|  |  |  |
| --- | --- | --- |
| **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 |
| 1.0 | Kamm | Ms | 24.05.19 | Vorgelegt |  |

# Inhaltsverzeichnis

0 Inhaltsverzeichnis 2

1 Einleitung 3

2 Theoretische Grundlagen 5

2.1 Deep Learning 5

2.2 Kontinuierliches Lernen 12

2.3 Inkrementelle Klassifikatoren 17

2.4 Verteiltes Lernen 20

3 Lifelong Deep Neural Network Algorithmus 24

3.1 Beschreibung 24

3.1.1 Kontinuierliches Lernen 27

3.1.2 Verteiltes Lernen 27

3.2 Vorteile 28

3.3 Nachteile 29

3.4 Zusammenfassung und Vergleich zu klassischen Ansätzen 30

4 Literaturverzeichnis 31

# Einleitung

Danteschutz 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 wie von einer Produktionsanlage, muss der Schutz dieser Daten gewährleistet werden. 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 Algorithmen eingesetzt, um mögliche Ausfälle von Maschinen vorherzusagen und vorbeugende Instandhaltungsarbeiten zu ermöglichen die wiederum lange und teure 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 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 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. Zudem kann es auch schlicht verboten bzw. unerwünscht sein, gesammelte Daten von Maschine A and Maschine B weiterzugeben, da diese eventuell 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 nicht erwünscht vom Anwender, dass diese persönlichen Daten auf einem zentralen Server gespeichert werden um dort ein neuronales Netzwerk zu trainieren.

Sogenannte Lifelong Deep Neural Network Algorithmen (L DNN A) könnten 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. In Kapitel 0 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 0 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. Diese einzelnen Bereiche haben einen hohen Überschneidungsgrad, da z.B. für das *Continual Learning* eine Vielzahl von Ansätzen des *Deep Learning* genutzt wird. Dennoch hat jeder Bereich seine eigenen spezifischen Probleme und unterschiedliche Methoden, um diese zu lösen. In dieser Arbeit werden aus allen Bereichen Methoden und Komponenten miteinander genutzt.

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 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 genannt. 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 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 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 und Erfahrungen von Experten sind notwendig um gute und relevante Features für die folgende Aufgabe, z.B. die Klassifikation, zu extrahieren. Die folgende Klassifikation wird von einem separaten 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 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, laufen oder rechnen. Es wird anhand von Beispielen das Verhalten soweit angepasst, dass das gewünschte Ergebnis erzielt werden kann. 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 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, dass 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 auch 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 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, da diese Netzwerke, auch als *Multilayer Perceptron* (MLP) bekannt, als Basis Modul innerhalb von Deep Learning 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. *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.

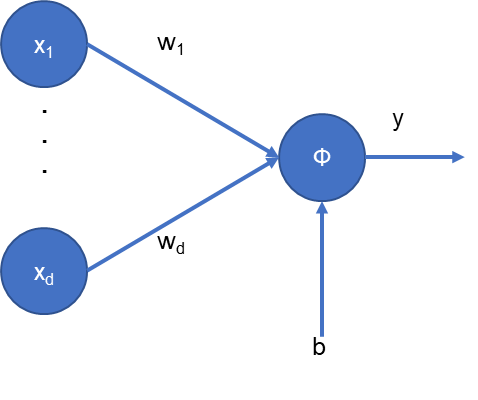


Abbildung 3: Einzelnes Neuron in einem *Feedforward Neural Network*

Diese graphische Darstellung kann auch mathematisch formuliert werden. Die mathematische Gleichung 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 drei- oder höherdimensionaler Tensor sein. Die trainierbaren und adaptierbaren Parameter des Neurons sind die Gewichte und der Bias .

Eine Schicht (*Layer*) des Netzwerks, 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.

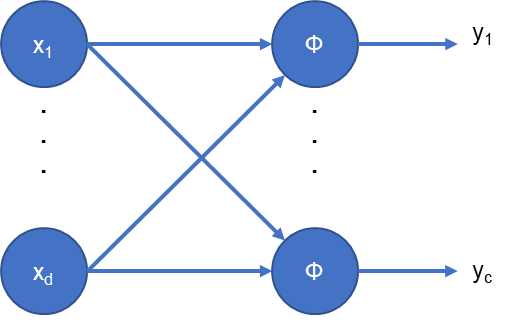


Abbildung 4: Neuronen-*Layer*

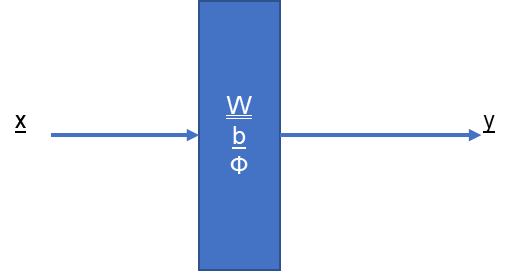


Abbildung 5: Neuronen-*Layer* in komprimierter Darstellung

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, dass nur aus solchen Schichten besteht wird F*ully Connected Network* (FCN) genannt. Der Nachteil dieser Netze sind 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 werden 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 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 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 ebenfalls definierte Größen haben und damit die Dimensionen dieser Schichten bestimmen. Alle Schichten zwischen diesen beiden sind die versteckten (*hidden*) Schichten, 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-lineare 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*).

|  |  |  |
| --- | --- | --- |
|  |  | (3) |
|  |  | (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 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 der gewünschten Ausgabe des Netzwerks (Label) und dem Eingang des Netzwerks des Samples . 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 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 der kategorische *loss* eine typische *loss*-Funktion. Diese Funktion ist in Gleichung **(8)** gegeben. Die letzte Schicht bei einer Klassifikationsaufgabe besitzt in der Regel eine Softmax-Aktivierungsfunktion, weshalb dort der finale Ausgang des neuronalen Netzwerks () die Wahrscheinlichkeiten für die Klassenzugehörigkeit darstellt.

