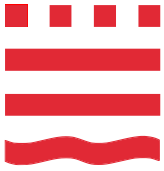

Technische
Hochschule
Brandenburg

Fachbereich Informatik und Medien

MASTERARBEIT

Deep Learning zur Objektdetektion in Bildern mit Region-based Convolutional Neural Networks und GPU-Computing

Vorgelegt von: Jonas Preckwinkel

am: 14.03.2018

zum Erlangen des akademischen Grades

MASTER OF SCIENCE

(M. Sc.)

Erstgutachter: Prof. Dr. Sven Buchholz

Zweitgutachter: Dipl.-Inf. Ingo Boersch

Inhaltsverzeichnis

1 Einleitung

Eines der Zugpferde der angeheizten Deep Learning Diskussionen in den letzten Jahren sind die Erfolge, die in Anwendungen der Bildklassifikation erzielt wurden. 2012 konnten die Autoren von „ImageNet Classification with Deep Convolutional Neural Networks" [1] Fehlerraten in einem Bildklassifikationswettbewerb um fast die Hälfte reduzieren (26,2% hatte der 2. Platz, 15,3% Platz 1). Erreichen konnten sie das durch Reduzierung der Trainingszeiten mit der Optimierung der Convolution Operation und Benutzung von GPUs, sowie durch die Anwendung von neuen Methoden wie Dropout und der ReLU Funktion. Seither bringt die Entwicklung von neuen Methoden und Hardwareunterstützung Deep Learning voran. Durch die Nähe zur Bildklassifikation wurden auch Lösungen zur Objektdetektion vorangetrieben, indem die Objektdetektion auf das Problem der Klassifikation reduziert wurde. Angefangen mit Algorithmen, die potentielle Objekte in einem Bild ausfindig machen und damit das Objektdetektionsproblem auf ein Klassifikationsproblem reduzieren, konnten die Autoren RCNN immer weiter optimieren, bis ausschließlich künstliche neuronale Netze zum Einsatz kamen. Mit der Verbesserung von Detektionsmethoden im Hinblick auf Treffergenauigkeit und benötigtem Rechenaufwand lassen sich immer neue Anwendungsgebiete erschliessen.

Die Detektion von Objektklassen in Bildern mit Hilfe von Region-based Convolutional Neural Networks soll unter der Verwendung von GPU-Computing durchgeführt werden. Verschiedene Implementierungen von Region-based Objektdetektionssystemen sollen erarbeitet werden, um ein generelles Verständnis dieser zu erlangen. Durch das Verändern von Netzparametern beim Training dieser Implementierungen und durch Evaluierung der resultierenden Modelle sollen Erkenntnisse für die Wahl von bedeutenden Konfigurationen in ähnlichen Anwendungssituationen gewonnen werden.

Im folgenden Kapitel werden die benötigten Bestandteile beschrieben, die zum Trainieren von künstlichen Neuronalen Netzen nötig sind. Dazu gehören die Netzstrukturen Feedforward-Netze und Convolutional Neuronale Netze und andere unabkömmliche Bestandteile, um den Netzen das Lernen zu ermöglichen. In Kapitel 3 wird die Objektklassendetektion von der Klassifikation abgeleitet und die Funktionsweise einer Reihe von aufeinander aufbauenden Lösungen des Problems erläutert. Kapitel 4 beschreibt das verwendete Systemsetup mit

Programmbibliotheken und Hardware, sowie die Vorgehensweise im Trainingsprozess. Daraufhin werden die Trainingsversuche vorgestellt und von den Ergebnissen Rückschlüsse auf Parameterwahl getroffen, sowie Best Practice Hinweise gegeben.

2 Deep Learning

Maschinelles Lernen hat zum Ziel, Computern Strategien zur Bearbeitung einer bestimmten Aufgabe beizubringen, ohne den Computer konkret dafür programmiert zu haben. Dazu entwickelte Methoden sollen dem Computer ermöglichen, selber einen Weg für die Lösung der Aufgabe zu finden. Sprachen zu verstehen oder die Bilderkennung sind Aufgaben, die ohne maschinelles Lernen ein erhebliches Problem darstellen, inzwischen jedoch durch diese Methoden mit den menschlichen Fähigkeiten mithalten oder sie übertreffen können. Das Lernen erfolgt in der Regel durch Daten in Form von Beispielen, bei denen die Eingabe und das dazugehörige Ergebnis bereits bekannt sind (Überwachtes Lernen). Eine der Methoden des maschinellen Lernens sind künstliche neuronale Netze. Künstliche neuronale Netze sind durch biologische neuronale Netze inspiriert und kopieren davon stark vereinfacht einige Funktionsweisen [2].

In den 1940ern wurden die ersten Schritte in der Entwicklung der künstlichen neuronalen Netze getätigt. Die anfängliche Blütezeit ging bis 1969, bis durch eine mathematische Analyse gezeigt wurde, dass einige wichtige Probleme (XOR, lineare Separierbarkeit) von künstlichen neuronalen Netzen nicht lösbar waren. Daraufhin wurden viele Forschungsgelder gestrichen, wodurch es bis 1985 keine Konferenzen mehr und nur wenige Veröffentlichungen gab. Die Renaissance wurde einerseits durch John Hopfield eingeleitet, der eine Lösung des Travelling Salesman Problems[1] mit einer eigenen Form der künstlichen neuronalen Netzen vorstellte [3], als auch durch die Veröffentlichung des Backpropagation Verfahrens für künstliche neuronale Netze mit mehreren Schichten [4], dass das Problem der linearen Separierbarkeit löste. Seit 2006 wird der Begriff Deep Learning oder Deep Neural Nets für die Anwendung von künstlichen neuronalen Netzen verwendet.

Deep Learning wird seit ein paar Jahren in den Bereichen Spracherkennung, Bilderkennung und Verhaltenserkennung besonders erfolgreich angewendet. Hier haben die traditionellen Techniken des maschinellen Lernens Probleme zu generalisieren, da der Merkmalsraum zu groß wird. Die Fähigkeit zu generalisieren ist eine wichtige Eigenschaft von gelernten Model-

[1] TSP ist ein Optimierungsproblem bei dem versucht wird, den kürzesten Weg durch eine Menge von Orten zu finden, wobei Anfangs- und Zielort gleich sind.

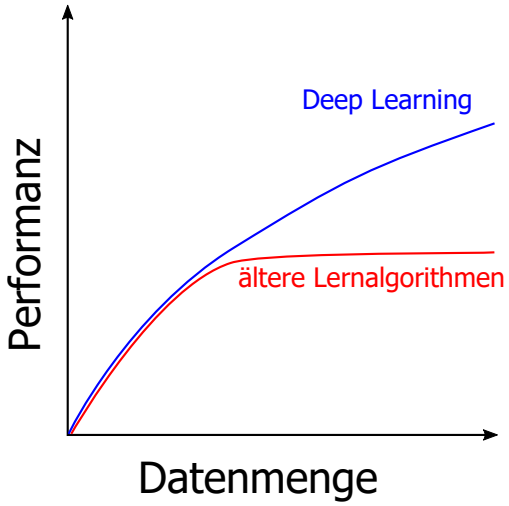

Abbildung 2.1: Skalierung von Lernalgorithmen mit der verfügbaren Menge von Daten (Bildidee von Andrew Ng im Rahmen seines Vortrags der Extract Data Conference)

len. Das gelernte Modell soll nicht nur die Fälle wiedererkennen, die es in den Trainingsdaten bereits gesehen hat, sondern auch Unbekanntes, richtig einordnen können. Viel-Dimensionale Daten[2] erhöhen die Schwierigkeit, zu neuen Beispielen zu generalisieren, exponentiell. Deep Learning ist designiert, um dieses Problem zu überwinden, an dem die traditionellen Methoden scheitern [5, S.152]. Deep Learning ist skalierbar. Durch die andauernd steigende Rechenleistung moderner Computer werden immer komplexere Modelle in der Praxis möglich. Außerdem profitieren künstliche neuronale Netze enorm von den Mengen der verfügbaren Daten, anhand derer sie lernen können. Diese werden ebenfalls leichter zu erheben durch die fortschreitende Digitalisierung der Welt. Die traditionellen Methoden erreichen bei geringen Datenmengen bereits ihre besten Ergebnisse (s. Abb. 2.1). Eine weitere Eigenschaft, die Deep Learning zugute kommt, ist das automatische Lernen von Merkmalen. Dem Computer die Merkmale eines Menschen formal zu beschreiben, damit dieser immer erkannt wird, ist ein schwieriges Unterfangen. Deep Learning Algorithmen haben zum Ziel, hierarchisch über eine Anzahl von Schichten, Merkmale zu erlernen. Wobei die vorderen Schichten grundsätzliche Merkmale lernen (z.B. Kanten, Konturen) und die weiter hinten in der Hierarchie liegenden Schichten speziellere Merkmale lernen, die beispielsweise einem Gesicht ähnlich sehen. Im „Deep Learning Book“ [5, S.1] definieren die Autoren Deep Learning als das Lernen komplizierter Konzepte aus einfachen Konzepten in einer Hierarchie von Schichten:

> The hierarchy of concepts allows the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these

[2] Beispielsweise bei Bilddaten zählt jeder Pixel als Merkmal bzw. Dimension.

> concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI deep learning.

Zum Anfang dieses Kapitels werden Grundlagen von künstlichen neuronalen Netzen erläutert, um im Anschluss Deep Learning Konzepte verstehen zu können. Dazu gehören die Feed Forward Neuronalen Netze (auch Multilayer Perceptrons genannt) und Konzepte, die ein künstliches neuronales Netz dazu befähigen lernen zu können. Daraufhin werden Regularization Methoden vorgestellt die verhindern, dass lernende Netze dem „overfitting" verfallen. Im Anschluss werden die Convolutional Neural Networks mit ihren eigenen Funktionsweisen erläutert.

2.1 Grundlagen künstlicher neuronaler Netze

Künstliche neuronale Netze bestehen aus Neuronen, sowie gerichteten und gewichteten Verbindungen untereinander. In einem Neuron j finden Berechnungen statt, um dessen Ausgabe zu ermitteln. Die Ausgabe beruht auf folgenden Werten:

1. der Ausgabe der Neuronen die eine gerichtete Verbindung zu j haben I
2. der Gewichte dieser Verbindungen W
3. einem Schwellenwert[3] der auch Bias genannt wird b
4. einer Propagierungsfunktion f_{prop}
5. sowie einer Aktivierungsfunktion f_{act}.

Zu jeder Verbindung die zwischen zwei Neuronen i und j besteht, gibt es ein Gewicht, das den Einfluss von i auf j verstärkt (falls das Gewicht positiv ist) oder hemmt (falls negativ). Die Gewichte eines KNN werden üblicherweise in einer Gewichtsmatrix W abgespeichert. Das Gewicht zwischen i und j wird daher mit $w_{i,j}$ bezeichnet. Jedes Neuron hat außerdem ein Bias. Das Bias kann als die Konstante einer linearen Funktion gesehen werden, mit der der Verlauf von ihr geändert werden kann. Abb. 2.2 veranschaulicht dies anhand einer Geraden $y = ax + b$, die ohne die Konstante b die beiden Datenmengen nicht optimal teilen könnte. Die Gewichte und Bias werden während der Trainingsphase erlernt und dienen als Eingaben der Propagierungsfunktion. Die Propagierungsfunktion f_{prop} von Neuron j berechnet einen Wert aufgrund der Ausgaben der Neuronen, die eine Verbindung zu j

[3] Die Bezeichnung Schwellenwert kommt von den Perzeptrons, der ersten Form von künstlichen Neuronen entwickelt von Frank Rosenblatt zwischen 1950 und 1960. Ein Perzeptron hat eine binäre Ausgabe die am Schwellenwert zwischen 0 und 1 wechselt [6]. Die jetzt in der Praxis vorkommenden Neuronen haben eine kontinuierliche Ausgabe (abhängig von der Aktivierungsfunktion).

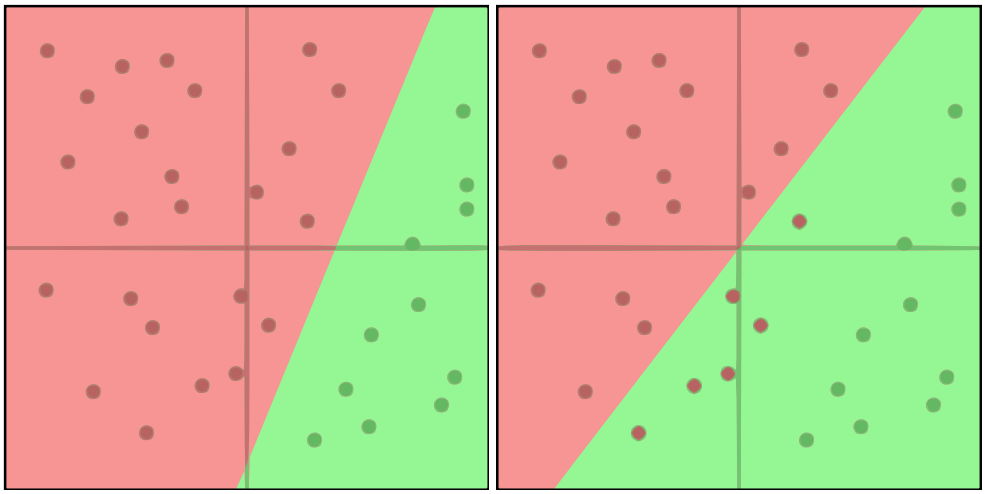

Abbildung 2.2: Veranschaulichung der Funktion des Bias anhand einer Geraden, die die Datenmenge optimal teilen soll. Links mit Bias. Rechts ohne Bias ist sie nicht in der Lage alle Datenpunkte richtig einzuordnen.

haben $x_1, ..., x_n$ und der dazugehörigen Gewichte $w_1, ..., w_n$. Üblicherweise kommt dafür die gewichtete Summe $f_{prop} = \sum_{i \epsilon I} x_i w_i + b$ zum Einsatz. Das Ergebnis von f_{prop} wird weiter zur Aktivierungsfunktion f_{act} geleitet (s. Abb. 2.3). Die Aktivierungsfunktion transformiert die Eingabe in einen Aktivierungswert, der in einem bestimmten Wertebereich abhängig von der Aktivierungsfunktion liegt. Der Aktivierungswert wird entweder als Eingabe in weiteren Neuronen verwendet oder trägt zur Problemlösung (Entscheidung bei der Klassifikation oder Schätzung einer Variablen (Regression)) bei. Eine Aktivierungsfunktion sollte ein paar Eigenschaften erfüllen. Um ein nicht-lineares Problem lösen zu können, müssen sie einen nicht-linearen Teil haben. Sonst ist die Lernfähigkeit des KNN auf das Lösen von linearen Problemen beschränkt. Des Weiteren muss die Aktivierungsfunktion an jeder Stelle differenzierbar sein, da das Backpropagation Lernverfahren darauf basiert. Die in der Praxis hauptsächlich verwendeten Aktivierungsfunktionen sind die Sigmoid Funktion, der Tangens Hyperbolicus (tanh) und die Rectified Linear Unit Funktion (ReLU) zu sehen in Abb. 2.4.

2.2 Feed Forward Neuronale Netze

Feed Forward Neuronale Netze (FFNN) sind die einfachsten Arten von künstlichen neuronalen Netzen. Sie bestehen immer aus einer Eingabe- und Ausgabeschicht, sowie beliebig vielen verdeckten Schichten. Neuronen einer Schicht sind mit allen Neuronen der darauf folgenden Schicht verbunden und werden daher Fully-connected Layer genannt. Informationen werden von vorn nach hinten durch die Schichten verarbeitet, daher der Name „Feed Forward". Modifikationen dieser Struktur gibt es für verschiedene Anwendungsgebiete, so werden in

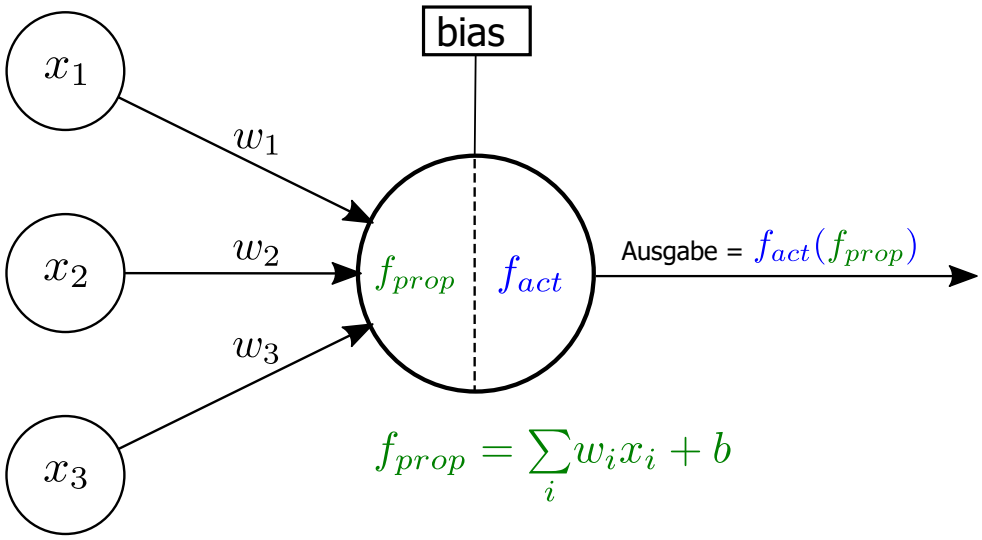

Abbildung 2.3: Veranschaulichung der Elemente eines technischen Neurons

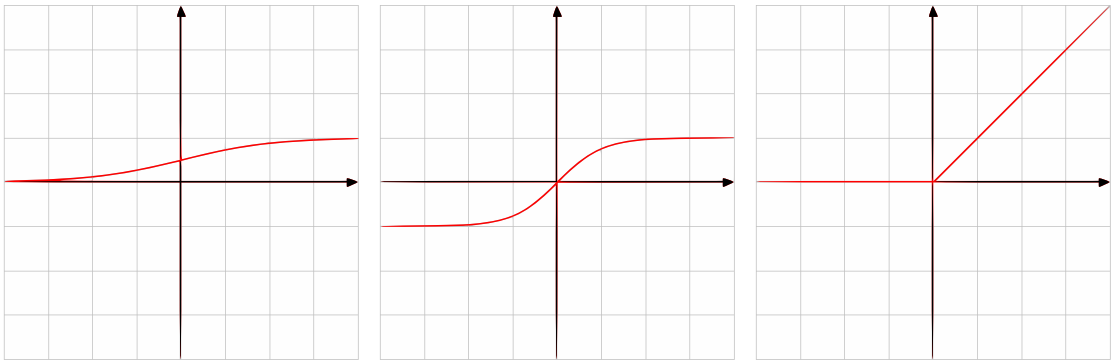

Abbildung 2.4: Neuronen Aktivierungsfunktionen: Sigmoid (links), tanh (mitte), ReLU (rechts)

rekurrenten neuronalen Netzen (RNN) auch Verbindungen einer Schicht in dieselbe oder davor liegende Schichten erlaubt und eingesetzt in Anwendungen, die die natürliche Sprache betreffen. Convolutional Neural Networks (CNN, zu Deutsch „faltendes neuronales Netz“) sind auch eine Modifikation der Feedforward-Netze und sind besonders geeignet für Bild- und Audioapplikationen, da sie kontextsensitiv sind und zum Beispiel nicht jeden Pixel in einem Bild einzeln auswerten, sondern die umliegenden Pixel mit einbeziehen. Die zu verarbeitenden Daten werden über die Eingabeschicht dem Netz übergeben. In den verdeckten Schichten wird die Eingabe in für die Ausgabeschicht brauchbare Informationen transformiert. In der Ausgabeschicht wird festgelegt, in welcher Form das Ergebnis der transformierten Eingaben ausgegeben wird.

