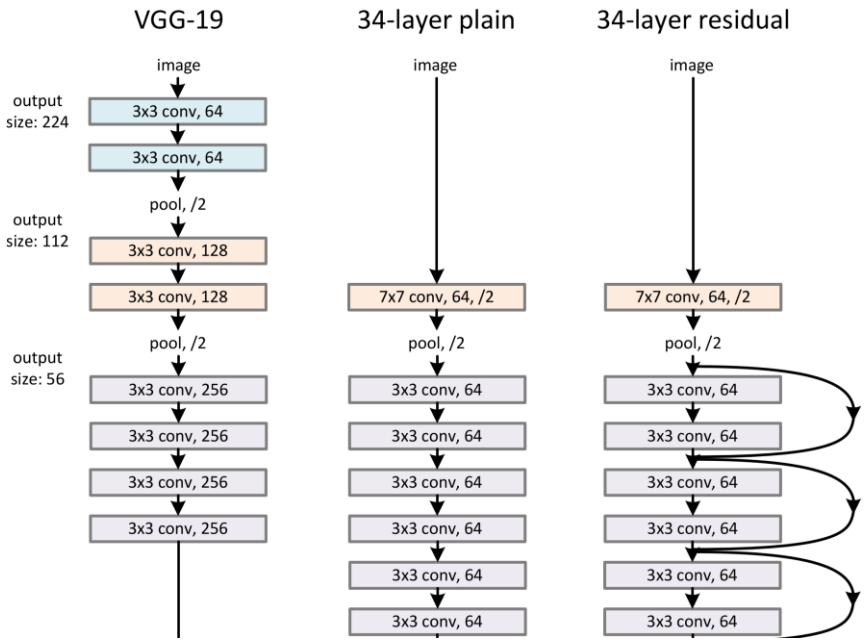

Abbildung 18: Beispielhafte Architektur VGG-19 (links), Standard-Architektur (Mitte), ResNet (rechts) [37]

Für das Beispiel von 34 Schichten kann auf Basis des ImageNet Datensatzes der Top-1 Klassifikationsfehler von 28.54% im Falle des *plain* Netzwerks auf 25.03% mithilfe des ResNets reduziert werden. Die gesamte Modellarchitektur (Anzahl Parameter, Schichten, Multiplikationen…) ist dabei identisch, lediglich die *Shortcut Connections* sind zusätzlich eingefügt. Ein ResNet kann mit einer beliebigen Anzahl an Schichten aufgebaut werden. In dieser Arbeit werden als Referenzen eine Architektur mit 50 und 101 Schichten angegeben. Die Anzahl an Parameter, der benötigte Speicherbedarf bei Umsetzung in TensorFlow [35], sowie der Top-1 und Top-5 Klassifikationsfehler auf dem ImageNet Datensatz nach [37] sind in Tabelle 6 angegeben.

Tabelle 6: Vergleich von ResNet-50 und ResNet-100

Netzwerk-Architektur	Anzahl Parameter	Speicher-bedarf Parameter	Anzahl FLOPs	Top-1 Klassifikations-fehler	Top-5 Klassifikations-fehler
ResNet-50	$25{,}6 \times 10^6$	102 MB	4×10^9	20,7 %	5,3 %
ResNet-101	$44{,}5 \times 10^6$	178 MB	8×10^9	19,9 %	4,6 %

Im Vergleich zu den bereits eingeführten Architekturen (VGG oder ALexNet) kann hier mit weniger Speicherbedarf (geringerer Anzahl an Parametern) sowie weniger FLOPs eine bessere Genauigkeit auf dem ImageNet-Datensatz erreicht werden.

3.1.4 GoogLeNet/Inception

Mithilfe der VGG-Architekturen konnte ein bemerkenswerter Sprung bei der Genauigkeit von DNNs auf dem ImageNet-Datensatz erzielt werden. Um diese VGG Netzwerke trainieren zu können bedarf es jedoch sehr teure Hardware (GPUs) aufgrund der hohen Anforderungen für die Berechnung. An diesem Punkt setzt der Aufbau des GoogLeNet an, häufig auch als Inception Netzwerk bekannt [38], [39]. GoogLeNet nutzt die Idee, dass die meisten Aktivierungen in einem DNN entweder unnötig (Aktivierung gleich 0) oder aufgrund von Korrelationen redundant sind. Daher ist die effizienteste Architektur eines DNN ein Aufbau mit sparsen Verbindungen zwischen den Aktivierungen. Das bedeutet, dass nicht alle Ausgangskanäle mit allen Eingangskanälen in einer *Convolutional* Schicht verbunden sind, wie es typischerweise der Fall wäre [34]. GoogLeNet setzt dafür sogenannte Inception-Module ein, welche ein sparses CNN mit normalen Konstruktionen approximieren. Der Aufbau eines Inception-Moduls ist in Abbildung 19 dargestellt.

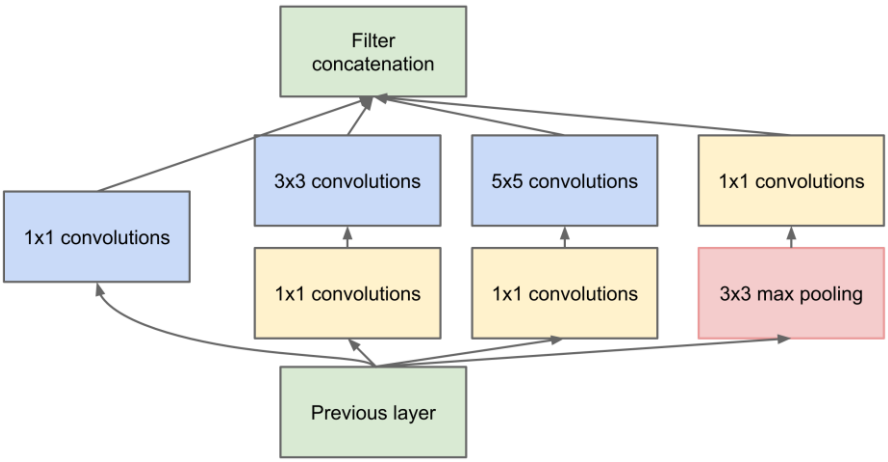

Abbildung 19: Inception-Modul [38]

Mithilfe dieser Architektur kann die Anzahl an benötigten Operationen (Multiplikation, Additionen) deutlich reduziert werden. Die 1x1 *Convolutions* dienen dabei der Dimensionsreduzierung der Kanäle, bevor größere Convolutional-Operationen (3x3 oder 5x5) durchgeführt werden. Als einfaches Beispiel kann dafür eine Berechnung von 192 Eingangskanälen an der 5x5 Convolution gesehen werden. Mit 32 Filtern würden $5 * 5 * 32 * 192 = 153.600$ Multiplikationen durchgeführt. Wird zuvor eine 1x1 Convolution mit 16 Filtern genutzt, sind nur $1 * 1 * 16 * 192 + 5 * 5 * 32 * 16 = 15.872$ Multiplikationen notwendig, um dieselbe Ausgangsdimension zu erhalten. Mithilfe der Inception-Module kann ein sehr tiefes und weites Netz (große Eingangsdimensionen) aufgebaut werden. Zusätzlich zu diesen Änderungen verzichtet GoogLeNet auf Fully-Connected Schichten am Ende des Netzwerkes und tauscht diese durch Pooling-Operationen aus. Dies reduziert die Anzahl an benötigten Parametern drastisch. Im AlexNet sind z.B. ca. 90% der Parameter in den Fully Connected Schichten enthalten [34]. Die dritte Version des Inception-Netzwerks (Inception-v3 [39]) besitzt 42 Schichten und erreicht einen Top-5 Klassifikationsfehler von 5,6% und einen Top-1 Klassifikationsfehler von 21,2%. Dabei werden 4,8 Giga-FLOPs ausgeführt und $24x10^6$

Parameter genutzt. Unter der Annahme von 32-bit Variablen [35] ergibt das einen Speicherbedarf von 96 MB.

3.1.5 MobileNet

Der Trend im Bereich DNN ist es, immer tiefere und komplexere Netzwerke zu gestalten, welche verbesserte Genauigkeiten für spezifische Aufgaben (z.B. ILSVRC) liefern. Als Folge von diesen Entwicklungen steigt häufig der Berechnungsaufwand und/oder der Speicherbedarf zur Umsetzung dieser Netzwerke. In vielen realen Anwendungen sind jedoch genau diese Themen kritisch, z.B. für Augmented Reality, autonome Fahrzeuge, Roboter und vieles mehr. In diesen Anwendungsgebieten sind echtzeitfähige Netzwerke notwendig, die auf kleinen Endgeräten (z.B. Smartphone oder Mikrocontroller) lauffähig sind und keinen GPU-Server zur Verfügung haben. Aus diesen Anforderungen heraus wurde MobileNet [40] entworfen. Das Ziel der Architektur ist ein kleines Netzwerk mit geringer Latenz, das für mobile Anwendungen genutzt werden kann. MobileNet nutzt dabei auch Inception-Module [38], um die Anzahl an Operationen zu reduzieren. Zusätzlich wird die *Depthwise Separable Convolution* eingesetzt. Dies ist eine Form der faktorisierten *Convolution*, bei dem die normale *Convolution* in eine *depthwise Convolution* und eine 1x1 *Convolution* (*pointwise* Convolution) zerlegt wird. Dadurch werden aus einer Schicht zwei Schichten, bei der die erste für die Filterung und die zweite für die Kombination der berechneten Filterausgänge verantwortlich ist [40]. Als Vergleich ist in Tabelle 7 der Unterschied für die Anzahl an FLOPs, der Anzahl der Parameter und der Genauigkeit auf dem ImageNet-Datensatz für ein MobileNet mit klassischen *Convolution*-Operatoren (Conv MobileNet) und einem MobileNet, welches *depthwise Convolutions* nutzt, gegeben.

Tabelle 7: Depthwise Separable vs. Full Convolution MobileNet [40]

Modell	Anzahl Parameter	Anzahl FLOPs	Top-1 Klassifikationsfehler
Conv MobileNet	$29{,}3 \times 10^6$	$4{,}87 \times 10^9$	28,3%
MobileNet	$4{,}2 \times 10^6$	$0{,}57 \times 10^9$	29,4%

Diese Ergebnisse zeigen, dass der Einsatz der d*epthwise Convolution* den Speicher- und Rechenaufwand drastisch reduziert, und der daraus resultierende Genauigkeitsverlust bei lediglich ca. 1 Prozentpunkt liegt, was angesichts der drastischen Reduzierung der Aufwände akzeptiert werden kann.

In [41] wird eine verbesserte Version 2 des MobileNets, MobileNet-V2, vorgestellt. Dabei werden zusätzlich zu den bereits beschriebenen Modulen neue Module eingesetzt, die zum einen den Speicherbedarf weiter reduzieren und zum anderen die Performanz steigern sollen. Dies geschieht durch den Einsatz von *Inverted Residuals* mit *Linear Bottelenecks*. Die *Linear Bottlenecks* sorgen dafür, dass keine großen Tensoren zwischen den Schichten über *Shortcut* Verbindungen übergeben werden,

sondern sorgen mithilfe einer linearen Transformation für ein *Bottleneck*. Durch die lineare Transformation soll sichergestellt werden, dass relevante Informationen nicht durch eine nichtlineare Dimensionsreduktion verloren gehen. Die *Inverted Residual* Blöcke nutzen klassische *Shortcut* Verbindungen und deren Vorteile des verbesserten Gradienten-Flusses [37]. Das invertierende ist die Stelle, an denen diese Verbindungen eingesetzt werden. Typischerweise sind bei *Shortcut*-Verbindungen zwei hochdimensionale Blöcke verbunden, zwischen denen ein *Bottleneck* liegt. Bei den *Inverted Residual* Blöcken sind zwei *Bottlenecks* verbunden, zwischen denen ein höherdimensionaler Block liegt. Dadurch soll der Speicherbedarf weiter reduziert werden, da niederdimensionale Blöcke (die *Bottlenecks*) anstatt hochdimensionaler Tensoren über die *Shortcut*-Verbindung weitergeleitet werden. In Abbildung 20 ist die Architektur des MobileNet-V2 vereinfacht graphisch dargestellt. An dieser Abbildung lässt sich gut der Aufbau der beschriebenen *Inverted Residual* mit *Linear Bottlenecks* erkennen.

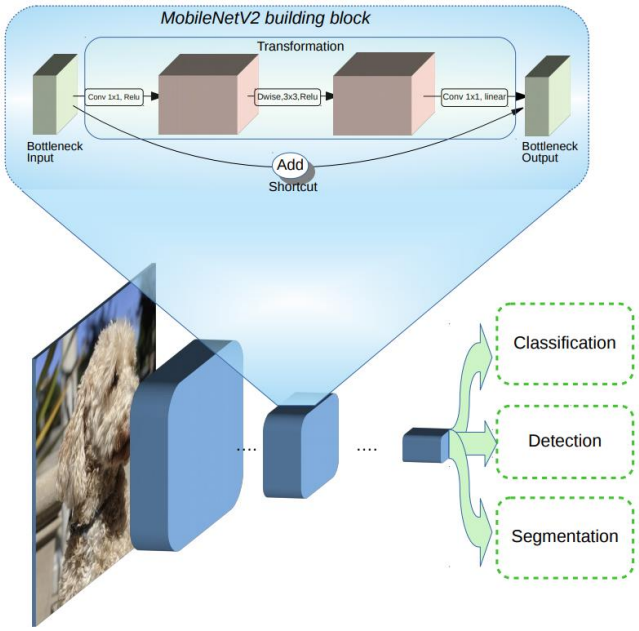

Abbildung 20: Schematische Übersicht über MobileNet-V2 Architektur [42]

MobileNet-V2 erreicht einen Top-1 Klassifikationsfehler von 28% und einen Top-5 Klassifikationsfehler von 9%. Das Modell besteht aus $3{,}5 \times 10^6$ Parametern (entspricht 14 MB Speicher) und führt $0{,}3 \times 10^9$ FLOPs pro Eingangsbild aus [41], [42].

3.2 Modul B

Modul B des L DNN Algorithmus wird mithilfe eines inkrementellen Klassifikator realisiert. Zu den grundlegenden Anforderungen eines inkrementellen Klassifikator, die in Kapitel 2.3 beschrieben wurden, kommen in dieser Arbeit weitere spezifische Anforderungen. Es ist in dieser Arbeit relevant, dass der inkrementelle Klassifikator ohne Speicherung vorheriger Eingangsdaten arbeitet. Dafür gibt es zwei konkrete Gründe. Zum einen sind Anwendungen interessant, bei denen die Algorithmen auf mobilen Endgeräten in Echtzeit laufen. Diese haben nur begrenzten Speicher zur

Verfügung und können nicht eine Vielzahl von Daten sichern. Bei Echtzeit-Anwendungen fallen zudem viele Daten an, die nicht ohne einen großen Daten-Server gespeichert werden können. Ein Transfer dieser Daten auf einen Server würde jedoch einen erheblichen Kommunikationsaufwand darstellen. Zweiter Punkt ist der Einsatz in sicherheitsrelevanten Anwendungen. Es kann sich dabei um sicherheitskritische und/oder persönliche Daten handeln. Beide müssen bei einer Sicherung mit hohem Aufwand geschützt werden. Dies kann auf mobilen Endgeräten nicht immer sichergestellt werden. Bei einer Kommunikation zu einem Server muss wiederum eine sichere Verbindung gewährleistet werden und der Server ausreichend und meistens kostspielig gesichert sein. Da die beiden genannten Punkte hohe Aufwände verursachen, soll der inkrementelle Klassifikator ohne abgespeicherte Trainingsdaten arbeiten, wodurch die genannten Probleme umgangen werden können.

Die beschriebenen Klassifikatoren, Incremental Classifier and Representation Learning (iCaRL) und Adaptive Resonance Theory (ART), erfüllen beide die in [24] generellen Anforderungen an einen inkrementellen Klassifikator:

- Er soll auf Basis eines Daten-Stream, in dem Sample der unterschiedlichen Klassen zu unterschiedlichen Zeitpunkten (zufällig) auftreten, trainierbar sein
- Zu jedem Zeitpunkt muss ein funktionierender Multi-Klassen Klassifikator für die bereits gesehenen und damit bekannten Klassen verfügbar sein
- Die Berechnungsanforderungen und der Speicherbedarf sollen beschränkt sein oder nur langsam ansteigen mit Bezug auf die Anzahl an bekannten Klassen

Jedoch benötigt iCaRL für die Berechnungen der Repräsentationen eine ausgewählte Menge an Exemplaren (Trainingssamples) der jeweiligen Klassen. Diese Anzahl an gespeicherten Daten pro Klasse kann variieren, jedoch wird eine Anzahl von 10 bis 20 Exemplaren pro Klasse empfohlen [43]. Dies widerspricht den oben eingeführten Kriterien, keine spezifischen Daten sichern zu müssen. Zudem müssen für einen Austausch von Wissen zwischen einzelnen, verteilten Netzen diese gespeicherten Exemplare ebenfalls ausgetauscht werden. Aus diesem Grund wird in dieser Arbeit iCaRL nicht für die Umsetzung von Modul B genutzt.

ART-Netzwerke sichern ebenfalls Repräsentationen von Klassen, jedoch keine einzelnen Eingangsdaten von Klassen. Es wird eine Repräsentation pro Klasse auf Feature-Ebene angelegt und inkrementell generalisiert, wenn neue Daten für diese Klasse eintreffen. Durch die Generalisierung der Daten können einzelne Eingangsdaten nicht rückverfolgt werden, weshalb die gestellten Anforderungen für Modul B erfüllt werden können.

Durch den Einsatz des in [1] vorgestellten „*Nothing I Know*“-Konzepts können neue, bisher unbekannte Klassen erkannt werden und ein neuer Knoten für die neue Klasse erstellt werden. Bei diesem Punkt muss erwähnt werden, dass vor Beginn der Anwendung eine Anzahl an maximal möglichen Klassen festgelegt werden muss, da die Anzahl an verfügbaren Kategorie-Knoten (Neuronen) in einem ART-Netzwerk

initial festgelegt wird. Somit können nicht beliebig viele Klassen inkrementell erlernt werden, aber es kann durch eine vorherige Abschätzung eine ausreichende Anzahl vorgegeben werden.

Es gibt eine Vielzahl an unterschiedlichen ART-Netzwerken, welche jeweils für verschiedene Anwendungen genutzt werden. In dieser Arbeit wird ein Fuzzy ARTMAP (FAM)-Netzwerk leicht verändert genutzt. Der Hauptvorteil besteht darin, dass durch den Einsatz der Fuzzy-Theorie anstatt binärer Vektoren (wie in einem klassischen ART1-Netzwerk) kontinuierliche Werte im Bereich zwischen 0 und 1 genutzt werden können. Zudem werden die Fuzzy-Operatoren eingesetzt. In einem FAM-Netzwerk ersetzt dabei der Fuzzy-AND Operator den binäre AND Operator. Der Fuzzy-AND Operator ist dabei wie folgend definiert: $(A \cap B)_i = \min(A_i, B_i)$ [26].

Zusätzlich arbeitet ein typisches FAM-Netzwerk mit komplementärer Codierung (*complement coding*), wodurch ein ursprüngliches Eingangssignal X umgewandelt wird zu $X_{cc} = X|\overline{X}$, mit dem komplementären Eingangssignal $\overline{X}$, das an das originale Eingangssignal angehängt wird [26].