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

Das Ziel des gesamten Trainingsprozess ist es, dass Minimum 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.

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

Dabei ist der Iterationsindex und die Schrittweite (oder auch Lernrate genannt), die Länge des Schrittes in Richtung des negativen Gradienten Vektors bestimmt. Zu Beginn müssen die Netzwerkparameter initialisiert werden, 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 in der Praxis der *Stochastic Gradient Descent* (SGD) Algorithmus genutzt, welcher die Berechnungsaufwände für jede Iteration reduziert, da der Gradienten Vektor und das Update lediglich für einen sogenannten Minibatch berechnet werden. Der stochastische Gradienten Vektor ist die 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 der Ausgangsschicht 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 die 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.

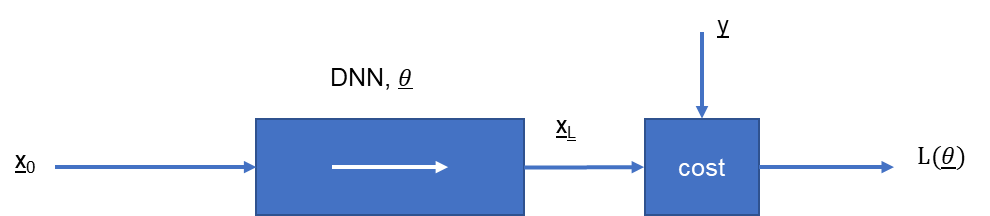


Abbildung 6: Vorwärts-Pfad durch ein Netzwerk

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.

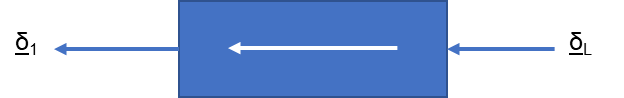


Abbildung 7: Error Backpropagation durch ein Netzwerk

*Backpropagation* 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 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 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. 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 Basis neuer, bisher nicht gekannter, 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. 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 Regularisation ( [4], [5], [6], [7]).

## Kontinuierliches Lernen

Menschen und auch Tiere haben die Fähigkeit kontinuierlich sich 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 geht. Dieses Vergessen durch das Hinzufügen neuen Wissens wird in diesem Kontext 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 muss in Echtzeit-Systemen, die eine typische Anwendung von kontinuierlichem Lernen sind, 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 und Ansätze zum Lösen dieser, die in diesem Kapitel beleuchtet werden.

Aktuelle DNNs, welche für viele Anwendungen genutzt werden, benutzen, wie in Kapitel 2.1 beschrieben, 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 als Stabilität-Plastizität Dilemm [15] a. 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“ höchstwahrscheinlich 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.

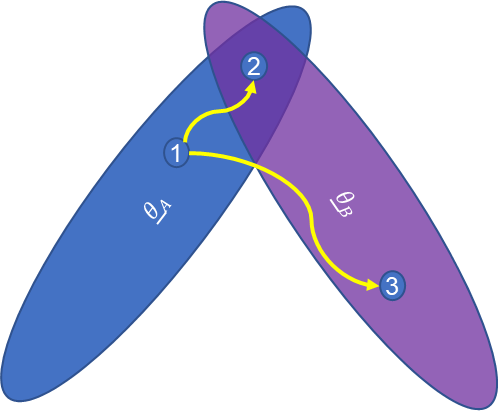


Abbildung 8: Erlernen neuer Aufgabe B und mögliche Folgen

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 Aufgabe B 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 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 von 1993 [16] sieht den Grund für das Vergessen beim Backpropagation Algorithmus. Dafür entwickelte er eine Variation des Backpropagation Algorithmus. 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 Repräsentation eingeführt. Die Reduzierung der repräsentativen Überlappung wird durch die Einführung von *sparsen* Vektoren erzielt. *Sparse* Vektoren bedeuten, dass nur einige wenige Neuronen aktiv sind für die Repräsentation eines speziellen Musters, 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 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 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 Ansätzen zur Vermeidung des C*atastrophic Forgetting* differenziert werden. Diese fünf Ansätze werden im Folgenden kurz vorgestellt.

Regularisierungsmethoden

Regularisierungsmethoden fügen generell Beschränkungen zu den Parameterupdates hinzu. Beispielhaft ist eine -Regularisierung, 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]. Es wird eine Bedingung zur der *Loss*-Funktion hinzugefügt, welche Verformbarkeit von den Parametern nimmt, die am relevantesten für die zuvor gelernte Aufgabe sind. Das Verhalten des EWC-Algorithmus kann graphisch in Abbildung 9 dargestellt werden.