FFNN haben zum Ziel, eine Funktion $f(x;\Theta)$ an die Zielfunktion[4] $f*$ einer Datenmenge anzunähern[5]. Dies geschieht mithilfe von verrauschten Trainingsbeispielen entnommen aus $f*$, anhand derer das FFNN die Parameter Θ lernt. Zu jedem Trainingsbeispiel x gehört eine Beobachtung (auch Label genannt) $y \approx f*(x)$, der das berechnete Label $\hat{y} = f(x;\Theta)$ nach dem Lernprozess möglichst nahe kommen soll. Informationen über die Schichten zwischen der Eingabe und Ausgabe sind jedoch nicht in den Trainingsbeispielen zu finden. Daher werden diese Schichten „verdeckte Schichten“ genannt[5, S.169]. Ein Lernalgorithmus ist verantwortlich für die Nutzbarmachung der verdeckten Schichten zur Schätzung der Zielfunktion $f*$. Die zu lernende Funktion f besteht aus mehreren verketteten Funktionen. Besteht ein Netzwerk aus drei Schichten so berechnet es die Funktion $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$, wobei $f^{(1)}$ für die erste verdeckte Schicht steht, $f^{(2)}$ für die zweite verdeckte Schicht und die letzte Schicht mit der Funktion $f^{(3)}$ für die Ausgabeschicht.

Im Folgenden wird die Funktionsweise von Feedforward-Netzen anhand von einem Beispiel deutlich gemacht. Daraufhin werden die prominentesten Aktivierungsfunktionen mit ihren Vor- und Nachteilen vorgestellt. Im Anschluss werden die benötigten Konzepte, die ein lernfähiges künstliches neuronales Netz ausmachen, erläutert. Dazu wird auf die Berechnung des Fehlers eines Modells mithilfe von Kostenfunktionen eingegangen. Dieser Fehler ist letztendlich die Ausgangsbasis des Optimierungsalgorithmus Stochastic Gradient Descend (SGD) der die Parameter Θ so verändern soll, dass der Fehler kleiner wird. Im Anschluss wird der Backpropagation Algorithmus erläutert, welcher SGD in eleganter Weise bei der Berechnung der Parameterupdates unterstützt.

[4] Die Zielfunktion ist unbekannt. Die zum Lernen verfügbare Datenmenge entspringt der Zielfunktion und wird verwendet, um diese möglichst gut zu schätzen.

[5] Man kann sich vorstellen, wenn $f*$ die Menge aller möglichen Bilder von Katzen beschreibt, dass mit f versucht wird, möglichst genau diese Bilder als „Bild mit Katze drin“ zu erkennen.

2.2.1 XOR-Beispiel

An diesem Beispiel wird demonstriert, wie FFNN die Berechnungen durchführen, um von der Eingabe zur Ausgabe zu kommen. Des Weiteren wird gezeigt, warum das FFNN einen nicht-linearen Funktionsanteil beinhalten muss, um bestimmte Probleme zu lösen. Es wird nicht gezeigt, wie das Netzwerk die Gewichte und Biase (die Parameter Θ) der einzelnen Neuronen erlernt, die folgenden Kapitel behandeln die dazu nötigen Techniken. Hier wird eine funktionierende Lösung vorgegeben die auf dem XOR Beispiel aus dem „Deep Learning Book“ [5, S.171 ff.] beruht.

XOR ist ein logischer Operator, der zwei binäre Zahlen als Eingabe hat. Das Ergebnis der Operation ist eine binäre 1, wenn die Eingabe eine ungerade Anzahl an Einsen hat und 0 bei einer geraden Anzahl. Damit kommen vier Möglichkeiten in Frage $\{00, 01, 10, 11\}$ zu sehen in Abbildung 2.5 links. Darin ist außerdem zu sehen, dass XOR eine nicht-lineare Funktion ist, da keine Gerade die Einsen von den Nullen trennen kann. In Abbildung 2.6 ist die Architektur des FFNN zu sehen, mit der die XOR-Funktion gelöst wird. Über die Neuronen in der Eingabeschicht werden die beiden binären Zahlen dem Netz übergeben. Diese Neuronen sind mit allen Neuronen der darauffolgenden verdeckten Schicht H verbunden, deren Ausgaben durch die Gewichtsmatrix W und den Biasvektor b_h beeinflusst sind. Die Ausgabe y wird wiederum aus den Ausgaben der verdeckten Schicht, dem Gewichtsvektor w und dem Bias b_y berechnet. Um es kurz zu fassen ist $y = f(h(x; W, b_h); w, b_y)$.

Die folgende Konfiguration des FFNN kann die XOR-Funktion lösen. Seien

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{2.1}$$

$$b_h = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \tag{2.2}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \tag{2.3}$$

und $b_y = 0$.

Im folgenden werden die Berechnungen Schritt für Schritt durchgeführt, bis am Ende zu sehen ist, dass das FFNN die XOR-Funktion erfolgreich implementiert. Die Eingabe findet

über X statt, in der die vier Möglichen Eingaben stehen:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{2.4}$$

Als erstes wird die Eingabe X mit der Gewichtsmatrix der ersten Schicht W multipliziert:

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \tag{2.5}$$

Dazu wird der Bias b_h addiert:

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{2.6}$$

Das ist das Ergebnis der verdeckten Schicht H, wie in der Mitte von Abbildung 2.5 dargestellt. Dort zu sehen ist außerdem, dass auch hier keine Trennung der Nullen und Einsen durch eine Gerade erfolgen kann, da sich alle Werte auf einer Geraden befinden. Auch mit einer anderen Konfiguration des FFNN lässt sich daran nichts ändern. Wenn die verdeckte Schicht jedoch mit einer nicht-linearen Aktivierungsfunktion ausgestattet ist, wie z.B. einer ReLU (zu sehen bereits in Abb. 2.4 rechts), die folgendermaßen definiert ist: $f(x) = max(0, x)$ verändert sich die Ausgabe von der verdeckten Schicht leicht:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}, \tag{2.7}$$

auch zu sehen in Abb. 2.5 rechts. Durch diese Transformation können die Nullen und Einsen linear getrennt werden. Zum Schluss wird das Ergebnis aus 2.7 noch mit w multipliziert,

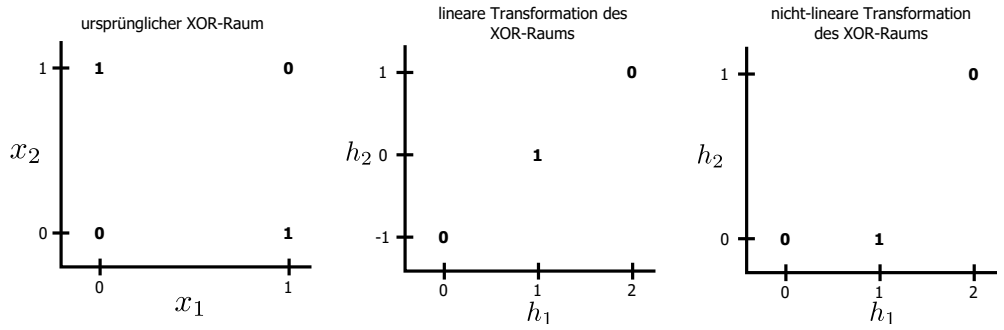

Abbildung 2.5: *(Links)* Der ursprüngliche XOR-Raum mit seinen vier möglichen Eingaben und den dazu erwarteten Ergebnissen. *(Mitte)* Hier zu sehen die Ausgabe der verdeckten Schicht bei einer linearen Transformation des originalen XOR-Raumes. Die Nullen und Einsen lassen sich hier nicht linear trennen, was bedeutet, dass das FFNN die XOR-Funktion so nicht korrekt lernen kann. *(Rechts)* Eine nicht-lineare Transformation des XOR-Raumes in der verdeckten Schicht ergibt die Möglichkeit, die Ergebnisse linear zu trennen. Dadurch kann die XOR-Funktion vollständig in der Ausgabe vom FFNN umgesetzt werden.

sowie der Bias b_y addiert:

$$Y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{2.8}$$

2.2.2 Aktivierungsfunktionen

Die nicht-lineare Sigmoid Aktivierungsfunktion $f(x) = \frac{1}{1+e^{-x}}$ reduziert die Eingabe auf den Wertebereich $(0, 1)$. Sie wurde in der Vergangenheit viel verwendet, inzwischen jedoch seltener, da sie zwei ungünstige Eigenschaften hat. Das eine Problem ist, dass die Ausgabe eines Sigmoid-Neurons bei kleinen oder großen Eingaben sich an 0 bzw. 1 annähert und dadurch der Gradient, der zum Lernen der Gewichte benötigt wird, gegen 0 geht (es kommt zur „Sättigung“ des Neurons). Dies hat zur Folge, dass die Fehlerrückführung durch Backpropagation in diesem Neuron zum Erliegen kommt und die davorliegenden Neuronen, einfach ausgedrückt, ihre eigenen Fehler nicht mehr zu spüren bekommen und daher keinen Lerneffekt erwarten können (hier wird etwas vorgegriffen, mehr dazu in Kapitel 2.2.5). Dadurch kann es passieren, dass die vorderen Schichten des KNN nicht oder sehr langsam lernen. Das zweite vergleichsweise kleine Problem ist, dass die Sigmoid-Funktion nicht symmetrisch um den Ursprung ist. Daraus folgt, dass das Lernen der optimalen Parameter

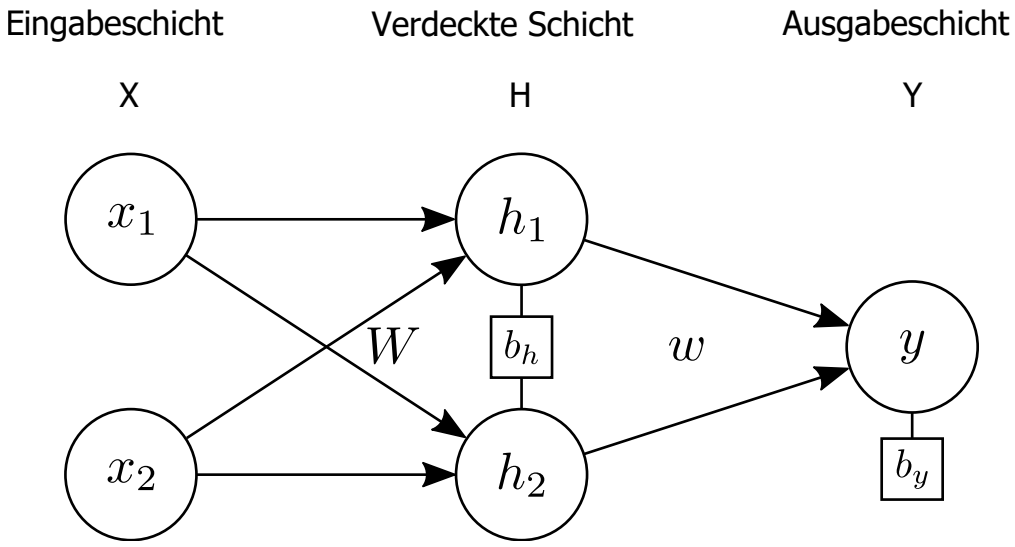

Abbildung 2.6: Architektur des FFNN zur Lösung der XOR-Funktion

(Gewichte) länger dauert[6] [7].

Die tanh-Funktion ist eine erweiterte Sigmoid-Funktion. Sie umfasst den Wertebereich $(-1, 1)$ und ist dadurch symmetrisch um den Ursprung. Somit ist sie der Sigmoid-Funktion immer vorzuziehen. Das „Sättigungs“-Problem bleibt jedoch bestehen.

Die ReLU Funktion hat die anderen beiden Funktionen in den letzten Jahren größtenteils abgelöst. Sie besteht aus zwei linearen Teilen $f(x) = \begin{cases} 0 & für \quad x < 0 \\ x & für \quad x \geq 0 \end{cases}$. Der Hauptgrund, der für die Verwendung von ReLU Neuronen spricht, ist, dass sie deutlich schneller als Sigmoid/tanh-Neuronen das Optimum beim Lernen finden, da ReLUs einer linearen Funktion sehr ähnlich ist (siehe Problem des kleinen Gradienten von Sigmoid-Neuronen nahe 0 und 1) [5, S.189]. Ein Nachteil von ReLUs ist, dass wenn sie einen negativen Aktivierungswert haben der Gradient null beträgt und in dem Fall nichts lernt. Falls dies durch einen großen negativen Gradienten aufgetreten ist und dadurch dieses Neuron keinen positiven Aktivierungswert mehr erreichen kann, ist das Neuron „abgestorben“. Diesem Problem kann vorgebeugt werden, indem kleine Lernraten verwendet werden. Es gibt einige Abwandlungen der ReLU, die das Absterben von Neuronen verhindern. Die „leaky ReLU“ setzt die Steigung für Werte unter Null auf 0.01, wodurch fast immer ein Gradient zustande kommt. „Absolute value

[6] Da die Begründung das Verständnis des Backpropagation Algorithmus voraussetzt, wird darauf im Text verzichtet. Begründung: Da die Sigmoid-Funktion nur positive Ausgaben hat, kommen bei der Rückpropagierung für die Gewichte eines Neurons ausschließlich positive oder negative Gradienten zustande (positiv oder negativ basierend auf dem rückpropagierten Fehler der das Neuron erreicht). Dieses Verhalten ist ungünstig, wenn der optimale Gradientenabstieg aus einer Kombination von positiven und negativen Änderungen an verschiedenen Gewichten besteht, da dadurch die Annäherung an das Optimum verlangsamt wird.

rectification“ $f(x) = |x|$ ist eine Variante die sich zur Objekt Detektion in Bildern eignet. Damit erkennen die Neuronen Merkmale unabhängig von der Helligkeit der Objekte [5, S.189].

2.2.3 Kostenfunktionen

Kostenfunktionen sind für den Lernprozess von künstlichen neuronalen Netzen von großer Bedeutung. Sie dienen zum einen als Maß der Güte eines Modells und zum anderen bilden sie eine Fehlerlandschaft, auf dessen Basis das Modell optimiert wird (s. SGD 2.2.4). Je kleiner der Fehler ist, desto besser klassifiziert bzw. schätzt das Modell die Trainingsdaten ein. Beim überwachten Lernen ist das Ergebnis einer Kostenfunktion der Mittelwert der Distanz zwischen den tatsächlichen Beobachtungen in den Trainingsdaten (das Label y_i passend zum Ereignis x_i) und den durch das Modell berechneten Vorhersagen ($\hat{y}_i = f(x_i|\Theta)$), sowie möglichen Regulierungsausdrücken (s. Regularization 2.3). Dabei kann die Distanz auf unterschiedliche Weisen berechnet werden.

