Karlsruher Institut für Technologie

Fraunhofer IOSB

Untersuchung der Kombination von 3D Convolution auf Bilddaten und optischem Fluss zur Erzeugung von Temporal Action Proposals

Masterarbeit
von

Patrick Schlosser

An der Fakultät für Informatik
Institut für Anthropomatik und Robotik

Erstgutachter:	Prof. Dr.-Ing. Rainer Stiefelhagen
Zweitgutachter:	Prof. Dr.-Ing. Jürgen Beyerer
Betreuender Mitarbeiter:	Dr.-Ing. David Münch

Bearbeitungszeit: 01.06.2018 – 30.11.2018

Computer Vision for Human-Computer Interaction Lab
Institut für Anthropomatik und Robotik
Fakultät für Informatik
Karlsruher Institut für Technologie

Abteilung Objekterkennung
Fraunhofer-Institut für Optronik, Systemtechnik und Bildauswertung

Titel: Untersuchung der Kombination von 3D Convolution auf Bilddaten und optischem Fluss zur Erzeugung von Temporal Action Proposals
Autor: Patrick Schlosser

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Quellen verwendet zu haben.

Karlsruhe, 30. November 2018

.....
(Patrick Schlosser)

Zusammenfassung

Im Bereich der Videoanalyse ist es eine wichtige Aufgabe zu bestimmen, welche Aktionen wann in einem Video stattfinden. Von besonderem Interesse sind dabei oft Aktionen, die von Menschen ausgeführt werden, beispielsweise im Rahmen der Videoüberwachung. Menschliche Aktionen werden dabei stark durch die ausgeführte Bewegung charakterisiert [Joh73], da kaum eine menschliche Aktion ohne eine zugehörige Bewegung ausgeführt wird. Die automatische Erkennung und zeitliche Eingrenzung von Aktionen ist kein triviales Problem, da durch den Faktor Mensch eine hohe Intra-Klassen-Varianz besteht.

In dieser Arbeit wird das Problem der *Temporal Action Proposals Generierung* bearbeitet: Die Detektion von Zeitfenstern, die mit hoher Wahrscheinlichkeit eine Aktion enthalten, und diese zeitlich möglichst genau eingrenzen. Es wird untersucht, ob Methoden, die 3D-Faltungen auf der Bildsequenz des Videos einsetzen, dadurch verbessert werden können, dass zu diesem einzelnen Stream ein zweiter Stream hinzugefügt wird, auf dem Bilder, die den optischen Fluss visualisieren, ebenfalls mit 3D-Faltungen verarbeitet werden. Darüber hinaus wird untersucht, wie beide Streams gewinnbringend fusioniert werden können.

Als Grundlage für diese Arbeit wird das SST-Netzwerk [BES+17] gewählt, welches das C3D-Netzwerk [TBF+15] verwendet, um Bildsequenzen mit 3D-Faltungen zu verarbeiten. Das gesamte Netzwerk wird zu einem Two-Stream-Netzwerk erweitert, das über einen Stream für die originale Bildsequenz und über einen Stream für die Bildsequenz, die den zugehörigen optischen Fluss visualisiert, verfügt. Beide Streams bleiben mindestens so lange separiert, bis beide Bildsequenzen durch 3D-Faltungen in den jeweiligen C3D-Netzwerken verarbeitet wurden. Im Anschluss an diese Verarbeitung werden verschiedene Varianten mit unterschiedlichen Stellen der Fusion untersucht. Zwei dieser Varianten folgen dem Prinzip der Mid-Fusion, indem das eine Mal die Fusion der C3D-Netzwerke dadurch geschieht, dass sie in ihrer letzten Fully Connected Layer zusammengeführt werden, und das andere Mal dadurch, dass die Streams nach der Verarbeitung durch die C3D-Netzwerke durch Konkatenation der Ausgaben zusammengeführt werden. Zwei weitere Varianten wenden das Prinzip der Late-Fusion an und untersuchen die Fusion der Streams dadurch, dass die SST-Netzwerke in ihrer letzten Fully Connected Layer zusammengeführt werden, und dadurch, dass ein gewichteter Durchschnitt über die Ausgabe beider separater SST-Netzwerke gebildet wird. Die verschiedenen Varianten der Fusion werden auf dem THUMOS'14-Datensatz experimentell untersucht, ebenso wie verschiedene Parametrisierungen der Modelle.

Unter allen betrachteten Varianten und Parametrisierungen werden experimentell die besten bestimmt. Ein Vergleich mit der originalen Implementierung des SST-Netzwerks und einer anders parametrisierten TensorFlow-Implementierung desselben Netzwerks, welche beide nur auf den originalen Bildsequenzen arbeiten, wird durchgeführt; es kann experimentell gezeigt werden, dass die hier entwickelten Two-Stream-Modelle bezüglich der betrachteten Metriken in fast allen Bereichen eine Verbesserung erzielen – in den übrigen Bereichen erzielen die besten Two-Stream-Modelle vergleichbare Ergebnisse. Zusätzlich wird ein Vergleich von Methoden zur Bestimmung des optischen Flusses durchgeführt; das primär verwendete Verfahren von Brox et al. [BBPW04] wird mit FlowNet2 [IMS+17] verglichen. Mit letzterem kann keine weitere Verbesserung erzielt werden.

Summary

In the area of video analysis one important task is to determine when actions take place and which actions take place. Actions performed by humans are of particular importance, for example in the field of video surveillance. Human actions are highly characterized by the performed motion [Joh73], as almost no human action is performed without it. Automatic classifiaction and temporal localization is a dificult task, as humans cause a high intra-class variance.

This work is focused on the creation of *temporal action proposals*: the detection of time slots which contain an action with high probability and localize it in the video as precise as possible. It is examined whether methods utilizing a single stream with 3D convolutions on the original image sequence of a video can be improved by adding a second stream utilizing 3D convolutions on an image sequence of optical flow. Different ways of fusing both streams gainfully are investigated as well.

As a base for this work the SST network [BES+17] is chosen, which utilizes the C3D network [TBF+15] to carry out 3D convolutions on image sequences. The whole network is transformed into a two-stream network with one stream operating on the original image sequence and the other stream operating on an image sequence derived from optical flow. Both streams stay separated at least until the 3D convolutions have been carried out. Different forms of fusing both separated streams are examined. Two of the designed variants follow the mid fusion principle: one fuses the two streams by fusing the separate C3D networks, making their last fully connected layer a shared one, the other one fuses the two streams by concatenating the separate outputs of both C3D networks. Another two variants are designed using the late fusion principle: one fuses both separate SST networks by making their last fully connected layer a shared one, the other one fuses both streams by calculating a weighted average of the outputs of both SST networks. An experimental evaluation of these variants is conducted, as well as an evaluation of different parametrizations for those variants.

Among all variants and parametrizations the best-performing ones are determined experimentally on the THUMOS'14 dataset. A comparison with the original implementation of the SST network and a differently parametrized TensorFlow implementation of the SST network – both operating on the original image sequences – is conducted. All of the designed two-stream models show superior performance regarding the used metrics in most areas. There were only small areas left where the best two-stream models show only comparable performance. In addition, a comparison between two optical flow extraction methods is conducted: the Brox flow [BBPW04] used throughout most experiments is compared with optical flow extracted by FlowNet2 [IMS+17] – the usage of FlowNet2 led to no improvements.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Im Bereich der Videoanalyse ist es eine wichtige Aufgabe zu bestimmen, welche Aktionen wann in einem Video stattfinden. Von besonderem Interesse sind dabei oft Aktionen, die von Menschen ausgeführt werden, beispielsweise im Rahmen der Videoüberwachung. Menschliche Aktionen werden dabei stark durch die ausgeführte Bewegung charakterisiert [Joh73], da kaum eine menschliche Aktion ohne eine zugehörige Bewegung ausgeführt wird. Die automatische Erkennung und zeitliche Eingrenzung von Aktionen ist kein triviales Problem, da durch den Faktor Mensch eine hohe Intra-Klassen-Varianz besteht. Die Aktionen unterscheiden sich je nachdem, von wem sie ausgeführt werden. Dabei muss das System nicht nur mit verschiedener Kleidung, verschiedener Farbgebung, unterschiedlichem Körperbau und unterschiedlichen Arten, ein und dieselbe Aktion auszuführen, zurechtkommen, sondern auch mit unterschiedlichen Kamerawinkeln und unterschiedlichen Orten, an denen die Aktion durchgeführt wird. Die Probleme werden anhand eines Anwendungsbeispiels leichter verständlich: Man stelle sich vor, dass aus dem Video eines Fußballspiels für einen kurzen Beitrag alle Highlights in Form der Torschüsse extrahiert werden sollen. Dabei müssen Torschüsse unabhängig von den Teams, die sich in der Farbe ihrer Trikots unterscheiden, und unabhängig von der Statur der Spieler erkannt werden. Auch ob mit dem rechten oder linken Fuß aufs Tor geschossen wird, darf keinen Unterschied machen; einfache Pässe dürfen nicht mit Torschüssen verwechselt werden. Ein Torschuss kann aus einem Sprint heraus ausgeführt werden oder auch als Freistoß mit langsamerem Anlauf. Auch die Position der Kamera kann sich ändern, da das Spiel aus mehreren Winkeln und mit beweglichen Kameras aufgenommen wird; ebenfalls kann sich die Beleuchtung ändern, abhängig davon, ob die Sonne scheint oder nicht. Alle diese Faktoren zeigen, dass dieselbe Aktion im Video sehr unterschiedlich ausgeprägt sein kann; trotz dieser vielen möglichen Ausprägungen muss sie von sehr ähnliche Aktionen – wie im Beispiel dem Pass – unterschieden werden können. Um das Problem zu lösen, kann beispielsweise so vorgegangen werden, dass das gemeinsame Problem der Lokalisierung und Bestimmung der Aktion in zwei Teilprobleme aufgeteilt wird, die separat zu lösen sind: Zuerst können Abschnitte im Video bestimmt werden, die höchstwahrscheinlich eine Aktion enthalten. Die genaue Bestimmung der Aktion kann dann auf Basis dieser Abschnitte erfolgen. Ein Beispiel für Zeitabschnitte, die es zu bestimmen gilt, ist in Abbildung 1.1 zu finden.

Abbildung 1.1.: Einzelbilder aus einer Videosequenz (oben). Exemplarischer Zeitstrahl, der Zeitspannen, die eine Aktion von Interesse enthalten, grün markiert (unten). Im diesem Beispiel soll die Aktion „Billiard spielen“ bestimmt und zeitlich eingegrenzt werden. Gilt es zuerst Zeitabschnitte zu bestimmen, die eine Aktion enthalten, entsprechen diese im Idealfall den grün markierten Zeitspannen. Bilder der Videosequenz aus [JLZ+14].

1.1. Problemstellung und Ziel der Arbeit

Begriffsdefinitionen (entsprechend ActivityNet Challenge 2017[1]):

Temporal Action Localization/Temporal Action Detection: Im Rahmen der Temporal Action Localization bzw. Temporal Action Detection sollen Aktionen von Interesse zeitlich in einem Video eingegrenzt und klassifiziert werden. Hierbei dienen ungekürzte Videos als Eingabe, die eine oder mehrere, teils unterschiedliche Aktionen enthalten. Es werden ein Zeitintervall, der Name der Aktion und ein Wert für die Sicherheit des Ergebnisses pro detektierter Aktion gefordert.

Temporal Action Proposal: Bei Temporal Action Proposals handelt es sich um Zeitfenster in ungekürzten Videos. Sie verfügen über eine zugehörige Bewertung, die einen hohen Wert annehmen soll, wenn das Zeitfenster eine Aktion von Interesse enthält und diese zeitlich möglichst genau eingrenzt. Ziel ist es, statt einer breiten Masse an Zeitfenstern wenige, vielversprechende Zeitfenster für eine nachfolgende, rechenintensive Weiterverarbeitung bereitzustellen und so den Aufwand nachfolgender Verarbeitungsschritte zu reduzieren.

Trimmed Action Recognition: Bei der Trimmed Action Recognition werden Videos, die mit lediglich einer einzelnen Aktion assoziiert sind, verarbeitet. Ziel ist es, diese eine Aktion zu bestimmen. Bei dem Kinetics-Datensatz [KCS+17] dienen so beispielsweise auf etwa zehn Sekunden gekürzte Videos als Eingabe; zugehörige Grundwahrheit ist jeweils die Aktion, die in dem Video auftritt. Im Rahmen dieser Arbeit wird für dieses Problem oft die deutsche Bezeichnung Aktionserkennung verwendet.

Die zuvor beschriebene Aufgabe auf dem Gebiet der Videoanalyse entspricht der Problemstellung der Temporal Action Localization: Es ist das Ziel, Aktionen, die in einem ungekürzten Video auftreten, auf eine bestimmte Zeitspanne im Video einzugrenzen und zu bestimmen, um welche Aktionen es sich handelt. Lösungsverfahren für das Problem sind meist hierarchisch aufgebaut: Zuerst werden Temporal Action Proposals generiert; es handelt sich um Zeitfenster im Video, denen eine Bewertung zugeordnet ist. Eine gute Bewertung sollte vergeben werden, wenn das Zeitfenster eine Aktion enthält und diese zeitlich möglichst genau eingrenzt. Nach der Erzeugung der Temporal Action Proposals

[1] `http://activity-net.org/challenges/2017/guidelines.html`

wird auf diesen eine Klassifikation durchgeführt, um zu bestimmen, ob eine Aktion enthalten ist, und falls ja, welche. Für die Temporal Action Proposal Generierung existieren Ansätze von einfachen Sliding-Window-Verfahren [SWC16] bis hin zu tiefen Neuronalen Netzen [EHNG16, BES+17]. Monolithische Ansätze für die Temporal Action Localization sind zwar aus verschiedenen Teilkomponenten aufgebaut, werden aber Ende-zu-Ende trainiert – so kann beispielsweise auf explizite Generierung von Temporal Action Proposal verzichtet werden [BEG+17].

Jüngste Methoden zur Temporal Action Proposal Generierung setzen zumeist auf die Verarbeitung der Videodaten durch 3D Convolutional Neural Networks (3D ConvNets) [EHNG16, BES+17, GYN17, GYS+17], teils kommt optischer Fluss im Rahmen eines Two-Stream-Netzwerks [LZS17, GYN17] zum Einsatz; hierbei wird der optische Fluss jedoch nicht mit Hilfe von 3D-Faltungen verarbeitet, lediglich 2D-Faltungen finden Anwendung. Erst jüngste Arbeiten [CVS+18, NLPH18] auf dem Gebiet der Temporal Action Localization, bei der die Temporal Action Proposal Generierung eine Teilaufgabe darstellt, verwenden Two-Stream 3D ConvNets auf Bilddaten und optischem Fluss. Bei der Aktionserkennung werden 3D-Faltungen auf dem optischen Fluss bereits länger erfolgreich eingesetzt [CZ17, VLS18, KT18], insbesondere im Rahmen von Two-Stream-Netzwerken.

Inspiriert durch die erfolgreiche Verwendung des optischen Flusses bei der Aktionserkennung wird bei dieser Arbeit die Frage geklärt, ob und wie sich optischer Fluss in Kombination mit 3D-Faltung (engl. *3D Convolution*) gewinnbringend zur reinen Temporal Action Proposal Generierung einbringen lässt. Bereits bei der Aktionserkennung zeigte die Verwendung von 3D-Faltung auf optischem Fluss bessere Resultate als auf RGB-Daten [VLS18]. Darüber hinaus wird in einer anderen Arbeit [SLLG+17] darauf verwiesen, dass das eigenständige Lernen des optischen Flusses durch ein Neuronales Netz zwar prinzipiell möglich ist, sich jedoch bereits das explizite, überwachte Lernen des optischen Flusses durch ein Neuronales Netz als schwierig gestalten kann. Mehrere Arbeiten [SLLG+17, CZ17] verweisen zudem auf die Nützlichkeit der Verwendung des optischen Flusses in Ergänzung zu den Bilddaten. Darüber hinaus stellt die Bewegung eine Kernkomponente menschlicher Aktionen dar [Joh73]; der optische Fluss beschreibt stattfindende Verschiebungen zwischen Bildpaaren [Ste08], welche auch die stattfindende Bewegung abbilden und unabhängig von der konkreten Erscheinungsform der ausführenden Person sind. Intuitiv ist daher zu erwarten, dass die 3D-Faltung auf dem optischen Fluss die Bewegung über Zeit einfängt. Diese Intuition zusammen mit den vorgestellten Erkenntnissen anderer Arbeiten liefert die Grundlage dieser Arbeit.

Für die Umsetzung bietet sich insbesondere die Verwendung des SST-Netzwerks [BES+17] an, welches das C3D-Netzwerk [TBF+15] zur Extraktion von Features verwendet und anschließend ein Recurrent Neural Network (RNN) einsetzt. Das C3D-Netzwerk setzt sich aus mehreren Blöcken zusammen, die 3D-Faltungen und 3D-Pooling ausführen; es erhält dabei den zeitlichen Zusammenhang über jeweils 16 aufeinanderfolgende Bilder, die zusammen verarbeitet werden. Die Ausgabe einer der beiden Fully Connected Layers (fc6, fc7), die sich an die Blöcke von 3D-Faltungen und 3D-Pooling anschließen, liefert die so genannten C3D-Features – eine kompakte Repräsentation der 16 Bilder in einem 4096-elementigen Vektor. Diese Vektoren bilden die Eingabe für den Rest des SST-Netzwerks, welcher Temporal Action Proposals generiert. Mit dem dort eingesetzten RNN ist es möglich, einen zeitlichen Zusammenhang zwischen den C3D-Features herzustellen. In dieser

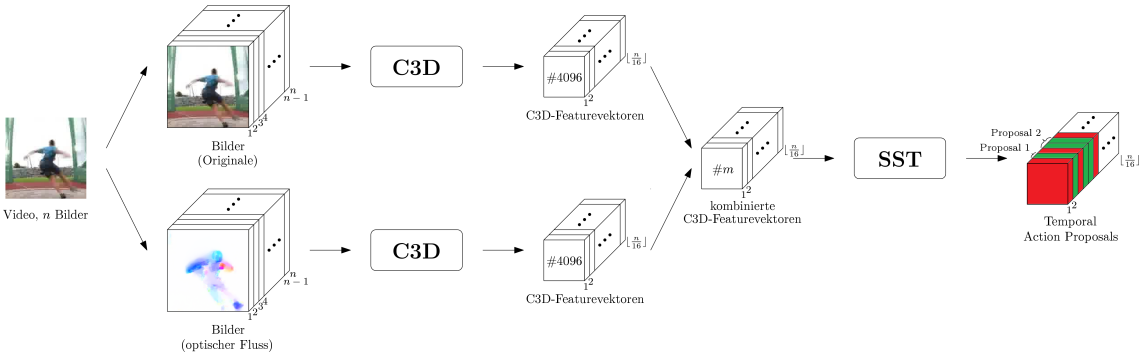


Abbildung 1.2.: Exemplarische Darstellung eines möglichen Two-Stream-Modells. Optischer Fluss wird aus dem Eingabevideo extrahiert, C3D-Features werden separat für Bilddaten und Bilder des optischen Flusses durch jeweils ein C3D-Netzwerk extrahiert. So entstehen für jeweils 16 Bilder des Eingabevideos zwei separate, 4096-elementige Vektoren. Die separaten Vektoren werden anschließend fusioniert und dienen als Eingabe für ein SST-Netzwerk, welches Temporal Action Proposals ausgibt. Die Stelle der Fusion wurde hier zur Veranschaulichung nach dem C3D-Netzwerk gewählt, kann aber auch an anderen Stufen in der Verarbeitungskette stattfinden. Eingabebild aus THUMOS'14 [JLZ+14].

Arbeit soll das gesamte SST-Netzwerk zu einem Two-Stream-Netzwerk erweitert werden, bei dem zusätzlich zu dem Stream, der mit Hilfe des C3D-Netzwerks 3D-Faltungen auf den originalen Bilddaten durchführt, ein weiterer Stream eingeführt wird, der mit Hilfe des C3D-Netzwerks 3D-Faltungen auf Bildern ausführt, die den optischen Fluss visualisieren. In Abbildung 1.2 ist die initial umzusetzende Architektur dargestellt (vgl. Abschnitt 4.3.1), dabei bezeichnet in dieser Abbildung SST das SST-Netzwerk ohne das C3D-Netzwerk. Die Fusion der anfangs separaten Netzwerke ist exemplarisch dargestellt, alternativ kann sie bereits im C3D-Netzwerk oder im bzw. nach dem SST-Netzwerk geschehen. Es soll untersucht werden, ob mit dem zusätzlichen Einsatz von 3D-Faltungen auf den Bildern des optischen Flusses und den daraus resultierenden Two-Stream-Modellen eine Verbesserung gegenüber dem klassischen SST-Netzwerk erzielt werden kann, welches nur einen Stream mit 3D-Faltungen auf den originalen Bilddaten einsetzt. Darüber hinaus soll untersucht werden, an welcher Stelle beide Streams gewinnbringend fusioniert werden können.

Die Auswertung findet auf dem THUMOS'14-Datensatz [JLZ+14] statt, evaluiert werden folgende Metriken: Durchschnittlicher Recall bezüglich der durchschnittlichen Anzahl an Proposals, Recall bei durchschnittlich 1000 Proposals bezüglich der temporal Intersection over Union (tIoU) und durchschnittlicher Recall bei durchschnittlich 1000 Proposals. Die Ergebnisse werden unter anderem mit denen des originalen SST-Netzwerks verglichen, um Rückschlüsse auf die Nützlichkeit des zusätzlichen Streams mit 3D-Faltungen auf den Bildern des optischen Flusses ziehen zu können.

1.2. Schwerpunkte und Beiträge der Arbeit

Im Nachfolgenden werden die einzelnen durch diese Arbeit erbrachten Beiträge vorgestellt.

1.2.1. Beitrag #1

Das gesamte SST-Netzwerk wurde zu einem Two-Stream-Netzwerk erweitert. Im einen Stream werden dabei nach wie vor die originalen Bildsequenzen mit Hilfe von 3D-Faltungen im C3D-Netzwerk verarbeitet, im ergänzten Stream werden Bildsequenzen, die den optischen Fluss visualisieren, mit Hilfe von 3D-Faltungen im C3D-Netzwerk verarbeitet. Nach der Anwendung der 3D-Faltungen wurden in dieser Arbeit verschiedene Varianten zur Fusion beider Streams untersucht.

1.2.2. Beitrag #2

Zusätzlich zur Untersuchung verschiedener Varianten der Fusion wurden Untersuchungen zur Parametrisierung der resultierenden Two-Stream-Netzwerke durchgeführt. Experimentell wurden geeignete Parametrisierungen bestimmt und verglichen.

1.2.3. Beitrag #3

Es wurde eine experimentelle Auswertung der entworfenen Two-Stream-Modelle mit unterschiedlichen, experimentell bestimmten Parametrisierungen durchgeführt. Die Ergebnisse der verschiedenen Varianten wurden miteinander verglichen, ebenso wurde ein Vergleich zu Single-Stream-Varianten des SST-Netzwerks durchgeführt.

1.2.4. Beitrag #4

Für die Bestimmung des optischen Flusses, der anschließend in Bildsequenzen visualisiert wird, wurden zwei Verfahren zur Bestimmung des optischen Flusses verglichen und vergleichend ausgewertet: Das Verfahren nach Brox et al. [BBPW04] und FlowNet2 [IMS+17].

1.3. Aufbau der Arbeit

Kapitel 2 liefert einen Überblick über wichtige verwandte Arbeiten, die die Basis für die weitere Arbeit bilden. In Kapitel 3 werden anschließend Grundlagen zu Neuronalen Netzen und zum optischen Fluss vorgestellt, die für das weitere Vorgehen in dieser Arbeit wichtig sind. Kapitel 4 geht auf die im Rahmen dieser Arbeit verwendeten Modelle und entworfenen Two-Stream-Modelle ein; auf die Implementierung wird in Kapitel 5 eingegangen. Eine Auswertung der verschiedenen Modelle wird in Kapitel 6 vorgenommen, Kapitel 7 liefert eine abschließende Zusammenfassung mit Diskussion und Ausblick.

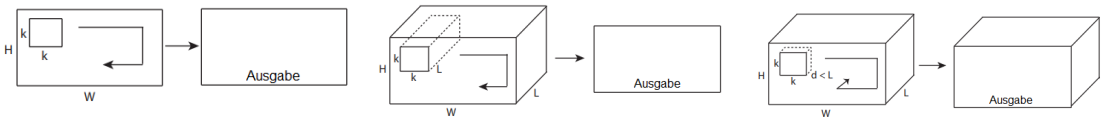
2. Verwandte Arbeiten

In diesem Kapitel werden Arbeiten vorgestellt, die direkt oder indirekt mit der Problemstellung der Generierung von Temporal Action Proposals verwandt sind, sowie Arbeiten, die sich mit dem Einsatz des optischen Flusses auf ähnlichen Gebieten beschäftigen. Dabei werden diejenigen Arbeiten im Detail betrachtet, die die Grundlage für diese Arbeit bilden. Auf verwandte Arbeiten im größeren Rahmen wurde bereits in Abschnitt 1.1 eingegangen.

2.1. Learning Spatiotemporal Features with 3D Convolutional Networks

Tran et al. verwenden im Rahmen ihrer Arbeit „Learning Spatiotemporal Features with 3D Convolutional Networks“ [TBF+15] ein Modell, das Bildsequenzen und Videos zum Zweck der Aktionserkennung verarbeitet; es wird durch ein 3D Convolutional Network realisiert. Hierbei wurde das Konzept der 2D Convolutional Neural Networks, die in der zweidimensionalen Bildebene operieren, um eine dritte, zeitliche Dimension erweitert. Das entworfene Neuronale Netz wird kurz als C3D bezeichnet – das Augenmerk liegt dabei darauf, dass ein kompakter Vektor auf Basis von Videodaten als Deskriptor für diese extrahiert und anschließend zur Klassifikation verwendet werden kann. Solche extrahierten Vektoren werden als C3D-Features bezeichnet. Sie werden aus jeweils 16 aufeinanderfolgenden Bildern eines Videos extrahiert, als Resultat werden die Bilddaten zu einem 4096-elementigen Vektor komprimiert. Dieser Vektor wird auf Basis der Aktivierungen der ersten der beiden Fully Connected Layers des Neuronalen Netzes gebildet – prinzipiell können jedoch auch die Aktivierungen der zweiten Fully Connected Layer verwenden werden, wie es beispielsweise in [BES+17] geschieht. Die extrahierten Aktivierungen unterlaufen anschließend noch eine L_2-Normalisierung.

Durch den Einsatz von 3D-Faltungen ist es möglich, zeitliche Informationen zu erhalten, die bei 2D-Faltungen verloren gehen würden [TBF+15]: So bewegt sich ein 2D-Kernel nur über die räumliche Ausdehnungen eines Bildes und erzeugt als Ausgabe wiederum nur ein einzelnes Bild. Ein 2D-Kernel kann zwar auch mehrere Bilder auf einmal verarbeiten, die Verschiebung des Filters findet jedoch wieder nur in der räumlichen Dimension statt und



(a) 2D-Faltung (b) 2D-Faltung (mehrere Bilder) (c) 3D-Faltung

Abbildung 2.1.: Darstellung verschiedener Möglichkeiten der Faltung [TBF+15]: a) Eine 2D-Faltung wird auf einem einzelnen Bild ausgeführt, was als Ausgabe ein einzelnes Bild erzeugt. b) Mehrere Bilder werden zu einer Eingabe zusammengefasst, indem die einzelnen Bildern auf verschiedene Kanäle verteilt werden. Eine 2D-Faltung wird auf diesen Kanälen ausgeführt, als Ausgabe entsteht wiederum ein einzelnes Bild. c) Eine 3D-Faltung wird auf den Bildern eines Videos ausgeführt. Durch die 3D-Faltung wird aus dem Raum-Zeit-Volumen, das als Eingabe fungiert, wiederum ein Raum-Zeit-Volumen erzeugt. Dabei wird der Kernel auch über die zeitliche Dimension verschoben. Abbildung aus [TBF+15].

es wird wieder nur ein einzelnes Bild als Ausgabe erzeugt. Gegensätzlich hierzu verhalten sich 3D-Kernels: Ihnen können Bildsequenzen mit einer zeitlichen Komponente übergeben werden; die Kernels operieren dabei dann sowohl auf der zeitlichen Dimension als auch auf den beiden räumlichen Dimensionen – in Folge wird aus einem Raum-Zeit-Volumen, definiert durch die Bildsequenz, wiederum ein Raum-Zeit-Volumen erzeugt. Entsprechend wird auch das Pooling auf die zeitliche Dimension erweitert. Abbildung 2.1 stellt den Sachverhalt dar. Basierend auf dieser Motivation ist es der nächste Schritt in der Arbeit von Tran et al., eine geeignete Architektur für den Einsatz von 3D Faltungen zu schaffen.

Als ein erster Schritt in Richtung einer solchen Architektur wird nach einer idealen Größe für die 3D-Kernels gesucht. Dabei wird für die Festlegung der räumlichen Dimension des Filters die Arbeit von Simonyan et al. [SZ14] herangezogen, die für 2D-Kernel die Dimension 3×3 als ideal identifiziert hat. Diese Größe wird für die räumliche Komponente des 3D-Kernels unverändert übernommen, während die Größe der zeitlichen Dimension des Kernels zu Untersuchungszwecken variiert wird. Zur Untersuchung wird eine vereinfachte Version des späteren C3D-Netzwerks eingesetzt, die aus fünf Convolutional Layers mit sich jeweils anschließender Max Pooling Layer besteht, sowie abschließend aus zwei aufeinanderfolgenden Fully Connected Layers. Zum Zwecke der Ausgabe endet die letzte Fully Connected Layer in einer Softmax Layer. Mit Hilfe dieser Architektur werden diverse Tests zum Finden einer idealen Größe der zeitlichen Dimension für den 3D-Kernel durchgeführt [TBF+15]: Zum einen werden Varianten getestet, bei denen die Größe der zeitlichen Dimension des Kernels über alle Convolutional Layers konstant bleibt; es werden die Größen 1, 3, 5 und 7 untersucht. Darüber hinaus werden Varianten mit zunehmender und abnehmender Größe getestet; die Größen betragen im zunehmenden Fall $3-3-5-5-7$ und im abnehmenden Fall $7-5-5-3-3$. Um zu verhindern, dass unterschiedliche Kernelgrößen Ausgaben unterschiedlicher Größe erzeugen, wird Padding eingesetzt. Für die genannten Varianten werden entsprechende Neuronale Netze trainiert, anschließend wird die erzielte Genauigkeit bei der Aktionserkennung ausgewertet. Bei dieser Auswertung erzielte die Architektur die besten Ergebnisse, bei der in allen Schichten die feste Größe 3 für die zeitliche Dimension der 3D-Kernels verwendet wurde. Entsprechend wurden 3D-Kernels mit der mit der Größe $3 \times 3 \times 3$ als experimentell bestimmte ideale Wahl identifiziert.

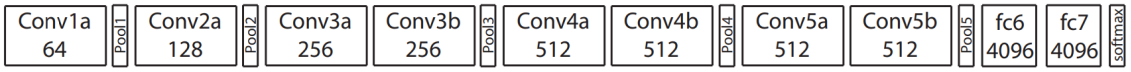


Abbildung 2.2.: Schematische Darstellung der Architektur des C3D-Netzwerks [TBF+15]: Auf fünf Blöcke von Convolutional und Pooling Layers mit ansteigender Anzahl an Kernels folgen zwei Fully Connected Layers und abschließend eine einzelne Softmax Layer. Abbildung aus [TBF+15].

Aus der Architektur, die zur Bestimmung der idealen Kernelgröße verwendet wurde, geht die endgültige Architektur des C3D-Netzwerks hervor [TBF+15]: Die Anzahl der Convolutional Layers wird von fünf auf acht erhöht, die Größe der einzelnen Fully Connected Layers wird auf 4096 verdoppelt. Nach wie vor werden die Convolutional Layers und die Pooling Layers in fünf Blöcken organisiert; die ersten beiden Blöcke verfügen jeweils über eine Convolutional Layer gefolgt von einer Pooling Layer, während die letzten drei Blöcke über jeweils zwei Convolutional Layers vor der Pooling Layer verfügen. Die erste Pooling Layer verwendet eine Kernelgröße von $1 \times 2 \times 2$, alle nachfolgenden Pooling Layers hingegen $2 \times 2 \times 2$. Dabei steht an erster Stelle die Größe der zeitlichen Dimension, an zweiter und dritter Stelle stehen die Größen der räumlichen Dimensionen. Der Stride wird entsprechend der Kernelgröße gewählt, sodass keine Überlappung beim Pooling stattfindet. Die Kernel der Convolutional Layers haben durchgehend die zuvor bestimmte Größe $3 \times 3 \times 3$, werden im Gegensatz zu den Kerneln der Pooling Layers jedoch durchgängig mit Stride $1 \times 1 \times 1$ verwendet. Die Anzahl der Kernels pro Schicht steigt von Block zu Block, innerhalb der Blöcke verwenden die Convolutional Layers gleich viele Kernels. Die einzelnen Blöcke verwenden die folgende Anazhl an Kernels pro Convolutional Layer: 64, 128, 256, 512 und nochmals 512. Eine schematische Darstellung des Netzwerks findet sich in Abbildung 2.2.

Im Rahmen der Evaluation ist vor allem die Verwendung der C3D-Features zusammen mit einem Klassifikator in Form einer Support Vector Machine (SVM) mit linearem Kernel von Interesse – im Folgenden wird das Vorgehen zur Aktionserkennung auf dem UCF101-Datensatz [SZS12] näher erläutert [TBF+15]: Hierzu werden C3D-Features als Aktivierungen der ersten Fully Connected Layer extrahiert, welche dann im Anschluss einer L_2-Normalisierung unterzogen werden. Dabei wird immer ein Block aus 16 aufeinanderfolgende Bildern verarbeitet. Um C3D-Features für ein ganzes Video zu extrahieren, wird mit den ersten 16 Bildern begonnen, anschließend wird ein Schritt mit einer Schrittweite von 8 Bildern unternommen, bevor wieder extrahiert wird – es kommt zu 50% Überlappung. Vor der Anwendung der L_2-Normalisierung wird der Durchschnitt über alle extrahierten Aktivierungen gebildet, so entsteht ein Vektor pro Video. Für die Bestimmung von Aktionen wird auf Basis der C3D-Features anschließend eine lineare SVM trainiert; das C3D-Netzwerk wird auf anderen Datensätzen vortrainiert, es findet auch kein Finetuning auf UCF101 statt. Insgesamt werden drei Netze vortrainiert, eines auf dem nicht veröffentlichten I380K-Datensatz, eines auf dem Sports-1M-Datensatz [KTS+14] und eines auf dem I380K-Datensatz mit anschließendem Finetuning auf dem Sports-1M-Datensatz [TBF+15]. Letzteres funktioniert im Rahmen der Tests für die Genauigkeit am besten. Für die C3D-Features dieses Netzes wurde eine lineare SVM trainiert; auch wurden die C3D-Features aller drei Varianten zu einem einzigen Featurevektor der Größe $12288 = 3 \cdot 4096$ zusammengefasst und eine weitere lineare SVM wurde auf Basis dieser Featurevektoren

trainiert. Die Ergebnisse auf dem UCF101-Datensatz können Tabelle 2.1 entnommen werden. Im Vergleich mit den anderen getesteten Methoden, die lediglich RGB-Bilder als Eingabe verwenden, erzeugt C3D die besten Ergebnisse, obwohl C3D selbst nicht auf UCF101 trainiert wurde und lediglich ein linearer Klassifizierer verwendet wurde.

Methode	Genauigkeit
ImageNet + lineare SVM	68.8%
iDT w/ BoW + lineare SVM	76.2%
Deep Networks	65.4%
Spatial Stream Network	72.6%
LRCN	71.1%
LSTM Composite Model	75.8%
C3D (1 Netz) + lineare SVM	**82.3**%
C3D (3 Netze) + lineare SVM	**85.2**%
iDT w/ Fisher Vector	87.9%
Temporal Stream Network	83.7%
Two-Stream Networks	88.0%
LRCN	82.9%
LSTM Composite Model	84.3%
Conv. Pooling on long clips	88.2%
LSTM on long clips	88.6%
Multi-Skip Feature Stacking	89.1%
C3D (3 Netze) + iDT + lineare SVM	**90.4**%

Tabelle 2.1.: Resultate von C3D bei der Aktionserkennung auf dem UCF101-Datensatz [TBF+15]: Der mittlere Abschnitt enthält nur Verfahren, die genau wie C3D nur RGB-Bilder als Eingabe verwenden, folglich vom Informationsgehalt der Eingabedaten vergleichbar sind. Im unteren Abschnitt sind Verfahren zu finden, die sich auf zusätzliche oder andere Eingaben stützen, wie beispielsweise die Two-Stream Networks, die zusätzlich zu den RGB-Daten noch den optischen Fluss verwenden. Der erste Abschnitt hingegen fungiert nur als eine einfache Baseline zu Vergleichzwecken. Tabelle aus [TBF+15].

2.2. SST: Single-Stream Temporal Action Proposals

In ihrer Arbeit „SST: Single-Stream Temporal Action Proposals“ [BES+17] befassen sich Buch et al. mit der Generierung sogenannter Temporal Action Proposals als Basis der Temporal Action Localization. Temporal Action Proposals sind dabei Zeitfenster in einem Video, von denen angenommen wird, dass in ihnen mit hoher Wahrscheinlichkeit eine Aktion stattfindet. Zusätzlich sollen sie möglichst genau der zeitlichen Ausdehnung der enthaltenen Aktion entsprechen – im Idealfall beginnt und endet das Zeitfenster dort, wo auch die Aktion beginnt und endet. Diese Zeitfenster werden dann als Basis für die Temporal Action Localization weiterverwendet. Zur Erzeugung der Temporal Action Proposals haben Buch et al. ein Recurrent Neural Network entworfen, das ein Video in einem einzelnen Durchlauf verarbeitet und dabei Temporal Action Proposals erzeugt, ohne dass dabei einzelne Bilder des Videos wiederholt betrachtet werden müssen. Das entworfene Recurrent Neural Network wird im Folgenden als SST-Netzwerk bezeichnet.

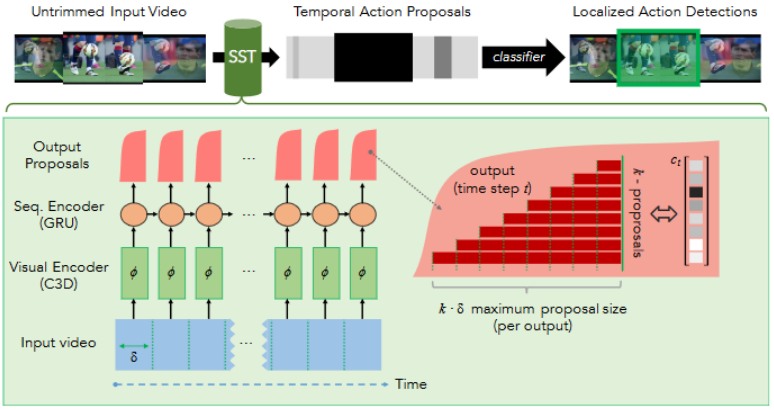


Abbildung 2.3.: Architektur des SST-Netzwerks [BES+17]: Ein ungekürztes Eingabevideo wird mit Hilfe der dreistufigen Architektur aus Visual Encoder, Sequence Encoder und Ausgabemodul verabeitet, Vorschläge für Zeitfenster mit jeweils zugehörigem Konfidenzwert werden ausgegeben. Abbildung aus [BES+17].

Bei der Erzeugung von Temporal Action Proposals sollen zeitliche Abschnitte in langen und ungekürzten Videos gefunden werden, die Aktionen enthalten. Als ungekürzte Videos werden hierbei Videos angesehen, die Abschnitte mit Aktionen und Abschnitte ohne Aktionen enthalten. Ein solches Video kann beispielsweise so aussehen: Ein Stadion wird von außen gezeigt (keine Aktion), dann werden Menschen im Stadion gezeigt, die Fußball spielen (Aktion), zwischenzeitlich werden wiederholt Statistiken zum Spiel eingeblendet (keine Aktion). Es gilt dann die zeitlichen Abschnitte zu identifizieren, in denen Fußball gespielt wird. Vor der Erfindung von Temporal Action Proposals mussten Sliding-Window-Verfahren eingesetzt werden, bei denen Zeitfenster verschiedener Größe über das Video verschoben wurden. Alle Stellen, an die sie geschoben wurden, fungierten dann als Grundlage für die Temporal Action Localization, indem beispielsweise mittels eines Klassifizierers bestimmt wurde, ob sie eine Aktion enthalten oder nicht. Mit diesem Ansatz müssen zahlreiche Zeitfenster betrachtet werden. Es wäre daher wünschenswert, die Anzahl von Zeitfenstern, die zur Klassifizierung betrachtet werden müssen, zu verringern. Dieses Ziel kann durch den Einsatz von Temporal Action Proposals erreicht werden. Dieses Vorgehen kann man als analog zu dem Prinzip der Kaskadierung beim Boosting, wie es von Viola & Jones [VJ01] verwendet wurde, betrachten: Erst werden irrelevante zeitliche Segmente verworfen, das vergleichsweise aufwendige Klassifizierungsverfahren schließt sich dann nur noch auf vielversprechenden zeitlichen Segmenten an – so wie beim Boosting nach und nach komplexere Klassifikatoren verwendet werden. Im Rahmen der Arbeit von Buch et al. sollen vielversprechende Temporal Action Proposals generiert werden – dabei wird erreicht, dass im Rahmen der Erzeugung kein Bild der zu verarbeitenden Videosequenz doppelt betrachtet werden muss, wie es bei vorangegangenen Arbeiten der Fall war [BES+17].