Nach [1] sind diese eben genannten Besonderheiten der Fuzzy-Operatoren und des *complement Coding* auch Probleme bei der Nutzung eines solchen Netzwerkes innerhalb des L DNN Algorithmus. Da DNNs als Feature Extrahierer zu sparsen Feature Repräsentationen tendieren, führt der Einsatz von komplementärer Codierung zu sehr hohen Aktivierungen im komplementären Teil. Kleine Unterschiede zwischen einzelnen Mustern lassen sich dann schwierig unterscheiden. Deswegen wird in [1] vorgeschlagen, den *complement Coding* Teil des FAM-Netzwerks nicht zu nutzen. Aus den genannten Gründen und da ein DNN in dieser Arbeit Modul A realisiert, wird dieser Teil des Netzwerkes nicht genutzt. Als zusätzliche Maßnahme wird der Austausch der Fuzzy AND Logik durch das Skalarprodukt von zwei Vektoren vorgeschlagen. Dadurch kann erzielt werden, dass das Ergebnis der Vergleichsoperation normalisiert zwischen 0 und 1 ist. Mit dieser Konzeption des Modul B können die gestellten Anforderungen in der Theorie bewältigt werden. Eine graphische Darstellung und eine kurze Zusammenfassung werden in Kapitel 3.3.2 gegeben.

3.3 Zusammenfassung

In diesem Kapitel werden eine Zusammenfassung und finale Bewertung der unterschiedlichen Ansätze für die jeweiligen Module gegeben. Es wird auf Basis der eingesetzten Metriken und genannten Kriterien jeweils ein Ansatz pro Modul ausgewählt. Abschließend wird dann der gesamte L DNN Algorithmus mit den ausgewählten Modulen dargestellt.

3.3.1 Modul A

Modul A hat wie in Kapitel 3.1 beschrieben unterschiedliche Kriterien, die ausschlaggebend für die Auswahl sind. Es gibt eine Vielzahl an möglichen Netzwerken, welche die Aufgaben von Modul A übernehmen können. Die konkrete

Auswahl ist stark vom gewünschten Anwendungsfall und der gegebenen Hardwareumgebung abhängig. In Tabelle 8 sind alle vorgestellten DNN-Architekturen nochmals direkt miteinander verglichen.

Tabelle 8: Übersicht aller vorgestellten DNNs für Modul A

Netzwerk-Architektur	Anzahl Parameter	Speicherbedarf Parameter	Anzahl FLOPs	Top-1 Klassifikations-fehler	Top-5 Klassifikations-fehler
AlexNet	$60x10^6$	240 MB	$0,7x10^9$	36,7 %	15,3 %
VGG-16	$138x10^6$	552 MB	$16x10^9$	25,6 %	8,1 %
VGG-19	$144x10^6$	576 MB	$20x10^9$	25,5 %	8,0 %
ResNet-50	$25,6x10^6$	102 MB	$4x10^9$	20,7 %	5,3 %
ResNet-101	$44,5x10^6$	178 MB	$8x10^9$	19,9 %	4,6 %
Inception-V3	$24x10^6$	96 MB	$4,8x10^9$	21,6 %	5,6 %
MobileNet-V2	$3,5x10^6$	14 MB	$0,3x10^9$	28 %	9 %

Im Rahmen dieser Arbeit gibt es keine definierten Hardware-Begrenzungen. Dennoch ist gefordert, dass der implementierte L DNN Algorithmus auf mobilen Endgeräten (z.B. Smartphone oder Mikrocontroller) lauffähig sein soll. Zunächst wird deshalb der errechnete Speicherbedarf der Netzwerke verglichen. Hier gibt es eine große Schwankung von 576 MB für VGG-19 bis 14 MB für MobileNet-V2. Trotzdem kann gesagt werden, dass auch der Speicherbedarf eines VGG-19 Netzwerkes akzeptabel wäre, da mobile Endgeräte heutzutage in der Regel die Möglichkeit besitzen ein zusätzliches Speichermodul (z.B. SD-Karte) einzusetzen, wodurch zusätzlicher Speicher verfügbar ist. Es muss jedoch in Relation gesetzt werden, dass z.B. Smartphones APPs nur selten größer als 300 MB sind, weshalb VGG-16 und VGG-19 in einigen mobilen Anwendungen nicht die optimale Wahl wäre.

Weiter wird die benötigte Anzahl an Operationen für die Bearbeitung eines einzelnen Eingangsbildes verglichen. Auch hier gibt es große Unterschiede, von 20 Giga-FLOPs in einem VGG-19 bis zu 0,3 Giga-FLOPs in MobileNet-V2. Dieser Punkt ist für mobile Anwendungen kritischer zu sehen als der Speicherbedarf. Der Grund liegt darin, dass bei vielen mobilen Anwendungen Echtzeitfähigkeit gefordert ist. Um dies zu gewährleisten müssen die benötigten Operationen für die Bearbeitung von Eingangsdaten von dem vorhandenen Prozessor in einem gewissen Zeitraum (z.B. innerhalb einer Sekunde) abgearbeitet werden. Abhängig vom Anwendungsfall können dabei unterschiedliche Raten erforderlich sein. Im Fall der Bildverarbeitung wird in diesem Kontext von Frames per Second (FPS) gesprochen. Für einen konkreten Vergleich wird als Hardware ein in der Praxis häufig genutzter Mikrocontroller, der Raspberry Pi 3 Modul B [44], angenommen. In dem Kontext von FLOPs ist die Taktfrequenz der Hardware relevant, welche für diesen Mikrocontroller mit 1,2 GHz angegeben ist [44]. Unter der Annahme, dass nur die in der Tabelle aufgeführten FLOPs zur Bearbeitung des Eingangsbildes ausgeführt werden müssen,

kann eine theoretische FPS-Rate berechnet werden. Mit einem VGG-19 würde ein Raspberry Pi 3 Modul B 0,06 FPS erreichen können. Die Bearbeitung eines Bildes würde somit 16,6 Sekunden brauchen. Mit dem ResNet-50 könnte eine FPS-Rate von 0,3 erreicht werden, und mit MobileNet-V2 nach diesen Berechnungen 4 FPS. Mit einem ResNet-50 würde die Bearbeitung eines Bildes ca. 3,3 Sekunden brauchen. Wenn dies noch als akzeptabel angenommen wird für die Anwendung, wäre auf Basis von diesem Kriterium lediglich AlexNet, ResNet-50 und MobileNet-V2 für die spätere Nutzung relevant.

Als letzten Punkt wird die Genauigkeit der Netzwerke verglichen. Die genannten Netzwerke werden im späteren Verlauf nicht als Klassifikatoren eingesetzt, aber die Klassifikationsgenauigkeit gibt eine gute Übersicht über die Fähigkeit, relevante Features zu extrahieren. Auch hier muss abhängig von der realen Anwendung entschieden werden. Für sicherheitskritische Anwendungen spielt die Genauigkeit eine große Rolle, während für sicherheitsunkritische Anwendungen die Genauigkeit eventuell geringer sein kann und dafür mehr Wert auf eine geringe Laufzeit gelegt wird. Im Rahmen dieser Arbeit wird das generelle Potenzial des L DNN Algorithmus untersucht und nicht die exakte Performanz von CNN-Architekturen. Deshalb kann auch eine geringere Genauigkeit auf dem ImageNet-Datensatz akzeptiert werden, wenn dafür die anderen bereits genannten Kriterien die Erwartungen erfüllen.

Unter Berücksichtigung aller genannten Kriterien wird im weiteren Verlauf der Arbeit MobileNet-V2 als Modul A des L DNN Algorithmus eingesetzt. Ein großer Punkt ist die spezielle Architektur für mobile Anwendungen, die eine Echtzeit(-nahe) Bearbeitung von Eingangsdaten auch auf mobilen Endgeräten erlaubt. Dies wird sowohl durch den geringen Speicherbedarf mit lediglich 14 MB, als auch durch die geringe Anzahl von 0,3 Giga-FLOPs gewährleistet. Dadurch kann dieses Netzwerk auch auf leistungsschwächeren Endgeräten laufen. Zudem kann trotz dieser Merkmale ein Top-5 Klassifikationsfehler von 9 % auf dem ImageNet-Datensatz erreicht werden, was im Rahmen dieser Arbeit eine ausreichende Genauigkeit darstellt, die auch von vielen komplexeren Netzwerken nicht erreicht (z.B. AlexNet) oder nur knapp unterboten wird (z.B. VGG-16 und VGG-19). Die Architektur des MobileNet-V2 Netzwerks ist in Tabelle 9 aufgeführt.

Tabelle 9: Architektur MobileNet-V2 [41]

Input	Operator	t	c	n	s
224x224x3	Conv2d	-	32	1	2
112x112x32	Bottleneck	1	16	1	1
112x112x16	Bottleneck	6	24	2	2
56x56x24	Bottleneck	6	32	3	2
28x28x32	Bottleneck	6	64	4	2
14x14x64	Bottleneck	6	96	3	1
14x14x96	Bottleneck	6	160	3	2
7x7x160	Bottleneck	6	320	1	1
7x7x320	Conv2d 1x1	-	1280	1	1
7x7x1280	Avgpool 7x7	-	-	1	-

Dabei stellt jede Zeile eine Sequenz von einer oder mehreren identischen Schichten dar, die *n*-Mal wiederholt wird. Alle Schichten innerhalb einer Sequenz (Zeile) haben dieselbe Anzahl *c* an Ausgangskanälen. Die erste Schicht einer Sequenz hat dabei die *Stride s* und alle weiteren Schichten dieser Sequenz nutzen *Stride* 1. Alle *spatial Convolutions* nutzen 3x3 Kernel. Der Expansionsfaktor *t* wird auf die Eingangsgröße angewendet, um das Prinzip der linearen Bottlenecks zu nutzen. Das Netzwerk besteht aus einer initialen *Convolutional* Schicht und darauffolgend 17 Bottleneck Schichten. Zum Schluss folgen eine 1x1 *Convolution* Schicht und eine Pooling-Schicht. Wenn das Netzwerk zur Klassifikation eingesetzt würde, würde noch eine zusätzliche Schicht mit Softmax-Aktivierung folgen. Da hier jedoch nur die Features genutzt werden, wird diese Schicht nicht genutzt. Damit ist der Ausgang von Modul A (und damit der Eingang von Modul B) ein Tensor mit den Dimensionen 1x1x1280, der wiederum direkt in einen Vektor der Größe 1280x1 übertragen werden kann.

3.3.2 Modul B

Modul B besteht wie in Kapitel 3.2 beschrieben aus einem Fuzzy ARTMAP (FAM) Netzwerk. Dieses Netzwerk kann initial mit bereits bekannten Klassen vortrainiert sein oder initial „leer“ sein (keine Klassen bekannt). Ausgehend davon werden neue Eingangsdaten klassifiziert. Falls die Eingangsdaten zu keiner bekannten Klasse passen, fallen diese Eingangsdaten in die Kategorie „Nothing I know“ und eine neue Repräsentation (neuer Knoten) wird im Netzwerk angelegt. Für diesen Schritt ist das Feedback des Benutzers oder ein Label des Trainingssamples notwendig, um das korrekte Klassenlabel zu erhalten. Die Architektur des genutzten FAM-Netzwerks ist in Abbildung 21 dargestellt. Die einzelnen Bestandteile und deren Nutzen werden im Folgenden erläutert.

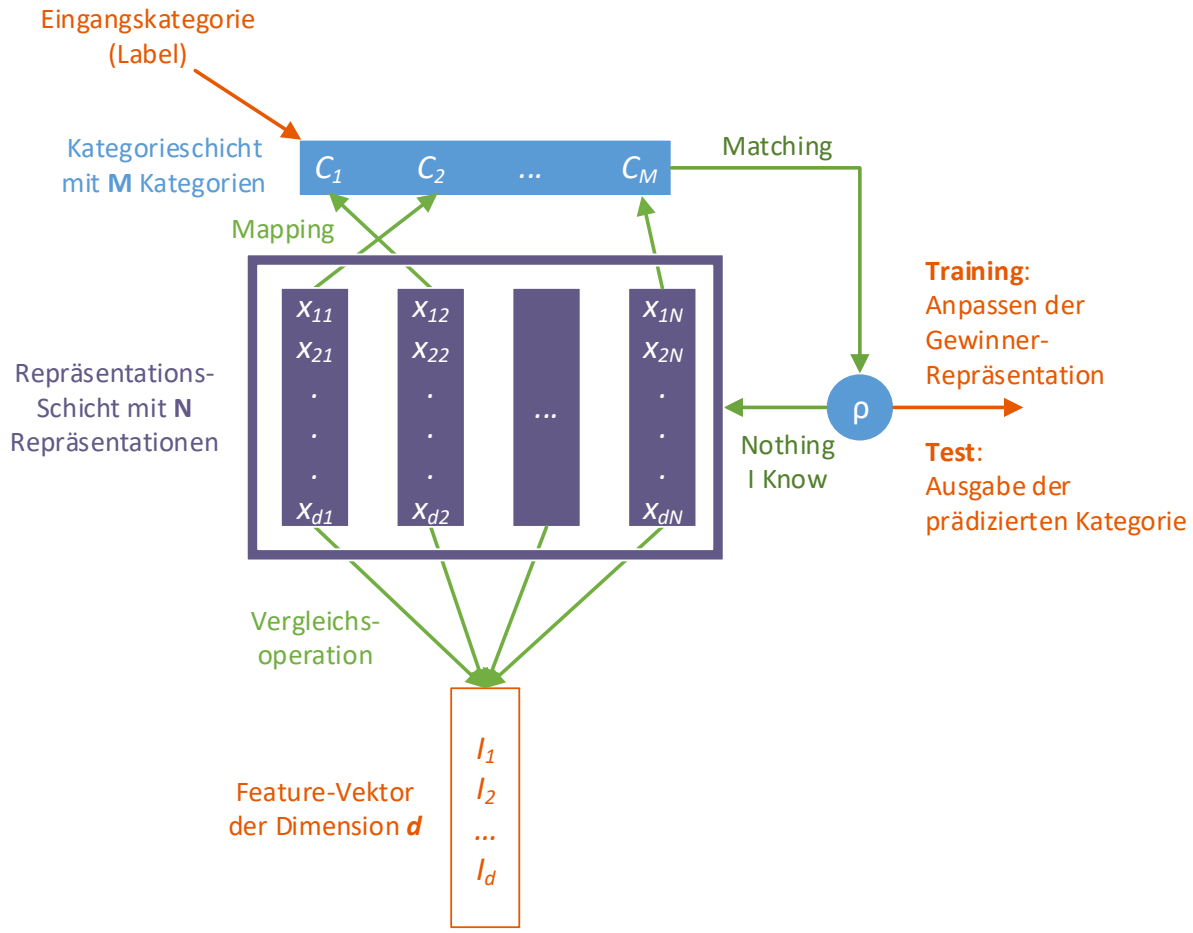

Abbildung 21: Architektur des FAM-Netzwerks

Das Eingangssignal I der Dimension (Größe) d kommt an der Eingangsschicht an. Daraufhin werden die bereits erlernten Repräsentationen x der N Kategorie-Knoten mit diesem Eingangssignal mithilfe einer Vergleichsoperation assoziiert. Die einzelnen Werte x_{ji} innerhalb einer Repräsentation $x_{:i}$ werden Top-Down Gewichte genannt, da sie von oben nach unten (Top-Down) mit den Eingangsdaten assoziiert werden. In dieser Arbeit wird das Skalarprodukt der Gewichte mit den Eingangsdaten I für die Assoziation genutzt. Auf Basis dieser berechneten Assoziationen erfolgen dann die Aktivierungen der M Ausgangsneuronen C in der Kategorie Schicht. Abhängig von den erlernten Gewichten können mehrere Repräsentationen auf dieselben Ausgangsneuronen C zeigen, wenn diese dieselbe Klasse repräsentieren. In der Kategorie Schicht wird während des Trainings (der Adaptionsphase) die korrekte Klasse hinzugefügt, um das richtige Label für ein korrektes Adaptieren der Neuronen im Netzwerk zu haben. Das Adaptieren einer Repräsentation mithilfe der Lernrate α ist in Formel **(16)(17)** angegeben mit dem Eingangssample $\underline{I}$ und der vorhandenen Repräsentation $\underline{x}$.

$$\underline{x}_{adaptiert} = (1 - \alpha)\underline{x} + \alpha\underline{I} \tag{16}$$

Die berechneten Aktivierungen (*Matching*-Werte) werden daraufhin mit dem *Vigilance* Parameter ρ verglichen. Wenn der Schwellwert überschritten ist, wird in der Inference die Gewinner-Kategorie/-Klasse ausgegeben und somit die Eingangsdaten dieser

Klasse zugeordnet. Während des Trainings wird die Gewinner-Repräsentation mithilfe der Eingangsdaten I weiter generalisiert (siehe Formel **(16)**).

Wenn das Matching der Gewinner-Kategorie den Schwellwert ρ unterschreitet, findet das *Nothing I know*-Konzept Anwendung. Da das höchste Matching nicht ausreichend ist, erkennt das Netzwerk, dass es dieses Eingangsmuster nicht kennt. In diesem Fall wird dann eine neue Repräsentation für das gegebene Eingangsmuster angelegt. Der Benutzer muss in diesem Fall die korrekte Kategorie angeben, damit für spätere Eingangsdaten die passende Kategorie-Bezeichnung ausgegeben werden kann. Während des Trainings wird das passende Label der Trainingsdaten statt der Benutzereingabe genutzt.

3.3.3 Lifelong DNN Algorithmus

Aus den beiden beschriebenen Modulen A und B lässt sich nun der gesamte L DNN Algorithmus bauen. Er besteht aus einem vortrainierten Feature-Extrahierer, welcher durch das MobileNet-V2 realisiert wird. Die letzte Schicht wird entfernt und die extrahierten Features an Modul B weitergeleitet. Dieses Modul wird durch ein angepasstes FAM-Netzwerk realisiert. Als Ausgabe des Modul B erhält der Benutzer das prädizierte Klassenlabel des Eingangsbildes. Die gesamte Architektur sowie Beispieldimensionen für ein 224x224x3 Eingangsbild sind in Abbildung 22 skizziert.

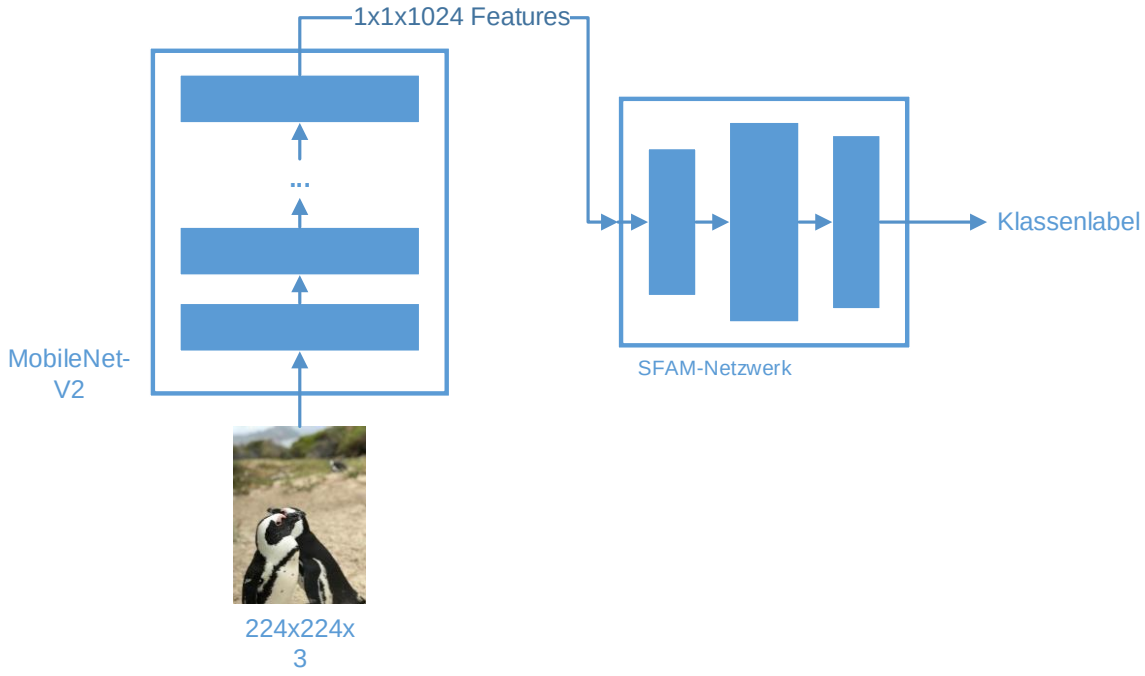

Abbildung 22: Gesamtarchitektur L DNN Algorithmus

Im weiteren Verlauf der Arbeit werden nun Testfälle definiert, um die Performanz und das Potenzial des L DNN Algorithmus zu untersuchen.