Eine Variante ist der Mean Squared Error (MSE), häufig bei der linearen Regression benutzt:

$$J_{MSE}(\Theta) = \frac{1}{N} \sum_{i=1}^{N} (f(x_i|\Theta) - y_i)^2 \tag{2.9}$$

In Abbildung 2.7 wird die MSE Kostenfunktion beispielhaft dargestellt, um eine bessere Vorstellung einer Fehlerlandschaft und der Berechnung des Fehlers zu bekommen. Während der MSE hauptsächlich zur Regression verwendet wird, kann er auch zur Klassifikation verwendet werden. In der Praxis ist der Cross Entropy Loss vorzuziehen, da der MSE im Training langsamer das Optimum findet oder frühzeitig aufgrund von kleinen Gradienten bereits in der Ausgabeschicht stagniert.[8]

Der Cross Entropy Loss (auch „negative log. likelihood“) berechnet den Fehler anhand von Klassenwahrscheinlichkeiten gegeben durch das Modell. Er ist definiert als:

$$J_{CE}(\Theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} \, ln \, p_{i,j} \tag{2.10}$$

Dabei ist N die Anzahl der Trainingssamples, M die Anzahl der Klassen, $y_{i,j}$ ist ein binärer Wert der 1 annimmt, wenn j der echten Klasse (true label) y vom Sample i entspricht und $p_{i,j}$ ist die vom Modell berechnete Wahrscheinlichkeit der Klasse j für das Trainingsbeispiel i. Somit berechnet sich der Fehler ausschliesslich auf der durch das Modell berechneten Wahrscheinlichkeit der echten Klasse $-ln(p_{i,j=truelabel})$ (siehe Beispiel in Abb. 2.8). Die

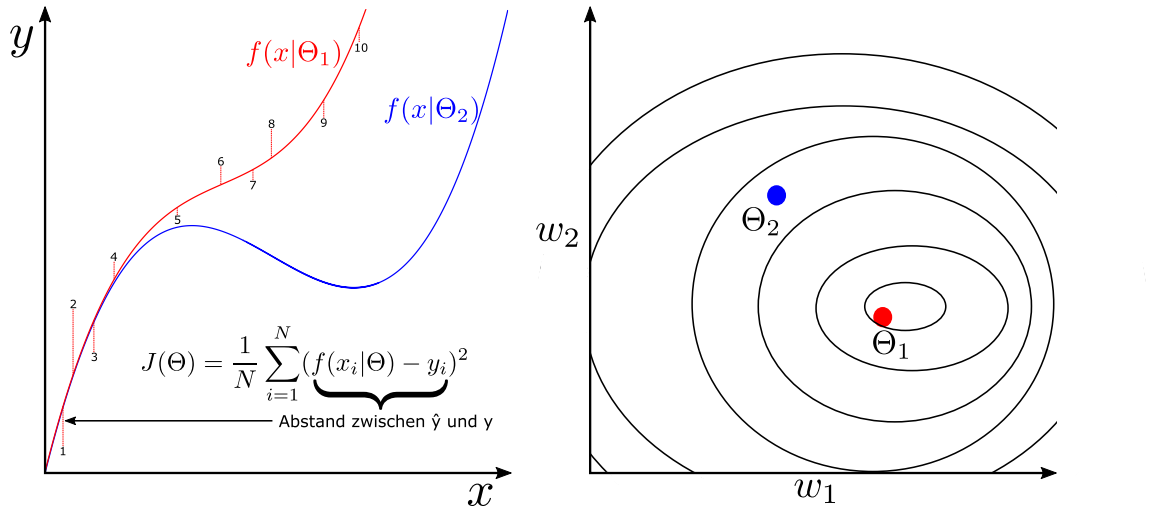

Abbildung 2.7: *(Links)* Zu sehen ist eine Datenmenge mit zehn Beispielen mit jeweils nur einem Merkmal (x) und zweimal die Funktion $f(x|\Theta)$ mit unterschiedlichen Parametern, die die Datenmenge approximieren soll. Die rote Funktion mit $\Theta_1$ist dabei deutlich näher am Ziel, als die blaue Funktion mit Θ_2. Der MSE berechnet sich dabei auf dem Abstand zwischen der Beobachtung y und der Vorhersage durch $\hat{y} = f(x|\Theta)$. *(Rechts)* Eine Fehlerlandschaft aufgespannt durch zwei Parameter w_1 und w_2. Die inneren Kreise deuten auf das globale Minimum der Kostenfunktion hin.

Wahrscheinlichkeiten $p_{i,j}$ werden durch diesen Ausdruck berechnet:

$$P(y_i|x_i; W, b) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \tag{2.11}$$

Dabei wird auf die Ausgabe des künstlichen neuronalen Netzes ($Wx_i + b$ = ein Vektor mit j Elementen, der die „class scores“ beinhaltet) die Exponentialfunktion angewendet und normalisiert ($\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$). Die Elemente des Ergebnisvektors werden häufig als Wahrscheinlichkeiten missverstanden, sind jedoch eher Werte der Zuversicht des Modells, wie sicher es sich ist, dass dieses Trainingssample Klasse j angehört. Das Ergebnis wird durch die Stärke des Regulierungsausdrucks beeinflusst (s. Kapitel 2.3).[7]. Der normalisierende Teil wird auch Softmax Funktion genannt. Im folgenden soll ein Rechenbeispiel diese Funktionen verdeutlichen. Die Berechnung der Klassenwahrscheinlichkeiten $\hat{p_0}$, $\hat{p_1}$ und $\hat{p_2}$ des ersten Trainingssamples aus Tabelle 2.1 wird in Abb. 2.8 vorgenommen. Der Gesamtfehler

x_i	m_1	m_2	y	$\hat{p_0}$	$\hat{p_1}$	$\hat{p_2}$	$\hat{y}$
1	4	8	1	0.333	0.463	0.204	1
2	0	15	0	0.4	0.381	0.219	0
3	-2	-9	2	0.226	0.309	0.465	2
4	6	-4	2	0.251	0.501	0.248	1

Tabelle 2.1: Vier Trainingsbeispiele mit den Merkmalen m_1 und m_2 und der Beobachteten Klasse y. In Grün die berechneten Klassenwahrscheinlichkeiten und daraus resultierenden Vorhersagen des Modells (siehe Gewichtsmatrix W und Bias Vektor b in Abb. 2.8).

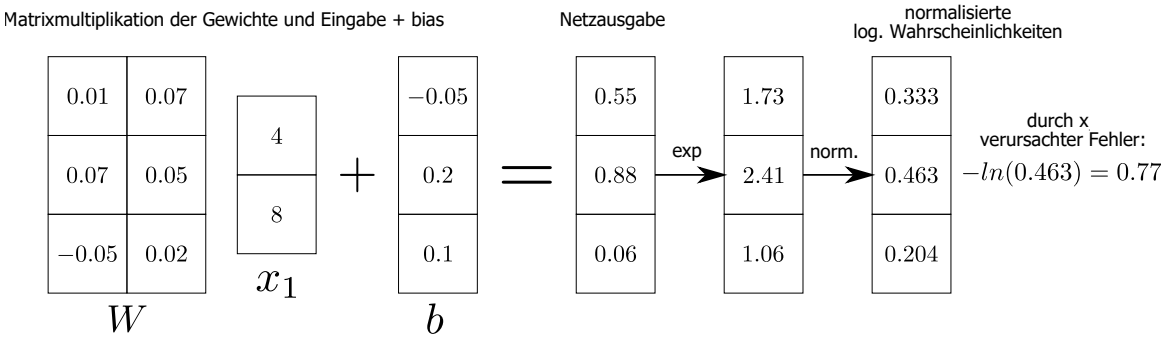

Abbildung 2.8: Die Berechnung der Klassenwahrscheinlichkeiten und des Cross Entropy Fehlers für x_1 (aus Tabelle 2.1)

$J_{CE}(\Theta)$ dieses Beispiels ergibt sich dann:

$$\begin{aligned} J_{CE}(\Theta) = &-\frac{1}{4}((0 \times ln(0.333) + 1 \times ln(0.463) + 0 \times ln(0.204)) \\ &+ (1 \times ln(0.4) + 0 \times ln(0.381) + 0 \times ln(0.219)) \\ &+ (0 \times ln(0.226) + 0 \times ln(0.309) + 1 \times ln(0.465)) \\ &+ (0 \times ln(0.251) + 0 \times ln(0.501) + 1 \times ln(0.248))) \\ = &-\frac{1}{4}(ln(0.463) + ln(0.4) + ln(0.465) + ln(0.248)) \\ = &\, 0.9616 \end{aligned} \tag{2.12}$$

Obwohl nur x_4 falsch klassifiziert wurde ist der Gesamtfehler des Modells relativ hoch, da sich das Modell auch bei den korrekt klassifizierten Beispielen nur mit geringer Sicherheit entscheiden konnte.

2.2.4 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) ist ein Optimierungsalgorithmus. Beim Maschinellen Lernen soll der Fehler einer Kostenfunktion reduziert werden. Dazu wird der Gradient der

Funktion an der durch die aktuellen Funktionsparameter angegebenen Stelle benötigt. Der Gradient ist die Steigung an dieser Stelle. Auf dem Gradienten basierend werden dann Schritt für Schritt die Funktionsparameter angepasst, um ein Optimum zu finden. SGD ist der meist verwendete Algorithmus zum Erlernen von Parametern bei Deep Learning Aufgaben [5, S.290]. Deep Learning profitiert von großen Modellen und möglichst vielen Daten, um die besten Parameter zu finden. Die meisten Algorithmen skalieren dabei jedoch nicht besonders gut im Vergleich zu SGD. Der Algorithmus Batch Gradient Descent (BGD), von dem SGD eine abgeänderte Form ist, berechnet den Gradienten auf allen im Trainingsdatensatz vorhandenen Samples und führt erst dann ein Update auf den Parametern der Kostenfunktion durch:

$$\Theta = \Theta - \alpha \nabla_{\Theta} J(\Theta) \tag{2.13}$$

Sehr große Datensätze passen jedoch häufig nicht in den Speicher, wodurch dieser Algorithmus nicht praktikabel für Deep Learning Anwendungen ist. BGD findet ein Optimum, indem er viele Male den Gradienten des kompletten Datensatzes berechnet, wodurch er sehr rechenlastig ist. SGD verwendet hingegen nur einzelne Trainingssamples oder Minibatches[7], um den Gradienten zu berechnen:

$$\Theta = \Theta - \alpha \nabla_{\Theta} J(\Theta; x_i, y_i) \tag{2.14}$$

Dies führt zu häufigeren Updates auf den Parametern. Auch SGD nähert sich dem globalen Optimum an, wenn die zu optimierende Funktion Konvex ist. Bei einem ausreichend großen Datensatz kann dazu ein einziger Durchlauf reichen und hat dadurch einen vergleichsweise kleinen Rechenaufwand gegenüber BGD. Abbildung 2.9 zeigt den Unterschied im Verlauf von SGD und BGD. Die Lernrate α ist mit SGD vergleichsweise klein zu wählen, da die Varianz der Parameterupdates deutlich größer ist als beim BGD. Außerdem ist es üblich, die Lernrate mit der Zeit zu verkleinern, sodass tatsächlich ein Optimum erreicht wird statt nur drum herum zu springen. Eine weitere Voraussetzung ist, dass die Trainingssamples in zufälliger Reihenfolge vorliegen. Sind sie in irgendeiner Weise sortiert, kann das den Lernprozess verlangsamen.

2.2.4.1 Momentum

Stochastic Gradient Descent mit Momentum kann die Fehleroptimierung beschleunigen. Dabei wirken die Gradienten der vorherigen Durchläufe auf die nächsten Parameterupdates

[7] Minibatches sind Mengen von Traingssamples, die verwendet werden, um möglichst immer Richtung Optimum führende Parameterupdates zu bekommen.

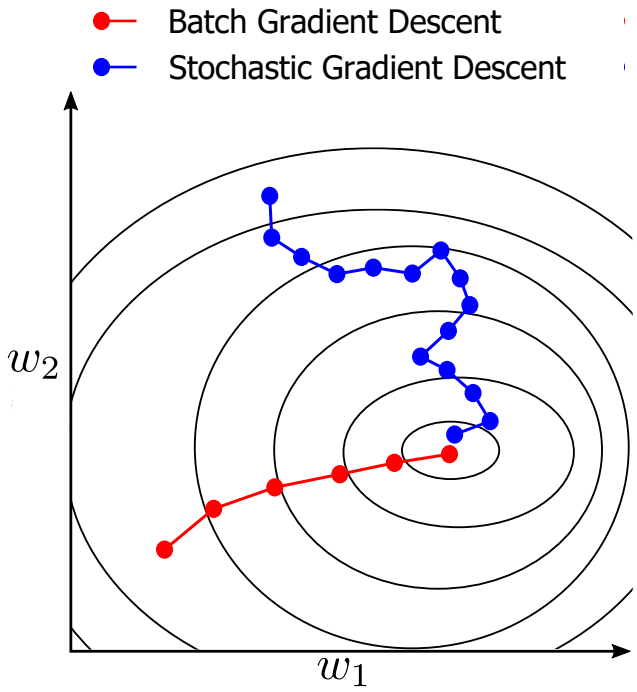

Abbildung 2.9: Beispielhafter Verlauf von Batch Gradient Descent und Stochastic Gradient Descent

nach. Das Parameterupdate ergibt sich durch den Gradienten der aktuellen Iteration, summiert mit einem Teil des Gradienten der vorherigen Iteration. Mit γ lässt sich die Verfallsrate der vorangegangenen Gradienten bestimmen.

$$v = \gamma v + \alpha \nabla_\Theta J(\Theta; x_i, y_i) \tag{2.15}$$

$$\Theta = \Theta - v \tag{2.16}$$

Momentum kann den Lernprozess beschleunigen, wenn die Parameterupdates verrauscht sind, bei kleinen Gradientenschritten oder wenn die Fehlerlandschaft eine Schlucht darstellt und der Gradient hauptsächlich zwischen den beiden Hängen hin- und herspringt.

2.2.5 Backpropagation

Im vohergehenden Teil 2.2.4 war die Rede vom Gradienten, anhand dessen die Kostenfunktion optimiert werden soll, aber nicht wie der Gradient berechnet wird. Dabei lässt sich der Ausdruck des Gradienten, wie in Abbildung 2.10 zu sehen, leicht bestimmen, jedoch ist die Berechnung desselben für jeden Parameter einzeln nicht praktikabel. Der Backpropagation

Algorithmus berechnet auf effiziente Weise den Gradienten für alle Parameter $\frac{\partial J}{\partial W_{ij}^{(l)}}$ und $\frac{\partial J}{\partial b_i^{(l)}}$. Dies geschieht in Bezug auf ein oder mehrere Trainingssamples oder die komplette Kostenfunktion $J(W, b)$:

$$\frac{\partial}{\partial W_{ji}^{(l)}} J(W,b) = \frac{1}{m} \sum_{n=1}^{m} \frac{\partial}{\partial W_{ji}^{(l)}} J(W,b;x^{(n)},y^{(n)}) \tag{2.17}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m} \sum_{n=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(n)},y^{(n)}) \tag{2.18}$$

Ausgangspunkt des Algorithmus ist, dass die Trainingssamples durch das künstliche neuronale Netz vorwärts durchpropagiert worden sind, um die Neuronenaktivierungen $a_i^{(l)}$ und den Gesamtfehler $J(W, b; x, y)$ zu bestimmen. Daraufhin wird für jedes Neuron ein Fehlerausdruck $\delta_i^{(l)}$ bestimmt, der berechnet, wie stark das Neuron für den Gesamtfehler mitverantwortlich ist. Anhand der Aktivierungen $o_i^{(l)}$ und des Fehlerausdrucks $\delta_i^{(l)}$ lassen sich dann die partiellen Ableitungen für die Gewichte und Biase berechnen. Dies geschieht rückwärts, von der Ausgabeschicht hin zur Eingabeschicht, da die partiellen Ableitungen der Parameter von Schicht l ($\frac{\partial J}{\partial \theta^{(l)}}$) abhängig von den partiellen Ableitungen der darauffolgenden Schicht $l+1$ sind. Im Folgenden der Backpropagation Algorithmus mit seinen vier Formeln, wobei

n_l für die Ausgabeschicht steht,

$z_i^{(l)}$ die Eingabe des Neurons i in Schicht l mit $z_i^{(l)} = \sum_{i \epsilon I} x_i w_i + b$ ist,

f die Aktivierungsfunktion von Neuron i ist und

S_l die Anzahl der Neuronen in Schicht l ist.

1. Vorwärtspropagierung der Trainingssamples zur Bestimmung der Neuronenaktivierungen $a_i^{(l)}$

2. Die Berechnung des Fehlerausdrucks der Neuronen der Ausgabeschicht ist direkt das Produkt der partiellen Ableitung der Aktivierung und der Ableitung der Aktivierungsfunktion der Ausgabeschicht.

$$\delta_i^{(n_l)} = \frac{\partial J}{\partial a_i^{(n_l)}} \times f'(z_i^{(n_l)}) \tag{2.19}$$

3. Der Fehlerausdruck für die Neuronen der verdeckten Schichten $l = n_l - 1, n_l - 2, ..., 2$ ist das Produkt der Gewichte der dahinterliegenden Schicht $l+1$ und deren Fehlerausdrücke,

und wiederum der Ableitung der Aktivierungsfunktion von Schicht l.

$$\delta_i^{(l)} = (\sum_{j=1}^{S_{l+1}} W_{ji}^{(l+1)} \delta_j^{(l+1)}) f'(z_i^{(l)}) \tag{2.20}$$

4. Mit dem Fehlerausdruck $\delta_j^{(l)}$ und der Neuronenaktivierung $a_i^{(l-1)}$ lässt sich dann der Gradient $\frac{\partial J}{\partial w_{ji}^{(l)}}$ als Produkt berechnen.

$$\frac{\partial J}{\partial w_{ji}^{(l)}} = a_i^{(l-1)} \delta_j^{(l)} \tag{2.21}$$

5. Der Gradient für Biase entspricht direkt dem Fehlerausdruck.

$$\frac{\partial J}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{2.22}$$

Dadurch, dass der Fehlerausdruck von Schicht l sich unter anderem aus dem Fehlerausdruck der dahinter liegenden Schicht $l+1$ ergibt, wird der Fehler durch das Netzwerk zurückpropagiert. Dadurch können die Ergebnisse der sich wiederholenden Teilausdrücke zur Berechnung der vorderen Schichten, wie im Beispiel in Abbildung 2.10 zu sehen, wiederverwendet werden. Im Beispiel sind die sich wiederholenden Teilausdrücke farblich (rot, blau und grün) gekennzeichnet. Diese ständig neu zu berechnen, würde den Rechenaufwand mit der Größe des Netzwerks exponentiell wachsen lassen.

