|  |  |  |
| --- | --- | --- |
| **MA 3062** | | |
| Untersuchung und prototypische Umsetzung eines Lifelong Deep Neural Network Algorithmus | | |
| **Simon Kamm** | | |
|  | | |
| **Grundlagen** | | |
|  | Prüfer: | Prof. Dr.-Ing. Michael Weyrich |
|  | Betreuer: | Benjamin Maschler, M.Sc. |
| Start: 29.04.2019 | | Abgabe: 29.10.2019 |
|  | |  |

**Dokument Versionsverwaltung**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Version | Autor | QS | Datum | Status | Änderungen |
| 0.1 |  |  |  | in Bearb. | Erstellung |
|  |  |  |  |  |  |

*Allgemeine Hinweise zur Benutzung von IAS-VM Dokumentvorlagen*

*Diese Hinweise und Erläuterungen in den einzelnen Kapiteln der Vorlage sind kursiv dargestellt und müssen im Dokument gelöscht werden.*

*Verwendete Symbole:*

*\* generisch (kann mehrfach vorkommen)*

*[...] optional (kann auch weggelassen werden)*

*<...> Platzhalter (Bereich inklusive Klammern muss entsprechend ersetzt werden)*

***Bitte ändern Sie die Formatierung des Deckblatts nicht sondern tragen Sie nur die abgefragten Daten ein!***

*Bei der Erstellung eines Dokumentes hat das Dokument zunächst den Status ‘in Bearb.’ und die Versionsnummer 0.1. Dokumente im Status ‘in Bearb.’ können beliebig geändert, gespeichert und gedruckt werden. Wichtige Änderungen sollten allerdings im Feld „Änderungen“ dokumentiert werden. Die erste fertige Version bekommt die Versionsnummer 1.0. Ab der Version 1.0 wird das Speichern im Zustand ‘vorgelegt’ erlaubt. Dokumente mit niedrigerer Versionsnummer dürfen nicht mit ‘vorgelegt’ gespeichert werden.* ***Diese Informationen müssen manuell eingetragen werden.***

*Dokumente unterliegen ab dem Status ‘vorgelegt’ dem Konfigurationsmanagement und dürfen nicht mehr überschrieben werden. Nach einer Änderung muss das Dokument unter Angabe der durchgeführten Änderungen mit neuer Versionsnummer gespeichert werden. Die Versionsnummern muss dabei um 0.1 hochgezählt werden.*

*Bei der Erstellung eines Dokuments ist folgendes zu beachten:*

* *Wurden die Randbedingungen berücksichtigt?*
* *Wurden verschiedene Ansätze gegenübergestellt und Vor- und Nachteile abgewogen?*
* *Wird die getroffene Entscheidung akzeptiert?*

# Inhaltsverzeichnis

0 Inhaltsverzeichnis 3

1 Einleitung 4

2 Theoretische Grundlagen 5

2.1 Deep Learning 5

2.2 Kontinuierliches Lernen 11

2.3 Verteiltes Lernen 17

3 Lifelong Deep Neural Network Algorithmus 20

3.1 Beschreibung 20

3.2 Vorteile 20

3.3 Nachteile 20

3.4 Zusammenfassung 20

4 Vergleich der Ansätze 21

5 Literaturverzeichnis 22

# Einleitung

Daten schützen ist in heutigen Anwendungen ein wichtiger Aspekt, der nicht missachtet werden kann. Der Schutz von diesen 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 eingesetzt, um mögliche Ausfälle von Maschinen vorherzusagen und vorbeugende Instandhaltungsarbeiten zu ermöglichen die wiederum lange Ausfallzeiten verhindern. Dafür werden bisher 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 weiter zu lernen und können sich so auf abweichende Ereignisse reagieren, die vorher nicht bekannt waren und somit nicht erlernt wurden. Durch verteiltes Lernen könnten gleiche Maschinen sich zudem austauschen, wodurch die Information über einen Vorfall, 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 einen erheblichen Speicher- und Rechenleistung erfordert bei „real-time“ Anwendungen.

Sogenannte Lifelong Deep Neural Network Algorithmen (L DNN A) könnten das Potenzial haben, diesen Widerspruch aufzulösen, indem sie verteiltes 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.

Dafür werden in Kapitel 2 die theoretischen Grundlagen eingeführt, in welchem zunächst Grundlagen zu Deep Learning gegeben werden. Auf Basis dieser Grundlagen werden detaillierter die Themen kontinuierliches und verteiltes Lernen erläutert, da diese die Hauptaspekte dieser Arbeit sind. In Kapitel 3 wird anschließend der Lifelong Deep Neural Network Algorithmus vorgestellt, mit einer anschaulichen Beschreibung und Darstellung des Ansatzes sowie einer detaillierten Erläuterung und Aufteilung des Ansatzes. Innerhalb von Kapitel 4 wird dieser Ansatz mit aktuellen Ansätzen des kontinuierlichen und verteilen Lernen verglichen und die Unterschiede zu gängigen Ansätzen herausgestellt.

# Theoretische Grundlagen

In diesem Kapitel wird eine Übersicht über die theoretischen Grundlagen gegeben, welche im weiteren Verlauf der Arbeit notwendig sind. Zunächst wird Deep Learning generell eingeführt mit dem Augenmerk auf die kritischen Punkte. Darauf folgt eine detailliertere Einführung in die Themen kontinuierliches sowie verteiltes Lernen. Für eine grobe Einordnung kann gesagt werden, dass Deep Learning, kontinuierliches Lernen (auch „Continual Learning“) und verteiltes Lernen (auch „Distributed Learning“) generelle spezifische Themen aus dem Bereich maschinelles Lernen („Machine Learning“) sind. Abbildung 1 gibt eine graphische, beispielhafte Darstellung der Verhältnisse.

Abbildung 1: Verhältnis von verschiedenen Lernansätzen zu Machine Learning

## 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 maschinellen Lernen im allgemeine 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 für diese Probleme genannt. Im Rahmen dieser Arbeit werden die grundlegende Punkte zu Deep Learning genannt und aufgeführt, jedoch wird nicht auf jeden Punkt detailliert eingegangen, da das Hauptaugenmerk auf der späteren Untersuchung und Bewertung des L DNN A liegt. Für detailliertere Erklärungen und Ausführung wird auf die genannten Referenzen bei den einzelnen Punkten verwiesen.