4 Aufbau der Evaluierung

Um eine belastbare Aussage über das Potenzial des L DNN Algorithmus treffen zu können müssen Tests sowie Metriken definiert werden, welche zur Evaluierung genutzt werden. Dafür werden in diesem Kapitel die genutzten Datensätze sowie die eingesetzten Kriterien (Metriken) definiert, die auch für vergleichbare Algorithmen genutzt wurden.

4.1 Datensätze

Zur Evaluierung werden öffentlich verfügbare Bilddatensätze genutzt, welche sehr häufig im Bereich der DNNs zur Bestimmung des Potenzials eines Algorithmus genutzt werden. Zur Auswertung der Performanz des Algorithmus wird die Klassifikationsgenauigkeit bewertet. In dieser Arbeit wird der MNIST- und ImageNet-Datensatz zur Evaluierung des inkrementellen und verteilten Klassen-Lernens genutzt. Dabei werden dem L DNN Algorithmus neue, bisher unbekannte Klassen gezeigt, welche das Netzwerk auf Basis weniger Beispielbilder erlernen muss. Die Aufgabe des Netzwerks ist es, inkrementell diese neuen Klassen zu erlernen ohne dabei alte, bereits bekannte Klassen zu vergessen. Im Folgenden wird der Aufbau der Datensätze sowie deren Einsatz im Rahmen dieser Arbeit thematisiert.

Split-MNIST

Der MNIST-Datensatz (Modified National Institute of Standards and Technology) [11] kann als Standard-Datensatz im Bereich des maschinellen Lernens angesehen werden. In diesem Datensatz sind insgesamt 70.000 handgeschriebene Ziffern von 0 bis 9 in *Grayscale*-Bildern im Format 28x28 gesammelt, aufgeteilt in 60.000 Trainings- und 10.000 Testbilder. Die Aufgabe für das Netzwerk besteht darin, die Ziffern korrekt zu klassifizieren. Für die konkrete Aufgabenstellung des inkrementellen Klassenlernens wird der sogenannte Split-MNIST Datensatz genutzt. Dafür werden jeweils 2 benachbarte Ziffern (z.B. 0/1 und 2/3) zu einer Gruppe zusammengefasst. Diese Gruppen werden dem Netzwerk gezeigt, womit dieser Datensatz fünf inkrementelle Schritte hat. Nach einer gewissen Anzahl an Bildern pro Klasse (Wiederholungen) wird die Klassifikationsgenauigkeit auf den bereits gezeigten Klassen bestimmt. Daraufhin wird dem Netzwerk die nächste Gruppe gezeigt. Nachdem alle Gruppen vom Netzwerk gesehen wurden, werden Testbilder aller Klassen gezeigt und die Klassifikationsgenauigkeit bestimmt. Dieser Testfall des Split-MNIST Datensatz kann als Grundlagenuntersuchung verstanden werden. Der Datensatz stellt kein allzu komplexes Problem dar, kann jedoch in einem ersten Schritt genutzt werden, um die prinzipielle Funktion eines Algorithmus zu untersuchen und mit anderen Ansätzen zu vergleichen, da es für diesen Datensatz eine Vielzahl an Untersuchungen gibt und somit die Performanz gut mit anderen Algorithmen verglichen werden kann.

Aufgrund des vortrainierten MobileNet-v2 konnten nur spezielle Eingangsdimensionen für die Bilder genutzt werden. Die kleinste verfügbare Dimension ist dabei 96x96. Deshalb wurden kleinere Bilder (wie MNIST) auf diese Dimension vergrößert. Zusätzlich müssen die Bilder in das RGB-Format umgewandelt werden, da 3 Kanäle von den Eingangsdaten erwartet werden in der vorhandenen Architektur. In Abbildung 23 ist ein Bild des MNIST-Datensatzes vor der Bild-Augmentation im originalen Format 28x28x1 zu sehen. Abbildung 23 zeigt die Zahl „7“.

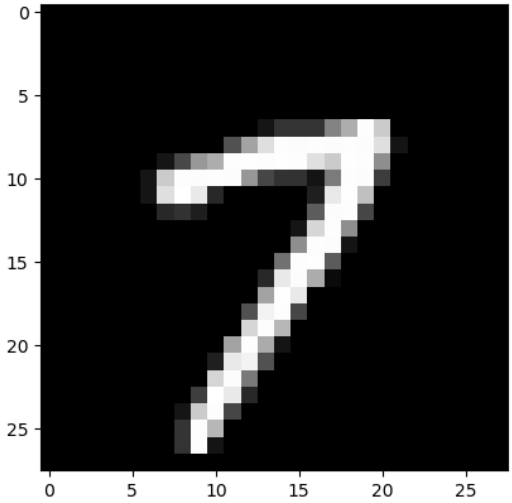

Abbildung 23: MNIST-Bild vor Bild-Augmentation

Abbildung 24 zeigt ein Bild des MNIST-Datensatzes nach der zuvor beschriebenen Bild-Augmentation. Es besitzt nun das Format 96x96x3. Jedoch ist zu sehen, dass das Bild in seiner Grundstruktur nicht wesentlich verändert wurde, wodurch durch diese Bild-Augmentation kein großer Einfluss auf Klassifikationsgenauigkeit erwartet wird. In diesem Beispielbild ist die Zahl „8“ zu sehen.

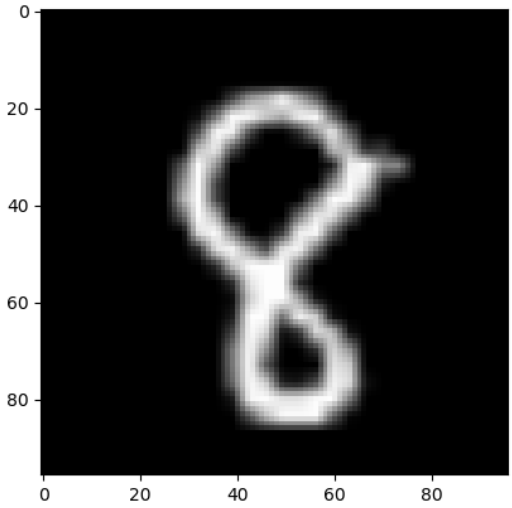

Abbildung 24: MNIST-Bild nach Augmentation

ImageNet

Zusätzliches wird der ImageNet-Datensatz für die Evaluierung genutzt. Dieser Datensatz stellt ein komplexes Problem mit 1.000 unterschiedlichen Klassen dar. Die Bilder sind RGB-Bilder im Format 224x224x3. Es wird konkret der Datensatz der *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) aus dem Jahr 2012 genutzt [45]. Als Test-Daten werden wie in der Literatur und für vergleichbare Modelle üblich die frei verfügbaren 50 Validationsbilder pro Klasse genutzt. In dem genutzten Datensatz gibt es für die 1000 Klassen insgesamt 1.281.167 Trainingsbilder. Die Anzahl an Trainingsbilder pro Klasse schwankt dabei zwischen 732 und 1300. Mit 50 Validationsbildern (hier später als Test-Bilder genutzt und bezeichnet) pro Klasse sind 50.000 Validationsbilder vorhanden. Weitere Informationen über die unterschiedlichen Klassen können in [45] nachgelesen werden. Allein die Größe und Farbe des Bildes erschwert die Aufgabe im Vergleich zum MNIST-Datensatz wesentlich. Weiterhin gibt es eine Vielzahl an unterschiedlichen Klassen und eine hohe Varianz zwischen diesen, von mehreren unterschiedlichen Hunderassen über Containerschiffe zu einem Polizeiauto. Der gesamte ImageNet-Datensatz wird für eine finale Bewertung des Algorithmus eingesetzt. Ein beispielhaftes Bild der Klasse „Strauß“ aus dem ImageNet-Datensatz ist in Abbildung 25 zu sehen.

Abbildung 25: Beispielbild der Klasse „Strauß“ aus dem ImageNet-Datensatz

ImageNet-10

Aus dem gesamten ImageNet-Datensatz wird nochmals ein kleinerer Datensatz mit 10 zufälligen Klassen generiert, im Folgenden ImageNet-10 genannt. Die Klassen sind im Folgenden, mit dem Index der jeweiligen Klasse in Klammern, aufgelistet: Königspinguin (145), Malteser (Hunderasse – 153), Schneeleopard (289), Passagierflugzeug (404), Zeppelin (405), Containerschiff (510), Fußball (805), Sportauto (817), Sattelzug (867) und Orange (950). Dieser verkleinerte Datensatz wird genutzt um das für MNIST gezeigte Verhalten mit wenigen Klassen auf einer komplexeren Aufgabe nachzustellen und zu überprüfen. Für ImageNet-10 wurden Bilder der Dimension 96x96x3 genutzt. Bei diesem Datensatz werden die Klassen einzeln trainiert und nicht in Gruppen, wodurch 10 inkrementelle Schritte durchgeführt werden.

4.2 Evaluierungskriterien

Es wird die Klassifikationsgenauigkeit als Kriterium für die Evaluierung genutzt, welche auch als *True Positive* (*TP*) -Rate bezeichnet wird. Diese Genauigkeit gibt an, wie viele Prozent der Testbilder korrekt klassifiziert werden und somit der korrekten Klasse zugeordnet werden können. Für die Bestimmung dieser Genauigkeit werden die in den Datensätzen enthaltenen, bisher für das Netzwerk unbekannten, Test-Bilder o genutzt. Es wird die vom Klassifikator-Netzwerk geschätzte Klasse $K(o)$ mit der korrekten Klasse des Bildes $C(o)$ verglichen und die Anzahl an korrekten Klassifizierungen gezählt. Diese Anzahl in Relation zu der Menge aller Testbilder $|TE|$ ergibt die Klassifikationsgenauigkeit (TP-Rate) in Gleichung **(17)**:

$$TP = \frac{|\{o \in TE | K(o) = C(o)\}|}{|TE|} \qquad (17)$$

Da der Speicherbedarf in dieser Arbeit berücksichtigt werden soll, wird zusätzlich bei einigen Testfällen der Speicherbedarf des Algorithmus angegeben und als Evaluierungskriterium eingesetzt. Der Speicherbedarf ist dabei der belegte Speicher des angegebenen Modells auf dem Endgerät.

5 Evaluierung und Ergebnisse

In diesem Kapitel werden die Testfälle sowie deren Ergebnisse explizit vorgestellt und diskutiert. Auf Basis dieser Testfälle wird schließlich der L DNN Algorithmus sowie dessen Potenzial bewertet. Die in diesem Kapitel vorgestellten Ergebnisse werden mit lediglich einer Epoche pro inkrementellen Schritt erzielt. Das heißt jedes Bild wird während der Trainingsphase lediglich einmal dem Netzwerk gezeigt und nicht wie bei klassischen Deep Learning Anwendungen mehrmals über mehrere Epochen.

5.1 Hyperparameter-Optimierung Modul B

Neuronale Netzwerke besitzen eine Vielzahl an Hyperparametern, welche abhängig vom konkreten Anwendungsfall und den vorliegenden Daten für eine optimale Performanz des Netzwerkes unterschiedlich eingestellt werden können. Um eine möglichst optimale Parametrierung des inkrementellen Klassifikator in Modul B zu erhalten, wurden die relevanten Parameter identifiziert, um mithilfe einer Gitter-Suche (*Grid Search*) die optimalen Werte für die weiteren Evaluierungsfälle zu finden. Als relevante Parameter des FuzzyARTMAP-Netzwerkes wurden die Lernrate α und der Vigilance-Parameter ρ identifiziert, da diese das Training des Netzwerkes und damit am Ende die Performanz im Testfall beeinflussen. Details zu den beiden Modell-Parametern sind in Kapitel 3.3.2 zu finden.

Der Fall des kontinuierlichen Lernens auf einem Endgerät für Split-MNIST und ImageNet-10 wird genutzt, um die oben beschriebenen Hyperparameter zu optimieren und deren Einfluss zu untersuchen. Diese beiden Datensätze erlauben es aufgrund ihres begrenzten Umfangs (lediglich 10 Klassen) viele Tests in kurzer Zeit durchzuführen, was für die Gitter-Suche relevant ist, da dort viele Tests für eine Abdeckung des Gitters erforderlich sind.

Für die Hyperparameter-Optimierung wird eine zwei-dimensionale Gitter-Suche mit den Parametern α und ρ durchgeführt. Dafür werden diese beiden Parameter jeweils im Bereich [0,1] in 0,1-er Schritten erhöht. Es werden 5 Wiederholungen pro mögliche Kombination durchgeführt, um eine statistische Aussagekraft zu erhalten. Das Vorgehen resultiert in jeweils 500 Tests für Split-MNIST und ImageNet-10. Die weiteren Parameter werden auf fixe Werte eingestellt, welche im Folgenden aufgelistet sind:

- modul_b_epsilon = 0,001
- modul_b_s = 1,05
- train_img_per_class = 20
- test_img_per_class = 100

Aufgrund der vielen Testfällen werden lediglich 20 Trainings- und 100 Testbilder pro Klassen eingesetzt, um die Dauer dieses Evaluierungsfalls einzuschränken. Auf der

Basis der erzielten Ergebnisse werden die konkreten Werte für α und ρ ausgewählt. Die Gruppen für Split-MNIST werden in einer festen Reihenfolge gezogen, da dies auch in der Literatur bei vergleichbaren Algorithmen so umgesetzt wurde. Bei ImageNet-10 wird die Reihenfolge der Klassen zufällig bestimmt, um eine zusätzliche Varianz der Trainingsdaten mit einzubeziehen. Für beide Datensätze werden in jeder Wiederholung zufällig die benötigte Anzahl (20) an Trainingsbilder aus dem großen Trainingsdatensatz gezogen. Die ermittelte Genauigkeit wird auf separaten Validationsdaten aus dem Trainings-Datensatz bestimmt (Hold-Out Daten). Diese Bilder werden nicht für das Training verwendet. Die Test-Datensätze werden dann im späteren Verlauf der Arbeit zur finalen Performanz-Bewertung eingesetzt, um ein Overfitting auf die Testdaten zu vermeiden.

Es wird erwartet, dass für sehr kleine und große Werte von α (Werte nahe 0 und 1) schlechte Ergebnisse erzielt werden, da bei diesen Fällen kaum ein Lernen stattfindet. Bei $\alpha = 0$ werden neue Trainingsdaten „nicht genutzt“ und bei $\alpha = 1$ wird lediglich das neue Trainingssample „genutzt“ und die alte Repräsentation damit „überschrieben“. Hier muss ein Wert gefunden werden, der es erlaubt altes, vorhandenes Wissen mit neuen Samples zu verbinden, um eine möglichst ideale und generalisierte Repräsentation dieser Klasse zu bilden.

Bei ρ wird erwartet, dass ab einem gewissen Wert keine Änderung mehr zu erkennen ist, da die Ähnlichkeit zwischen zwei Samples einer Klasse selten nahe 1 liegt. Dieser Schwellwert für ρ wird für die beiden Datensätze unterschiedlich erwartet. Ab diesem Schwellwert kann gesagt werden, dass jedes Sample, das im Training gesehen wird, als neue Repräsentation angelegt wird. Dies erfordert einen hohen Speicherbedarf. Zudem sind die erhaltenen Repräsentationen nicht generalisiert, wodurch auf späteren, abweichenden Test-Daten keine guten Generalisierungsfähigkeiten erwartet werden. Deshalb muss ein Wert für ρ unterhalb dieser Schwelle gefunden werden. Ein zu kleiner Wert für ρ führt dazu, dass die einzelnen Klassen durch sehr wenige (im Extremfall durch eine) Repräsentationen dargestellt werden. Je nach Klasse kann das positiv sein (wenn alle Samples der Klasse sehr ähnlich aussehen). Im Allgemeinen ist das aber nicht wünschenswert, da dadurch abweichende Samples der Klasse (zum Beispiel eine verdrehte Zahl) nicht gut erkannt werden können. Deshalb muss für ρ ein guter Mittelwert zwischen „zu vielen und zu wenigen“ Repräsentationen gewählt werden.

In Abbildung 26 sind die Ergebnisse der beschriebenen Gitter-Sucher zur Hyperparameter-Optimierung von Modul B auf Basis des Split-MNIST Datensatzes dargestellt. Entlang der x-Achse sind die unterschiedlichen Werte von ρ zu sehen und entlang der y-Achse sind die Werte für α aufgetragen. In den jeweiligen Feldern ist der Mittelwert der Klassifikationsgenauigkeit aus den fünf Wiederholungen für diese Parameterkombination eingetragen.

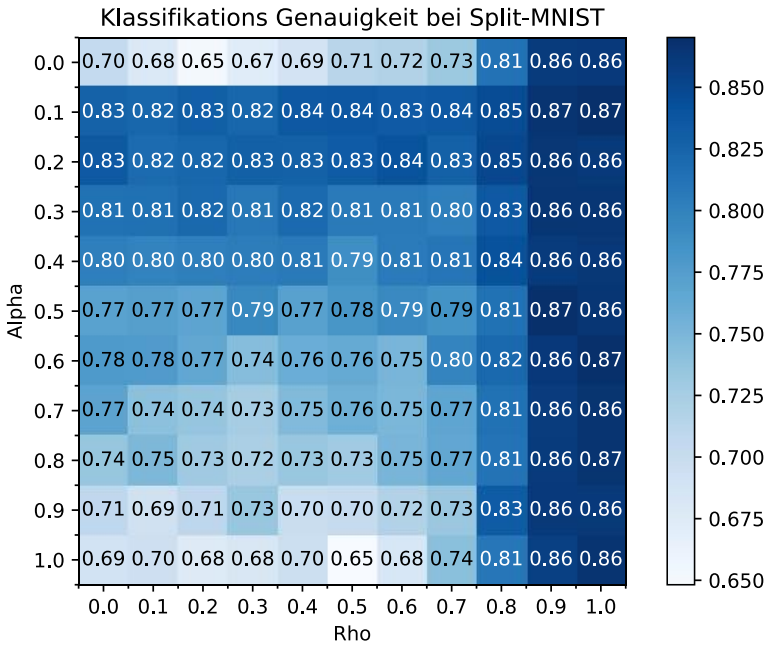

Abbildung 26: Ergebnisse der Gitter-Suche für α und ρ auf Basis des Split-MNIST Datensatzes

In Abbildung 27 sind die Ergebnisse für dieselben Testfälle auf Basis des ImageNet-10 Datensatzes dargestellt. Die Darstellungsart ist dabei identisch wie bereits für Abbildung 26 beschrieben.

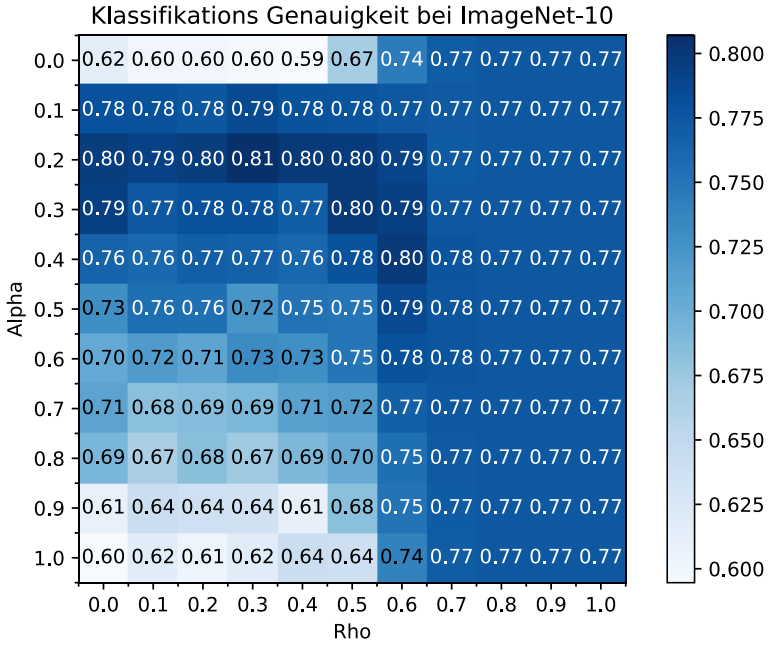

Abbildung 27: Ergebnisse der Gitter-Sucher für α und ρ auf Basis des ImageNet-10 Datensatzes

Die Ergebnisse werden für beide Parameter getrennt bewertet werden, da es keine relevante Korrelation der beiden Parameter gibt.