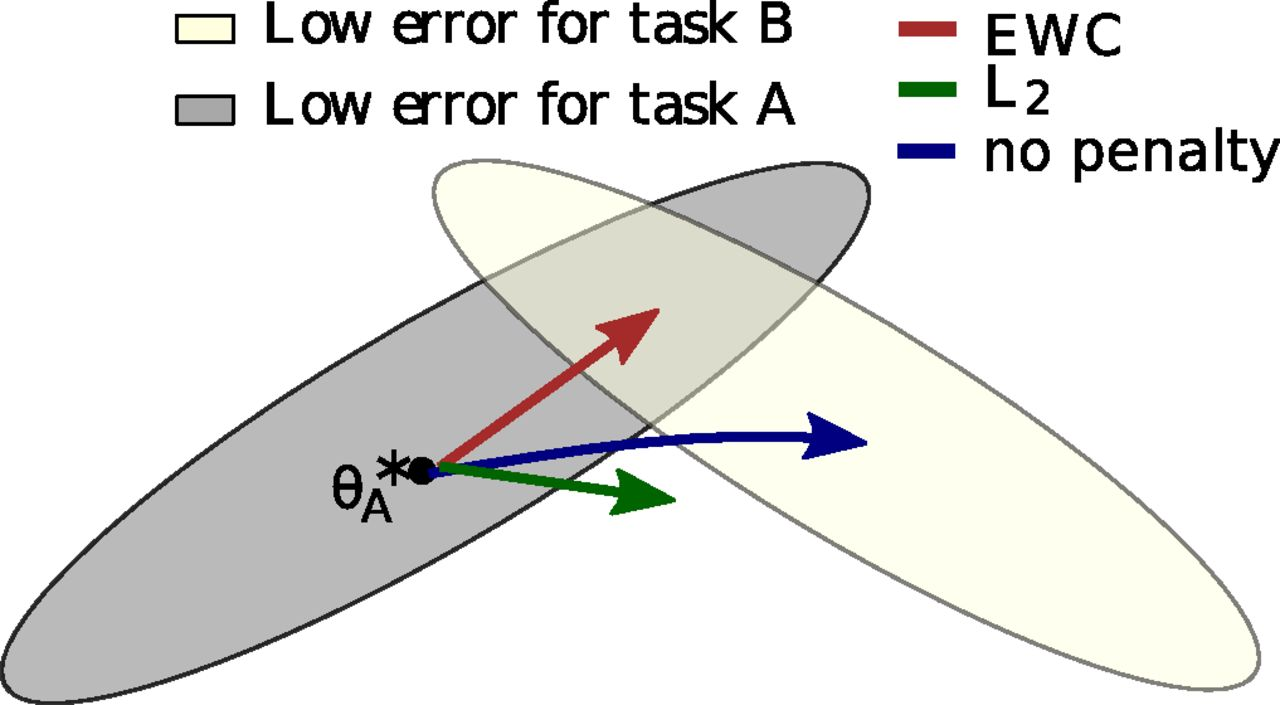


Abbildung 9: Einfluss von EWC auf Parameteranpassungen [18]

Dabei wird wieder eine einfache Darstellung des Parameterraums gewählt. Wenn keine Regularisierung gewählt wird, 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 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 Vergessen von Aufgabe A (roter Pfeil). Die Funktion , welche im EWC-Algorithmus minimiert werden soll, ist in Gleichung **(11)** gegeben.

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

Dabei ist die Kostenfunktion für Aufgabe B und gibt die Gewichtung der Regularisierung an. Über diese Gewichtung wird angegeben, 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 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 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 *Catstrophic Forgetting* zu vermeiden [14].

Rehearsal Methoden

Diese 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 haben. Diese Methoden wurden bereits bei frühen Ansätzen genutzt, und es lassen sich gute Ergebnisse damit 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 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* [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. 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* Repräsentation 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 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 definiert und der Unterschied zwischen den einzelnen Szenarios herausgestellt. Beispielhaft werden dafür zwei Aufgaben A und B angenommen, mit den Verteilungen der Eingangsdaten und , 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).