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



Abbildung 2: Generelle Problemstellung für maschinelles Lernen

Das Lernen dieser Ablaufregel geschieht basierend auf sogenannten Trainingsdaten. Darum ist es offensichtlich, 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 wurde. Der Unterschied ist, dass in konventionellen Machine Learning Algorithmen die Features mithilfe einer vordefinierten Regel extrahiert werden [4]. Für diese Aufgabe existiert keine Theorie und Erfahrungen von Experten sind notwendig um gute und relevante Features für die folgende Aufgabe, z.B. Klassifikation, zu extrahieren. Die folgende Klassifikation wird von einem Klassifikator durchgeführt, wie beispielsweise *kNN* (k-nearest Neighbour) oder *SVM* (Support Vector Machine). In 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 passend *Loss* Funktion. Das effiziente Anpassen der Parameter kann mithilfe des *Backpropagation*-Algorithmus umgesetzt werden ( [4], [5], [6], [7]).

Die Lernstrategie von DNNs ist kopiert von der Art wie Menschen lernen zu sprechen, laufen, rechnen. Obwohl Deep Learning häufig als begeisternde 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, wo es als *Cybernetics* bekannt war. Zwischen 1980 und 1990 als *Connectionism* und das aktuelle Wiederaufleben seit 2006 unter 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, das ein spezielles neuronales Netzwerk, das sogenannte „Deep Belief Network“ effizient trainiert werden kann mithilfe der Strategie „greedy layer-wise pretraining“ [8]. Seit diesem Durchbruch stieg und steigt weiterhin die Anzahl der Anwendungen von DNNs deutlich an. Beispielhafte Anwendungen heutzutage für DNNs sind Empfehlungssystem (z.B. bei Amazon), automatische Spracherkennung, Text zu Sprache Übersetzung, Objekterkennung/-klassifizierung oder -segmentierung 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, da diese Netzwerke, auch als *Multilayer Perceptron* (MLP) bekannt, als Basis Deep Learning Module bezeichnet werden [4]. Der Name *feedforward* kommt von der Eigenschaft des Netzwerks, dass Information nur vorwärts (*forward*) durch das Netzwerk fließt, vom Eingangssignal durch das Netz zum Ausgangssignal (siehe auch Abbildung 2). Diese Netzwerke besitzen keine Feedback Verbindungen. Diese *Feedworward* 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.

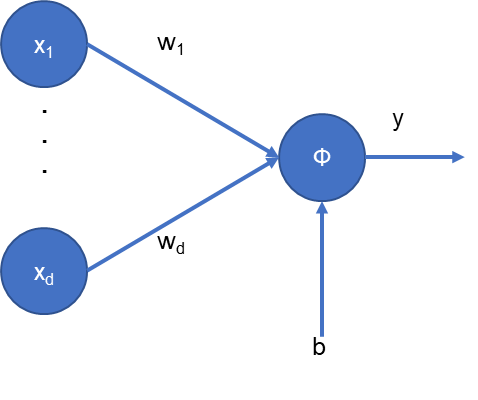


Abbildung 3: Einzelnes Neuron in einem *Feedforward Neural Network*

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

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

Der Ausgang eines einzelnen Neurons wird durch die (typischerweise nicht-lineare) Aktivierungsfunktion der Aktivierung beschrieben. Die Aktivierung wiederum ist eine affine Funktion des Eingangs . Das Eingangssignal kann ein Vektor, ein zweidimensionales Bild oder ein dreidimensionaler oder höherdimensionaler Tensor sein. Die trainierbaren und adaptierbaren Parameter des Neurons sind die Gewichte und der Bias .

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

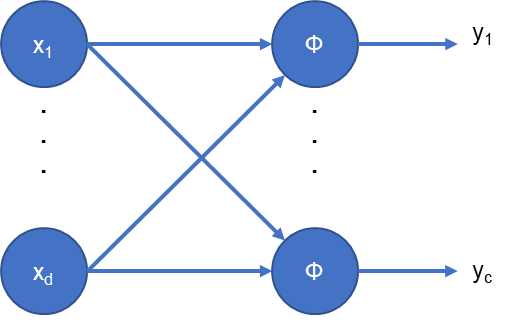


Abbildung 4: Neuronen-*Layer*

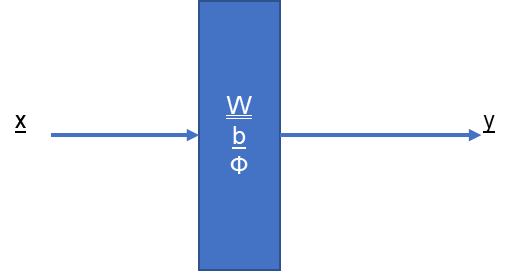


Abbildung 5: Neuronen-*Layer* in komprimierter Darstellung

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 mit jedem Eingang verbunden. Ein Netzwerk das nur aus solchen Schichten besteht wird F*ully Connected Network* (FCN) genannt. Der Nachteil dieser Netze sind die sehr große Anzahl an Paramatern, da jeder Verbindung eine Gewichtung benötigt. Diese große Anzahl an Parametern resultiert in einer sehr hohen Komplexität bei der Berechnung und einem hohen Speicherbedarf. Zudem werden aufgrund der Verbindungen in diesen Netzwerken keine lokalen Verhalten/Features 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. Diese CNN bestehen hauptsächlich aus *convolutional Layer*. Aufgrund der Eigenschaften von diesen Schichten mit ihren *sparse* 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 verschiedener Architekturen zusammengeführt werden. Die resultierende Architektur wird schließlich als Neuronales Netzwerk bezeichnet. Die finale Länge dieser Kette von Schichten wird auch als tiefe des Modells bezeichnet. Von dieser Terminologie entstand der Name „Deep Learning“. Die erste und letzte Schicht wird als Eingangs- (*Input Layer*) und Ausgangsschicht (*Output Layer*) bezeichnet. Diese Schichten haben definierte Größen, da der Eingang und der Ausgang des Netzwerks ebenfalls definierte Größen haben und damit die Dimensionen dieser Schichten definieren. Alle Schichten zwischen diesen beiden werden als versteckte (*hidden*) Schichten bezeichnet, weil diese Schichten weder von der Eingangs- noch von der Ausgangsseite ersichtlich sind. Im Gegensatz zur Ein- und Ausgangsschicht ist die Größe dieser Schichten nicht fest vorgegeben durch die Daten. Typischerweise formen diese Schichten einen Flaschenhals, welcher das Netzwerk zwingt ein einfaches Modell des Systems zu erstellen. Dieses erlernte Modell soll in der Lage sein generelle Verhalten der Daten zu erlernen um auch auf neuen, bisher unbekannten Daten (Test Daten) die gewünschte Aufgabe durchzuführen [10].