Zunächst werden die Ergebnisse für den Parameter α bewertet. Hier ist das erwartete Verhalten zu erkennen. Bei beiden Datensätzen sind für hohe Werte von α (>0,8) schlechte Ergebnisse zu sehen, da in diesem Fall lediglich die neuen Daten verwendet werden, um die Repräsentation zu bilden. Bei $\alpha = 0$ sind ebenfalls schlechte Ergebnisse zu beobachten, weil nur das erste gesehene Sample genutzt wird, um die Repräsentation zu bilden. Anhand der in Abbildung 26 und Abbildung 27 dargestellten Ergebnisse lassen sich für $\alpha = 0{,}1$ und $\alpha = 0{,}2$ die besten Ergebnisse erzielen. Für Split-MNIST variieren die besten Ergebnisse zwischen diesen beiden Werten von α. Bei ImageNet-10 können jedoch die besten Ergebnisse mit $\alpha = 0{,}2$ erzielt werden, weshalb dieser Wert im weiteren Verlauf der Arbeit genutzt wird.

Der Parameter ρ ist maßgeblich dafür verantwortlich, wie viele Repräsentationen angelegt werden. Damit beeinflusst ρ direkt den Speicherbedarf des FuzzyARTMAP-Netzwerks. Deshalb muss bei diesem Parameter auf ein Trade-Off zwischen Performanz und Speicherbedarf geachtet werden. Das erwartete Verhalten, bei dem ab einem gewissen Schwellwert kaum Veränderung in der Klassifikationsgenauigkeit beobachtet werden kann, kann durch die Versuche bestätigt werden. Bei Split-MNIST ist dieser Schwellwert bei ca. 0,9 zu sehen, während bei ImageNet-10 dieser bereits bei 0,7 zu beobachten ist. Dieser Unterschied kann mit der Komplexität der Bilder erklärt werden. Aufgrund der komplexen Bilder von ImageNet-10 sind die Ähnlichkeiten der einzelnen Bilder zueinander geringer, wodurch bereits bei niedrigeren Werten von ρ nahezu jedes Trainingssample als Repräsentation angelegt wird. Bei Split-MNIST sind die besten Ergebnisse mit den Werten 0,9 und 1 zu beobachten. Das sind die Fälle, bei denen jedes Trainingssample als Repräsentation abgelegt wird. Bei ImageNet-10 sind die besten Ergebnisse zwischen 0,3 und 0,5 zu sehen.

Für Split-MNIST zeigt Abbildung 26 das erwartete Ergebnis für $\rho > Schwellwert$. Es wird die beste Klassifikationsgenauigkeit (87%) erreicht, da sehr viele Repräsentationen pro Klasse angelegt werden. Für ImageNet-10 ist dieses Verhalten nicht zu sehen. Dort werden die besten Ergebnisse (80-81%) für $\rho < Schwellwert$ erreicht. Bei $\rho > Schwellwert$ ist die Performanz stabil auf einem Niveau, allerdings nur bei ca. 77% Klassifikationsgenauigkeit. Ein Grund dafür könnte in den Bildern des ImageNet-Datensatzes liegen. Die Objekte sind in realer Umgebung zu sehen und der Hintergrund/die Umgebung ist nicht schwarz wie bei MNIST. Dadurch kann es vorkommen, dass der Feature-Extrahierer (Modul A) irrelevante Features aus dem Hintergrund extrahiert (zum Beispiel starkes Vorkommen der Farbe Blau im Hintergrund bei Flugzeugen aufgrund des Himmels). Diese extrahierten Features geben jedoch keine Information über das Objekt. So können Features von Flugzeugen gleich (oder sehr ähnlich) denen von Vögeln sein, wenn beide im Himmel mit blauem Hintergrund abgebildet sind. Deshalb können sich einzelne, nicht generalisierte Repräsentationen den Validationsbildern von anderen Klassen ähneln, da einzelne Bilder der unterschiedlichen Klassen aufgrund nicht relevanter Features einen ähnlichen Feature-Vektor besitzen.

Die genannten Schwellen spiegeln sich direkt im Speicherbedarf von Modul B wider. Ab dem genannten Schwellwert steigt der Speicherbedarf stark an, da viele Repräsentationen angelegt werden. Der mittlere Speicherbedarf von Modul B für Split-MNIST und ImageNet-10 ist in Abbildung 28 und Abbildung 29 graphisch dargestellt.

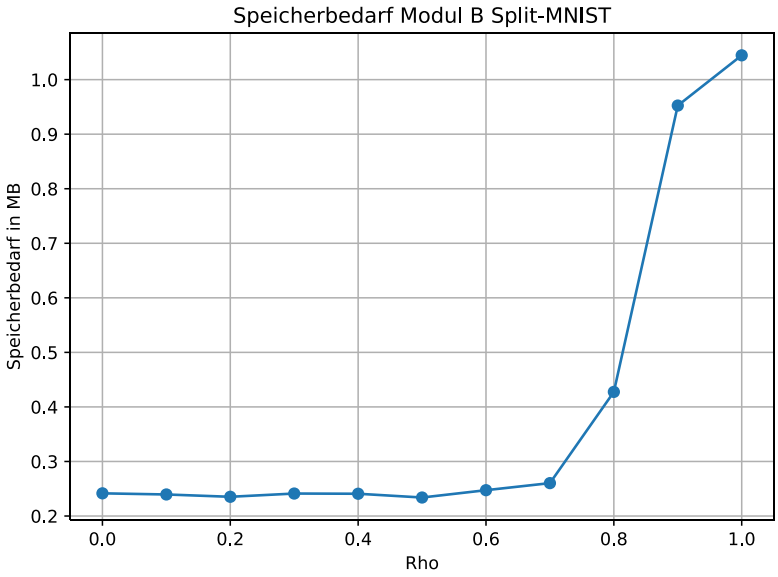

Abbildung 28: Speicherbedarf von Modul B in Abhängigkeit von ρ für Split-MNIST

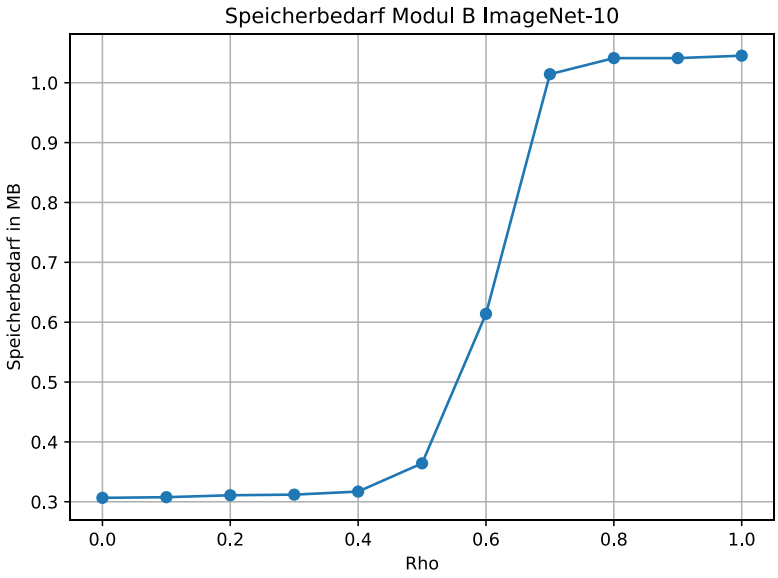

Abbildung 29: Speicherbedarf von Modul B in Abhängigkeit von ρ für ImageNet-10

In diesen Abbildungen ist zu sehen, dass der Speicherbedarf ab dem Schwellwert (~0,9 für Split-MNIST und ~0,7 für ImageNet-10) auf das 3- bist 4-fache des davor benötigten Speicherbedarfs ansteigt.

Aufgrund der geringen Anzahl an Trainingsbildern (20 pro Klasse) ist der Speicherbedarf mit ca. 1MB in diesen Beispielen immer noch gering. Bei späteren Anwendungen mit mehr Klassen oder Trainingsbildern kann der Speicherbedarf jedoch schnell sehr groß werden. Aufgrund der erzielten Ergebnisse hinsichtlich Genauigkeit und Speicherbedarf wird für ρ der Wert 0,5 gewählt. Mit diesem Parameterwert können auf beiden Datensätzen eine gute Klassifikationsgenauigkeit erzielt werden und der Speicherbedarf ist mit ca. 250 KB für Split-MNIST und ca. 350 KB für ImageNet-10 gering. Für weitere komplexere Anwendungen (z.B. gesamter ImageNet-Datensatz) wird der Speicherbedarf mit dieser Parametrierung in einem akzeptablen Bereich erwartet.

5.2 Einfluss der Anzahl von Trainingsdaten

Schnell lernende inkrementelle Klassifikatoren sollen auf Basis von wenigen Beispieldaten einer Klasse diese erlernen können. Ein wichtiges Indiz ist dabei die Klassifikationsgenauigkeit in Abhängigkeit der gesehenen Anzahl an Samples pro Klasse. Damit kann untersucht werden, ab wie vielen Trainingssamples ein Maximum an Genauigkeit erreicht werden kann und wie sich der Performanz-Gewinn durch eine höhere Anzahl an Trainingsbilder verhält. In diesem Evaluierungsfall wird untersucht, welchen Einfluss die Anzahl von Bildern einer neuen Klasse auf die spätere Klassifikationsgenauigkeit hat. Damit soll untersucht werden, wie viele Trainingsbilder pro Klasse notwendig sind, um eine neue Klasse zu erlernen und somit wie schnell das Netzwerk eine neue Klasse erlernt. Dafür wird das inkrementelle Klassenlernen auf Split-MNIST und ImageNet-10 durchgeführt und die Anzahl an Bilder einer neuen Klasse variiert. Zum Schluss wird die Klassifikationsgenauigkeit auf bisher nicht gesehenen Validationsdaten ermittelt. Mit diesen Ergebnissen kann eine Kurve der Klassifikationsgenauigkeit über die Anzahl an Trainingsbilder gezeichnet und der Einfluss bewertet werden.

Es werden die zuvor in Kapitel 5.1 ermittelten Hyperparameter von Modul B verwendet und die weiteren Parameter auf dieselben Werte gesetzt. Mit diesen Parametern wird als weitere Untersuchung die Anzahl an Trainingsbildern variiert. Die Anzahl der Trainingsbilder wird dabei auf folgende Werte gesetzt: 1, 2, 5, 10, 20, 50, 100, 500, 1000. Es werden pro Wert 5 Wiederholungen durchgeführt, um einen aussagekräftigen statistischen Mittelwert und die Standardabweichung bilden zu können.

Es wird erwartet, dass mit mehr Trainingsbildern pro Klasse bessere Ergebnisse erzielt werden können, da mehr und besser generalisierte Repräsentationen für die einzelnen Klassen angelegt werden können. Mit steigender Anzahl an Trainingsbildern pro Klasse wird vermutet, dass die Varianz der Klassifikationsgenauigkeit abnimmt und damit stabilere Ergebnisse erzielt werden können. Die Annahme ist, dass bei wenigen Trainingsbildern die spätere Klassifikationsgenauigkeit stark von der Auswahl dieser wenigen Trainingsbilder abhängig ist. Bei steigender Anzahl an Trainingsbilder sinkt die Abhängigkeit von den einzelnen Trainingsbildern. Zudem ist die Erwartung, dass der Speicherbedarf mit der Anzahl an Trainingsbildern steigt, da mehr

Repräsentationen durch die höhere Zahl an gesehenen Trainingsdaten angelegt werden.

Die Ergebnisse für die Klassifikationsgenauigkeit werden in einem Balkendiagramm dargestellt. Die schwarzen Linien stellen dabei die Standardabweichung um den Mittelwert dar. Aufgrund der großen Unterschiede zwischen den einzelnen Werten für die Anzahl an Trainingsbilder, wird die x-Achse in logarithmischer Skala dargestellt.

In Abbildung 30 ist die Klassifikationsgenauigkeit für Split-MNIST dargestellt.

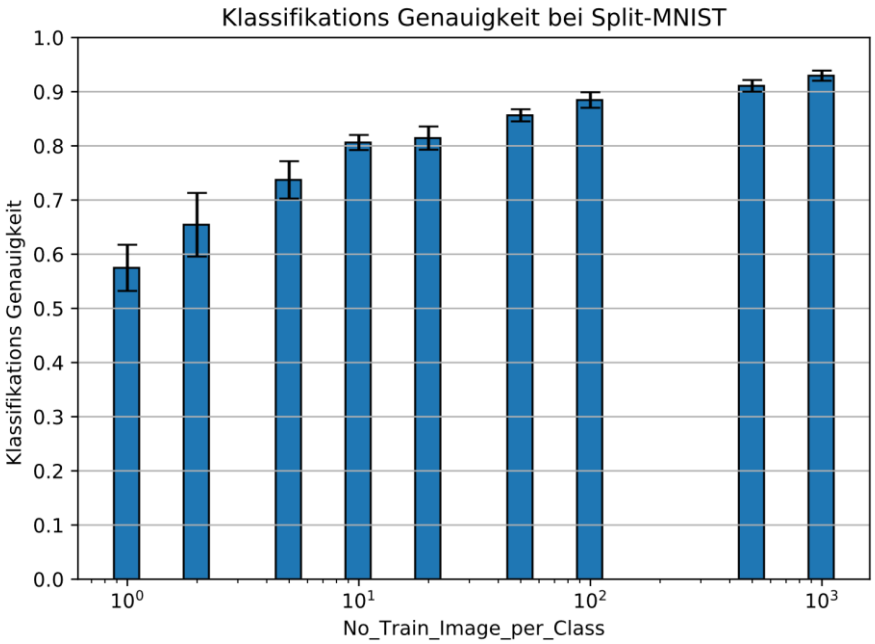

Abbildung 30: Klassifikationsgenauigkeit über die Anzahl an Trainingsbildern Split-MNIST

Abbildung 31 stellt die Klassifikationsgenauigkeit über die Anzahl an Trainingsbildern für ImageNet-10 dar.

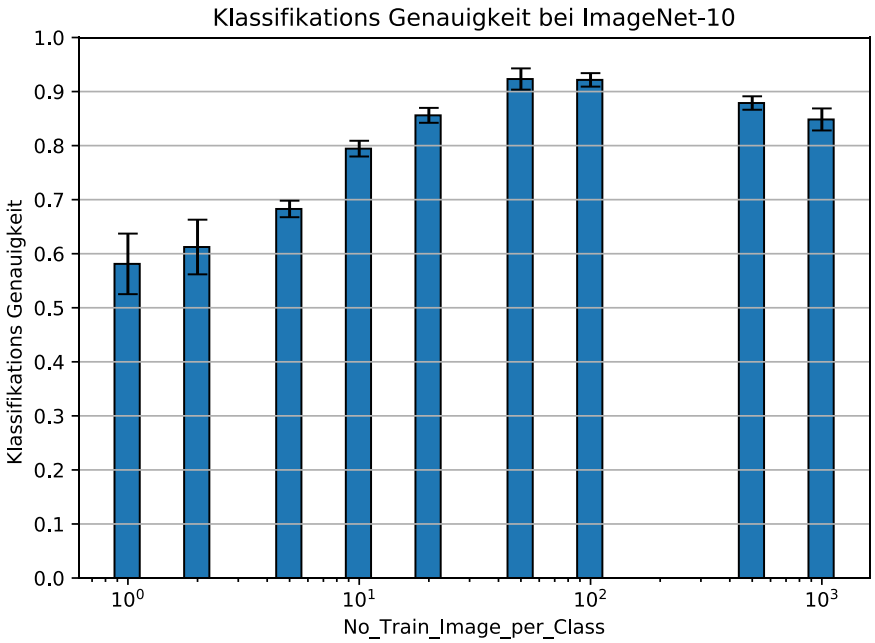

Abbildung 31: Klassifikationsgenauigkeit über die Anzahl an Trainingsbildern ImageNet-10

Die Annahme, dass mit einer geringen Anzahl an Trainingsbildern eine hohe Varianz auftritt, kann auf Basis der erzielten Ergebnisse bestätigt werden. Für Split-MNIST (Abbildung 30) beträgt die Standardabweichung bei einem beziehungsweise zwei Trainingsbildern pro Klasse 4,3 und 5,9 Prozentpunkte. Mit 500 und 1000 Trainingsbildern pro Klasse beträgt die errechnete Standardabweichung lediglich 1,1 und 0,9 Prozentpunkte. Bei ImageNet-10 ist ebenfalls eine höhere Standardabweichung bei einer geringen Anzahl Trainingsbildern zu sehen mit 5,6 und 5,1 Prozentpunkten bei einem und zwei Trainingsbildern pro Klasse. Die geringste Abweichung ist bei ImageNet-10 mit jeweils 1,2 Prozentpunkten für 100 und 500 Trainingsbildern pro Klasse zu finden. Wenn die mittlere Klassifikationsgenauigkeit betrachtet wird, kann für Split-MNIST gesagt werden, dass mit mehr Trainingsbildern pro Klasse die Genauigkeit verbessert wird. Bei 1000 Trainingsbildern pro Klasse wird im Mittel eine Klassifikationsgenauigkeit von 92,96% bei einer Standardabweichung von +/-0,94 erreicht. Mit 500 Trainingsbildern pro Klasse kann ebenfalls bereits eine gute Klassifikationsgenauigkeit von 91,08% +/- 1,1 erreicht werden. Für ImageNet-10 lässt sich ein anderes Verhalten der Klassifikationsgenauigkeit beobachten. Die Genauigkeit steigt zunächst bis 100 Trainingsbildern pro Klasse an, jedoch fällt sie im Gegensatz zu Split-MNIST daraufhin wieder ab. Ein möglicher Grund könnte der bereits beschriebene Fall von zu vielen Repräsentationen sein (siehe die Auswertung des Parameters ρ in Kapitel 5.1). Einzelne Repräsentationen ähneln möglicherweise den Bildern anderer Klassen mehr als der eigenen Klasse. Die beste Klassifikationsgenauigkeit lässt sich für ImageNet-10 mit 50 und 100 Trainingsbildern pro Klasse erzielen, mit 92,31% +/- 1,9 und 92,16% +/- 1,2.

Zusätzlich wird der mittlere Speicherbedarf von Modul B für die jeweiligen Datensätze in Abhängigkeit von der Anzahl an Trainingsbildern in Abbildung 32 für Split-MNIST und in Abbildung 33 für ImageNet-10 dargestellt.