2.3 Regulierungsmethoden

Regulierungsmethoden haben zum Ziel, eine Überanpassung (engl. overfitting) des Modells an die Trainingsdaten zu vermeiden. Damit wird verhindert, dass das Modell die Trainingsdaten auswendig lernt und schlechte Ergebnisse auf ungesehenen Daten erzielt. Mit diesen Methoden lässt sich die Kapazität eines Modells regulieren. Hat ein Modell eine zu große oder kleine Kapazität für das Problem, das es lösen soll, so fällt es der Überanpassung bzw. Unteranpassung (engl. underfitting) zum Opfer. Im Folgenden werden einige der am häufigsten verwendeten Regulierungsmethoden vorgestellt.

2.3.1 Weight Decay

Weight Decay verkleinert die Gewichte bei jedem Gradient Descent Schritt. Dabei wirkt es sich auf sehr große Gewichte eher aus als auf kleine. Dadurch verteilt sich die Entschei-

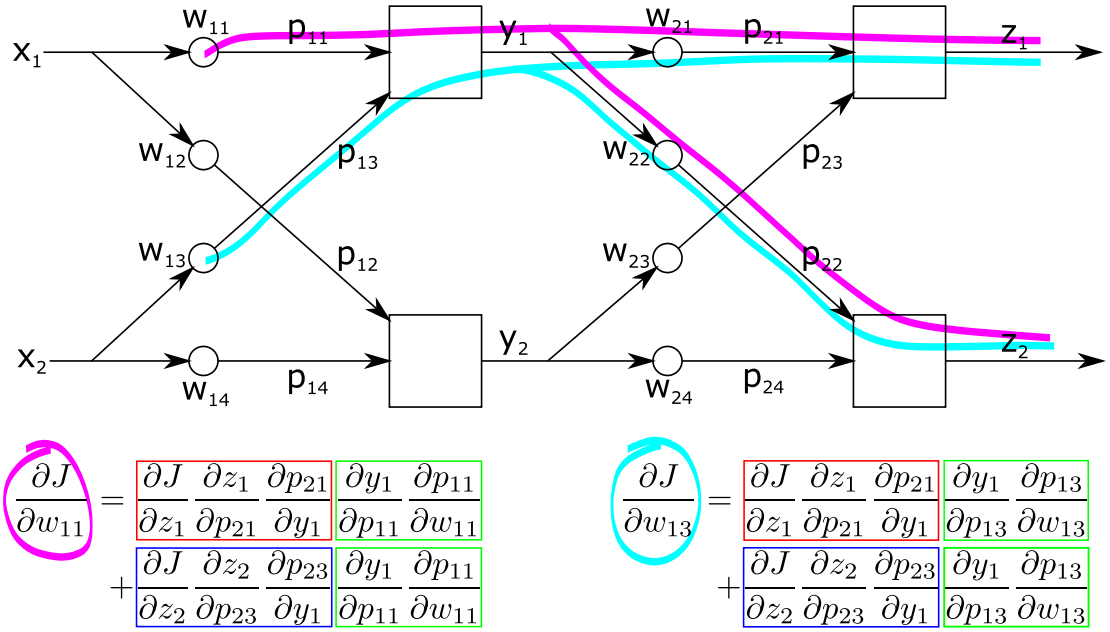

Abbildung 2.10: Beispiel der Abhängigkeiten bei der Gradientenberechnung von zwei Gewichten (Türkis und Magenta) und sich dabei wiederholende Teilausdrücke (Rot, Blau und Grün gekennzeichnet).

dungskraft eines jeden Neurons besser auf das gesamte Netz. Im Endeffekt bewirkt es, dass die resultierende Funktion abgeflacht wird, wodurch sich das lernende Modell nicht jeden Ausreißer einprägen kann. Der Weight Decay Ausdruck $\frac{1}{2}\lambda w^2$ wird zur Kostenfunktion hinzuaddiert, wodurch

$$\tilde{J}(\theta; x, y) = J(\theta; x, y) + \frac{1}{2}\lambda w^2$$

die regulierte Kostenfunktion ist. λ ist dabei ein konstanter Faktor, der die Stärke des Regulierungsausdrucks festlegt. Bei einem Gradientenschritt wird er mit der Kostenfunktion nach w abgeleitet, sodass ein Update auf den Gewichten folgendes ergibt und klar wird, dass ein großes Gewicht stärker bestraft wird als kleine Gewichte.

$$w = w - \alpha(\lambda w + \frac{\partial J}{\partial w}) \tag{2.23}$$

2.3.2 Early-Stopping

Early-Stopping reduziert die Kapazität eines Modells, indem es die Anzahl der erlaubten Durchläufe des Lernalgorithmus beschränkt, wodurch die Entfernung, die die Gewichte von ihrem ursprünglichen Initialisierungspunkt erreichen können, beschränkt wird. Während des Trainings wird immer wieder der Fehler des Modells auf dem Validationsdatensatz berechnet,

der nicht dem Trainingsdatensatz oder Testdatensatz entsprechen darf und auch keine Schnittmenge dieser bildet. Der berechnete Fehler ist der zu erwartende Testfehler des Modells. Early-Stopping bricht das Training ab, wenn der Validationsfehler keine Verbesserungen (wobei ein Faktor reguliert, wie groß die Verbesserung sein muss um als solche zu zählen) über die festgelegte Anzahl an Durchläufen erfährt. Zu diesem Zeitpunkt hat das Modell höchstwahrscheinlich (sofern die Abbruchkriterien gut gewählt worden sind) angefangen die Trainingsdaten auswendig zu lernen und der Validationsfehler wird bereits größer. Nun ist der Durchlauf, nach dem abgebrochen wurde, nicht das beste Modell, daher werden die Parameter immer zwischengespeichert, wenn der Validationsfehler sich verbessert. Am Ende des Trainingsprozesses kann dann auf das beste gelernte Modell zurückgegriffen werden.

2.3.3 Dropout

Dropout lässt Neuronen mit einer bestimmten Wahrscheinlichkeit absterben. Bei der Vorwärtspropagierung der Eingaben werden die Aktivierungen der „toten“ Neuronen mit null multipliziert, sodass sie keinen Einfluss auf die ihnen folgenden Aktivierungen mehr haben. Dadurch erhalten sie auch keine Parameter Updates, da sie nicht zum Fehler der Kostenfunktion beitragen. Dropout verteilt die Spezialisierung von einem Neuron auf die umliegenden Neuronen. Zum Beispiel wenn ein Neuron für die Erkennung eines bestimmten Merkmals zuständig, aber nicht immer verfügbar ist, lernen die umliegenden Neuronen mit dem Ausfall umzugehen. Dies führt zu einer besseren Generalisierung, da das Modell weniger abhängig von einzelnen Parametern wird. Ein Nachteil ist, dass Dropout die benötigte Trainingszeit um das zwei- bis dreifache verlängert. Das kommt daher, dass die Parameterupdates vergleichsweise verrauscht sind, da die Kostenfunktion durch die wechselnden Parameter in jedem Durchgang einer anderen entspricht.[9]

2.4 Convolutional Neural Nets

Convolutional Neural Networks (zu Deutsch: faltendes neuronales Netzwerk) wurden entworfen um in Bildern die Korrelation der beieinanderliegenden Pixel auszunutzen. Auch bei Audiodaten ergeben die Samples nur im Verbund ein aussagekräftiges Geräusch, weshalb Anwendungen auch hier von CNNs profitieren. So sind CNNs für den Einsatz in Bildklassifikations- und Sprachanwendungen besonders geeignet bzw. in allen Anwendungsgebieten, in denen die Merkmalsspalten in der Reihenfolge eine Bedeutung haben und sie nicht einfach ausgetauscht werden können. Ein CNN besteht aus Convolutional Layern, gefolgt von

fully-connected Layern, wie sie in FFNN vorkommen. Convolutional Layer unterscheiden sich zu fully-connected Layern in den folgenden Punkten. Sie bekommen statt einem Vektor, wie es bei fully-connected Layern der Fall ist, als Eingabe Matrizen bzw. Tensoren. Ein weiterer Unterschied liegt darin, wie die Neuronen vernetzt sind. Zum einen sind die Neuronen zwischen zwei Schichten nicht alle miteinander verbunden („Lokale Konnektivität“) und zum anderen werden Gewichte von mehreren Neuronen gemeinsam benutzt („Geteilte Gewichte“). Dadurch reduziert sich die Menge der Parameter deutlich im Vergleich zu fully-connected Layern.

Im Folgenden werden die Bestandteile des Convolution Layers beschrieben, wie die Eingaben aussehen, wie sie umgeformt werden und auf welche Hyperparameter es dabei ankommt. Daraufhin wird die Convolution Operation an einem Beispiel durchgeführt und zum Schluss die Max-Pooling Operation und deren Effekte erläutert.

2.4.1 Convolution Layer

Ein Convolution Layer besteht aus einer Menge von Filtern. Diese Filter lernen im Lernprozess dazu trainiert bestimmte Merkmale zu erkennen. Die Anzahl der Filter ist gleich der Tiefe eines Convolution Layers (s. Abb. 2.11) und kann als Hyperparameter der jeweiligen Schicht gewählt werden. Die Größe der Filter eines Convolution Layers wird durch den Hyperparameter „rezeptives Feld“ festgelegt. Bei einer rezeptiven Feldgröße von 3 bekommt ein Neuron des Filters Reize von einer 3x3 Fläche der Eingabe ab, geht dabei aber auch durch die komplette Tiefe der Eingabe. Ist die Eingabe z.B. ein Farbbild, so ist die Tiefe 3 für die Farbchannel. Damit haben alle Neuronen dieses Convolution Layers 3x3x3 Gewichte, wobei die Neuronen eines bestimmten Filters alle dieselben Gewichte haben („geteilte Gewichte“). Der Sinn der geteilten Gewichte innerhalb eines Filters liegt darin, dass wenn ein Filter dazu trainiert wurde, ein bestimmtes Merkmal zu erkennen, so soll er dieses Merkmal nicht nur an einer Stelle in der Eingabe erkennen können, sondern in der kompletten Eingabe. Hat ein Neuron n, mit den Gewichten eines Filters f, welcher trainiert wurde, um ein Merkmal m zu erkennen, eine starke Aktivierung, so ist m mit großer Übereinstimmung innerhalb des rezeptiven Feldes von n zu sehen. Jedes Neuron von f wird an einer anderen Stelle der Eingabe angewendet, wodurch die Ausgabe des Filters eine Matrix ist, in der die Werte beschreiben, wie stark m an allen Stellen der Eingabe zu sehen ist und wird daher auch Merkmalskarte (Feature Map) genannt. Durch die Anwendung des Filters auf allen Stellen der Eingabe spricht man von einer Faltung, daher der Name „faltendes neuronales Netzwerk“.

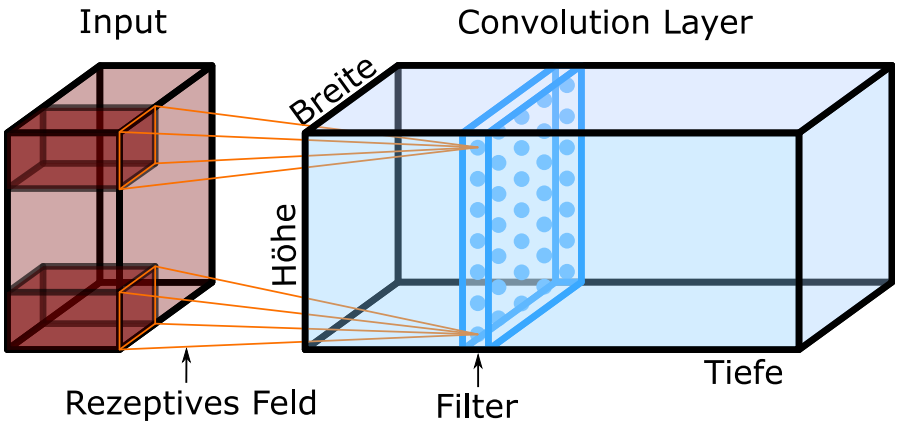

Abbildung 2.11: Darstellung eines Convolution Layers. Die Tiefe der Schicht wird mit der Anzahl der Filter festgelegt. Die Neuronen des Filters bekommen Eingaben der davorliegenden Schicht von einer festen Stelle (Höhe,Breite) aber durch die komplette Tiefe.

Die Dimensionen der Ausgabe des Convolution Layers sind abhängig von den Hyperparametern Zero-padding, Stride (Schrittgröße) und der Anzahl der Filter. Die Tiefe wird, wie schon gesagt, beeinflusst von der Anzahl der Filter. Zero-padding und Stride beeinflussen die Höhe und Breite der Schicht. Stride gibt an, mit welchem Abstand der Filter auf der Eingabe aneinander gelegt wird. Mit Stride 1 wird der Filter jedes mal um einen Pixel verschoben. Ein Stridewert größer als 1 hat den Effekt, dass die Daten herunter gesamplet werden, vergleichbar mit der Max-Pooling Operation (s. 2.4.3). Mit Zero-padding wird der Rand der Eingabe mit Nullen versehen, wodurch die Ausgabe in der Höhe und Breite kontrolliert werden kann. Ohne das schrumpft die Ausgabe mit jedem Convolution Layer, wodurch es schwierig wird, passende Werte für Stride und das rezeptive Feld zu finden. Mit der folgenden Formel lässt sich die Größe der Ausgabe berechnen, außerdem muss das Ergebnis einen ganzzahligen Wert ergeben, sonst ist die Konfiguration der Hyperparameter nicht zulässig.

$$(W - F + 2P)/S + 1 \tag{2.24}$$

Dabei ist W die Größe der Eingabe, F die Größe des rezeptiven Feldes, P ist die Anzahl der Zero-padding Reihen am Rand und S für den Stride. Wenn ein Bild zum Beispiel 100x100 Pixel groß ist, das rezeptive Feld $F = 3$, zero-padding $P = 1$ und stride $S = 1$, so ist die Ausgabe des Convolution Layers $(100 - 3 + 2 * 1)/1 + 1 = 100$. Ohne zero-padding schrumpft die Ausgabe auf$(100 - 3 + 2 * 0)/1 + 1 = 98$.

Im nächsten Unterabschnitt wird die Convolution Operation mit Zero-padding beispielhaft angewendet.

2.4.2 Convolution Operation

In diesem Beispiel ist die Eingabe x ein 5x5x2 Tensor. Der Convolution Layer hat zwei 3x3x2 Filter $w0$ und $w1$. Der Hyperparameter Stride ist $S = 2$ und Zero-padding $P = 1$, wodurch sich bereits die Größe der Ausgabe berechnen lässt mit $(5 - 3 + 2)/2 + 1 = 3$. Der Ausgabetensor o ist demnach 3x3 hoch und breit und die Tiefe ist durch die Anzahl der Filter gegeben (3x3x2). Die folgenden beiden Matrizen sind die Eingabe x mit Zero-padding:

$$x[:,:,0] = \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 2 & 0 \\ 0 & 0 & 1 & 2 & 2 & 2 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \tag{2.25}$$

$$x[:,:,1] = \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 & 2 & 1 & 0 \\ 0 & 2 & 2 & 3 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \tag{2.26}$$

Jeder Filter besteht aus einem Gewichtstensor, der so groß ist wie das rezeptive Feld des Convolution Layers und der Tiefe der Eingabe (3x3x2). Der erste Filter $w0$:

$$w0[:,:,0] = \begin{matrix} -1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{matrix} \tag{2.27}$$

$$w0[:,:,1] = \begin{matrix} -1 & 1 & -1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{matrix} \tag{2.28}$$

Der zweite Filter $w1$:

$$w1[:,:,0] = \begin{matrix} 0 & 0 & 0 \\ -1 & 1 & -1 \\ 1 & 1 & 1 \end{matrix} \tag{2.29}$$

$$w1[:,:,1] = \begin{matrix} 1 & 1 & -1 \\ 1 & 0 & 0 \\ 1 & -1 & 0 \end{matrix} \tag{2.30}$$

Die Ausgabe berechnet sich nun durch die Anwendung der Filter auf die Eingabe. Dabei wird jeweils $w0[:,:,0]$ und $w1[:,:,0]$ nur auf $x[:,:,0]$ angewendet, genauso $w0[:,:,1]$ und $w1[:,:,1]$ nur auf $x[:,:,1]$. Die Ausgabe des Filters $w0$ ist eine Matrix $o[:,:,0]$. Es ergibt sich die Ausgabe an der Stelle $o[0,0,0]$ dadurch, dass der Filter in der linken oberen Ecke aufgelegt wird und die Elemente der Ergebnismatrix des Hadamard Produkts addiert werden:

$$\begin{matrix} -1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{matrix} \circ \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix} = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{matrix} = -1 \tag{2.31}$$

$$\begin{matrix} -1 & 1 & -1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{matrix} \circ \begin{matrix} 0 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 3 \end{matrix} = \begin{matrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{matrix} = 5 \tag{2.32}$$

$$o[0,0,0] = -1 + 5 = 4 \tag{2.33}$$

Daraufhin wird der Filter um die Größe des Stride-hyperparameters nach rechts (bzw. nach unten falls der Filter bereits am rechten Rand liegt) geschoben (s. Abb. 2.12).