Das SST-Netzwerk hat folgende dreistufige Architektur: Zuerst wird ein Visual Encoder verwendet, auf dem ein Sequence Encoder aufbaut, welcher in einem Ausgabemodul endet. Eine schematische Darstellung der Architektur kann Abbildung 2.3 entnommen werden. Als Eingabe erhält das SST-Netzwerk ein ungekürztes Video. Im Folgenden schließen sich

die einzelnen Module an, die hier näher erläutert werden [BES+17]:

Visual Encoder: Bei diesem Teil der Architektur handelt es sich lediglich um ein C3D-Netzwerk, welches in Abschnitt 2.1 näher erläutert wurde. Im Rahmen des SST-Netzwerks wird es dazu verwendet, für jeweils 16 zusammenhängende Bilder eines Eingabevideos einen 4096-elementigen Vektor zu extrahieren, der diese beschreibt. Dieser Vektor entspricht hier im Gegensatz zu Abschnitt 2.1 der Ausgabe der zweiten Fully Connected Layer des C3D-Netzes ohne $L2$-Normalisiserung; auf diese Weise extrahierte Vektoren werden in Zukunft als C3D-Features bezeichnet. Vektoren werden hierbei ohne Überlappung bei den Bildern extrahiert, so beispielsweise für Bild 1-16 und 17-32; dies setzt sich fort, bis keine vollen 16 Bilder mehr im Video enthalten sind. Die verwendete Schrittweite entspricht hier folglich 16 Bildern. Buch et al. geben in ihrer Arbeit an, dass das C3D-Netzwerk mit den Gewichten, die Tran et al. [TBF+15] bestimmt haben, initialisiert wird. Es wird keine eindeutige Aussage darüber getroffen, ob das C3D-Netzwerk mit dem Rest des SST-Netzwerks zusammen trainiert wird; Tests mit der via Github bereitgestellten Implementierung[1] zeigen, dass das C3D-Netzwerk nach der Initialisierung nicht weiter trainiert wurde. Die durch das C3D-Netzwerk erhaltenen C3D-Features für jeweils 16 Bilder werden abschließend einer Hauptkomponentenanalyse unterzogen, um die Größe des jeweiligen Vektors zu reduzieren.

Sequence Encoder: Dieser Teil der Architektur dient dazu, zeitliche Zusammenhänge zwischen den einzelnen aufeinanderfolgenden C3D-Featurevektoren zu erkennen und auf dieser Basis zu bestimmen, ob in bestimmten Zeitfenstern eine Aktion stattfindet oder nicht. Hierzu wird ein Recurrent Neural Network eingesetzt, das Gated Recurrent Units (GRUs) verwendet; Long Short-Term Memory als bekannte Alternative wurde experimentell als leicht unterlegen eingestuft. Die GRUs liefern die Eingabe für das sich anschließende Ausgabemodul – zu jedem Zeitschritt berechnen die GRUs des Sequence Encoders auf Basis des aktuell eingegebenen C3D-Featurevektors einen neuen verborgenen Zustand; der verborgene Zustand der letzten Schicht von GRUs wird anschließend ausgelesen und an das Ausgabemodul weitergeleitet.

Ausgabemodul: Dieser abschließende Teil der Architektur dient dazu, den verborgenen Zustand aus der letzten Schicht des vorgeschalteten Sequence Encoders in Konfidenzwerte für Temporal Action Proposals umzuwandeln. Dabei werden pro Zeitschritt Konfidenzwerte für k Zeitfenster berechnet; diese sehen wie folgt aus: Es wird mit einem Zeitfenster begonnen, das die 16 Bilder der aktuellen Eingabe umfasst, dieses definiert das erste der k Zeitfenster; anschließend wird das Zeitfenster jeweils so erweitert, dass es zusätzlich die 16 vorangegangenen Bilder umfasst, bis auf diese Weise k Zeitfenster für die Ausgabe definiert sind. Die so betrachteten Zeitfenster erspannen sich über $16, 32, ..., k \cdot$ 16 Bilder. Um für diese Zeitfenster die zugehörigen Konfidenzwerte zu bestimmen, wird eine Fully Connected Layer mit logitischer Sigmoidfunktion als Aktivierungsfunktion verwendet. Um abschließend die Zahl der Zeitfenster zu reduzieren und nur vielversprechende Zeitfenster zu behalten, schließen sich Nachbearbeitungsschritte an. So werden beispielsweise mit der Non-Maxima-Suppression Ergebnisse entfernt, die in ihrer Umgebung nicht den höchsten Konfidenzwert haben, und ein Schwellwert sorgt dafür, dass Zeitfenster, die eine geringen Konfidenzwert aufweisen, verworfen werden.

Da das SST-Netzwerk später auf langen, ungekürzten Videosequenzen arbeiten soll, wird es während des Trainings mit Eingaben konfrontiert, die deutlich länger als die maximale

[1] `https://github.com/shyamal-b/sst`

Länge $k \cdot 16$ eines Zeitfensters für die Temporal Action Proposals, die generiert werden, sind. Darüber hinaus werden diese Eingaben zu Trainingszwecken überlappend erzeugt. Man erhofft sich damit, dass eine Sättigung des verborgenen Zustands des eingesetzten Recurrent Neural Networks vermieden wird, wie sie in vorangegangenen Arbeiten bei langen Eingaben aufgetreten ist [BES+17]. Durch die zusätzliche Überlappung der Eingaben kann darüber hinaus während des Trainings erreicht werden, dass Abschnitte des Videos mit verschiedenem Kontext betrachtet werden können. Um das Fußball-Beispiel wieder aufzugreifen: Eine Trainingssequenz könnte zuerst das Stadion und anschließend das eigentliche Fußballspiel zeigen – eine weitere Trainingssequenz könnte denselben Teil des Fußballspiels zeigen und anschließend Zwischenergebnisse, die eingeblendet werden. In beiden Fällen wird die Aktion Fußball spielen mit verschiedenem Kontext gezeigt. Um das Training mit Hilfe des Adam-Algorithmus und Backpropagation zu ermöglichen, muss darüber hinaus noch geklärt werden, ab wann ein Temporal Action Proposal, das durch das SST-Netzwerk generiert wird, bezüglich der annotierten Zeitintervalle in den Trainingsdaten als korrekt gilt. Hierzu wird die temporal Intersection over Union, kurz tIoU, verwendet: Die zeitliche Überlappung zweier Zeitfenster in Relation zu ihrer Vereinigung. Beträgt die tIoU bezüglich eines annotierten Zeitintervalls der Trainingsdaten mindestens 50%, so wird angenommen, dass das Temporal Action Proposals als korrekt detektiert werden muss, ansonsten als falsch. Der Fehler des Netzwerkes, der für das Training notwendig ist, wird dann mit Hilfe der gewichteten binären Kreuzentropie zwischen erwarteter Ausgabe und tatsächlicher Ausgabe bestimmt. Die Gewichte werden dazu eingesetzt, die unterschiedlichen Häufigkeiten des Auftretens positiver und negativer Beispiele für die unterschiedlichen Längen der Temporal Action Proposals auszugleichen. Darüber hinaus werden sowohl Dropout als auch L_2-Regularisierung während des Trainings eingesetzt. Trainiert wurde auf dem THUMOS'14-Validierungsdatensatz [JLZ+14] mit einer Aufteilung in 80% Trainingsdaten und 20% Validierungsdaten.

Da es die Aufgabe des SST-Netzwerks ist, Zeitfenster zu finden, die mit hoher Wahrscheinlichkeit Aktionen enthalten und diese zeitlich möglichst genau eingrenzen, wird die Metrik Recall zur Evaluation verwendet – sie gibt in diesem Fall den Prozentsatz der annotierten Zeitfenster für Aktionen an, die durch das SST-Netzwerk gefunden wurden. Ein annotiertes Zeitfenster gilt dabei als gefunden, wenn das SST-Netzwerk nach der Nachverarbeitung ein entsprechendes Temporal Action Proposal ausgibt, welches mindestens eine gewisse tIoU mit dem annotierten Zeitfenster aufweist. Mit Hilfe des Recalls werden zwei verschiedene Evaluationen durchgeführt: Zum einen wird der durchschnittliche Recall bezüglich diverser tIoU-Schwellwerte innerhalb eines Intervalls in Relation zur durchschnittlich Anzahl der betrachteten Temporal Action Proposals bestimmt, zum anderen wird der Recall bei fester durchschnittlicher Anzahl von 1000 Temporal Action Proposals pro Video in Relation zur tIoU bestimmt. Ergebnisse diverser Tests können Abbildung 2.4 entnommen werden. Die verwendeten Schwellwerte für die tIoU entstammen dabei im linken Bild dem Intervall $[0.5, 1.0]$ und im mittleren Bild dem Intervall $[0.7, 0.95]$ [BES+17]. Wie zu sehen ist, gelingt es dem SST-Netzwerk, sich im Rahmen der Evaluation des durchschnittlichen Recalls in Relation zur durchschnittlichen Anzahl an Proposals gegen alle anderen getesteten Methoden durchzusetzen. Bei der Evaluation des Recalls bei durchschnittlich 1000 Temporal Action Proposals pro Video in Relation zur tIoU schlägt sich das SST-Netzwerk im niedrigen tIoU-Bereich etwas schlechter als das SCNN-prop-Verfahren, schneidet dafür jedoch bei hoher tIoU deutlich besser ab als die Verfahren, mit denen verglichen wird.

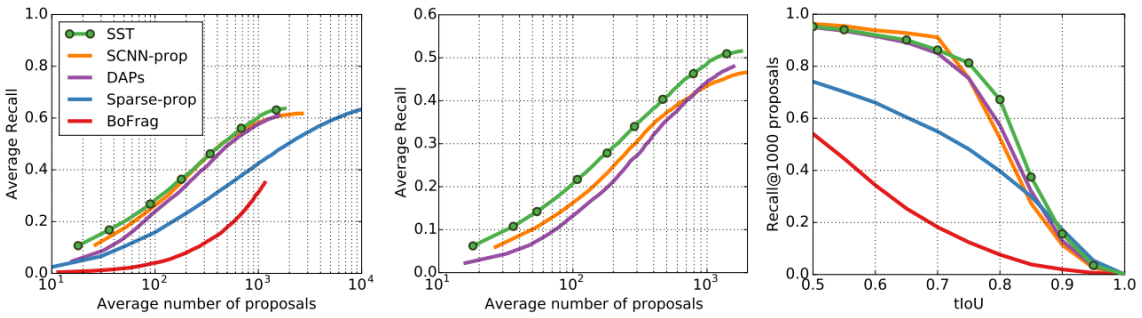


Abbildung 2.4.: Evaluation des SST-Netzwerks bezüglich des Recalls [BES+17]: Links und in der Mitte wird der durchschnittliche Recall bezüglich der durchschnittlichen Anzahl an Temporal Action Proposals pro Video evaluiert. In beiden Fällen muss ein Intervall gewählt werden, über das der Durchschnitt des Recalls bezüglich mehrerer tIoU-Schwellwerte gebildet wird – diese Schwellwerte entstammen dem Intervall $[0.5, 1.0]$ bzw. $[0.7, 0.95]$. Rechts wird der Recall bei durchschnittlich 1000 Temporal Action Proposals pro Video in Relation zur tIoU bestimmt – hier schlägt sich das SST-Netzwerk bei hoher tIoU besser als die Verfahren, mit denen verglichen wird. Abbildung aus [BES+17].

Über die hier vorgestellten Ergebnisse hinaus wurden noch weitere Auswertungen im Rahmen der Arbeit von Buch et al. [BES+17] durchgeführt und können bei Interesse dort nachgeschlagen werden; sie sind jedoch für diese Arbeit nicht relevant.

2.3. On the Integration of Optical Flow and Action Recognition

Im Rahmen ihrer Arbeit „On the Integration of Optical Flow and Action Recognition“ [SLLG+17] untersuchen Sevilla-Lara et al. den Einfluss des optischen Flusses bei der Aktionserkennung. Es wird näher untersucht, ob und warum optischer Fluss für die Aktionserkennung nützlich ist; darüber hinaus wird darauf eingegangen, wie optischer Fluss für die Aktionserkennung optimiert werden kann.

Zur Untersuchung des optischen Flusses wurde ein Temporal Segment Network (TSN) [WXW+16] verwendet; es handelt es sich um ein Two-Stream Convolutional Neural Network. Als mögliche Eingaben fungieren beispielsweise RGB-Bilder und der zugehörige optische Fluss; zur Auswertung wird der gewichtete Durchschnitt der Streams verwendet. Dies ermöglicht es, auch die einzelnen Streams auszuwerten und so beispielsweise die Klassifikationsgenauigkeit des Teilnetzwerks für den optischen Fluss alleine zu bestimmen. In einer ersten Untersuchung wurden die Teilnetzwerke für die RGB-Daten und für den optischen Fluss separat betrachtet. Die Klassifikationsgenauigkeit für RGB-Daten und den zugehörigen optischen Fluss wurden dabei jeweils separat für den UCF101-Datensatz bestimmt, hierbei zeigte das Teilnetzwerk für den optische Fluss eine um etwa 1.4% höhere Klassifikationsgenauigkeit.

Da der optische Fluss die Bewegungsinformationen zwischen Bildern enthält, läge die Vermutung nahe, dass der Hauptwert des optischen Flusses in der abgebildeten Bewegung liegt. Sevilla-Lara et al. kommen in ihrer Arbeit jedoch zu dem Schluss, dass der Hauptwert des optischen Flusses an einer anderen Stelle liegt, nämlich bei der Eigenschaft, dass

Abbildung 2.5.: Beispiele für Farbänderungen, wie sie für diverse Experimente vorgenommen wurden [SLLG+17]: Links: Originalbild, Mitte: zufällig gewichtete Farbkanäle, Rechts: geänderte Colormap. Abbildung aus [SLLG+17].

von der farblichen Darstellung in Bildern abstrahiert wird. Unterscheiden sich beispielsweise zwei Objekte weder in ihrer Form noch in ihrer Bewegung, sondern lediglich in ihrer Farbgebung, erzeugen beide denselben optischen Fluss. Hierdurch reduzieren sich die Möglichkeiten, die erlernt werden müssen, entsprechend leichter fällt der Lernprozess aus [SLLG+17]. Die Grundlage für die Schlussfolgerung von Sevilla-Lara et al. bilden die Experimente, die im Rahmen ihrer Arbeit durchgeführt wurden: Es wurde zuerst versucht, die Informationen über den zeitlichen Verlauf von Bewegungen aus dem optischen Fluss zu eliminieren. Das TSN verarbeitet im Teilnetz für den optischen Fluss jeweils den zu sechs Bildern gehörenden optischen Fluss – folglich fünf Flussfelder. Diese wurden im Rahmen des ersten Experiments zufällig gemischt. Hierdurch gehen die Informationen über den zeitlichen Zusammenhang der Bewegungen der einzelnen Flussfelder verloren. In einem zweiten Versuch wurden darüber hinaus die Bilder vor der Berechnung des optischen Flusses gemischt, sodass nicht einmal mehr die einzelnen Flussfelder tatsächliche Bewegungen abbilden. Entsprechend sank die Genauigkeit bei der Aktionserkennung auf dem UCF101-Datensatz im ersten Fall auf 78.68% und im zweiten Fall auf 59.55%. Gemessen an der Tatsache, dass keine tatsächlichen Bewegungen mehr abgebildet werden, ist dieses Ergebnis immer noch gut – würde man jede Klasse zufällig wählen, wäre eine Genauigkeit von etwa 1% zu erwarten [SLLG+17]. Im Rahmen weiterer Experimente wurde die Farbe der Eingabebilder einmal durch Einsatz einer geänderten Colormap manipuliert, ein anderes Mal durch eine zufällige Skalierung der einzelnen Farbkanäle (zwischen 0.3 und 1.0). Ein Bild vor und nach der jeweiligen Manipulation ist in Abbildung 2.5 zu sehen. Der optische Fluss wurde im Anschluss auf den manipulierten Bildern berechnet. Die Ergebnisse bezüglich der Klassifikationsgenauigkeit können Tabelle 2.2 entnommen werden.

Eingabe	Genauigkeit
Opt. Fluss	86.85%
RGB	85.43%
Opt. Fluss (gemischte Flussfelder)	78.64%
Opt. Fluss (gemischte Bilder)	59.55%
Opt. Fluss (geänderte Colormap)	84.30%
RGB (geänderte Colormap)	34.23%
Opt. Fluss (gewichtete Farbkanäle)	85.71%
RGB (gewichtete Farbkanäle)	62.65%

Tabelle 2.2.: Klassifikationsgenauigkeit des TSNs bezüglich verschiedener Eingabemodalitäten. Tabelle aus [SLLG+17].

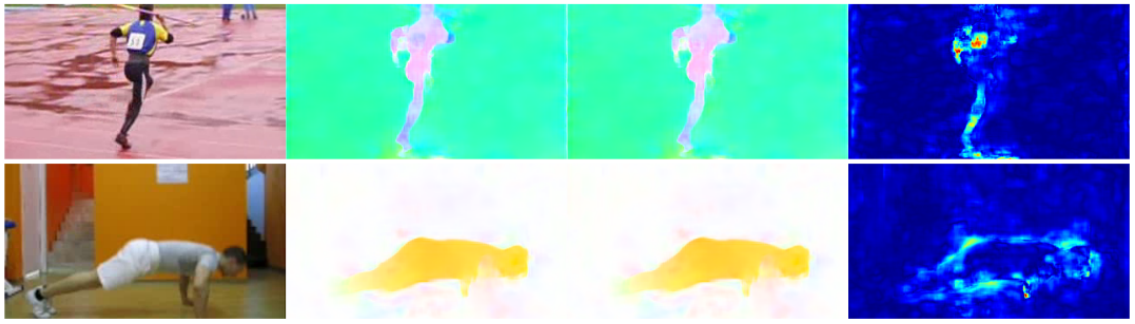


Abbildung 2.6.: Beispiele für optischen Fluss, der von Neuronalen Netzen mit unterschiedlicher Optimierung erzeugt wurde, sowie die Differenz der erzeugten Flussfelder [SLLG+17]. Dabei wird links zuerst eines der Bilder gezeigt, auf dessen Basis der optische Fluss berechnet wurde. Die beiden Bilder in der Mitte zeigen jeweils den optischen Fluss, der mit Hilfe der Netzwerke erzeugt wurde – auf einem Bild wurde das Netzwerk mit der EPE-Metrik optimiert, auf dem anderen mit dem Klassifikationsfehler. Ganz rechts wird der Unterschied zwischen den Flussfeldern mit Hilfe der euklidschen Distanz visualisiert. Abbildung aus [SLLG+17].

Die Klassifikationsgenauigkeit für das Teilnetz, das mit den RGB-Daten arbeitet, nimmt in Folge des Einsatzes der farblich veränderten Bilder stark ab, während die Klassifikationsgenauigkeit des Teilnetzes, das den optischen Fluss auf Basis der veränderten Bilder verwendet, nur minimale Einbußen aufweist. Dies gilt für beide Arten der Modifikation. Zusammen mit den vorherigen Ergebnissen führen diese Ergebnisse zur Schlussfolgerung von Sevilla-Lara et al., dass der Hauptwert des optischen Flusses in der Unabhängigkeit von der farblichen Repräsentation der Szene liegt.

Abgesehen von den Untersuchungen zur Nützlichkeit des optischen Flusses und deren Ursachen wurde der Frage nachgegangen, wie sich der optische Fluss für die Aktionserkennung optimieren lässt. Hierbei kommen die Autoren im Rahmen ihrer Arbeit zu dem Schluss, dass nur eine schwache Korrelation zwischen der klassischen Evaluationsmetrik EPE (End-Point-Error) für die Genauigkeit des optischen Flusses und dem Nutzen des optischen Flusses im Rahmen der Aktionserkennung besteht. Aus diesem Grund wurden Neuronale Netze zur Berechnung des optischen Flusses mit dem TSN in einem Ende-zu-Ende-Netzwerk gekoppelt, das es ermöglicht, die Neuronalen Netze mit dem Klassifikationsfehler des TSNs statt mit der klassischen EPE-Metrik zu trainieren. Auf diese Art trainierte Netze funktionieren für die Aktionsklassifikation etwas besser als auf der EPE-Metrik trainierte Netze; eine Verbesserung von bis zu 5% bezüglich der Klassifikationsgenauigkeit tritt ein. Eine Tabelle mit genauen Zahlen kann der Arbeit von Sevilla-Lara et al. [SLLG+17] entnommen werden. Besonders interessant ist die Visualisierung dessen, was die neu trainierten Netze zur Erzeugung des optischen Flusses an Stelle der Netze, die auf der EPE-Metrik trainiert wurden, gelernt haben. Die Ergebnisse sind in weiten Bereichen sehr ähnlich, Unterschiede finden sich vor allem an den Stellen, an denen sich Menschen befinden, und an Bewegungsgrenzen [SLLG+17]. Dies legt nahe, dass vor allem diese Bereiche im Rahmen der Aktionsklassifikation von besonderem Interesse sind. Beispiele für die Unterschiede werden in Abbildung 2.6 dargestellt.

Zusammenfassend sind vor allem folgende Punkte wichtig:

- Optischer Fluss ist nützlich im Rahmen der Aktionsklassifikation.
- Optischer Fluss abstrahiert von der farblichen Erscheinung.
- Optischer Fluss lässt sich für die Aktionserkennung verbessern, wenn er mit Hilfe eines Neuronalen Netzes erzeugt wird und dieses auf Basis des Klassifikationsfehlers trainiert wird.

3. Grundlagen

In diesem Kapitel werden Grundlagen für das weitere Vorgehen in dieser Arbeit vorgestellt. Dabei wird die Funktionsweise von künstlichen Neuronalen Netzen näher erläutert und verschiedene Architekturen von künstlichen Neuronalen Netzen werden vorgestellt. Im Anschluss findet eine Betrachtung des optischen Flusses, dem im Rahmen dieser Arbeit eine zentrale Funktion zukommt, statt.

3.1. Künstliche Neuronale Netze

In diesem Abschnitt wird näher auf künstliche Neuronale Netze eingegangen. Es wird ein Überblick über den Aufbau und die Funktionsweise von künstlichen Neuronalen Netzen geliefert; anschließend werden die einzelnen Teilkomponenten und Verfahren eingehend betrachtet. In weiten Teilen wird hierzu auf die Arbeiten von Goodfellow et al. [GBC16] und Stuart et al. [RN12] zurückgegriffen. Da in dieser Arbeit ausschließlich künstliche Neuronale Netze und künstliche Neuronen behandelt und verwendet werden, werden diese oftmals nur als Neuronale Netze bzw. Neuronen bezeichnet.

3.1.1. Aufbau und Funktionsweise

Um einen kurzen Überblick über den Aufbau und die Funktionsweise von Neuronalen Netzen zu erhalten, wird vornehmlich die Arbeit von Stuart et al. [RN12] herangezogen, auf die auch für eine detailierte Betrachtung verwiesen wird. Um den Aufbau von Neuronalen Netzen zu verstehen, wird zuerst die kleinste Einheit betrachtet, aus der sich Neuronale Netze zusammensetzen: ein einzelnes Neuron. In Abbildung 3.1 ist ein solches Neuron dargestellt. Als Eingabe erhält ein Neuron $n + 1$ Werte, wobei n dieser Werte von einem Eingabevektor $\mathbf{x}$ stammen. Im Folgenden bezeichnet x_i den i-ten Wert des Vektors $\mathbf{x}$. Dem Eingabevektor ist ein Gewichtsvektor $\mathbf{w}$ derselben Länge zugeordnet, wobei w_i das i-te Element dieses Vektors bezeichnet und dem Eingabewert x_i zugeordnet ist. Zusätzlich zur Multiplikation von Eingabevektor und Gewichtsvektor findet eine anschließende Addition mit dem sogenannten Bias b, teilweise auch als w_0 bezeichnet, statt – dieser ist die letzte fehlende Eingabe in das Neuronale Netz. Die Multiplikation des Eingabevektors

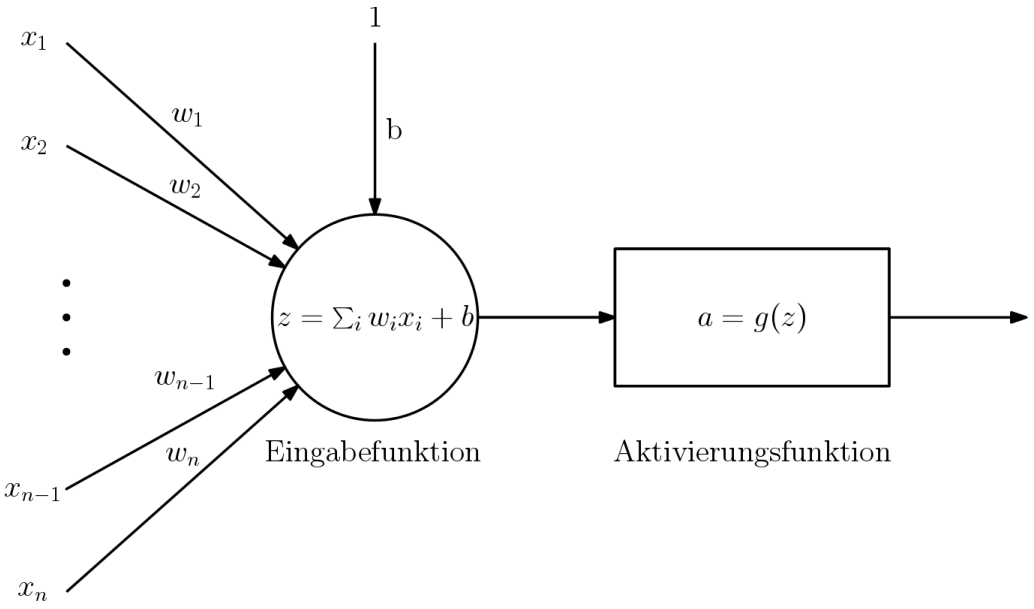


Abbildung 3.1.: Darstellung der Funktionsweise eines einzelnen künstlichen Neurons. Als Eingabe erhält das Neuron n Werte x_i, die mit den zugeordneten Gewichten w_i im Rahmen der Eingabefunktion eine gewichtete Summe bilden. Zusätzlich erhält es den Bias b als Eingabe, der auch im Rahmen der Eingabefunktion aufaddiert wird. Insgesamt erhält ein Neuron so $n + 1$ Eingaben. Die gebildete Summe z wird anschließend an die nichtlineare Aktivierungsfunktion g weitergereicht, die auf z angewendet die Aktivierung a erzeugt. Abbildung auf Basis von [Fan18].

.

$\mathbf{x}$ mit dem Gewichtsvektor $\mathbf{w}$ und die anschließende Addition des Bias b bildet die sogenannte Eingabefunktion, die als Ausgabe ein Skalar $z = \mathbf{w}^T\mathbf{x} + b$ erzeugt. Dieses dient anschließend als Eingabe für eine nichtlineare Aktivierungsfunktion g. Setzt man einzelne Neuronen zu einem Netzwerk zusammen, ist die Nichtlinearität der Aktivierungsfunktion besonders wichtig: Würde das Netzwerk lediglich lineare Aktivierungsfunktionen verwenden, wäre es nur in der Lage, lineare Funktionen darzustellen, denn die Komposition von linearen Funktionen ist nach Umformung wieder eine einzige lineare Funktion. Der Einsatz von nichtlinearen Aktivierungsfunktionen sorgt dafür, dass diese Einschränkung nicht gilt [RN12]. Die nichtlineare Aktivierungsfunktion des Neurons erzeugt die Ausgabe a, bei der es sich wie bereits bei z um ein Skalar handelt. Der Ausgabewert a wird auch als Aktivierung bezeichnet.

Nachdem die „Bausteine“ von Neuronalen Netzen nun vorgestellt wurden, wird erklärt, wie sich ein Netz aus ihnen zusammensetzt. Bei der klassischen Form von Neuronalen Netzen, den sogenannten *Feedforward-Netzen*, wird die Eingabe in das Neuronale Netz durch diverse Neuronen zur Ausgabe weitergereicht, ohne dass dabei Zyklen vorkommen. Die gängigste Variante solcher Netze sieht vor, Neuronen in Schichten (engl. *Layers*) zusammenzufassen und mehrere dieser Schichten nacheinander anzuordnen. Dabei ist jedes Neuron der Schicht i mit bis zu allen Neuronen der Schicht $i - 1$ und $i + 1$ verbunden. Von allen verbundenen Neuronen der Schicht $i - 1$ erhält es die jeweilige Aktivierung als Eingabe, während es selbst seine Aktivierung an alle verbundenen Neuronen der Schicht

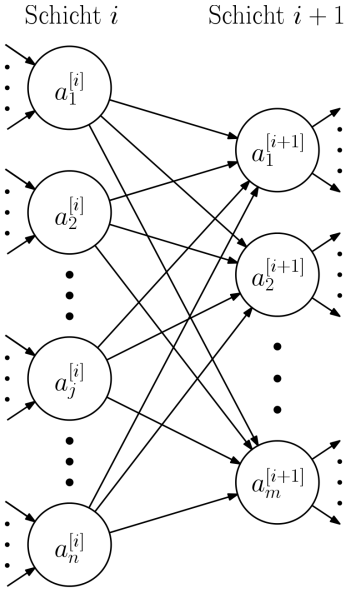


Abbildung 3.2.: Darstellung der Verbindung zwischen zwei Schichten eines Neuronalen Netzes. Schicht $i+1$ stellt eine Fully Connected Layer dar, da jedes Neuron dieser Schicht alle Aktivierungen der Neuronen der vorangegangenen Schicht i als Eingabe erhält.

$i+1$ weiterleitet [GBC16]. Erhält jedes Neuron einer Schicht alle Aktivierungen der vorangegangenen Schicht, spricht man von einer *Fully Connected Layer*. Alternativen hierzu werden zu einem späteren Zeitpunkt vorgestellt. In Abbildung 3.2 wird der Aufbau und die Verknüpfung von zwei Schichten exemplarisch dargestellt. Im Folgenden sind noch die Schichten für Eingabe und Ausgabe von Daten zu betrachten. Die Schicht für die Eingabe von Daten wird als Eingabeschicht (engl. *Input Layer*) bezeichnet, die Schicht zur Ausgabe von Daten als Ausgabeschicht (engl. *Output Layer*). Die Eingabeschicht besteht aus einer beliebigen Anzahl an Einheiten, die jeweils einen skalaren Wert des Eingabevektors ausgeben und an die Neuronen der nächsten Schicht weiterleiten. Die Größe des Eingabevektors muss dabei der Anzahl an Einheiten der Eingabeschicht entsprechen. Die Ausgabeschicht besteht aus einer oder mehreren Ausgabeeinheiten, die die Ausgabe des Netzes produzieren, beispielsweise eine Wahrscheinlichkeitsverteilung. Eingabe- und Ausgabeschicht sind für die Interaktion nach außen gedacht, während bei den Schichten dazwischen keine Interaktion außerhalb des Neuronalen Netzes vorgesehen ist. Diese Schichten im Inneren des Neuronalen Netzes werden als verborgene Schichten (engl. *Hidden Layers*) bezeichnet, ihre zugehörigen Neuronen als verborgene Einheiten (engl. *Hidden Units*).

Abschließend ist noch zu klären, wie die Gewichte und Biase des Neuronalen Netzes gelernt werden können. Ziel des Lernvorgangs ist es, dass das Neuronale Netz die gewünschte Ausgabe erzeugt, beispielsweise korrekt entscheidet, ob auf einem Bild eine Katze zu sehen ist oder nicht. Damit das Neuronale Netz lernen kann, sind Trainingsbeispiele nötig. Im Rahmen des überwachten Lernens müssen diese mit der Zielausgabe des Neuronalen Netzes annotiert sein. Es exisiteren auch unüberwachte Lernverfahren, die ohne Annotation auskommen, diese sind jedoch im Rahmen dieser Arbeit nicht relevant. Die jeweilige Zielausgabe, die den Trainingsbeispielen zugeordnet ist, beschreibt die gewünschte Ausgabe des Neuronalen Netzes; es kann sich beispielsweise um einen binären Wert handeln. Um

zu lernen, wird die Ausgabe des Neuronalen Netzes für die Trainingsbeispiele bestimmt und mit der Zielausgabe verglichen. Der Fehler der Ausgabe des Netzes im Bezug zur Zielausgabe wird bestimmt; hierzu kommt eine Loss-Funktion (engl. *Loss Function*) zum Einsatz, die durch einen Vergleich zwischen tatsächlicher Ausgabe und annotierter Zielausgabe den Fehler bestimmt. Im Anschluss wird der Wert der Loss-Funktion dazu verwendet, die Gewichte und die Biase im Neuronalen Netz anzupassen, indem er von der Ausgabeschicht durch die verborgenen Schichten bis hin zur Eingabeschicht zurückgeführt wird. Im Folgenden werden die einzelnen Teilkomponenten von Neuronalen Netzen, verwendete Funktionen und weitere Netzarchitekturen näher erläutert.

3.1.2. Eingabefunktion eines Neurons

Bei der gängigen Eingabefunktion $f(\mathbf{x}; \mathbf{w}, b)$ handelt es sich um den parametrischen Teil eines jeden Neurons, deren Parameter der Gewichtsvektor $\mathbf{w}$ und der Bias b sind. Zusammen mit der Eingabe $\mathbf{x}$ erzeugt die Eingabefunktion die Ausgabe z. Die Gewichtsvektoren $\mathbf{w}$ und Biase b enthalten sozusagen das *Wissen* des Neuronalen Netzes. In Neuronalen Netzen, wie sie im vorigen Abschnitt vorgestellt wurden, treten erlernbare Parameter nur im Rahmen der Eingabefunktionen der Neuronen als Gewichte und Biase auf [GBC16]. Für die Eingabefunktion gilt folgende Gleichung [GBC16]:

$$z = f(\mathbf{x}; \mathbf{w}, b) = \mathbf{w}^T \mathbf{x} + b \tag{3.1}$$

Hierbei handelt es sich um eine lineare Funktion. Ein Neuron stellt also ein lineares Modell dar, wenn sich keine nichtlineare Aktivierungsfunktion an die Eingabefunktion anschließt.

Es stellt sich nun die Frage, warum man ein Neuronales Netz nicht aus Neuronen aufbauen kann, die nur über die lineare Eingabefunktion verfügen, ohne dass sich eine nichtlineare Aktivierungsfunktion anschließt. Seien $f^{(1)}$ und $f^{(2)}$ die zu zwei Neuronen gehörenden lineare Eingabefunktionen und wie in Gleichung 3.1 definiert. Man nehme nun an, dass diese Neuronen in einem Netz hinereinandergeschaltet sind; $f^{(2)}$ erhält also die Ausgabe von $f^{(1)}$ als Eingabe. Der Bias wird im Folgenden der Einfachheit halber nicht mitbetrachtet. Mit der Eingabe $\mathbf{x}$ ergibt sich dann folgende Formel [GBC16]:

$$f^{(2)}(f^{(1)}(\mathbf{x})) = \mathbf{w}^{(2)T}(\mathbf{w}^{(1)T}\mathbf{x}) \tag{3.2}$$

Da die Multiplikation von Vektoren wie bei Matritzen assoziativ ist, können die Vektoren $\mathbf{w}_{(2)}^T$ und $\mathbf{w}_{(1)}^T$ zu einem neuen Vektor zusammengefasst werden[GBC16]:

$$\mathbf{w}^{(2)T}(\mathbf{w}^{(1)T}\mathbf{x}) = (\mathbf{w}^{(2)T}\mathbf{w}^{(1)T})\mathbf{x} = \mathbf{w}'^{T}x = f'(\mathbf{x}) \tag{3.3}$$

Aus Betrachtung der Formeln 3.2 und 3.3 ergibt sich, dass die Verkettung der linearen Eingabefunktion $f^{(1)}$ und $f^{(2)}$ lediglich einer weiteren linearen Eingabefunktion $f'(\mathbf{x})$ entspricht. Es kann folglich durch die Verkettung nicht mehr erreicht werden als durch eine einzelne lineare Eingabefunktion; Neuronale Netze, die nur lineare Eingabefunktionen verwenden, können folglich in ihrer Gesamtheit nur lineare Funktionen realisieren, da sich die gesamte Funktionalität durch eine einzelne lineare Eingabefunktion darstellen lässt. Werden nichtlineare Funktionen verwendet, gilt diese Regel nicht mehr. Daher schließt sich in der Praxis an eine lineare Eingabefunktion eine nichtlineare Aktivierungsfunktion an.

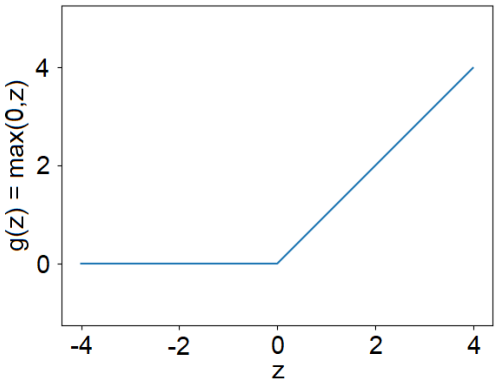


Abbildung 3.3.: Verlauf der ReLU-Funktion. Sie setzt sich aus den linearen Funktionen $g(z) = 0$ für $z \leq 0$ und $g(z) = z$ für $z > 0$ zusammen, mit einer Unstetigkeit an Stelle $z = 0$.

3.1.3. Aktivierungsfunktion eines Neurons

Bei der Aktivierungsfunktion handelt es sich in der Regel um eine nichtlineare Funktion, die sich in einem Neuron an die Eingabefunktion anschließt und auf deren Ausgabe angewandt wird. Sie erzeugt die Aktivierung eines Neurons, die dann im nächsten Schritt an die Neuronen der nachfolgenden Schicht des Neuronalen Netzes weitergeleitet wird. In Abschnitt 3.1.2 wurde bereits erläutert, dass es wenig Sinn ergibt, ein neuronales Netz aus mehreren Schichten zusammenzusetzen, die nur über lineare Funktionen verfügen. Um mehrere Schichten sinnvoll einzusetzen, müssen sie folglich durch eine nichtlineare Funktion getrennt werden. Dies ist die Aufgabe der Aktivierungsfunktion. Neben der Nichtlinearität der Funktion ist es vor allem wichtig, dass sie differenzierbar ist, damit auf Basis von Gradienten gelernt werden kann; fehlende Differenzierbarkeit in vereinzelten Punkten ist jedoch in der Praxis möglich und kann funktionieren [GBC16]. Im Folgenden werden einige der gängigsten Aktivierungsfunktionen vorgestellt.

3.1.3.1. Rectified Linear Unit (ReLU)

Bei der Rectified Linear Unit, kurz ReLU, handelt es sich um die jüngst am häufigsten eingesetzte verborgenen Einheit. Durch ihre Verwendung wird die zugehörige nichtlineare Aktivierungsfunktion g, die im Folgenden als ReLU-Funktion bezeichnet wird, mit Eingabe z wie folgt definiert [GBC16]:

$$g(z) = max(0, z) \tag{3.4}$$

Die ReLU-Funktion liefert also 0, falls die Eingabe kleiner oder gleich 0 ist, während sie die Eingabe unverändert zurückgibt, falls sie größer 0 ist. Veranschaulicht wird die ReLU-Funktion in Abbildung 3.3. Zu beachten ist die Unstetigkeit der Funktion an der Stelle $z = 0$. Dies widerspricht der Forderung nach Differenzierbarkeit, die zur Berechnung der Gradienten nötig ist, welche wiederum von Lernverfahren benötigt werden. In der Praxis funktioniert die Rectified Linear Unit dennoch gut. Als Eingabe erhält die Aktivierungsfunktion reelle Zahlen; folglich ist die Wahrscheinlichkeit, dass die Eingabe exakt 0 entspricht, sehr gering. Sollte die Stelle dennoch getroffen werden, kann die Ableitung

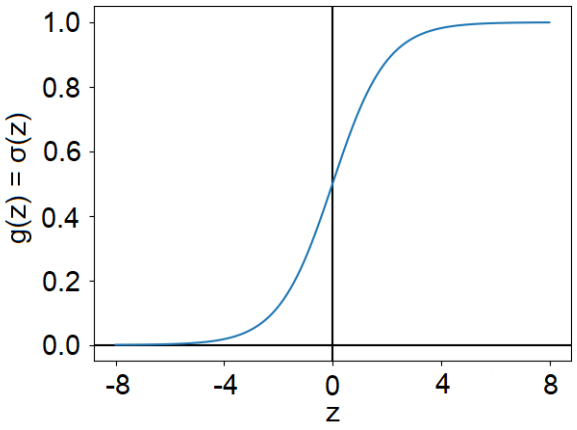


Abbildung 3.4.: Verlauf der logistischen Sigmoidfunktion. Für große und kleine Werte nähert sich die Steigung der Funktion 0 an; nur in einem schmalen Abschnitt verfügt die Funktion über eine starke Steigung.

manuell gewählt werden, ohne sie zu berechnen – entweder als Ableitung an der Stelle minimal kleiner 0 oder minimal größer 0. Der große Vorteil der ReLU-Funktion ist, dass sie aus zwei linearen Funktionen zusammengesetzt ist; folglich kann die Ableitung zwei konstante Werte annehmen. Für Eingaben kleiner 0 ist sie 0 und für Eingaben größer 0 ist sie 1:

$$g'(z) = \begin{cases} 0 & \text{für } z < 0 \\ 1 & \text{für } z > 0 \\ \text{undefiniert} & \text{für } z = 0 \end{cases} \tag{3.5}$$

Für $z = 0$ muss in der Praxis wie bereits erwähnt ein Wert von Hand gewählt werden, da die Funktion an dieser Stelle nicht ableitbar ist; hier kann entweder 0 oder 1 gewählt werden. Da die Ableitungen konstant sind, muss keine Funktion berechnet werden, um sie zu erhalten; dies spart Rechenzeit im Vergleich zu Funktionen, die nicht über diese Eigenschaft verfügen.

3.1.3.2. Logistische Sigmoidfunktion

Bevor die ReLU-Funktion bei Neuronalen Netzen eingesetzt wurde, war die logistische Sigmoidfunktion eine der beiden gängigsten Aktivierungsfunktionen. Sie wird mit dem Symbol σ bezeichnet und ist auf der Eingabe z ist wie folgt definiert [RN12]:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.6}$$

Folglich gilt für die nichtlineare Aktivierungsfunktion: $g(z) = \sigma(z)$. Dargestellt wird der Verlauf der Funktion in Abbildung 3.4.

Im Gegensatz zur ReLU-Funktion ist die logistische Sigmoidfunktion an jeder Stelle ableitbar, weist jedoch auch zwei starke Nachteile auf:

1. Die logistische Sigmoidfunktion ist nicht stückweise linear; ihre Ableitung ist somit nicht auf eine feste Anzahl an Konstanten einzugrenzen, sondern muss an jeder Stelle neu berechnet werden. Dies verursacht im Vergleich zur ReLU-Funktion einen erhöhten Rechenaufwand beim Berechnen der Gradienten.
2. Wie Abbildung 3.4 unschwer zu entnehmen ist, nähert sich die Funktion für große bzw. kleine Werte 1 bzw. 0 an. Dies hat als Nebeneffekt zur Folge, dass sich Gradienten für große bzw. kleine Eingaben 0 annähern. Nur in naher Umgebung um $z = 0$ existieren starke Gradienten, was Lernen auf Basis von Gradienten deutlich erschwert [GBC16].