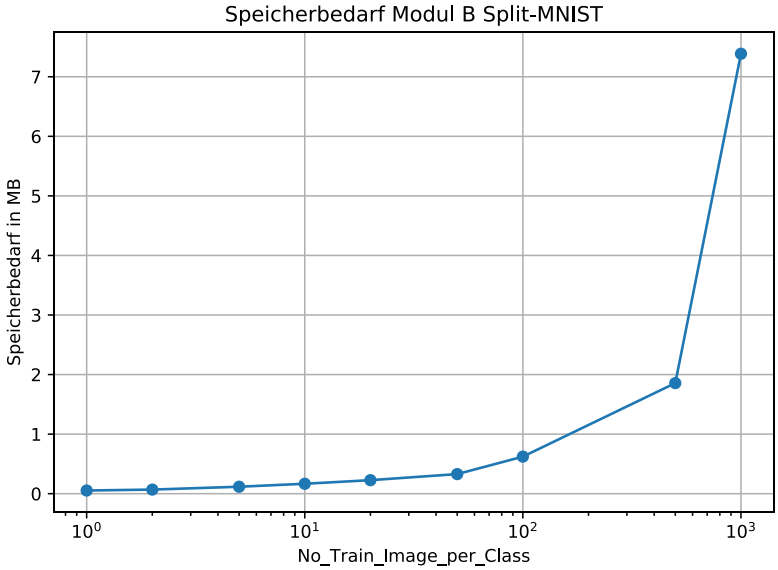

Abbildung 32: Speicherbedarf Modul B über die Anzahl an Trainingsbildern Split-MNIST

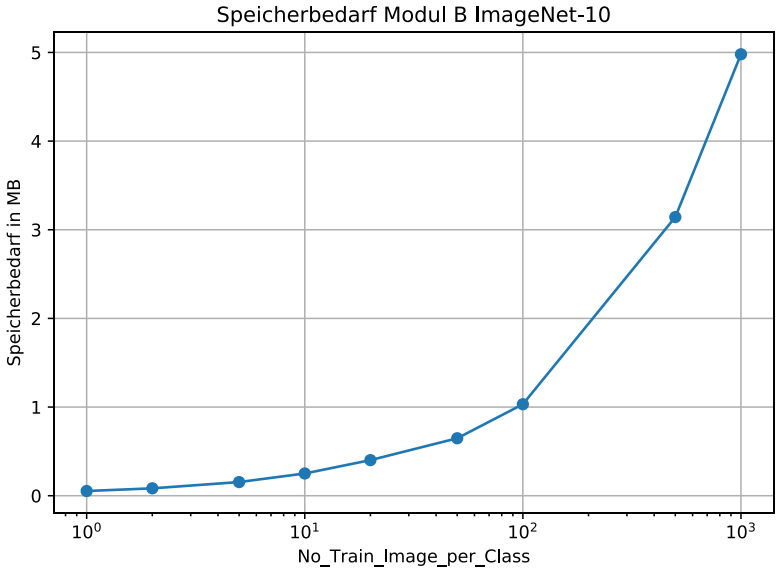

Abbildung 33: Speicherbedarf Modul B über die Anzahl an Trainingsbildern ImageNet-10

Wie bereits erwähnt, wird der der Speicherbedarf zusätzlich zu der Klassifikationsgenauigkeit in Betracht gezogen. Der Speicherbedarf steigt für Split-MNIST zwischen 500 und 1000 Trainingsbildern pro Klasse von ca. 1,9 MB auf ca. 7,2 MB an. Da Speicherbedarf in dieser Arbeit eine Rolle spielt, kann die leicht geringere Genauigkeit (ca. 1 Prozentpunkt) akzeptiert werden, wenn dafür weniger als ein Drittel an Speicher benötigt wird. Somit werden unter Berücksichtigung beider Metriken im

weiteren Verlauf der Arbeit 500 Trainingsbilder pro Klasse für Split-MNIST genutzt. Für ImageNet-10 sind bereits bei 50 und 100 Trainingsbildern pro Klasse die besten Ergebnisse zu sehen. Bei beiden Werten ist der Speicherbedarf mit ca. 0,75 und 1 MB akzeptabel. Aufgrund der minimal geringeren Varianz und damit dem besseren „Worst-Case“ Ergebnis (90,41% vs. 90,96%) werden 100 Trainingsbilder pro Klasse für die weiteren Untersuchungen mit dem ImageNet-10 Datensatz genutzt.

Es konnte gezeigt werden, dass die Anzahl an Trainingsbildern pro Klasse einen direkten Einfluss auf die spätere Klassifikationsgenauigkeit hat. Bereits mit wenigen Trainingsbildern pro Klasse (z.B. 10) können jedoch akzeptable Klassifikationsgenauigkeiten von ca. 80% für die genutzten Validationsdaten von Split-MNIST und ImageNet-10 erreicht werden. Generell kann gesagt werden, dass mit steigender Anzahl an Trainingsbilder die Klassifikationsgenauigkeit ebenso wie der Speicherbedarf des inkrementellen Klassifikators zunimmt. Je nach Anwendungsfall kann eine geringere Genauigkeit infolge von weniger Trainingsdaten akzeptiert werden, wenn z.B. die Erzeugung von Trainingsdaten sehr kosten- und zeitaufwändig ist. Mit diesen Ergebnissen kann gezeigt werden, dass der hier untersuchte Algorithmus für spätere Anwendungen, bei denen wenigen Daten pro Klasse verfügbar sind, geeignet ist, da er bereits mit wenigen Trainingsbildern pro Klasse gute Ergebnisse erzielen kann.

5.3 Ergebnisse Testdaten MNIST und ImageNet-10

Für eine Bewertung und Einordnung des Potenzials des L DNN Algorithmus werden auf Basis der zuvor untersuchten Hyperparameter finale Tests für das kontinuierliche und verteilte Lernen auf den beiden bisher genutzten Datensätzen (Split-MNIST und ImageNet-10) durchgeführt und soweit möglich mit anderen Algorithmen aus der Literatur verglichen. Es wird eine größere Anzahl an Wiederholungen mit festen Parametern durchgeführt. Zudem werden die finalen Ergebnisse auf Basis der Testdaten der Datensätze ermittelt, nachdem zuvor für die Untersuchungen der einzelnen Parameter Trainings- und Validationsdaten genutzt wurden. Somit werden für die Erstellung der folgenden Metriken Daten genutzt, die das Netzwerk bisher noch nicht gesehen hat. Dadurch kann eine Optimierung und *Overfitting* des Netzwerks auf diesen Testdaten verhindert werden.

5.3.1 Kontinuierliches Lernen

Es werden die Testfälle des kontinuierlichen Lernens auf einem Gerät durchgeführt. Die Parameter werden auf Basis der vorherigen Ergebnisse ausgewählt. Als Testdaten werden alle verfügbaren Testbilder der jeweiligen Datensätze genutzt. Pro Datensatz werden 10 Wiederholungen durchgeführt.

Für die Darstellung der Ergebnisse wird für Split-MNIST eine Tabelle mit Ergebnissen anderer *Continual Learning*-Verfahren sowie Ergebnisse mit traditionellen Deep Learning Ansätzen, hier einem Multy-Layer Perceptron (MLP), angelegt.

Für ImageNet-10 sind keine weiteren Ergebnisse von *Continual Learning*-Verfahren bekannt. Dieser Datensatz diente der Untersuchung der Parametereinflüsse auf komplexeren Eingangsdaten und der Anwendbarkeit des Algorithmus auf komplexere Aufgaben als MNIST. Für einen Vergleich zu anderen Verfahren werden Test auf dem gesamten ImageNet-Datensatz durchgeführt (siehe Kapitel 5.5).

In Tabelle 10 sind die Ergebnisse für den Split-MNIST Datensatz dargestellt. Die ersten beiden Zeilen sind dabei der hier untersuchte Algorithmus. *L DNN Algorithmus inkrementell* ist der wie in der Evaluierungsprozedur beschriebene inkrementelle Algorithmus. *L DNN Algorithmus gesamt* ist dieselbe Architektur mit derselben Parametrierung, jedoch werden die Trainingsbilder aller Klassen gemeinsam in einem großen Batch und nicht inkrementell trainiert. Bei den weiteren Algorithmen ist angegeben, von welcher Quelle die genannte Klassifikationsgenauigkeit stammt. Die Ergebnisse gelten für den Fall des inkrementellen Klassen Lernens.

Tabelle 10: Klassifikationsgenauigkeit verschiedener Algorithmen auf Split-MNIST

Algorithmus	Klassifikationsgenauigkeit in %
L DNN Algorithmus gesamt	90,66 +/- 0,32
L DNN Algorithmus inkrementell	86,94 +/- 1,29
Deep Generative Replay (DGR) **[13]**	91,24 +/- 0,33
Elastic Weight Consolidation (EWC) **[46]**	19,90 +/- 0,05
Multi-Layer Perceptron (MLP) – inkrementell trainiert **[46]**	19,90 +/- 0,02
MLP – offline trainiert **[46]**	97,93 +/- 0,04

EWC stellt eine typische Methode des kontinuierlichen Lernens dar. Diese Methode speichert keine Trainingsdaten und nutzt auch keine gespeicherten Repräsentationen zum Training. EWC gehört zu den Regularisierungsmethoden. Die Abspeicherung und Verwendung der gespeicherten Daten im weiteren Trainingsverlauf wird in der Literatur *Replay* oder *Rehearsal* genannt. Deep Generative Replay (DGR) nutzt diese Methode, in dem es komprimierte Repräsentationen der Trainingsdaten abspeichert. Wenn neue Klassen hinzukommen, werden aus den gespeicherten Komprimierungen der alten Klassen sowie mithilfe eines trainierten Generators Trainingsbilder dieser Klassen erzeugt (Generative) und in die neuen Trainingsdaten eingebracht. Das *inkrementell trainierte MLP* kann als untere Grenze gesehen werden, da hier katastrophales Vergessen aufgrund des Backpropagation-Algorithmus auftritt. Das *offline trainierte MLP* wurde mit allen Klassen offline („klassisch“) trainiert und kann als obere Grenze angesehen werden. Im Vergleich zu EWC kann der L DNN Algorithmus deutlich bessere Ergebnisse (86,94% gegenüber 19,90%) für das hier untersuchte inkrementelle Klassen Lernen erzielen.

Algorithmen, die mit generativen Methoden arbeiten (wie DGR), erreichen eine bessere Klassifikationsgenauigkeit für diesen Testfall. Allerdings besitzen sie auch eine deutlich erhöhte Komplexität während des Trainings. Bei diesen Modellen muss zusätzlich zu dem inkrementellen Klassifikator ein generatives Modell trainiert werden, welches ausgewählte Trainingsdaten komprimiert und aus den komprimierten Darstellungen wiederherstellt (z.B. mithilfe eines *Variational Auto-Encoder*). Zudem müssen diese komprimierten Darstellungen abgespeichert werden. All diese Punkte ermöglichen generativen Modellen eine bessere Klassifikationsgenauigkeit: Jedoch sind diese Modelle aus den genannten Gründen (noch) nicht für den Einsatz auf einem mobilen Endgerät geeignet, was in dieser Arbeit ein wichtiger Auswahlpunkt für den Algorithmus war. Insgesamt ist festzuhalten, dass eine Genauigkeit von ca. 87% auf Split-MNIST ein sehr gutes Resultat für das inkrementelle Klassenlernen ohne Replay/Rehearsal ist. Wenn lediglich Algorithmen ohne Rehearsal betrachtet werden, konnte in der Literatur keine bessere Klassifikationsgenauigkeit auf diesem Datensatz für das inkrementelle Klassen Lernen gefunden werden. Als zusätzliche Referenz dient der *L DNN Algorithmus gesamt* mit 90,66% Genauigkeit. Dabei ist zu sehen, dass durch das inkrementelle Erlernen der Klassen ca. 3,5 Prozentpunkte Klassifikationsgenauigkeit verloren geht bei Split-MNIST. Der Algorithmus ist somit auch in der Lage eine gute Genauigkeit über „klassisches“ Training zu ermöglichen. Die Daten müssen somit nicht inkrementell eingespeist werden.

Für ImageNet-10 wird eine gemittelte Klassifikationsgenauigkeit von 76,4% +/-1,2 beim inkrementellen Erlernen erreicht. Vergleichbare Ergebnisse für diesen Anwendungsfall sind in der Literatur nicht zu finden. Als Referenz wird das Training der Architektur mit den Trainingsbildern aller Klassen herangezogen (*L DNN Algorithmus gesamt*). Mit diesem Training wird eine Klassifikationsgenauigkeit von 76,08% +/- 1,67 erreicht. Die Ergebnisse für ImageNet-10 können nicht mit anderen Algorithmen verglichen werden, jedoch kann damit geprüft werden, ob der Algorithmus auch auf komplexeren Eingangsdaten (64x64 RGB-Bilder) mit komplexen Klassen funktioniert, bevor ein großer und aufwändiger Test auf dem gesamten ImageNet-Datensatz durchgeführt wird. Mit einer finalen mittleren Klassifikationsgenauigkeit von 76,4% kann gesagt werden, dass der Algorithmus auch komplexe Klassen und Eingangsdaten korrekt klassifizieren und inkrementell erlernen kann. Auch wird beim inkrementellen Klassen Lernen eine identische Genauigkeit (sogar minimal besser) wie beim Training der Architektur mit allen Trainingsbildern der Klassen (*L DNN Algorithmus gesamt*) erreicht.

5.3.2 Verteiltes Lernen

Die Testfälle des verteilten Lernens werden auf zwei virtuellen Geräten durchgeführt. Es werden zwei unabhängige Instanzen des L DNN Algorithmus erzeugt. Die Parameter werden auf Basis der vorherigen Testfälle ausgewählt. Als Testdaten werden alle verfügbaren Testbilder der jeweiligen Datensätze genutzt. Pro Datensatz werden 10 Wiederholungen pro Datensatz durchgeführt.

Für den Fall des verteilten Lernens auf zwei Geräten wird die Klassifikationsgenauigkeit der einzelnen Geräte nach dem Erlernen ihrer gesehenen lokalen Klassen angegeben. Zusätzlich wird die finale Genauigkeit des „verschmolzenen“ Netzwerks auf allen Klassen bestimmt. Als Referenz dienen die Ergebnisse des kontinuierlichen Lernens auf einem Gerät, da im besten Fall durch das verteilte Lernen keine schlechteren Ergebnisse erzielt werden.

Zunächst findet die Auswertung für Split-MNIST statt. Dabei wurden auf Gerät 1 die Gruppen 0/1, 2/3 und 4/5 trainiert, während auf Gerät 2 die Gruppen 6/7 und 8/9 trainiert wurden. Die gemittelten Ergebnisse aus 10 Läufen sowie deren Standard-Abweichung sind in Tabelle 11 gegeben.

Tabelle 11: Klassifikationsgenauigkeit des verteiltem L DNN Algorithmus auf Split-MNIST

Algorithmus	Klassifikations-genauigkeit Gerät 1 in %	Klassifikations-genauigkeit Gerät 2 in %	Klassifikations-genauigkeit final in %
L DNN Algorithmus 1 Gerät	-	-	86,94 +/- 1,29
L DNN Algorithmus 2 Geräte	88,44 +/- 0,82	95,02 +/- 0,44	86,56 +/- 0,91

Dieselbe Evaluation wird auf Basis des ImageNet-10 Datensatzes durchgeführt. Die Ergebnisse dieser Untersuchung sind in Tabelle 12 zu sehen. Die beiden Geräte haben hier dieselbe Anzahl an Klassen (fünf) gesehen. Die jeweiligen Klassen, die auf den Geräten trainiert werden, sind ebenso wie die Reihenfolge der Klassen zufällig ausgewählt.

Tabelle 12: Klassifikationsgenauigkeit des verteiltem L DNN Algorithmus auf ImageNet-10

Algorithmus	Klassifikations-genauigkeit Gerät 1 in %	Klassifikations-genauigkeit Gerät 2 in %	Klassifikations-genauigkeit final in %
L DNN Algorithmus 1 Gerät	-	-	76,40 +/- 1,2
L DNN Algorithmus 2 Geräte	84,64 +/- 4,73	87,44 +/- 4,04	76,26 +/- 1,5

Die Ergebnisse für Split-MNIST und ImageNet-10 zeigen, dass mit separatem Training auf mehreren Geräten (hier zwei) dieselbe Genauigkeit erreicht werden kann wie mit dem Training auf einem einzelnen Gerät (siehe Tabelle 11 und Tabelle 12). Zudem kann gesagt werden, dass durch das „Verschmelzen“ von Wissen eine schlechtere Genauigkeit auftritt. Die einzelnen Genauigkeiten von Gerät 1 und Gerät 2 auf ihren jeweiligen Testdaten sind besser als die finale Klassifikationsgenauigkeit. Dies hängt

mit der komplexeren finalen Aufgabe zusammen, da dort statt 6 und 4 (für Split-MNIST) oder 5 und 5 Klassen (für ImageNet-10) nun 10 Klassen im Test-Datensatz vorkommen. Insgesamt kann auf Basis dieser Ergebnisse gesagt werden, dass das verteilte Lernen mit diesem Algorithmus funktioniert, da dort kein nennenswerter Performanz Verlust gegenüber dem zentralen Erlernen der Aufgabe auf einem Gerät auftritt. Somit eignet sich dieser Algorithmus zum Einsatz auf verteilten Systemen, die ihr Wissen ohne die jeweiligen Rohdaten austauschen.

5.4 Einfluss von Konsolidierungsschritten

Wie in Kapitel 3.2 beschrieben, kann das erlernte Wissen des inkrementellen Klassifikator konsolidiert werden. In diesem Prototyp wird die Konsolidierung durch eine Mittelwertbildung aller Repräsentationen einer einzelnen Klasse realisiert. Durch die Konsolidierung soll der Speicherbedarf deutlich verringert werden. In diesem Testfall wird nun der genaue Einfluss der Konsolidierung auf die Klassifikationsgenauigkeit und den Speicherbedarf des inkrementellen Klassifikator in Modul B untersucht.

Es wird dieselbe Prozedur wie für die finalen Untersuchungen des kontinuierlichen Lernens genutzt (siehe Kapitel 5.3.1). Zusätzlich wird eine Konsolidierung der Repräsentationen durchgeführt. Um den Einfluss zu untersuchen, werden zwei Testfälle durchgeführt. Beim ersten Testfall wird nach dem Training jeder Gruppe eine Konsolidierung durchgeführt. Dieser Testfall wird im weiteren Verlauf in Graphiken und Tabellen „*Konsolidierung jeder Schritt*“ genannt. Im zweiten Testfall findet keine Konsolidierung während des Trainings der einzelnen Gruppen statt. Die Konsolidierung findet nach dem Training aller Gruppen/Klassen und vor der Bestimmung der Test-Genauigkeit statt. Dieser Fall wird im weiteren Verlauf „*Konsolidierung finaler Schritt*“ genannt. Als Referenz dienen die zuvor erzielten Ergebnisse ohne Konsolidierung (siehe Tabelle 10). Für die beiden Testfälle werden jeweils 10 Wiederholungen durchgeführt. Die im Folgenden genannten Ergebnisse sind die Mittelwerte der unterschiedlichen Läufe. Die Tests werden für Split-MNIST und ImageNet-10 durchgeführt.

Zunächst wird die Klassifikationsgenauigkeit der unterschiedlichen Testfälle auf Basis von Split-MNIST dargestellt. Die Ergebnisse für die beiden Testfälle sowie der Referenz (Keine Konsolidierung) sind in Abbildung 34 zu sehen. Die unterschiedlichen Kurven stellen dabei die Genauigkeiten für die jeweiligen Methoden der Konsolidierung über die Anzahl an erlernten Klassen dar. Die Ergebnisse für ImageNet-10 sind in Abbildung 35 zu sehen.