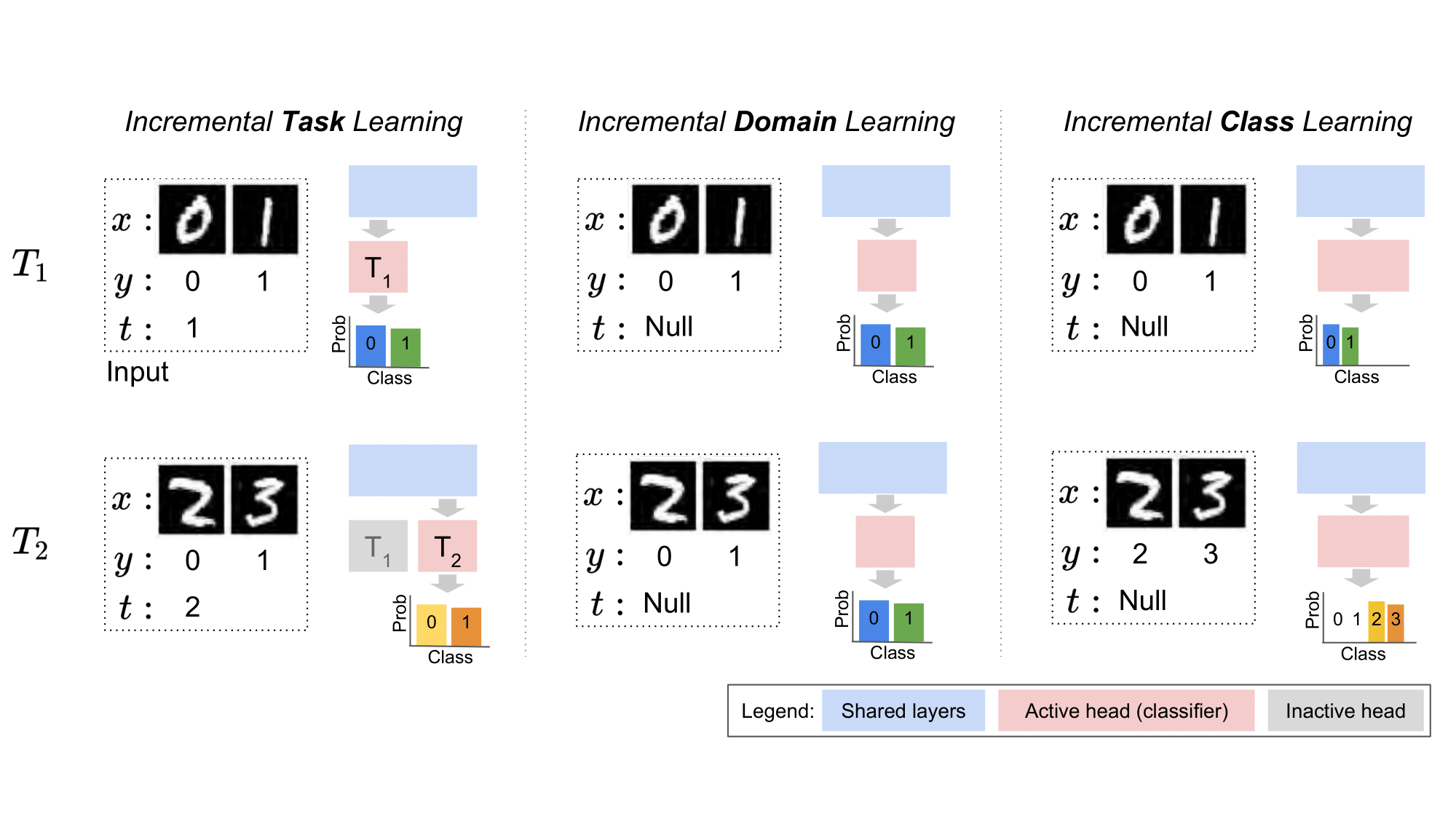


Abbildung 10: Darstellung der drei Continual Learning Szenarien am Beispiel von Split MNIST [13]

Incremental Task Learning

In diesen Szenarios haben die Aufgaben A und B unterschiedliche Ausgänge, . Daraus ergibt sich eine verschiedene Verteilung der Ausgänge, . Durch die Aufgabenstellung des kontinuierlichen Lernens gilt , 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 die Regression eines einzelnen Wertes ist. Es wird eine komplett neue Aufgabe erlernt (Regression statt Klassifikation). Um die 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 ein notwendiger Input für das Netzwerk [13].

Incremental Domain Learning

Beim inkrementellen Domain Learning variieren die Eingangsdaten, und damit . Das gesamte Netzwerk wird nicht angepasst, weshalb die Ausgabe des Netzwerks identisch bleibt mit und für ausgeglichene Datensätze auch gitl. Am Beispiel von Split MNIST mit der in Abbildung 10 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 exklusiv neue Klassen erlernt. Aufgrund der Vielzahl an Klassen gilt und aufgrund der grundlegenden Eigenschaften neuer Aufgaben 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]. 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 mit.

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*“ lernende Algorithmen vorgestellt.

## Inkrementelle Klassifikatoren

Im Rahmen dieser Arbeit liegt der untersuchte Anwendungsfall im Bereich des inkrementellen Klassen Lernens (*Incremental Class Learning*, siehe 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:

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

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 priorisierte Exemplar-Auswahl und *Representation Learning* mithilfe von Wissens-Destillierung und prototypischen Samples. Bei diesem Algorithmus werden Klassen-Prototypen/-Repräsentationen angelegt für die unterschiedlichen Klassen. 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)**.

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

Dabei ist das prädizierte Label eines Inputs auf Basis der Feature-Extraktion . Der Prototypen-Vektor für jede bisher bekannte Klasse ist definiert durch Gleichung **(13)**.

|  |  |  |
| --- | --- | --- |
|  |  | (13) |

Dabei sind die Prototypen der einzelnen Klassen, definiert in der Menge . Die Anzahl an relevanten Prototypen kann dabei beliebig gewählt werden. Durch die direkte Verknüpfung der Feature-Extraktion in die Berechnung der exemplarischen Repräsentation ist sichergestellt, dass sich bei ändernder Feature-Extraktionsregel 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 Routine 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 alten Klassen weiterhin richtig klassifiziert werden sollen. Dieser Schritt wird *Representation Learning* genannt, da auch der Feature-Extraktor und damit die Repräsentation der Eingangsdaten im Netzwerk angepasst wird. Der Algorithmus kann auch ohne *Representation Learning* genutzt werden. Es werden dann nur die Prototypen-Vektoren angepasst beziehungsweise neue Vektoren hinzugefügt, wenn neue Klassen hinzukommen.

Mit diesem Ansatz werden auf Test-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 Resonanz Theorie (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 den 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. Wenn Eingangsdaten eintreffen, werden diese 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 bleibt ohne Veränderung dieselbe. Das ist speziell für interessant, wenn eine neue Klasse mit wenigen Samples erlernt wurde. Diese Klasse kann auch nach der „Repräsentation“ im Netzwerk weiter inkrementell trainiert und damit verbessert werden. Falls der Schwellwerts 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 zur Veranschaulichung des beschriebenen Ablaufs ist in Abbildung 11 gegeben.