Da nur in einem schmalen Bereich starke Gradienten geliefert werden, wird die logistische Sigmoidfunktion als nichtlineare Aktivierungsfunktion inzwischen in der Regel von der ReLU-Funktion abgelöst, die für alle positiven Eingabewerte starke Gradienten liefert. Anwendung findet die logistische Sigmoidfunktion zum Teil noch in der Ausgabeschicht [GBC16]. Auch abseits von den bisher betrachteten Feedforward-Netzen findet sie noch Anwendung, beispielsweise im Rahmen von sogenannten Recurrent Neural Networks oder in Fällen, in denen die stückweise Linearität der ReLU-Funktion unerwüsnchte Eigenschaften aufweist [GBC16].

3.1.3.3. Tangens-hyperbolicus-Funktion

Eine weitere Aktivierungsfunktion ist die Tangens-hyperbolicus-Funktion *tanh* [GBC16]. Die Funktion ist mit Eingabe z wie folgt definiert [BHL+12]:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.7}$$

Für die nichtlineare Aktivierungsfunktion gilt dann: $g(z) = tanh(z)$. Darüber hinaus ist die Funktion stark mit der logistischen Sigmoidfunktion verwandt und lässt sich direkt über diese definieren:

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{1 + e^z} \tag{3.8}$$

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z} - 1}{e^{2z} + 1} = \frac{2 \cdot e^{2z} - (e^{2z} + 1)}{e^{2z} + 1} = 2 \cdot \frac{e^{2z}}{1 + e^{2z}} - 1 \tag{3.9}$$

Setzt man nun Gleichung 3.8 in die umgeformte Tangens-hyperbolicus-Gleichung 3.9 ein, ergibt sich die angesprochene Gleichung in Abhängigkeit der logistischen Sigmoidfunktion, wie sie bei Goodfellow et al. angegeben wird [GBC16]:

$$tanh(z) = 2 * \frac{e^{2z}}{1 + e^{2z}} - 1 = 2 \cdot \sigma(2z) - 1 \tag{3.10}$$

Betrachtet man den Verlauf der Funktion, zu sehen in Abbildung 3.5, sieht man, dass die Funktion um den Wert $x = 0$ der Identitätsfunktion $f(x) = x$ stark ähnelt. Das Training in diesem Bereich ähnelt folglich stark dem Training eines linearen Modells; dies sorgt dafür, dass Tangens-hyperbolicus-Funktionen in der Praxis meistens besser abschneiden als logistische Sigmoidfunktionen [GBC16]. Für Eingabewerte, die sich nicht nahe bei 0 befinden, ist die Tangens-hyperbolicus-Funktion jedoch genauso problematisch wie die logistische Sigmoidfunktion, da sich, wie in Abbildung 3.5 zu sehen, die Steigung für deutlich kleinere

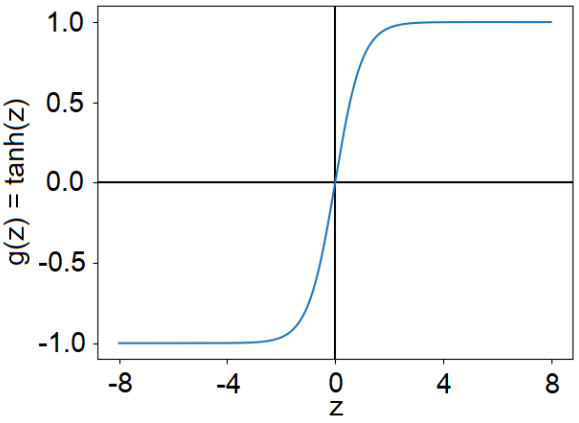


Abbildung 3.5.: Verlauf der Tangens-hyperbolicus-Funktion. Wie bei der logistischen Sigmoidfunktion nähert sich die Steigung der Funktion für große und kleine Werte 0 an; nur in einem schmalen Abschnitt verfügt die Funktion über eine starke Steigung.

und größere Eingabewerte 0 annähert. Daher wird sie im Normalfall bei Feedforward-Netzen auch durch die ReLU-Funktion ersetzt, verbliebene Anwendungsgebiete entsprechen denen der logistischen Sigmoidfunktion.

3.1.4. Ausgabeeinheiten und Ausgabefunktionen

Die Ausgabe eines Neuronalen Netzes wird durch die Neuronen der letzten Schicht, auch Ausgabeeinheiten oder Ausgabeneuronen genannt, erzeugt. Wie in den vorangegangenen Schichten wird die Eingabefunktion auf die Aktivierungen aus der vorangegangenen Schicht angewendet, anschließend kommt jedoch eine Ausgabefunktion statt einer Aktivierungsfunktion zum Einsatz. Dabei ist die Ausgabefunktion und die Anzahl der Ausgabeeinheiten so zu wählen, dass das Netz eine Ausgabe erzeugen kann, die der Zielausgabe der Trainingsbeispiele entspricht. Soll beispielsweise eine Ausgabe im Intervall $[-1, 1]$ erzeugt werden, wäre die logistische Sigmoidfunktion (ohne weitere Nachverarbeitung) eine inadäquate Wahl, da sie nur Ausgaben im Intervall $[0, 1]$ erzeugt. Im Folgenden werden zwei der häufigsten Ausgabemodalitäten vorgestellt.

3.1.4.1. Sigmoid Unit

Die Sigmoid Unit stellt das Mittel der Wahl dar, wenn mit Hilfe eines Neuronalen Netzes eine binäre Entscheidung getroffen werden soll. Um eine einzelne binäre Entscheidung zu treffen, wird in diesem Fall in der Ausgabeschicht ein einzelnes Neuron mit der bereits bekannten Eingabefunktion verwendet. An die Stelle der nichtlinearen Aktivierungsfunktion rückt die Ausgabefunktion, in diesem Fall die bereits als Aktivierungsfunktion bekannte logistische Sigmoidfunktion, die in Abschnitt 3.1.3.2 vorgestellt wurde. Dieses Neuron bildet die Sigmoid Unit.

Statistisch lässt sich die Wahrscheinlichkeitsverteilung einer binären Entscheidung als Bernoulli-Verteilung modellieren, die durch die Wahrscheinlichkeit $P(y = 1|x)$ vollständig charakterisiert wird, wobei x die Eingabe darstellt und y die Ausgabe, $y \in \{0, 1\}$

und $P(y = 1|x) \in [0, 1]$. Sei nun z die Ausgabe der Eingabefunktion wie in Abschnitt 3.1.2 vorgestellt, die unter anderem von der Eingabe x abhängt. Mit Hilfe der logistischen Sigmoidfunktion lässt sich nun eine Bernoulli-Verteilung definieren [GBC16]:

$$P(y|x) = \sigma((2y - 1)z) \tag{3.11}$$

Um die Bernoulli-Verteilung vollständig zu definieren, ist nur ein Wert notwendig, entweder $P(y = 1|x)$ oder $P(y = 0|x)$; der jeweils andere Wert ergibt sich dann automatisch, da sich beide Werte zu 1 aufsummieren müssen. Wählt man nun $y = 1$ ergibt sich aus Gleichung 3.11 folgende Gleichung [GBC16]:

$$P(y = 1|x) = \sigma(z) \tag{3.12}$$

Somit definiert die Ausgabe einer logistischen Sigmoidfunktion eine Bernoulli-Verteilung und damit auch eine binäre Verteilung vollständig. Sie ist daher geeignet, um eine Ausgabe für binäre Entscheidungsprobleme zu erzeugen.

3.1.4.2. Softmax Unit

Mit der zuvor vorgestellten Sigmoid Unit ist es möglich, binäre Entscheidungen zu treffen. In der Praxis kommt es jedoch oft vor, dass nicht nur zwischen zwei Möglichkeiten entschieden werden muss. Man stelle sich vor, das Neuronale Netz erhalte ein Bild als Eingabe und es soll entschieden werden, ob das Bild ein Auto enthält oder nicht. Genauso von Interesse könnte jedoch sein, ob das Bild ein Auto, ein Fahrrad, ein Motorad oder nichts davon enthält, mit der Einschränkung, dass nur eine einzelne dieser vier Möglichkeiten zutreffen kann. Der Einsatz mehrerer Sigmoid Units ist für diesen Fall nicht praktikabel, da sich die Wahrscheinlichkeiten von mehreren dieser Einheiten nicht zu 1 aufsummieren und somit keine gültige Wahrscheinlichkeitsverteilung liefern. Eine gültige Wahrscheinlichkeitsverteilung ist jedoch für das vorgestellte Beispiel vonnöten – man spricht von einer Multinoulli-Verteilung. Statt einer Ausgabeschicht mit einer einzelnen Sigmoid Unit kommt eine Ausgabeschicht mit n beziehungsweise $n - 1$ Neuronen, die als Softmax Units bezeichnet werden, zum Einsatz, wobei n der Anzahl an Fällen, die unterschieden werden sollen, entspricht. Prinzipiell sind $n-1$ Ausgabeeinheiten ausreichend, da sich die Wahrscheinlichkeiten einer gültigen Wahrscheinlichkeitsverteilung zu 1 summieren müssen und sich somit die letzte Wahrscheinlichkeit aus den $n - 1$ anderen direkt ergibt; die Verwendung von n Ausgabeeinheiten sorgt für eine Überparametrisierung. Was auf den ersten Blick nach einem Problem aussehen könnte, spielt in der Praxis in der Regel keine Rolle: Im Normalfall liefern beide Varianten sehr ähnliche Ergebnisse; für die überparametrisierte Variante spricht, dass sie einfacher zu implementieren ist [GBC16]. Betrachtet man nun in der Ausgabeschicht die Softmax Unit Nr. i, ergibt sich folgende Gleichung für die Ausgabe [GBC16]:

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{3.13}$$

Bei der Eingabe $\mathbf{z}$ handelt es sich um die Ausgabe der Eingabefunktion aller Neuronen der Ausgabeschicht. Bei den z_j im Nenner handelt es sich entsprechend jeweils um die Ausgabe der Eingabefunktion des j-ten Neurons der Ausgabeschicht. Die endgültigen Ausgaben aller n Ausgabeneuronen nach Anwendung der *softmax*-Funktion summieren sich zu 1 auf und bilden so eine gültige Wahrscheinlichkeitsverteilung für n Möglichkeiten.

3.1.5. Trainings-, Validierungs- und Testdaten

Bevor die Berechnung des Fehlers des Netzwerks und die darauf basierenden Lernverfahren betrachtet werden, wird zuerst auf die in beiden Fällen benötigten Trainingsdaten näher eingegangen. Im Rahmen dieser Arbeit spielen nur Verfahren des überwachten Lernens eine Rolle. Bei den Trainingsdaten für überwachtes Lernen ist es wichtig, dass von vorneherein für jedes Element der Trainingsdaten auch die Zielausgabe (engl. *Ground Truth*) bekannt ist – man spricht von kommentierten oder annotierten Trainingsdaten. Es werden folglich Paare der Form (x, y) erwartet, wobei x für die Eingabe in das Neuronale Netz und y für die erwartete Ausgabe steht. Eine Menge von Trainingsdaten mit n Beispielen lässt sich folglich als eine Menge von Tupeln $\{(x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)})\}$ ansehen [RN12]. Aufgabe des Neuronalen Netzes ist es, eine Funktion zu erlernen, die alle $x^{(i)}$ möglichst genau auf ihre zugehörigen $y^{(i)}$ abbildet. Zusätzlich zu den Trainingsdaten ist eine separate Testmenge aus Eingabe-Ausgabe-Tupeln nötig, um zu überprüfen, wie gut das Neuronale Netz auf bisher ungesehenen Daten funktioniert. Diese Testmenge ist nötig, da es auf den Trainingsdaten zu Overfitting kommen kann: Das Neuronale Netz kann im schlechtesten Fall damit beginnen, die Trainingsdaten anhand von Details „auswendig“ zu lernen. In diesem Fall kann das Gelernte aber nur noch schlecht auf ungesehene Eingaben verallgemeinert werden. Mit einer bisher ungesehenen Testmenge kann daher nach dem Training überprüft werden, ob das Neuronale Netz gut verallgemeinert – also ob es für die Eingaben der Testmenge die zugeordneten Ausgaben gut abschätzt – oder ob Overfitting aufgetreten ist und keine gute Verallgemeinerungsfähigkeit vorliegt [RN12]. Zur Vermeidung von Overfitting ist es unter anderem wichtig, über eine angemessene Menge an Trainingsdaten zu verfügen. Um beispielsweise bereits während des Trainings eine Abschätzung der Verallgemeinerungsfähigkeit des Netzwerks zu ermöglichen, können darüber hinaus die Trainingsdaten weiter in Trainings- und Validierungsdaten unterteilt werden, wobei die Validierungsdaten nicht für das Training sondern zum Abschätzen der Verallgemeinerungsfähigkeit verwendet werden [GBC16].

Abgesehen von der Trennung in Trainings- und Testdaten gibt es eine Organisation der Trainingsdaten für den Lernvorgang. Für jeden Schritt in Lernalgorithmen wird in der Regel ein sogenannter *Minibatch* als Teilmenge der Trainingsdaten mit zufällig ausgewählten Elementen verwendet. Sollte die Teilmenge exakt der Menge der Trainingsdaten entsprechen, spricht man lediglich von einem *Batch* und nicht von einem Minibatch. Bei der Wahl der Mächtigkeit eines Minibatches sind mehrere Faktoren zu beachten [GBC16]: Da nur eine Teilmenge der Trainingsdaten betrachtet wird, können die Eigenschaften der gesamten Menge und somit insbesondere deren zu berechnender Gradient nur abgeschätzt werden. Je mächtiger die Teilmenge ist, desto besser funktioniert dieser Vorgang. Im Gegensatz hierzu sorgt eine geringe Mächtigkeit der Minibatches in der Regel für eine bessere Verallgemeinerung durch das Neuronale Netz – die besten Ergebnisse werden hier mit nur einem Element pro Minibatch erzielt. Sehr kleine Minibatches wie die einelementige Variante haben jedoch den Nachteil, dass Lernalgorithmen sehr viele Iterationen benötigen und Möglichkeiten der Parallelverarbeitung nicht voll ausgeschöpft werden können, was einen hohen Zeitaufwand bedeutet. Diese und weitere Faktoren beeinflussen die Wahl der Mächtigkeit der Minibatches. Häufig verwendete Mächtigkeiten sind beispielsweise 32, 64, 128 und 256 [GBC16].

3.1.6. Cost- und Loss-Funktion

Damit ein Neuronales Netz etwas lernen kann, muss es wissen, wo es Fehler macht. Der Bestimmung dieses Fehlers dienen die Cost- und Loss-Funktion. Sie bestimmen anhand der bereits vorgestellten annotierten Trainingsdaten, wie groß der Fehler des Neuronalen Netzes bei deren Verarbeitung ist. Auf Basis dieses Fehlers können anschließend durch ein Lernverfahren die Gewichte und Biase des Neuronalen Netzes angepasst werden. Goodfellow et al. [GBC16] verwendet die Begriffe Cost- und Loss-Funktion äquivalent, während im Rahmen dieser Arbeit die Loss-Funktion L den Fehler für ein einzelnes Trainingsbeispiel und die Cost-Funktion J den Fehler für den ganzen Minibatch einer Iteration berechnet.

Für Neuronale Netze, die eine Wahrscheinlichkeitsverteilung als Ausgabe erzeugen, kann die Kreuzentropie als Cost-Funktion eingesetzt werden [GBC16]. Es stehe im Folgenden $E_{\mathbf{x},\mathbf{y}\sim\hat{p}_{Daten}}[.]$ für den Erwartungswert der Funktion in den eckigen Klammern, wenn $(\mathbf{x}, \mathbf{y})$ aus der Verteilung $\hat{p}_{Daten}$ stammt, welche hierbei für die Verteilung der Daten im aktuellen Minibatch steht. Die Wahrscheinlichkeitsverteilung, die durch das Neuronale Netz erzeugt wird, wird im Folgenden als p_{Modell} bezeichnet. Das Symbol θ steht für die Gewichte und Biase des Neuronalen Netzes. Die Kreuzentropie ist dann definiert wie folgt [GBC16]:

$$J(\theta) = -E_{\mathbf{x},\mathbf{y}\sim\hat{p}_{Daten}}[log(p_{Modell}(\mathbf{y}|\mathbf{x}))] \tag{3.14}$$

Die Loss-Funktion ist in diesem Rahmen die Funktion, über die der Erwartungswert definiert wird. Äquivalent zur Bestimmung des negativen Erwartungswerts des Funktionswerts ist die Bestimmung des Erwartungswerts des negativen Funktionswerts. Nimmt man diese Änderung vor, ergibt sich für die Loss-Funktion, für die der Erwartungswert bestimmt werden soll, folgende Formel [GBC16]:

$$L(\mathbf{x}, \mathbf{y}, \theta) = -log(p_{Modell}(\mathbf{y}|\mathbf{x})) \tag{3.15}$$

Für die m Elemente eines Minibatches errechnet sich der Erwartungswert und somit die Cost-Funktion durch die Anwendung der Loss-Funktion auf die einzelnen Elemente des Minibatches; die Ergebnisse werden anschließend aufsummiert und durch die Anzahl der Elemente geteilt. Es ergibt sich folgende Formel [GBC16]:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \theta) \tag{3.16}$$

Der große Vorteil der Verwendung der Kreuzentropie ist es, dass sie auf alle Neuronalen Netze angewendet werden kann, die eine entsprechende Wahrscheinlichkeitsverteilung $p_{Modell}(\mathbf{y}|\mathbf{x})$ modellieren; somit müssen für solche Neuronalen Netze nicht immer individuelle Cost-Funktionen entworfen werden [GBC16]. In der Praxis erfolgt die Berechnung der Loss-Funktion, über die wiederum die Cost-Funktion berechnet werden kann, auf Basis der Ausgabe $\hat{\mathbf{y}}$ des Neuronalen Netzes für die Eingabe $\mathbf{x}$ sowie der zugeordneten Zielausgabe $\mathbf{y}$; folglich wird $L(\hat{\mathbf{y}}, \mathbf{y})$ bestimmt.

3.1.7. Lernverfahren

Abschließend ist noch zu erläutern, wie der Fehler, der mit Hilfe der Cost- und Loss-Funktion auf den Traingsdaten bestimmt wird, dazu benutzt werden kann, dass das Neuronale Netz lernt und so der Fehler auf den Trainingsdaten minimiert wird. Dies ist die

Aufgabe des Lernverfahrens, das für ein Neuronales Netz verwendet wird. Es sorgt dafür, dass der berechnete Fehler zu einer Anpassung der Parameter, also der Gewichte und Biase des Netzwerks, verwendet wird. Dabei verfügt das Lernverfahren selbst über Hyperparameter, die normalerweise von Hand angepasst werden, wie beispielsweise die Geschwindigkeit, mit der gelernt wird – Lernrate genannt. Wichtig für die im Folgenden vorgestellten Lernverfahren ist es, dass alle Funktionen im Neuronalen Netz differenzierbar sind, da für den Lernvorgang Gradienten für alle Schichten berechnet werden müssen. Ausnahmen sind möglich, wenn die Funktionen lediglich in vereinzelten Punkten nicht differenzierbar sind, wie beispielsweise bei der vorgestellten ReLU-Funktion in Abschnitt 3.1.3.1. Im Folgenden wird vorgestellt, wie die Gradienten, die zur Anpassung aller Gewichte und Biase im Neuronalen Netz nötig sind, berechnet werden können; im Anschluss werden Verfahren zur tatsächlichen Anpassung dieser Parameter betrachtet.

3.1.7.1. Backpropagation

Unter dem Backpropagation-Algorithmus, kurz *Backprop*, versteht man ein Verfahren, um die Gradienten für die Parameteranpassung in den verschiedenen Schichten eines Neuronalen Netzes zu berechnen. Ausgehend von der Cost-Funktion am Ende des Netzes, die den Fehler bezüglich der Trainingsdaten berechnet, werden die Gradienten für die jeweils zugehörigen Parameter einer Schicht durch schichtweises Zurückgehen im Neuronalen Netz auf Basis vorheriger Ergebnisse berechnet, bis am Ende die Eingabeschicht erreicht wird. Diese Gradienten sind dann im Rahmen anderer Verfahren nötig, um die Gewichte und Biase der Neuronen einer jeden Schicht anzupassen. Entwickelt wurde der Algorithmus von Rumelhart et al. [RHW86] im Jahr 1986, im Folgenden wird er dennoch auf Basis der Arbeit von Goodfellow et al. [GBC16] vorgestellt.

Der Backpropagation-Algorithmus beruht auf der rekursiven Anwendung der Kettenregel. Im Folgenden wird eine schichtweise Sicht und keine neuronenweise Sicht auf das Neuronale Netz verwendet. Es sei $\mathbf{x}$ der Eingabevektor für Schicht k eines Neuronalen Netzes, $\mathbf{y} = g(\mathbf{x})$ sei der Ausgabevektor derselben Schicht. Darüber hinaus sei $z = f(\mathbf{y}) = f(g(\mathbf{x}))$ die Ausgabe von Schicht $k+1$ und der Einfachheit halber in diesem Fall ein Skalar. f und g bezeichnen die Funktionen, die durch die Schichten realisiert werden. Nach der Kettenregel lässt sich die Ableitung von z in Abhängigkeit eines einzelnen Elements x_i des Vektors $\mathbf{x}$ wie folgt beschreiben [GBC16]:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{3.17}$$

Möchte man nun die Ableitung von z nach dem gesamten Vektor $\mathbf{x}$ in einer Formel ausdrücken, ist eine Formulierung unter Zuhilfenahme der Jakobimatrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ der Funktion g möglich [GBC16]:

$$\nabla_{\mathbf{x}} z = (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T \nabla_{\mathbf{y}} z \tag{3.18}$$

Der Ausdruck $\nabla_{\mathbf{y}} z$ beschreibt die Ableitung von z bezüglich $\mathbf{y}$, $\nabla_{\mathbf{x}} z$ entsprechend dasselbe für $\mathbf{x}$. Wie zu sehen ist, lässt sich mit Hilfe von Formel 3.18 die Ableitung der Cost-Funktion in Bezug auf die Parameter einer jeden Schicht rekursiv bis zur Eingabe zurückführen. Die eigentliche Leistung des Backpropagation-Algorithmus liegt jedoch nicht in der Anwendung der Kettenregel und der rekursiven Definition der Gradienten bis hin zu den

Eingabewerten, sondern darin, diesen Vorgang effizient zu gestalten. Wird die Kettenregel rekursiv angewendet, entsteht eine Berechnung, die aus vielen Teilausdrücken besteht. Bei einer naiven Herangehensweise, bei der für jeden Gradienten alle Teilausdrücke neu berechnet werden, kommt es zu vielen unnötigen Berechnungen. Im schlimmsten Fall kann bei naiver Herangehensweise sogar exponentieller Rechenaufwand entstehen [GBC16]. Backpropagation verhindert dies, indem Teilausdrücke intelligent zusammengefasst werden; in der Berechnung kann auf vorherige Ergebnisse – die zusammengefassten Teilausdrücke – zurückgegriffen werden. Auf diese Art werden die gemeinsamen Teilausdrücke in den Berechnungen der Gradienten und somit die redundanten Berechnungen reduziert [GBC16].

Die Funktionsweise des Backpropagation-Algorithmus lässt sich am besten bildhaft mit einem sogenannten *Computational Graph* beschreiben. Der Graph behandelt jede zu berechnende Funktion als Knoten, die Eltern eines jeden Knotens sind die Eingaben, die für dessen Berechnung nötig sind – sie sind mit von den Eltern ausgehenden Pfeilen verbunden. Mit ihm lässt sich die Verknüpfung vieler Funktionen graphisch darstellen. Auch die gesamte Berechnung eines neuronalen Netzes lässt sich mit solch einem Graph darstellen. Zur Berechnung der Cost-Funktion wird der Graph von vorne her durchlaufen; zu jedem Zeitpunkt lässt sich auf Basis der Ergebnisse aus bisherigen Knoten die Ausgabe eines weiteren Knoten berechnen, bis am Ende die Cost-Funktion berechnet wird. Basierend auf diesem Graph werden die Gradienten in genau der umgekehrten Reihenfolge berechnet und jeweils als Ausgabe der Knoten gespeichert. Die jeweiligen Vaterknoten des Computational Graph können so auf die gespeicherten Gradienten der Kindknoten zurückgreifen und ihre Gradienten jeweils ohne redundante Berechnungen zu bestimmen. Eine detailiertere Erklärung inklusive Bildern kann der Arbeit von Goodfellow et al. [GBC16] entnommen werden. Mit dieser Vorgehensweise erreicht es der Backpropagation-Algorithmus, dass die Anzahl der Berechnungen linear zur Anzahl der Kanten in diesem Graph ist [GBC16]. Man nehme nun einen vollständigen Graphen als schlimmsten Fall an, um eine Abschätzung der Anzahl der Berechnungen nach oben zu erhalten. Bekanntermaßen ist die Anzahl der Kanten in einem vollständigen Graphen quadratisch im Bezug auf die Anzahl der Knoten. Da die Anzahl der Berechnungen sich linear zur Anzahl der Kanten verhält, kann somit die Anzahl der Berechnungen im Rahmen des Backpropagation-Algorithmus nicht mehr exponentiell werden.

Trotz der effizienten Berechnung gibt es immer noch effizientere Alternativen zum vorgestellten Backpropoagation-Algorithmus. Beispielsweise findet keine Vereinfachung des Graphen statt – so können beispielsweise weitere Berechnungen eingespart werden [GBC16]. Solche verbesserten Varianten werden jedoch im Rahmen dieser Arbeit nicht betrachtet.

3.1.7.2. Gradientenabstieg

Nun, da bekannt ist, wie alle Gradienten eines Neuronalen Netzes berechnet werden können, ist noch zu klären, wie das Neuronale Netz auf Basis dieser Gradienten lernen, also seine Gewichte und Biase anpassen kann. Eines der grundlegendsten Verfahren hierzu ist der Gradientenabstieg. Sei im Folgenden $f(x)$ eine Funktion mit skalarer Eingabe x, deren Funktionswert $y = f(x)$ minimiert werden soll. Mit Hilfe des Vorzeichens der ersten Ableitung $f'(x)$ lässt sich bestimmen, ob der Eingabewert x vergrößert oder verkleinert werden muss, um den Funktionswert zu minimieren. Eine negativer Wert bei der Ableitung steht für eine negative Steigung an der aktuellen Stelle, die eine Vergrößerung des

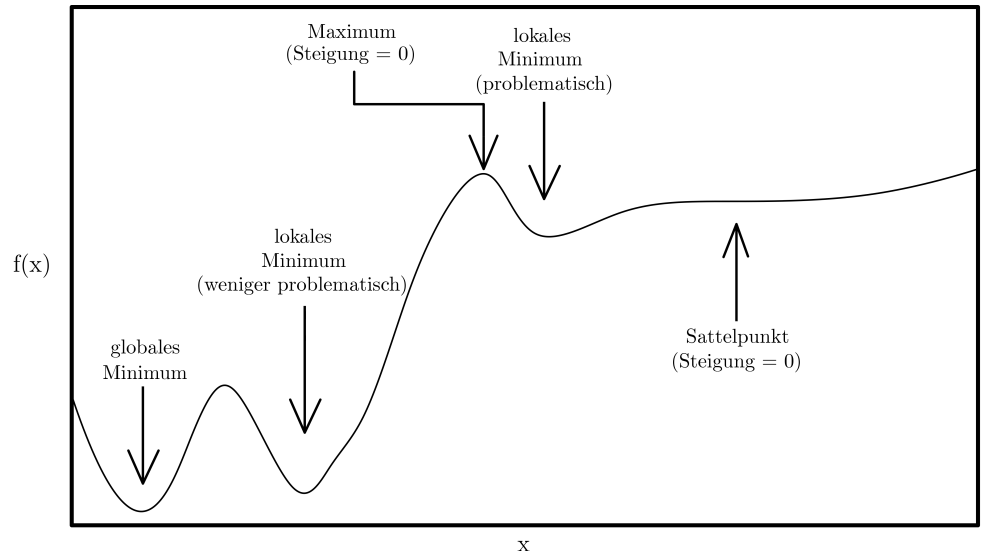


Abbildung 3.6.: Exemplarischer Funktionsverlauf mit interessanten Stellen für den Gradientenabstieg. Neben dem eingezeichneten globalen Minimum können auch die lokalen Minima gefunden werden, von denen das linke weniger problematisch ist, da sein Funktionswert nahe dem des globalen Minimums liegt. Neben den Minima hat der Gradient auch bei Maxima und Sattelpunkten den Wert 0, was Lernen verhindern kann.

Eingabewertes nötig macht, während eine positive Ableitung für eine positive Steigung steht, die wiederum eine Verkleinerung des Eingabewertes nötig macht. Addiert man nun einen kleinen, häufig fest gewählten Wert ε zu x hinzu bzw. subtrahiert man ihn von x und wertet die Funktion an der neuen Stelle aus, so gilt, falls das verwendete ε klein genug gewählt wurde [GBC16]:

$$f(x - \varepsilon \cdot sign(f'(x))) < f(x) \tag{3.19}$$

Setzt man nun $x = x - \varepsilon \cdot sign(f'(x))$ und wiederholt das ganze Vorgehen iterativ, so nähert man sich einem Minimum von $f(x)$ an. Nicht immer liefert der Gradientenabstieg ein optimales Ergebnis; es gibt einige Fälle, die in der Praxis problematisch sind. Am wichtigsten ist der Fakt, dass es sich bei einem gefundenen Minimum nicht um ein globales, sondern um ein lokales Minimum handeln kann. In der lokalen Umgebung ist der Funktionswert zwar am niedrigsten, jedoch nicht über den gesamten Verlauf der Funktion. Es kann passieren, dass man sich einem solchen lokalen Minimum annähert – ist der Funktionswert ähnlich gering wie beim globalen Minimum, stellt dies in der Praxis bei Neuronalen Netzen weniger ein Problem dar [GBC16]. Weitere problematische Punkte sind Maxima und Sattelpunkte, da bei ihnen die Ableitung wie bei Minima auch den Wert 0 annimmt und sich x nach Formel 3.19 somit nicht mehr verändert. Eine Darstellung dieser Problemstellen kann Abbildung 3.6 entnommen werden. Bei einem Maximum ist beispielsweise darüber hinaus auch bei Betrachtung der direkten Umgebung unklar, ob das globale Minimum rechts oder links des Maximums zu suchen ist, da sich in beiden Fällen der Funktionswert verringert.

Im Rahmen von Neuronalen Netzen ist die Eingabe für eine Funktion im Normalfall kein skalarer Wert, sondern vielmehr ein Vektor $\mathbf{x}$ mit m Elementen. Daher wird im Folgenden noch der Fall des Gradientenabstiegs betrachtet, bei dem die Funktion $f(\mathbf{x})$ einen

skalaren Funktionswert $y = f(\mathbf{x})$ auf Basis eines m-elementigen Eingabevektors $\mathbf{x}$ erzeugt. Wie bereits im eindimensionalen Fall geht man in kleinen Schritten in die Richtung, in die der Funktionswert am schnellsten sinkt: dies ist wiederum die Richtung, die dem Gradienten entgegengesetzt ist. Der Gradient bezüglich der Funktion f und dem Eingabevektor $\mathbf{x}$ wird im mehrdimensionalen Fall mit $\nabla_{\mathbf{x}} f(\mathbf{x})$ bezeichnet; er gibt die Richtung im m-dimensionalen Raum mit der stärksten Steigung des Funktionswerts an. Mit dem bekannten ε, das die Größe der gemachten Schritte bestimmt, ergibt sich folgende Regel, um $\mathbf{x}$ zu aktualisieren und somit $f(\mathbf{x})$ zu minimieren [GBC16]:

$$\mathbf{x_{neu}} = \mathbf{x} - \varepsilon \nabla_{\mathbf{x}} f(\mathbf{x}) \tag{3.20}$$

Da im Rahmen von Neuronalen Netzen meistens nicht die ganzen Trainingsdaten sondern Minibatches mit zufällig ausgewählten Elementen für die einzelnen Iterationen verwendet werden, kommt dort stochastischer Gradientenabstieg zum Einsatz: Die Daten des Minibatches liefern nur eine Abschätzung des Gradienten über die gesamten Trainingsdaten und unterscheiden sich folglich von Minibatch zu Minibatch. Damit effektiv gelernt werden kann muss daher die Lernrate im Lauf der Zeit verringert werden, um Oszillationen durch die unterschiedlichen Gradienten der Minibatches zu verringern, sobald sich einem Minimum angenähert wird. Unterschiedliche Strategien für die Wahl der Lernrate und ihre Anpassung über die Zeit können Goodfellow et al. [GBC16] entnommen werden.

Im Folgenden wird ein weiteres Lernverfahren vorgestellt, das auf der individuellen Optimierung und Anpassung der Lernrate für die einzelnen Parameter des Neuronalen Netzes beruht.

3.1.7.3. Adam

Durch den *Adam*-Algorithmus von Kingma et al. [KB14], der hier auch anhand ihrer Arbeit vorgestellt wird, soll eine Verbesserung gegenüber vorangegangenen Arbeiten dadurch erzielt werden, dass für die einzelnen Parameter des Neuronalen Netzes separate Lernraten verwaltet werden, die adaptiv auf Basis einer Schätzung der ersten beiden Momente der Gradienten – Mittelwert und unzentrierte Varianz – angepasst werden [KB14]. Dabei greift Adam auf die Ergebnisse zweier früherer Methoden zurück, *AdaGrad* [DHS11] sowie *RMSProp* [TH12], und möchte deren Vorteile vereinen [KB14].

Um eine individuelle Lernrate zu erzielen, zieht Adam den Mittelwert (erstes Moment) und die unzentrierte Varianz (zweites Moment) der jeweiligen Gradienten heran, um eine stochastische Zielfunktion f zu optimieren, die von gewissen Parametern abhängt. Im Fall von Neuronalen Netzen sind die Parameter der Funktionen des Netzes zu optimieren; die stochastische Eigenschaft der zu optimierenden Funktionen rührt wie bereits beim stochastischen Gradientenabstieg von der Verwendung von Minibatches her, da zufällige Elemente der gesamten Trainingsdaten für jeden Minibatch ausgewählt werden. Die unzentrierte Varianz ist bei dem vorliegenden Algorithmus nicht mit der klassischen Varianz (zweites zentrales Moment) zu verwechseln, bei der vor dem Quadrieren der Mittelwert subtrahiert wird. Sowohl Mittelwert als auch unzentrierte Varianz werden mit Hilfe des sogenannten *Exponential Moving Average*, kurz *EMA*, abgeschätzt, dessen exponentielle Verfallsrate im Folgenden mit β_1 für den Mittelwert und β_2 für die unzentrierte Varianz bezeichnet wird. Auf Deutsch ist der EMA auch als exponentiell geglätteter Mittelwert bekannt. Die mit Hilfe des EMA geschätzen Werte für Mittelwert und unzentrierte Varianz

werden im Anschluss einer Biaskorrektur unterzogen, bevor sie abschließend zusammen mit einer festen Schrittweite ε und einem sehr klein gewählten δ (letzteres zum Gewährleisten numerischer Stabilität [GBC16]) verwendet werden, um die Parameter der Funktion zu aktualisieren. Mathematische Details von Adam können Algorithmus 1 entnommen werden. Die Symbole mancher Parameter weichen von denen der Originalarbeit ab, um Konsistenz mit anderen Teilen dieser Arbeit zu erreichen.

Algorithmus 1 : Der Adam-Algorithmus, wie er in der Arbeit von Kingma et al. [KB14] vorgestellt wird. Operationen auf Vektoren werden elementweise ausgeführt:

Eingabe : $f(\theta)$ – Stochastische Zielfunktion, deren Parameter optimiert werden sollen
θ_0 – Zur Zielfunktion gehörende initiale Parameter
ε – Schrittweite (Skalar)
β_1 & β_2 – Exponentielle Verfallsrate des jeweiligen EMA
δ – Wert zur numerischen Stabilisierung
Ausgabe : θ_t – Optimierte Parameter der Zielfunktion

$\mathbf{m}_0 \leftarrow 0$ (Initialisierung EMA-Vektor für Mittelwert)
$\mathbf{v}_0 \leftarrow 0$ (Initialisierung EMA-Vektor für unzentrierte Varianz)
$t \leftarrow 0$ (Initialisierung Zeitschritt)
solange θ_t *noch nicht konvergiert* **tue**
$t \leftarrow t + 1$ (Aktualisierung Zeitschritt)
$\mathbf{g}_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Berechnung des aktuellen Gradientenvektors)
$\mathbf{m}_t \leftarrow \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot \mathbf{g}_t$ (Aktualisierung EMA Mittelwert)
$\mathbf{v}_t \leftarrow \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot \mathbf{g}_t^2$ (Aktualisierung EMA unzentrierte Varianz)
$\hat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$ (Biaskorrektur des EMA des Mittelwertsvektors)
$\hat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$ (Biaskorrektur des EMA des unzentrierten Varianzvektors)
$\theta_t \leftarrow \theta_{t-1} - \varepsilon \cdot \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \delta)$ (Aktualisierung Funktionsparameter)
Ende
zurück θ_t

Wie in Algorithmus 1 zu sehen ist, rührt die individuelle Anpassung der Lernrate nicht von der Schrittweite ε her, die fest gewählt wird, sondern vom Verhältnis der Elemente der Vektoren von $\hat{\mathbf{m}}_t$ und $\sqrt{\hat{\mathbf{v}}_t}$. So erhält jeder Parameter eine individuelle Lernrate. Wird die Wurzel der unzentrierten Varianz im Verhältnis zum Mittelwert groß, spricht das dafür, dass keine große Sicherheit bezüglich der einzuschlagenden Richtung besteht, entsprechend sind die ausgeführten Schritte im Parameterraum um $\Delta_t = \varepsilon \cdot \hat{\mathbf{m}}_t / \sqrt{\hat{\mathbf{v}}_t}$ klein [KB14]; δ wird hier aufgrund seiner geringen Größe vernachlässigt. Dieser Effekt kann zum Beispiel bei häufigen Vorzeichenwechseln der Gradienten auftreten, da der Mittelwert das Vorzeichen beachtet, es jedoch im Rahmen der unzentrierten Varianz durch das Quadrieren wegfällt. Dies passiert unter anderem häufig in der Nähe von Minima, folglich findet hier eine automatische Verringerung der Größe der Schritte statt, sodass eine Annäherung an das Minimum in kleinen Schritten eher möglich ist. In fast allen Fällen gilt darüber hinaus $|\Delta_t| < \varepsilon$, folglich fungiert ε als eine obere Grenze für die Größe der einzelnen Schritte zur Anpassung der Parameter [KB14]. Abschließend bleibt die Biaskorrektur zu erläutern. Diese ist vor allem in den ersten Iterationen wichtig, da die EMA-Vektoren für Mittelwert und unzentrierte Varianz mit Nullen initialisiert werden. Da β_1 und β_2 sehr nahe bei 1.0 gewählt werden (typischerweise $\beta_1 = 0.9$ und $\beta_2 = 0.999$ [KB14]), bleiben die entsprechenden EMA-Vektoren anfangs reltiv klein – um diesen Bias zu korrigieren und

EMA-Vektoren der Größe wie nach vielen Iterationen zu erhalten, findet die Biaskorrektur mit Division durch $(1 - \beta_1^t)$ bzw. $(1 - \beta_2^t)$ statt. Mit zunehmender Anzahl an Iterationen nimmt die Größe der EMA-Vektoren vor der Biaskorrektur zu, da der Einfluss der Initialisierung der EMA-Vektoren schwindet. Ebenso nimmt der Einfluss der Biaskorrektur ab; der Divisor nähert sich 1 an. Die starke Nähe von $\beta_2 = 0.999$ zu 1.0, die die Biaskorrektur insbesondere nötig macht, ist dabei notwendig, um die unzentrierte Varianz dünn besiedelter Gradienten gut abschätzen zu können [KB14].

Für die Anwendung in der Praxis ist der Adam-Algorithmus vor allem interessant, da er bezüglich der Wahl der Hyperparameter in vielen Fällen sehr robust ist und zu den Algorithmen zählt, die derzeit die besten Ergebnisse liefern, ohne dass unter ihnen ein eindeutig bester Algorithmus festgestellt werden kann [GBC16].

3.1.8. Convolutional Neural Networks

Bisher wurden nur Fully Connected Layers als Schichten für Neuronale Netze vorgestellt, bei denen jedes Neuron einer Schicht n die Ausgaben aller Neuronen der Schicht $n - 1$ erhält. Für viele Anwendungen funktionieren diese Schichten gut; für den Bereich der Bildverarbeitung ist ihre Verwendung jedoch nur bedingt geeignet. Moderne Auflösungen wie beispielsweise 1920×1080 Pixel verfügen bereits über mehr als zwei Millionen Bildpunkte, die im Fall von Farbbildern mit 3 Farbkanälen durch über sechs Millionen Werte repräsentiert werden. Für die Eingabe in ein Neuronales Netz ist für jeden Wert eine Eingabeeinheit nötig. Neuronen der nachfolgenden Schicht sind in klassischen Neuronalen Netzen mit Fully Connected Layers mit allen Eingabeeinheiten verbunden und müssen jeweils für jede Verbindung ein zugehöriges Gewicht speichern – also jeweils über 6 Millionen Gewichte. Formal ausgedrückt: Sei die Anzahl der Eingabeeinheiten m und die Anzahl der Neuronen der nachfolgenden Schicht n. In diesem Fall müssen $m \cdot n$ Gewichte gespeichert werden. Bei wachsender Größe der Eingabeschicht steigt folglich auch der Speicheraufwand für die Gewichte sowie der Rechnaufwand für die jeweilige gewichtete Summe. Wie am Beispiel der Bildpunkte zu sehen war, gibt es in der Praxis Szenarien, bei denen m sehr groß werden kann und somit der Einsatz von Fully Connected Layers problematisch wird.