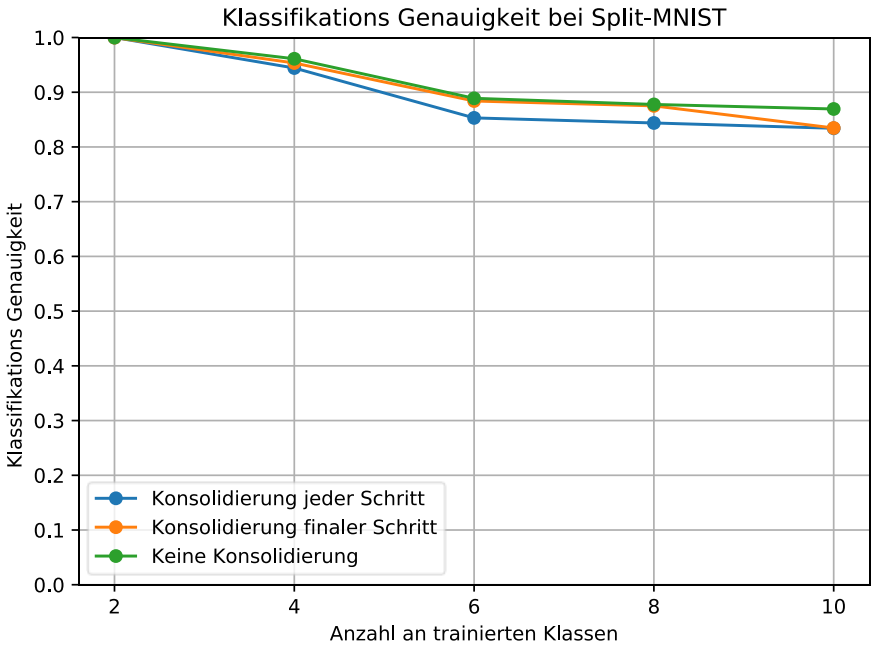

Abbildung 34: Klassifikationsgenauigkeit für unterschiedliche Konsolidierungsmethoden Split-MNIST

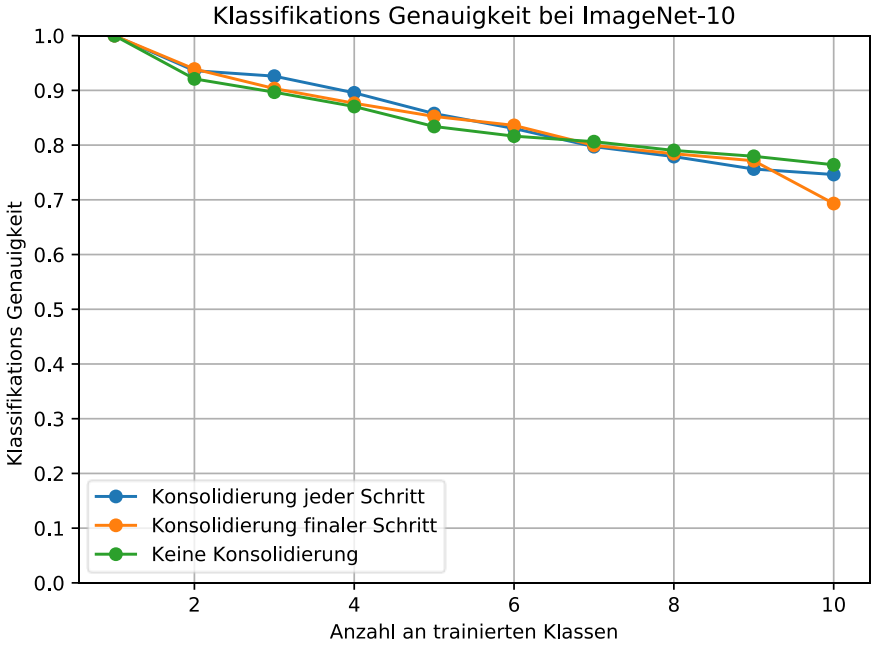

Abbildung 35: Klassifikationsgenauigkeit für unterschiedliche Konsolidierungsmethoden ImageNet-10

Für eine sinnvolle Einschätzung und Bewertung wird zusätzlich der finale Speicherbedarf der unterschiedlichen Methoden betrachtet. Dieser wird inklusive der finalen Klassifikationsgenauigkeiten für Split-MNIST in Tabelle 13 zusammengefasst. Tabelle 14 zeigt diesen Zusammenhang für ImageNet-10.

Tabelle 13: Vergleich von Speicherbedarf und finaler Klassifikationsgenauigkeit für unterschiedliche Methoden der Konsolidierung Split-MNIST

Konsolidierungs-Methode	Finale Klassifikations-Genauigkeit Split-MNIST in %	Finaler Speicherbedarf Split-MNIST in MB
Keine Konsolidierung	86,94 +/- 1,29	1,83
Jeder Schritt	83,41 +/- 1,64	0,1
Finaler Schritt	83,47 +/- 1,8	0,1

Tabelle 14: Vergleich von Speicherbedarf und finaler Klassifikationsgenauigkeit für unterschiedliche Methoden der Konsolidierung ImageNet-10

Konsolidierungs-Methode	Finale Klassifikations-Genauigkeit ImageNet-10 in %	Finaler Speicherbedarf ImageNet-10 in MB
Keine Konsolidierung	76,40 +/- 1,2	0,96
Jeder Schritt	74,62 +/- 2,19	0,1
Finaler Schritt	69,32 +/- 4,81	0,1

Anhand der erzielten Ergebnisse kann gesagt werden, dass bei Anwendungen mit sehr begrenztem Speicher die Konsolidierung eine gute Maßnahme sein kann, um den Speicherbedarf drastisch zu reduzieren und dabei geringe Genauigkeitseinbußen zu haben. Für Split-MNIST wird eine finale Klassifikationsgenauigkeit von ca. 83% mit beiden Konsolidierungsmethoden erreicht, was ca. 3 Prozentpunkte schlechter ist als ohne Konsolidierung. Jedoch ist der Speicherbedarf mit 0,1 MB auch nur 1/18 im Vergleich zu keiner Konsolidierung (siehe Tabelle 13). In Abbildung 34 kann gezeigt werden, dass bei der *Konsolidierung finaler Schritt* die Genauigkeit bis zum letzten finalen Testlauf identisch bleibt, da dort noch keine Konsolidierung stattgefunden hat und somit das Training und Testen bis dorthin identisch ist. Für Split-MNIST kann kein direkter Unterschied zwischen den beiden Konsolidierungsmethoden („*Jeder Schritt*“ und „*Finaler Schritt*“) gesehen werden, da beide nahezu identische Ergebnisse liefern. Diese Aussage gilt nicht für ImageNet-10. Auch hier ist der Speicherbedarf mit 0,1 MB im Vergleich zu 0,96 MB um einiges geringer. Bei der Klassifikationsgenauigkeit gibt es jedoch auffällige Unterschiede. Mit der *Konsolidierung jeder Schritt* kann mit 74,62% Genauigkeit ein ähnliches Resultat wie ohne Konsolidierung (76,4%) erreicht werden. Bei der *Konsolidierung finaler Schritt* wird eine Genauigkeit von lediglich 69,32% mit deutlich erhöhter Varianz (4,81) erzielt (siehe Tabelle 14). Dies verdeutlicht den Einfluss der Konsolidierungsmethode für ein FuzzyARTMAP-Netzwerk, da das Training eines solchen Netzwerks stark von bereits vorhandenen Repräsentationen abhängt.

Der Grund für die schlechtere Performanz der Methode *finaler Schritt* könnte darin liegen, dass während der Trainingsschritte viele Repräsentationen anderer Klassen vorliegen. Dies führt in der Regel zu einer erhöhten Anzahl an Repräsentationen für neue Klassen, um einen eindeutigen Sieger für diese Kategorie stellen zu können. Für einen Datensatz mit komplexen Klassen und Bilder wie ImageNet wird bei einer Konsolidierung im finalen Schritt separates Wissen über eine Klasse/Kategorie mit einer simplen Mittelwert-Logik verbunden. Dadurch werden unterschiedliche Repräsentationen „in ihrer Mitte“ zusammengefasst. Wenn die Methode „*Jeder Schritt*“ angewendet wird, ist während des Trainings einer neuen Klasse nur eine Repräsentation pro alte Klassen vorhanden. Dadurch bildet das FuzzyARTMAP-Netzwerk weniger Repräsentationen, die jedoch mithilfe der optimierten Lernrate α generalisiert sind. Diese generalisierten Repräsentationen scheinen robuster zu sein als die mithilfe des Mittelwertes konsolidierten Repräsentationen.

Alles in allem kann mithilfe der Konsolidierung der Speicherbedarf drastisch reduziert werden. In Abhängigkeit der gewählten Konsolidierungsmethode wird dabei die Klassifikationsgenauigkeit nur leicht verringert. Mit der Methode j*eder Schritt* können ähnliche Ergebnisse wie ohne Konsolidierung erreicht werden mit 1/18 (Split-MNIST) und 1/9 (ImageNet-10) des ursprünglichen Speicherbedarfs. Dies zeigt, dass Konsolidierung für eine spätere Anwendung in einem speicherarmen Endgerät interessant sein kann.

Als Ausblick könnte für die Konsolidierung eine mathematisch komplexere Formel anstatt dem simplen Mittelwert genutzt werden. Zum Beispiel könnten nur Repräsentationen einer Klasse konsolidiert werden, die eine Ähnlichkeit (z.B. *cosine*-Similarity) über einem definierten Schwellwert haben. Dadurch kann das Konsolidieren von im Feature-Raum unterschiedlichen Repräsentationen einer Klasse vermieden werden. Als Folge würden die Klassen nicht durch eine, sondern durch mehrere ausgewählten Repräsentationen dargestellt, die wiederum eine Vielzahl an ähnlichen Repräsentationen bündeln und darstellen. Dies kann im Rahmen dieser Arbeit jedoch aufgrund von Zeitgründen nicht weiterverfolgt werden.

Als weiteren Ausblick kann mit der hier vorhandenen Logik ein intelligenter Zeitpunkt für die Konsolidierung gewählt werden. Es muss nicht nach dem Training einer neuen Klasse stattfinden, wenn dafür keine Notwendigkeit herrscht. In realen Anwendungsfällen könnte die Konsolidierung zum Beispiel zu dem Zeitpunkt erfolgen, an dem z.B. 90% des verfügbaren Speichers belegt ist. Wie in den hier beschriebenen Testfällen gezeigt, kann bei geringem Genauigkeitsverlust (2-3 Prozentpunkte) eine Menge Speicher freigegeben werden (Reduktion auf 1/18 und 1/9 des ursprünglichen Bedarfs).

5.5 ImageNet

Für eine finale Untersuchung des Algorithmus auf einem komplexen Datensatz mit vielen Klassen wird der ImageNet-2012 Datensatz genutzt (siehe Kapitel 4.1). Die Ergebnisse auf diesem Datensatz dienen als Vergleich zu anderen inkrementellen

Klassifikatoren. Aus Zeitgründen konnte lediglich das inkrementelle Klassenlernen auf einem Gerät untersucht werden auf diesem Datensatz. Es werden die in den vorherigen Kapiteln optimierten Parameterwerte für ImageNet-10 auch für den gesamten ImageNet Datensatz genutzt. Aufgrund der großen Anzahl an Klassen und dem damit verbundenen Rechenaufwand werden lediglich 10 Trainingsbilder pro Klasse genutzt anstatt 100 wie für ImageNet-10. Es wird auch anstatt wie bisher 10 lediglich eine Wiederholung durchgeführt. Als Referenz werden die in [47] aufgeführten Ergebnisse für ImageNet genutzt.

Zunächst wurde der Algorithmus mit einem inkrementellen Schritt trainiert. Das heißt, dass kein inkrementelles Lernen stattfindet, sondern alle 1000 Klassen im ersten Trainingsschritt erlernt werden. Es konnte eine finale Klassifikationsgenauigkeit auf den Testdaten von 39,1% erreicht werden bei einem Speicherbedarf von 85,2 MB für Modul B.

Mit 10 inkrementellen Schritten (100 Klassen pro Schritt) wird am Ende auf allen Testdaten eine Klassifikationsgenauigkeit von 39,3% bei einem Speicherbedarf von 87,2 MB erreicht. Da diese Anzahl an inkrementellen Schritten auch in [47] für die finale Auswertung genutzt wird, werden die erreichten Ergebnisse in Relation zu dem bereits in der Konzeption beschriebenen *iCaRL*-Algorithmus sowie dem *Learning without Forgetting* (LwF) [48] gesetzt. Beide Algorithmen nutzen die Dual-Memory Methode, weshalb sie als Referenz ausgewählt wurden. Die Ergebnisse sind in Abbildung 36 dargestellt.

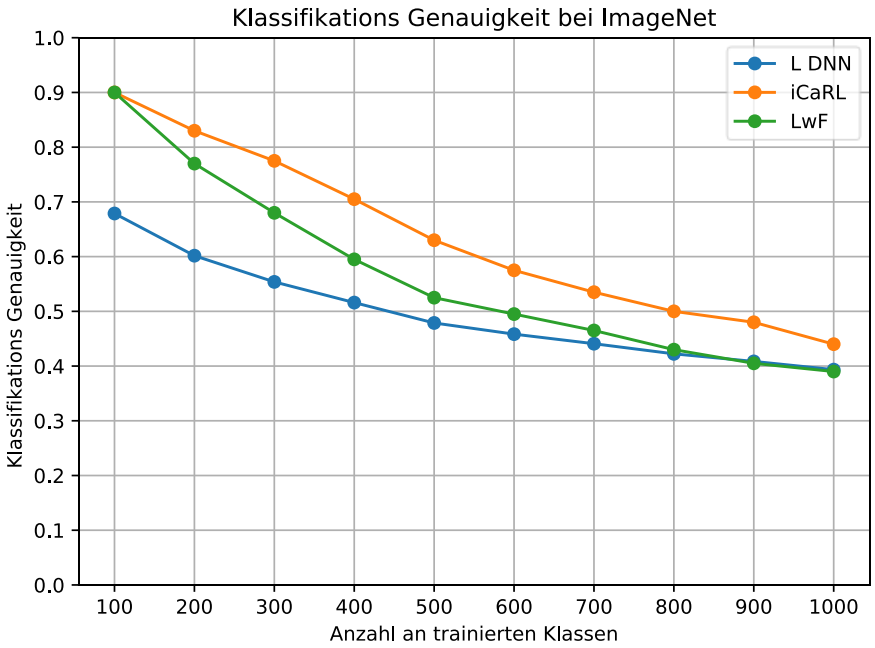

Abbildung 36: Klassifikationsgenauigkeit bei ImageNet für unterschiedliche inkrementelle Lernalgorithmen

Zusätzlich wird in Abbildung 37 zu den bereits beschriebenen Kurven die Klassifikationsgenauigkeit des L DNN Algorithmus mit 20 inkrementellen Schritten

dargestellt (rote Kurve). Die anderen Kurven sind identisch zu Abbildung 36 und nur als Referenz eingezeichnet.

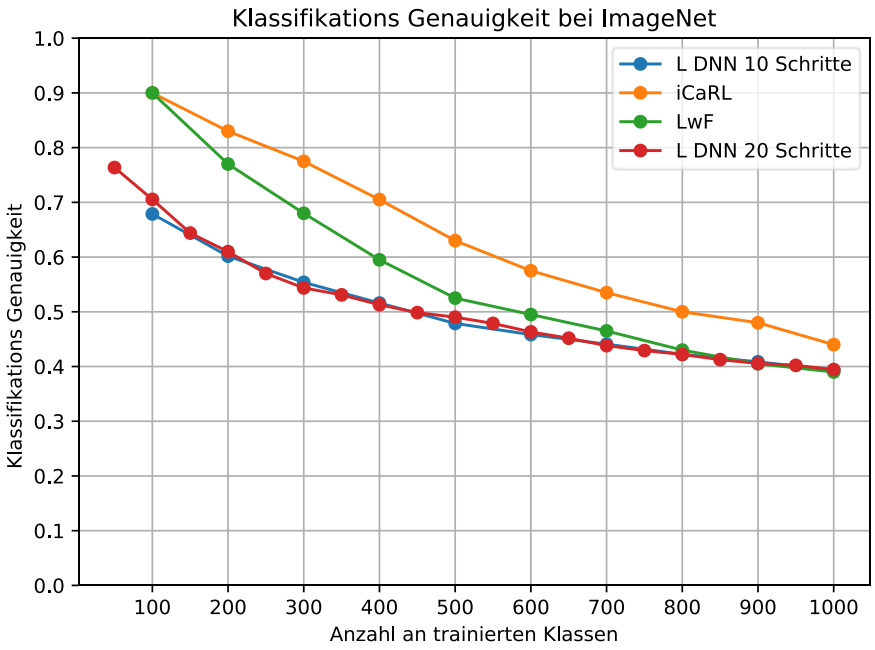

Abbildung 37: Klassifikationsgenauigkeit bei ImageNet mit unterschiedlicher Anzahl an inkrementellen Schritten

Beide Algorithmen, iCaRL und LwF, nutzen ein 18-Layer ResNet zur Feature-Extrahierung. Der iCaRL-Algorithmus speichert 20.000 Exemplare in dem referenzierten Versuch ab. LwF arbeitet ohne Speicherung von Exemplaren und trainiert neue unabhängige Fully Connected Layer für jeden inkrementellen Schritt. Jeder inkrementelle Lernschritt hat bei diesen beiden Algorithmen 100 Epochen, während der L DNN Algorithmus lediglich eine Epoche pro inkrementellen Schritt durchführt. In Tabelle 15 sind die finalen Genauigkeiten und der Speicherbedarf der genannten Algorithmen zusammengefasst.

Tabelle 15: Finale Klassifikationsgenauigkeiten ImageNet

Algorithmus	Finale Klassifikations-Genauigkeit ImageNet in %	Finaler Speicherbedarf ImageNet-10 in MB
L DNN (ein inkrementeller Schritt)	39,1	527,2
L DNN (10 inkrementelle Schritte)	39,3	530,5
L DNN (50 inkrementelle Schritte)	39,4	528,8
iCaRL	44	2123
LwF	39	-

Für L DNN ist der gesamte Speicherbedarf für Modul A und Modul B angegeben. Für iCaRL und LwF konnten keine konkreten Angaben zum Speicherbedarf gefunden werden. Der Speicherbedarf für iCaRL wurde mithilfe der durchschnittlichen Größe eines Bildes (ca. 106 KB) des ImageNet-Datensatz für 20.000 gespeicherte Exemplare ermittelt. Zusätzlich kommt noch der Speicherbedarf des Feature-Extrahierer hinzu. Für LwF konnte keine Angabe für den Speicherbedarf gefunden werden und auch keine sinnvolle Abschätzung getroffen werden.

Auf Basis der erzielten Ergebnisse in Tabelle 15, Abbildung 36 und Abbildung 37 wird der L DNN Algorithmus final bewertet. Dafür wird zunächst der L DNN Algorithmus mit 10 inkrementellen Schritten mit iCaRL und LwF verglichen. In Abbildung 36 ist zu sehen, dass der L DNN Algorithmus vom ersten inkrementellen Schritt (100 Klassen) an eine deutlich geringe Klassifikationsgenauigkeit hat (67,9%) als iCaRL und LwF (beide 90%). Dies kann mit dem gewählten Feature-Extrahierer begründet werden. Mit MobileNet-v2 wurde für den L DNN Algorithmus ein Kompromiss zwischen Speicherbedarf und Genauigkeit getroffen. LwF und iCaRL nutzen jeweils eine ResNet-Architektur für die Feature-Extraktion. Diese Architektur erzielt auf ImageNet bessere Genauigkeiten als MobileNet-v2 (siehe Kapitel 3.3.1), wodurch eine höhere Klassifikationsgenauigkeit mit den hier genutzten Algorithmen möglich ist. Ebenfalls ist in Abbildung 36 der Verlauf über die Anzahl an trainierten Klassen zu sehen. Der L DNN Algorithmus ist hier wesentlich stabiler als die anderen beiden Algorithmen, welche mit zunehmender Anzahl an trainierten Klassen deutlich stärker an Genauigkeit verlieren.

Tabelle 16 stellt den relativen Erhalt der Klassifikationsgenauigkeit für die einzelnen Algorithmen dar. Dafür wird die Genauigkeit nach dem letzten finalen Schritt (1000 Klassen) durch die Genauigkeit nach dem ersten Schritt (100 Klassen) dividiert, um das relative Verhältnis zu erhalten.