2.4.3 Max-Pooling

Die Max-Pooling Operation wird häufig nach einem Convolution Layer verwendet, um die Merkmalserkennung weniger abhängig von der Rotation, Neigung, Größe und dem Ort des Merkmals in der Eingabe zu machen. Ähnlich wie bei der Convolution Operation wird ein Fenster einer festen Größe über die Eingabe mit einer bestimmten Schrittgröße geschoben. Aus jedem Fenster wird der größte Wert beibehalten. Die Schrittgröße bestimmt, wie auch bei der Convolution Operation, die Ausgabegröße. Bei einer Schrittgröße von $S = 1$ würden sich die ergebenden Werte häufig wiederholen und wären damit redundant, daher ist es üblich $S > 1$, wie im Beispiel in Abbildung 2.13, zu wählen. Durch das Downsampling der Daten wird eine Verringerung des Rechenaufwands für die Folgeschichten und eine Vergrößerung

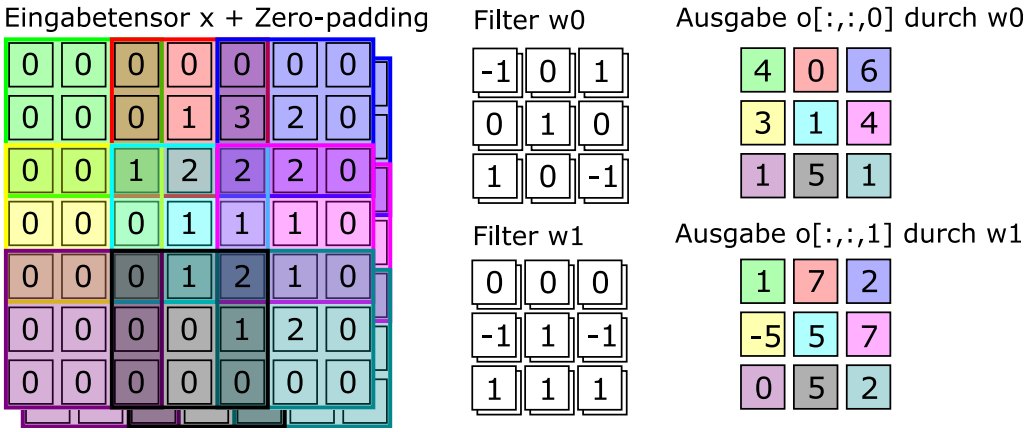

Abbildung 2.12: Veranschaulichung der Berechnung des Ausgabetensors. Die Zellen der Ausgabe sind farblich gleich gekennzeichnet mit dem Bereich der Eingabe, aus dem sie berechnet werden.

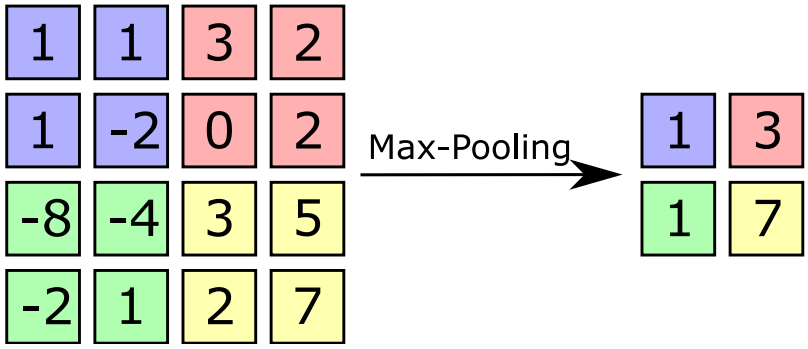

Abbildung 2.13: Max-Pooling mit Fenstergröße $F = 2$ und Schrittgröße $S = 2$

des rezeptiven Feldes der Neuronen der Folgeschichten in Bezug auf die Netzeingabe erreicht. Dies kann zu einer Bildung von spezielleren Merkmalen führen. Dieser Effekt ist ohne die Max-Pooling Operation weniger vorhanden.

3 Detektion von Objektklassen in Bildern

Die Detektion von Objektklassen in Bildern kann in das Forschungsfeld „Computer Vision“ eingeordnet werden. In diesem Bereich werden Methoden und Algorithmen entwickelt, um die Fähigkeiten der visuellen Wahrnehmung des Menschen in einem Computer zu verwirklichen. Diese Methoden werden zur Klassifikation von Bildern (z.B. Erkennung von Krebszellen, Erkennung von handgeschriebenen Zeichen, uvm.) verwendet. Auf der Klassifikation baut die Detektion von Objekten auf, indem nicht nur ein Bild einer Klasse aus einer vorher festgelegten Menge von Kategorien zugeordnet wird, sondern auch die Positionen der möglicherweise mehrfach vorhandenen Objekte im Bild bestimmt. Zum einen werden die Methoden immer weiterentwickelt, um die Präzision, mit der verschiedenste Objekte erkannt werden, zu verbessern, damit sie auch in Anwendungsgebieten eingesetzt werden können, die nur kleine Fehlermargen erlauben, aber auch, um die Verarbeitung eines Bildes zu beschleunigen, sodass in Echtzeit die Erkennung durchgeführt werden kann (z.B. Verarbeitung eines Videofeeds einer Kamera beim autonomen Fahren).

Da die Detektion von Objektklassen in Bildern auf der Klassifikation von Bildern aufbaut, wird zum Anfang dieses Kapitels auf die Probleme eingegangen, die der Computer bewältigen muss, um diese Aufgabe zu erfüllen. Daraufhin...

3.1 Klassifikation von Bildern

Bei der Klassifikation wird ein Bild einer Klasse aus einer vorher festgelegten Menge von Kategorien zugeordnet. Ein Modell wird dazu auf Beispielen der Kategorien erlernt, die es voneinander unterscheiden können soll. Das erlernte Modell ordnet dann jedem zu klassifizierenden Bild für alle Kategorien eine Wahrscheinlichkeit zu. Das Klassifikationsmodell muss dabei Probleme überwinden, die für einen Menschen trivial erscheinen, für den Computer jedoch keine leichte Aufgabe darstellen. Dazu gehören

- unterschiedliche Größen von Objekten, sei es die Größe im Bild oder der realen Größe,
- Verdeckung von Teilen des Objekts,

- Helligkeitsunterschiede,
- Variation des Blickpunktes auf das Objekt

und andere erhebliche Veränderungen der Pixelwerte, die jedoch keine Auswirkung auf die Klassifikation haben sollen [7]. Damit Modelle trotzdem Bilder korrekt klassifizieren können, müssen sie zum einen eine gewisse Kapazität aufweisen, um all das zu erlernen und zum anderen Daten zur Verfügung haben, die diese Eigenschaften bzw. Variationen aufweisen.

3.1.1 Trainingsprozess

Der Trainingsprozess eines Modells wird begleitet von drei Kenngrößen, die die Fähigkeit des Modells als Klassifikator auf verschiedenen Datensätzen angeben. Begonnen wird der Prozess mit der Eingabe einer Bildmenge, für die bereits bekannt ist, welcher Klasse jedes Bild angehört. Das ist die Trainingsmenge, auf der das Modell trainiert wird und auf der der Trainingsfehler berechnet wird. Während des Trainings wird das Modell auf dem Validationsdatensatz periodisch auf den Validationsfehler überprüft. Dieser Datensatz bildet keine Schnittmenge mit dem Trainingsdatensatz. Der Validationsfehler wird dabei beobachtet, um festzustellen, ob das Modell angefangen hat, die Trainingsdaten auswendig zu lernen. Dies ist der Fall, wenn der Validationsfehler anfängt größer zu werden. Zum Schluss wird das Modell auf einem weiteren Datensatz ausgewertet. Auch der Testdatensatz bildet keine Schnittmenge mit den anderen beiden. Das Maß für den Fehler auf dem Testdatensatz ist der Generalisierungsfehler. Zum Vergleich der Güte von verschiedenen Modellen kann auch die Präzision genommen werden, die aus den Vorhersagen des Modells auf den Testdaten und den Ground Truth Daten berechnet wird. Das Ziel des Trainings ist es immer, den Traingsfehler zu verringern und den Abstand zwischen Trainings- und Generalisierungsfehler möglichst klein zu bekommen.

3.2 Regionenbasierte Objektdetektion

Die Detektion baut auf der Klassifikation von Bildern auf, denn die Merkmale, die ein Klassifikator erlernt hat sind dieselben, die zur Bestimmung der Objektklasse(n) bei der Detektion benötigt werden. Im Folgenden werden aufeinander aufbauende Lösungen der Objektdetektion vorgestellt, die das Problem auf die Klassifizierung von sogenannten Region-Proposals reduzieren. Dabei wird für jedes Bild eine Menge von Regionen bestimmt, in denen möglicherweise ein Objekt zu sehen ist. Diese Regionen werden daraufhin klassifiziert. Den Anfang und die grundlegende Vorgehensweise macht R-CNN [10] im Jahr 2014. Darauf gefolgt

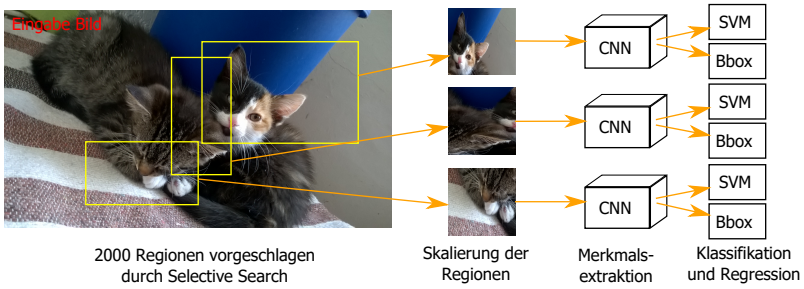

Abbildung 3.1: R-CNN Ablauf

von Fast R-CNN [11], mit dem einige Veränderungen einhergehen, um die Geschwindigkeit und Genauigkeit der Erkennung zu verbessern. Daraufhin wurde mit Faster R-CNN [12] die Erstellung der Region-Proposals umgestellt auf ein künstliches neuronales Netz, dem Region-Proposal-Netzwerk (RPN).

3.2.1 R-CNN

Die R-CNN (Regionen mit CNN Merkmalen) Vorgehensweise (in Abb. 3.1) beginnt damit, ein Bild nach möglichen Objekten mit dem Selective Search Algorithmus[13] zu untersuchen. Dieser schlägt 2000 Regionen vor, die wahrscheinlich ein Objekt beinhalten (Region of Interest (RoI)). Jeder Vorschlag wird auf eine feste Bildgröße skaliert und durch ein CNN und fully-connected Layer propagiert. Anhand des extrahierten Merkmalsvektors werden die Regionen mit klassenspezifischen SVMs (Support Vector Machine) klassifiziert und für den Fall, dass die Region nicht als Hintergrund klassifiziert wurde, wird die Lage des Objekts (Bounding Box) mit einem klassenspezifischen Regressor verfeinert. Herausgefiltert werden Regionen, die einen niedrigeren Klassifikationswert als andere Regionen erreicht haben und gleichzeitig mit diesen Regionen eine „Intersection over Union“ (IoU[1]) über einem bestimmten Schwellwert besitzen.

3.2.2 Fast R-CNN

Die Eingabe für Fast R-CNN ist wieder das Eingabebild, sowie eine Menge Regionen, vorgeschlagen vom Selective Search Algorithmus. Nun wird jedoch nur das komplette Eingabebild durch das CNN propagiert. Nach der letzten Convolution Schicht, wird jede Region durch eine RoI-Pooling Schicht auf eine feste Größe skaliert, da der darauf folgende Fully-connected Layer immer dieselbe Größe erwartet. Der darauf extrahierte Merkmalsvektor

[1] IoU ist die Prozentuale Abdeckung einer Fläche, durch eine andere Fläche.

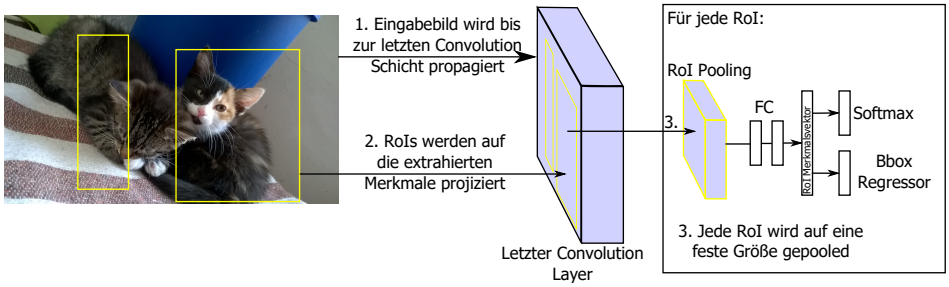

Abbildung 3.2: Fast R-CNN Ablauf (Bildidee aus [11])

wird dann durch eine Softmax Schicht klassifiziert und ein Regressor verfeinert die finale Position der Bounding Box.

3.2.2.1 RoI-Pooling

Die RoI-Pooling Operation formt die Eingaben (Regionen), die in Form von unterschiedlichen Skalierungen und Seitenverhältnissen eintreffen, in eine Merkmalskarte von festgelegter Größe um (Höhe H und Breite W können als Hyperparameter festgelegt werden, die Tiefe ergibt sich aus der Anzahl der Filter der Convolution Schicht), auf die der darauffolgende Fully-connected Layer angewendet werden kann. Eine RoI ist ein rechteckiges Fenster in der Merkmalskarte der letzten Convolution Schicht und besteht aus Koordinaten der oberen linken Ecke und der Höhe und Breite des Fensters (x, y, h, w). Dieses Fenster wird dann in $H \times W$ Zellen geteilt, aus denen daraufhin, wie beim Max-Pooling, der größte Wert entnommen und in die entsprechende Ausgabezelle übertragen wird. Da die Division von $h/H \times w/W$ wahrscheinlich Nachkommastellen ergibt und die Zellen damit eine unklare Größe haben, wird die Größe jeder Zelle am linken und oberen Rand abgerundet und am rechten und unteren Rand aufgerundet. In Abbildung 3.3 wird farblich illustriert, wie RoI-Pooling funktioniert, wenn die Division keine ganzen Zahlen ergibt. Dazu wird eine RoI mit $h = 8$ und $w = 5$ aus den Aktivierungen eines Filters einer Convolution Schicht und durch die RoI-Pooling Operation auf eine 3×3 Merkmalskarte reduziert.

3.2.3 Faster R-CNN

Mit Faster R-CNN wird das Region Proposal Netzwerk (RPN) eingeführt und ersetzt damit den Selective Search Algorithmus in den Vorgängerversionen. Das RPN setzt auf der letzten Convolution Schicht auf. Im folgenden wird darauf eingegangen, was das RPN vorhersagt und wie aus der Ausgabe des RPN die RoIs entstehen. Vorher wird das Konzept der Anker erklärt, auf denen die Region Proposals basieren. Die RoIs werden dann wie in Fast R-CNN

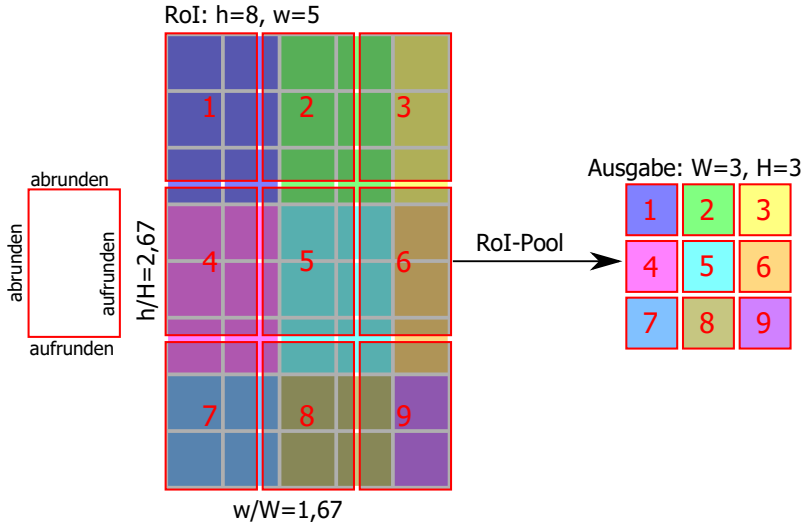

Abbildung 3.3: RoI Pooling Operation bei der eine RoI mit $h = 8$ und $w = 5$ auf eine 3×3 Merkmalskarte reduziert wird.

mit der RoI-Pooling Operation auf eine feste Größe reduziert und daraufhin klassifiziert.

3.2.3.1 Region Proposal Netzwerk

Das RPN bekommt als Eingabe die Merkmalskarten des letzten Convolution Layers. Daraus wird mit einem Convolution Layer für jede Position ein Merkmalsvektor extrahiert. Jeder Merkmalsvektor wird dann entweder mit einer zwei-Klassen Softmax Funktion[2] klassifiziert in „Objekt“ oder „nicht Objekt“ bzw. Hintergrund. Dies geschieht wieder mit einem Convolution Layer mit rezeptivem Feld von 1×1, sodass eine Merkmalskarte entsteht, in der jede Position einen „Objectness Score“ besitzt. Die Merkmalsvektoren des Convolution Layers im RPN werden außerdem vom Regressor der Bounding Box Koordinaten verwendet. Auch hier wird ein Convolution Layer mit rezeptivem Feld von 1×1 verwendet, um an jeder Position Multiplikatoren[3] für x, y, *width* und *height* einer Bounding Box vorherzusagen, wobei (x, y) den Mittelpunkt der Bbox angeben. In Abbildung 3.4 werden die Vorgänge im RPN visualisiert.

[2] Die Softmax Funktion ist der normalisierende Teil bei der Anwendung des Cross Entropy Fehlers, beschrieben in 2.2.3 und zu sehen in Abb. 2.8

[3] Die Koordinaten werden nicht direkt vorhergesagt, sondern Multiplikatoren, die eine sogenannte Ankerbox umformt. Mit den Multiplikatoren und den Koordinaten der Ankerboxen lassen sich die finalen Koordinaten der Bounding Boxen berechnen. Im Folgenden mehr dazu.