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

|  |  |  |
| --- | --- | --- |
|  |  | (3) |
|  |  | (4) |

Falls lineare Aktivierungsfunktionen genutzt werden ist das gesamte Netzwerk, unabhängig von der Tiefe des Netzes, insgesamt nur eine lineare Transformation der Eingangsdaten. Damit können lediglich linear-lösbare Probleme gelöst werden. Dies zeigt die Bedeutung von nicht-linearen 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 minimiert bezüglich den Netzwerkparametern. Allgemein ist das Ziel des Trainings die Kostenfunktion auf Basis der verfügbaren Trainingsdaten zu minimieren, wie beschrieben in Formel (**5**).

|  |  |  |
| --- | --- | --- |
|  |  | (5) |

Dabei kann eine willkürliche *loss*-Funktion sein mit ist die gewünschte Ausgabe des Netzwerks (Label) und ist der Eingang des Netzwerks der Probe . Die Kostenfunktion ist der gemittelte Wert der *loss*-Funktion über alle Trainingsdaten. Die Wahl der *loss*-Funktion ist abhängig von Problemstellung, die gelöst werden soll. Für Regressionsprobleme ist der l2-*Loss* eine typische Funktion, bei dem die l2 Norm des Fehlers als Optimierungskriterium genutzt wird. Der Fehler (in der Literatur *error* genannt) ist definiert als Unterschied zwischen dem gewünschten Ausgang und dem tatsächlichen Ausgang des neuronalen Netzwerks, welcher generell mit der nicht-linearen Funktion beschrieben werden kann, das den willkürlichen Zusammenhang zwischen Eingang und Ausgang darstellt. Der genannte l2-*Loss* ist formuliert in Gleichung (**6**).

|  |  |  |
| --- | --- | --- |
|  |  | (6) |

Für die Klassifikation ist eine typische *loss*-Funktion der kategorische *loss*. Typischerweise besitzt die letzte Schicht bei einer Klassifikationsaufgabe eine Softmax-Aktivierungsfunktion, weshalb dort der finale Ausgang des neuronalen Netzwerks die Wahrscheinlichkeiten für die Klassenzugehörigkeit darstellt.

|  |  |  |
| --- | --- | --- |
|  |  | (7) |

Das Ziel des gesamten Trainingsprozesse ist es, dass minima zu finden, welches im Allgemeinen keine geschlossene Lösung besitzt. Deshalb werden numerische Optimierungsverfahren im Bereich Deep Learning eingesetzt. Der meist verwendete Algorithmus zur Minimierung der Kostenfunktion in Deep Learning ist der *Gradient Descent* (GD) Algorithmus. In diesem Algorithmus werden die Netzwerkparameter so angepasst, dass ein kleiner Schritt in Richtung des negativen Gradienten getan wird [5]. Der Anpassungsschritt des GD-Algorithmus kann mit Formel (**8**) beschrieben werden.

|  |  |  |
| --- | --- | --- |
|  |  | (8) |

Dabei ist der Iterationsindex und ist die Schrittweite (oder auch Lernrate genannt) die den Schritt in Richtung des negativen Gradienten Vektors bestimmt. Zu Beginn werden die Netzwerkparameter initialisiert, was eine initiale Schätzung 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 der *Stochastic Gradient Descent* (SGD) Algorithmus genutzt, welcher die Berechnungsaufwände für jede Iteration reduziert, da der Gradienten Vektor und das Update für einen sogenannten Minibatch berechnet wird. Der stochastische Gradienten Vektor ist die unverfälschte Schätzung des Gradienten Vektors . Diese Schätzung führt zu einem verzerrten Gradienten, was wiederum zu dem Namen **stochastischer** GD führt [7].

Die Berechnung des Gradienten Vektors ist elementar um die Parameter anpassen zu können. Für die Berechnung des Gradienten Vektors müssen die partiellen Ableitungen und der Kostenfunktion nach den Netzwerkparametern und berechnet werden. Dies kann mithilfe des *Error Backpropagation* Algorithmus effizient umgesetzt werden (dieser Algorithmus wird häufig auch nur *backpropagation* oder *backprop* genannt). Wie der Name bereits sagt, werden dabei die Error-Vektoren durch das Netzwerk zurück propagiert. Dabei wird das Netzwerk rückwärts durchpropagiert, beginnend bei Ausgangsschicht . Der beispielhafte Error-Vektor dieser Schicht für die Kostenfunktion abgleitet nach den Gewichten ist in Formel (**9**) gegeben.

|  |  |  |
| --- | --- | --- |
|  |  | (9) |

Dabei ist der Ausgang und die Aktivierung von Schicht . Die partiellen Ableitungen nach folgen denselben Gleichungen, bis auf dem Austausch von durch .

Für die restlichen Schichten kann der Error-Vektor mithilfe der Kettenregel für Differenzierung berechnet werden. Der Error-Vektor der Schicht kann mithilfe Gleichung **(10)** berechnet werden [9].

|  |  |  |
| --- | --- | --- |
|  |  | (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 dann wiederum notwendig sind um die Error-Vektoren rückwärts durch das Netzwerk propagieren zu können.

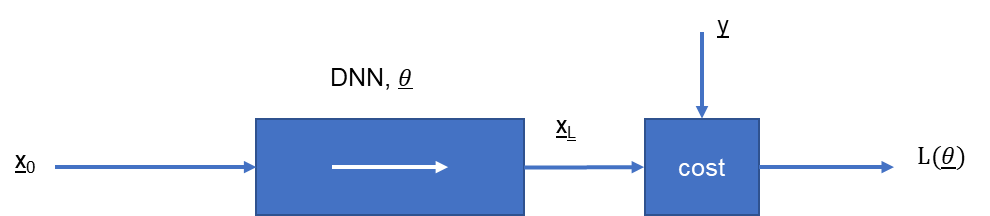


Abbildung 6: Vorwärts-Pfad durch ein Netzwerk

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.