Convolutional Neural Networks (*CNNs*), vorgestellt anhand von Goodfellow et al. [GBC16] und ursrpünglich begründet durch LeCun et al. [LBD+89], stellen eine Lösung für dieses Problem dar. Statt dass alle Neuronen der ersten verborgenen Schicht eine Verbindung zu jedem Neuron der Eingabeschicht haben, wird die Anzahl der Verbindungen auf einen festen Wert limitiert. Man stelle sich die Anordnung der Neuronen wie in einer Matrix vor, so, wie die Pixel in einem Bild angeordnet sind. Neuronen der Schicht $i + 1$ werden dann jeweils mit einem relativ kleinen, lokal zusammenhängendem Bereich der vorangegangenen Schicht i verbunden, wodurch die Anzahl an Verbindungen und somit an Gewichten sehr stark reduziert werden kann. Der Gedanke hinter diesem Vorgehen ist, dass in Bildern wichtige Informationen meist in einer kleinen Umgebung um jeden Pixel und nicht in weit voneinander entfernten Pixeln gefunden werden können, wie beispielsweise Kanten und Ecken [GBC16]. In der Praxis setzen CNNs hierfür sogenannte Kernels, auch Filter genannt, in Kombination mit Faltung (engl. *Convolution*) oder Kreuzentropie ein. Für Bilder kommen normalerweise zweidimensionale Kernels zum Einsatz: Matritzen, die im Normalfall deutlich kleiner als die Eingabe sind und deren Einträge Gewichte des Neuronalen Netzes sind, die gelernt werden. Diese Matritzen werden über die ebenfalls zweidimensional angeordneten Eingabedaten geschoben und mit den Daten an den Stellen, an die sie

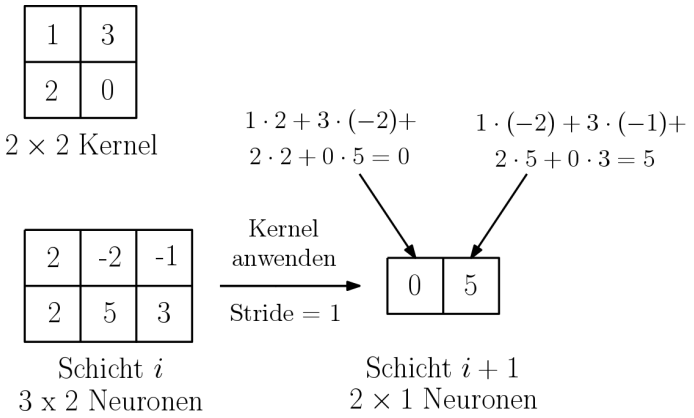


Abbildung 3.7.: Darstellung der Anwendung eines Kernels. Die Neuronen einer Schicht i sind in Matrixform angeordnet. Über diese Matrix wird ein Kernel geschoben, der in diesem Fall die Größe 2×2 hat. An jeder Stelle, an die der Kernel geschoben wird, werden die Gewichte des Kernels mit den Ausgaben der Neuronen multipliziert. Der Kernel wird dabei überall mit denselben Gewichten angewendet. Im Beispiel wird der Kernel jeweils um 1 verschoben; die Größe, die die Verschiebung festlegt, wird Stride genannt. Stellt man sich das Neuronale Netz mit einzelnen Neuronen und gewichteten Verbindungen vor, führen im vorliegenden Beispiel zu jedem Neuron der Schicht $i+1$ vier Verbindungen von quadratisch angeordneten Neuronen der Schicht i, wobei die Gewichte jeweils denen des Kernels entsprechen, also überall geteilt werden.

geschoben werden, elementweise multipliziert und aufaddiert – das Ganze entspricht dem Erstellen einer gewichteten Summe. Wichtig bei dem Verfahren ist, dass die Gewichte des Kernels unabhängig von der Position sind, also geteilt werden. Mathematisch entspricht der beschriebene Vorgang der Berechnung der Kreuzkorrelation, das Neuron der nachfolgenden Schicht an Stelle i, j berechnet folgende Formel für Eingabebild I und Kernel K [GBC16]:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{3.21}$$

Die Kreuzkorrelation entspricht dabei einer Faltung, bei der der Kernel nicht gespiegelt wird – bei der Formel für die Faltung werden daher lediglich die Additionen beim Zugriff auf das Bild I durch Subtraktionen ersetzt [GBC16]. Namensgebend für das CNN ist der englische Begriff für die Faltung: Convolution. In der Praxis macht es jedoch selten einen Unterschied, ob Kreuzkorrelation oder Faltung verwendet wird; oft wird deshalb die Kreuzkorrelation verwendet und nicht selten auch einfach als Faltung bezeichnet [GBC16]. Auch in dieser Arbeit wird im Folgenden der Begriff Faltung als Synonym für die Kreuzkorrelation verwendet werden. Die Schichten, in denen Kernels zur Berechnung der Faltung eingesetzt werden, werden *Convolutional Layers* genannt. Zum besseren Verständnis der Funktionsweise eines CNNs wird die Verwendung eines Kernels exemplarisch in Abbildung 3.7 dargestellt.

CNNs erzeugen nach der Anwendung eines Kernels auf eine in Matrixform organisierte Schicht wiederum eine Schicht, die wie eine Matrix organisiert ist. Hierauf kann wiederum ein Kernel angewendet werden. Auch können mehrere Kernels pro Schicht angewendet werden; jeder Kernel gibt dann einen sogenannten Kanal aus. Die Kanäle werden dann aneinandergereiht. Sind mehrere Kanäle vorhanden, erstrecken sich die Kernels in der Regel auch über mehrere Kanäle; es kommt eine weitere Dimension für die Kernels hinzu, über die jedoch nicht verschoben wird [GBC16]. Beispielsweise macht es Sinn, die Werte der drei Farbkanäle eines Farbbildes auch als drei separate Kanäle in ein Neuronales Netz einzugeben. Zusätzlich zu den zwei Dimensionen, über die sich das Bild erstreckt, gibt es an jeder zweidimensionalen Stelle nun mehrere Eingabewerte, also eine dritte Dimension. Kernels können nun über alle drei Dimensionen angewendet werden. Die Anzahl der Kanäle der nachfolgenden Schicht entspricht dabei der Anzahl an Kernels, die angewendet werden; jeder Kernel erzeugt einen eigenen, zugehörigen Kanal. Zur Veranschaulichung siehe Abbildung 3.8. Zur Architektur eines CNNs gehört es festzulegen, wieviele Kernels es pro Schicht gibt. Darüber hinaus muss der sogenannte Stride festgelegt werden; dieser bestimmt, wie weit die Kernels jeweils verschoben werden, beispielsweise jeweils nur um einen Schritt oder um zwei. Dies beeinflusst auch die Form der Schichten, da bei höherem Stride die Kernels an weniger Stellen der Schicht, auf der sie operieren, verschoben werden und so weniger Ausgaben erzeugt werden. Siehe dazu Abbildung 3.8. Abschließend muss festgelegt werden, wie mit dem Rand des Bildes umgegangen wird. Möchte man in Schicht $i+1$ die gleiche räumliche Ausdehnung wie in Schicht i erreichen, muss neben der Verwendung von einem Stride der Größe 1 der Rand der Schicht i entsprechend erweitert werden. Der Kernel kann ansonsten nur an Stellen von Schicht i angewendet werden, auf die er vollständig passt – infolgedessen schrumpft die räumliche Ausdehnung, wie bereits in Abbildung 3.7 gesehen. Eine Möglichkeit hierfür wäre das *Zero Padding*: Das Füllen des Randes mit Nullen. Dadurch kann erreicht werden, dass der Kernel an mehr Stellen verwendet werden kann und Schicht $i+1$ nicht mehr schrumpft. Auch dieser Vorgang ist visuell leichter verständlich, siehe dazu Abbildung 3.9.

Wie bereits bei den klassischen Neuronalen Netzen wird üblicherweise Nichtlinearität eingebracht. Auf die Ausgaben der Convolutional Layer, erzeugt durch die Faltung, wird daher noch eine nichtlineare Aktivierungsfunktion angewendet, z.B. die logistische Sigmoidfunktion oder die ReLU-Funktion. Im Anschluss findet sich meistens noch eine sogenannte *Pooling Layer*, bevor eine weitere Convolutional Layer zum Einsatz kommt. Diese Pooling Layer setzt eine *Pooling*-Operation ein, um bestimmte Regionen nach Anwendung der Nichtlinearität zusammenzufassen. Eine Pooling-Operation kann so beispielsweise ein 2×2-Areal betrachten, den höchsten vorkommenden Wert auswählen und dann ausgeben; danach wird das Areal jeweils um einen festen Wert verschoben und die Extraktion des höchsten Wertes wird wiederholt. So werden Werte über die gesamte Ausgabe nach der Nichtlinearität extrahiert. Die Pooling-Operation, die im Beispiel verwendet wurde, wird *Max Pooling* genannt. Veranschaulicht wird die Funktionsweise von Pooling in Abbildung 3.10. Neben dem bereits erwähnten Max Pooling existieren noch weitere Pooling-Operationen, beispielsweise kann statt des Maximums auch der Durchschnitt über das Areal berechnet werden – diese Operation nennt sich *Average Pooling*. Alternativ kann auch eine Variante verwendet werden, bei der eine Gewichtung der einzelnen Elemente abhängig von der Distanz zum Mittelpunkt des Areals erfolgt, oder man berechnet statt des Durchschnitts die L_2-Norm über das Areal [GBC16]. Die Größe des Areals kann ebenso

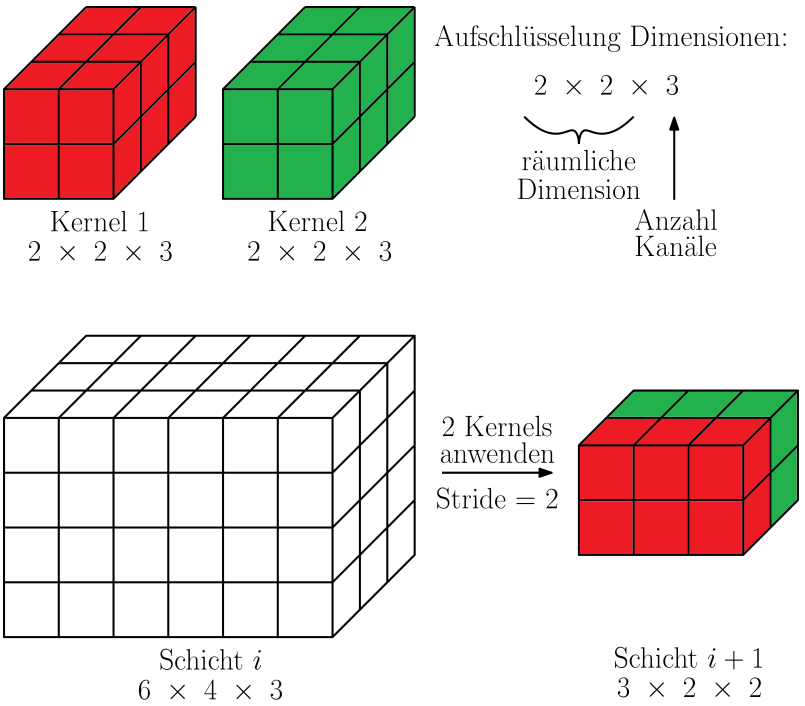


Abbildung 3.8.: Darstellung einer Faltung mit mehreren Kanälen. Schicht i verfügt über drei Kanäle, über entsprechend viele Kanäle verfügen auch die Kernels, die auf die Schicht angewendet werden. Da zwei Kernels angewendet werden, hat Schicht $i+1$ in Folge zwei Kanäle, wovon jeder durch einen der Kernels erzeugt wird. Da hier $Stride = 2$ gewählt wurde, werden die Kernels auch jeweils um zwei Stellen verschoben, was eine kleinere Größe von Schicht $i+1$ zur Folge hat als wenn $Stride = 1$ eingesetzt worden wäre.

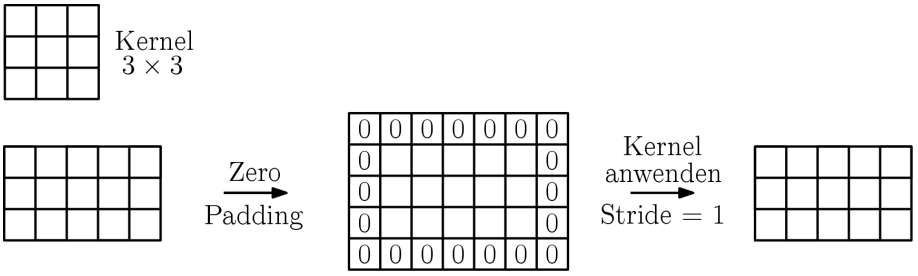


Abbildung 3.9.: Exemplarische Darstellung von Zero Padding. Möchte man vor und nach der Anwendung eines Kernels bei der Faltung die Größe der Schicht beibehalten, muss als Zwischenschritt beispielsweise Zero Padding durchgeführt werden. Dabei muss ein Rand von Nullen eingefügt werden. Für einen Kernel der Größe 3×3 wie im Beispiel reicht es, einen Rand der Größe 1 mit Nullen einzufügen. Für größere Kernels muss der Rand jeweils so groß gewählt werden, dass der Mittelpunkt des Kernels nach dem Zero Padding auf dem Rand der Schicht vor dem Zero Padding verschoben werden kann.

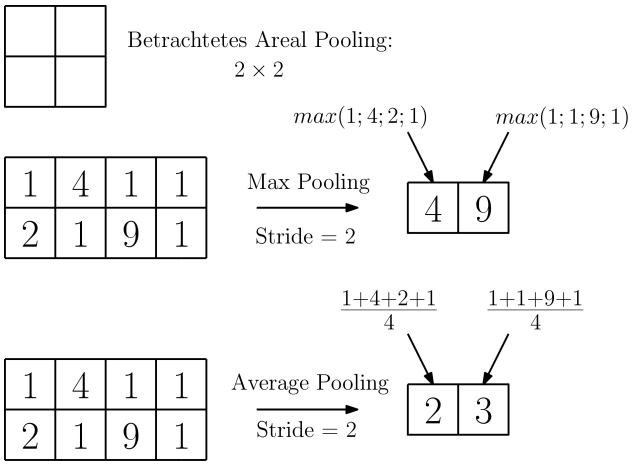


Abbildung 3.10.: Veranschaulichung zweier bekannter Pooling-Operationen. In beiden Fällen wird jeweils ein 2×2-Areal für das Pooling betrachtet. Wie bei den Kernels im Falle der Faltung wird das Areal jeweils um den Stride-Wert über die Neuronen in Matrixform verschoben. Beim Max Pooling wird dabei jeweils der größte Ausgabewert der Neuronen im betrachteten Areal weitergereicht, während beim Average Pooling der Durchschnitt über alle Ausgaben im Areal berechnet und weitergereicht wird.

Abbildung 3.11.: Klassischer Block in einem CNN [GBC16]: Eine Eingabe durchläuft zuerst eine Convolutional Layer, die eine Faltung mit Kernels durchführt. Auf die Ergebnisse dieser Schicht wird anschließend beispielsweise die ReLU-Funktion angewendet, um Nichtlinearität zu erhalten. Abschließend wird im Rahmen der Pooling Layer Pooling durchgeführt, beispielsweise um Ergebnisse kompakter zusammenzufassen. Abbildung auf Basis von [GBC16].

variabel gewählt werden, sollte jedoch bei den meisten Pooling-Operationen eine quadratische oder rechteckige Form beibehalten [GBC16]. Durch den Einsatz von Pooling Layers lässt sich zum einen die räumliche Dimension der Schichten verringern (wie in Abbildung 3.10), zum anderen lässt sich durch Pooling in gewissem Maße eine Invarianz gegenüber Transformationen wie beispielsweise Translationen erreichen [GBC16].

Insgesamt werden folglich meistens Blöcke aus Convolution Layer, Nichtlinearität und Pooling Layer verwendet – diese Blöcke werden zum Teil auch einfach als Convolutional Layer bezeichnet [GBC16]. Eine Darstellung eines solchen Blocks ist in Abbildung 3.11 zu sehen.

3.1.9. Recurrent Neural Networks

Bisher wurden lediglich Modelle vorgestellt, die nicht explizit für die Verarbeitung sequentieller Daten, bei denen zeitliche Abhängigkeiten unter den Daten bestehen, vorgesehen sind: Erhalten klassische Neuronale Netze oder CNNs Eingaben sequentiell, wird für jede separate Eingabe ein separates Ergebnis erzeugt, ohne dass andere Daten der Sequenz

einen Einfluss auf das Ergebnis haben. Möchte man vorangegangene Daten der Sequenz mit einbeziehen, müsste die Eingabe der Sequenz in das neuronale Netz nicht separat Schritt für Schritt, sondern als ein Ganzes erfolgen. Dies wirft wiederum mehrere Probleme auf [GBC16]: Bei der Eingabe sehr langer Sequenzen wächst die Anzahl der Verbindungen von der Eingabeschicht zur ersten verborgenen Schicht sehr stark, was den Speicheraufwand für die zugehörigen Gewichte und den Rechenaufwand für die gewichteten Summen stark erhöht und schnell nicht mehr praktikabel wird, ähnlich wie bei der Verarbeitung von Bildern durch klassische Neuronale Netze statt durch CNNs. Sei die Sequenz k-elementig, wobei jedes Element der Sequenz über m Werte verfügt, die durch die Eingabeschicht an die n Elemente der ersten verborgenen Schicht weitergeleitet werden. Dann sind $k \cdot m \cdot n$ Gewichte notwendig. Darüber hinaus benötigen die bisher vorgestellten Neuronalen Netze Eingaben fester Länge, ausgenommen manche CNNs, bei denen Pooling Layers so eingesetzt werden können, dass Eingaben unterschiedlicher Größe auf dieselbe Größe skaliert werden können [GBC16]. Folglich würde man für Eingaben unterschiedlicher Länge separate Neuronale Netze benötigen. Ein weiteres Problem ist, dass solche Netze für jede Position in der Eingabesequenz separat lernen, da Parameter nicht geteilt werden. Dies hat zur Folge, dass viele abstrakte Konzepte durch das Neuronale Netz abhängig von der Position jedes mal neu erlernt werden müssen. All diese Probleme zeigen, dass die bisher vorgestellten Neuronalen Netze schlecht zur Verarbeitung sequentieller Daten geeignet sind und ein neues Modell benötigt wird.

Bei den Neuronalen Netzen, die explizit für die Verarbeitung von sequentiellen Daten ins Leben gerufen wurden, handelt es sich um die sogenannten *Recurrent Neural Networks*, zu Deutsch Rekurrente Neuronale Netze, die kurz als *RNNs* bezeichnet werden. Im Folgenden werden sie anhand der Arbeit von Goodfellow et al. [GBC16] vorgestellt. Das „Recurrent“ im Namen der RNNs bezieht sich darauf, dass bei dieser Form von Neuronalen Netzen der Datenfluss nicht nur von der Eingabe zur Ausgabe „nach vorne“ erfolgt, sondern eine Rückkopplung innerhalb des Netzes in Form von Zyklen vorhanden ist. Als Eingabe erhält ein RNN eine Sequenz von Daten $\mathbf{x} = \mathbf{x}^{(1)}\mathbf{x}^{(2)} \ldots \mathbf{x}^{(\tau)}$, wobei $\mathbf{x}^{(t)}$, $1 \leq t \leq \tau$, für die Daten der Eingabesequenz an Stelle t steht. Bei der Eingabe kann es sich beispielsweise um einen skalaren Wert, einen Vektor oder auch um eine Matrix von Werten wie bei CNNs handeln. Im Gegensatz zu den bisher vorgestellten Netzen ergibt sich die Ausgabe bezüglich der Eingabe jedoch nicht nur aus der aktuellen Eingabe $\mathbf{x}^{(t)}$ und den trainierbaren Parametern θ des Netzes, sondern auch aus einem verborgenen Zustand $\mathbf{h}^{(t-1)}$ der verborgenen Einheiten des Netzes, der sich aus den bisherigen Eingaben und verborgenen Zuständen ergibt. Sei f die Funktion, die das RNN für die Zustandsübergänge realisiert. Durch Eingabe von $\mathbf{x}^{(t)}$ kann ein neuer verborgener Zustand $\mathbf{h}^{(t)}$ beispielsweise wie folgt berechnet werden [GBC16]:

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}, \theta) \tag{3.22}$$

Die Berechnung des verborgenen Zustands $\mathbf{h}^{(t)}$ erfolgt folglich rekursiv. Durch diese Formulierung wird es möglich, Eingaben unterschiedlicher Länge zu verarbeiten, indem jeweils die durch das Neuronale Netz realisierte Funktion f mit denselben Parametern θ und dem aktuellen versteckten Zustand auf das aktuelle Element der Eingabesequenz angewendet wird. Anders ausgedrückt: Pro Schritt wird ein Element der Eingabesequenz in das Neuronale Netz eingegeben, die Parameter θ bleiben dabei dieselben und nur der versteckte Zustand ändert sich. Man kann $\mathbf{h}^{(t)}$ vereinfacht als eine Art Zusammenfassung der bisherigen Sequenz von Daten ansehen, wobei das Neuronale Netz entscheiden muss, welche

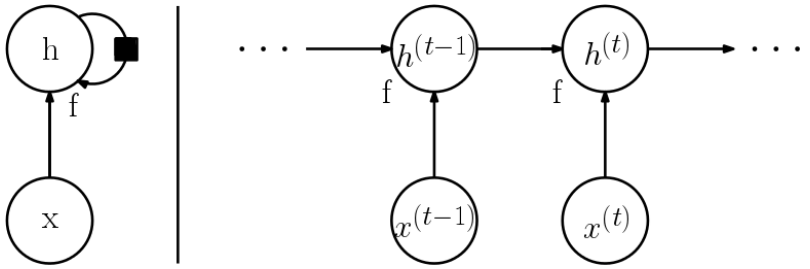


Abbildung 3.12.: Minimalistische Darstellung eines RNNs, das lediglich Eingaben erhält und den versteckten Zustand h aktualisiert [GBC16]. In der Praxis können auf Basis von h noch an verschiedenen Stellen Ausgaben erzeugt werden. Links ist eine kompakte Darstellung zu sehen, bei der durch f auf Basis von h und x der verborgene Zustand h aktualisiert wird. Das Quadrat steht dafür, dass eine zeitliche Verzögerung stattfindet und der Zustand h nicht sofort wieder verwendet wird. Rechts ist die ausgeschriebene Variante mit explizit dargestellten Zeitschritten zu sehen [GBC16]. Abbildung auf Basis von [GBC16].

Charakteristika erhalten bleiben und welche nicht, da die Eingabesequenz $\mathbf{x}^{(1)}$ ... $\mathbf{x}^{(t)}$ beliebiger Länge in einem Vektor $\mathbf{h}^{(t)}$ fester Länge „zusammengefasst“ wird [GBC16]. Auf Basis von $\mathbf{h}^{(t)}$ wird abschließend noch die Ausgabe des RNNs berechnet, hier können aus klassischen Neuronalen Netzen bekannte Schichten und Neuronen verwendet werden. Eine stark vereinfachte Darstellung eines RNNs kann Abbildung 3.12 entnommen werden.

Wie ein RNN tatsächlich aufgebaut ist, kann sich von Fall zu Fall stark unterscheiden. Besonders interessant sind dabei die verschiedenen Möglichkeiten, wie das RNN seine Ausgaben erzeugen kann. Bei den bisher vorgestellten Neuronalen Netzen wurde für eine Eingabe $\mathbf{x}$ eine zugehörige Ausgabe $\hat{\mathbf{y}}$ erzeugt. Bei RNNs gibt es durch die sequentielle Eingabe der $\mathbf{x}^{(t)}$ auch andere Möglichkeiten, die Ausgabe zu erzeugen: Statt für jedes Element der Eingabesequenz direkt eine Ausgabe zu erzeugen, können beispielsweise auch erst alle Elemente eingelesen werden. Die Erzeugung einer einzelnen Ausgabe erfolgt dann erst, wenn Informationen aller Elemente der Eingabesequenz vorhanden sind. Auch andere Ausgabemodalitäten sind denkbar. Wie genau die Ausgabe erfolgt, hängt von der Aufgabe des RNN ab.

Im Folgenden wird eine konkrete Form für verborgene Einheiten im Rahmen von RNNs vorgestellt, die in dieser Arbeit Verwendung findet.

3.1.9.1. Gated Recurrent Unit

Wie bereits erwähnt, kann der verborgene Zustand durch seine feste Größe bedingt nur einen Teil der Informationen der Eingabesequenz speichern – es muss folglich eine Entscheidung getroffen werden, welche Informationen in Zukunft relevant sein könnten und welche verworfen werden können. Hierfür werden im Rahmen dieser Arbeit *Gated Recurrent Units* (*GRUs*) verwendet, die erstmals im Rahmen der Arbeit von Cho et al. [CVMG+14] vorgestellt wurden, wenn auch noch nicht unter diesem Namen – auf Basis dieser Arbeit werden sie auch im Folgenden vorgestellt.

GRUs kontrollieren die Aktualisierung des verborgenen Zustands durch zwei sogenannte Gates, das *Reset Gate* r und das *Update Gate* z. Es sei nun eine Schicht mit n GRUs

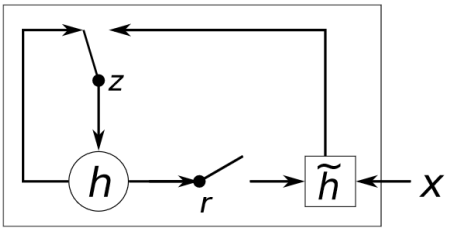


Abbildung 3.13.: Schematische Darstellung einer GRU [CVMG+14]: Das Reset Gate r bestimmt, wie stark der vorherige verborgene Zustand h in den neuen potentiellen verborgenen Zustand $\tilde{h}$ einfließt. Anschließend bestimmt das Update Gate z, wie sich der neue verborgene Zustand h durch seinen bisherigen Wert und den neuen potentiellen verborgenen Zustand zusammensetzt. Abbildung aus [CVMG+14].

gegeben und es stehe j, $1 \leq j \leq n$, für die j-te GRU. Für diese GRU sind Reset Gate $r_j^{(t)}$ und Update Gate $z_j^{(t)}$ mit Eingabe $\mathbf{x}^{(t)}$ zum Zeitpunkt t definiert wie folgt [CVMG+14]:

$$r_j^{(t)} = \sigma([\mathbf{W}_r \mathbf{x}^{(t)}]_j + [\mathbf{U}_r \mathbf{h}^{(t-1)}]_j) \tag{3.23}$$

$$z_j^{(t)} = \sigma([\mathbf{W}_z \mathbf{x}^{(t)}]_j + [\mathbf{U}_z \mathbf{h}^{(t-1)}]_j) \tag{3.24}$$

Die Variablen $\mathbf{W}_r$ und $\mathbf{U}_r$ bzw. $\mathbf{W}_z$ und $\mathbf{U}_z$ stehen hierbei für die Gewichtsmatritzen der gesamten GRU Layer in Hinsicht auf Reset Gate bzw. Update Gate; in Zeile j befindet sich der zum j-ten Neuron gehörende Gewichtsvektor. Daher wird nach der Multiplikation der jeweiligen Matrix mit der Eingabe oder dem vorangegangenen verborgenen Zustand das Ergebnis an j-ten Stelle ausgelesen. Bei $\mathbf{h}^{(t-1)}$ handelt es sich um den vorangegangenen verborgenen Zustand der gesamten Schicht; σ steht für die logistische Sigmoidfunktion. Bevor man zur Berechnung des neuen verborgenen Zustands übergeht, wird ein potentieller neuer verborgener Zustand $\tilde{h}_j^{(t)}$ berechnet [CGCB14]:

$$\tilde{h}_j^{(t)} = tanh([\mathbf{W}\mathbf{x}^{(t)}]_j + [\mathbf{U}(\mathbf{r}^{(t)} \odot \mathbf{h}^{(\mathbf{t-1})})]_j) \tag{3.25}$$

Bei $\mathbf{W}$ und $\mathbf{U}$ handelt es sich wiederum um erlernte Gewichtsmatritzen, $\mathbf{r}^{(t)}$ steht für den Vektor, der die Reset Gates aller Neuronen der Schicht enthält, und $\odot$ steht für eine elementweise Multiplikation. Das Reset Gate bestimmt dabei in Formel 3.25, wie stark der vorherige verborgene Zustand in den neuen potentiellen verborgenen Zustand mit einfließt. Anschließend wird der neue verborgene Zustand berechnet. Hier bestimmt das Update Gate, wie stark der potentielle neue verborgene Zustand $\tilde{h}_j^{(t)}$ und der alte verborgene Zustand $h_j^{(t-1)}$ in den neuen verborgenen Zustand $h_j^{(t)}$ einfließen [CVMG+14]:

$$h_j^{(t)} = z_j h_j^{(t-1)} + (1 - z_j)\tilde{h}_j^{(t)} \tag{3.26}$$

Abbildung 3.13 kann eine bildliche Darstellung der Funktionsweise entnommen werden. Wie sich Formel 3.23 für das Reset Gate und Formel 3.24 für das Update Gate entnehmen lässt, verfügt jede einzelne GRU über separate erlernte Gewichte, die beide Gates steuern. Dadurch können Neuronen derselben Schicht entweder dazu tendieren, ein „Langzeitgedächtnis“ zu entwickeln, indem der alte verborgene Zustand oft weitergereicht wird, ohne

stark vom neuen potentiellen verborgenen Zustand beeinflusst zu werden, während andere Neuronen in derselben Schicht ein „Kurzzeitgedächtnis“ entwickeln können [CVMG+14]. Das richtige Verhältnis kann das Neuronale Netz durch die Gewichte erlernen.

3.2. Optischer Fluss

Da die Verwendungen des optischen Flusses im Rahmen dieser Arbeit eine große Rolle spielt, wird im Folgenden anhand von Steinmüller [Ste08] kurz vorgestellt, worum es sich bei optischem Fluss handelt. Anschließend wird ein Überblick über die Verfahren gegeben, die im Rahmen dieser Arbeit zur Berechnung des optischen Flusses verwendet werden.

Möchte man Folgen von Bildern analysieren, kann es von Interesse sein, die stattfindende Verschiebung zu bestimmen, die von Bild zu Bild stattfindet. Es sei $I_1, I_2, ..., I_n$ eine Bildfolge, in der die Bilder nacheinander aufgenommen wurden. Im Idealfall soll für jeden Pixel des Bildes I_t mit Position (x, y) die neue Position (x^*, y^*) in Bild I_{t+1} bestimmt werden; der Vektor $d = (x^* - x, y^* - y)$ gibt dabei die Verschiebung an und wird als lokaler Verschiebungsvektor bezeichnet. Eine Menge von lokalen Verschiebungsvektoren zwischen demselben Bildpaar wird entsprechend als lokales Verschiebungsvektorfeld bezeichnet. Möchte man ein lokales Verschiebungsvektorfeld für alle Pixel eines Bildes zum nächsten Bild berechnen, stößt dies auf erhebliche Schwierigkeiten: Für jeden Punkt von Bild I_t muss der korrespondierende Punkt in Bild I_{t+1} gefunden werden, um jeweils den lokalen Verschiebungsvektor zu berechnen. Selbstverständlich können im Normalfall nicht für alle Punkte der Bilder Korrespondenzen bestimmt werden: Manche Punkte bewegen sich aus dem Bild heraus oder werden verdeckt, andere bewegen sich in das Bild hinein. Für andere Punkte, beispielsweise im Rahmen periodischer Muster, gibt es mehrere mögliche Korrspondenzen; es ist nicht garantiert, dass die richtige Korrespondenz gefunden werden kann. Auch anderweitige Faktoren können dazu führen, dass falsche oder keine Korrespondenzen gefunden werden.

Wegen der Schwierigkeit beim Finden exakter Korrespondenzen kann man dazu übergehen, das lokale Verschiebungsvektorfeld zu approximieren – dies geschieht im Rahmen des optischen Flusses. Als ein erster Schritt werden die Bilder der Bildfolge in Graustufenbilder umgewandelt, sofern sie nicht bereits in dieser Form vorliegen. Auf Basis der Grauwerte lokale Verschiebungsvektoren zu bestimmen, kann problematisch sein [Ste08]: Die Anzahl der möglichen Grauwerte ist stark beschränkt; hierdurch kommt es zu Mehrdeutigkeiten. Eine einfache Bedingung an den optischen Fluss kann dabei sein, dass die Verschiebung so angegeben wird, dass der Grauwert des Pixels vor der Verschiebung in Bild I_t dem Grauwert nach der Verschiebung in Bild I_{t+1} entspricht – diese Anforderung nennt sich Bildwerttreue. Sie reicht jedoch nicht aus, dass durch ihre Einhaltung der optische Fluss den tatsächlichen lokalen Verschiebungsvektoren entspricht. Neben der Mehrdeutigkeit gibt es noch weitere Probleme [Ste08]: Umwelteinflüsse wie veränderte Beleuchtungsverhältnisse zwischen aufeinanderfolgenden Bildern können die Grauwerte verändern und so dafür Sorgen, dass Vektoren, die die Bildwerttreue einhalten, nicht den tatsächlichen lokalen Verschiebungsvektoren entsprechen. Ebenso ist es nicht möglich, die Rotation von Objekten zu erfassen, deren Grauwerte und Form sich im Kamerabild nicht verändern, obwohl Bewegung stattfindet. Wegen solcher Probleme ist es nötig, weitere Bedingungen zu formulieren, damit eine brauchbare Approximation erzeugt werden kann. Die zusätzliche Annahme, die beispielsweise im Rahmen des bekannten Horn-Schunk-Verfahrens [HS81] getroffen wird,

ist die Forderung nach lokaler Glattheit – bei dieser Forderung wird verlangt, dass sich benachbarte Punkte ähnlich bewegen.

Über das Horn-Schunk-Verfahren hinaus exisiteren weitere Methoden zur Berechnung des optischen Flusses. Im Rahmen dieser Arbeit wird in weiten Teilen das Verfahren von Brox et al. [BBPW04] zur Bestimmung des optischen Flusses eingesetzt, welches in der Arbeit von Varol et al. [VLS18] im Vergleich mit einigen anderen Verfahren zur Berechnung des optischen Flusses als der beste auf dem Gebiet der Aktionserkennung bei Verwendung von C3D identifiziert wurde. Zu Vergleichszwecken wird zusätzlich optischer Fluss herangezogen, der mit Hilfe des Neuronalen Netzes FlowNet2 [IMS+17] berechnet wurde – dies ist ein Verfahren, das nicht im Vergleich von Varol et al. betrachtet wurde. Im Nachfolgenden werden diese beiden Arten der Bestimmung des optischen Flusses näher vorgestellt.

3.2.1. Verfahren von Brox et al.

Wie bereits erwähnt wird im Rahmen dieser Arbeit zur Bestimmung des optischen Flusses primär das Verfahren von Brox et al. aus der Arbeit „High accuracy optical flow estimation based on a theory for warping“ [BBPW04] verwendet, welche im Folgenden zur Vorstellung des Verfahrens herangezogen wird.

Wie bereits beim Horn-Schunk-Verfahren müssen bei der Berechnung des optischen Fluses mit dem Verfahren von Brox et al. zusätzliche Annahmen zur Bildwerttreue getroffen werden, um eine gute Approximation der tatsächlichen Bewegung zu erhalten. Im Folgenden werden diese zusätzlich getroffenen Annahmen vorgestellt. Man definiere nun analog zu Brox et al. [BBPW04] die Bildsequenz, die betrachtet wird, als Funktion $I : \Omega \subset \mathbb{R}^3 \to \mathbb{R}$, die Tripel $\mathbf{x} = (x, y, t)$ aus gültigen Koordinaten x und y sowie Zeitpunkt t innerhalb der Bildsequenz auf den zugehörigen Grauwert abbildet. Die Verschiebungsvektoren werden entsprechend auch als Tripel $w = (u, v, 1)$ mit den Verschiebungen um u und v angegeben. Der Wert an dritter Stelle gibt die zeitliche Änderung an und ist als 1 festgesetzt, da Verschiebungen jeweils nur zum nachfolgenden Bild bestimmt werden sollen. Unter diesen Bedingungen lässt sich die Annahme der Bildwerttreue wie folgt definieren [BBPW04]:

$$I(x, y, t) = I(x + u, y + v, t + 1) \tag{3.27}$$

Wie bereits festgestellt ist diese Annahme empfindlich gegenüber der Änderung von Grauwerten von Punkten zweier aufeinanderfolgender Bilder, die eigentlich zusammengehören, auch wenn diese Änderung nur minimal ist. Daher führen Brox et al. die Annahme ein, dass der Gradient des Grauwertes bezüglich der räumlichen Ableitung nach x und y nach der Verschiebung derselbe sein muss – Grauwertänderungen haben auf diese Annahme keinen Einfluss; formal wird sie wie folgt beschrieben [BBPW04]:

$$\nabla I(x, y, t) = \nabla I(x + u, y + v, t + 1) \quad \text{mit } \nabla = (\partial_x, \partial_y)^T \tag{3.28}$$

Über diese Annahme hinaus wird noch eine weitere getroffen: die Glattheit der Bewegung. Anders ausgedrückt: Benachbarte Pixel bewegen sich ähnlich. Diese Annahme ist beispielsweise in Fällen hilfreich, in denen Ausreißer auftreten oder Gradienten verschwinden [BBPW04]. Es gibt jedoch auch Fälle, in denen die Annahme einer strikten Glattheit nicht korrekt wäre, beispielsweise beim Übergang eines im Vordergrund befindlichen, beweglichen Objektes zu einem statischen Hintergrund. Aus diesem Grund wird lediglich angenommen, dass das Flussfeld stückweise glatt ist [BBPW04].

Nun gilt es eine Funktion aufzustellen, mit der erreicht werden kann, dass der optische Fluss die getroffenen Annahmen möglichst einhält. Hierzu definieren Brox et al. Energiefunktionale, die in einem zu minimierenden Ausdruck resultieren. Um den Einfluss von Ausreißern zu begrenzen wird darüber hinaus bei der Definition eine konvexe Funktion $\Psi(s^2)$ verwendet, die wie folgt definiert ist, wobei das in diesem Rahmen verwendete ε lediglich eine kleine, positive Konstante darstellt [BBPW04]:

$$\Psi(s^2) = \sqrt{s^2 + \varepsilon^2} \tag{3.29}$$

Mit Hilfe dieser Funktion werden zwei Energiefunktionale definiert. Das erste Energiefunktional E_{Daten} vereint die Annahme der Bildwerttreue und des Erhalts der Gradienten in sich [BBPW04]:

$$E_{Daten}(u, v) = \int_{\Omega} \Psi(|I(\mathbf{x} + \mathbf{w}) - I(\mathbf{x})|^2 + \gamma|\nabla I(\mathbf{x} + \mathbf{w}) - \nabla I(\mathbf{x})|^2)d\mathbf{x} \tag{3.30}$$

Mithilfe des Parameters γ lässt sich dabei der Einfluss der Komponenten zueinander steuern. Das zweite Energiefunktional $E_{Glattheit}$ bringt die Annahme der stückweisen Glattheit mit ein [BBPW04]:

$$E_{Glattheit}(u, v) = \int_{\Omega} \Psi(|\nabla_3 u|^2 + |\nabla_3 v|^2)d\mathbf{x} \quad \text{mit } \nabla_3 = (\partial_x, \partial_y, \partial_t)^T \tag{3.31}$$

Einen Sonderfall für diese Formulierung stellt das Vorhandensein von Bildfolgen mit lediglich 2 Bildern dar, in diesem Fall wird ∇ anstelle von ∇_3 verwendet [BBPW04]. Beide Energiefunktionale sind so formuliert, dass eine Abweichung von den jeweiligen Annahmen eine Erhöhung der Energie bedeutet; entsprechend kann durch Minimierung der Energien ein optischer Fluss gefunden werden, der die Annahmen möglichst genau einhält. Hierzu müssen die beiden separaten Energiefunktionale noch zusammengebracht werden. Unter Verwendung eines Parameters $\alpha > 0$, der den Einfluss der Glättung steuert, ergibt sich der Ausdruck, der beim Verfahren von Brox et al. zur Bestimmung des optischen Flusses minimiert wird, wie folgt [BBPW04]:

$$E(u, v) = E_{Daten} + \alpha E_{Glattheit} \tag{3.32}$$

Mathematische Details, wie genau der Energieterm $E(u, v)$ minimiert werden kann, können Brox et al. [BBPW04] entnommen werden.

3.2.2. FlowNet2

Eine Alternative zur Bestimmung des optischen Flusses ist das Neuronale Netz FlowNet2 von Ilg et al. aus der Arbeit „FlowNet 2.0: Evolution of Optical Flow Estimation with Deep Networks“ [IMS+17], welche im Folgenden zur Vorstellung von FlowNet2 herangezogen wird.

Im Gegensatz zu klassischen Methoden wie das zuvor vorgestellte Verfahren von Brox et al. müssen für die Berechnung des optischen Flusses keine Annahmen wie die Äquivalenz von Grauwerten oder die lokale Glattheit getroffen werden. Stattdessen erlernt das Netzwerk mit Hilfe von annotierten Trainingsbeispielen, den optischen Fluss zu bestimmen. Hierfür

setzt FlowNet2 auf der ursprünglichen Arbeit zu FlowNet von Dosovitskiy et al. [DFI+15] auf.

In dieser Arbeit wurden zwei Architekturen für Neuronale Netze zur Extraktion des optischen Flusses vorgestellt: FlowNetS und FlowNetC. Diese werden bei FlowNet2 weiterverwendet und um eine leicht veränderte Architektur auf Basis von FlowNetS ergänzt, FlowNet2-SD genannt. Beim vollständigen FlowNet2 kommen parallel zwei Teilnetzwerke zum Einsatz, von denen eines auf die Extraktion des optischen Flusses für große Verschiebungen und eines auf die Extraktion des optischen Flusses für kleine Verschiebungen spezialisiert ist; diese werden abschließend durch ein kleines Neuronales Netz fusioniert.

Das Teilnetzwerk zur Bestimmung der großen Verschiebungen setzt sich aus drei in Reihe geschalteten FlowNet-Varianten zusammen: Zuerst kommt ein FlowNetC zum Einsatz, gefolgt von zwei FlowNetS. Als Eingabe dienen Bildpaare; diese erhält nicht nur das erste, sondern jedes der in Reihe geschalteten Netzwerke. Das zweite und dritte Netzwerk erhalten darüber hinaus die Ausgabe des jeweils vorangegangenen Netzwerks als zusätzliche Eingabe; es kommt außerdem eine Zwischenverarbeitung in Form von Warping auf einem der Eingabebilder auf Basis des bis dahin bestimmten optischen Flusses zum Einsatz. Das resultierende verarbeitete Bild sowie die noch vorhandene Abweichung vom anderen Bild des Bildpaares dienen als weitere Eingaben in das nächste Netzwerk. Für die Berechnung kleiner Verschiebungen wird parallel FlowNet2-SD eingesetzt – als Eingabe dient dasselbe Bildpaar. Die Ausgabe von FlowNet2-SD und die des letzten Netzwerks der drei in Reihe geschalteten FlowNet-Varianten wird anschließend einer weiteren Verarbeitung unterzogen, bei der jeweils die Größe des optischen Flusses und wiederum die Abweichung nach dem Warping bestimmt wird; zusammen mit dem jeweils bestimmten optischen Fluss dienen diese Daten als Eingabe in das kleine Neuronale Netz, das zur Fusion eingesetzt wird und die endgültige Ausgabe erzeugt. Die gesamte Architektur bildet FlowNet2; mit ihr lassen sich vergleichbare Ergebnisse bei der Bestimmung des optischen Flusses im Vergleich zu anderen state-of-the-art Verfahren erzielen. Daher wird optischer Fluss, der auf Basis von FlowNet2 bestimmt wurde, im Rahmen dieser Arbeit als Alternative zum optischen Fluss, der mit dem Verfahren nach Brox et al. bestimmt wurde, untersucht. Weitere Details, beispielsweise über das Training oder den detaillierten Aufbau des Netzwerks, können Ilg et al. [IMS+17] entnommen werden.