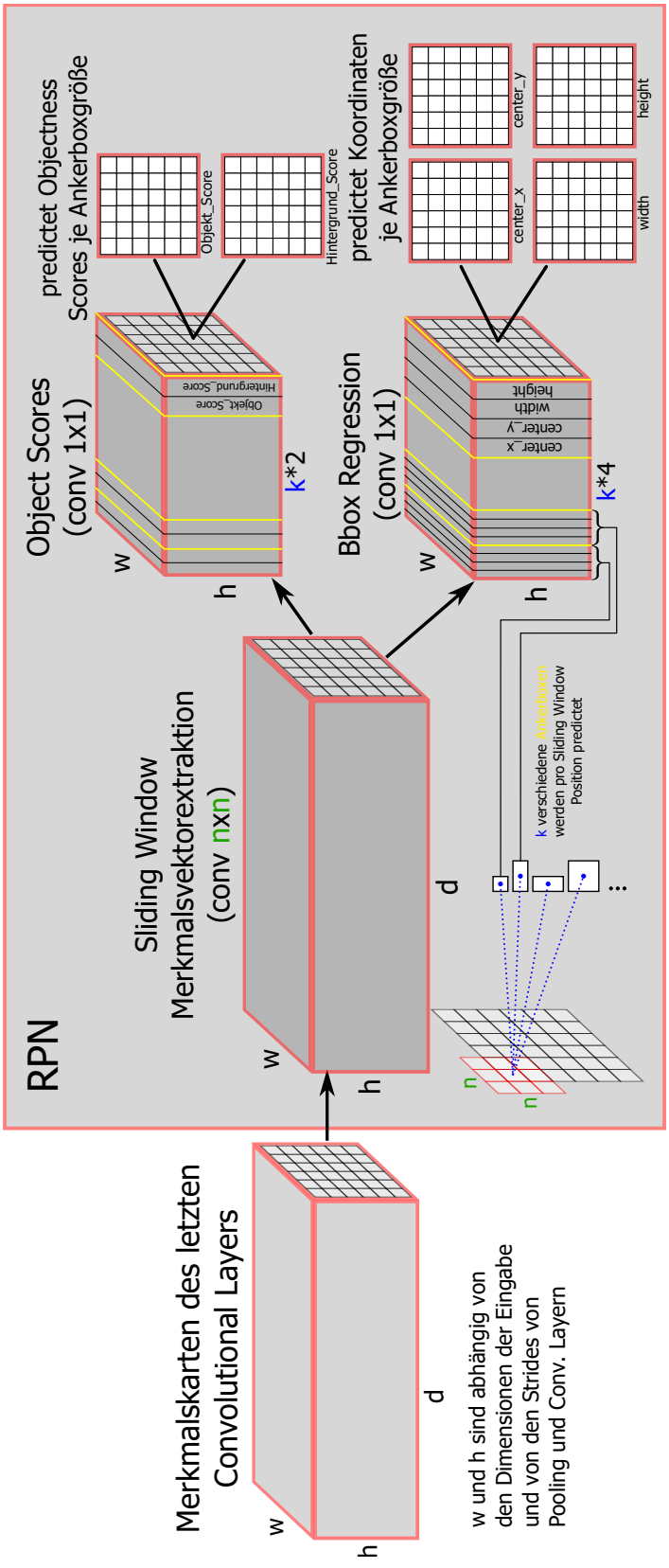

Abbildung 3.4: Region Proposal Netzwerk

3.2.3.2 Anker

An jedem Punkt der Merkmalskarten des letzten Convolution Layers wird ein Anker festgelegt. Für jeden Anker predictet das RPN k verschieden große „Ankerboxen“. In [12] werden z.B. drei Dimensionen (128^2, 256^2, 512^2) von Ankerboxen mit jeweils drei verschiedenen Seitenverhältnissen ($2:1$, $1:1$, $1:2$) verwendet was neun größen von Ankerboxen ergibt. Für jeden Anker predicten Klassifikator und Regressor für jede Ankerboxgröße (also k mal) die Object Scores und Multiplikatoren für x, y, *width* und *height*. Bei einem Bild mit einer Größe von 960×640 Pixeln, einem Pooling stride von insgesamt 2^4 resultieren die Merkmalskarten des letzten Convolution Layers in 60×40 Pixeln. Daraus ergeben sich 2400 Anker und $2400 \times k$ predictet Ankerboxen.

3.2.3.3 RPN Ausgabe wandeln zu RoI

Die vorhergesagten Multiplikatoren sollen eine Ankerbox in ihrer Höhe und Breite, sowie ihre Position (x, y) anpassen, sodass eine Bbox ensteht, die ein mögliches Objekt besser lokalisiert und umschließt (s. Abb. 3.5). Die resultierenden Koordinaten ergeben die RoI. Sie werden auf folgende Weise berechnet:

$$cx_{RoI} = x_{mult} \times w_{anker} + cx_{anker} \tag{3.1}$$

$$cy_{RoI} = y_{mult} \times h_{anker} + cy_{anker} \tag{3.2}$$

$$w_{RoI} = exp(w_{mult}) \times w_{anker} \tag{3.3}$$

$$h_{RoI} = exp(h_{mult}) \times h_{anker} \tag{3.4}$$

Daraufhin werden RoIs, die über den Bildrand hinaus gehen, abgeschnitten und RoIs, die eine zu geringe Breite oder Höhe haben, werden herausgefiltert. Auf die übrigbleibenden RoIs wird Non-Maximum Suppression (NMS) angewendet. NMS entfernt redundante RoIs, indem bei überlappenden Regionen nur die mit den besten Object Scores beibehalten werden. Die besten N RoIs werden mit dem Fast R-CNN Klassifikator klassifiziert und daraufhin ein weiteres Mal mit NMS gefiltert, um die finalen Detektionen zu erhalten.

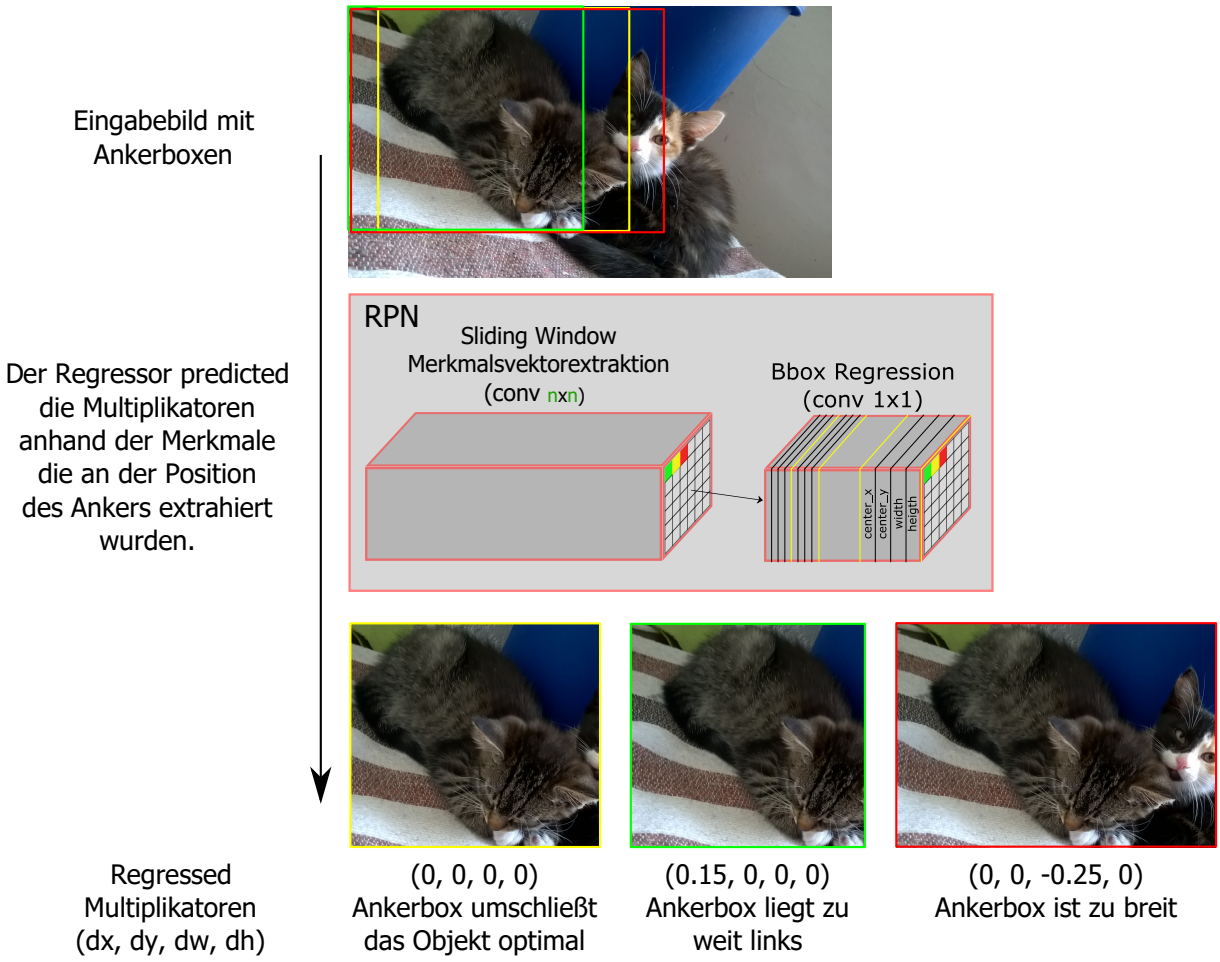

Abbildung 3.5: Predictete Multiplikatoren der Ankerboxkoordinaten

4 Implementierung und Methodik

4.1 Software und Hardware

4.1.1 GPU-Computing

Das Training von künstlichen Neuronalen Netzen profitiert enorm von der Verwendung von Grafikkarten (GPU - Graphical Processing Unit) an Stelle von CPUs. GPUs wurden entworfen, um Matrixoperationen effizient durchführen zu können. Dazu haben GPUs eine Vielzahl an Rechenkernen, auf die die Matrixoperationen aufgeteilt und parallelisiert abgearbeitet werden. CPUs hingegen haben vergleichsweise nur wenige Kerne, die für eine sequentielle Abarbeitung von Befehlen optimiert sind und die Elemente einer Matrixoperationen hintereinander berechnen müssen. GPUs können ihre Rechenstärke jedoch nur ausnutzen, wenn die Daten, auf denen die Operationen ausgeführt werden sollen, geliefert werden können. Daher ist abgesehen von der Anzahl der Rechenkerne, der Taktrate und Speichergröße auch die Speicherbandbreite ein wichtiger Faktor, um Deep Learning mit großen Modellen zu praktizieren. Aktuelle Verbrauchergrafikkarten erreichen dabei bis zu 10,6 TFLOP/s Single Precision (SP) parallele Rechenpower und 484 GB/s Speicherbandbreite (GTX 1080 Ti), während Intel Haswell CPUs 1,4 TFLOP/s SP und eine Speicherbandbreite von unter 100 GB/s aufweisen[14].

Für das Training von Modellen in dieser Arbeit stand ein System mit einer Geforce GTX 1060 (6GB) und eins mit einer Geforce GTX 1080 zur Verfügung.

4.1.2 Tensorflow

Tensorflow ist eine Open-Source-Programmbibliothek und wurde von Google entwickelt, um die Erforschung vom maschinellen Lernen mit künstlichen Neuronalen Netzen zu verbreiten, indem ein leichterer Einstieg ermöglicht wird. Dazu bietet Tensorflow low-level Schnittstellen für Programmiersprachen wie Python, Java, C++ und Go. Tensorflow erlaubt es, Code ohne Mehraufwand auf unterschiedlichster Hardware laufen zu lassen. Dieser kann auf

CPUs oder GPUs auf Desktoprechnern, Servern oder mobilen Geräten ausgeführt werden. In Tensorflows mid-level und high-level APIs, werden Funktionalitäten bereitgestellt, um Modelle zu erstellen, diese zu trainieren, evaluieren, predicten und zum Speichern und Laden von Modellen.

Ein Modell wird in Tensorflow in Form eines Graphen (`tf.Graph`) gebaut. Dieser besteht aus Knoten und Kanten. Die Knoten sind mathematische Operationen, die Eingabe- und Ausgabetensoren haben. Die Kanten sind die Tensoren (`tf.Tensor`), beschrieben durch den *rank* (Anzahl Dimensionen) und *shape* (die Länge des Tensors in jeder Dimension). Im Graphen selbst sind die eigentlichen Werte nicht enthalten. Ein `tf.Session` Objekt beinhaltet die Tensorflow Runtime. Wenn die `tf.Session` auf einem `tf.Graph` ausgeführt wird, backtrackt Tensorflow durch den Graphen und führt alle Knoten, die von der Eingabe hin zur Ausgabe führen, aus.

4.1.3 Keras

Keras ist eine high-level API zur Erstellung von Deep Neural Networks, die auf anderen APIs wie Theano, Tensorflow und CNTK aufbaut. Keras verfolgt das Ziel, eine einfach zu benutzende API zu sein. Der Benutzer soll in wenigen Schritten eigene Modelle erstellen und trainieren können. Dazu bietet Keras eine große Auswahl vordefinierter Schichten, Kostenfunktionen, Optimierungsfunktionen, Aktivierungsfunktionen und viele weitere essenzielle Bestandteile künstlicher Neuronaler Netze an. Auch können die vorhandenen Funktionalitäten mit eigenen Modulen erweitert werden.

Keras stellt zwei verschiedenen Modelltypen bereit. Mit dem `Sequential` Model lassen sich geradlinige Modelle erstellen, deren Schichten hintereinander gereiht sind, eine Eingabe und eine Ausgabe haben. Für komplexere Aufgaben bietet Keras die „functional API", mit der Modelle mit mehreren Ausgaben und mehrfach genutzten Schichten, wie sie in Faster RCNN benötigt werden, realisiert werden können.

Im folgenden Beispiel wird die XOR-Funktion aus 2.2.1 in einem Keras `Sequential` Model umgesetzt. Dazu werden die benötigten Klassen importiert:

```
from keras.models import Sequential
from keras.layers import Dense
import numpy as np
```

Daraufhin kann das `Sequential` Model instanziiert werden. Die erste Schicht ist ein Fully-Connected Layer (in Keras „`Dense`" genannt) mit zwei Neuronen und der tanh Aktivierungsfunktion. Mit der ersten Schicht muss außerdem die Form der Eingabedaten festgelegt werden

(hier ein Vektor von zwei Werten: `input_dim=2`). Darauf folgt die Ausgabeschicht wieder in Form eines `Dense` Layers mit nur einem Neuron und der Sigmoid Aktivierungsfunktion.

```
model = Sequential()
model.add(Dense(2, activation='tanh', input_dim=2))
model.add(Dense(1, activation='sigmoid'))
```

Das Modell wird kompiliert mit Stochastic Gradient Descent als Optimierungsfunktion und der binary Crossentropy Kostenfunktion, um zwischen den beiden Klassen 0 und 1 zu unterscheiden.

```
model.compile(optimizer='sgd', loss='binary_crossentropy')
```

Nun fehlen noch die Daten, auf denen das Modell trainieren soll. Die `fit` Methode des Modells, mit der das Training startet, bekommt als Parameter die Daten X und ihre Label y sowie die Anzahl der Epochen, die es trainieren soll. Keras gibt dann epochenweise den Traingsfehler aus.

```
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
model.fit(X, y, epochs=10000)
```

Mit der `predict` Methode wird das Modell auf die gegebenen Daten X angewendet.

```
results = model.predict(X)
print(results)
#[[0.03734707][0.93873811][0.93955719][0.03086801]]
```

4.1.4 Faster RCNN Keras Implementierung

Zum Training der Region-based Convolutional Neural Networks wurde eine Keras Implementierung von Faster RCNN verwendet [15]. Der Autor hält sich an die Standardeinstellungen der Ankerboxgrößen von 128, 256 und 512, sowie Seitenverhätnissen von $1:1$, $1:2$ und $2:1$ für das RPN, wie sie im Faster RCNN Paper [12] beschrieben worden sind. Das Modell wird mit der Keras „functional API" zusammengesetzt.