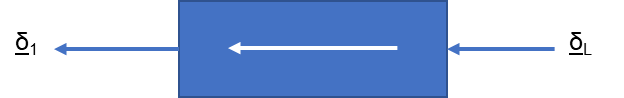


Abbildung 7: Error Backpropagation durch ein Netzwerk

*Backpropagating* durch ein Netzwerk gibt final den Error-Vektor des gesamten Netzwerks aus. Mit der gewonnen Information können, wie bereits beschrieben, die Netzwerkeparameter 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:

1. Gebe die Eingangsdaten in das Netzwerk ein und propagiere wie in Abbildung 6 vorwärts durch das Netzwerk um alle Aktivierungen der versteckten und Ausgangs Neuronen zu berechnen
2. Berechne den Error-Vektor für alle Ausgangsneuronen mithilfe Formel **(9)**
3. Propagiere die Error-Vektoren rückwärts durch das Netzwerk (Formel **(10)**) um die Error-Vektoren für alle versteckten Neuronen im Netzwerk zu erhalten (siehe Abbildung 7)
4. Wende eine numerische Optimierungsmethode (z.B. SGD) an um die Netzwerkparameter anzupassen

Die genannten Schritten 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 Optimierungsprobleme leiden. In Tabelle 1 werden einige Schwierigkeiten für die Optimierung eines neuronalen Netzwerks kurz zusammengefasst sowie mögliche Lösungen genannt [9].

Tabelle 1: Optimierungsprobleme in neuronalen Netzwerken

|  |  |
| --- | --- |
| Optimierungsproblem | Lösung |
| Stochastischer Gradient | Größere Minibatches, Momentum |
| *ill conditioned* | Momentum, Skalierung des Eingangs, Batch Normalisierung |
| Sattelpunkt/Plateau | Verrauschter Gradient |
| Sensitivität zur Lernrate | Lernraten Plan |
| Lokales Minimum | Parameter Initialisierung |
| Verschwindender (vanishing) Gradient | Parameter Initialisierung, Verbessertes Modell |

Wie in [7] beschrieben, bietet die Optimierung eine Möglichkeit die Kostenfunktion zu minimieren. Aber generell sind die Ziele der Optimierung und von Deep Learning unterschiedliche. In der reinen Optimierung ist es das Ziel das Minimum einer (Tranings-) Kostenfunktion zu finden. Im Gegensatz dazu liegt der Fokus in Deep Learning darauf, den Generalisierung-Fehler zu minimieren, welcher der Wert der Kostenfunktion auf Basis neuer, bisher nicht gekannter Eingangsdaten ist. Deshalb ist es wichtig dem Netzwerk regelmäßig Validationsdaten einzuspielen während dem Training, um zu überprüfen ob es eine Überanpassung (*overfitting*) des Netzwerks auf die Trainingsdaten gibt. Überanpassung im Kontext von Deep Learning bedeutet, dass der Error auf Basis der Trainingsdaten sehr gering ist im Vergleich zu dem Generalisierungs-/Test-Error. Es gibt wiederum einige verschiedene Methoden, um einer Überanpassung vorzubeugen, wie z.B. Dropout oder Regularisierung ( [4], [5], [6], [7]).

## Kontinuierliches Lernen

Menschen und auch Tiere haben die große Fähigkeit kontinuierliche neues Wissen und neue Fähigkeiten zu erlernen. Diese Fähigkeit wird in der Literatur als lebenslanges (*lifelong*) oder kontinuierliches (*continual*) Lernen bezeichnet.

In diesem Kapitel werden zunächst Schwierigkeiten beim kontinuierlichen Lernen aufgeführt und der Grund für diese auftretenden Schwierigkeiten erläutert. Darauffolgend werden unterschiedliche Methoden vorgestellt, welche die Probleme verringern bzw. lösen sollen. Schließlich wird der Überbegriff des kontinuierlichen Lernens in unterschiedliche konkrete Anwendungsbereiche unterteilt, um konkrete Aufgaben definieren zu können.

Kontinuierliches Lernen kann generell als eine besondere Form des Deep Learning gesehen, bei der dieselben Architekturen (DNNs) genutzt werden, jedoch aufgrund spezieller Probleme teilweise andere Algorithmen im Einsatz sind. Ein wichtiger Punkt dabei ist, dass durch das Erlernen neuen Wissens altes, bereits erlerntes Wissen nicht verloren geht. Dieses Vergessen bereits erlernter Dinge durch das hinzufügen neuen Wissens wird im Bereich Deep Learning häufig 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 einem Echtzeit System, die eine typische Anwendung von kontinuierlichem Lernen ist, unbedingt 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, Ansätze und Metriken, die in diesem Kapitel beleuchtet werden.

Aktuelle DNNs, welche für viele Anwendungen genutzt werden, benutzen wie in Kapitel 2.1 Gradienten-basierte Methoden (z.B. SGD). Wenn ein DNN mit solch einer Methodik inkrementell angepasst wird, erliegen diese Netze dem Problem des katastrophalen Vergessens ([12], [13], [14]). Der Grund dafür ist, dass diese Gradienten-basierten Methoden die Netzwerkparameter entsprechend den aktuellen Error-Vektoren anpassen, welche lediglich abhängig sind von den Eingangsdaten des aktuellen Minibatches. Der Grund für *catastrophic forgetting* ist bekannt unter dem Stabilität-Plastizität Dilemma [15]. Das Modell benötigt ausreichend Plastizität (Verformbarkeit) um neue Aufgaben zu erlernen, aber große Parameteränderungen bewirken das Vergessen vorher erlernter Aufgaben. Wenn die Netzwerk-Parameter stabil gehalten werden, werden vorher erlernte Aufgaben nicht vergessen, jedoch verhindert eine zu große Stabilität das erlernen neuer Aufgaben.

Wenn nun ein Netzwerk beispielhaft für eine Objektklasse „Hund“ trainiert wurde, und im weiteren Verlauf nur noch Katzen angezeigt werden, wird die Objektklasse „Katze“ erlernt während die bereits erlernte Klasse „Hund“ verlernt wird.

Intuitiv kann als Lösung für dieses Problem gefordert werden, dass die Anpassungsregel so begrenzt wird, das 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 von einer neuen Aufgabe B nachdem Aufgabe A erlernt wurde.

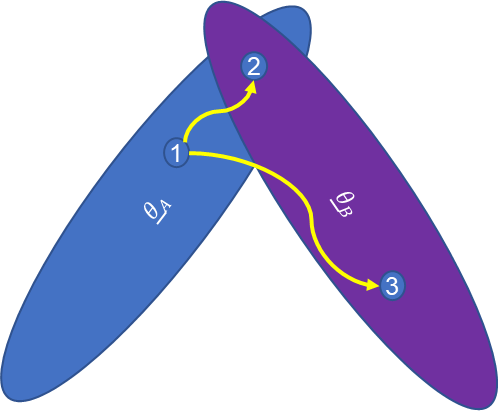


Abbildung 8: Erlernen neuer Aufgabe B und mögliche Folgen

Dabei stellt die blaue Ellipsoide einen Lösungsraum mit geringem Fehler für die Aufgabe A dar mit der erlernten und gefundenen Lösung in Punkt 1. Wenn nun die Aufgabe B gelernt 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 und und kann somit beide Aufgaben mit geringem Fehler lösen. Wenn jedoch Punkt 3 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 überlappenden Bereiche zwischen verschiedenen Aufgaben geben kann.