4. Modelle

In diesem Kapitel werden die verschiedenen Modelle vorgestellt, die im Rahmen dieser Arbeit zum Einsatz kommen. Hierbei dienen die verwendeten Modelle des C3D-Netzwerks und des SST-Netzwerks als Grundlage, auf deren Basis unterschiedliche Two-Stream-Netzwerke entworfen und vorgestellt werden.

4.1. Klassisches C3D-Modell

Das originale C3D-Modell [TBF+15] wird in weiten Teilen dieser Arbeit eingesetzt; es dient darüber hinaus als Grundlage für die entworfenen Two-Stream-Netzwerke. Das C3D-Modell wurde bereits in Abschnitt 2.1 bei der Vorstellung der Originalarbeit zum C3D-Netzwerk eingeführt und in Abbildung 2.2 graphisch dargestellt. Für den Trainingsvorgang wird das Modell genau so verwendet, wie es dort vorgestellt wurde. Die Extraktion der C3D-Features für jeweils 16 aufeinanderfolgende Bilder läuft leicht abgewandelt. Statt die Aktivierungen von der ersten Fully Connected Layer, in Abbildung 2.2 mit *fc6* bezeichnet, zu extrahieren und direkt auf diese eine L_2-Normalisierung anzuwenden, werden die Aktivierungen der ersten und zweiten Fully Connected Layer extrahiert, ohne dass anschließend direkt eine L_2-Normalisierung angewendet wird. Es wird ebenfalls kein Durchschnitt über alle so extrahierten Aktivierungen für ein Video gebildet, sie bleiben separat erhalten. Ob auf diesen Daten eine weitere Verarbeitung wie die erwähnte L_2-Normalisierung oder beispielsweise eine Hauptkomponentenanalyse stattfindet, wird im Rahmen des C3D-Netzwerkes vorerst offen gelassen; ebenso, welcher der beiden so extrahierten Featurevektoren weiterverwendet werden soll. Als Ausgabe entstehen folglich zwei 4096-elementige Vektoren für jeweils 16 Bilder. Eine Darstellung ist Abbildung 4.1 zu entnehmen.

Das C3D-Netzwerk wird vor dem Einsatz des SST-Netzwerks verwendet; wie beschrieben läuft die Extraktion der Features und das Training separat vom SST-Netzwerk ab. Das SST-Netzwerk arbeitet im Anschluss auf Basis der durch das C3D-Netzwerk extrahierten Features. Pro Video werden C3D-Features für alle vollen Blöcke von 16 Bildern ohne Überlappung der Blöcke extrahiert.

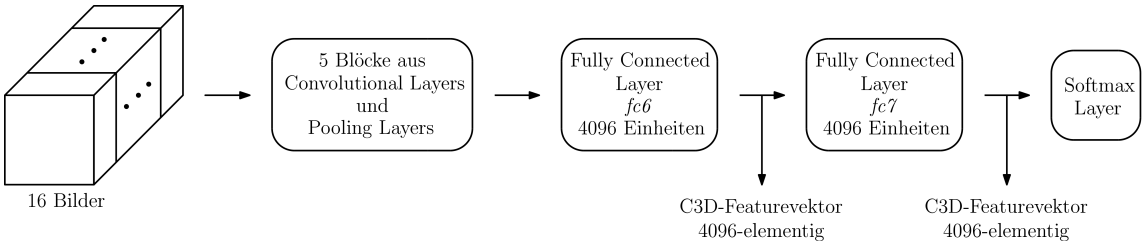


Abbildung 4.1.: Darstellung der Extraktion von C3D-Featurevektoren durch das C3D-Netzwerk. Die Aktivierungen der fc6- und der fc7-Schicht werden ausgelesen und bilden zwei separate C3D-Featurevektoren, die auf 16 aufeinanderfolgenden Eingabebildern basieren; es findet vorerst keine weitere Verarbeitung der C3D-Featurevektoren statt. Diese Variante lässt für Experimente offen, ob die C3D-Featurevektoren der fc6- oder fc7-Schicht weiterverwendet werden und ob eine weitere Verarbeitung der C3D-Featurevektoren stattfindet oder nicht.

4.2. Klassisches SST-Modell

Das klassische SST-Modell, das im Rahmen dieser Arbeit verwendet wird, unterscheidet sich vom Modell, das bei der Vorstellung der Originalarbeit in Kapitel 2.2 erläutert wird. Die Abbildung 2.3 zeigt das C3D-Netzwerk als Bestandteil der dreistufigen Architektur des SST-Netzwerks; als Eingabe für das SST-Netzwerk dienen Videodaten. Da das C3D-Netzwerk dort als reine Komponente zur Extraktion von Features fungiert, mit vortrainierten Gewichten initialisiert und nicht zusammen mit dem SST-Netzwerk trainiert wird, wird es im Rahmen dieser Arbeit nicht Teil des SST-Modells. Stattdessen werden dem SST-Netzwerk während des Trainings und während Tests vorab extrahierte C3D-Featurevektoren als Eingabe zur Verfügung gestellt. Für das SST-Modell bleiben folglich Sequence Encoder und Ausgabemodul übrig. Ebenfalls wird die Eingabe für den Sequence Encoders nicht auf die Aktivierungen der zweiten Fully Connected Layer des C3D-Netzwerks festgelegt, die zuvor einer Hauptkomponentenanalyse unterzogen wurden. Das Modell sieht stattdessen vor, dass kein fester Wert für die Eingabegröße festgelegt wird - es muss lediglich pro Zeitschritt ein Vektor mit n Elementen eingegeben werden, wobei n bezüglich aller Eingaben in das Netz gleich sein muss. Bei Tests können so verschiedene Größen für alle C3D-Featurevektoren untersucht werden. Die Anwendung der Hauptkomponentenanalyse und dadurch in ihrer Größe reduzierter C3D-Featurevektoren wird hierdurch optional, ebenso wie die Größe der C3D-Featurevektoren, falls die Hauptkomponentenanalyse angewendet wird. Es findet auch keine Festlegung weiterer Vorverarbeitungsschritte statt, diese können optional für verschiedene Trainings und Tests eingesetzt werden. Abbildung 4.2 zeigt das neue Modell ohne das C3D-Netzwerk als Visual Encoder.

Die C3D-Featurevektoren, die als Eingabe des SST-Netzwerks fungieren, werden durch den Sequence Encoder verarbeitet, der aus Schichten von GRUs besteht. In der Arbeit von Buch et al. [BES+17] wird der Aufbau des Sequence Encoders nicht näher beleuchtet; die Arbeit veweist jedoch auf ein Git-Repository, dem eine Implementierung entnommen werden kann. Dort kommt eine GRU Layer mit 256 verborgenen Einheiten zum Einsatz; es werden 500-elementige C3D-Featurevektoren als Eingabe erwartet. Da Alternativen zu dieser Architektur in der Arbeit jedoch nicht untersucht wurden, wird vorerst die Anzahl

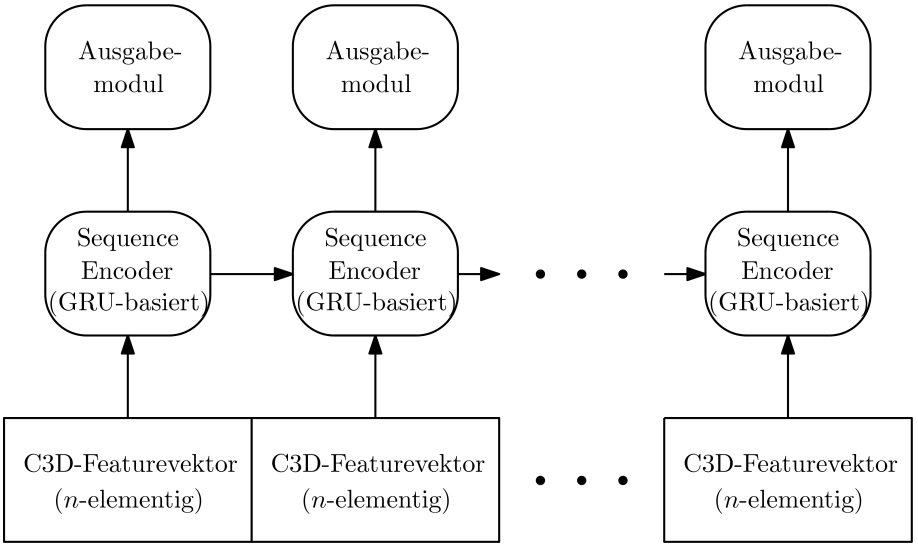


Abbildung 4.2.: Darstellung des SST-Netzwerks ohne den Visual Encoder. Als Eingabe dienen nun vorab extrahierte C3D-Featurevektoren, Sequence Encoder und Ausgabemodul finden nach wie vor Verwendung.

an verborgenen Einheiten pro Schicht sowie die Anzahl der Schichten und die Anzahl der Elemente der C3D-Featurevektoren variabel gehalten, damit prinzipiell verschiedene Varianten evaluiert werden können. Da als Alternative zu GRUs bereits LSTMs betrachtet und als unterlegen eingestuft wurden, findet bereits eine Festlegung auf GRUs statt. Das sich anschließende Ausgabemodul wird so beibehalten, wie es von Buch et al. [BES+17] vorgestellt wurde: Es wird eine Fully Connected Layer mit logistischer Sigmoidfunktion als nichtlineare Aktivierungsfunktion verwendet, die Anzahl der Neuronen der Schicht entsprechen der Anzahl an Zeitfenstern k, für die eine Ausgabe erzeugt wird; eine Veranschaulichung der Ausgabe findet sich in Abbildung 2.3. Bezüglich der Größe von k wurde eine Untersuchung im Rahmen der Zusatzmaterialen[1] zur Arbeit von Buch et al. durchgeführt, auf die im in der Arbeit referenzierten Git-Repository verwiesen wird. Diesen Zusatzmaterialen kann entnommen werden, dass sich $k = 32$ als geeignet herausgestellt hat; diese Wahl wird im Rahmen der hier verwendeten Architektur beibehalten. Zur Erzeugung der Temporal Action Proposals für ein Video wird das SST-Netzwerk sequentiell mit allen C3D-Features für ein Video versorgt; pro C3D-Featurevektor werden für $k = 32$ Zeitfenster Konfidenzwerte erzeugt. Sind alle C3D-Featurevektoren für ein Video verarbeitet, findet auf der Gesamtzahl der betrachteten Zeitfenster, denen Konfidenzwerte zugewiesen wurden, eine Nachverarbeitung statt, indem Zeitfenster mit zu geringem Konfidenzwert verworfen werden und eine Non-Maxima-Suppression durchgeführt wird. Eine detaillierte Darstellung des resultierenden SST-Modells findet sich in Abbildung 4.3.

4.3. Two-Stream-Modelle

Im Folgenden werden die Two-Stream-Modelle, die in dieser Arbeit verwendet werden, auf Basis der beiden klassischen Modelle aus Abschnitt 4.1 und 4.2 vorgestellt. Diese Modelle

[1] `https://drive.google.com/file/d/0B_-dKvCH2VL7dGV1ankxWnJVQmM/view`

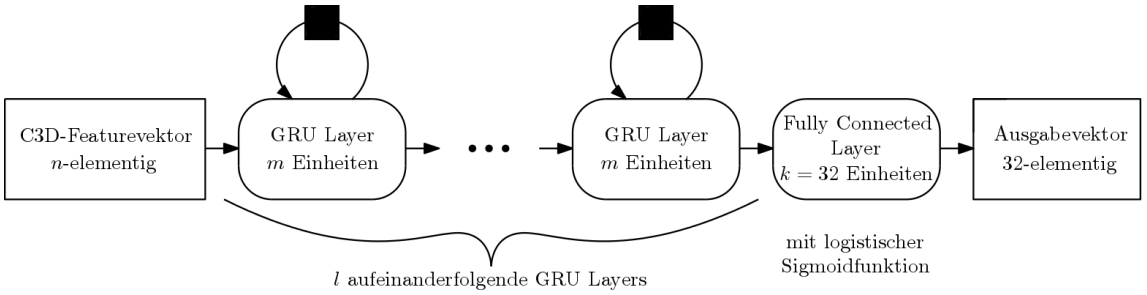


Abbildung 4.3.: Detaillierte Darstellung der SST-Architektur, die im Rahmen dieser Arbeit verwendet wird. Als Eingabe pro Zeitschritt dient ein C3D-Featurevektor mit n Elementen. Es schließen sich l GRU Layers an, wobei jede dieser Schichten über m verborgene Einheiten verfügt - m ist einheitlich für alle diese Schichten. Abschließend erzeugt eine Fully Connected Layer mit logistischer Sigmoidfunktion als nichtlineare Aktivierungsfunktion die Ausgabe, sie verfügt über 32 Einheiten - eine für jedes Zeitfenster, denen durch die Ausgabe ein Konfidenzwert zugeordnet wird.

werden immer als Kombination von C3D-Modell und SST-Modell dargestellt, zeigen also die komplette Verarbeitung von den Eingabevideos bis hin zu den Temporal Action Proposals. Wie bei der klassischen Kombination von C3D- und SST-Netzwerk werden für jedes Video sequentiell nicht überlappende Blöcke von jeweils 16 zusammenhängenden Bildern verarbeitet. Bei den Two-Stream-Modellen unterscheidet sich jeweils die Art und Weise, wie die zwei Streams fusioniert werden. Das Training erfolgt jeweils nicht Ende-zu-Ende; das C3D-Teilnetz wird separat vom SST-Teilnetz trainiert.

Der Entwurf von Two-Stream-Modellen unter Verwendung der genannten Modelle wird durch die jüngsten Entwicklungen auf dem Gebiet der Temporal Action Proposal Generierung und anderer Gebiete motiviert. Zur Temporal Action Proposal Generierung setzt man in jüngsten Methoden zumeist auf die Verarbeitung der Videodaten durch 3D Convolutional Neural Networks (3D ConvNets) [EHNG16, BES$^+$17, GYN17, GYS$^+$17], teils kommt auch optischer Fluss im Rahmen eines Two-Stream-Netzwerks [LZS17, GYN17] zum Einsatz - bei den referenzierten Arbeiten wird der optische Fluss jedoch nicht mit Hilfe von 3D-Faltungen verarbeitet, lediglich 2D-Faltungen finden Anwendung. Erst die Arbeiten [CVS$^+$18, NLPH18] auf dem Gebiet der Temporal Action Localization, bei der die Temporal Action Proposal Generierung eine Teilaufgabe darstellt, verwenden Two-Stream 3D ConvNets auf Bilddaten und optischem Fluss. Bei dem dafür eingesetzten Netzwerk handelt es sich in beiden Fällen um das I3D-Netzwerk, nicht um das hier verwendete C3D-Netzwerk. Auch auf dem Gebiet der Aktionserkennung werden Two-Stream 3D ConvNets auf Bilddaten und dem optischen Fluss bereits erfolgreich eingesetzt [CZ17, VLS18, KT18], die beiden letzten referenzierten Arbeiten setzen Two-Stream-Varianten des C3D-Netzwerks ein; auf diesem Gebiet werden jedoch keine Temporal Action Proposals generiert. Auf Basis all dieser Entwicklungen ist es der nächste logische Schritt, die bestehende Kombination aus C3D-Netzwerk und SST-Netzwerk zu einem Two-Stream-Netzwerk weiterzuentwickeln.

4.3.1. Variante 1: Mid-Fusion durch Kombination der C3D-Features

Die Variante 1 der Fusion wurde durch die Arbeit von Khong et al. [KT18] auf dem Gebiet der Human Action Recognition inspiriert. Eine der dort untersuchten Architekturen sieht die Extraktion von C3D-Featurevektoren von der fc6-Schicht zweier separater C3D-Netzwerke vor, von denen eines für Bilddaten und eines für den optischen Fluss vorgesehen ist. Diese C3D-Featurevektoren werden anschließend unter anderem konkateniert und dienen als Eingabe für eine lineare SVM.

Auf dieser Grundlage wird auch bei Variante 1 eines Two-Stream-Modells, das im Rahmen dieser Arbeit entwickelt wurde, die Konkatenation der C3D-Featurevektoren zur Fusion der separaten Streams verwendet. Es werden zwei separate C3D-Netzwerke eingesetzt – eines, das auf den originalen Bilddaten arbeitet und eines, das auf Bildern arbeitet, die den optischen Flusses visualisieren. Es dienen jeweils 16 aufeinanderfolgende Bilder als Eingabe. Das Bild des optischen Flusses an Stelle i der Eingabe ist dabei mit dem originalen Bild an Stelle i der Eingabe assoziiert. Es visualisiert den optischen Fluss, der zu dem assoziierten Bild führt; das assoziierte Bild war also das zweite Bild des Bildpaars, das für die Berechnung des optischen Flusses verwendet wurde. Das Modell sieht vor, dass beide C3D-Netzwerke separat trainiert werden; dabei unterscheidet sich jeweils nur die Eingabe in das C3D-Netzwerk: Die originalen Bilder und die Bilder des optischen Flusses. Anschließend werden C3D-Featurevektoren von der fc6- bzw. fc7-Schicht beider Netzwerke für die originalen Bilder und die Bilder des optischen Flusses erzeugt. Diese C3D-Featurevektoren können anschließend optional einer Nachverarbeitung unterzogen werden und werden abschließend dadurch kombiniert, dass sie konkateniert werden. Diese kombinierten C3D-Featurevektoren dienen als Eingabe für das SST-Netzwerk, welches auf deren Basis trainiert wird; die Evaluation findet auch auf Basis der kombinierten C3D-Featurevektoren statt. Für ein leichteres Verständnis zeigt Abbildung 4.4 den schematischen Aufbau des vorgeschlagenen Modells.

4.3.2. Variante 2: Mid-Fusion in der fc7-Schicht des C3D-Netzwerks

Bei Variante 1 werden die zwei Streams „von Hand“ zusammengeführt, indem der zusammengesetzte C3D-Featurevektor außerhalb der Neuronalen Netze erzeugt wird. Eine mögliche logische Konsequenz hierzu stellt eine Kombination der separaten C3D-Netzwerke in einer der späteren Schichten dar. Auf dem Gebiet der Aktionserkennung wurde beispielsweise von Varol et al. [VLS18] ein Two-Stream-C3D-Netzwerk verwendet, das aus zwei separaten C3D-Netzwerken für originale Bilder und optischen Fluss besteht, die in der fc6-Schicht fusioniert werden und so ein Two-Stream-Netzwerk bilden.

Die Idee der Fusion zweier C3D-Netzwerke durch eine Fully Connected Layer bildet die Grundlage für Variante 2. Die Eingabe der einzelnen Streams gestaltet sich wie in Abschnitt 4.3.1. Die einzelnen Streams der beiden C3D-Netzwerke laufen im Anschluss bis einschließlich zur fc6-Schicht separat. Es schließt sich eine einzelne fc7-Schicht an, die zur Fusion beider Netzwerke verwendet wird, indem sie so organisiert wird, dass sie als Eingabe 8192 statt 4096 Eingaben erhält - die Aktivierungen der beiden fc6-Schichten der separaten Streams. Als Ausgabe erzeugt die fc7-Schicht wie gehabt 4096 Aktivierungen, die als Eingabe in die nachfolgende Softmax-Schicht dienen und als C3D-Featurevektor extrahiert werden können - im Rahmen dieser Architektur werden die C3D-Featurevektoren nur von der fc7-Schicht extrahiert, um anschließend als Eingabe für das SST-Netzwerk zu

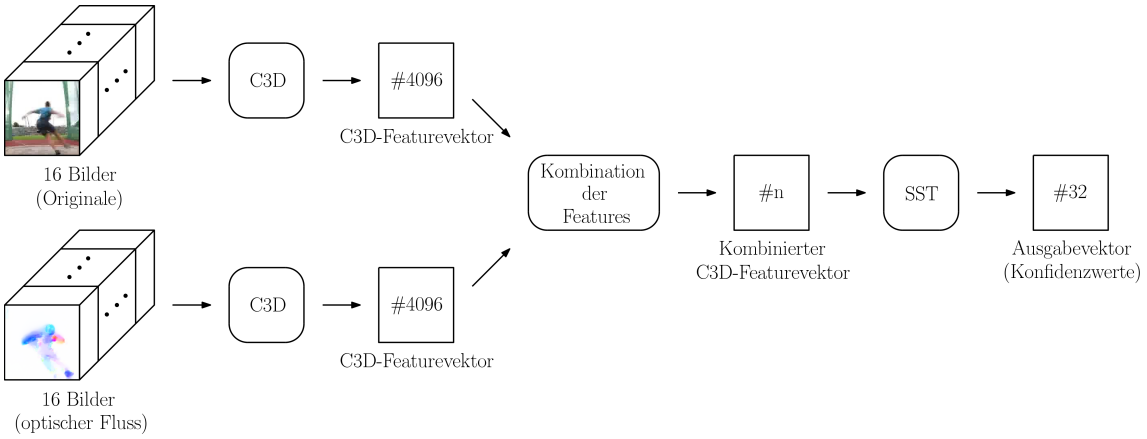


Abbildung 4.4.: Schematische Darstellung von Variante 1 der Fusion. Für zwei separate Streams werden C3D-Featurevektoren extrahiert - einmal für jeweils 16 originale Bilder, einmal für 16 Bilder, die den zugehörigen optischen Fluss visualisieren. Das Bild des optische Flusses an Stelle i visualisiert den optischen Fluss, für dessen Berechnung das originale Bild an Stelle i als zweites Bild des benötigten Bildpaares fungiert hat. Vor der Eingabe in das SST-Netzwerk durchlaufen die beiden C3D-Featurevektoren ein Kombinationsmodul, das optionale Verarbeitungsschritte wie beispielsweise eine L_2-Normalisierung vornimmt. Nach der optionalen Verarbeitung werden in diesem Modul beide (eventuell vorverarbeitete) C3D-Featurevektoren konkateniert, um so einen einzelnen Vektor der Größe n zu erzeugen - n muss hierbei bei jedem Schritt des Netzwerks gleich sein. Die Größe ist variabel angegeben, da im Rahmen der optionalen Vorverarbeitung theoretisch auch die Anzahl der Elemente beider Vektoren reduziert werden kann. Der kombinierte C3D-Featurevektor dient anschließend als Eingabe für das SST-Netzwerk, welches auf dessen Basis den Ausgabevektor bestimmt, der die Konfidenzwerte für 32 Zeitfenster enthält. Eingabebild aus THUMOS'14 [JLZ+14].

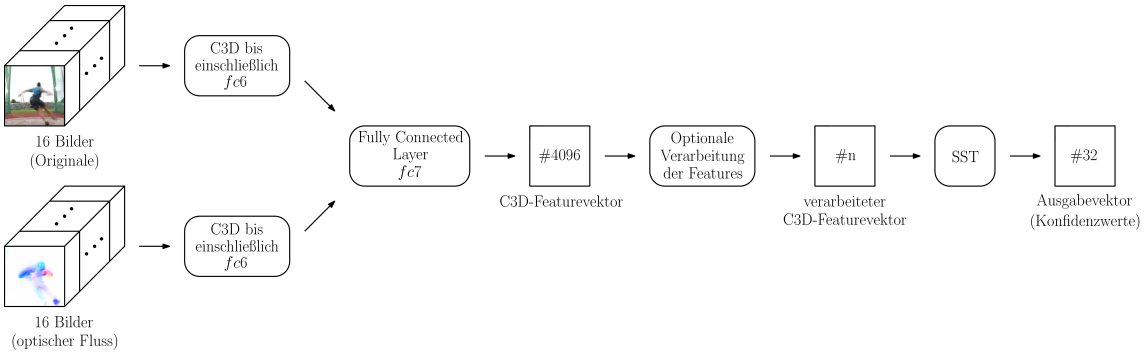


Abbildung 4.5.: Schematische Darstellung von Variante 2 der Fusion. Die Fusion der einzelnen Streams findet im Gegensatz zu Variante 1 bereits vor der Extraktion der C3D-Features statt, indem zwei separate C3D-Netzwerke durch eine gemeinsame fc7-Schicht zusammengeführt werden. Die optionale Verarbeitung der C3D-Featurevektoren ist explizit angegeben, bei Abbildung 4.4 der Variante 1 war sie im Modul zur Fusion der Vektoren enthalten. Eingabebild aus THUMOS'14 [JLZ+14].

dienen, da bei der fc6-Schicht noch nicht fusioniert wurde. Das Training ist so vorgesehen, dass zuerst zwei klassische C3D-Netzwerke trainiert werden, eines für die originalen Bilddaten und eines für die Bilder des zugehörigen optischen Flusses. Diese separaten C3D-Netzwerke dienen im Anschluss zur Initialisierung für das beschriebene Modell mit der Fusion in der fc7-Schicht – anschließend wird noch einmal trainiert. Die extrahierten C3D-Featurevektoren der fc7-Schicht können anschließend eine optionale Nachverarbeitung, wie beispielsweise eine L_2-Normalisierung oder eine Hauptkomponentenanalyse, durchlaufen. Die hieraus resultierenden verarbeiteten C3D-Featurevektoren dienen dann dem Training und der Evaluatiuon des SST-Netzwerks. Eine Veranschaulichung des gesamten Modells kann Abbildung 4.5 entnommen werden.

4.3.3. Variante 3: Late-Fusion durch Bilden eines gewichteten Durchschnitts im SST-Netzwerk

Bisher sind nur Varianten der Fusion beider Streams vorgestellt worden, die bereits vor dem SST-Netzwerk stattfinden – entsprechend ist es logisch, auch Modelle in die Untersuchungen mit einzubeziehen, bei denen die Fusion später stattfindet. Eine angemessene Grundlage für ein solches Modell liefern Wang et al. mit ihrer Arbeit zu Temporal Segment Networks [WXW+16] auf dem Gebiet der Aktionserkennung. Dort werden beide Streams des entwickelten TSNs bis zum Ende separat gehalten, die Bewertungen der einzelnen Klassen durch die separaten Streams werden abschließend mittels eines gewichteten Durchschnitts fusioniert.

Diese grundlegende Idee lässt sich auf zwei separate SST-Netzwerke übertragen, indem nach der Ausgabe der jeweiligen Konfidenzwerte der gewichtete Durchschnitt über die jeweils zusammengehörenden Konfidenzwerte gebildet wird. Hierauf basiert Variante 3 der Fusion. Es werden zuerst zwei separate C3D-Netze trainiert, eines für die originalen Bilddaten und eines für die Bilder des zugehörigen optischen Flusses. Von beiden Netzwerken werden anschließend die C3D-Featurevektoren extrahiert; ob von der fc6- oder fc7-Schicht

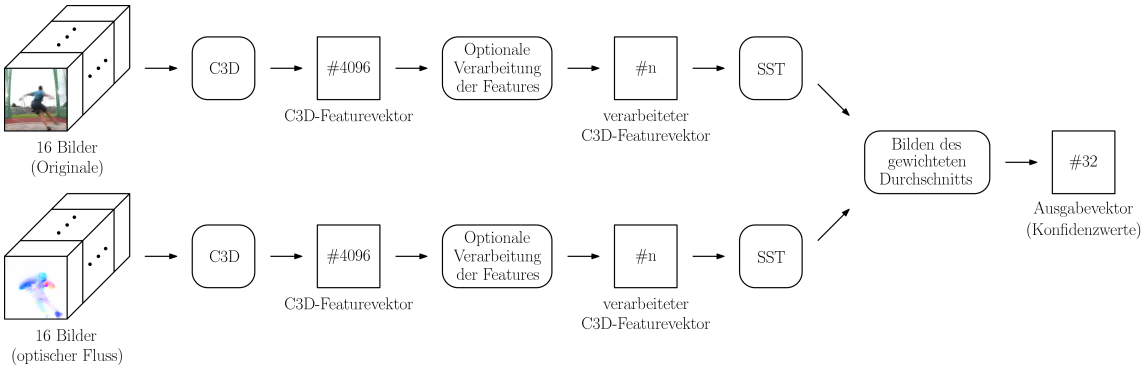


Abbildung 4.6.: Schematische Darstellung der Variante 3 der Fusion. Beide Streams bleiben im Gegensatz zu den bisherigen Varianten bis zum Ende separat; über die Ausgaben der einzelnen Streams wird ganz am Ende ein gewichteter Durchschnitt gebildet. Eingabebild aus THUMOS'14 [JLZ+14].

bleibt bei dieser Architektur vorerst offen. Die entsprechenden C3D-Featurevektoren werden vorab extrahiert und können wieder optionale Verarbeitungsschritte durchlaufen; sie dienen anschließend als Eingabe für zwei separate SST-Netzwerke, die auf deren Basis trainiert werden – eines für die C3D-Featurevektoren der originalen Bilddaten und eines für die C3D-Featurevektoren der Bilder des zugehörigen optischen Flusses. Nach dem Training der separaten SST-Netzwerke werden beide so zusammengeführt, dass ein gewichteter Durchschnitt über die Konfidenzwerte, die durch beide Netze für jeweils 32 Zeitfenster ausgegeben werden, berechnet wird. Die so vortrainierten und zusammengeführten SST-Netzwerke können auf diese Art und Weise anschließend auch noch zusammen trainiert werden. Wie bereits bei den vorangegangenen Varianten und klassischen Modellen werden eventuelle Zwischenverarbeitungsschritte für die C3D-Featurevektoren vor der Eingabe in die SST-Netzwerke nicht festgelegt. Eine Abbildung des gesamten resultierenden Modells kann Abbildung 4.6 entnommen werden.

4.3.4. Variante 4: Late-Fusion durch Fully Connected Layer im SST-Netzwerk

Bei Variante 3 erfolgt die Zusammenführung der beiden Streams am Ende der SST-Netzwerke durch einen gewichteten Durchschnitt, für den der Gewichtungsparameter nicht erlernt wird - die Zusammenführung erfolgt folglich wie bereits bei der Variante 1 „von Hand“. Entsprechend ist es wiederum eine logische mögliche Konsequenz, beide separate SST-Netzwerke durch eine gemeinsame Schicht zu kombinieren. Eine Grundlage hierfür bildet Variante 2, bei der beide separate C3D-Netzwerke nach den fc6-Schichten in einer gemeinsamen fc7-Schicht zusammengeführt wurden.

Bei Variante 4 wird dieses Prinzip auf zwei separate SST-Netzwerke übertragen, indem die Sequence Encoder beider SST-Netzwerke separat erhalten bleiben, diese aber in einer gemeinsamen Fully Connected Layer statt in zwei separaten Fully Connected Layers enden. Bei diesem Modell kommen wiederum zwei separate C3D-Netzwerke zum Einsatz, die separat auf den originalen Bilddaten und den Bildern des zugehörigen optischen Flusses trainiert werden. Anschließend werden C3D-Featurevektoren mit Hilfe der beiden C3D-

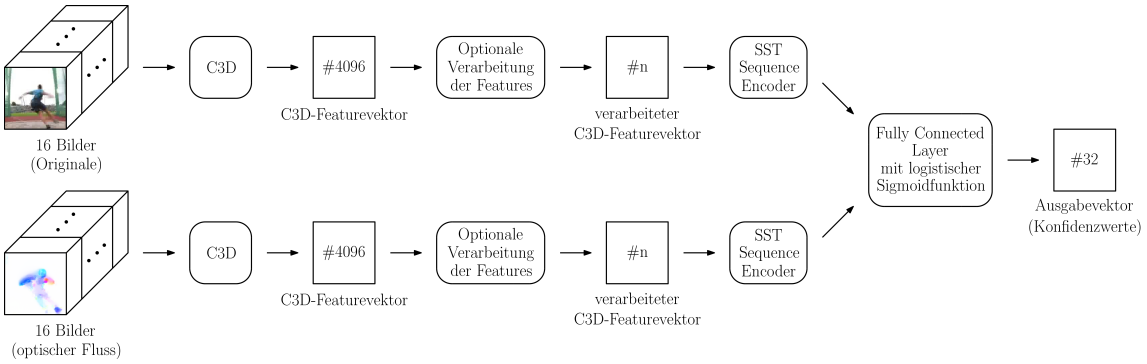


Abbildung 4.7.: Schematische Darstellung von Variante 4 der Fusion. Im Gegensatz zu Variante 3 werden die Streams dadurch vereint, dass zwei separate SST-Netzwerke nur bis zum Ende des Sequence Encoders separat bleiben und dann in einer gemeinsamen Fully Connected Layer zusammengeführt werden. Eingabebild aus THUMOS'14 [JLZ+14].

Netze extrahiert, ob von der fc6- oder fc7-Schicht kann frei gewählt werden. Vor dem Training der SST-Netzwerke werden alle dafür benötigten C3D-Featurevektoren extrahiert. Es werden zwei separate SST-Netzwerke trainiert, eines auf Basis der C3D-Featurevektoren der originalen Bilddaten und eines auf Basis der C3D-Featurevektoren, die zu den Bildern des zugehörigen optischen Flusses gehören. Diese trainierten SST-Netzwerke dienen zur Initialisierung der endgültigen Architektur. Diese endgültige Architektur sieht die Fusion der SST-Netzwerke mit Hilfe der Fully Connected Layer nach dem Sequence Encoder vor. Beide Streams laufen im Sequence Encoder separat, ihre jeweilige Ausgabe dient als Eingabe in eine gemeinsame Fully Connected Layer. Diese erzeugt mit Hilfe der logistischen Sigmoidfunktion wie beim klassischen SST-Modell die Ausgabe. Die separaten Sequence Encoder werden mit Hilfe der zuvor trainierten SST-Netzwerke initialisiert; wegen der nachfolgenden Fully Connected Layer wird das Netzwerk anschließend noch trainiert. Wie bereits bei den vorangegangenen Architekturen erfolgt wieder keine Festlegung einer eventuellen Zwischenverarbeitung der C3D-Featurevektoren, bevor sie an die SST-Netzwerke weitergereicht werden. Eine Darstellung des resultierenden Netzwerks kann Abbildung 4.7 entnommen werden.

5. Implementierungen

In diesem Kapitel werden Details der Implementierung der Modelle aus Kapitel 4 betrachtet. Dabei wird vorgestellt, welche Bibliotheken und Werkzeuge für Verarbeitungsschritte herangezogen, welche Implementierungen der SST- und C3D-Netze eingesetzt und wie auf deren Basis die Two-Stream-Modelle realisiert wurden. Konkrete technische Details können Anhang A entnommen werden.

5.1. Arbeitsumgebung

Die nachfolgenden Implementierungen wurden unter dem Betriebssystem Windows 10 64bit umgesetzt. Als Python Distribution für Python 3.6.6[1] wurde Anaconda 4.5.4[2] verwendet. Darüber hinaus wurden NVIDIA CUDA 9.0[3] mit NVIDIA cuDNN 7.1[4] und tensorflow-gpu 1.8[5] eingesetzt. Der verwendete Rechner verfügte über einen Intel Core i9-7980XE Prozessor mit 18 Kernen, 64 GB Arbeitsspeicher, zwei Nvidia GTX 1080Ti Grafikkarten und SSD Festplatten.

Eine Ausnahme stellt die Arbeitsumgebung dar, die für FlowNet2 verwendet wurde. Aus technischen Gründen wurden auf einem vergleichbaren Rechner Ubuntu 16.04, Anaconda 4.5.11 für Python 2.7.15, NVIDIA CUDA 8.0 mit NVIDIA cuDNN 5.1 und tensorflow-gpu 1.0.1 eingesetzt.

5.2. Extraktion von Bildern

Sowohl der UCF101-Datensatz als auch der THUMOS'14-Datensatz liegen in Form von Videos vor. Für die verwendete Implementierung des C3D-Netzwerks muss die Eingabe jedoch in Form von Bildern vorliegen. Diese Extraktion wurde mit Hilfe des Frameworks

[1] `https://www.python.org/`
[2] `https://www.anaconda.com/`
[3] `https://developer.nvidia.com/cuda-zone`
[4] `https://developer.nvidia.com/cudnn`
[5] `https://pypi.org/project/tensorflow-gpu/`

FFmpeg Version 4.0[6] durchgeführt; nach der offiziellen Website[7] handelt es sich um das führende Multimedia-Framework. Konkret wurde das enthaltene und gleichnamige Kommandozeilenwerkzeug ffmpeg verwendet, um aus den Videos der oben genannten Datensätze Bilder entsprechend der Bildrate der jeweiligen Videos zu extrahieren.

5.3. Bestimmung und Visualisierung des optischen Flusses

In dieser Arbeit wird der optischen Fluss primär mit dem Verfahren von Brox et al. [BBPW04] bestimmt. Zur Bestimmung des optischen Flusses aus aufeinanderfolgenden Bildern, die zuvor durch ffmpeg aus den Videos der verwendeten Datensätze extrahiert wurden, kommt die Klasse *cv::cuda::BroxOpticalFlow* der OpenCV-Bibliothek zum Einsatz. Für die anschließende Visualisierung wird die Methode verwendet, die die OpenCV-Bibliothek in der Beispieldatei zur Verwendung von *cv::cuda::BroxOpticalFlow* mitliefert. Bei OpenCV handelt sich um eine Bibliothek, die vor allem für den Bereich der Bildverarbeitung eine große Bandbreite an Algorithmen bereitstellt. Es wurde OpenCV 3.4.3[8] verwendet.

Darüber hinaus kommt in geringem Umfang optischer Fluss, der auf Basis von FlowNet2 bestimmt wurde, zum Einsatz. Um Einheitlichkeit mit den Implementierungen der nachfolgenden Netzwerke bezüglich des verwendeten Frameworks zu wahren, wird eine Implementierung[9] von FlowNet2 in TensorFlow verwendet. Diese umfasst bereits die nötige Funktionalität für die Bestimmung und die Visualisierung des optischen Flusses sowie die nötigen vortrainierten Modelle.

5.4. C3D-Netzwerk

Als Grundlage für das C3D-Netzwerk wird eine auf dem TensorFlow-Framework basierende Implementierung[10] verwendet. Die Wahl fiel hierbei aus zwei Gründen auf die Implementierung mit TensorFlow: Zum einen ist die Portierung dieser Implementierung auf das bei dieser Arbeit verwendete Betriebssystem Windows ohne großen Aufwand möglich, zum anderen wird so das Framework einheitlich zu der verwendeten Implementierung des SST-Netzwerkes gewählt, die später vorgestellt wird. Die Implementierung des C3D-Netzwerks umfasst Kernfunktionen wie das Training und das Testen, bei welchem jedem getesteten Video eine Aktionsklasse zugeordnet wird. Die anschließende Evaluation der Genauigkeit bezüglich zugeordneter Aktionsklasse und tatsächlicher Aktionsklasse sowie die Extraktion der C3D-Features wurden bei dieser Implementierung noch nicht unterstützt. Da beide Funktionalitäten jedoch in dieser Arbeit benötigt wurden, wurden sie entsprechend ergänzt. Die Implementierung auf Basis von TensorFlow stellt darüber hinaus die für TensorFlow konvertierten Gewichte der originalen Implementierung bereit, die durch Training auf Sports-1M erzeugt wurden.

[6] `https://www.ffmpeg.org/`
[7] `https://www.ffmpeg.org/about.html`
[8] `https://github.com/opencv/opencv/tree/master`
[9] `https://github.com/sampepose/flownet2-tf`
[10] `https://github.com/hx173149`

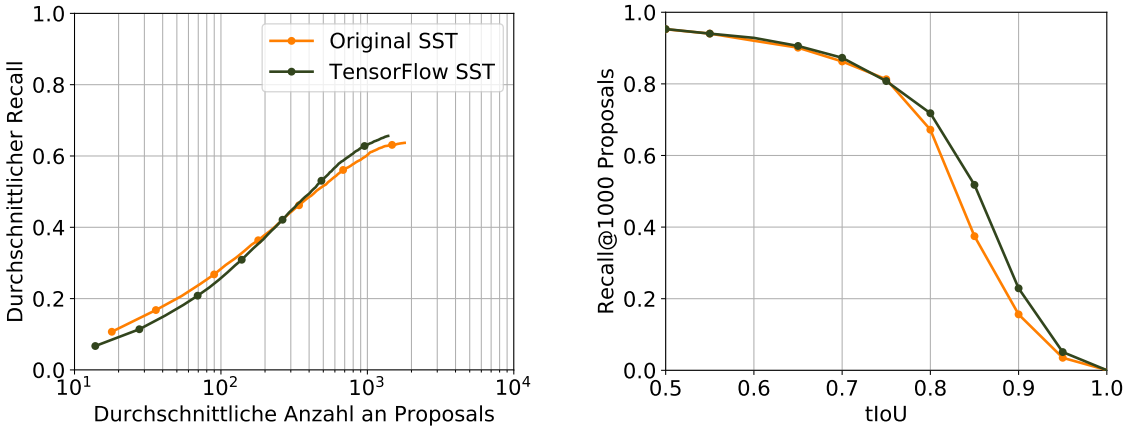


Abbildung 5.1.: Vergleich der Performance des originalen SST-Netzwerks mit dem der TensorFlow-Implementierung. Bezüglich der links dargestellten Metrik ist die originale Implementierung für eine geringere durchschnittliche Anzahl an Proposals besser als die TensorFlow-Implementierung. Abgesehen davon liefert die TensorFlow-Implementierung meistens bessere und ansonsten vergleichbare Ergebnisse.

5.5. Zwischenverarbeitungsschritte

Die meisten (optionalen) Zwischenverarbeitungsschritte, die zwischen C3D- und SST-Netzwerk ausgeführt werden, wurden bei dieser Arbeit selbst implementiert; technische Implementierungsdetails könnnen Anhang A.3 entnommen werden. Eine Ausnahme stellt die Implementierung der Hauptkomponentenanalyse zur Dimensionsreduktion der C3D-Featurevektoren dar. Um Einheitlichkeit mit Buch et al. [BES+17] zu wahren, wird auf die Hauptkomponentenanalyse der Arbeit „DAPs: Deep Action Proposals for Action Understanding“ [EHNG16] zurückgegriffen. Die Implementierung[11] dieser Arbeit ist öffentlich verfügbar; sie enthält die benötigten Skripte zur Durchführung der Hauptkomponentenanalyse.