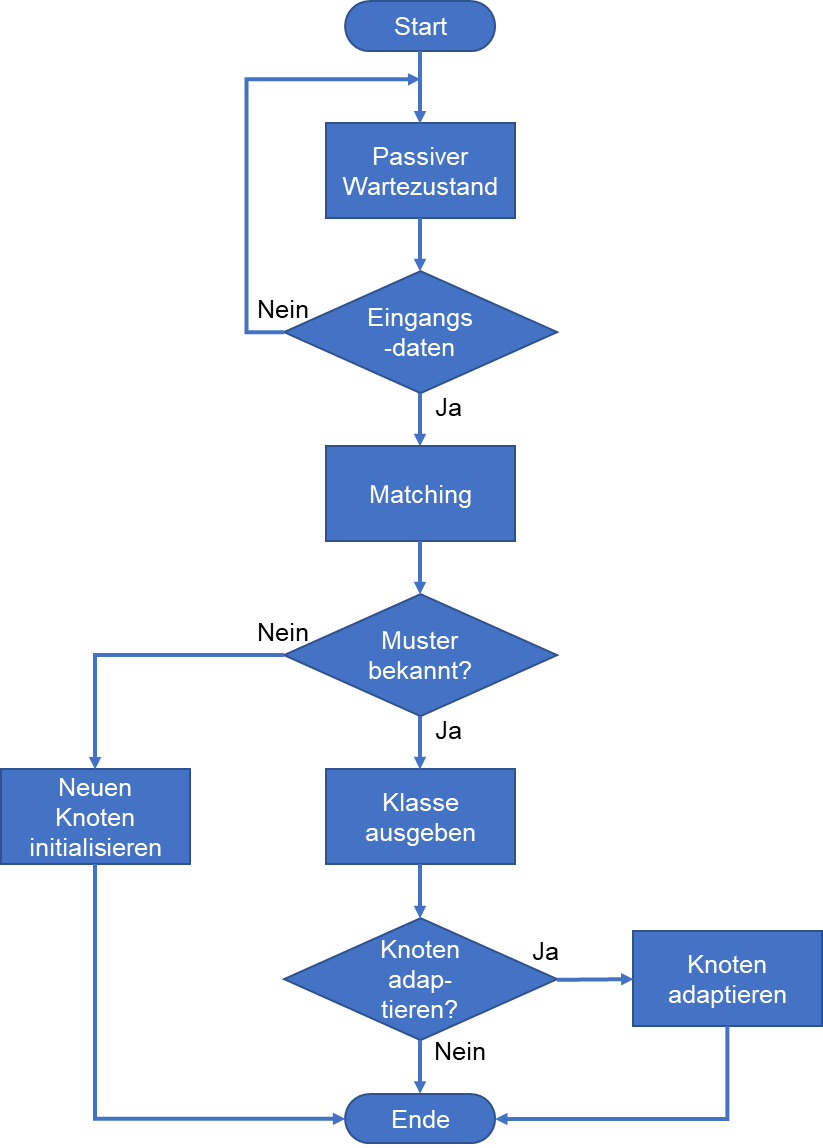


Abbildung 11: Ablaufdiagramm des Betriebs eines ART-Netzwerk

Weiter gibt es zwei unterschiedliche Methoden zum Trainieren eines ART-Netzwerks, 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 es 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 später für den in dieser Arbeit relevanten Anwendungsfall ein passender inkrementeller Klassifikator ausgewählt werden.

## Verteiltes Lernen

Verteiltes Lernen wird in der Literatur unter den Namen *Distributed* oder *Parallel* *Learning* beschrieben. Es gibt viele unterschiedliche Gründe, warum verteiltes Lernen genutzt 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 verteilten Geräten (z.B. Smartphones) sein. Da nur begrenzte Speicher zur Verfügung stehen 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 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, im Rahmen dieser Arbeit relevante, 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 dem jeweiligen 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 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 wird zwischen der Parallelisierung in Netzwerken allgemein und der Parallelisierung im Training von diesen Netzwerken unterschieden werden. Für die Parallelisierung in Netzwerken kann 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 des Netzes. In den nachfolgenden Methoden existiert 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 2 zusammengefasst werden [28].

Tabelle 2: 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.

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 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. Die unterschiedlichen Knoten können untereinander kommunizieren und Parameter austauschen.

Generell ist eine PS Infrastruktur förderlich für die Leistung und die Fehlertoleranz des Netzwerkes, da mithilfe eines zentralisierten PS zentral Checkpoints 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 entstehen im Vergleich zu einem dezentralisierten Ansatz.

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 dieselben Ergebnisse wie mit größeren Ensembles erzielt werden [28].

Als zusätzliche, spezifische Methode wird verbündetes Lernen (*federated* Learning) vorgestellt. Diese Methode wurde 2016 in [29] vorgestellt. Das Ziel dieser Methode ist ein hochqualitatives, zentralisiertes Modell auf Basis vieler verteilter Netzwerke zu trainieren. Dabei liegen die Daten ungleichmäßig verteilt über diese Knoten vor. Die lokalen Knoten werden dabei als Rechnerknoten benutzt, die mithilfe der lokalen Daten Optimierungen durchführen. Die lokalen Daten müssen bei dieser Methode 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 verbündeten Lernens werden lediglich Anpassungen an den zentralen Server geschickt (z.B. Gradienten-Vektor), wodurch die benötigte Kommunikationsbandbreite im Vergleich zu den kompletten Trainingsdaten drastisch reduziert wird. Gleichzeitig ist die geschickte Information deutlich abstrahierter von den möglicherweise personalisierten Daten. *Federated Learning* kann mit folgenden Notationen definiert werden: Es gibt Besitzer von Knoten (z.B. Smartphones), mit den gesammelten persönlichen 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-negativen realen Zahl in Gleichung **(14)** beschrieben werden:

|  |  |  |
| --- | --- | --- |
|  |  | (14) |

Ein föderiertes System hat somit einen -Genauigkeit Verlust. Föderierte Systeme können weiter in unterschiedliche Anwendungsfälle kategorisiert werden, die in Tabelle 3 dargestellt werden [30].

Tabelle 3: 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 (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 auf Basis der Features 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 unterschiedliche Netzwerke unterschiedliche Bilder bei diesem Anwendungsbeispiel des verteilten Lernens. 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 dargestellt. Zudem wurden Schwierigkeiten und unterschiedliche Methoden dargestellt. Die unterschiedlichen Methoden wurden in Kategorien eingeteilt. Zum Ende wurde ein aktueller spezifischer Algorithmus des verteilten Lernens beschrieben, der für den im Rahmen dieser Arbeit untersuchten Anwendungsfall mögliche Lösungen bereithält.