Um *Catastrophic Forgetting* zu vermeiden oder den Einfluss zu minimieren, gibt es unterschiedliche Ansätze, die genutzt werden. Eine der ersten Ansätze kam bereits 1993 von Kortge [16], der den Grund für das Vergessen beim Backpropagation Algorithmus sah. Dafür entwickelte er eine Variation des Backpropagation Algorithmus. Die Idee dahinter ist, dass nur die aktiven Neuronen angepasst werden während dem Training, 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 katastrophalen Vergessens ist die Reduzierung der internen überlappenden Verteilungen, da in diesem Ansatz die Überlappung der einzelnen internen Verteilung für verschiedene Muster als Grund für das *Catastrophic Forgetting* gesehen wurden [12]. Als Lösung dafür werden Semi-Distributed Repräsentation eingeführt. Die Reduzierung der repräsentativen Überlappung wird durch die Einführung von *sparse* Vektoren erzielt. Diese *sparse* Vektoren bedeuten, dass nur einige wenige Neuronen aktiv sind für eine Repräsentation eines speziellen Musters, was automatisch die Überlappung reduziert zu anderen Mustern. 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 Neuronen, welche am aktivsten sind, 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.

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 verschiedenen Ansätze stehen.

Nach [17], kann zwischen fünf unterschiedlichen Ansätzen zur Vermeidung des C*atastrophic Forgetting* differenziert werden. Diese fünf Ansätze werden im Folgenden kurz vorgestellt, um ein Verständnis dafür zu erhalten, in welchen Richtungen Maßnahmen durchgeführt werden können, um *Catastrophic Forgetting* zu vermeiden oder zu reduzieren.

Regularisierungsmethoden

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

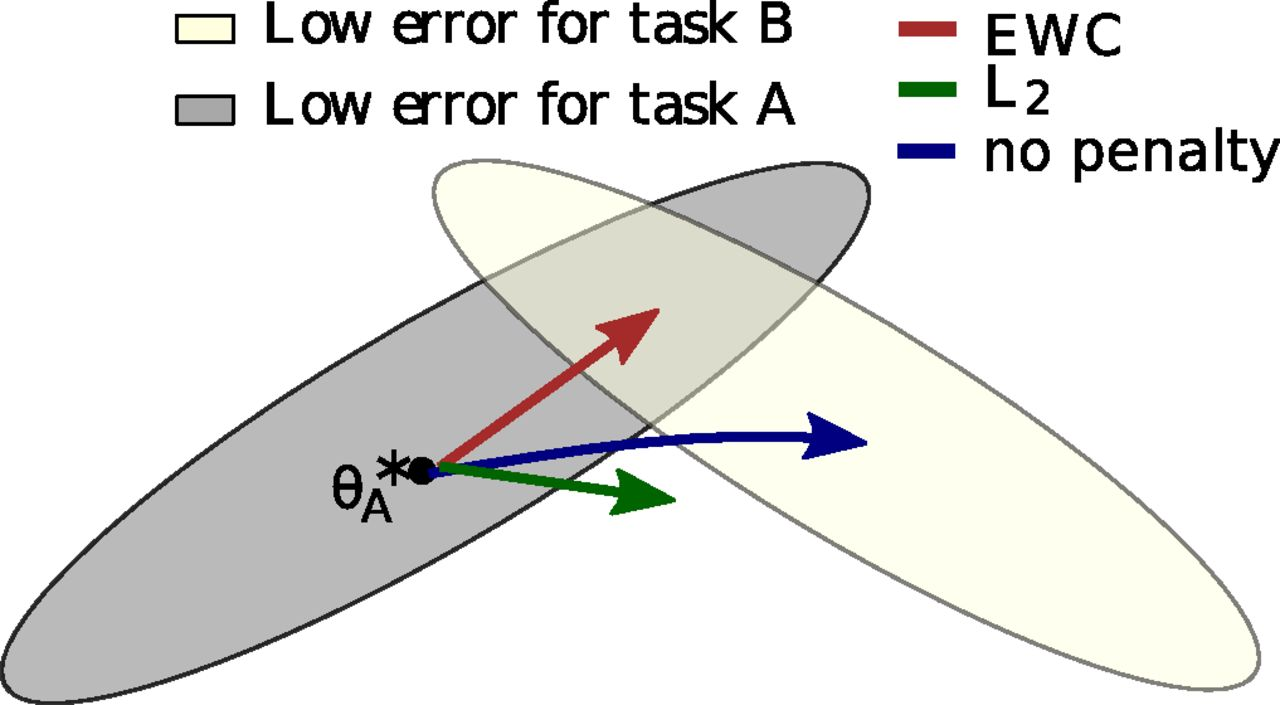


Abbildung 9: Einfluss von EWC auf Parameteranpassungen [18]

Dabei wird wieder eine einfache Darstellung des Parameterraums gewählt. Wenn keine Regularisierung gewählt wird, wird durch das erlernen der neuen Aufgabe B die alte Aufgabe A verlernt (blauer Pfeil). Wenn alle Parameter gleich und zu stark gewichtet werden, kann die neue Aufgabe B nicht korrekt gelernt werden aufgrund der geringen Anpassbarkeit der Parameter (grüner Pfeil). Mithilfe von EWC kann schließlich eine Lösung für die Aufgabe B gefunden werden ohne ein erhebliches Vergessen von Aufgabe A (roter Pfeil). Die gesamte Funktion welche minimiert werden soll in EWC für den dargestellten Anwendungsfall ist in Gleichung **(11)** gegeben.

|  |  |  |
| --- | --- | --- |
|  |  | (11) |

Dabei ist die Kostenfunktion für Aufgabe B und gibt die Gewichtung der Regularisierung an, und damit wie wichtig die alte Aufgabe im Vergleich zur neuen ist. ist der Index um die Parameter zu bestimmen und ist die Fisher Information für jeden Parameter, welche angibt wie wichtig dieser Parameter zur Darstellung von Aufgabe A ist. sind schließlich die erlernten Parameter der idealen 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 zusammengesetzt 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 bezügliche des Speicherbedarfs, da mit steigender Anzahl an Aufgaben der Speicherbedarf ansteigt, um neue Klassifikatoren trainieren zu können. Neuere Ansätze begrenzen die Modelgröße mithilfe verschiedener Ansätze um den Speicherbedarf zu limitieren. Der bekannteste Algorithmus dieser Methoden ist aktuell der sogenannte Pathnet-Ansatz [19]. Bei diesem Ansatz werden Agenten in einem neuronalen Netzwerk eingesetzt, welche die Teile des Netzwerks identifizieren welche wiederverwendet werden können für eine neue Aufgabe. Die relevanten Pfade für die vorherige Aufgabe werden eingefroren, um *Catstrophic Forgetting* zu vermeiden [14].