5.6. SST-Netzwerk

Für das SST-Netzwerk wird eine Implementierung[12] in TensorFlow verwendet. Diese Implementierung übertrifft die Ergebnisse der originalen Implementierung, folglich bildet sie eine geeignete Grundlage. Ein Vergleich der Ergebnisse ist in Abbildung 5.1 zu sehen. Alle für Training und Tests benötigten Funktionalitäten werden durch diese Implementierung bereits zur Verfügung gestellt, ebenso wie Skripte zur Auswertung der erzeugten Temporal Action Proposals. Es werden C3D-Features der originalen Bilddaten des THUMOS'14-Datensatzes und Gewichte für das SST-Netzwerk zur Verfügung gestellt. Ebenso wird eine Parametrisierung für das SST-Netzwerk bereitgestellt, die laut der mitgelieferten *Readme*-Datei bei den Experimenten die besten Ergebnisse erzeugt hat. Für die Implementierung des C3D-Netzwerks in TensorFlow werden keine Gewichte bereitgestellt, die benötigt werden, um die mitgelieferten C3D-Features selbst extrahieren zu können.

[11] `https://github.com/escorciav/deep-action-proposals`
[12] `https://github.com/JaywongWang/SST-Tensorflow`

5.7. Two-Stream-Modelle

Die Implementierung der Two-Stream-Modelle aus Abschnitt 4.3 erfolgt auf Basis der in Abschnitt 5.4 und 5.6 vorgestellten Implementierungen des C3D- und SST-Netzwerks. Diese Implementierungen werden so erweitert, dass die resultierende Two-Stream-Netzwerke den Two-Stream-Modellen entsprechen. Welche Anpassungen an den Implementierungen vorgenommen werden mussten und technische Details darüber, wie die einzelnen Streams bei den verschiedenen Implementierungen zusammengeführt werden, können Anhang A.5 entnommen werden.

5.8. Evaluation

Für die Auswertung der durch das SST-Netzwerk erzeugten Temporal Action Proposals werden zwei verschiedene Implementierungen verwendet.

Während eines Trainingsvorgangs werden in festen Intervallen die erlernten Gewichte gespeichert - um die Ergebnisse des SST-Netzwerks unter Verwendung dieser Gewichte auszuwerten und die diesbezüglich besten Gewichte zu bestimmen, wird die Implementierung der Evaluation verwendet, die bei der verwendeten Implementierung des SST-Netzwerks in TensorFlow mitgeliefert wird.

Für die Erzeugung der Ergebnisse und Grafiken, die in dieser Arbeit präsentiert werden, wird hingegen die Implementierung[13] der Evaluation verwendet, die von Buch et al. [BES+17] verwendet und veröffentlicht wurden, um eine ideale Vergleichbarkeit zu erreichen.

Beide Implementierungen berechnen dieselben Evaluationsmetriken, sie unterscheiden sich primär dadurch, dass die Implementierung von Buch et al. diejenigen Zeitfenster des THUMOS'14-Datensatzes mit einbezieht, für die es bereits während des Annotierens kaum zu entscheiden war, ob es sich tatsächlich um eine Aktion handelt oder nicht – diesen Zeitfenstern ist auch keine Aktionsklasse zugewiesen. Sie werden bei Buch et al. wie Zeitfenster behandelt, denen eine Aktionsklasse zugewiesen ist, und sollen entsprechend gefunden werden. Die andere Implementierung hingegen verzichtet auf die Verwendung dieser Zeitfenster.

[13] `https://github.com/shyamal-b/sst`

6. Experimente und Ergebnisse

In diesem Kapitel werden die für Experimente verwendeten Datensätze, Parametrisierungen und Vorgehensweisen vorgestellt. Anschließend werden die einzelnen vorgenommenen Experimente erläutert und ihre Resultate werden präsentiert.

6.1. Datensätze

In diesem Abschnitt werden die beiden für Training und Tests verwendeten Datensätze vorgestellt. Es wird ein kurzer Überblick geliefert und das Einsatzgebiet der einzelnen Datensätze wird angegeben. Darüber hinaus wird ein weiterer Datensatz vorgestellt, der bei der Bestimmung der mitgelieferten Gewichte der verwendeten Implementierung des C3D-Netzwerks eingesetzt wurde.

6.1.1. UCF101

Beim UCF101-Datensatz [SZS12] handelt es sich um einen Datensatz mit realen Aktionen in realen Umgebungen, der auf Basis von YouTube-Videos erstellt wurde. Er umfasst 101 verschiedene Aktionen, für die 13320 Videoclips bereitgestellt werden - dabei entstammen jeweils einige dieser Clips demselben Video. Jedem Videoclip ist eine einzige Aktion zugeordnet. Insgesamt haben die Videoclips des Datensatzes eine Gesamtlänge von 1600 Minuten, ihre Auflösung beträgt 320×240 Pixel, die Bildrate beträgt 25 Bilder pro Sekunde. In Abbildung 6.1 sind drei Bilder aus Videos des Datensatzes zu sehen.

Bei den Experimenten wird der UCF101-Datensatz immer dann verwendet, wenn das C3D-Netzwerk trainiert und getestet werden muss – also dann, wenn kein vortrainiertes C3D-Netzwerk oder keine vorab extrahierten C3D-Features zur Verfügung stehen. Dazu erfolgt in dieser Arbeit eine rein zufällige Aufteilung der einzelnen Videoclips in 80% Trainings- und 20% Testdaten.

6.1.2. THUMOS'14

Der THUMOS'14-Datensatz [JLZ+14] besteht aus zwei Datensätzen, wovon einer für die Aufgabe der Temporal Action Localization geeignet ist – als THUMOS'14-Datensatz wird

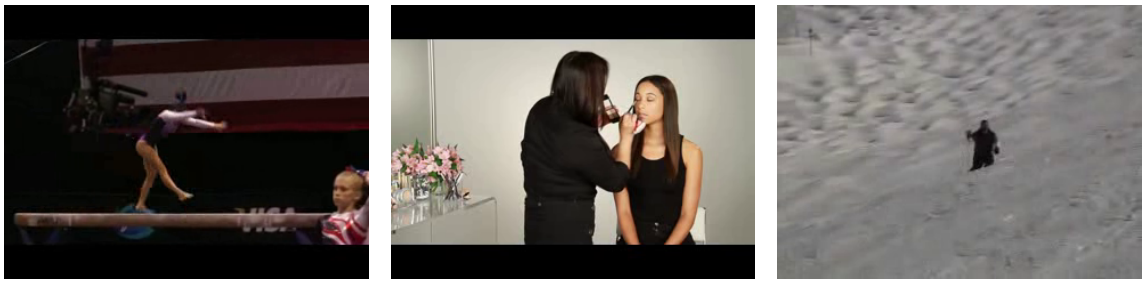


Abbildung 6.1.: Drei Bilder aus drei verschiedenen Videos des UCF101-Datensatzes [SZS12]. Dargestellte Aktionen: Turnen auf dem Schwebebalken (links), Auftragen von Make-up (mittig), Skifahren (rechts).

im Folgenden nur dieser eine Datensatz bezeichnet. Wie bei Buch et al. [BES+17] werden die Validierungsdaten zu Trainingszwecken verwendet und in Trainings- und Validierungsdaten aufgeteilt. Die Validierungsdaten umfassen 200 ungekürzte Videos; insgesamt kommen 20 verschiedene Aktionsklassen vor. Es werden Annotationen bereitgestellt, die angeben, in welcher Zeitspanne welche Aktion in welchem Video stattfindet.

Als Testdaten umfasst der Datensatz für die Temporal Action Localization 213 Videos, die dieselben 20 Klassen von Aktionen enthalten. Wie bei den Validierungsdaten werden entsprechende Annotationen bereitgestellt. Von den eigentlich 213 Videos werden 212 Videos verwendet, Video *video_test_0001292* und dessen annotierte Zeitfenster werden der Vergleichbarkeit halber außen vor gelassen, da für dieses Video in der verwendeten Implementierung von SST in TensorFlow keine C3D-Features mitgeliefert werden. Es handelt sich um ein Video, das über keine Annotation für eine der 20 Aktionsklassen verfügt. Lediglich in der *Ambiguous_test.txt*, die mit den Annotationen mitgeliefert wird, finden sich drei annotierte Zeitfenster für dieses Video – der beiliegenden *Readme.txt* kann entnommen werden, dass in dieser Datei nur Zeitfenster zu finden sind, für die schwer zu annotieren ist, ob es sich tatsächlich um eine Aktion handelt oder nicht; laut derselben Datei sind diese Zeitfenster von der Evaluation ausgenommen. Die verwendete TensorFlow-Implementierung hält sich bei der Evaluation an diese Vorgabe, das Evaluationsskript des originalen SST-Netzwerks von Buch et al. jedoch nicht. Die *Ambiguous_test.txt* enthält 99 Zeitfenster, ihre Hinzunahme erhöht die Gesamtzahl der annotierten Zeitfenster von 3358 auf 3457. Zum schnellen Vergleich diverser trainierter Modelle wird die Evaluation der TensorFlow-Implementierung herangezogen – die so identifizierten besten Modelle werden dann noch einmal durch das Evaluationsskript der originalen SST-Implementierung ausgewertet, um Vergleichbarkeit zu Buch et al. zu erreichen. Für Daten wie Auflösung und Bildrate der einzelnen Videos ist darüber hinaus eine Metadatei verfügbar, die diese Informationen enthält. In Abbildung 6.2 sind drei Bilder aus Videos von THUMOS'14 zu sehen.

Der THUMOS'14-Datensatz wird für das Training des SST-Teilnetzwerks und zum Test des gesamten Netzwerks eingesetzt. Dazu werden jeweils die C3D-Features der Videos des Datensatzes durch das C3D-Teilnetz extrahiert und anschließend im SST-Teilnetz weiterverwendet.

Abbildung 6.2.: Drei Bilder aus drei verschiedenen Videos des THUMOS'14-Datensatzes [JLZ+14]. Dargestellte Aktionen: Hochsprung (links), Dunking (mittig), Klippenspringen (rechts).

6.1.3. Sports-1M

Der Sports-1M-Datensatz [KTS+14] ist ein Datensatz, der deutlich größer als der UCF101-Datensatz ist. Die Videos des Datensatzes teilen sich in insgesamt 487 Klassen von Aktionen auf, die, wie der Name des Datensatzes bereits andeutet, alle dem Sport entstammen. Pro Klasse enthält der Datensatz 1000-3000 Videos, insgesamt sind es eine Million Videos; sie stammen alle vom Videoportal YouTube. Mit einer durchschnittlichen Länge von 336 Sekunden pro Video verfügen die Videos des Datensatzes über eine Gesamtlänge von 5,6 Millionen Minuten.

Der Sports-1M-Datensatz findet in dieser Arbeit nur indirekt Anendung, indem ein auf Sports-1M vortrainiertes C3D-Netzwerk verwendet wird und indem C3D-Featurevektoren verwendet werden, die durch ein auf Sports-1M vortrainiertes C3D-Netzwerk extrahiert wurden.

6.2. Parametrisierung und Verwendung des C3D-Netzwerks

In Anlehnung an Buch et al. [BES+17], die das C3D-Netzwerk unverändert verwenden, soll im Rahmen dieser Arbeit das C3D-Netzwerk über verschiedene Experimente hinweg mit gleichbleibender Parametrisierung verwendet werden. Diese Parametrisierung entspricht bis auf die Dropout-Rate der Parametrisierung, mit der die verwendete Implementierung bereitgestellt wird. Die Dropout-Rate wird, wie in Anhang A.2 zur Implementierung beschrieben, lediglich so angepasst, dass während Tests und während der Extraktion von Features kein Dropout stattfindet. Wichtige Parameter sowie ihre Werte können Tabelle 6.1 entnommmen werden.

Parameter	Wert
Lernverfahren	Adam
Lernrate (außer Softmax-Schicht)	1e-4
Lernrate (Softmax-Schicht)	1e-3
Dropout-Rate (nur während Training)	0.5
Mächtigkeit Minibatches	10

Tabelle 6.1.: Wichtige Parameter des verwendeten C3D-Netzwerks sowie deren Werte.

Sind C3D-Features der originalen Bilddaten nötig, wird auf die Verwendung des C3D-Netzwerks verzichtet; es werden die im Rahmen der verwendeten Implementierung des

SST-Netzwerks bereitgestellten C3D-Features verwendet. Um C3D-Features für die Bilder des optischen Flusses zu erhalten, muss das C3D-Netzwerk zuerst mit der besagten Parametrisierung neu trainiert werden – dies erfolgt auf den Bildern des optischen Flusses des UCF101-Datensatzes. Das Training ist nötig, da für die Bilder des optischen Flusses kein auf einem großen Datensatz wie beispielsweise Sports-1M vortrainiertes Modell zu finden war. Das so trainierte C3D-Netzwerk wird anschließend zur Extraktion von Features verwendet. Falls nicht explizit anders angegeben, wurde der optische Fluss für die verwendeten Bilder mit der Methode nach Brox et al. [BBPW04] bestimmt.

Für das Training des C3D-Netzwerks kommt im Rahmen von Experimenten eine rein zufällige Aufteilung der Videoclips in Trainings- und Testdaten zum Einsatz – dieselbe Aufteilung wird für alle Experimente verwendet. Da die Videoclips rein zufällig aufgeteilt wurden und jeweils mehrere Videoclips aus dem gleichen, längeren Video stammen, teilen sich die Trainings- und die Testdaten gewisse Charakteristiken, auch wenn sie nicht dieselben Abschnitte der Videos enthalten. Im Gegensatz zu dem Vorgehen, dass alle Videoclips aus demselben Video einheitlich den Trainings- oder den Testdaten zugeteilt werden, hat dieses Vorgehen den Vorteil, dass aus allen Videos Ausschnitte betrachtet werden können und das C3D-Netzwerk so mehr unterschiedliche Fälle der entsprechenden Aktionen zu sehen bekommt. Da der UCF101-Datensatz bereits deutlich weniger Videos enthält als der Sports-1M-Datensatz, wird es als wichtig erachtet, die Anzahl der Videos, aus denen Ausschnitte während des Trainings betrachtet werden, nicht noch weiter zu verringern. Dieses Vorgehen hat jedoch auch einen Nachteil: Da sich wie bereits erwähnt Videoclips aus den Trainings- und den Testdaten gewisse Charakteristiken teilen, kann nicht mehr uneingeschränkt davon ausgegangen werden, dass Overfitting durch eine Auswertung auf den Testdaten zuverlässig erkannt werden kann. Für die Extraktion von Features werden daher zwei trainierte Modelle betrachtet: Eines mit den Gewichten, für die das C3D-Netzwerk bezüglich der Testdaten die höchste Genauigkeit erzielt hat, und eines mit Gewichten zu einem früheren Zeitpunkt des Trainings, zu dem bereits eine hohe, wenn auch nicht die höchste Genauigkeit erzielt wurde. Mit den frühen Gewichten extrahierte C3D-Features werden als frühe C3D-Features bezeichnet, die anderen als späte C3D-Features. Die frühen C3D-Features werden zusätzlich extrahiert, da davon ausgegangen wird, dass diese bessere Ergebnisse als die späten C3D-Features beim Einsatz mit dem SST-Netzwerk liefern, für den Fall, dass unbemerktes Overfitting beim C3D-Netzwerk aufgetreten sein sollte.

6.3. Standardparametrisierung des SST-Netzwerks

Das verwendete SST-Netzwerk verfügt über eine Vielzahl an Parametern, die sich anpassen lassen. Einige wichtige Parameter werden im Rahmen von Experimenten untersucht, indem SST-Netzwerke mit unterschiedlichen Werten für diese Parameter trainiert und anschließend die mit ihnen erzeugten Ergebnisse verglichen werden. Um nicht für jedes durchgeführte Experiment eine vollständige Liste mit Parametern und ihren Werten angeben zu müssen, wird im Folgenden eine Standardparametrisierung des SST-Netzwerks festgelegt, die alle untersuchten Parameter sowie Standardwerte für diese Parameter enthält. Entsprechend müssen bei Experimenten nur die Abweichungen von der Standardparametrisierung angegeben werden. Als Ausgangspunkt für alle Parameter des Netzwerks dienen die Werte, die die verwendete Implementierung des SST-Netzwerks liefert – es wurde angegeben, dass diese Werte bei durchgeführten Experimenten die besten Resultate erzielten. Daher

werden sie als geeigneter Ausgangspunkt für weitere Untersuchungen angesehen. Tabelle 6.2 können alle untersuchten Parameter und deren Standardwerte entnommen werden. Zusätzlich zu den Parametern des Netzwerks wird angegeben, welche C3D-Features standardmäßig als Eingabe verwendet werden – der für sie angegebene Standard basiert nicht auf Angaben in der verwendeten Implementierung. Die Auswahl der verwendeten C3D-Features wird als zusätzlicher Parameter angesehen. Als Lernverfahren kommt immer der Adam-Algorithmus zum Einsatz.

Parameter	Wert
C3D-Features	späte C3D-Features der fc7-Schicht
Lernrate	1e-3
Neuronen pro GRU Layer	128
Anzahl GRU Layers	2
Dropout-Rate (nur während Training)	0.3

Tabelle 6.2.: Tabellarische Übersicht über die Parameter des SST-Netzwerks, die in nachfolgenden Experimenten untersucht werden, sowie die Werte, die standardmäßig für sie verwendet werden.

6.4. Evaluationsmetriken und -skripte

Wie bereits bei Buch et al. [BES+17] erfolgt die Evaluation bezüglich zweier Metriken, die im Folgenden anhand der Implementierung durch Buch et al. vorgestellt werden. Beide Metriken liefern eine Vielzahl an Werten, daher wird eine dritte Metrik eingeführt, deren Ausgabe ein einzelner Zahlenwert ist und so den Vergleich zwischen verschiedenen Modellen, Parametrisierungen und Trainingsdurchläufen stark erleichtert. Diese wird im Anschluss vorgestellt.

Recall bei durchschnittlich 1000 Proposals bezüglich der tIoU: Bei dieser Metrik werden zuerst Temporal Action Proposals von allen Videos extrahiert. Für jedes Video werden so unterschiedlich viele Temporal Action Proposals erzeugt. Um durchschnittlich 1000 Proposals pro Video zu erhalten, wird zuerst die Gesamtzahl der extrahierten Temporal Action Proposals über alle Videos bestimmt. Anschließend wird ein Prozentsatz an Temporal Action Proposals bestimmt, der pro Video extrahiert werden muss, damit durchschnittlich 1000 Proposals betrachtet werden. Dieser wird als Wert p auf dem Intervall $[0.0, 1.0]$ dargestellt und berechnet sich wie folgt:

$$p = \frac{1000 \cdot \text{Anzahl an Videos}}{\text{Gesamtzahl an Temporal Action Proposals}} \tag{6.1}$$

Für jedes Video wird anschließend eine individuelle Anzahl k_i an Temporal Action Proposals extrahiert, wobei i für das i-te Video steht:

$$k_i = p \cdot \text{Anzahl Temporal Action Proposals des Videos } i \tag{6.2}$$

Von jedem Video i werden anschließend k_i Temporal Action Proposals mit den höchsten Konfidezwerten extrahiert – auf diese Art und Weise werden durchschnittlich 1000 Temporal Action Proposals pro Video verwendet. Anschließend wird der Recall dieser Temporal

Action Proposals bezüglich der annotierten Zeitfenster der Videos für verschiedene tIoU-Schwellwerte bestimmt. Die tIoU gibt die Überlappung zweier Zeitfenster relativ zu ihrer Vereinigung an. Buch et al. [BES+17] beginnen bei einer geforderten tIoU von mindestens 0.05 zwischen Temporal Action Proposals und annotierten Zeitfenstern und berechnen den Recall für tIoU-Schritte von 0.05 bis hin zu einer tIoU von 1.0. Die Ergebnisse dieser Berechnungen werden anschließend graphisch auf dem tIoU-Intervall $[0.5, 1.0]$ dargestellt. Bei den nachfolgenden Experimenten wird analog zu diesem Vorgehen verfahren.

Durchschnittlicher Recall bezüglich der durchschnittlichen Anzahl an Proposals: Bei dieser Metrik werden zuerst verschiedene Prozentsätze p_i auf dem Intervall $[0.0, 1.0]$ festgelegt. Für jedes Video werden die p_i Prozent der Temporal Action Proposals mit den besten Konfidenzwerten extrahiert. Mit den so extrahierten Temporal Action Proposals werden Recall-Werte bezüglich unterschiedlicher tIoU-Schwellwerte bestimmt, anschließend wird ein Durchschnitt über diese Recall-Werte gebildet. Zusätzlich wird für den verwendeten Prozentsatz die Anzahl der durchschnittlichen Proposals, die mit diesem extrahiert wurden, bestimmt. Dazu wird die Zahl der Proposals, die für jedes Video extrahiert wurden, aufsummiert, und anschließend durch die Anzahl der Videos geteilt. Bei der Visualisierung der Ergebnisse wird an die Stelle der so berechneten durchschnittlichen Anzahl an Temporal Action Proposals der berechnete durchschnittliche Recall geschrieben. Buch et al. [BES+17] verwenden bei ihrer Implementierung 100 Prozentwerte aus dem Intervall $[0.01, 1.0]$ in Schritten von 0.01 – 1.0 steht 100%. Für den durchschnittlichen Recall werden elf tIoU-Schwellwerte aus dem Intervall $[0.5, 1.0]$ in Schritten von 0.05 herangezogen. Diese Vorgehensweise wird auch in dieser Arbeit verwendet.

Durchschnittlicher Recall bei durchschnittlich 1000 Proposals: Diese Metrik wird eingeführt, da bei den beiden zuvor vorgestellten Metriken jeweils eine Vielzahl an Werten berechnet wird, die für die Visualisierung und Vergleiche in Liniendiagrammen gedacht sind. Um nicht auf Basis so dargestellter Kurven, sondern auf Basis eines einzelnen Wertes entscheiden zu können, welches von zwei Modellen bessere Ergebnisse liefert, wird wie bei der ersten vorgestellten Metrik der Recall bei durchschnittlich 1000 Proposals für elf tIoU-Schwellwerte aus dem Intervall $[0.5, 1.0]$ mit Schritten von 0.05 bestimmt. Anschließend wird über diese elf Werte der Durchschnitt berechnet; dieser Wert dient zur Entscheidung zwischen verschiedenen Modellen. Er fasst die erste Metrik komplett zusammen und stellt bei der zweiten Metrik den Punkt bei durchschnittlich 1000 Proposals dar. Zur Berechnung kann auf die bereits bestimmten Werte der ersten vorgestellten Metrik zurückgegriffen werden.

6.5. Experimente zum C3D- und SST-Netzwerk auf Bildern des optischen Flusses

Für die Kombination von C3D-Netzwerk und SST-Netzwerk auf den originalen Bilddaten stehen bereits vorextrahierte C3D-Features und ein vortrainiertes SST-Netzwerk im Rahmen der verwendeten TensorFlow-Implementierung bereit, die für die originalen Bilddaten weiterverwendet werden. Für die Bilder des optischen Flusses hingegen muss zuerst ein C3D-Netzwerk trainiert werden, das anschließend zur Extraktion von C3D-Featurevektoren verwendet wird. Auf Basis dieser C3D-Features wird anschließend ein SST-Netzwerk trainiert. Für keine dieser Aufgaben steht ein vortrainiertes Modell zur Verfügung. Daher soll im Rahmen der nachfolgenden Experimente bestimmt werden, welche

C3D-Featurevektoren und welche Parametrisierung des SST-Netzwerks für die Verarbeitung der Bilder des optischen Flusses möglichst gut geeignet sind.

6.5.1. Vorgehen

Für das C3D-Netzwerk werden keine unterschiedlichen Parametrisierungen untersucht, da in der Arbeit von Buch et al. [BES+17] auch keine Optimierung des C3D-Netzwerkes unternommen wird. Das Training des C3D-Netzwerks für Bilder des optischen Flusses wird wie in Abschnitt 6.2 beschrieben vorgenommen. Das verwendete C3D-Netzwerk wird nur einmal trainiert und für alle Experimente verwendet. Für das anschließende SST-Netzwerk wird die Performance der C3D-Features der fc6-Schicht, wie sie bei der TensorFlow-Implementierung des SST-Netzwerks verwendet werden, und die der C3D-Features der fc7-Schicht, wie sie bei Buch et al. [BES+17] verwendet werden, untersucht. Außerdem werden frühe und späte C3D-Features untersucht, die in Abschnitt 6.2 definiert wurden. Es werden zwei mögliche Zwischenverarbeitungsschritte untersucht: Die L_2-Normalisierung, wie sie in der Originalarbeit zum C3D-Netzwerk [TBF+15] nach der Extraktion der Features vorgesehen ist, und eine Hauptkomponentenanalyse zur Reduktion der Größe der einzelnen C3D-Featurevektoren auf 500 Elemente, wie sie Buch et al. [BES+17] in ihrer Arbeit zum SST-Netzwerk vorsehen.

Neben diesen Untersuchungen, die bisher nur die Eingabe in das SST-Netzwerk betreffen, werden diverse Parameter des SST-Netzwerks untersucht, um so eine angemessene Parametrisierung bei Verwendung der C3D-Featurevektoren, die auf Basis der Bilder des optischen Flusses extrahiert wurden, zu bestimmen. Der Ausgangspunkt für die Untersuchung der Parametrisierung des SST-Netzwerks stellt dabei die in Abschnitt 6.3 vorgestellte Standardparametrisierung dar. Zuerst werden einzelne Parameter verändert, die dort angegeben sind: die Lernrate, die Anzahl der Neuronen pro GRU Layer, die Anzahl der GRU Layers und die Dropout-Rate. Es wird jeweils ein größerer und ein kleinerer Wert bezüglich der Standardparametrisierung verwendet. Die Untersuchungen zu den als Parameter angegebenen C3D-Features wurden bereits zuvor vorgestellt.

Für jede Parameteränderung wurden zwei Experimente mit einem eigenen Trainingsvorgang des SST-Netzwerks durchgeführt. Bei jedem Trainingsvorgang wird das beste trainierte Modell dadurch bestimmt, dass die Ergebnisse auf den Testdaten mit der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals ausgewertet und verglichen werden. Der so berechnete Wert wird auch zum Vergleich mit anderen Experimenten herangezogen; es werden die Parameteränderungen bestimmt, die bezüglich dieser Metrik bessere Ergebnisse liefern als die Standardparametrisierung. Auf Basis der Ergebnisse dieser Experimente werden anschließend einige Kombinationen der vorgenommenen Änderungen an der Standardparametrisierung untersucht.

6.5.2. Resultate

Bevor die Resultate der einzelnen Experimente vorgestellt werden, wird in Tabelle 6.3 eine Nummerierung für die verschiedenen Experimentgruppen eingeführt; dort wird auch der genaue Wert des von der Standardparametrisierung abweichenden Parameters angegeben. Die Experimentgruppe zur Standardparametrisierung wird nicht mit einer Nummer referenziert, sondern mit dem Begriff *Standard*. Taucht im Folgenden in Tabellen die Abkürzung *Exp.* auf, steht diese für Experimentgruppe.

Exp.	Änderung zur Standardparametrisierung
#1	C3D-Features der fc6-Schicht
#2	frühe C3D-Features
#3	500-elementige C3D-Features durch Hauptkomponentenanalyse
#4	L_2-normalisierte C3D-Features
#5	Lernrate 1e-2
#6	Lernrate 1e-4
#7	Dropout-Rate 0.1
#8	Dropout-Rate 0.5
#9	64 Neuronen pro GRU Layer
#10	256 Neuronen pro GRU Layer
#11	Anzahl GRU Layers: 1
#12	Anzahl GRU Layers: 3

Tabelle 6.3.: Auflistung der Nummern für die Experimentgruppen mit zugehöriger Parameteränderung.

Wie bereits erwähnt wurden für jede Parameteränderung zwei Experimente durchgeführt. Die Ergebnisse auf den Testdaten werden anhand der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals verglichen; die Ergebnisse können Tabelle 6.4 entnommen werden. Verschiedene Experimente für dieselbe Experimentgruppe unterscheiden sich dabei dadurch, dass für sie ein separates Training des SST-Netzwerks durchgeführt wurde – diese Experimente werden durch die Angabe einer Trainingsnummer unterschieden.

Exp. \ Training	1	2
Standard	0.6039	0.5976
#1	0.5993	0.6028
#2	0.6073	0.6063
#3	0.5814	0.5894
#4	0.6103	0.6052
#5	0.6165	0.6217
#6	0.6006	0.5971
#7	0.6042	0.6114
#8	0.6018	0.6070
#9	0.6015	0.6042
#10	0.6083	0.6108
#11	0.6148	0.6027
#12	0.5981	0.6020

Tabelle 6.4.: Ergebnisse der Experimente auf den Testdaten bei Variation einzelner Parameter bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals. Der geänderte Parameter sowie der verwendete Wert können mit der angegeben Nummer für die Experimentgruppe Tabelle 6.3 entnommen werden; Standard steht für die Standardparametrisierung. Die Nummer bezüglich des Trainings wird verwendet, um einzelne Experimente mit derselben Parameteränderung, aber separatem Training des SST-Netzwerks zu unterscheiden.

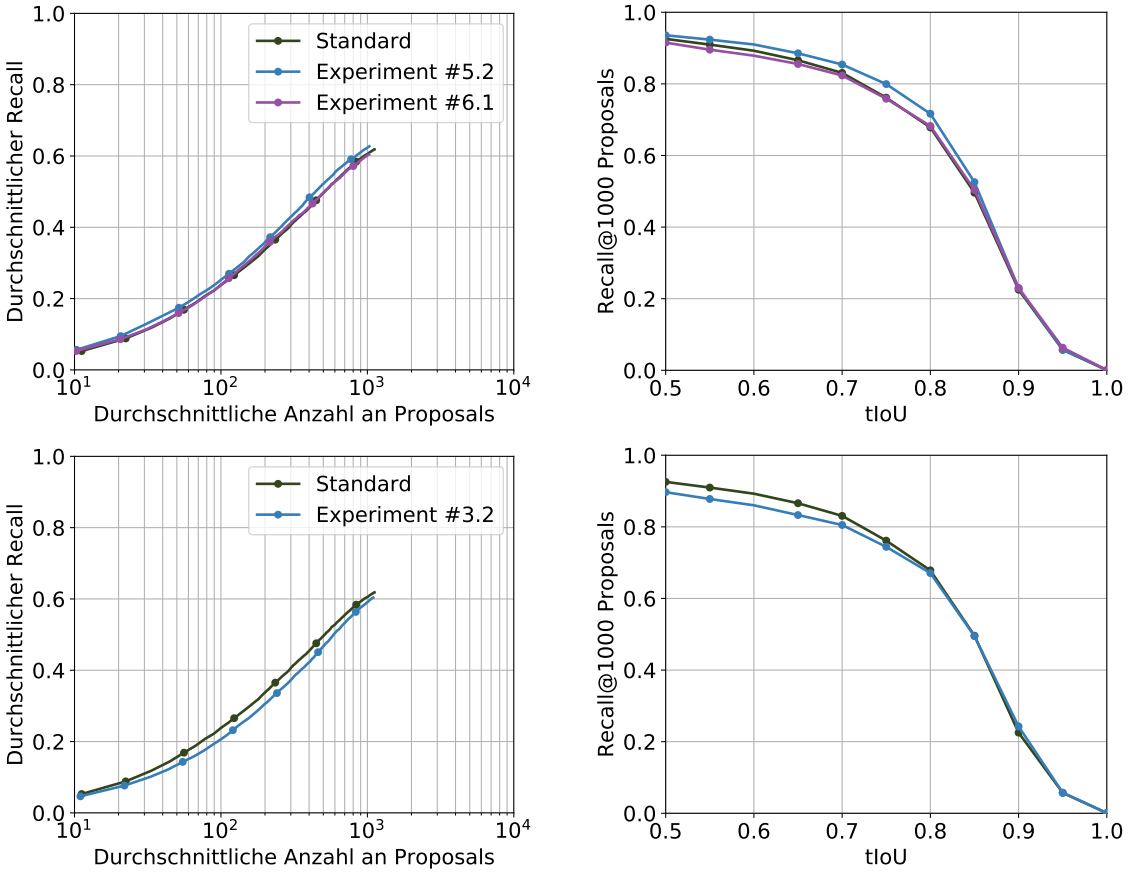


Abbildung 6.3.: Ergebnisse der Experimente bezüglich der Variation einzelner Parameter; für jede Variation eines Parameters wird das beste Ergebnis der beiden durchgeführten Experimente dargestellt. Die Experimente werden mit *Experimentgruppe.Training* referenziert; Standard steht für die Ergebnisse des besten Experiments mit der Standardparametrisierung. Oben sind die Ergebnisse der Untersuchungen zur Lernrate zu sehen und unten die Ergebnisse der Untersuchung der Hauptkomponentenanalyse, bei der die C3D-Featurevektoren auf jeweils 500 Elemente reduziert werden.

Der Tabelle kann entnommen werden, dass vor allem die Änderung der Lernrate von 1e-3 auf 1e-2 und die Reduktion von zwei auf eine GRU Layer zu einer Verbesserung gegenüber der Standardparametrisierung geführt haben. Auch die L_2-Normalisierung der C3D-Featurevektoren und die Erhöhung der Anzahl an Neuronen pro GRU Layer von 128 auf 256 Stück haben sich als nützlich erwiesen. Da sowohl die Erhöhung als auch die Verringerung der Dropout-Rate zu einer kleinen Verbesserung geführt haben, wird bei diesem Parameter bei dem Wert der Standardparametrisierung verblieben – die Abweichung ist nur gering und könnte eventuell vom Trainingsvorgang herrühren, vor allem, da eine Änderung in beide Richtungen zu einer kleinen Verbesserung geführt hat. Eine weitere kleine Verbesserung konnte durch die Verwendung der frühen C3D-Features erreicht werden. Andere Änderungen erzielten kaum eine Verbesserung oder sorgten sogar für eine Verschlechterung. Daher werden in Abbildung 6.3 ausgewählte Ergebnisse dargestellt, die deutlich besser oder deutlich schlechter als die bei der Standarparametrisierung

sind. Es werden die beiden Evaluationsmetriken durchschnittlicher Recall bezüglich der durchschnittlichen Anzahl an Proposals und Recall bei durchschnittlich 1000 Proposals bezüglich der tIoU dargestellt.

Auf Basis der Ergebnisse zur Variation von einzelnen Parametern bieten sich Untersuchungen von Kombinationen von Parametern an, deren Änderung zu einer Verbesserung geführt hat. Für diese Tests wird im Folgenden immer die Lernrate 1e-2 verwendet, ebenso werden immer die frühen C3D-Features eingesetzt. Es werden zusätzlich Kombinationen mit nur einer GRU Layer, normalisierten frühen C3D-Features und 256 oder 512 Neuronen pro GRU Layer durchgeführt; verschiedene Kombinationen dieser Parametrisierungen werden untersucht. 512 Neuronen pro GRU Layer werden in die Untersuchungen mit aufgenommen, da beispielsweise eine GRU Layer mit 512 Neuronen eine logische Alternative zu zwei GRU Layers mit 256 Neuronen darstellt. Die Nummern der Experimentgruppen und die genaue Kombination der Parameteränderungen kann Tabelle 6.5 entnommen werden.

Exp.	Änderungen zur Standardparametrisierung
#13	Lernrate: 1e-2 frühe C3D-Features
#14	Lernrate: 1e-2 frühe und L_2-normalisierte C3D-Features
#15	Lernrate: 1e-2 frühe C3D-Features Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 256
#16	Lernrate: 1e-2 frühe und L_2-normalisierte C3D-Features Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 256
#17	Lernrate: 1e-2 frühe und L_2-normalisierte C3D-Features Neuronen pro GRU Layer: 256
#18	Lernrate: 1e-2 frühe C3D-Features Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 512
#19	Lernrate: 1e-2 frühe und normalisierte C3D-Features Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 512
#20	Lernrate: 1e-2 frühe und normalisierte C3D-Features Neuronen pro GRU Layer: 512

Tabelle 6.5.: Untersuchte Kombinationen von Parameteränderungen auf Basis von einzelnen Parameteränderungen, die zu einer Verbesserung bei der Temporal Action Proposal Generierung bezüglich des durchschnittlichen Recalls bei durchschnittlich 1000 Proposals geführt haben.

Pro Parametrisierung wurden wie bereits zuvor zwei Experimente durchgeführt. Bei den Experimenten der Experimentgruppen #13-#17 zeigte sich, dass die besten Ergebnisse fast immer am Ende des Trainingsvorgangs erreicht wurden, daher wurde die maximale Anzahl an Trainingsiterationen erhöht und diese Experimente wurden wiederholt; die Experimente der Experimentgruppen #18-#20 wurden direkt mit der erhöhten Anzahl an maximalen Trainingsiterationen durchgeführt. Die Ergebnisse der Experimente, die mit zu wenig Trainingsiterationen durchgeführt wurden, werden nicht verworfen, sondern ebenfalls präsentiert; entsprechend existieren für diese vier statt zwei Ergebnisse. Die Ergebnisse bezüglich der Testdaten können Tabelle 6.6 entnommen werden.

Exp. \ Training	1	2	3	4
13	0.6169	0.6125	0.6099	0.6160
14	0.6100	0.6224	0.6196	0.6226
15	0.6204	0.6224	0.6225	0.6193
16	0.6320	0.6135	0.6239	0.6237
17	0.6215	0.6099	0.6288	0.6266
18	0.6273	0.6079	-	-
19	0.5726	0.5464	-	-
20	0.5132	0.5324	-	-

Tabelle 6.6.: Ergebnisse der einzelnen Experimente zu den kombinierten Parameteränderungen; es wird der durchschnittliche Recall bei durchschnittlich 1000 Proposals angegeben. Die jeweiligen von der Standardparametrisierung abweichenden Parameter sowie ihre Werte sind in Tabelle 6.5 unter der Nummer für die jeweilige Experimentgruppe zu finden. Das beste Ergebnis konnten bei dem Experiment #16.1 erzielt werden; dieses erzielte im Vergleich zum besten Experiment mit der Standardparametrisierung eine Verbesserung von 0.0281.

Das beste Ergebnis wird bei Experiment #16.1 erzielt, in Abbildung 6.4 findet sich daher eine Visualisierung des Ergebnisses im Vergleich zum Ergebnis bei Verwendung der Standardparametrisierung. Wie zu sehen ist, gelang eine deutliche Verbesserung.

Weitere Verbesserungen werden bei den Experimenten der Experimentgruppen #17 und #18 erzielt; Experimente der Experimentgruppe #15 erzielten geringere Verbesserungen der Ergebnisse, dafür aber sehr konstante über alle Experimente. Die bei diesen Experimenten verwendeten Parametrisierungen und teilweise auch die während dieser Experimente bestimmten Gewichte für das SST-Netzwerk werden zusammen mit denen der Experimente zu Experimentgruppe #16 für weitere Untersuchungen bei den Two-Stream-Modellen verwendet. Werden die Gewichte zur Initialisierung benötigt, werden jeweils die des besten Experiments jeder Experimentgruppe weiterverwendet.

Unerwartet sind die deutlich schlechteren Ergebnisse der Experimente zu Experimentgruppe #19 im Vergleich zu #18, die sich bezüglich der Parametrisierung lediglich darin unterscheiden, dass bei #19 die C3D-Features zusätzlich einer L_2-Normalisierung unterzogen werden, bevor sie als Eingabe in das SST-Netzwerk dienen. Bei der Variation der einzelnen Parameter hatte die L_2-Normalisierung zu einer Verbesserung und nicht zu einer deutlichen Verschlechterung geführt. Eine mögliche Erklärung wäre, dass das Training

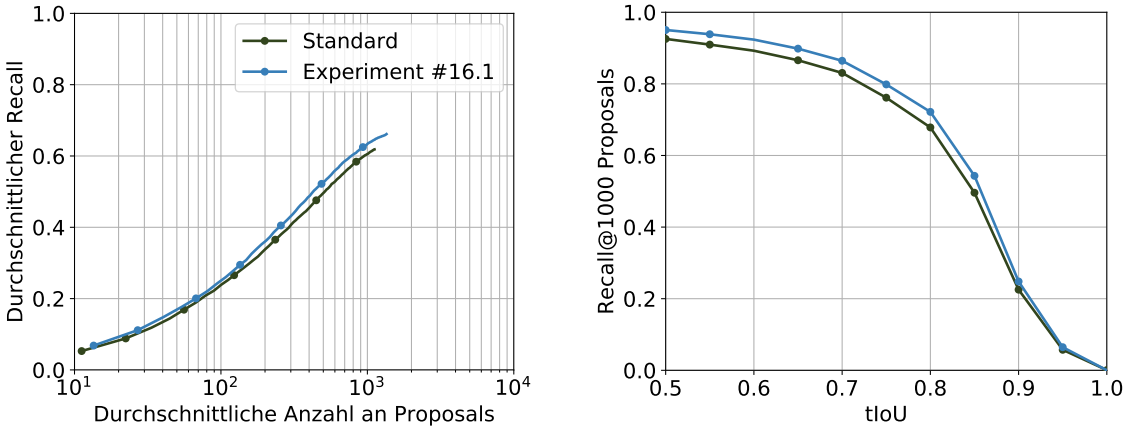


Abbildung 6.4.: Darstellung des besten Ergebnisses der Experimente zu den kombinierten Parametrisierungen im Vergleich zur Standardparametrisierung. Das bestes Ergebnis wurde bei Experiment #16.1 erzielt. Wie zu sehen ist, konnte bezüglich beider dargestellten Metriken eine deutliche Verbesserung erzielt werden.

mit zu wenigen Iterationen angesetzt war. Bei den Experimenten zu Experimentgruppe #19 wurden beides Mal die besten Ergebnisse nach 90% der Trainingsiterationen erzielt; die zweitbesten Ergebnisse wurden jedoch bereits nach 25% der Trainingsiterationen bei Experiment #19.1 und 65% der Trainingsiterationen bei Experiment #19.2 erzielt, mit jeweils minimal schlechteren Werten – etwa 0.0009 Differenz bei Experiment #19.1 und 0.0002 Differenz bei Experiment #19.2. Daher wird angenommen, dass weitere Trainingsiterationen nicht den gewünschten Effekt erbringen.

6.6. Experimente zur Untersuchung von Variante 1: Mid-Fusion durch Kombination der C3D-Features

In diesem Abschnitt wird das Modell der Variante 1 der Fusion aus Abschnitt 4.3.1 experimentell untersucht. Verschiedene Parametrisierungen werden untersucht, die Performance des Modells wird mit der der originalen Implementierung des SST-Netzwerks nach Buch et al. [BES+17] und mit der der TensorFlow-Implementierung verglichen.