# Lifelong Deep Neural Network Algorithmus

In dieser Arbeit wird ein Lifelong Deep Neural Network Algorithmus untersucht und prototypisch umgesetzt. Der Algorithmus ist in [1] detailliert beschrieben. In diesem Kapitel wird der Algorithmus, sowie Vor- und Nachteile davon beschrieben. Auch werden Behauptungen, die nach [2] durch den Algorithmus möglich sind, genannt und auch in Relation zu bereits bekannten Methoden gesetzt. Zum Schluss wird eine kurze Zusammenfassung gegeben.

## Beschreibung

Der Lifelong Deep Neural Network 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:

1. Es ist unmöglich das System „on-the-fly“ mit neuem Wissen upzudaten
2. Lernen während dem gesamten Einsatzzyklus eines Gerätes ist ohne regelmäßige Kommunikation mit Servern und ohne eine erhebliche Wartezeit für ein Wissensupdate unmöglich
3. Das Erlernen neuer Informationen verbraucht Serverspeicher, Energie und lokalen Gerätespeicher um alle Eingangsdaten unendlich lange für das weitere Training abzuspeichern
4. Es ist unmöglich auf einem kleinen Endgerät zu lernen (trainieren)
5. 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.

In diesem Lifelong Deep Neural Network Algorithmus sollen nun mithilfe von zwei Modulen diese Probleme überwunden werden. Dafür wird ein langsam lernendes neuronales Netzwerk (z.B. DNN) als erste Stufe genutzt (Modul A). Dieses Netzwerk ist ein vortrainierter Feature-Extraktor. Je nach Anwendungsfall kann dieses Netzwerk theoretisch während der Laufzeit weiter trainiert werden mit kleinen, langsamen Updateschritten 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 solche Feature-Extraktoren für Bilderklassifizierung genutzt wird, ist Imagenet [31]. Dieser Datensatz besitzt mit 1000 verschiedenen Klassen eine sehr große Varianz der Objekte, wodurch eine gute Generalisierbarkeit erreicht werden kann.

Von dem vortrainierten DNN wird eine der letzten Feature-Schichten als Eingang des zweiten Moduls, Modul B, genutzt. Typischerweise wird als Interface die letzte Schicht des DNN vor der DNN-eigenen Klassifikationsschicht genutzt. Abbildung 12 zeigt ein beispielhaftes Modul A sowie das Interface zu Modul B. Jedes Rechteck stellt dabei eine willkürliche (Convolutional, Fully Connected) Schicht eines DNN dar. Die letzte grüne Schicht stellt die Schicht dar, welche für die Klassifikation verantwortlich ist. Innerhalb des L DNN Algorithmus wird diese Schicht nicht genutzt, sondern die Features der Schicht davor werden als Eingang für Modul B genutzt. Somit stellt Modul A eine modifizierte Version des vortrainierten DNN dar.

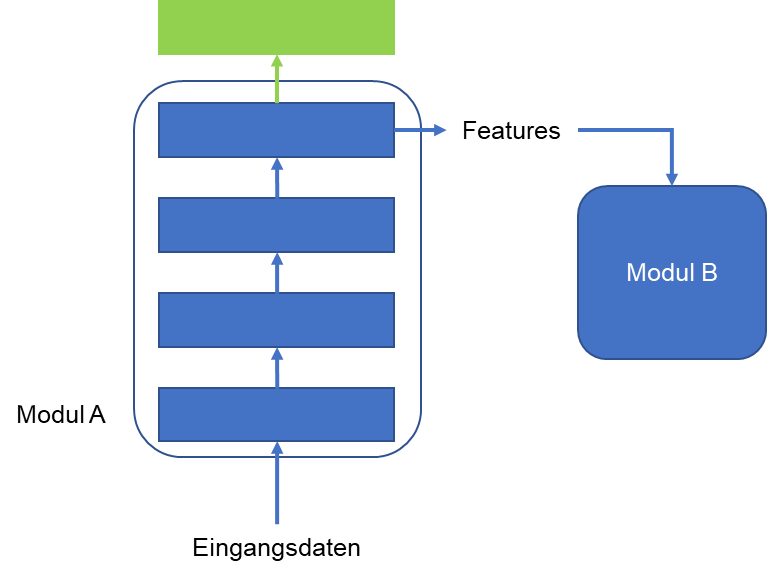


Abbildung 12: Modul A und Interface zu Modul B

Modul B ist ein inkrementeller Klassifikator, 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. die Klassifikation durchführen. Falls die erhaltenen Repräsentationen nicht bekannt sind, bzw. zu keiner bekannten Klasse passen, kann Modul B durch Interaktion mit dem User das korrekte Label erhalten und aufgrund seiner schnell lernenden Eigenschaft diese neue Klasse mit 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 genutzt werden.

Die grundsätzliche, allgemeine Architektur des L DNN Algorithmus ist in Abbildung 13 dargestellt.

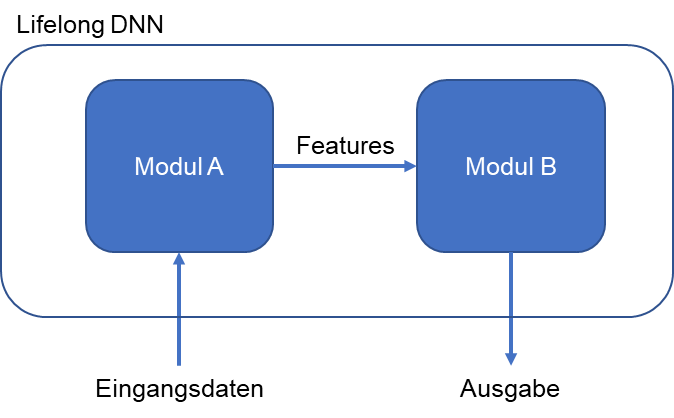


Abbildung 13: Graphische Darstellung des L DNN A

Je nach Anwendungsfall variieren die Eingangsdaten und die Ausgabe. Die konkreten Module A und B können nach Bedarf gewählt und geändert werden.