Das Modell besteht aus den Shared Layern, deren Aktivierungen von RPN und dem anschließenden Detektor genutzt werden. Darin sind die Convolution Schichten und Max-Pooling Operationen zur Merkmalsextraktion der Bilder definiert. Die Dimensionen der Eingabedaten werden bei der Erstellung des Modells über den `img_input` übergeben. Dieser besteht aus einem 3D Tensor für die Bilddaten, wobei Bildhöhe und -breite nicht relevant für die

Erstellung der Convolution Schichten sind. Die Anzahl der Parameter wird nur durch die Tiefe der Eingabe beeinflusst, welche aus den Farbchanneln des Bildes besteht: `img_input = (None, None, 3)`. Mit dem `padding = 'same'` Parameter der Convolution Schichten wird festgelegt, dass die Ausgabe durch Zero-Padding (s. 2.4.2) dieselbe Höhe und Breite aufweisen wie die Eingabe.

```
def nn_base(img_input):
  # Block 1
  x = Conv2D(64, (3, 3), activation='relu', padding='same')(img_input)
  x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
  x = MaxPooling2D((2, 2), strides=(2, 2))(x)
  ...
  return x
```

Das RPN baut auf den Shared Layern auf, daher bekommt es diese mit übergeben. Außerdem wird die Anzahl der unterschiedlichen Ankerboxen benötigt, um die Größe der Ausgabeschichten („Objectness" Score und BBox Regression s. 3.4) festzulegen.

```
def rpn(base_layers, num_anchors):
  x = Conv2D(512, (3, 3), padding='same', activation='relu')(base_layers)
  x_class = Conv2D(num_anchors, (1, 1), activation='sigmoid')(x)
  x_regr = Conv2D(num_anchors * 4, (1, 1), activation='linear')(x)
  return [x_class, x_regr, base_layers]
```

Der letzte Teil des Modells ist der Detektor bzw. Klassifikator aus Fast RCNN [11]. Auch dieser baut auf den Shared Layern auf. Die erste Schicht im Klassifikator ist der RoI-Pooling Layer (s. 3.2.2.1), der die zu klassifizierenden Regionen (`input_rois`) und Shared Layer Aktivierungen auf die festgelegte Größe gegeben durch `pooling_regions`, pooled. Mit `num_rois` wird die Batchgröße für den Klassifikator festgelegt. `TimeDistributed` Layer sind Keras Layer Wrapper, mit denen die ihnen übergebene Schicht bzw. Operation um eine Dimension erweitert werden, sodass sie mehrfach ausgeführt werden, in diesem Fall auf die Anzahl der zu klassifizierenden RoIs `num_rois`. Am Ende stehen die Softmax Wahrscheinlichkeiten in `out_class` und die BBox Regression Parameter in `out_regr` (s. auch 3.2).

```
def classifier(base_layers, input_rois, num_rois, nb_classes = 21):
  pooling_regions = 7
  out_roi_pool = RoiPoolingConv(pooling_regions, num_rois)([base_layers,
                                        input_rois])
  out = TimeDistributed(Flatten())(out_roi_pool)
  out = TimeDistributed(Dense(4096, activation='relu'))(out)
  out = TimeDistributed(Dropout(0.5))(out)
```

```
...
out_class = TimeDistributed(Dense(nb_classes, activation='softmax'))(out)
out_regr = TimeDistributed(Dense(4 * (nb_classes-1), activation='linear'))
                                        (out)
return [out_class, out_regr]
```

4.2 Pascal VOC Daten

Das Pascal VOC Projekt hat in den Jahren 2005 bis 2012 Challenges zur Erkennung von Objektklassen in Bildern angeboten in denen Teilnehmer ihre für verschiedene Kategorien erstellten Modelle mit anderen Teilnehmern messen konnten. Dazu wurden jährlich Datensätze mit Bildern und Annotationen für Bildklassifikation, Objektdetektion, Objektsegmentierung und weitere Aufgaben bereitgestellt. Für das Training der Modelle in dieser Arbeit wurde der Trainings- und Validationsdatensatz der VOC Challenge 2012 verwendet. Evaluiert wurden die Modelle mit dem Testdatensatz der VOC Challenge 2007, da in den anderern Jahren kein Testdatensatz mit Annotationen verfügbar gemacht wurde. Die Bilder in den Datensätzen sind ausgewählt, um eine möglichst große Bandbreite an Variationen wie Größe, Belichtung, Pose und Verdeckung der Objektklassen zu beinhalten, um eine repräsentative Menge von natürlichen Bildern zu erlangen. Jedes Bild kommt mit einer Annotation-Datei, in der Bounding Boxen der Objekte und deren Klassenlabel von 20 verschiedenen Klassen zu finden sind. Diese Klassen sind:

- person
- bird, cat, cow, dog, horse, sheep
- aeroplane, bicycle, boat, bus, car, motorbike, train
- bottle, chair, diningtable, pottedplant, sofa, tvmonitor

4.3 Vorgehen Modelltraining und Modellevaluation

Im Folgenden wird der grundsätzliche Trainingsprozess und die Evaluation erläutert, die bei allen ausgewerteten Modellen in Kapitel 4.4 angewendet wurde.

4.3.1 Trainingsprozess

Trainiert werden die Modelle auf dem „VOC 2012 trainval" Datensatz, welcher aus 17125 Bildern besteht. Zum Trainingsbeginn werden die Bilder zufällig gemischt und aufgeteilt in

80% Trainingsbilder und 20% Validationsbilder. Es wird auf allen 20 in den VOC Daten vorkommenden Klassen trainiert. Eine Epoche besteht aus dem Training des RPN und des Klassifikators in Folge auf je 1000 Bildern (nicht unbedingt auf denselben Bildern). Der Validationsfehler wurde nach jeder Epoche auf beiden Modellen berechnet. Cross Validation ist hier nicht praktikabel, da das Training eines einzigen Modells bereits Tage dauern kann. Um die Zeit des Trainings etwas zu reduzieren, wird in den ersten Epochen auf 300 Bildern validiert, im späteren Verlauf aber erhöht, sodass ein stabilerer Validationsfehler berechnet wird. Die Modellparameter werden nach jeder Epoche gespeichert, um ein eventuell unterbrochenes Training an derselben Stelle fortführen zu können. Bei jeder Verbesserung des Gesamtvalidationsfehlers wird separat auch dieses Modell gespeichert. Earlystopping beendet das Training nach 20 Epochen ohne Verbesserung des Gesamtvalidationsfehlers. Außerdem werden die Validationsfehler von RPN und Klassifikator separat beobachtet, um eine unabhängige Reduktion der Lernraten vornehmen zu können, da das Training des RPN deutlich früher stagniert.

4.3.1.1 Keras-FRCNN Trainingsablauf

Im Folgenden wird der Programmablauf der verwendeten Implementierung im Training erläutert (begleitend in Abb. 4.1 grob dargestellt) .

Nach der Instanziierung und Kompilierung des RPN- und Klassifikator-Modells beginnt das Training auf dem RPN. Der `model_rpn.fit_generator()` Methode wird ein Generator übergeben, der die Bilder vorbereitet und die Ground-Truth Daten für das RPN berechnet. Der Generator berechnet für jeden Anker, ob dieser mit einem GT-Objekt übereinstimmt, sowie Bbox Regression Targets, mit denen die Ankerbox umgeformt werden soll, um das Objekt besser umschliessen zu können. Jedes Bild, mit dem trainiert oder validiert wird, wird „zero-centered". Dabei wird der Mittelwert der Farbchannel des kompletten Datensatzes von allen Pixeln des Bildes subtrahiert. Dadurch können Schichten, deren Anfangsinitialisierung der Parameter auch um den Nullpunkt verteilt liegen, schneller lernen. Die predicteten Object Scores und Regression Multiplikatoren des RPNs werden in `roihelpers.rpn_to_roi()` in RoIs umgeformt. Dazu wird jede der $\sim 2400 \times k$ Ankerboxen (s. 3.2.3.2) mit den predicteten Multiplikatoren x, y, w und h verrechnet, über den Bildrand stehende Ankerboxen werden abgeschnitten und zu kleine Ankerboxen werden herausgefiltert. Auf die übrigbleibenden RoIs wird Non Maximum Suppression angewendet, um die Regionen mit den 300 höchsten Objectness Scores beizubehalten. Diese werden weiterverwendet, um die Label für den Klassifikator zu berechnen. Dabei gelten Regionen, die eine Intersection over Union (IoU) mit den Ground Truth Objekt Bboxen von unter $0,5$ haben, als Hintergrund und darüber als

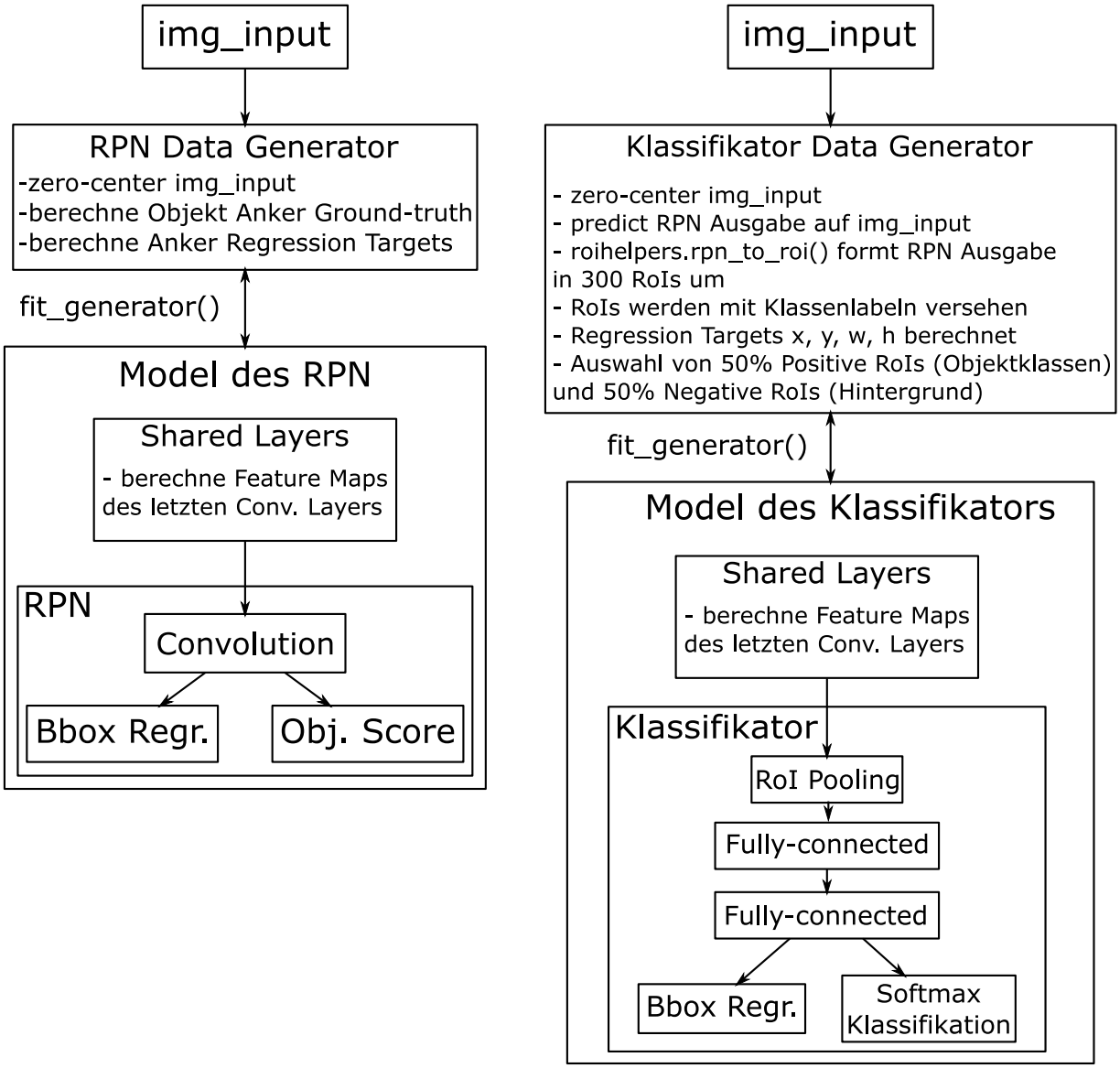

Abbildung 4.1: Grober Ablauf des Trainings mit Keras-FRCNN[15]

Klasse der Objekt Bbox mit der höchsten IoU. Für das Training des Klassifikators werden in gleichem Verhältnis Hintergrund-RoIs und Objekt-RoIs ausgewählt, um dem deutlich größeren Anteil an Hintergrund entgegen zu wirken.

Das Trainingsskript wurde von der ursprünglichen Implementierung abgeändert, indem die Batcherstellung für den Klassifikator von einem Data Generator durchgeführt wird, um das Training durch Multithreading beschleunigen zu können. In der ursprünglichen Implementierung wurden RPN und Klassifikator hintereinander auf demselben Bild trainiert. Die Batcherstellung des Klassifikators und Propagierung der Batch auf der GPU wurden hintereinander ausgeführt. Dadurch wurde das Training jedoch nur wenig beschleunigt. Multiprocessing hätte einen großen Vorteil gebracht, ist mit Data Generatoren in Keras aber nicht vorgesehen. Dafür bietet Keras extra Funktionalität mit der `keras.utils.Sequence` Klasse. Eine Änderung des Codes war jedoch nicht ohne Nebenwirkungen machbar.

4.3.2 Evaluationsprozess

Die Evaluation findet auf dem „VOC 2007 test" Datensatz statt. Die Vorverarbeitung der Bilder wird wie im Training auf gleiche Weise vorgenommen. Daraufhin werden die RPN Predictions und Merkmale der Shared Layer berechnet. Die RPN Predictions werden wie im Training in RoIs gewandelt und mit den Merkmalen der Shared Layer dem Klassifikator zum predicten der Softmax Klassenwahrscheinlichkeiten und Bbox Regression Multiplikatoren übergeben. Die predicteten Multiplikatoren der Bbox-Koordinaten werden auf die RoI-Koordinaten angewendet und auf die original Bildgröße zurückskaliert. Die RoIs werden für jede Klasse mit NMS gefiltert um Mehrfachdetektionen zu entfernen. Die Detektionen werden mit den Ground Truth Daten abgeglichen und als Treffer gezählt bei einer $IoU \geq 0.5$. Über alle Testbilder wird dann die „Average Precision" jeder Klasse berechnet, um die Güte des Modells zu bestimmen.

4.4 Training und Evaluation

In diesem Teil werden die Hyperparameterkonfigurationen der trainierten Modelle und die Erwartungen an die Ergebnisse vorgestellt. Dazu werden die Evaluationsergebnisse präsentiert und Rückschlüsse gezogen.

Um ausschließen zu können, dass die Netzstruktur für ein Scheitern des Lernens von wertvollen Merkmalen verantwortlich ist, wurde die Architektur von VGG16 [16] übernommen. Lediglich die Menge der Convolution Filter und die Neuronenanzahl der Fully-connected Schichten

ConvBlock1	ConvBlock2	ConvBlock3	ConvBlock4	ConvBlock5	FC
$3 \times 3 \times 64$	$3 \times 3 \times 128$	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	FC1 4096
$3 \times 3 \times 64$	$3 \times 3 \times 128$	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	Dropout 0.5
Max-Pool	Max-Pool	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	FC2 4096
		Max-Pool	Max-Pool	Max-Pool	Dropout 0.5
#Parameter:	138 Mio.				Softmax1000

Tabelle 4.1: VGG16 Netzarchitektur

wurden variiert. VGG16 (s. Tabelle 4.1) besteht aus fünf Blöcken Convolution Schichten, gefolgt von je einer Max-Pooling Schicht. Durchweg bestehen die Convolution Schichten aus 3×3 Filtern. Die Filtergröße wählen die Autoren bewusst unterschiedlich von der Netzarchitektur von [1] (7×7 Filter der ersten Convolution Schicht) und begründen es damit, dass ein Block von drei Convolution Schichten ohne Max-Pooling auch ein rezeptives Feld von 7×7 hat, jedoch eine niedrigere Parameteranzahl und dadurch eine regulierende Wirkung erhält.

4.4.1 Training #1

Die ersten Trainingsversuche wurden mit reduzierten Neuronen der VGG16 Fully-connected Schichten, angepasst an die Faster RCNN Architektur (s. Tabelle 4.2), durchgeführt. Dabei

ConvBlock1	ConvBlock2	ConvBlock3	ConvBlock4	ConvBlock5	RPN	Klass.
$3 \times 3 \times 64$	$3 \times 3 \times 128$	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	$1 \times 1 \times 512$	RoI Pool
$3 \times 3 \times 64$	$3 \times 3 \times 128$	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	Obj. Scores	FC1 2048
Max-Pool	Max-Pool	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	Bbox Regr.	Dropout 0.5
		Max-Pool	Max-Pool			FC2 2048
						Dropout 0.5
						Softmax21
#Parameter:	73 Mio.					Bbox Regr.

Tabelle 4.2: VGG16 (mit Veränderung der Anzahl Neuronen in FC Layern) eingefügt in Faster RCNN

wurde versucht, eine passende Lernrate der SGD Optimierungsfunktion zu finden. Es wurden Versuche mit vier verschiedenen Lernraten durchgeführt mitsamt Nesterov Momentum und eine Gewichtsinitialisierung „he_normal" [17], die mit Bedacht für ReLU Aktivierungsfunktionen und Modellen mit sehr vielen Schichten entwickelt wurde. Damit soll ein besseres lokales Minimum zu finden sein als ohne bzw. anderen Gewichtsinitialisierungen. Es ergeben sich die folgenden Hyperparameter:

- Fully-connected Schichten mit jeweils 2048 Neuronen

- Nesterov Momentum von 0.9
- Gewichtsinitialisierung mit „he_normal”
- Lernrate $0,001$
- Lernrate $0,0001$
- Lernrate $0,00001$
- Lernrate $0,00003$

Bei den beiden höheren Lernraten inklusive Momentum war zu erwarten, dass der Lernerfolg ausbleibt aufgrund von zu großen Gradienten. Nachdem sich das in den ersten Epochen bestätigte (in der Zeit sollten die größten Lernerfolge stattfinden), wurden die Trainings schnell abgebrochen. Die beiden Trainings mit niedrigeren Lernraten konnten leichte Lernerfolge erzielen, konnten aber auch nicht überzeugen und wurden beendet (s. Abb. A.1 und A.2).

Ein weiteres Training wurde mit ähnlichem Setup durchgeführt. Dabei wurde kein Dropout verwendet, eine Lernrate mit SGD von $0,001$ wurde in betracht gezogen mit dem Verzicht von Momentum. Zu erwarten waren noch schlechtere Ergebnisse durch den Verzicht von Momentum. Da die Bilder beim Training einzeln verarbeitet werden, ist zu erwarten, dass der Gradient stark verrauscht ist und eine vergleichsweise langsame Verbesserung des Losses stattfindet. Dass der Loss in diesem Fall in einem ähnlichem Tempo (s. Abb. A.3) wie in den vorherigen Trainings verbessert wird, kann mit dem Fehlen von Dropoutregulierung zu tun haben. Die Verwendung von Dropout verlängert die Trainingszeit üblicherweise um das zwei- bis dreifache [9]. Zu beobachten ist auch, dass das Modell ab Epoche 65 anfängt zu overfitten, was wieder auf das Fehlen von Dropout zurückzuführen ist. Die Evaluation des Modells mit dem besten Validationsfehler nach Epoche 118 auf den Testdaten ergibt daraufhin eine Mean Average Precision von $0,29$ (in Tabelle A.1 zu sehen).