Tabelle 16: Relativer Erhalt der Klassifikationsgenauigkeit auf ImageNet

Algorithmus	Relativer Erhalt Klassifikationsgenauigkeit in %
L DNN (10 inkrementelle Schritte)	57,9
iCaRL	48,9
LwF	43,3

Der L DNN Algorithmus erreicht nach dem finalen Training auf 1000 Klassen noch ca. 58% seiner anfänglichen Genauigkeit, während iCaRL und LwF mit ca. 49% (iCaRL) und ca. 43% (LwF) jeweils weniger als die Hälfte erreichen. Diese Ergebnisse deuten darauf hin, dass der L DNN Algorithmus für eine große Anzahl an Klassen stabiler ist und im Vergleich zu den anderen Algorithmen auf großen Daten „besser skaliert“. Die finale Genauigkeit des L DNN Algorithmus ist dabei identisch wie die des LwF-

Algorithmus, der jedoch eine komplexere Trainingsstrategie verfolgt (Training einzelner Fully-Connected Layer mithilfe des Backpropagation-Algorithmus auf Basis der neuen Samples). Hier ist kein Vergleich beim Speicherbedarf möglich, da für LwF keine Werte vorliegen und der Speicherbedarf nicht abgeschätzt werden kann. Der iCaRL-Algorithmus erreicht eine bessere finale Klassifikationsgenauigkeit mit 44% gegenüber dem L DNN Algorithmus (ca. 39%), jedoch ist der Speicherbedarf hier um einiges höher. Allein für die Speicherung der Exemplare benötigt iCaRL bei diesem Test über 2 GB, während der gesamte L DNN Algorithmus lediglich ca. 530 MB benötigt. Unter Berücksichtigung der genutzten Feature-Extrahierer erscheint es möglich, dass der L DNN Algorithmus mit einem leistungsfähigeren Extrahierer eine bessere Genauigkeit als iCaRL erreichen kann. Der Speicherbedarf des L DNN würde steigen, jedoch vermutlich weiterhin geringer sein als der des iCaRL. Der gesteigerte Speicherbedarf könnte in einzelnen Anwendungsfällen für eine bessere Genauigkeit akzeptiert werden.

Im Weiteren wird der Einfluss von einer unterschiedlichen Anzahl an inkrementellen Schritten für den L DNN Algorithmus bei einem großen Datensatz untersucht. Hierzu wurden 20 statt 10 inkrementelle Schritte durchgeführt. Damit werden bei einem inkrementellen Trainingsschritt 50 statt 100 Klassen trainiert. Auf Basis der in [47] angegebenen Resultate führt eine erhöhte Anzahl an inkrementellen Schritten bei LwF und iCaRL zu einer schlechteren finalen Klassifikationsgenauigkeit. Speziell LwF ist deutlich sensitiver zu der Anzahl an inkrementellen Trainingsschritten. Dieses Verhalten ist für den L DNN Algorithmus nicht zu beobachten. In Abbildung 37 kann kein nennenswerter Unterschied zwischen den Kurven mit 10 und 20 inkrementellen Schritten gesehen werden. Auch die finale Klassifikationsgenauigkeit ist (nahezu) identisch. Dieses Verhalten ist für spätere reale Anwendungen von großer Bedeutung. In Anwendungen des kontinuierlichen Lernens können viele inkrementelle Schritte auftreten, da einzelne Klassen nach und nach während dem Betrieb auftreten. Dort soll eine gute Klassifikationsgenauigkeit ermöglicht werden. In diesem Fall ist für LwF und iCaRL eine deutlich geringere Performanz im Vergleich zu den bekannten Ergebnissen auf den hier genutzten Test-Datensätzen zu erwarten. Für den L DNN Algorithmus kann erwartet werden, dass die hier gezeigte und erreichte Performanz auch bei vielen inkrementellen Schritten erreicht werden kann.

In Zukunft sollte für den L DNN Algorithmus untersucht werden, inwieweit die Klassifikationsgenauigkeit des Algorithmus durch einen besseren Feature-Extrahierer in Modul A verbessert werden kann. Wenn hier eine weitere Verbesserung der Genauigkeit erreicht werden kann, hat der Algorithmus das Potenzial auch in realen Anwendungen sehr gute Resultate zu erzielen. Zudem wäre eine Untersuchung auf ImageNet mit 1000 inkrementellen Schritten (1 Klasse pro Schritt) interessant für die im vorigen Abschnitt getroffene Aussage. Weiterhin sollte das Verhalten des verteilten Lernens für 2 (oder mehrere) Endgeräte auf ImageNet untersucht werden.

6 Demonstrator

Im Rahmen dieser Arbeit wird zusätzlich zu der prototypischen Umsetzung ein Demonstrator für das inkrementelle Klassen- (oder Objekt-) Lernen in Echtzeit auf einem speicher- und rechenbegrenztem Edge Device aufgebaut. Mithilfe dieses Demonstrators soll gezeigt werden, dass der L DNN Algorithmus auf mobilen Endgeräten mit eingeschränktem Rechen- und Speicherbedarf lauf- und lernfähig ist. Der Aufbau des Demonstrators mit den einzelnen verwendeten Hardware-Bauteilen ist in Abbildung 38 zu sehen.

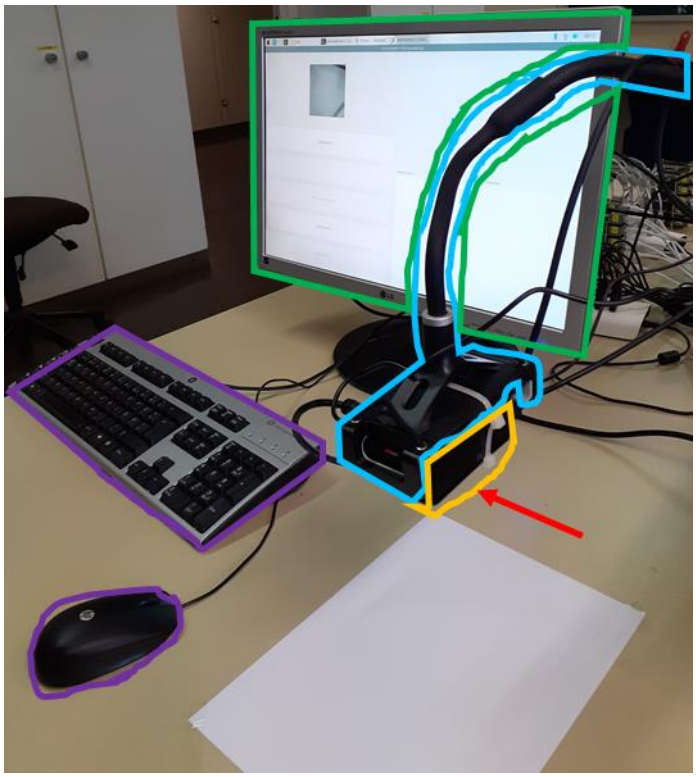

Abbildung 38: Demonstrator-Aufbau

Der Demonstrator besteht aus einem Raspberry Pi 3 Model B, der in einem Gehäuse platziert ist (orange). An diesem Raspberry Pi ist eine PiCamera V2 für die Bildaufnahme angeschlossen (rot, am Gehäuse montiert). Um ein sauberes und stabiles Bild des Objektes aufnehmen zu können, ist der Raspberry mit Gehäuse und Kamera an einer Gehäusehalterung fixiert (blau). Zur Visualisierung des Bildes und der entwickelten GUI wird ein Bildschirm (grün) eingesetzt. Die Bedienung der GUI erfolgt mithilfe einer Maus und Tastatur (lila). Um einen einheitlichen Hintergrund für das Objekt zu haben, wurde ein Papier auf den Tisch geklebt, auf dem das Objekt willkürlich platziert werden kann. Der Raspberry Pi 3 Model B besitzt 1 GB RAM und hat einen Prozessor mit 1,2 GHz Taktfrequenz [44]. Somit sind die verfügbaren Rechen-Ressourcen auf diesem Gerät stark begrenzt. Als Speicher dient eine 32 GB SD-Karte, womit der Speicher in diesem Demonstrator kein limitierender Faktor ist.

Zur Veranschaulichung des Demonstrators wird im Folgenden ein einfacher Test-Fall mit 3 Objekten genutzt. Die 3 Objekte sind ein 5€-Geldschein (Klasse: Geldschein), ein Kugelschreiber (Klasse: Stift) und ein HDMI-Kabel (Klasse: Kabel). Diese werden inkrementell dem Netzwerk gezeigt. Zunächst ist in Abbildung 39 der Aufbau der GUI nach dem Starten des Programms zu sehen.

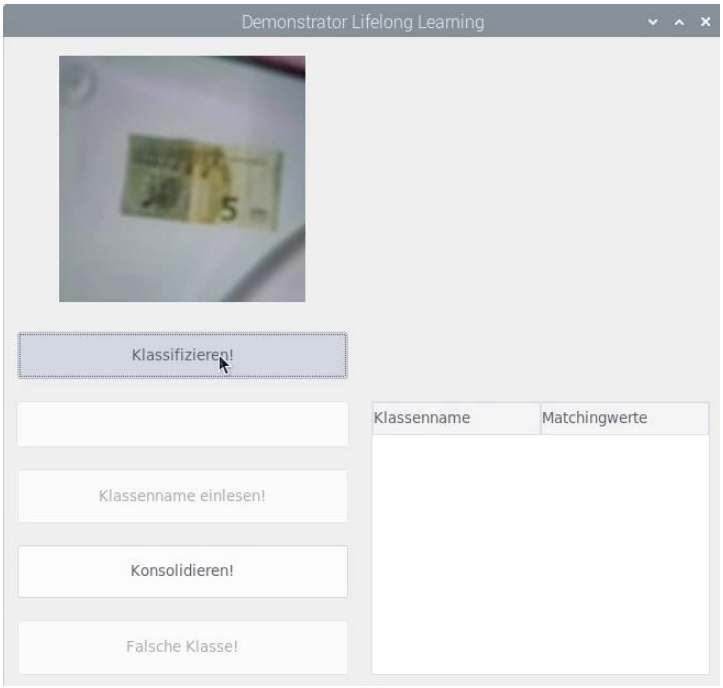

Abbildung 39: Aufbau der GUI

Im oberen linken Eck ist der Video-Stream der Kamera angezeigt. Mit dem „*Klassifizieren!*“-Button wird ein Bild aufgenommen und als Eingangsbild dem L DNN Algorithmus zur Verfügung gestellt. Zudem wird das aufgenommene Bild im rechten oberen Eck dargestellt. Da der L DNN Algorithmus bisher keine Klassen kennt wird die Klasse als „*Nothing I Know*“ gekennzeichnet. Der Benutzer erhält die Information und Aufforderung, dass die Klasse unbekannt ist und ein Klassenname eingegeben werden muss. Dies ist in Abbildung 40 zu sehen.

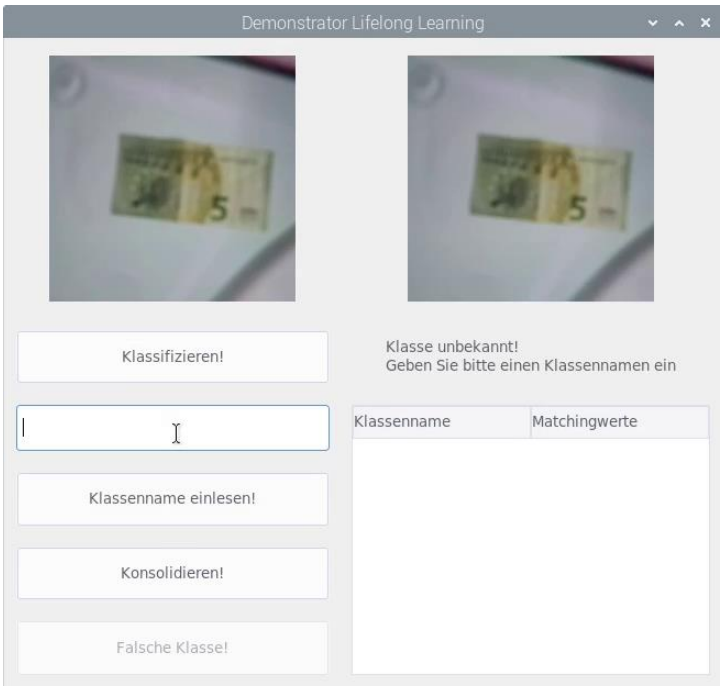

Abbildung 40: GUI nach der Klassifizierung des ersten Objekts

Über das Eingabefeld kann der Klassenname (hier: Geldschein) eingegeben werden und über den Button „*Klassenname einlesen!*“ wird dieser Name der Klasse zugeordnet. Die Klasse Geldschein ist nun dem L DNN Algorithmus bekannt und kann klassifiziert werden. Es können weitere Objekte dem Demonstrator gezeigt werden, wie z.B. ein Kugelschreiber (siehe Abbildung 41).

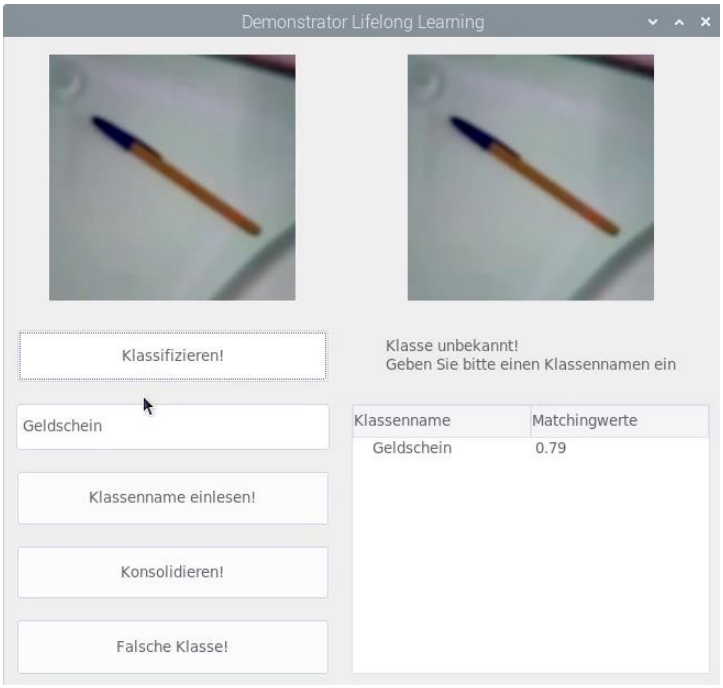

Abbildung 41: GUI bei neuem Objekt Kugelschreiber

Die Klasse Geldschein ist bereits bekannt und wird mit dem dazugehörigen Matchingwert auf der rechten Seite in einer Tabelle aufgeführt. Der L DNN Algorithmus

erkennt jedoch eine ihm unbekannte Klasse und fordert den Nutzer zur Eingabe des Klassennamens auf. Dies wiederholt sich auch für das HDMI-Kabel. Wenn nun die drei Klassen trainiert wurden, können die Objekte nochmals dem Algorithmus gezeigt und klassifiziert werden. Beispielhaft ist in Abbildung 42 der Geldschein als Eingangsbild zu sehen, nachdem die 3 Klassen trainiert wurden.

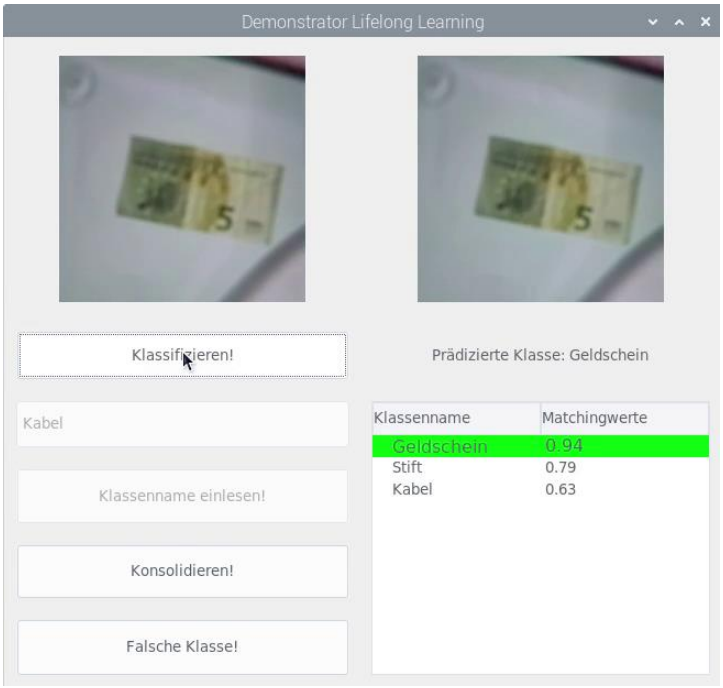

Abbildung 42: Klassifikation des Geldscheins nach dem Training der 3 Klassen

Unterhalb des Eingangsbildes wird die prädizierte Klasse ausgegeben. In der Tabelle wird zusätzlich die „Gewinner"-Klasse hervorgehoben. Diese drei Objekte wurden mit jeweils einem Bild trainiert. Der L DNN Algorithmus war in der Lage die drei Objekte daraufhin korrekt zu klassifizieren und jeweils auch eine neue bisher unbekannte Klasse mithilfe des „*Nothing I Know*"-Mechanismus zu erkennen.

Zusätzlich können mit dem „*Konsolidieren!*"-Button mehrere Repräsentationen einer Klasse konsolidiert werden. Mit dem Button „*Falsche Klasse!*" kann der L DNN Algorithmus bei einer fehlerhaften Prädiktion korrigiert und korrekt trainiert werden.

7 Fazit und Ausblick

In dieser Arbeit wurde der Lifelong Deep Neural Network (L DNN) Algorithmus hinsichtlich seiner Funktionalität und Anwendbarkeit auf andere Aufgabengebiete analysiert. Dafür wurde eine prototypische Implementierung des Algorithmus zur Evaluierung auf frei verfügbaren Bilddatensätzen umgesetzt.

Es wurden zunächst die notwendigen theoretischen Hintergründe in den Bereichen Deep Learning, Kontinuierliches Lernen, Verteiltes Lernen und Inkrementelle Klassifikatoren dargelegt. In diesen Grundlagen wurden unterschiedliche Probleme der einzelnen Bereiche sowie verschiedene Modelle und Algorithmen zur Lösung dieser Probleme aufgezeigt. Das Hauptproblem des kontinuierlichen Lernens ist dabei das katastrophale Vergessen, bei dem zuvor erlernte Aufgaben während des Trainings von neuen Aufgaben verlernt werden. Für die Lösung dieses Problems gibt es unterschiedliche Methoden. Beim verteilten Lernen muss im Rahmen dieser Arbeit darauf geachtet werden, dass kein Austausch von Rohdaten erfolgen soll, da z.B. sicherheitskritische Daten vorliegen.

Auf Basis dieser Grundlagen wurde der L DNN Algorithmus eingeführt. Die Ansätze des Algorithmus wurden dabei separat für das kontinuierliche und das verteilte Lernen untersucht und deren Vor- und Nachteile beschrieben. Zum Schluss wurden die einzelnen Ansätze in den Vergleich zu den zuvor eingeführten klassischen Algorithmen gesetzt. Dabei beruht der Ansatz des L DNN Algorithmus für das kontinuierliche Lernen auf der Dual-Memory Methode, bei der zwei komplementär lernende Systeme eingesetzt werden. Zum einen wird ein langsam lernendes (oder fixes) Modul zur Extraktion von generellen Informationen eingesetzt. Auf Basis dieser generellen Informationen erinnert sich ein schnell lernendes Modul an spezifische Informationen und prädiziert die Klasse des Eingangsbildes. Dieses Modul wird durch einen inkrementellen Klassifikator umgesetzt. Beim verteilten Lernen wird die Methode des Federated Learning angewendet, bei der die lokalen Modelle nur auf Basis der lokal vorhandenen Daten trainiert werden. Es werden lediglich Parameteränderungen zwischen verteilten Geräten kommuniziert. Die großen theoretischen Vorteile des L DNN Algorithmus liegen in dem Einsatz des kontinuierlichen und verteilten Lernens ohne den Austausch und der Speicherung von Rohdaten, wodurch der Schutz der Daten gewährleistet ist. Zudem soll der Algorithmus auch auf mobilen Endgeräten mit begrenztem Speicher- und Rechenbedarf lauf- und lernfähig sein.