Der Algorithmus ist so aufgebaut, dass auf mehreren Geräten parallel neues Wissen erlernt werden kann. Dieses Wissen kann durch den direkten Austausch einzelner Geräte 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 14 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.

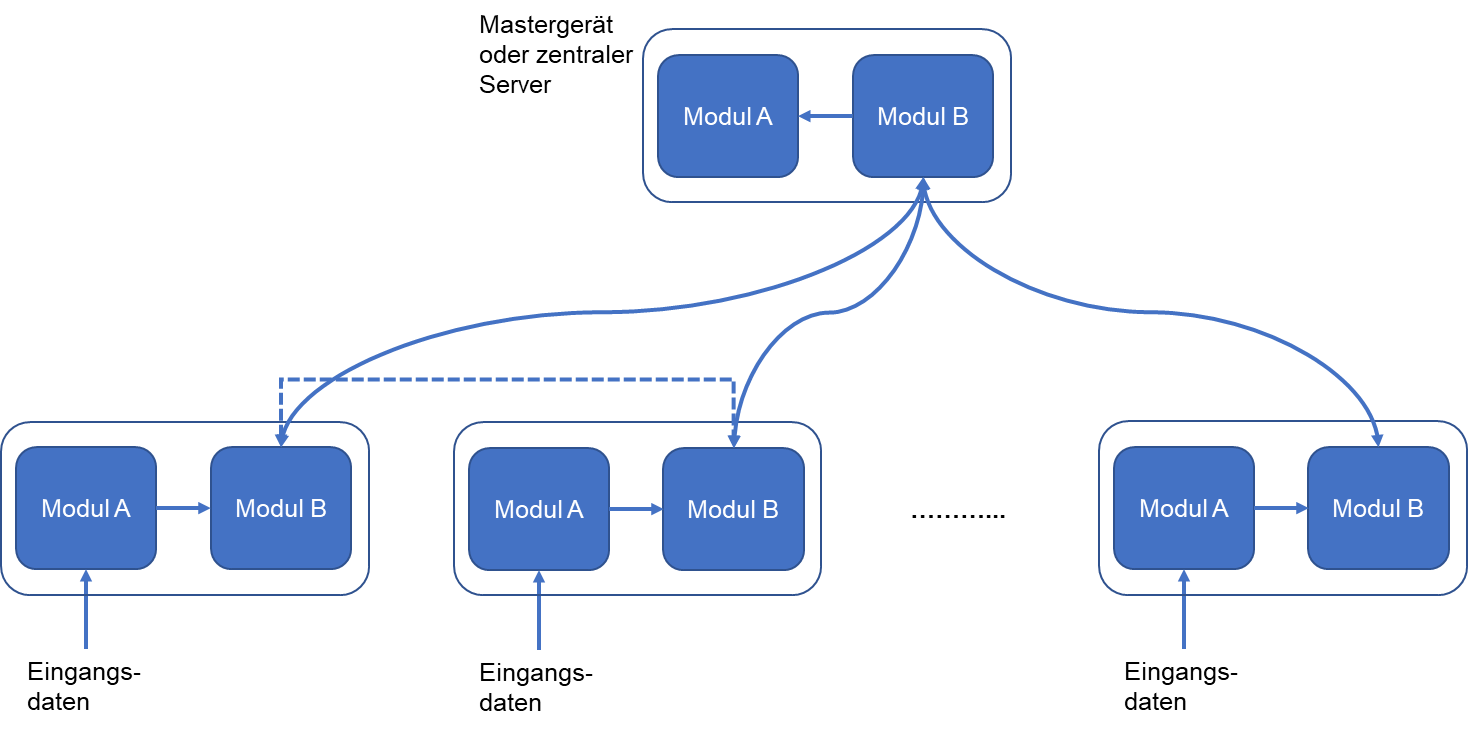


Abbildung 14: Beispielhaftes Szenario mit mehreren Endgeräten und einem zentralen Server

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, zwei Themen separat zu behandeln. Die Fähigkeit und Methoden des kontinuierlichen Lernens und des verteilten Lernens. Diese beiden Themen werden mit Bezug auf den beschriebenen Algorithmus im Folgenden detaillierter untersucht.

### Kontinuierliches Lernen

Der grundlegende Aufbau des L DNN Algorithmus hinsichtlich kontinuierlichen Lernens basiert auf der Dual-Memory Methode. Es werden zwei Submodule genutzt. Ein langsam lernendes oder fixiertes Modul und ein schnell lernendes Modul, welches neue 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-Extraktor auf Basis der Eingangsdaten fix ist oder sich nur sehr langsam ändert. Die Aufgabe, welche mit diesem Ansatz gelöst werden kann ist das inkrementelle Klassen Lernen (*Incremental Class Learning*). Das bedeutet, dass die grundlegende Aufgabe, z.B. Objekterkennung, gleich bleibt während des Lebenszyklus des Algorithmus. Es können jedoch neue Klassen erlernt werden.