6.6.1. Vorgehen

Für die Auswertung dieses Modells muss kein neues C3D-Netzwerk trainiert werden. Für die C3D-Features, die auf Basis der originalen Bilddaten extrahiert werden, werden die in der TensorFlow-Implementierung des SST-Netzwerks mitgelieferten C3D-Features verwendet. Für die C3D-Features, die auf Basis des optischen Flusses extrahiert werden, werden die bereits extrahierten C3D-Features der vorangegangenen Experimente aus Abschnitt 6.5 verwendet. Konkret werden für die Bilder des optischen Flusses die frühen C3D-Features der fc7-Schicht aus diesen Experimenten eingesetzt. Für die Experimente werden die beiden Sets von C3D-Features konkateniert.

Auch die Durchführung und Auswertung der Experimente wird analog zu den vorangegangenen Experimenten durchgeführt. Für das SST-Netzwerk, das sich bei dieser Variante

nach der Konkatenation der C3D-Features anschließt, werden die bei den vorangegangenen Experimenten erfolgreich eingesetzten Parameterkombinationen verwendet. Die Standardparametrisierung wird ebenfalls untersucht, da die mitgelieferten C3D-Features in den konkatenierten Vektoren enthalten sind und diese Parametrisierung in der verwendeten TensorFlow-Implementierung des SST-Netzwerks für diese Features vorgesehen war. Da das SST-Netzwerk im Vergleich zu vorangegangenen Experimenten neue Eingaben verwendet, wird es auf Basis der konkatenierten C3D-Featurevektoren bei den verschiedenen Experimenten neu trainiert.

6.6.2. Resultate

Als erster Schritt werden die von der Standardparametrisierung des SST-Netzwerks abweichenden Parameter sowie ihre Werte vorgestellt. Diese sind in Tabelle 6.7 aufgelistet; den einzelnen Kombinationen von Parameteränderungen werden dort wie zuvor Nummern für Experimentgruppen zugeordnet.

Exp.	Änderungen zur Standardparametrisierung
#21	konkatenierte C3D-Features
#22	Lernrate: 1e-2 L_2-normalisierte konkatenierte C3D-Features Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 256
#23	Lernrate: 1e-2 konkatenierte C3D-Features Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 256
#24	Lernrate: 1e-2 L_2-normalisierte konkatenierte C3D-Features Neuronen pro GRU Layer: 256
#25	Lernrate: 1e-2 konkatenierte C3D-Features Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 512

Tabelle 6.7.: Untersuchte Parameteränderungen des SST-Netzwerks für Variante 1 der Two-Stream-Modelle, basierend auf den Ergebnissen der Experimente aus Abschnitt 6.5. Jeder Kombination von Parameteränderungen wird eine Nummer zugeordnet, die eine Experimentgruppe referenziert.

Pro Experimentgruppe werden wieder zwei Experimente mit separaten Trainings durchgeführt. Die Ergebnisse der Experimente auf den Testdaten bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals werden in Tabelle 6.8 aufgelistet.

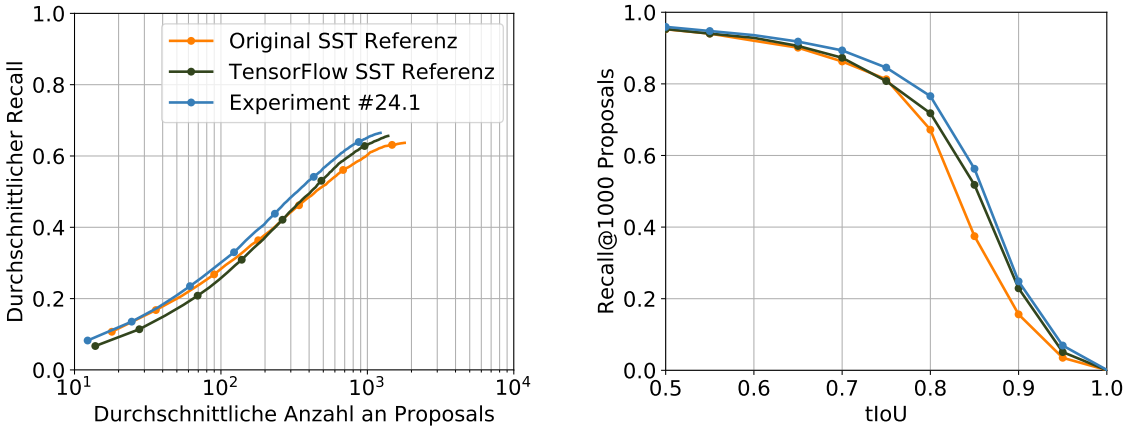


Abbildung 6.5.: Vergleich des besten Ergebnisses der Experimente zu Variante 1 der Two-Stream-Modelle aus Experiment #24.1 mit den Ergebnissen von Buch et al. [BES+17] auf dem originalen SST-Netzwerk und den Ergebnissen der TensorFlow-Implementierung des SST-Netzwerks mit den mitgelieferten Gewichten und C3D-Features. Die Auswertung erfolgte auf den Testdaten des THUMOS'14-Datensatzes.

Exp. \ Training	1	2
#21	0.6294	0.6274
#22	0.6441	0.6383
#23	0.6289	0.6264
#24	0.6497	0.6328
#25	0.6337	0.6237

Tabelle 6.8.: Ergebnisse der verschiedenen Experimente zu Variante 1 der Two-Stream-Modelle auf den Testdaten.

Dieser Tabelle kann entnommen werden, dass das beste Ergebnis unter den durchgeführten Experimenten bei Experiment #24.1 erzielt wurde. Dieses Ergebnis wird nun verwendet, um die Variante 1 der Two-Stream-Modelle mit den Ergebnissen des originalen SST-Netzwerks und der TensorFlow-Implementierung des SST-Netzwerks zu vergleichen. Der Vergleich ist in Abbildung 6.5 dargestellt.

Wie in der Abbildung zu sehen ist, erzielt die Variante 1 der Two-Stream-Modelle für beide dargestellten Metriken in weiten Bereichen eine deutliche Verbesserung. Lediglich bei einer geringen durchschnittlichen Anzahl an Proposals entspricht der durchschnittliche Recall in etwa dem des originalen SST-Netzwerks, dasselbe gilt für den Recall bei durchschnittlich 1000 Proposals wenn niedrige tIoU-Werte betrachtet werden.

6.7. Experimente zur Untersuchung von Variante 2: Mid-Fusion in der fc7-Schicht des C3D-Netzwerks

In diesem Abschnitt wird das Modell der Variante 2 der Fusion aus Abschnitt 4.3.2 experimentell untersucht. Wie bei den vorangegangenen Experimenten zu Variante 1 werden

verschiedene Parametrisierungen untersucht, die Performance des Modells wird mit der der originalen Implementierung des SST-Netzwerks nach Buchet al. [BES+17] und mit der der TensorFlow-Implementierung verglichen.

6.7.1. Vorgehen

Bei dem für diese Experimente verwendeten Two-Stream-Modell laufen die beiden separaten Streams zweier C3D-Netzwerke nach separaten fc6-Schichten in einer gemeinsamen fc7-Schicht zusammen. Solange die Streams in den C3D-Netzwerken separat sind, können sie mit vorab bestimmten Gewichten initialisiert werden, die gemeinsame fc7-Schicht und die sich anschließende Softmax Layer müssen jedoch von Grund auf trainiert werden. Zur Initialisierung des Streams für die originalen Bilddaten werden die auf Sports-1M bestimmten Gewichte eingesetzt, die bei der verwendeten Implementierung des C3D-Netzwerks in TensorFlow mitgeliefert wurden. Der Stream für die Bilder des optischen Flusses wird mit den Gewichten initialisiert, die bei den Experimenten in Abschnitt 6.5 zur Extraktion der frühen C3D-Features herangezogen wurden. Bei einem anschließenden Training werden nur die Gewichte der fc7-Schicht und der Softmax Layer trainiert, während dieses Trainingsvorgangs wird das beste trainierte Modell bezüglich der Genauigkeit auf den Testdaten des UCF101-Datensatzes bestimmt. Dieser wird mit derselben Aufteilung wie bei den anderen Experimenten verwendet. Anschließend findet analog ein Finetuning aller Schichten gemeinsam statt, hierfür werden vergleichsweise wenig Trainingsiterationen durchgeführt. Bei den Experimenten in diesem Abschnitt existieren durch das Vorgehen bedingt keine frühen oder späten C3D-Features; es wird lediglich zwischen C3D-Features von der fc7-Schicht der fusionierten C3D-Netzwerke mit und ohne Finetuning unterschieden. Diese werden im Folgenden entsprechend als C3D-Features mit bzw. ohne Finetuning bezeichnet.

Die Performance der C3D-Features der trainierten C3D-Netzwerke mit und ohne Finetuning wird bei Einsatz mit dem SST-Netzwerk ermittelt. Dieses wird für beide Sets an C3D-Features trainiert und ausgewertet. Die C3D-Features, die sich als geeigneter herausstellen, werden für weitere Experimente verwendet. Bei diesen Experimenten werden SST-Netzwerke mit in Abschnitt 6.5 erfolgreich angewandten Kombinationen von Parametern untersucht; das Vorgehen bei der Untersuchung dieser so parametrisierten SST-Netzwerke ist analog zu den vorangegangenen Experimenten.

6.7.2. Resultate

Zuerst wurde ein Vergleich der Performance der C3D-Features mit und ohne Finetuning durchgeführt, indem vergleichende Experimente auf dem SST-Netzwerk unter Verwendung der Standardparametrisierung und der jeweiligen zu untersuchenden C3D-Features durchgeführt wurden. Die Zuordnung von Experimentgruppen zu verwendeter Parametrisierung kann Tabelle 6.9 entnommen werden.

Exp	Parameter
#26	Standard, C3D-Features mit Finetuning
#27	Standard, C3D-Features ohne Finetuning

Tabelle 6.9.: Experimentgruppen und zugehörige Parametrisierungen bei den Untersuchungen zu C3D-Features mit und ohne Finetuning.

Wie bereits aus vorangegangenen Experimenten bekannt, wurden pro Experimentgruppe zwei Experimente mit separatem Training durchgeführt; die Auswertung auf den Testdaten bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals kann Tabelle 6.10 entnommen werden.

Exp. \ Training	1	2
#26	0.6201	0.6202
#27	0.6176	0.6216

Tabelle 6.10.: Ergebnisse der vergleichenden Auswertung von C3D-Features mit und ohne Finetuning bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals.

Die Ergebnisse aller Experimente ähneln sich sehr stark, das beste Ergebnis wurde bei Experiment #27.2 unter Verwendung der C3D-Features ohne Finetuning erzielt. Daher werden für alle nachfolgenden Experimente die C3D-Features ohne Finetuning weiterverwendet. Die weiteren durchgeführten Experimente, deren Parameteränderungen auf den Erkenntnissen der Experimente aus Abschnitt 6.5 beruhen, können zusammen mit den zugewiesenen Nummern für die Experimentgruppen Tabelle 6.11 entnommen werden.

Exp.	Änderungen zur Standardparametrisierung
#28	Lernrate: 1e-2 L_2-normalisierte C3D-Features ohne Finetuning Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 256
#29	Lernrate: 1e-2 C3D-Features ohne Finetuning Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 256
#30	Lernrate: 1e-2 L_2-normalisierte C3D-Features ohne Finetuning Neuronen pro GRU Layer: 256
#31	Lernrate: 1e-2 C3D-Features ohne Finetuning Anzahl GRU Layers: 1 Neuronen pro GRU Layer: 512

Tabelle 6.11.: Experimentgruppen sowie zugewiesene Parameteränderungen für die Experimente zur Untersuchung der Variante 2 der Two-Stream-Modelle.

Die Experimente der Experimentgruppen werden nach dem bekannten Schema durchgeführt, die Ergebnisse der Experimente finden sich in Tabelle 6.12.

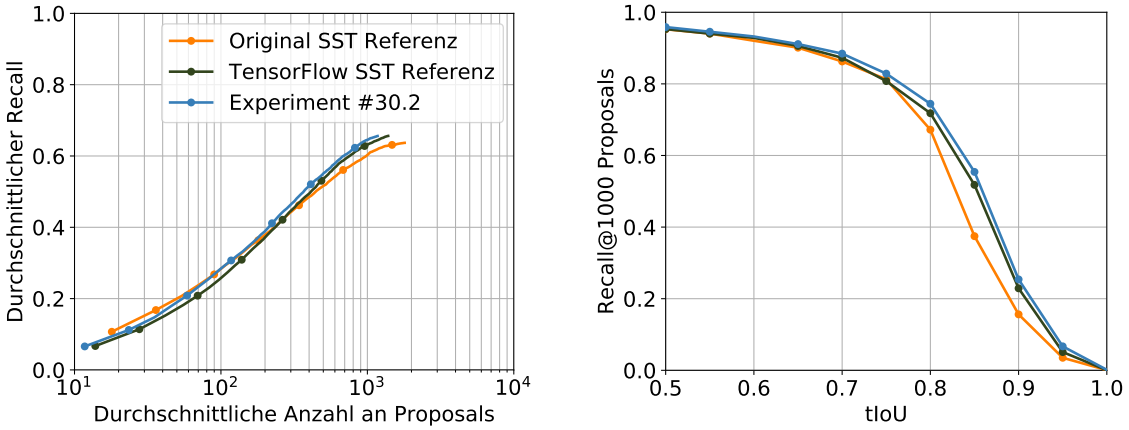


Abbildung 6.6.: Ergebnisse des besten Experiments #30.2 mit Variante 2 der Two-Stream-Modelle im Vergleich zu den Ergebnissen, die mit dem originalen SST-Netzwerk und der TensorFlow-Implementierung des SST-Netzwerks erzielt wurden. In weiten Bereichen wurde eine Verbesserung zu beiden Netzwerke, mit denen verglichen wird, erzielt.

Exp. \ Training	1	2
#28	0.6291	0.6322
#29	0.6158	0.6202
#30	0.6290	0.6438
#31	0.6287	0.6166

Tabelle 6.12.: Ergebnisse der Experimente zu Tabelle 6.11 auf den Testdaten bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals.

Das beste Ergebnis wurde bei Experiment #30.2 mit einem Wert von 0.6438 erzielt und bleibt damit um 0.0059 hinter dem besten Ergebnis der Experimente zu Variante 1 zurück. Ein Vergleich mit den Ergebnissen des originalen SST-Netzwerks und der TensorFlow-Implementierung des SST-Netzwerks ist in Abbildung 6.6 zu sehen. Bei einer geringen durchschnittlichen Anzahl an Temporal Action Proposals ist der durchschnittliche Recall etwas schlechter als beim originalen SST-Netzwerk, bei einer geringen tIoU werden vergleichbare Ergebnisse zu den anderen Netzwerken erzielt. Ansonsten erzielt auch Variante 2 bezüglich der betrachteten Metriken bessere Ergebnisse als die beiden Netzwerke, mit denen verglichen wird.

6.8. Experimente zur Untersuchung von Variante 3: Late-Fusion durch Bilden eines gewichteten Durchschnitts im SST-Netzwerk

In diesem Abschnitt wird das Modell der Variante 3 der Fusion aus Abschnitt 4.3.3 experimentell untersucht. Wie bei den vorangegangenen Experimenten zu Variante 1 und 2 werden verschiedene Parametrisierungen untersucht, die Performance des Modells wird

mit der der originalen Implementierung des SST-Netzwerks nach Buchet al. [BES+17] und mit der der TensorFlow-Implementierung verglichen.

6.8.1. Vorgehen

Bei Variante 3 der Two-Stream-Modelle laufen zwei Streams, die jeweils aus einem C3D-Netzwerk und einem SST-Netzwerk bestehen, bis zur Ausgabe der Konfidenzwerte separat; auf den Konfidenzwerten wird abschließend ein gewichteter Durchschnitt gebildet. Für die nachfolgenden Experimente ist daher kein neues Training nötig. Für den Stream für die originalen Bilddaten werden die bei der TensorFlow-Implementierung des SST-Netzwerks mitgelieferten C3D-Features als Eingabe verwendet und das SST-Netzwerk wird mit den mitgelieferten Gewichten initialisiert. Bei diesem Stream werden keine unterschiedlichen Parametrisierungen untersucht. Für den Stream für die Bilder des optischen Flusses werden die frühen C3D-Features, die bei den Experimenten in Abschnitt 6.5 extrahiert wurden, verwendet. Für diesen Stream wird zum einen die Standardparametrisierung unter Verwendung der frühen C3D-Features untersucht, um so über zwei symmetrische Streams zu verfügen, zum anderen werden erfolgreiche Kombinationen von Parameteränderungen aus den Experimenten in Abschnitt 6.5 untersucht. Zur Initialisierung dienen die Gewichte, die bei dem Experiment verwendet wurden, das zu der entsprechenden Kombination von Parameteränderungen die besten Ergebnisse geliefert hat.

Zusätzlich soll bei den Experimenten die Gewichtung der Ergebnisse des Streams für die Bilder des optischen Flusses untersucht werden. Zu jeder Kombination von Parameteränderungen werden drei Experimente durchgeführt. Es findet kein Training statt, die Experimente unterscheiden sich in diesem Abschnitt dadurch, dass die Ergebnisse des Streams für die Bilder des optischen Flusses einmal mit 1/3, einmal mit 1/2 und einmal mit 2/3 gewichtet werden. Das Gewicht der Ergebnisse des Streams auf den originalen Bilddaten wird so gewählt, dass sich als Summe der Gewichte 1 ergibt.

Es wird sowohl die beste Parametrisierung als auch die beste Gewichtung bestimmt. Für diese Kombination wird anschließend ein weiteres Experiment durchgeführt, bei dem ein Training zum Finetuning auf Basis der Ausgabe der gewichteten Summe der Konfidenzwerte erfolgt. Es wird überprüft, ob durch dieses gemeinsame Finetuning beider Streams eine weitere Verbesserung erreicht werden kann. Da keine zufällige Initialisierung stattfindet, wird nur ein Experiment durchgeführt und keine zwei Experimente mit separatem Training.

6.8.2. Resultate

Wie bereits im vorangegangenen Abschnitt beschrieben, bleiben die verwendeten Gewichte, Parameter und C3D-Features für den Stream, der die originalen Bilddaten verarbeitet, unverändert – sie werden alle im Rahmen der TensorFlow-Implementierung bereitgestellt. Variationen finden folglich nur auf dem Stream für die Bilder des optischen Flusses statt. Dieser wird entsprechend erfolgreicher Experimente aus Abschnitt 6.5 parametrisiert und mit Gewichten und C3D-Features versorgt. Es wird ebenfalls auf ein Experiment zurückgegriffen, dessen Verwendung dafür sorgt, dass die SST-Netzwerke beider Streams symmetrisch sind. Eine Zuweisung von Experimentgruppen zu dem jeweiligen Vorgehen bezüglich dieses Streams kann Tabelle 6.13 entnommen werden.

Exp.	Parameter und Initialisierung des Streams für die Bilder des optischen Flusses
#32	Gewichte und Parameter wie in Experiment #13.1
#33	Gewichte und Parameter wie in Experiment #16.1
#34	Gewichte und Parameter wie in Experiment #15.3
#35	Gewichte und Parameter wie in Experiment #17.3
#36	Gewichte und Parameter wie in Experiment #18.1

Tabelle 6.13.: Zuordnung von Experimentgruppen zu dem Vorgehen auf dem Stream für die Bilder des optischen Flusses. Es wird angegeben, auf Basis welcher Experimente das SST-Netzwerk dieses Streams parametrisiert und initialisiert wird. Die Verwendung des Streams für die originalen Bilddaten erfolgt für alle Experimente analog auf Basis der bei der verwendeten TensorFlow-Implementierung bereitgestellten Parameter, Gewichte und C3D-Features, daher wird dieser hier nicht explizit mit angegeben.

Auf Basis der Parametrisierungen in dieser Tabelle werden pro Experimentgruppe drei Experimente durchgeführt, bei denen die Ergebnisse des Streams für die Bilder des optischen Flusses und die des Streams für die originalen Bilder unterschiedlich gewichtet werden; es findet kein Training statt. Es wird jeweils das Gewicht des Streams für die Bilder des optischen Flusses angegeben, dieses summiert sich mit dem Gewicht für den anderen Stream zu 1 auf. Die Ergebnisse bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals lassen sich Tabelle 6.14 entnehmen.

Exp. \ Gewichtung	Gewicht 1: 1/3	Gewicht 2: 1/2	Gewicht 3: 2/3
#32	0.6427	0.6376	0.6335
#33	0.6450	0.6495	0.6444
#34	0.6436	0.6407	0.6364
#35	0.6444	0.6451	0.6418
#36	0.6422	0.6433	0.6411

Tabelle 6.14.: Ergebnisse der Experimente zu Variante 3 der Two-Stream-Modelle für verschiedene Parametrisierungen bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals. Der Begriff *Gewicht* in der Tabelle steht für die verwendete Gewichtung des Streams für die Bilder des optischen Flusses. Experimente aus dieser Tabelle werden mit dem Format *Experimentgruppe.Gewichtungsnummer* referenziert.

Das beste Ergebnis in dieser Tabelle liefert Experiment #33.2; auf Basis dieses Experiments wird ein gemeinsames Finetuning beider Streams durchgeführt. Das initialisierte Netzwerk wird einem Training mit Lernrate 1e-4 unterzogen und es wird untersucht, ob das gemeinsame Finetuning beider Streams auf Basis der gewichteten Summe der Konfidenzwerte zu einer weiteren Verbesserung führt. Das Ergebnis findet sich in Tabelle 6.15.

Exp.	Vorgehen	Ergebnis
#37	Finetuning auf Basis von Experiment #33.2 mit Lernrate 1e-4	0.6475

Tabelle 6.15.: Darstellung der Ergebnisse des Finetunings. Das Ergebnis wird bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals dargestellt.

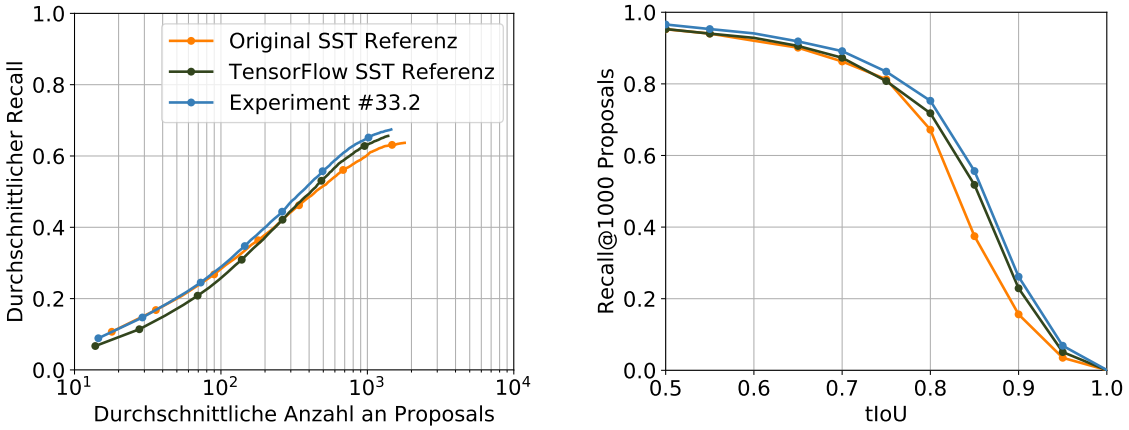


Abbildung 6.7.: Ergebnisse des besten Experiments #33.2 mit Variante 3 der Two-Stream-Modelle im Vergleich zum originalen SST-Netzwerk und zur TensorFlow-Implementierung des SST-Netzwerks. Abgesehen von dem erreichten durchschnittlichen Recall bei einer geringen durchschnittlichen Anzahl an Proposals, der vergleichbar zu dem des originalen SST-Netzwerks ist, wurde durchgehend eine Verbesserung zu beiden Netzwerken erzielt.

Wie zu sehen ist sorgt das Finetuning für eine leichte Verschlechterung des Ergebnisses. Für die Variante 3 der Two-Stream-Modelle verbleibt folglich Experiment #33.2 als bestes durchgeführtes Experiment. Die Ergebnisse dieses Experiments werden wie bei den anderen Varianten zuvor mit denen des originalen SST-Netzwerks und denen der TensorFlow-Implementierung des SST-Netzwerks verglichen; Abbildung 6.7 können die Ergebnisse des Vergleichs entnommen werden. Bei geringer durchschnittlicher Anzahl an Proposals erzielt die betrachtete Variante einen zum originalen SST-Netzwerk vergleichbaren durchschnittlichen Recall; abgesehen davon erzielt Variante 3 eine Verbesserung zu den anderen betrachteten Netzwerke.

6.9. Experimente zur Untersuchung von Variante 4: Late-Fusion durch Fully Connected Layer im SST-Netzwerk

In diesem Abschnitt wird das Modell der Variante 4 der Fusion aus Abschnitt 4.3.3 experimentell untersucht. Wie bei den vorangegangenen Experimenten zu Variante 1, 2 und 3 werden verschiedene Parametrisierungen untersucht, die Performance des Modells wird mit der der originalen Implementierung des SST-Netzwerks nach Buchet al. [BES+17] und mit der der TensorFlow-Implementierung verglichen.

6.9.1. Vorgehen

Bei Variante 4 der Two-Stream-Modelle bleiben beide Streams bis einschließlich zum Sequence Encoder der verwendeten SST-Netzwerke separat; anschließend werden sie in einer gemeinsamen Fully Connected Layer fusioniert, welche auch die Konfidenzwerte ausgibt. Dieses Modell weist starke Ähnlichkeit mit dem der Variante 3 auf, folglich wird in weiten Teilen bei den Experimenten zu Variante 4 wie bei den Experimenten zu Variante 3 in Abschnitt 6.8 verfahren. Das Vorgehen unterscheidet sich primär dadurch, dass die Parametrisierung und Initialisierungen mit Gewichten nur auf die separaten Sequence Encoder

angewendet werden; die nachfolgende Fully Connected Layer muss neu trainiert werden. Daher werden auch wieder pro Parametrisierung zwei Experimente mit separaten Trainings durchgeführt. Wiederum wird nur die Parametrisierung und Initialisierung des Streams für die Bilder des optischen Flusses variiert, die Parametrisierung und Initialisierung des Streams für die originalen Bilddaten erfolgt wie bei den Experimenten in Abschnitt 6.8. Die Streams verwenden wieder die entsprechenden, bereits extrahierten C3D-Features.

Bei den Trainingsvorgängen wird nur die gemeinsame Fully Connected Layer trainiert, es wird auf Basis der bereits extrahierten C3D-Features trainiert. Die Gewichte, die für die anderen Schichten zur Initialisierung verwendet werden, werden während des Trainings nicht angepasst. In den Experimenten wird so das Experiment mit dem besten Ergebnis für Variante 4 bestimmt; abschließend findet auf Basis des Modells, das in diesem Experiment trainiert wurde, noch ein Finetuning mit geringer Lernrate statt, bei dem alle Gewichte des SST-Netzwerks gemeinsam angepasst werden. Es wird untersucht, ob dieses Finetuning noch zu einer weiteren Verbesserung führt.

6.9.2. Resultate

Wie gehabt werden zuerst die verwendeten Parametrisierungen und Initialisierungen mit vorab bestimmten Gewichten sowie die zur Identifikation erforderlichen Nummern der Experimentgruppen angegeben; diese finden sich in Tabelle 6.16.

Exp.	Parameter und Initialisierung
#38	Sequence Encoder des Bild-Streams: Wie bei TensorFlow-Implementierung Sequence Encoder des Optischen-Fluss-Streams: Wie bei Experiment #13.1 Lernrate: 1e-3
#39	Sequence Encoder des Bild-Streams: Wie bei TensorFlow-Implementierung Sequence Encoder des Optischen-Fluss-Streams: Wie bei Experiment #16.1 Lernrate: 1e-3
#40	Sequence Encoder des Bild-Streams: Wie bei TensorFlow-Implementierung Sequence Encoder des Optischen-Fluss-Streams: Wie bei Experiment #15.3 Lernrate: 1e-3
#41	Sequence Encoder des Bild-Streams: Wie bei TensorFlow-Implementierung Sequence Encoder des Optischen-Fluss-Streams: Wie bei Experiment #17.3 Lernrate: 1e-3
#42	Sequence Encoder des Bild-Streams: Wie bei TensorFlow-Implementierung Sequence Encoder des Optischen-Fluss-Streams: Wie bei Experiment #18.1 Lernrate: 1e-3

Tabelle 6.16.: Zuordnung von Experimentgruppen zu der verwendeten Parametrisierung und Initialisierung. Wird beispielsweise ein Experiment für die Parameter und Initialisierung angegeben, werden die dort verwendeten Parameter und die im Rahmen des Experiments bestimmten Gewichte verwendet.

Dort bezeichnet der Bild-Stream den Stream für die originalen Bilddaten und der Optische-Fluss-Stream den Stream für die Bilder des optischen Flusses. Die SST-Netzwerke, die durch die Fully Connected Layer fusioniert werden, arbeiten wieder auf den entsprechenden

bereits extrahierten C3D-Features. In allen Fällen wurde die Lernrate 1e-3 zur Anpassung der Gewichte der Fully Connected Layer verwendet, da die Verwendung der Lernrate 1e-2 bei den Experimenten dazu geführt hat, dass die Anzahl der generierten Temporal Action Proposals bereits nach wenigen Iterationen so stark abnahm, dass nicht mehr genügend Proposals vorhanden waren, um durchschnittlich 1000 Proposals extrahieren zu können.

Pro Experimentgruppe werden zwei Experimente mit separatem Training durchgeführt. Wie bei den anderen Experimenten wird das Ergebnis auf den Testdaten bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals angegeben, das diesbezüglich beste Experiment wird bestimmt. Die Ergebnisse sind Tabelle 6.17 zu entnehmen.

Exp. \ Training	1	2
#38	0.6359	0.6341
#39	0.6466	0.6437
#40	0.6374	0.6383
#41	0.6386	0.6409
#42	0.6441	0.6401

Tabelle 6.17.: Ergebnisse der Experimente zu Variante 4 der Two-Stream-Modelle bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals.

Die besten Ergebnisse werden bei Experiment #39.1 erzielt. Im Folgenden dient das im Rahmen dieses Experiments trainierte Netzwerk als Ausgangspunkt für ein Finetuning. Dabei werden alle Gewichte der fusionierten SST-Netzwerke zusammen mit einer Lernrate von 1e-4 trainiert; es wird untersucht, ob eine Verbesserung erzielt werden kann. Die als Eingabe verwendeten C3D-Features bleiben gleich. Das Ergebnis des Finetunings ist Tabelle 6.18 zu entnehmen; beim Finetuning trat wie bereits bei Variante 3 eine leichte Verschlechterung ein.

Exp.	Vorgehen	Ergebnis
#43	Finetuning auf Basis von Experiment #39.1 mit Lernrate 1e-4	0.6460

Tabelle 6.18.: Ergebnisse des Finetunings auf dem trainierten Modell des Experiments #39.1 zu Variante 4 der Two-Stream-Modelle. Als Metrik wurde durchschnittlicher Recall bei durchschnittlich 1000 Proposals verwendet.

Damit verbleibt Experiment #39.1 als Experiment mit dem besten Ergebnis. Es erfolgt wiederum ein Vergleich mit den Ergebnissen des originalen SST-Netzwerks und denen der Implementierung des SST-Netzwerks in TensorFlow. Bis auf die Ergebnisse für den durchschnittlichen Recall bei geringer durchschnittlicher Anzahl an Proposals und die Ergebnisse für den Recall bei durchschnittlich 1000 Proposals bei geringen tIoU, bei denen vergleichbare Ergebnisse zum originalen SST-Netzwerk erzielt wurden, sorgte auch dieses Modell in weiten Bereichen für eine Verbesserung. Die Ergebnisse sind in Abbildung 6.8 zu sehen.

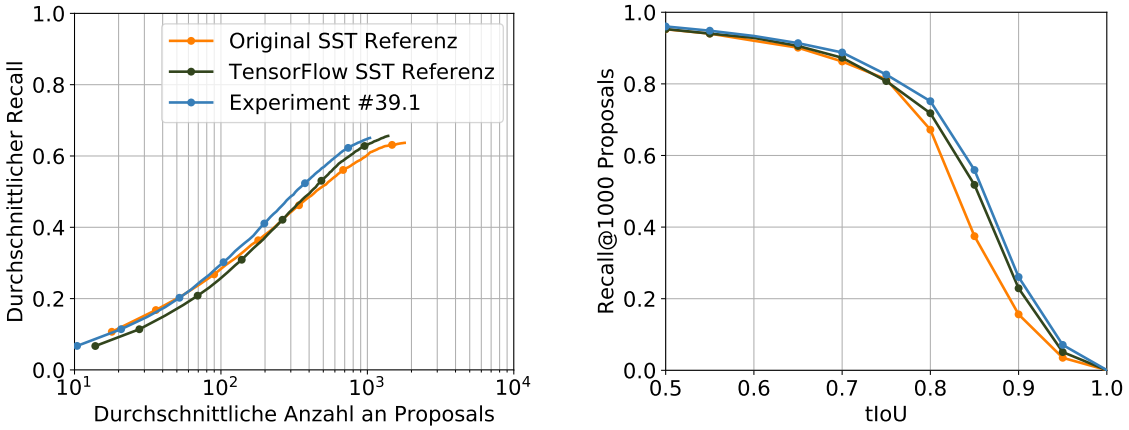


Abbildung 6.8.: Ergebnisse des besten Experiments #39.1 mit Variante 4 der Two-Stream-Modelle im Vergleich zum originalen SST-Netzwerk und zur TensorFlow-Implementierung des SST-Netzwerks. In weiten Bereichen sorgt Variante 4 für eine Verbesserung gegenüber den Netzwerken, mit denen verglichen wird; an den anderen Stellen werden vergleichbare Ergebnisse erzielt.

6.10. Experimente zu mit FlowNet2 extrahiertem optischen Fluss

In den bisherigen Experimenten wurde nur auf Bildern des optischen Flusses gearbeitet, der mit Hilfe der Methode nach Brox et al. [BBPW04] bestimmt wurde. Im Folgenden soll ein kurzer Vergleich mit einer anderen Methode zur Extraktion gegeben werden: Bei den sich anschließenden Experimenten wird der optische Fluss mit FlowNet2 [IMS+17] bestimmt, der anschließend wieder durch Bilder visualisiert wird. Ein Vergleich wird auf Variante 3 der Fusion geführt, da sie zusammen mit Variante 1 die besten Ergebnisse erzielt hat. Variante 3 wird an Stelle von Variante 1 bevorzugt, da bei dieser Variante viele Ergebnisse der unterschiedlichen Experimente nahe am besten Ergebnis waren. Auf Basis der Bilder des optischen Flusses, der mit FlowNet2 berechnet wurde, wird ein C3D-Netzwerk trainiert; frühe C3D-Features für den THUMOS'14-Datensatz werden extrahiert. Die so extrahierten C3D-Features werden im Folgenden für Training und Test des SST-Netzwerks eingesetzt, wo ansonsten C3D-Features auf Basis der Bilder des optischen Flusses nach Brox eingesetzt worden wären. Bei Variante 3 wurden die besten Ergebnisse bei Experiment #33.2 erzielt. Dieses nutzte das vortrainierte SST-Netzwerk von Experiment #16.1 zur Initialisierung. Daher werden zwei Experimente mit separatem Training entsprechend der zu Experimentgruppe #16 gehörenden Parametrisierung durchgeführt. Die Ergebnisse können Tabelle 6.19 entnommen werden; den Experimenten wurde die Experimentgruppe #44 zugewiesen.

Exp. \ Training	1	2
#44	0.6200	0.6140

Tabelle 6.19.: Ergebnisse der Experimente der Experimentgruppe #44 auf den Testdaten; die verwendete Metrik ist durchschnittlicher Recall bei durchschnittlich 1000 Proposals.

Das beste Ergebnis wurde bei Experiment #44.1 erzielt. Für weitere Experimente wird eine neue Experimentgrupppe #45 eingeführt, deren Parametrisierung der von Experimentgruppe #33 entspricht, abgesehen davon, dass die in Experiment #44.1 bestimmten Gewichte und die C3D-Features auf Basis von FlowNet2 verwendet werden. Wie in Abschnitt 6.8, welcher die Experimente zu Variante 3 enthält, werden zu der Experimentgruppe drei Experimente bezüglich verschiedener Gewichtungen des Streams für die Bilder des optischen Flusses durchgeführt. Die Ergebnisse der Experimente können Tabelle 6.20 entnommen werden.

Exp. \ Gewichtung	Gewicht 1: 1/3	Gewicht 2: 1/2	Gewicht 3: 2/3
#45	0.6429	0.6436	0.6395

Tabelle 6.20.: Ergebnisse der Experimente der Experimentgruppe #45 bezüglich unterschiedlicher Gewichtungen des Streams für die Bilder des optischen Flusses. Die Ergebnisse wurden auf den Testdaten bestimmt, es wurde die Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals verwendet.

Wie zu sehen ist, konnte bei keinem der Experimente das Ergebnis von 0.6495 des Experiments #33.2 übertroffen werden. Das beste Ergebnis wurde mit einem Wert von 0.6436 bei Experiment #45.2 erzielt; es handelt sich lediglich um eine leichte Verschlechterung. Durch den Einsatz von FlowNet2 zur Bestimmung des optischen Flusses konnte folglich für dieses Vorgehen und diese Variante der Fusion keine Verbesserung gegenüber dem Einsatz des Verfahrens von Brox et al. zur Bestimmung des optischen Flusses erzielt werden.

6.11. Fazit

Bei den durchgeführten Experimenten gelang es, mit allen vier Varianten der Two-Stream-Modelle im Regelfall Verbesserungen gegenüber dem originalen SST-Netzwerk und der TensorFlow-Implementierung des SST-Netzwerks zu erzielen. Starke Verbesserungen wurden beispielsweise für den Recall bei durchschnittlich 1000 Proposals für tIoU-Werte zwischen 0.7 und 0.9 erzielt. Eine Ausnahme stellen die Werte für den durchschnittlichen Recall bei einer geringen durchschnittlichen Anzahl an Proposals dar: Hier wurden lediglich vergleichbare Ergebnisse zum originalen SST-Netzwerk erzielt; im Fall von Variante 2 wurden – wie in Abbildung 6.6 zu sehen – sogar etwas schlechtere Ergebnisse erzielt. Ebenso wurden für den Recall bei durchschnittlich 1000 Proposals für geringe tIoU-Werte nur vergleichbare Ergebnisse zum originalen SST-Netzwerk und zur TensorFlow-Implementierung des SST-Netzwerks erzielt. Unter den vier Varianten der Two-Stream-Netzwerke erzielten Variante 1 und 3 die besten Ergebnisse; die Darstellungen der Ergebnisse sind in Abbildung 6.5 und 6.7 zu sehen. Ein abschließender tabellarischer Vergleich der besten Ergebnisse aller Netzwerke bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals kann Tabelle 6.21 entnommen werden.

Netzwerk	Experiment	Ergebnis
Originales SST-Netzwerk	-	0.6025
TensorFlow-Implementierung des SST-Netzwerks	-	0.6295
SST-Netzwerk (Bilder des optischen Flusses)	#16.1	0.6320
Variante 1	#24.1	0.6497
Variante 2	#30.2	0.6438
Variante 3	#33.2	0.6495
Variante 4	#39.1	0.6466
Variante 3 (FlowNet2 statt Brox)	#45.1	0.6429

Tabelle 6.21.: Vergleich der besten Ergebnisse der untersuchten Netzwerke mit den Ergebnissen des originalen SST-Netzwerks von Buch et al. [BES+17] und denen der verwendeten TensorFlow-Implementierung des SST-Netzwerks. Der Vergleich wird bezüglich der Metrik durchschnittlicher Recall bei durchschnittlich 1000 Proposals geführt. Alle Varianten der Two-Stream-Netzwerke erzielen eine Verbesserung gegenüber dem originalen SST-Netzwerk, der TensorfFlow-Implementierung des SST-Netzwerks und dem SST-Netzwerk aus Experiment #16.1, für dessen Arbeit die Bilder des optischen Flusses die Grundlage bilden.

7. Diskussion und Ausblick

In dieser Arbeit wird untersucht, ob Methoden zur Generierung von Temporal Action Proposals, die 3D-Faltungen auf der Bildsequenz des Videos einsetzen, dadurch verbessert werden können, dass zu diesem einzelnen Stream ein zweiter Stream hinzugefügt wird, auf dem Bilder, die den optischen Fluss visualisieren, ebenfalls mit 3D-Faltungen verarbeitet werden. Darüber hinaus wird untersucht, wie beide Streams gewinnbringend fusioniert werden können, sodass eine Verbesserung bei der Generierung von Temporal Action Proposals erreicht wird.

Als Grundlage für die Arbeit wurde das SST-Netzwerk herangezogen, welches das C3D-Netzwerk verwendet, um aus Videos – also Bildsequenzen – mit Hilfe von 3D-Faltungen sogenannte C3D-Featurevektoren zu extrahieren. Diese dienen im Folgenden als Grundlage für ein Recurrent Neural Network, mit dessen Hilfe Temporal Action Proposals erzeugt werden.

Dieses Netzwerk wurde zu einem Two-Stream-Netzwerk erweitert, bei dem ein Stream auf den originalen Bilddaten und ein Stream auf Bildern, die den optischen Fluss visualisieren, arbeitet. Dabei wurden verschiedene Varianten der Fusion untersucht. Eine Fusion nach den C3D-Teilnetzen durch Konkatenation der Ausgaben und eine Fusion in der letzten Fully Connected Layer der C3D-Netzwerke wurden untersucht, ebenso eine Fusion in der letzten Fully Connected Layer der SST-Netzwerke sowie eine Fusion durch Bilden des gewichteten Durchschnitts der Ausgabe beider SST-Netzwerke. Weitere Untersuchungen wurden bezüglich verschiedener Parametrisierungen der Modelle durchgeführt.

Es konnte experimentell auf dem THUMOS'14-Datensatz gezeigt werden, dass durch Hinzunahme eines Streams, auf dem die Bildsequenzen von Bildern des optischen Flusses mit Hilfe von 3D-Faltungen im C3D-Netzwerk verarbeitet werden, eine Verbesserung gegenüber SST-Netzwerken mit nur einem Stream auf den originalen Bilddaten erzielt werden konnte – dies gilt für alle untersuchten Varianten der Two-Stream-Modelle. Ebenso wurde experimentell unter diversen Parametrisierungen die geeignetste für jedes Modell bestimmt. Für das primär verwendete Verfahren von Brox et al. zur Bestimmung des optischen Flusses wurde FlowNet2 als Alternative untersucht, eine Verbesserung konnte damit nicht erzielt werden.