Rehearsal Methoden

Diese Methoden nutzen Daten von vorhergehenden Aufgaben und fügen diese dem Trainingsprozess für die neue Aufgabe zu. Dies erfordert einen hohen Speicherbedarf, um die Trainingsdaten vorheriger Aufgaben zur Verfügung zu haben. Diese Methode wurde bereits bei einigen frühen Ansätzen genutzt, und es lassen sich gute Ergebnisse damit erzielen. Neuere Ansätze mit dieser Methode nutzen verschiedene Ansätze, um eine sinnvolle Auswahl oder Komprimierung der „alten“ Trainingsdaten zu ermöglichen, damit wenige relevante Daten gespeichert werden. Beispielsweise werden generative Modelle wie ein Variational Autoencoder oder Generative Adversarial Networks (GAN) trainiert, welche aus komprimierten Darstellungen pseudo-reale Eingangsdaten erstellen können [20], [21]. Zusammengefasst werden diese Methoden unter dem Namen *Deep Generative Replay* [22].

Dual-Memory Methoden

Die Grundlage für die Dual-Memory Methoden liegen in den *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. Dieses Zusammenspiel eines langsam lernenden Netzes und einem schnell auffassenden Netz wird in vielen Dual-Memory Methoden genutzt. Generell 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* Repräsentation einer Aufgabe zu erzeugen, und damit ausreichend Parameter für das Erlernen einer neuen Aufgabe frei verfügbar zu haben.

Nachdem verschiedene Methoden vorgestellt wurde, die das *Catastrophic Forgetting* verhindern können und sollen, können auch die Anwendungen des kontinuierlichen Lernens in unterschiedliche Kategorien unterteilt werden, um eine sinnvolle Vergleichbarkeit und Bewertung zu ermöglichen. Nach Hsu et al. [13] können Anwendungen des kontinuierlichen Lernens in drei Gebiete unterteilt werden: *Incremental* ***Task*** *Learning, Incremental* ***Domain*** *Learning und Incremental* ***Class*** *Learning*. Diese unterschiedlichen Szenarios werden im Folgenden eingeführt und der Unterschied zwischen den einzelnen Szenarios herausgestellt. Beispielhaft werden dafür zwei Aufgaben A und B angenommen, mit den Verteilungen der Eingangsdaten und sowie den dazugehörigen Labels und und den jeweiligen Verteilungen und . Abbildung 10 stellt die drei unterschiedlichen Szenarien am Beispiel des Split MNIST Datensatzes dar. In den gepunkteten Rechtecken wird der Eingang für das Training dargestellt, mit () für (Eingangsbild, Zielausgang, Aufgaben-ID).

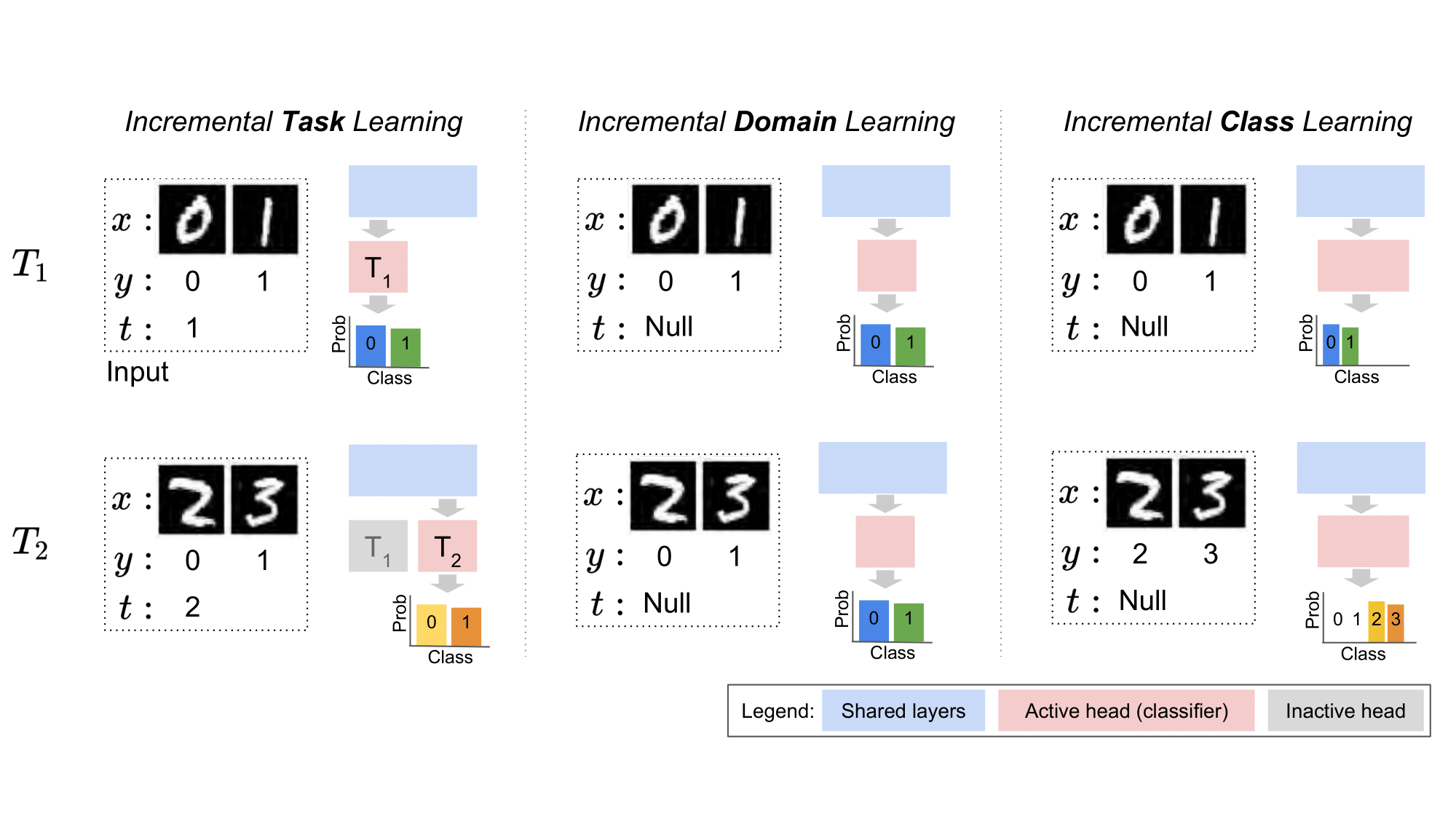


Abbildung 10: Darstellung der drei Continual Learning Szenarien am Beispiel von Split MNIST [13]