Modul B ist ein schnell lernender Klassifikator, der bereits vortrainiert sein kann für manche Klassen oder komplett untrainiert während des Betriebes trainiert werden kann. Dafür ist ein Feedback des Nutzers nötig, um korrekte Labels und Klassenbezeichnungen zu erhalten. Das gewünschte Verhalten von Modul B wird im Folgenden kurz erläutert:

Wenn Feature-Vektoren von Modul A eintreffen, soll auf Basis der bereits bekannten Klassen die Klassenzugehörigkeit ermittelt werden. Wenn User-Feedback für dieses Sample vorhanden ist, soll eine weitere Anpassung für diese Klassen stattfinden, um eine bessere Generalisierbarkeit zu erreichen. Somit soll auch für bereits bekannte Klassen kontinuierlich weitergelernt werden. Für den speziellen 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 definiert, 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 folgende Gleichung **(15)** genannt:

|  |  |  |
| --- | --- | --- |
|  |  | (15) |

Dabei stellt die Anzahl an bereits erlernten Kategorien (Klassen) dar und die gesamte Anzahl an Kategorie-Knoten im Netzwerk. Der Skalierungsfaktor 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 häufig neue Klassen erstellt werden. Ein zu niedriger Wert von vergrößert die False Positive Rate, da Samples von neuen, bisher unbekannten Klassen potentiell zu bekannten Klassen zugeordnet werden.

Ein ausgewähltes Klassifikator-Netzwerk, welches diese Anforderungen erfüllen kann, wird dann schließlich als Modul B genutzt. Unterschiedliche Ansätze für das Modul B werden in der folgenden Konzeptionsphase beleuchtet und untersucht.

### 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 2.3). Wie in Abbildung 14 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 Prinzip des *Federated Learning* [29],[30]. Es werden auf mehreren Geräten lokale Daten generiert, welche jedoch nur genutzt werden um das lokale Netzwerk zu trainieren. Diese geänderten und erlernten Parameter können dann wiederum ausgetauscht werden, um neues Wissen auszutauschen. Nach der Kategorisierung aus Tabelle 3 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, die Konsolidierung neuen Wissens. Dieser Schritt wird genutzt um eine effiziente Struktur der einzelnen Klassifikatoren zu erzielen, wodurch der Speicherbedarf gesenkt werden kann und dadurch auch 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 werden und der Speicherbedarf reduziert werden.

Dieses konsolidierte Wissen der einzelnen Netzwerke kann dann kommuniziert werden. Dafür werden zu willkürlichen Zeitpunkten (z.B. nachdem neue Klassen erlernt wurden, nach einem definierten Zeitraum etc.) die konsolidierten Parameter des schnell lernenden Moduls B ausgetauscht. Dies kann geschehen, indem alle Geräte ihre Parameter einem zentralen Server/Gerät melden, oder einzelne Geräte tauschen sich „peer-to-peer“ untereinander aus. 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 werden Redundanzen zwischen einzelnen Geräten entfernt und der Speicherbedarf wird durch das Kombinieren mehrerer Parameter gesenkt im Vergleich zu einem naiven Ansatz, bei dem alle Repräsentationen aneinandergehängt werden. 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 erhalten werden 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, das einzelne Geräte erlernt haben, mit anderen Geräten ausgetauscht werden.

## Vorteile

Der dargestellte Algorithmus hat einige Vorteile gegenüber klassischen DNN-Architekturen und Algorithmen zu bieten, die im Folgend genannt und erläutert werden.

Der große 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 Method). Mithilfe des Algorithmus können inkrementell neue Klassen schnell erlernt werden, ohne dafür alte Trainingsdaten zu benötigen. Dadurch ist keine Speicherung großer Datenmengen notwendig. Zudem ist es durch diese beiden Vorteile es möglich, auf einem lokalen Endgerät (z.B. Smartphone) ohne große Rechen- oder Speicherkapazität weiter zu lernen. Aufgrund der beschriebenen Architektur sind deutlich weniger Instanzen pro Klasse notwendig, um diese zu erlernen, wodurch die Trainingszeiten von einzelnen Klassen erheblich reduziert wird. Deshalb kann dieser Algorithmus von theoretischer Perspektive aus in Echtzeit trainiert werden. In Tabelle 4 sind die genannten Vorteile im Vergleich zu einem traditionellen DNN nach [2] dargestellt.

Tabelle 4: Übersicht der 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 großer Vorteil des Algorithmus 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 erlerntes Wissen auf Basis von sicherheitskritischen oder persönlichen Daten ausgetauscht werden, ohne dass diese Daten versendet und/oder kostspielig gespeichert und gesichert werden müssen.

## 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 Feature für Modul B extrahiert, muss vortrainiert werden. Um eine gute Generalisierung zu erzielen, sollten möglichst viele unterschiedliche Bilder gesehen werden. Dafür wird in der Regel der Imagenet-Datensatz genutzt, welcher 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 muss erwähnt werden, dass durch das vortrainierte Modul A keine weitere Aufgabe durch den L DNN Algorithmus erlernt werden kann. 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.

## Zusammenfassung und Vergleich zu klassischen Ansätzen

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

Die grundlegende Architektur basiert auf bereits bekannten Dual-Memory Methoden, mit einem langsam (oder gar nicht) lernenden Modul A und einem schnell lernenden Modul B. Modul A ist dabei ein typisches DNN, welche genutzt wird um sinnvolle Features für Modul B auf Basis der Eingangsdaten zu extrahieren. Dieses DNN wird mithilfe standardmäßiger 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 gefunden, und benötigen das 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 dabei folgt dem Ansatz des *Federated Learning*. Mit diesem Ansatz können sich einzelne Netzwerke auf Basis lokal verfügbarer Trainingsdaten separat trainieren. Die erlernten Parameter werden wiederum mit einem zentralen Server oder 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.

# 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] | 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] | D. Jia, D. Wei, R. Socher, L. Li-Jia, L. Kai und F.-F. Li, „ImageNet: A large-scale hierarchical image database,“ *2009 IEEE Conference on Computer Vision and Pattern Recognition,* pp. 248-255, 2009. |
| [32] | 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. |