4.4.2 Training #2

In den folgenden Trainings wurde bei den ersten beiden die originale VGG16 Netzarchitektur benutzt und in den darauffolgenden Trainings die Anzahl der Convolution Filter und Neuronen der Fully-connected Schichten reduziert. Als Optimierungsfunktion wurde Adam (Adaptive moment estimation [18]) mit einer Lernrate von $0,00001$ benutzt. Adam ist besonders gut zu gebrauchen bei verrauschten Parameterupdates, indem es für jeden Parameter eine eigene Lernrate speichert und während des Trainings anpassen kann. Dadurch wird jedoch geringfügig mehr Speicher benötigt. Adam selbst hat einige Hyperparameter, bei denen die Standardwerte einen guten Ausgangspunkt bilden. Im Paper zu Adam werden

einige Optimierungsmethoden mit Adam verglichen, dabei weist Adam etwas bessere Optimierungsraten als SGD mit Nesterov Momentum auf. In diesen Trainings wurde der Fehler gemacht, dass eine Standardeinstellung von Keras-FRCNN übersehen wurde, mit der die Klassen, auf denen trainiert wird, reihum dran kommen. Hier ist die Einstellung deaktiviert, sodass ein Bias der folgenden Modelle in Richtung der Klasse „Person" entstanden ist, da überwiegend viele „Person" Bilder im VOC trainval Datensatz vorhanden sind.

4.4.2.1 VGG16 Original mit Pretrain

Durch das Laden der VGG16 Gewichte konnte die Trainingszeit auf 50 Epochen reduziert werden. Wenn die Möglichkeit besteht, die Gewichte eines bereits trainierten Merkmalsextraktors zu übernehmen und darauf „finetuning" mit dem eigenen Datensatz zu betreiben, sollte dies in Erwägung gezogen werden. Vor allem grundlegende Merkmale der vorderen Convolution Schichten werden oft wiederverwendet und bloß die hinteren Schichten neu trainiert, um optimale Ergebnisse beim Training eines Modells zu erreichen. Tabelle 4.3 zeigt die genaue Netzkonfiguration. Epoche 44 weist den besten Validationsfehler auf mit einer mAP von $0,43$.

ConvBlock1	ConvBlock2	ConvBlock3	ConvBlock4	ConvBlock5	RPN	Klass.
$3 \times 3 \times 64$	$3 \times 3 \times 128$	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	$1 \times 1 \times 512$	RoI Pool
$3 \times 3 \times 64$	$3 \times 3 \times 128$	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	Obj. Scores	FC1 4096
Max-Pool	Max-Pool	$3 \times 3 \times 256$	$3 \times 3 \times 512$	$3 \times 3 \times 512$	Bbox Regr.	Dropout 0.5
		Max-Pool	Max-Pool			FC2 4096
						Dropout 0.5
						Softmax21
#Parameter:	138 Mio.					Bbox Regr.

Tabelle 4.3: VGG16 eingefügt in Faster RCNN

4.4.2.2 VGG16 Original ohne Pretrain

In diesem Modell beträgt die mAP $0,27$.

4.4.2.3 VGG16 mittel

In diesem Modell beträgt die mAP $0,22$.

4.4.2.4 VGG16 klein

In diesem Modell beträgt die mAP $0,26$.

4.4.3 Training #3

4.4.3.1 VGG16 balanced

In diesem Modell beträgt die mAP $0,51$.

4.4.3.2 VGG16 balanced locked

In diesem Modell beträgt die mAP 0.42, während im Vergleich zu „VGG16 balanced" ein höherer Wert erwartet wurde. Eigentlich hätte durch die ersten vier Blöcke von nicht trainierbaren Convolution Schichten und damit beibehaltenen soliden lower level Merkmalen ein besseres Ergebnis erzielt werden müssen. Der Loss wird in Abb. A.9 dargestellt.

5 Ausblick und Fazit

Jedes Jahr werden neue Weiterentwicklungen von Deep Learning Techniken vorgestellt Im Hardwarebereich entwickelt Google mit ihren Tensor Processing Units leistungsfähigere Prozessoren speziell für Deep Learning. Softwareseitig gibt es bereits Mask RCNN, die nächste Weiterentwicklung von Faster RCNN, die die Objekterkennung vorangetrieben hat. Die Verwendung des „Focal Loss", wie er im RetinaNet verwendet wird, ermöglicht besser Erkennungsgenauigkeiten, indem vom Modell leicht klassifizierte Beispiele einen geringeren Einfluss auf den Gradienten haben als weniger gut klassifizierte Objekte.

Das Problem der Objektdetektion konnte mit einer Implementierung von Faster RCNN nachvollzogen werden. Einige Objektdetektionsmodelle wurden trainiert und einige Schlüsse konnten aus den Ergebnissen gezogen werden, was ein besseres Vorgehen beim Training von Modellen in Zukunft ermöglicht. Die CPU der verwendeten Systeme stellte ein Bottleneck im Trainingsprozess dar, weil der Code zur Batchvorbereitung nicht multiprocessingfähig war. Hier wäre eine Modifikation des Codes angebracht, um das Training zu beschleunigen.

A Anhang

A.1 Verwendung der Python Skripte

Beispiel starten eines Trainings mit:

```
python train_gen.py -p ~/VOCdevkit
        --network vgg16
        --output_model_path ~/train_results/
```

Beispiel abgebrochenes Training fortführen:

```
python train_gen.py -p ~/VOCdevkit --network vgg16
        --input_weight_path ~/train_results/model_frcnn.hdf5
        --output_model_path ~/train_results/
        --resume_train
```

Beispiel Evaluation eines Modells:

```
python evaluate.py -p ~/VOCdevkit
        --network vgg16
        --input_weight_path ~/train_results/model_frcnn.hdf5
```

sheep	horse	bicycle	motorbike	cow	sofa	aeroplane	dog	bus	cat
0, 35	0, 37	0, 32	0., 44	0, 27	0, 15	0, 37	0, 24	0, 37	0, 27

person	train	boat	bottle	car	pottedplant	tvmonitor	chair	bird	diningtable
0, 35	0, 35	0, 17	0, 31	0, 38	0, 15	0, 30	0, 17	0, 20	0, 23

Tabelle A.1: Average Precision von Training #1

A.2 Loss Plots der Modelle

A.2.1 Training #1

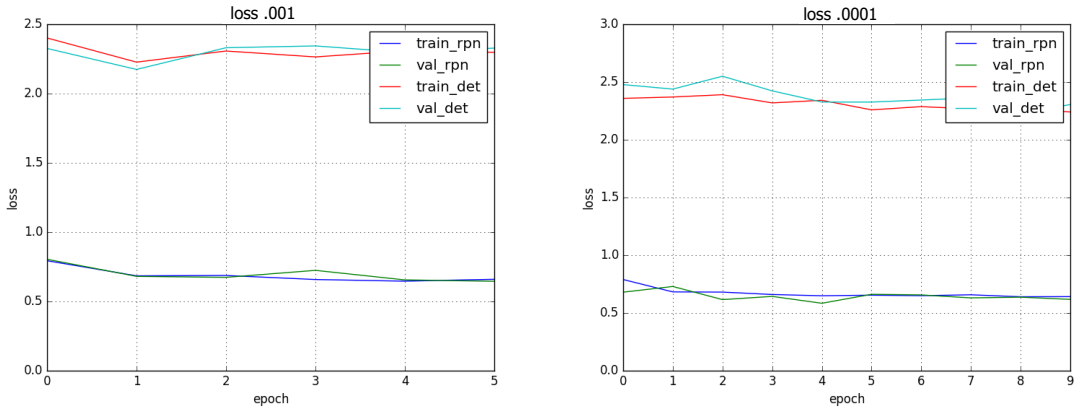

Abbildung A.1: Loss Plots mit hohen Lernraten von Training #1

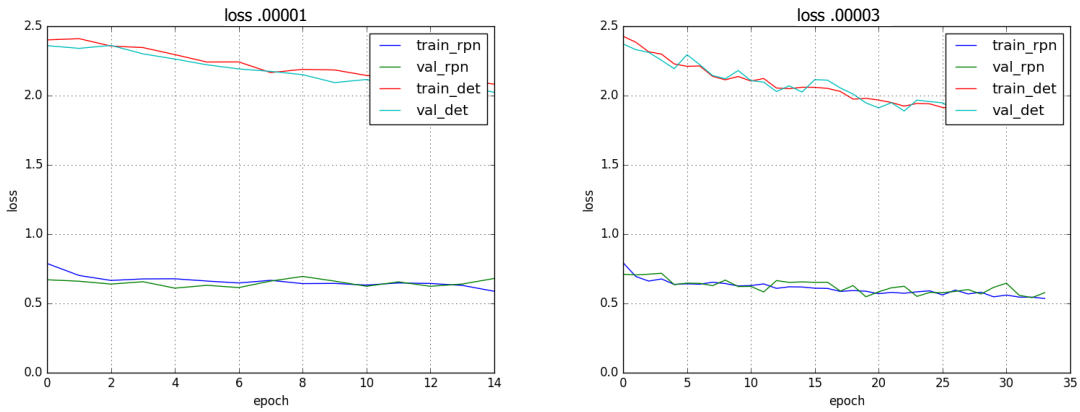

Abbildung A.2: Losses mit niedrigen Lernraten von Training #1

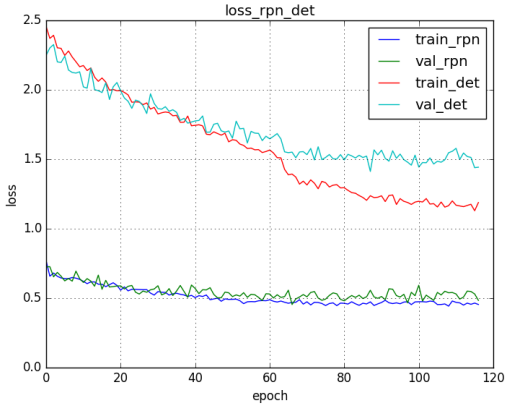

Abbildung A.3: Loss ohne Momentum und ohne Dropout

A.2.2 Training #2

A.2.2.1 VGG16 Original mit Pretrain

sheep	horse	bicycle	motorbike	cow	sofa	aeroplane	dog	bus	cat
0.44	0.43	0.48	0.46	0.41	0.36	0.49	0.51	0.48	0.60

person	train	boat	bottle	car	pottedplant	tvmonitor	chair	bird	diningtable
0.75	0.39	0.28	0.33	0.55	0, 15	0.36	0.32	0.53	0.23

Tabelle A.2: Average Precision von Training #2 VGG16 Original mit Pretrain

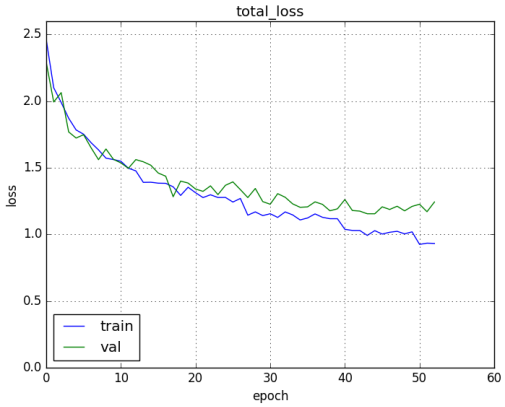

Abbildung A.4: Loss VGG16 Original mit Pretrain

A.2.2.2 VGG16 Original ohne Pretrain

sheep	horse	bicycle	motorbike	cow	sofa	aeroplane	dog	bus	cat
0.22	0.42	0.29	0.46	0.32	0.20	0.49	0.31	0.32	0.34

person	train	boat	bottle	car	pottedplant	tvmonitor	chair	bird	diningtable
0.55	0.27	0.12	0.07	0.43	0.08	0.27	0.11	0.18	0.15

Tabelle A.3: Average Precision von Training #2 Original ohne Pretrain

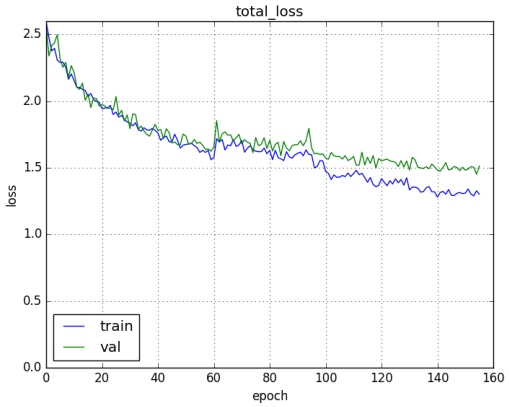

Abbildung A.5: Loss VGG16 Original ohne Pretrain

A.2.2.3 VGG16 mittel

sheep	horse	bicycle	motorbike	cow	sofa	aeroplane	dog	bus	cat
0.18	0.31	0.16	0.33	0.38	0.08	0.21	0.21	0.15	0.27

person	train	boat	bottle	car	pottedplant	tvmonitor	chair	bird	diningtable
0.42	0.13	0.17	0.27	0.29	0.19	0.14	0.13	0.24	0.08

Tabelle A.4: Average Precision von Training #2 VGG16 mittel

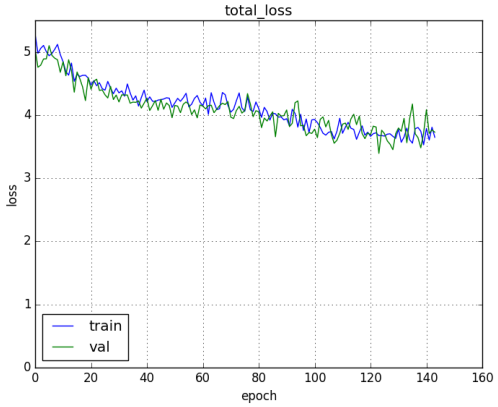

Abbildung A.6: Loss VGG16 mittel

A.2.2.4 VGG16 klein

sheep	horse	bicycle	motorbike	cow	sofa	aeroplane	dog	bus	cat
0.25	0.34	0.44	0.36	0.48	0.54	0.26	0.19	0.12	0.17

person	train	boat	bottle	car	pottedplant	tvmonitor	chair	bird	diningtable
0.34	0.27	1.0	1.0	0.43	1.0	0.06	0.16	0.49	0.29

Tabelle A.5: Average Precision von Training #2 VGG16 klein

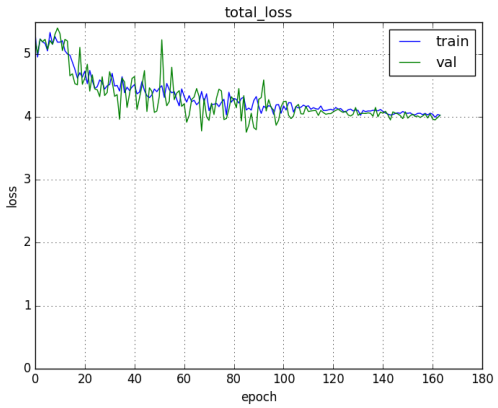

Abbildung A.7: Loss VGG16 klein

A.2.3 Training #3

A.2.3.1 VGG16 balanced

sheep	horse	bicycle	motorbike	cow	sofa	aeroplane	dog	bus	cat
0.62	0.55	0.57	0.57	0.58	0.40	0.48	0.57	0.61	0.67

person	train	boat	bottle	car	pottedplant	tvmonitor	chair	bird	diningtable
0.65	0.54	0.37	0.38	0.63	0.27	0.54	0.34	0.53	0.36

Tabelle A.6: Average Precision von Training #3 VGG16 balanced

Abbildung A.8: Loss VGG16 balanced

A.2.3.2 VGG16 balanced locked

sheep	horse	bicycle	motorbike	cow	sofa	aeroplane	dog	bus	cat
0.50	0.50	0.43	0.52	0.56	0.23	0.41	0.45	0.46	0.50

person	train	boat	bottle	car	pottedplant	tvmonitor	chair	bird	diningtable
0.54	0.45	0.30	0.32	0.52	0.22	0.53	0.25	0.44	0.21

Tabelle A.7: Average Precision von Training #3 VGG16 balanced locked

Abbildung A.9: Loss VGG16 balanced locked

A.3 Systemsetup Dokumentation GTX1060

Systemsetup Dokumentation und Liste installierter Python Packages auf der beiliegenden CD zu finden.

A.4 Systemsetup Dokumentation GTX1080

Liste installierter Python Packages auf der beiliegenden CD zu finden.

Für die Ausführung des Evaluationsskriptes „evaluate.py” wird eine Installation von „scikit-learn” benötigt.

Literaturverzeichnis

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] D. Kriesel, *Ein kleiner Überblick über Neuronale Netze*, 2007, p. 9-13, 33-34. [Online]. Available: http://www.dkriesel.com

[3] J. J. H. und D. W. Tank, "Neural computation of decisions in optimization problems," *Biological Cybernetics*, vol. 52:141-152, 1985.

[4] R. W. D. Rumelhart, G. Hinton, "Learning representations by back-propagating errors," *Nature*, vol. 323:533-536, 1986.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[6] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, pp. 65–386, 1958.

[7] A. K. Justin Johnson, Fei-Fei Li, "Cs231n: Convolutional neural networks for visual recognition," 2016. [Online]. Available: http://cs231n.github.io

[8] P. Golik, P. Doetsch, and H. Ney, "Cross-entropy vs. squared error training: a theoretical and experimental comparison," in *Interspeech*, Lyon, France, Aug. 2013, pp. 1756–1760.

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

[10] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Computer Vision and Pattern Recognition*, 2014.

[11] R. Girshick, “Fast r-cnn,” in *International Conference on Computer Vision (ICCV)*, 2015.

[12] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[13] J. Uijlings, K. van de Sande, T. Gevers, and A. Smeulders, “Selective search for object recognition,” *International Journal of Computer Vision*, 2013. [Online]. Available: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf

[14] N. Corporation, “Cuda c programming guide,” 2018. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[15] Y. Henon, “keras-frcnn,” https://github.com/yhenon/keras-frcnn, 2017.

[16] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[17] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[18] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[19] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The pascal visual object classes (voc) challenge,” *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, Jun. 2010.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Master-Abschlussarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Brandenburg an der Havel, den 14.03.2018

Jonas Preckwinkel