Incremental Task Learning

In diesen Szenarios sind die Ausgänge unterschiedliche zwischen den Aufgaben A und B, somit . Dadurch sind direkt auch die Verteilungen der Ausgänge nicht identisch mit . Durch die Aufgabenstellung des kontinuierlichen Lernens gilt allgemein da eine neue Aufgabe inkrementell erlernt werden soll. Die Ausgänge unterscheiden sich in ihrer Dimension und semantische Bedeutung. Beispielhaft kann die erste Aufgabe eine Klassifizierung zwischen 5 Klassen sein, während die zweite Aufgabe eine Regression eines einzelnen Wertes ist. Es wird eine komplett neue Aufgabe erlernt dabei (Regression statt Klassifikation). Um die aufgabenabhängige korrekte Ausgabe zu ermöglichen, sind aufgabenabhängige Ausgangskomponenten ( und in der linken unteren Graphik von Abbildung 10) notwendig, die abhängig von der Aufgaben-ID ausgewählt werden. Dafür ist bei diesen Szenarien zusätzlich die Aufgaben-ID notwendiger Input für das Netzwerk [13].

Incremental Domain Learning

Beim inkrementellen Domain Learning variieren die Eingangsdaten, und damit . Das Netzwerk wird nicht angepasst zu der neuen Ausgabe, wodurch resultiert das die Ausgabe des Netzwerks identisch bleibt mit und typischerweise für ausgeglichene Datensätze auch . Am Beispiel von Split MNIST mit der in Abbildung 10 dargestellten Ausgabe (binärer Klassifikator) können damit gerade von ungeraden Zahlen unterschieden werden. Es werden dadurch durch neue Aufgaben neue Bereiche erlernt [13].

Incremental Class Learning

In diesem Szenario werden inkrementell exklusiv neue Klassen erlernt. Aufgrund der Vielzahl an Klassen gilt deswegen und aufgrund der grundlegenden Eigenschaften neuer Aufgabe auch . In diesem Aufbau wird angenommen unter der Annahme, dass die gesamte Anzahl an Klassen bekannt ist und zu Beginn die Dimension der Ausgabe auf die Anzahl an Klassen über alle Aufgaben gesetzt wird. Auch in diesem Fall behält das Netzwerk über alle Aufgaben hinweg seine Architektur und besitzt keine aufgabenabhängigen Stellen [13].

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*“ lernende Algorithmen vorgestellt.

## Verteiltes Lernen

Verteiltes Lernen wird in der Literatur häufig unter den Namen *Distributed* oder *Parallel* *Learning* beschrieben. Es gibt viele unterschiedliche Gründe, warum verteiltes Lernen genutzt wird und interessant ist, abhängig vom Anwendungsfall. Beispielsweise kann es aufgrund der Größe des Netzwerkes notwendig sein, das Modell auf mehrere Prozessoren zu verteilen. Auch kann es aufgrund der enormen Trainingszeiten von DNNs bei großen Datenmengen gewünscht sein, paralleles, verteiltes Training durchzuführen mit aufgeteilten Datensätzen und die trainierten Modelle nach dem Training (in der Literatur *post-training* genannt) wieder zusammenzuführen. Ein anderer Anwendungsfall kann schließlich das Sammeln von riesigen Datenmengen auf gleichen, verteilten Geräten (z.B. Smartphones). Aufgrund von begrenzten Speichern oder Datenschutzrichtlinien können die Daten nicht auf einen zentralen Server geladen werden, wo ein zentralisiertes Training stattfindet. Deshalb kann es notwendig sein, auf den jeweiligen Endgeräten verteilt zu lernen, und lediglich Parameteränderungen auf einem Server zu sammeln, um am Ende ein besseres Netzwerk trainiert zu haben.

In diesem Kapitel werden unterschiedliche Anwendungsgebiete, sowie die dazugehörigen Ansätze und Methoden des verteilten und parallelen Lernens eingeführt. Aufgrund der Vielzahl an unterschiedlichen Methoden werden nur ausgewählte, im Rahmen dieser Arbeit relevante, Methoden vorgestellt.

Ursprünglich kommt der Wunsch nach verteiltem und parallelen Lernen von dem Hintergrund der Trainingszeiten von komplexen neuronalen Netzwerken. Komplexe Netzwerke, die auf großen Datensätzen trainiert werden, können Tage bis Wochen auf einzelnen Prozessoren benötigen, um die Parametrisierung solcher Netze zu erlernen. Durch die Weiterentwicklung und Nutzung von *Graphical Processing Units* (GPU) 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 auch Modelle, zu groß sind um auf dem jeweiligen Gerät gespeichert zu werden.

Dabei kann generell 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. Als 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 oder nicht erwünscht, den gesamten Datensatz oder das gesamte Modell auf einem einzelnen Gerät, sondern auf mehreren Geräten verteilt zu speichern.

Für eine feinere Aufteilung kann zwischen der Gleichzeitigkeit in Netzwerken allgemein und der Gleichzeitigkeit im Training von diesen Netzwerken unterschieden werden. Für die Gleichzeitigkeit in Netzwerken kann weiter zwischen Daten-Parallelisierung und Modell-Parallelisierung unterschieden werden, wobei auch unterschiedliche Begriffe dafür in der Literatur benutzt 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 typischerweise angewendet, wenn das Modell speichertechnisch nicht auf einem Gerät gespeichert werden kann. Dabei können dann beispielsweise 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 [24].

In [25] wird weiter die Gleichzeitigkeit im Training von diesen 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 des Netzes. Nachfolgend existiert generell mehr als eine Instanz der Netzwerk-Parameter. Die Ansätze können dabei in drei Kategorie unterteilt werden: Modell-Übereinstimmung, Parameter-Verteilung und Training-Verteilung. Zu jeder Kategorie gibt es verschiedene Methoden, von denen einige beispielhaft in Tabelle 2 zusammengefasst werden [25]. Nachfolgend wird auf einzelne Methoden spezifischer eingegangen.