7.1. Diskussion

Beim Vergleich der vier Varianten von Two-Stream-Modellen zeigte sich, dass Variante 1 und 3 etwas bessere Ergebnisse erzielen als Variante 2 und 4. Diese Modelle haben gemeinsam, dass bei Variante 1 und 3 die Fusion nicht innerhalb eines der beiden Teilnetze, sondern zwischen beiden Teilnetzen beziehungsweise nach dem letzten Teilnetz passiert. Variante 2 und Variante 4 setzen hingegen jeweils eine gemeinsame Fully Connected Layer zur Fusion der separaten Streams ein. Hieraus wird die Schlussfolgerung gezogen, dass die Fusion innerhalb des Neuronalen Netzes weniger zur Fusion geeignet ist – zumindest im Rahmen der hier untersuchten Netzwerke und Schichten der Fusion.

Es stellt sich darüber hinaus auch die Frage, ab wann ein Einsatz der entwickelten Two-Stream-Modelle gegenüber dem originalen SST-Modell sinnvoll ist. Ist bei den Temporal Action Proposals eine möglichst hohe tIoU erwünscht, wird der Einsatz als sinnvoll erachtet, da hier für höhere tIoU-Werte deutlich höhere Werte für den Recall bei durchschnittlich 1000 Proposals pro Video erreicht werden konnten als beim originalen SST-Netzwerk und der TensorFlow-Implementierung des SST-Netzwerks. Soll hingegen nur eine geringe durchschnittliche Anzahl an Proposals extrahiert werden, wird der Einsatz eines der Two-Stream-Modelle nicht als sinnvoll erachtet: Hier konnten bezüglich des durchschnittlichen Recalls nur vergleichbare Ergebnisse mit dem originalen SST-Netzwerk erzielt werden. Da dieses nur über einen Stream verfügt, kann mit geringerem Aufwand ein vergleichbares Ergebnis erzielt werden. Erst bei einer höheren durchschnittlichen Anzahl an Proposals liefern die Two-Stream-Modelle eindeutig bessere Ergebnisse.

Auch den bereits im vorangegangenen Abschnitt kurz angesprochenen Aufwand gilt es zu beachten: Durch die Verwendung eines zweiten Streams bei den Two-Stream-Modellen ist sowohl der Speicher- als auch Berechnungsaufwand erhöht und mit letzterem auch die Zeit, die zur Erzeugung von Temporal Action Proposals benötigt wird. Zusätzlich zur Berechnung auf dem Two-Stream-Netzwerk benötigt auch die Berechnung des optischen Flusses Zeit und Speicher. Ist die Zeit ein entscheidender Faktor, kann über die Verwendung der in dieser Arbeit verwendeten TensorFlow-Implementierung des SST-Netzwerks nachgedacht werden. Gegenüber dem originalen SST-Netzwerk werden dort bereits deutliche Verbesserungen bei hohen tIoU-Werten erzielt, wenn auch nicht so große Verbesserungen wie bei den Two-Stream-Modellen. Für zeitkritische Einsatzgebiete stellt es eine geeignete Alternative dar. Ist das Einsatzgebiet nicht so zeitkritisch, dass ein schnelleres Netzwerk mit nur einem Stream eingesetzt werden muss, oder steht genügend Rechenleistung und eine effiziente Implementierung zur Bestimmung des optischen Flusses bereit, ist das Two-Stream-Netzwerk mit seinen besseren Ergebnissen das Mittel der Wahl.

7.2. Ausblick

Eine weitere Verbesserung könnte möglicherweise dadurch erzielt werden, dass die im Two-Stream-Modell vorhandenen C3D-Teilnetzwerke zusammen mit dem Rest des Netzwerks trainiert werden. Bisher wurden die C3D-Netzwerke bezüglich der Aktionserkennung auf dem UCF101-Datensatz trainiert und später ohne weiteres Training zu Extraktion von C3D-Features verwendet. Von nachfolgendem Finetuning zusammen mit dem Rest des Netzwerks bezüglich der Generierung von Temporal Action Proposals auf dem

THUMOS'14-Datensatz wird erhofft, dass C3D-Features, die besser an die zu lösende Aufgabe angepasst sind, erzeugt werden. Motiviert wird dieses mögliche Vorgehen durch die Arbeit von Sevilla-Lara et al. [SLLG+17], die bei ihrer Arbeit zur Aktionserkennung eine Verbesserung erzielen konnten, indem das dort verwendete Netzwerk zur Extraktion des optischen Flusses zusätzlich einem Finetuning auf Basis des Klassifikationsfehler des gesamten Netzwerks unterzogen wird. In dieser Arbeit wurde das vorgeschlagene Vorgehen nicht untersucht, da für jede Trainingsiterationen C3D-Featurevektoren neu extrahiert werden müssen, was einen erheblichen Zeitaufwand bedeutet.

Darüber hinaus könnten auch Alternativen zum C3D-Netzwerk untersucht werden, um zusammenhängende Bilder in Vektoren zusammenzufassen, die dann an Stelle der C3D-Featurevektoren verwendet werden können. In dieser Arbeit wurde eine solche Untersuchung nicht durchgeführt, um die Vergleichbarkeit zu Buch et al. [BES+17] zu erhalten.

Des Weiteren wurde mit dem hier vorgestellten Vorgehen zur Generierung von Temporal Action Proposals ein Verfahren geschaffen, das robuste Zwischenergebnisse für die weiteren Schritte bei der Temporal Action Localization liefert. Somit stellt es einen soliden Ausgangspunkt für weitere Untersuchungen auf diesem Gebiet dar.

Anhang

A. Technische Implementierungsdetails

Dieser Anhang umfasst technische Details von Implementierungen, die in Kapitel 5 vorgestellt wurden. Es wird auf Abläufe sowie auf vorgenommene Änderungen, Erweiterungen und Implementierungen eingegangen; auch relevante Details existierender Implementierungen werden betrachtet.

A.1. Bestimmung und Visualisierung des optischen Flusses

Wie bereits in Abschnitt 5.3 erwähnt, erfolgt die Berechnung des optischen Flusses nach Brox et al. mit Hilfe der Klasse *cv::cuda::BroxOpticalFlow* der OpenCV-Bibiliothek. Eine beispielhafte Verwendung dieser Klasse findet sich in einer der von OpenCV mitgelieferten Beispieldatei: *optical_flow.cpp*. Die dort eingesetzte Parametrisierung für die Klasse *cv::cuda::BroxOpticalFlow* wird für die Berechnung des optischen Flusses übernommen. Mit der so parametrisierten Klasse wurde anschließend der optische Fluss auf Bildpaaren berechnet. Im Rahmen dieser Arbeit soll jedem originalen Bild einer Bildfolge ein weiteres Bild zugeordnet werden, das den optischen Fluss visualisiert. Da der optische Fluss aber für Bildpaare bestimmt wird, können aus n originalen, aufeinanderfolgenden Bildern eigentlich nur $n-1$ Bilder des optischen Flusses erzeugt werden. Dieses Problem wurde dadurch gelöst, dass das erste Bild des optischen Flusses zum ersten originalen Bild der Eingabesequenz als optischer Fluss zu demselben originalen Bild bestimmt wurde. Für alle anderen originalen Bilder der Bildsequenz wurde der optische Fluss wie folgt bestimmt: Sei das originale Bild an Position i. Dann wird das zugehörige Bild des optischen Flusses als Visualisierung des optischen Flusses von Bild $i-1$ zu Bild i bestimmt. Somit zeigt das Bild des optischen Flusses an Stelle i die Visualisierung der Verschiebung, die in den originalen Bildern von Bild $i-1$ zu Bild i stattfindet. Theoretisch wäre es auch möglich gewesen, dass für das letzte Bild der Eingabesequenz der Fluss zu sich selbst berechnet wird, um für alle Bilder einen zugehörigen optischen Fluss zu erhalten. Bei dieser Variante würden jedoch zur Berechung des optischen Flusses für Bild i Informationen aus der Zukunft nötig, nämlich Bild $i+1$. Daher wurde diese Variante nicht gewählt. Darüber hinaus enthält die

Datei *optical_flow.cpp* Code zur Visualisierung des berechneten optischen Flusses - dieser wird verwendet, um das das Bild für den jeweiligen optischen Fluss zu bestimmen.

Bei der Bestimmung des optischen Flusses mit FlowNet2 wurde die verwendete Implementierung nur geringfügig angepasst, indem unter Einsatz der Klasse *tf.placeholder* die Verarbeitung von Minibatches ermöglicht wurde. Dadurch kann der optische Fluss pro Iteration für mehrere Bildpaare bestimmt werden. Die Organisation der Bildpaare, für die der optische Fluss bestimmt wird, erfolgt so, wie sie zuvor für die Bestimmung des optischen Flusses nach Brox et al. beschrieben wurde.

A.2. C3D-Netzwerk

Die verwendete Implementierung des C3D-Netzwerks in TensorFlow ermöglicht bereits Training und Tests, es fehlt ein Skript zur Bestimmung der Genauigkeit für den UCF101-Datensatz auf Basis der Testausgabe. Ebenfalls fehlt ein Skript zur Extraktion der C3D-Features; diese Funktionalität ist für das in Abschnitt 4.1 vorgestellte Modell für das klassische C3D-Netzwerk nötig.

Die Implementierung des Trainings und der Tests kann beinahe unverändert übernommen werden. Diverse Parameter wie die Anzahl der Trainingsdurchläufe, die Größe der verwendeten Batches, die Lernrate und die Dropout-Rate können gewählt werden. Das Setzen der Parameter muss im Code selbst erfolgen; es existiert keine zentrale Konfigurationsdatei, auch die Übergabe von Parametern beim Ausführen über die Kommandozeile wird nicht unterstützt. Das Initialisieren mit Gewichten aus vorherigen Trainingsvorgängen ist möglich. Für Training und Tests benötigt das Skript zwei Listen, die Pfade zu Ordnern enthalten, die die extrahierten Bilder der einzelnen Trainings- und Testvideos enthalten. Das Skript, das diese Listen für den UCF101-Datensatz erzeugt, wurde neu implementiert; die Neuimplementierung teilt die Ordner zu den einzelnen Videoclips rein zufällig in 80% Trainings- und 20% Testdaten auf und erzeugt die entsprechenden Listen. Wird ein Video für Training oder Test verwendet, werden zufällig 16 zusammenhängende Bilder aus dem entsprechenden Ordner verwendet, um die Aktionsklasse des durch die Bilder repräsentierten Videos zu bestimmen. Im mitgelieferten Testskript wurde darüber hinaus eine wichtige Parameteränderung vorgenommen. Dort war die Dropout-Rate während der Tests standardmäßig auf 40% gesetzt; diese wurde auf 0% geändert. Dabei ist zu beachten, dass das Netzwerk mit einer Wahrscheinlichkeit für die Beibehaltung von Werten arbeitet statt mit einer Dropout-Rate – gibt ein Wert d zwischen 0.0 und 1.0 die Dropout-Rate an, dann gibt $k = 1.0 - d$ die Wahrscheinlichkeit für das Beibehalten von Werten an; dieser Wert muss übergeben werden.

Für die Extraktion von C3D-Features wurde als erster Schritt das C3D-Modell in der Datei *c3d_model.py* so angepasst, dass es neben dem Zugriff auf die Ausgabe auch Zugriff auf die Aktivierungen der beiden Fully Connected Layers gewährt, nachdem jeweils die ReLU-Funktion in diesen Schichten angewendet wurde. Das so veränderte Modell wird anschließend in einem neuen Skript zur Extraktion der C3D-Features verwendet, welches sich grob am *predict_c3d_ucf101.py*-Skript orientiert, das für den Test auf dem UCF101-Datensatz mitgeliefert wurde. Dieses Testskript testet alle zu testenden Videos, indem für jedes Video 16 zusammenhängende Bilder von einer zufälligen Stelle ausgewählt und anschließend verarbeitet werden. Zur Auswahl der Bilder setzt es die *read_clip_and_label*-Methode ein. Sie erhält eine Liste von Pfaden zu Ordnern als Eingabe, von denen jeder

die extrahierten Bilder für jeweils eines der zu testenden Videos enthält. Darüber hinaus wird ein Startindex übergeben, ab welchem die Liste betrachtet werden soll. Ab diesem Startindex werden für eine bestimmte Anzahl an Videos jeweils 16 zusammenhängende Bilder von einer zufälligen Stelle aus extrahiert. Durch die Erhöhung des Startindexes kann so über alle Videos iteriert werden. Im Testskript werden im Anschluss die so extrahierten Bilder durch das C3D-Modell verarbeitet und den jeweiligen Videos wird mit Hilfe der Softmax Layer eine Aktionsklasse zugeordnet. Die so bestimmte Aktionsklasse wird jeweils zusammen mit dem Namen des Videos und dessen tatsächlicher Aktionsklasse aus den Annotationen der Testdaten abgespeichert. Für die Extraktion der Features müssen mehrere Änderungen vorgenommen werden: Pro Video müssen von Anfang an alle vollen 16-Bilder-Blöcke chronologisch extrahiert werden. Dazu wird erst die Gesamtzahl der Bilder für ein zu verarbeitendes Video bestimmt, anschließend wird die Anzahl voller Blöcke berechnet. Es werden anschließend so oft 16 aufeinanderfolgende Bilder extrahiert, wie volle Blöcke exisitieren. Sobald alle vollen Blöcke für ein Video verarbeitet wurden, wird mit dem nächsten Video fortgefahren. Dieses Vorgehen zur Extraktion der Bilder wird in einer überarbeiteten *read_clip_and_label*-Funktion realisiert. Nach wie vor erhält sie als Eingabe eine Liste von Pfaden zu Ordnern und einen Startindex. Dieser verweist jedoch nicht mehr nur auf eine Position in der Liste, sondern auch auf den aktuellen Block an 16 Bildern, ab dem weitergearbeitet werden soll. Ab dieser Stelle werden nach und nach die Blöcke von jeweils 16 Bildern extrahiert; sind für ein Video alle Blöcke verarbeitet, wird mit dem nächsten fortgefahren. Dies wird wiederum für eine feste Anzahl an Blöcken durchgeführt, durch Übergabe entsprechender Startpositionen können alle Blöcke für alle Videos extrahiert werden. Die so erhaltenen Blöcke von jeweils 16 Bildern werden jeweils als Eingabe für das überarbeitete C3D-Modell verwendet, die Aktivierungen der beiden Fully Connected Layers werden extrahiert. Abschließend gilt es noch, die so extrahierten Features sowie weitere durch das SST-Netzwerk benötigte Daten so zu speichern, dass sie durch das SST-Netzwerk verwendet werden können. Das SST-Netzwerk erwartet als Eingabe eine HDF5-Datei, die für jedes Video eine Gruppe mit drei Datensätzen enthält. Im *c3d_features*-Datensatz werden alle C3D-Featurevektoren für das Video als chronologisch angeordnete Liste erwartet, im *total_frames*-Datensatz wird die Gesamtzahl der Bilder des Videos erwartet und im *valid_frames*-Datensatz wird die Anzahl an Frames erwartet, für die die C3D-Features extrahiert wurden - also die Gesamtzahl an Bildern ohne die Bilder des letzten, unvollständigen Blocks. Alle diese Daten stehen nach der Verarbeitung der Bildblöcke und Extraktion der C3D-Featurevektoren zur Verfügung; die Gruppen und Datensätze werden mit Hilfe des *h5py*-Pakets[1] erstellt - mit *h5py.File* kann eine HDF5-Datei erstellt werden, mit der *create_group*-Methode kann auf dieser Datei eine Gruppe erstellt werden und mit der *create_dataset*-Methode kann auf der Gruppe ein Datensatz erstellt werden. Unter Verwendung der vorgestellten Methoden werden zwei HDF5-Dateien erstellt: *fc6_features.hdf5* für die C3D-Features der ersten Fully Connected Layer und *fc7_features.hdf5* für die C3D-Features der zweiten Fully Connected Layer. Für die Weiterverwendung kann dann eine dieser Dateien ausgewählt werden.

Neben dem Skript zur Extraktion der Features wurde ein weiteres Skript erstellt, um die Genauigkeit bei der Aktionserkennung auf dem UCF101-Datensatz zu bestimmen. Dieses operiert auf der Ausgabe des *predict_c3d_ucf101.py*-Skripts, welches für jedes Video die bestimmte Aktionsklasse und die tatsächliche Aktionsklasse enthält. Das Skript muss

[1] `https://www.h5py.org/`

folglich lediglich die Aktionsklassen auslesen, die Gesamtzahl der korrekt bestimmten Aktionsklassen ermitteln und durch die Gesamtzahl der betrachteten Videos teilen.

A.3. Zwischenverarbeitungsschritte

Zwischen der Extraktion der C3D-Features durch das C3D-Netzwerk und der Verwendung im klassischen SST-Netzwerk oder in einem der in Abschnitt 4.3 vorgestellten Two-Stream-Modelle können diverse notwendige und optionale Zwischenverarbeitungsschritte stattfinden. Im Folgenden wird vorgestellt, wie wichtige Zwischenverarbeitungsschritte implementiert wurden, für die keine existierende Implementierung verwendet wurde.

Für die Variante 1 der Two-Stream-Modelle, vorgestellt in Abschnitt 4.3.1, ist es nötig, zwei HDF5-Dateien zu kombinieren. Das Two-Stream-Modell sieht zwei separate C3D-Netzwerke vor, eines für die Verarbeitung der originalen Bilddaten und eines für die Verarbeitung der Bilder, die den zugehörigen optischen Fluss visualisieren. Beide Netzwerke erzeugen dabei eigene C3D-Featurevektoren und speichern diese in HDF5-Dateien. Nachdem entschieden wurde, ob die C3D-Features der fc6- oder fc7-Schicht weiterverwendet werden sollen, steht pro C3D-Netzwerk eine HDF5-Datei mit C3D-Featurevektoren zur Verfügung. Vor der Weiterverarbeitung durch das SST-Netzwerk müssen die Paare von C3D-Featurevektoren, die dieselben Blöcke von 16 Bildern beschreiben, konkateniert werden; anschließend muss eine neue HDF5-Datei mit den konkatenierten C3D-Featurevektoren angelegt werden. Dazu werden die separaten Dateien, die die C3D-Features enthalten, mit Hilfe des *h5py*-Pakets ausgelesen. Für jede Gruppe der ersten Datei wird überprüft, ob sie auch in der zweiten Datei vorhanden ist. Ist dies der Fall, werden beide Listen von C3D-Featurevektoren extrahiert. Im Anschluss wird von vorne beginnend aus beiden Listen jeweils ein C3D-Featurevektor ausgelesen und beide so erhaltenen Vektoren werden konkateniert; mit den nachfolgenden Vektoren wird äquivalent verfahren, bis das Ende einer der beiden Listen erreicht ist. Sollten in der anderen Liste noch Vektoren vorhanden sein, werden diese verworfen. Auf Basis der konkatenierten C3D-Featurevektoren wird anschließend nach bekannter Weise eine neue HDF5-Datei angelegt. Der Abgleich in dieser Implementierung ist dabei nötig, da im Rahmen dieser Arbeit nicht nur selbst berechnete C3D-Features verwendet werden, sondern auch C3D-Features, die von Dritten bereitgestellt wurden. Es kann daher nicht von vorne herein vollständig ausgeschlossen werden, dass beispielsweise ein Video fehlt oder einige C3D-Featurevektoren zu wenig extrahiert wurden.

Aus diesem Grund wurde ein weiteres Skript erstellt, das separate HDF5-Dateien, die C3D-Features enthalten, einander angleicht. Das Vorgehen bei den Vergleichen entspricht dem im vorangegangenen Abschnitt, es wird jedoch für jede Eingabedatei eine eigene Ausgabedatei erzeugt, aus der in der anderen Datei nicht vorhandene Videos und C3D-Featurevektoren gestrichen wurden.

Einer der optionalen Zwischenverarbeitungsschritte ist die L2-Normalisierung der C3D-Features. Diese wird auf den bereits extrahierten und in einer HDF5-Datei gespeicherten C3D-Featurevektoren angewendet. Mit Hilfe des bereits bekannten *h5py*-Pakets wird die HDF5-Datei eingelesen. Für die einzelnen C3D-Featurevektoren wird jeweils mit Hilfe des *numpy*-Pakets[2] die L2-Norm berechnet, anschließend werden die einzelnen Elemente durch

[2] `http://www.numpy.org/license.html`

die berechnete Norm geteilt. Das verwendete *numpy*-Paket ermöglicht dabei eine effiziente Verarbeitung aller C3D-Featurevektoren eines Videos auf einmal unter Verwendung der *numpy.linalg.norm*-Funktion zur Berechnung der L2-Norm und der *numpy.divide*-Funktion für das Teilen durch die Norm. Die so L_2-normalisierten C3D-Features werden wiederum mit Hilfe des *h5py*-Pakets in eine neue Ausgabedatei geschrieben, die genauso strukturiert ist wie die Eingabedatei und sich nur dadurch unterscheidet, dass die enthaltenen C3D-Features mit der L2-Norm normalisiert wurden.

Für den optionalen Zwischenschritt der Hauptkomponentenanalyse wurde die in Abschnitt 5.5 vorgestellte Implementierung verwendet. Es waren keine funktionalen Anpassungen nötig, die Implementierung wurde lediglich auf die verwendete Arbeitsumgebung portiert.

A.4. SST-Netzwerk

Für das SST-Netzwerk wird die in Abschnitt 5.6 genannte existierende Implementierung verwendet. Sie umfasst bereits alle nötigen Funktionen, es sind lediglich Anpassungen für die verwendete Arbeitsumgebung nötig. Im Folgenden wird auf einige Details dieser Implementierungen eingegangen, um eine Grundlage für die vorgenommenen Anpassungen im Rahmen der später vorgestellten Implementierungen der Two-Stream-Modelle zu schaffen.

Im Rahmen der Implementierung steht ein Skript für die nötige Aufteilung der Daten des THUMOS'14-Datensatzes in Trainings-, Validierungs- und Testdaten bereit. Für das Training wird wie bei Buch et al. [BES+17] der Validierungsteil des THUMOS'14-Datensatzes verwendet. Das Skript teilt diesen zufällig in 80% Trainings- und 20% Validierungsdaten auf. Der Testteil des THUMOS'14-Datensatzes wird vollständig den Testdaten zugewiesen. Es existiert eine zentrale Datei, über die Parameter für das SST-Netzwerk festgelegt werden können - unter anderem die Anzahl der GRU-Schichten, die Anzahl der Neuronen pro GRU-Schicht, die Lernrate und viele mehr; alle für die SST-Architektur in Abschnitt 4.2 geforderten Anpassungen können hier vorgenommen werden. Die Versorgung des Netzwerkes mit C3D-Featurevektoren erfolgt durch HDF5-Dateien, deren Erstellungen und Struktur bereits in den vorangegangenen Abschnitten erläutert wurde. Aus ihnen werden die C3D-Featurevektoren ausgelesen und dem Netzwerk zugeführt.

A.5. Two-Stream-Modelle

In diesem Abschnitt werden Implementierungsdetails der verschiedenen entworfenen Two-Stream-Modelle aus Abschnitt 4.3 vorgestellt. Die Implementierung dieser Two-Stream-Modelle erfolgt dabei auf Basis der existierenden und bereits vorgestellten Implementierungen des C3D- und des SST-Netzwerks. Es wird auf notwendige Änderungen an den Netzwerken eingegangen, insbesondere auf die Aufteilung in zwei Streams sowie deren Zusammenführung.

A.5.1. Variante 1: Mid-Fusion durch Kombination der C3D-Features

Für diese Variante, bei der die Fusion der einzelnen Streams nach der Extraktion der C3D-Features durch zwei separate C3D-Netzwerke und vor der Verarbeitung durch ein SST-Netzwerk erfolgt, wurde bereits die Implementierung aller wichtigen Komponenten vorgestellt. Es werden zwei C3D-Netzwerks mit der in Abschnitt 5.4 vorgestellten und

in Anhang A.2 detaillierter betrachteten Implementierung verwendet. Eines dieser C3D-Netzwerke wird auf den originalen Bilddaten trainiert, das andere auf den Bildern des zugehörigen optischen Flusses. Anschließend werden beide zur Extraktion der C3D-Features verwendet - wieder jeweils einmal für die originalen Bilddaten und einmal für die Bilder des optischen Flusses. Prinzipiell können statt den C3D-Netzwerken auch vorab extrahierte C3D-Features für diese separaten Streams verwendet werden. Die Implementierung der sich anschließenden optionalen Zwischenverarbeitungsschritte wurde bereits in Anhang A.3 vorgestellt, ebenso wie die Implementierung der Zusammenführung der der C3D-Featurevektoren beider Streams zu konkatenierten C3D-Featurevektoren. An der Implementierung des sich anschließenden SST-Netzwerks muss ebenfalls keine Änderung vorgenommen werden: Die Größe der C3D-Featurevektoren, die eingegeben werden, kann bereits parametrisch angepasst werden – also auch auf die Größe der konkatenierten C3D-Featurevektoren.

A.5.2. Variante 2: Fusion in der fc7-Schicht des C3D-Netzwerks

Bei Variante 2 eines Two-Stream-Modells erfolgt die Fusion durch zwei separate C3D-Netzwerke, die nach den separaten fc6-Schichten in einer gemeinsamen fc7-Schicht zusammengeführt werden. Für diese Variante wird die bekannte Implementierung des C3D-Netzwerks als Grundlage verwendet und so erweitert, dass sie diesem Modell entspricht. Die Implementierung der sich anschließenden optionalen Zwischenverarbeitungsschritte und des SST-Netzwerks bleibt unverändert.

Die bestehende Implementierung des C3D-Netzwerk wird zuerst so abgeändert, dass eine Kopie der Schichten bis einschließlich der fc6-Schicht angelegt wird. So entstehen zwei separate Streams; als Eingabe werden nun entsprechend 16 zusammenhängende Bilder für jeden der Streams benötigt. Einer der Streams erhält die originalen Bilddaten, der andere die zugehörigen Bilder des optischen Flusses. Nach der Aktivierungsfunktion und der Anwendung der Dropout-Rate in der fc6-Schicht beider Streams werden die Ausgaben unter Einsatz der TensorFlow-Funktion *tf.concat* von zwei 4096-elementigen Vektoren zu einem 8192-elementigen Vektor kombiniert, der als Eingabe für die fc7-Schicht fungiert. Entsprechend wird die fc7-Schicht so modifiziert, dass sie 8192-elementige Vektoren als Eingaben akzeptiert. Für das Modell wird darüber hinaus die Möglichkeit implementiert, vorab bestimmte Gewichte für die einzelnen Streams bis einschließlich zur fc6-Schicht zu laden - hierzu wird die Klasse *tf.train.Saver* verwendet. Ihr kann bei der Erstellung eine Variablenliste übergeben werden, die angibt, welche Gewichte wiederhergestellt werden sollen; mit dem Aufruf der *restore*-Methode der Klasse lassen sich dann die dort angegebenen Gewichte mit abgespeicherten Werten aus vorherigen Trainingvorgängen wiederherstellen. Zusätzlich wird die Möglichkeit implementiert, nur die Gewichte der fc7-Schicht und der nachfolgenden Softmax Layer anzupassen - hierzu wird wieder eine Variablenliste mit den anzupassenden Gewichten verwendet, die dann bei der Berechnung der Gradienten der *compute_gradients*-Methode der *tf.train.AdamOptimizer*-Klasse übergeben wird.

Zusätzlich zu den Veränderungen am Netzwerk selbst müssen Änderungen bei der Zuführung der zu verarbeitenden Daten vorgenommen werden: Die Aufteilung in Trainings- und Testdaten soll sowohl für die originalen Bilddaten als auch für die Bilder des optischen Flusses identisch sein; darüber hinaus sollen dem Netzwerk für beide Streams jeweils 16 Bilder zugeführt werden, die aus dem gleichen Abschnitt des gleichen Videos stammen -

einmal 16 originale Bilder und einmal 16 Bilder des zugehörigen optischen Flusses. Ersteres wird durch eine Modifikation des Skriptes zum Erstellen der Listen für die Trainings- und Testdaten erreicht: Sowohl für die originalen Bilddaten als auch für den optischen Fluss werden jeweils zwei Listen erstellt, eine für die Trainingsdaten und eine für die Testdaten. Es wird für jeden Ordner, der die Bilder eines Videos enthält, bestimmt, ob er zur Trainings- oder zur Testliste gehören soll - für den entsprechenden Ordner, der den zugehörigen optischen Fluss enthält, wird dann die gleiche Entscheidung getroffen. Alternativ können aber auch nur die Trainings- und Testliste für die originalen Bilddaten wie bisher erzeugt werden, die Trainings- und Testliste für die Bilder des optischen Flusses kann dann durch Umbenennen der Pfade zu den Ordnern erhalten werden.

Um dem Netzwerk die Daten für Training und Tests auf Basis der Listen zuzuführen, wird die *read_clip_and_label*-Methode so angepasst, dass sie statt einmal 16 Bilder zweimal 16 Bilder liefert. Die zufällige Wahl von Video und Position im Video, die während des Trainings für die Extraktion 16 zusammenhängender Bilder getroffen werden muss, wird ebenfalls modifiziert. Die Entscheidung, von welchem Video und von welcher Stelle im Video extrahiert werden soll, wird nun zu Beginn der Methode getroffen. Anschließend wird die Entscheidung sowohl für die originalen Bilddaten als auch für die Bilder des optischen Flusses einheitlich angewendet. Würde man die ursprüngliche Methode einmal für die originalen Bilddaten und ein weiteres Mal für die Bilder des optischen Flusses aufrufen, würden die Bilder des optischen Flusses nicht zu den originalen Bilddaten passen, da die Wahl des Videos und der Stelle im Video zur Extraktion unabhängig voneinander zufällig getroffen werden würde. Für die spätere Extraktion der Features lässt sich die Methode wie in Anhang A.2 beschrieben durch die Übergabe einer Startposition steuern und extrahiert dann die benötigten zueinander gehörenden Blöcke von jeweils 16 Bildern.

A.5.3. Variante 3: Fusion durch Bilden eines gewichteten Durchschnitts im SST-Netzwerk

Bei Variante 3 der Fusion werden die Streams zusammengeführt, indem über die Ausgabe der Konfidenzwerte zweier separater SST-Netzwerke der gewichtete Durchschnitt gebildet wird. Eines dieser SST-Netzwerke operiert auf C3D-Featurevektoren, die aus den originalen Bilddaten extrahiert wurden, das andere operiert auf C3D-Featurevektoren, die aus Bildern des optischen Flusses extrahiert wurden. Für die Extraktion der C3D-Featurevektoren werden C3D-Netzwerke mit der bekannten Implementierung verwendet; alternativ können vorab extrahierte C3D-Featurevektoren verwendet werden. Für die optionalen Zwischenverarbeitungsschritte werden die bereits bekannten Implementierungen verwendet. Als Grundlage für dieses Netzwerk dient die verwendete Implementierung des SST-Netzwerks. Es wird eine Kopie aller Schichten des Netzwerks erstellt, einschließlich der Fully Connected Layer mit der logistischen Sigmoidfunktion als Aktivierungsfunktion, welche die Konfidenzwerte für verschiedene Zeitfenster ausgibt. Nun werden von zwei Streams Konfidenzwerte ausgegeben. Es werden neue Variablen eingeführt, sodass die Anzahl an GRU Layers und an Neuronen pro GRU Layer für beide Streams separat festgelegt werden kann - die Streams müssen folglich nicht symmetrisch sein. Für die Gewichtung wird ein weiterer neuer Parameter *flow_stream_weight* eingeführt, der das Gewicht für die Konfidenzwerte des Streams für den optischen Fluss angibt. Der Wert muss sich zwischen 0 und 1 befinden; die Konfidenzwerte des Streams für die originalen Bilddaten werden dann mit $1 - \textit{flow\_stream\_weight}$ gewichtet. Mit Hilfe der TensorFlow-Funktion *tf.scalar_mul* werden

die jeweiligen Wahrscheinlichkeitsvektoren mit dem zugehörigen Gewicht multipliziert, mit der Funktion *tf.add* dann anschließend aufsummiert. Die so gewichtet aufsummierten Konfidenzwerte werden anschließend ausgegeben. Die Versorgung mit C3D-Featurevektoren erfolgt nun durch zwei HDF5-Dateien: Eine für die C3D-Featurevektoren, die auf Basis der originalen Bilddaten extrahiert wurden, und eine für die C3D-Featurevektoren, die auf Basis der Bilder des optischen Flusses extrahiert wurden. Die Methode, die die Features für Training und Tests ausliest, wird so modifiziert, dass beide Dateien jeweils an derselben Stelle ausgelesen werden und somit die korrespondierenden C3D-Featurevektoren für beide Streams bereitgestellt werden. Wie bei der Implementierung von Variante 2 in Abschnitt A.5.2 wird darüber hinaus ermöglicht, die einzelnen SST-Netzwerke, die am Ende zusammengeführt werden, mit vorab bestimmten Gewichten zu initialisieren – da sich der Aufbau der einzelnen SST-Netzwerke bis auf die Zusammenführung durch eine gewichtete Summe nicht von denen klassischer SST-Netzwerke unterscheidet, können sie vollständig mit Gewichten von solchen Netzwerken initialisiert werden, ohne dass ein weiteres Training zwingend notwendig wäre.

A.5.4. Variante 4: Fusion durch Fully Connected Layer im SST-Netzwerk

Die hier betrachtete Variante 4 der Fusion ähnelt in ihrer Implementierung stark der Variante 3. Die Implementierung der C3D-Netzwerke und Zwischenverarbeitungsschritte wird mit ihr geteilt, ebenso wie die Implementierung der Versorgung mit C3D-Features. Wie bei der Implementierung von Variante 3 wird eine Kopie der Schichten des SST-Netzwerks erstellt, jedoch nur bis zur Fully Connected Layer, die nicht mitkopiert wird. Wiederum werden neue Parameter eingeführt, sodass die Anzahl der GRU Layers und die Anzahl der Neuronen pro GRU Layer in beiden Streams nicht gleich sein muss. Wie bei Variante 3 wird so implementiert, dass die Gewichte der einzelnen Streams durch vorheriges Training bestimmte Gewichte initialisiert werden können - die Gewichte der Fully Connected Layer ausgenommen, die zur Fusion verwendet wird. Diese Fusion erfolgt dadurch, dass die Ausgaben der jeweils letzten GRU Layer der einzelnen Streams unter Verwendung der *tf.concat*-Funktion konkateniert werden. Entsprechend vergrößert sich die Eingabe in die Fully Connected Layer auf die Summe der Größe der Ausgaben beider GRU Layers, festgelegt durch die Anzahl der Neuronen beider Schichten. Um das Training der Fully Connected Layer zu ermöglichen, ohne dass die vorab bestimmten Gewichte der anderen Schichten angepasst werden, wurde ein zusätzlicher Parameter eingeführt, der bestimmt, ob die zum Lernen verwendete Funktion *minimize* der *tf.train.AdamOptimizer*-Klasse nur auf den Gewichten der Fully Connected Layer oder auf allen Gewichten ausgeführt wird.

Literaturverzeichnis

[BBPW04] Thomas Brox, Andrés Bruhn, Nils Papenberg und Joachim Weickert: *High Accuracy Optical Flow Estimation Based on a Theory for Warping.* In: *European Conference on Computer Vision*, Seiten 25–36. Springer, 2004.

[BEG+17] Shyamal Buch, Victor Escorcia, Bernard Ghanem, Li Fei-Fei und Juan C. Niebles: *End-to-End, Single-Stream Temporal Action Detection in Untrimmed Videos.* In: *Proceedings of the British Machine Vision Conference (BMVC)*, 2017.

[BES+17] Shyamal Buch, Victor Escorcia, Chuanqi Shen, Bernard Ghanem und Juan C. Niebles: *SST: Single-Stream Temporal Action Proposals.* In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Seiten 6373–6382, 2017.

[BHL+12] Ilja N. Bronstein, Juraj Hromkovic, Bernd Luderer, Hans Rudolf Schwarz, Jochen Blath, Alexander Schied, Stephan Dempe, Gert Wanka und Siegfried Gottwald: *Taschenbuch der Mathematik*, Band 1. Springer-Verlag, 2012, ISBN 978-3-8351-0123-4.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho und Yoshua Bengio: *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.* arXiv preprint arXiv:1412.3555, 2014.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk und Yoshua Bengio: *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* arXiv preprint arXiv:1406.1078, 2014.

[CVS+18] Yu Wei Chao, Sudheendra Vijayanarasimhan, Bryan Seybold, David A. Ross, Jia Deng und Rahul Sukthankar: *Rethinking the Faster R-CNN Architecture for Temporal Action Localization.* In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Seiten 1130–1139, 2018.

[CZ17] Joao Carreira und Andrew Zisserman: *Quo Vadis, Action Recognition? A New Model and the Kinetics Dataset.* In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Seiten 4724–4733, 2017.

[DFI+15] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers und Thomas

Brox: *FlowNet: Learning Optical Flow with Convolutional Networks.* In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Seiten 2758–2766, 2015.

[DHS11] John Duchi, Elad Hazan und Yoram Singer: *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.* Journal of Machine Learning Research, 12(Jul):2121–2159, 2011.

[EHNG16] Victor Escorcia, Fabian C. Heilbron, Juan C. Niebles und Bernard Ghanem: *DAPs: Deep Action Proposals for Action Understanding.* In: *European Conference on Computer Vision*, Seiten 768–784. Springer, 2016.

[Fan18] Armando Fandango: *Mastering TensorFlow 1.x: Advanced machine learning and deep learning concepts using TensorFlow 1.x and Keras.* Packt Publishing, 2018, ISBN 978-1-78829-206-1.

[GBC16] Ian Goodfellow, Yoshua Bengio und Aaron Courville: *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[GYN17] Jiyang Gao, Zhenheng Yang und Ram Nevatia: *Cascaded Boundary Regression for Temporal Action Detection.* arXiv preprint arXiv:1705.01180, 2017.

[GYS+17] Jiyang Gao, Zhenheng Yang, Chen Sun, Kan Chen und Ram Nevatia: *TURN TAP: Temporal Unit Regression Network for Temporal Action Proposals.* arXiv preprint arXiv:1703.06189, 2017.

[HS81] Berthold K. P. Horn und Brian G. Schunck: *Determining Optical Flow.* Artificial intelligence, 17(1-3):185–203, 1981.

[IMS+17] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy und Thomas Brox: *FlowNet 2.0: Evolution of Optical Flow Estimation with Deep Networks.* In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Band 2, Seite 6, 2017.

[JLZ+14] Yu Gang Jiang, Jingen Liu, Amir R. Zamir, George Toderici, Ivan Laptev, Mubarak Shah und Rahul Sukthankar: *THUMOS Challenge: Action Recognition with a Large Number of Classes.* `http://crcv.ucf.edu/THUMOS14/`, 2014.

[Joh73] Gunnar Johansson: *Visual Perception of Biological Motion and a Model for its Analysis.* Perception & Psychophysics, 14(2):201–211, 1973.

[KB14] Diederik P. Kingma und Jimmy Ba: *Adam: A Method for Stochastic Optimization.* arXiv preprint arXiv:1412.6980, 2014.

[KCS+17] Will Kay, Joao Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev und andere: *The Kinetics Human Action Video Dataset.* arXiv preprint arXiv:1705.06950, 2017.

[KT18] Van Minh Khong und Thanh Hai Tran: *Improving Human Action Recognition with Two-Stream 3D Convolutional Neural Network.* In: *1st International Conference on Multimedia Analysis and Pattern Recognition (MAPR)*, Seiten 1–6. IEEE, 2018.

[KTS+14] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar und Li Fei-Fei: *Large-Scale Video Classification with Convolutional Neural Networks.* In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Seiten 1725–1732, 2014.

[LBD+89] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard und Lawrence D. Jackel: *Backpropagation Applied to Handwritten Zip Code Recognition.* Neural Computation, 1(4):541–551, 1989.

[LZS17] Tianwei Lin, Xu Zhao und Zheng Shou: *Temporal Convolution Based Action Proposal: Submission to ActivityNet 2017.* arXiv preprint arXiv:1707.06750, 2017.

[NLPH18] Phuc Nguyen, Ting Liu, Gautam Prasad und Bohyung Han: *Weakly Supervised Action Localization by Sparse Temporal Pooling Network.* In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Seiten 6752–6761, 2018.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams: *Learning Representations by Back-Propagating Errors.* nature, 323(6088):533, 1986.

[RN12] Stuart Russell und Peter Norvig: *Künstliche Intelligenz: Ein moderner Ansatz.* Pearson Studium, 3., aktualisierte Auflage, 2012, ISBN 978-3-86894-098-5.

[SLLG+17] Laura Sevilla-Lara, Yiyi Liao, Fatma Guney, Varun Jampani, Andreas Geiger und Michael J. Black: *On the Integration of Optical Flow and Action Recognition.* arXiv preprint arXiv:1712.08416, 2017.

[Ste08] Johannes Steinmüller: *Bildanalyse: Von der Bildverarbeitung zur räumlichen Interpretation von Bildern.* Springer-Verlag, 2008, ISBN 978-3-540-79743-2.

[SWC16] Zheng Shou, Dongang Wang und Shih Fu Chang: *Temporal Action Localization in Untrimmed Videos via Multi-Stage CNNs.* In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Seiten 1049–1058, 2016.

[SZ14] Karen Simonyan und Andrew Zisserman: *Very Deep Convolutional Networks for Large-Scale Image Recognition.* arXiv preprint arXiv:1409.1556, 2014.

[SZS12] Khurram Soomro, Amir R. Zamir und Mubarak Shah: *UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild.* In: *CRCV-TR-12-01*, 2012.

[TBF+15] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani und Manohar Paluri: *Learning Spatiotemporal Features with 3D Convolutional Networks.* In: *Proceedings of the IEEE International Conference on Computer Vision*, Seiten 4489–4497, 2015.

[TH12] Tijmen Tieleman und Geoffrey Hinton: *Lecture 6.5-RMSProp, COURSERA: Neural Networks for Machine Learning.* University of Toronto, Technical Report, 2012.

[VJ01] Paul Viola und Michael Jones: *Rapid Object Detection using a Boosted Cascade of Simple Features*. In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2001. CVPR 2001.*, Band 1, Seiten I–I, 2001.

[VLS18] Gül Varol, Ivan Laptev und Cordelia Schmid: *Long-term Temporal Convolutions for Action Recognition*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 40(6):1510–1517, 2018.

[WXW+16] Limin Wang, Yuanjun Xiong, Zhe Wang, Yu Qiao, Dahua Lin, Xiaoou Tang und Luc Van Gool: *Temporal Segment Networks: Towards Good Practices for Deep Action Recognition*. In: *European Conference on Computer Vision*, Seiten 20–36. Springer, 2016.