Mit den eingeführten Modellen wurde der konkrete L DNN Algorithmus für die prototypische Implementierung konzipiert. Dafür wurden die konkreten Anforderungen an die beiden Module definiert, um auf Basis dieser Anforderungen die passenden Modelle auszuwählen. Modul A stellt das fixe Modul zur Feature-Extrahierung der Dual-Memory Methode dar. Es soll Features extrahieren, welche eine gute Klassifikation ermöglichen. Zudem soll es echtzeitfähig auf mobilen Endgeräten sein, wodurch ein geringer Speicher- und Rechenaufwand gewährleistet werden muss. Es wurden verschiedene DNN-Architekturen auf Basis ImageNet anhand der Merkmale

Parameteranzahl, Speicherbedarf für die Parameter, Anzahl an Operationen pro Eingangsbild und dem Klassifikationsfehler verglichen. Aufgrund seiner speziellen Architektur für mobile Endgeräte wurde MobileNet-V2 in dieser Arbeit als Modul A gewählt. Dieses Netzwerk bietet einen guten Trade-Off zwischen geringem Speicher- und Rechenaufwand und einer akzeptablen Genauigkeit und damit einer ausreichenden Güte der Features. Modul B ist der inkrementeller Klassifikator der Dual-Memory Methode. Er soll auf einer zufälligen Reihenfolge an Datensamples trainierbar sein und zu jedem Zeitpunkt einen funktionsfähigen Klassifikator für bereits bekannte Klassen stellen. Zudem soll auch hier der Rechen- und Speicherbedarf beschränkt sein. Es wurde ein FuzzyARTMAP-Netzwerk ausgewählt, das die gestellten Anforderungen erfüllen kann. Diese Netzwerke lösen durch ihre Architektur das Stabilität-Plastizität Dilemma, das für klassische neuronale Netzwerke ein Problem darstellt.

Die Evaluierung wurde anhand zwei verschiedener Bilddatensätzen (MNIST und ImageNet) durchgeführt. Als Evaluierungskriterien dienten die Klassifikationsgenauigkeit und der Speicherbedarf des L DNN Algorithmus. Für die Untersuchungen über den Einfluss von einzelnen Parametern, bei denen viele Tests durchgeführt wurden, wurden zehn zufällige Klassen vom ImageNet-Datensatz gezogen. Diese zehn Klassen stellten den ImageNet-10 Datensatz dar. Für die finale Untersuchung wurde der gesamte ImageNet-Datensatz genutzt.

Zunächst wurde eine Hyperparameter-Optimierung für Modul B, das Fuzzy ARTMAP-Netzwerk, mithilfe einer zweidimensionalen Gittersuche durchgeführt. Es wurden die Lernrate α und der Vigilance Parameter ρ jeweils im Bereich 0 bis 1 optimiert. Die Optimierung fand auf Validationsdaten für MNIST und ImageNet-10 statt. Für ρ wurde schließlich 0,5 als optimaler Wert ermittelt, während für α mit 0,2 die besten Ergebnisse erzielt wurden. Eine direkte Korrelation zwischen beiden Parametern konnte nicht festgestellt werden.

Zudem wurde der Einfluss der Anzahl an Trainingsbildern pro Klasse untersucht. Allgemein sinkt die Varianz der Klassifikationsgenauigkeit mit größerer Anzahl an Trainingsbildern und der Speicherbedarf von Modul B steigt. Die finale Klassifikationsgenauigkeit steigt für MNIST auch mit größerer Anzahl an Trainingsbildern. Bei ImageNet-10 wurde mit 100 Trainingsbildern pro Klasse eine bessere Genauigkeit als mit 500 und 1000 Trainingsbildern erzielt. Für die weiteren Untersuchungen wurden 500 Trainingsbilder pro Klasse für MNIST und 100 für ImageNet-10 gewählt.

Mit den optimierten Parametern wurde auf den bisher ungesehenen Testdaten die Genauigkeit für diese beiden Datensätze ermittelt. Für MNIST konnte mit 86,94% eine sehr gute Genauigkeit, auch im Vergleich zu bekannten Algorithmen aus der Literatur, erreicht werden. In der Literatur konnten lediglich Rehearsal Methoden, wie z.B. Deep Generative Replay, bessere Ergebnisse erzielen. Diese arbeiten mit der Wiederbenutzung von Trainingsdaten beim Trainieren einer neuen Aufgabe. Für ImageNet-10 konnte 76,4% Genauigkeit erreicht werden. Somit ist der Algorithmus

auch auf komplexeren Klassen und Eingangsdaten einsetzbar. Für das verteilte Lernen auf zwei Geräten konnten nahezu identische finale Genauigkeiten erzielt werden wie für das Lernen auf einem Gerät. Es wurde 86,56% für MNIST und 76,26% für ImageNet-10 erreicht. Mit diesem Algorithmus kann somit durch verteiltes Lernen die gleiche Performanz erreicht werden wie wenn die Daten auf einem zentralen Server liegen.

Da bei einer Vielzahl an Klassen der Speicherbedarf des FuzzyARTMAP stetig ansteigt, wurde untersucht wie sich die Performanz durch die Konsolidierung von Repräsentationen verhält. Zwei mögliche Methoden wurden im Vergleich zur vorherigen Performanz ohne Konsolidierung untersucht. Insgesamt kann der Speicherbedarf durch die beiden Konsolidierungsmethoden deutlich gesenkt werden, auf 1/18 (MNIST) und 1/9 (ImageNet-10) des ursprünglichen Speicherbedarfs. Mit der Konsolidierung nach jedem inkrementellen Schritt wurde die finale Genauigkeit lediglich leicht schlechter. Auf MNIST verschlechterte sich die Genauigkeit um drei Prozentpunkte, bei ImageNet-10 sogar nur um zwei Prozentpunkte. Mit der zweiten Methode, der Konsolidierung nach dem finalen Schritt, verschlechterte sich die Genauigkeit bei MNIST ebenfalls nur um drei Prozentpunkte, auf ImageNet-10 jedoch um sieben Prozentpunkte. Insgesamt kann durch die Konsolidierung deutlich Speicher gespart werden. Je nach Methode geht dies nur auf Kosten eines leichten Performanzverlustes.

Für die finale Untersuchung wurde der gesamte ImageNet-Datensatz genutzt. Sowohl mit nur einem inkrementellen Schritt (alle Klassen auf einmal), zehn inkrementellen Schritten als auch 20 inkrementellen Schritten erreicht der Algorithmus jeweils ca. 39% Genauigkeit. Der Algorithmus scheint somit stabil gegenüber der Anzahl an inkrementellen Schritten zu sein. Im Vergleich zu anderen Algorithmen der Dual-Memory Methode (iCaRL und LwF) mit 44% (iCaRL) und 39% (LwF) Genauigkeit kann eine ähnliche finale Performanz erreicht werden. Diese beiden Algorithmen nutzen jedoch einen leistungsstärkeren und rechenaufwändigeren Feature-Extrahierer (ResNet-Architektur) und führen pro inkrementellen Schritt 100 Epochen auf den Trainingsdaten durch (L DNN lediglich 1 Epoche pro Schritt).

Insgesamt sprechen die erzielten Ergebnisse für die Nutzung des L DNN Algorithmus in weiteren (realen) Anwendungen. Er erfüllt die grundlegenden Eigenschaften eines kontinuierlich lernenden Algorithmus und kann mit wenigen Trainingsdaten robust neue Klassen erlernen. Zudem verhält er sich gegenüber einer großen Anzahl an Klassen stabil und ist ebenfalls nicht sensitiv gegenüber der Anzahl an inkrementellen Trainingsschritten.

Zusätzlich wurde die prototypische Implementierung um eine Live-Bild-Klassifikation erweitert. Dabei wurde die entworfene Konzeption auf einem Raspberry Pi3 implementiert. Mithilfe einer entwickelten GUI kann diese Implementierung als Demonstrator zur Veranschaulichung des inkrementellen Klassen Lernens auf einem rechen- und speicherbegrenzten Endgerät genutzt werden.

Auf Basis dieser Arbeit können weitere interessante Untersuchung durchgeführt werden, die das Potenzial des Algorithmus weiter untersuchen. So wäre eine Evaluierung der Genauigkeit mit einem leistungsstärkeren Feature-Extrahierer als Modul A (z.B. eine ResNet-Architektur) interessant und ob damit z.B. auf ImageNet bessere Ergebnisse als vergleichbare Algorithmen (wie iCaRL) erzielt werden könnten. In dieser Arbeit konnten auch für das verteilte Lernen lediglich Testfälle mit zwei Geräten durchgeführt werden. Hier wäre eine Untersuchung mit mehreren Geräten interessant, um das Verhalten bei einer größeren Anzahl von Geräten zu beobachten. Weiterhin könnten einzelne Teile des Algorithmus weiter verfeinert werden. Zum Beispiel wird aktuell bei der Konsolidierung eine einfache Mittelwertbildung eingesetzt. Hier könnten Algorithmen entwickelt werden, die nur sehr ähnliche Repräsentationen zusammenfassen, da eine Klasse häufig nicht nur aus ähnlichen Repräsentationen besteht und dadurch mehr Robustheit bei komplexen Klassen erreicht werden könnte. Auch für das „Verschmelzen“ von Wissen bei verteilten Geräten könnte die Logik weiter verfeinert werden, da bisher lediglich die Repräsentationen aneinandergereiht werden. Möglich wäre hier eventuell das automatische Konsolidieren und Kombinieren, falls gleiche Klassen auf mehreren Geräten erlernt werden. Als großen und auch finalen Ausblick kann der Algorithmus auf weitere Anwendungsgebiete angewendet werden. Die prototypische Implementierung in dieser Arbeit wurde hinsichtlich der Objekterkennung auf Bildern evaluiert und hat gute Ergebnisse gezeigt. Interessant wäre der Einsatz in anderen Aufgabengebieten, wie z.B. Verschleißdaten eines Gegenstandes/einer Anlage für die Zustandsbestimmung beziehungsweise für eine prädiktive Ausfallvorhersage. Hier müsste für Modul A eine neue Methode ausgewählt werden, beziehungsweise auf Basis der Daten der Aufgabe ein eigener Feature-Extrahierer (z.B. AutoEncoder) trainiert werden. Zusätzlich könnten die gespeicherten und generalisierten Repräsentationen des Fuzzy-ARTMAP Netzwerks mithilfe von Transformationsalgorithmen (wie z.B. t-SNE [49], UMAP [50] oder PCA [51]) vom hochdimensionalen Raum (aktuell 1024 Dimensionen) in einen zwei- oder dreidimensionalen Raum transformiert werden. Dadurch könnten die gelernten und generalisierten Repräsentationen visualisiert und für Menschen in einer einfachen graphischen Darstellung nachvollziehbar dargestellt werden. Dadurch könnte die Güte der Feature-Extraktion untersucht werden, z.B. ob ähnliche Eingangsdaten (Bilder der gleichen Klasse) in einem ähnlichen Gebiet im Feature-Raum landen. Zudem könnten Algorithmen für die Konsolidierung und deren Einfluss in einem einfachen zweidimensionalen Raum veranschaulicht werden.

Literatur

[1] M. Luciw, S. Olivera, A. Gorshechnikov, J. Wurbs, H. M. Ames und M. Versace, „Systems and Methods to enable Continual, Memory-Bounded learning in Artificial Intelligence and Deep Learning Continuously operating Applications across networked Compute Edges“. United States of America Patent US 2018/0330238 A1, 15 November 2018.

[2] Neurala Inc., „Lifelong Deep Neural Networks - Tech Summary,“ [Online]. Available: https://info.neurala.com/hubfs/docs/Neurala_LifelongDNNWhitepaper.pdf. [Zugriff am 7 Mai 2019].

[3] Neurala Inc., „Neurala vs. Open Soruce,“ [Online]. Available: https://info.neurala.com/hubfs/docs/Open%20source%20vs.%20Neurala.pdf. [Zugriff am 7 Mai 2019].

[4] I. Goodfellow, Y. Bengio und A. Courville, Deep Learning, MIT Press, 2016.

[5] C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.

[6] M. A. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.

[7] A. Zang, Z. C. Lipton, M. Li und A. J. Smola, Dive into Deep Learning, 2019.

[8] G. E. Hinton, S. Osindero und Y.-W. Teh, „A fast learning algorithm for deep belief nets,“ *Neural Computation,* Bd. 18, Nr. 7, pp. 1527-1554, 2006.

[9] B. Yang, *Lecture Notes Deep Learning,* Stuttgart, 2018.

[10] D. Michie und D. J. Spiegelhalter, Machine Learning, Neural and Statistical Classification, Leeds: C. C. Taylor, 1994.

[11] Y. LeCun, L. Bottou, Y. Bengio und P. Haffner, „Gradient-based Learning applied to document recognition,“ *Proceedings of the IEEE,* pp. 2279-2324, November 1998.

[12] R. M. French, „Catastrophic forgetting in connectionists networks,“ *Trends in Cognitive Sciences,* pp. 128-135, April 1999.

[13] Y.-c. Hsu, Y.-c. Liu und Z. Kira, „Re-evaluating Continual Learning Scenarios : A Categorization and Case for Strong Baselines,“ in *32nd Conference on Neural Information Processing Systems (NIPS2018)*, Montréal, 2018.

[14] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan und S. Wermter, „Continual lifelong learning with neural networks: A review,“ *Neural Networks,* pp. 54-71, 2019.

[15] W. Abraham und A. Robins, „Memory retention - The synaptic stability versus plasticity dilemma,“ *Trends in neurosciences,* pp. 73-8, 03 2005.

[16] C. Kortge, „Episodic Memory in Connectionist Networks,“ *Proceedings of the 12th Annual Conference of Cognitive Science Society,* pp. 764-771, 01 Januar 1993.

[17] R. Kemker, M. McClure, A. Abitino, T. Hayes und C. Kanan, „Measuring Catastrophic Forgetting in Neural Networks,“ November 2017.

[18] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran und R. Hadsell, „Overcoming catastrophic forgetting in neural networks,“ *Proceedings of the National Academy of Sciences,* pp. 3521-3526, 2017.

[19] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel und D. Wierstra, „PathNet: Evolution Channels Gradient Descent in Super Neural Networks," 2017.

[20] A. Seff, A. Beatson, D. Suo und H. Liu, „Continual Learning in Generative Adversarial Nets," 23 Mai 2017.

[21] C. V. Nguyen, Y. Li, T. D. Bui und R. E. Turner, „Variational Continual Learning," 03 November 2017.

[22] H. Shin, J. K. Lee, J. Kim und J. Kim, „Continual Learning with Deep Generative Replay," *Proceedings of 31st Conference on Neural Information Processing Systems (NIPS 2017),* 2017.

[23] J. L. McClelland und B. L. McNaughton, „Why There Are Complementary Learning Systems in the Hippocampusand Neocortex: Insights From the Successes and Failures ofConnectionist Models of Learning and Memory," *Psychological Review,* pp. 419-457, 1995.

[24] S. A. Rebuffi, A. Kolesnikov, G. Sperl und C. H. Lampert, „iCaRL: Incremental classifier and representation learning," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017,* pp. 5533-5542, Januar 2017.

[25] G. A. Carpenter und S. Grossberg, „Adaptive Resonance Theory," in *The Handbook of Brain Theory and Neural Networks*, Boston, MIT Press, 2002.

[26] A. A.-m. Merten, „Adaptive Resonance Theory [ART] - Ein neuer Ansatz lernender Computer -," Universität Ulm, Ulm.

[27] V. Hedge und S. Usmani, „Parallel and Distributed Deep Learning," *Tch Report,* 2016.

[28] T. Ben-Nun und T. Hoefler, „Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," 2018.

[29] J. Konecny, B. H. McMahan, D. Ramage und P. Richtarik, „Federated Optimization: Distributed Machine Learning for On-Device Intelligence," pp. 1-38, 2016.

[30] Q. Yang, Y. Liu, T. Chen und Y. Tong, „Federated Machine Learning: Concept and Applications," Bd. 10, Nr. 2, pp. 1-19, 2019.

[31] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li und L. Fei-Fei, „ImageNet: A Large-Scale Hierarchical Image Database," *2009 IEEE Conference on Computer Vision and Pattern Recognition,* pp. 248-255, 2009.

[32] M. A. E. Muhammed, A. A. Ahmed und T. A. Khalid, „Benchmark Analysis of Popular ImageNet Classification Deep CNN Architectures," *Proceedings of the 2017 International Conference On Smart Technology for Smart Nation, SmartTechCon 2017,* pp. 902-907, 2018.

[33] A. Krizhevsky, I. Sutskever und H. G. E., „ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems 25,* pp. 1097-1105, 2012.

[34] K. Sinhal, „CV-Tricks," 2017. [Online]. Available: https://cv-tricks.com/cnn/understand-resnet-alexnet-vgg-inception/. [Zugriff am 5 Juni 2019].

[35] Google Brain Team, „TensorFlow Guide," Google Brain, [Online]. Available: https://www.tensorflow.org/guide/tensors. [Zugriff am 29 Mai 2019].

[36] K. Simonyan und A. Zisserman, „Very Deep Convolutional Networks For Large-Scale Image Recognition," in *International Conference on Learning Representations 2015*, San Diego, 2015.

[37] K. He, X. Zhang, S. Ren und J. Sun, „Deep residual learning for image recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* pp. 770-778, Dezember 2015.

[38] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke und A. Rabinovich, „Going deeper with convolutions," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* pp. 1-9, Juni 2015.

[39] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens und Z. Wojna, „Rethinking the Inception Architecture for Computer Vision," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* pp. 2818-2826, Dezember 2016.

[40] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto und H. Adam, „MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *CoRR,* 2017.

[41] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov und L. C. Chen, „MobileNetV2: Inverted Residuals and Linear Bottlenecks," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* pp. 4510-4520, 2018.

[42] M. Sandler und A. G. Howard, „Google AI Blog," Google AI, 3 April 2018. [Online]. Available: https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html. [Zugriff am 5 Juni 2019].

[43] C. Lampert, „Institue of Science and Technology Austria," 25 August 2018. [Online]. Available: https://pub.ist.ac.at/~chl/talks/lampert-vsssw2018.pdf. [Zugriff am 07 Juni 2019].

[44] Generation Robots, „Generation Robots," [Online]. Available: https://www.generationrobots.com/media/raspi3-datasheet.pdf. [Zugriff am 06 Juni 2019].

[45] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg und L. Fei-Fei, „ImageNet Large Scale Visual Recognition Challenge," 30 Januar 2015.

[46] G. M. van de Ven und A. S. Tolias, „Three continual learning scenarios and a case for generative replay," in *International Conference on Learning Representations*, New Orleans, 2019.

[47] Y. Wu, C. Yinpeng, L. Wang, Y. Ye, Z. Liu, G. Yandong und Y. Fu, „Large Scale Incremental Learning," *CoRR,* Mai 2019.

[48] Z. Li und D. Hoiem, „Learning without Forgetting," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* pp. 2935-2947, 2018.

[49] L. v. d. Maaten und G. Hinton, „Visualizing data using t-SNE," *Journal of machine learning research,* Bd. 9, pp. 2579-2605, 2008.

[50] L. McInnes, J. Healy und J. Melville, „Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426,* 2018.

[51] S. Wold, K. Esbensen und P. Geladi, „Principal component analysis," *Chemometrics and intelligent laboratory systems,* Bd. 2, pp. 37-52, 1987.

Erklärung

Ich erkläre, die Arbeit selbständig verfasst und bei der Erstellung dieser Arbeit die einschlägigen Bestimmungen, insbesondere zum Urheberrechtsschutz fremder Beiträge, eingehalten zu haben. Soweit meine Arbeit fremde Beiträge (z. B. Bilder, Zeichnungen, Textpassagen) enthält, erkläre ich, dass diese Beiträge als solche gekennzeichnet sind (z. B. Zitat, Quellenangabe) und ich eventuell erforderlich gewordene Zustimmungen der Urheber zur Nutzung dieser Beiträge in meiner Arbeit eingeholt habe.

Unterschrift:

Stuttgart, den 29.10.2019