Tabelle : 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 |

Modell-Übereinstimmung

In den Methoden, welche in der Kategorie Modell-Übereinstimmung zusammengefasst werden, werden die Berechnungen der Update-Schritte parallel auf unterschiedlichen Knoten ausgeführt. Diese Methoden können als eine spezielle Form der Daten-Parallelisierung angesehen werden. Aktuelle Parameter werden als konsistentes Modell angesehen, und diese aktuellen Parameter können auf einem Parameter Server (PS) oder dezentralisiert auf unterschiedlichen Knoten liegen. Bei synchronen Methoden wird senden alle Knoten zum gleichen Zeitpunkt ihre entsprechend berechneten Änderungen, welche zentral zu einem neuen konsistenten Modell zusammengefasst werden. Dieses Modell wird wieder verteilt und die unterschiedlichen Knoten können den nächsten Zyklus 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.

Parameter-Verteilung und Kommunikation

Eine zentralisierte Netzwerkarchitektur beinhaltet in der Regel eine (geteilte) PS-Infrastruktur. Mit dieser Infrastruktur senden die einzelnen Knoten ihre Änderungen, was im Fall eines DNN der errechnete Gradient ist. Die eintreffenden Gradienten werden von dem zentralen PS benutzt, um die neuen Parameterwerte zu berechnen. Es gibt in diesem Szenario somit nur einen zentralen Optimierer. Die neu optimierten Parameter werden dann 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. Bei diesen Ansätzen können theoretisch die unterschiedlichen Knoten untereinander kommunizieren und Parameter austauschen und/oder mitteln.

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

Training-Verteilung

In diesen Ansätzen finden nur selten und unregelmäßig Parameterupdates beziehungsweise der Austausch von Parametern statt. Bei diesen Ansätzen wird auf unterschiedlichen Knoten Kopien der Parameter angelegt, und die durch das Training resultierenden 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 dieselben Ergebnisse wie mit größeren Ensembles erzielt werden [25].

Als zusätzliche, spezifische Methode wird verbündetes Lernen (*federated* Learning) vorgestellt. Diese Methode wurde 2016 in [26] vorgestellt. Das Ziel dieser Methode ist ein hochqualitatives, zentralisiertes Modell auf Basis eines über viele Knoten verteilter Netzwerke zu trainieren. Dabei liegen die Daten ungleichmäßig verteilt über diese Knoten vor. Die lokalen Knoten werden dabei als Rechnerknoten benutzt, die auf Basis der lokalen Daten Optimierungen durchführen, um ein globales Modell zu verbessern. Diese lokalen Daten müssen bei dieser Methode nicht auf einem zentralen Server gespeichert werden, sondern liegen nur auf den lokalen Knoten vor. Mithilfe diesem Aufbaus müssen mögliche private, sicherheitskritische Daten nicht auf einen Server geladen werden, was die Reduzierung des Sicherheitsrisikos zur Folge hat. Zudem soll mit dieser Methode der Kommunikationsaufwand zwischen den Knoten und zwischen einem zentralen Server (globales Modell) und den Knoten minimiert werden. In dem Ansatz des verbündeten Lernens werden lediglich Anpassungen an den zentralen Server geschickt (z.B. Gradienten-Vektor), wodurch die benötigte Kommunikationsbandbreite im Vergleich zu den kompletten Trainingsdaten drastische reduziert wird, und zudem die geschickte Information deutlich abstrahierter von den möglicherweise personalisierten Daten ist. *Federated Learning* kann mit folgenden Notationen definiert werden: Es gibt Besitzer von Knoten (z.B. Smartphones), mit den auf den persönlichen Knoten gesammelten Daten . In klassischen Ansätzen würden die Daten zusammengelegt mit um ein zentrales Modell zu trainieren. Ein föderiertes System ist ein lernender Prozess in dem die einzelnen Knoten jeweils ein eigenes Modell trainieren. Dabei werden die Daten der einzelnen Knoten nicht mit den anderen Knoten geteilt. Zusätzlich soll die Genauigkeit der einzelnen Modelle, beschrieben durch , annähernd die Genauigkeit des hypothetischen zentralen Modells , , erreichen. Mathematisch kann das mithilfe der nicht-negative reale Zahl in Gleichung **(12)** beschrieben werden:

|  |  |  |
| --- | --- | --- |
|  |  | (12) |

Ein föderiertes System hat somit einen -Genauigkeit Verlust. Föderierte Systeme kann weiter in drei Anwendungsfälle kategorisiert werden, die in Tabelle 3 dargestellt werden [27].

Tabelle : 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 für die unterschiedlichen Kategorien konkrete Beispiele gegeben:

*Horizontales Federated Learning*

Zwei regionale Banken mit unterschiedlichen Benutzergruppen aus ihren jeweiligen Regionen mit geringer (keiner) Überschneidung. 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 unterschiedliche. Die Bank speichert zum Beispiel das monatlich einkommende Gehalt und das Kreditranking, während der Internetshop Browserverläufe und Einkaufsverhalten abspeichert. Durch das Verbünden beider auf Basis der Features trainierten Netzwerke kann auf Basis der Kundendaten das Einkaufsverhalten einzelner Gruppen bestimmt werden

# Lifelong Deep Neural Network Algorithmus

*Je nach Themenstellung der Arbeit werden an dieser Stelle Literaturstellen, Spezi-fikationen, Publikationen, etc. aufgeführt und deren Vor- und Nachteile diskutiert.*

## Beschreibung

## Vorteile

## Nachteile

## Zusammenfassung

*Kurze Bewertung des Ansatzes.*

# Vergleich der Ansätze

*Hier werden, so vorhanden, die verschiedenen Ansätze verglichen.*

# Literaturverzeichnis

|  |  |
| --- | --- |
| [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] | V. Hedge und S. Usmani, „Parallel and Distributed Deep Learning,“ *Tch Report,* 2016. |
| [25] | T. Ben-Nun und T. Hoefler, „Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis,“ 2018. |
| [26] | F. R. M., „Using semi-distributed representations to overcome castrophic forgetting in connectionist networks.,“ *Proceedings of the Thirteenth Annual Cognitive Science Society Conference,* pp. 173-178, 1991. |
| [27] | J. Konecny, B. H. McMahan, D. Ramage und P. Richtarik, „Federated Optimization: Distributed Machine Learning for On-Device Intelligence,“ pp. 1-38, 2016. |
| [28] | Q. Yang, Y. Liu, T. Chen und Y. Tong, „Federated Machine Learning: Concept and Applications,“ Bd. 10, Nr. 2, pp. 1-19, 2019. |