UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

WeST People and Knowledge Networks
Institute for Web Science and Technologies

Vorhersage von Gefechts-Ausgängen im Echtzeit-Strategiespiel StarCraft II mittels Convolutional Neural Networks

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von
Frank Schaust

Erstgutachter:	Prof. Dr. Steffen Staab Institute for Web Science and Technologies
Zweitgutachter:	Lukas Schmelzeisen Institute for Web Science and Technologies

Koblenz, im Mai 2019

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbstständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐
Der Text dieser Arbeit ist unter einer Creative Commons Lizenz (CC BY-SA 4.0) verfügbar.	☐	☐
Der Quellcode ist unter einer GNU General Public License (GPLv3) verfügbar.	☐	☐
Die erhobenen Daten sind unter einer Creative Commons Lizenz (CC BY-SA 4.0) verfügbar.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Anmerkung

- If you would like us to contact you for the graduation ceremony, please provide your personal E-mail address: .....
- If you would like us to send you an invite to join the WeST Alumni and Members group on LinkedIn, please provide your LinkedIn ID : .....

Zusammenfassung

Abstract

1 Motivation

Das Thema der Bachelorarbeit ist die Adaption von Algorithmen aus dem Machine-Learning (zu Deutsch: Maschinelles Lernen) zur Vorhersage von Gefechts-Ausgängen zweier feindlicher Armeen in dem Echtzeit-Strategiespiel StarCraft II. Das Ziel der Arbeit ist die Untersuchung, ob existierende Algorithmen, die im Bereich der Image-Classification (zu Deutsch: Bildklassifikation) Anwendung finden, für komplexe Aufgaben in einer atypischen Domäne genutzt werden können. Kernbestandteil ist daher der Vergleich bestehender Architekturen auf dieser neuen Domäne. Die Domäne unterscheidet sich von konventionellen Bildformaten, da die Eingabedaten in Form von domänenspezifischen Eigenschaftsmatrizen vorliegen. Es werden vorwiegend Convolutional Neural Networks (folgend CNNs, zu Deutsch etwa: Faltende Neuronale Netze) genutzt um Matrix-Repräsentationen von Spielzuständen zu verarbeiten und auf Grundlage derer den Ausgang der Kampfszenarien vorherzusagen.

CNNs sind eine Machine-Learning-Methode, die unter anderem von den Gewinnern der ImageNet Large Scale Visual Recognition Challenge *GoogLeNet* [19] genutzt wurde. In diesem Aufgabenfeld konnten CNNs im Vergleich zu anderen Image Classification Verfahren gute Resultate erzielen. Außerdem fanden CNNs in der Vergangenheit erfolgreich Anwendung in anderen Forschungsgebieten, welche keinen direkten Bezug zu Image Classification hatten, wie zum Beispiel bei der Entdeckung von medizinischen Wirkstoffen (AtomNet [21]).

Gefechte in StarCraft II bestehen aus zwei oder mehr Armeen. Jede Armee wird von jeweils einem Spieler kontrolliert, dessen Ziel es ist mit den Fähigkeiten seiner eigenen Einheiten sämtliche Einheiten des Gegners zu zerstören. Gefechte beinhalten komplexe Interaktionen, da jede Einheit ihre Position auf dem Schlachtfeld ändern und feindliche Einheiten angreifen und schlussendlich zerstören kann. Erfahrene menschliche Spieler können Gefechts-Ausgänge solcher Aufeinandertreffen im Allgemeinen mit einer guten Wahrscheinlichkeit vorhersagen.

Um den Zustand des Spiels darzustellen werden Matrix-Repräsentationen genutzt. Jede Matrix repräsentiert einen anderen Aspekt (wie zum Beispiel Einheiten-Gesundheit, Einheiten-Typ, usw.) des Spiel-Zustandes in Bezug auf die Position auf dem Spielfeld. Die Aufgabe der Gefechts-Vorhersage kann als Image-Classification-Aufgabe gesehen werden, da die genutzten Matrizen in einer ähnlichen Form klassifiziert werden können wie Bilder. Es werden Matrizen als Eingabe genutzt und es werden diskrete Werte als Ausgabe erhalten. Der Unterschied zwischen Bildern und Matrix-Repräsentation ist, dass jedes Bild explizit ein oder mehrere Objekte zeigt, welche klassifiziert werden, während jede Matrix-Repräsentation eine implizierte Bedeutung hat. Die impliziten Bedeutungen zu lernen und zu kombinieren ist eine interessante Aufgabe für eine CNN-Architektur.

Der Versuch das Verhalten von Computerspielen vorherzusagen ist interessant, weil sie eine hervorragende Umgebung bieten um die Leistungsfähigkeit von Machine-Learning-Methoden zu testen. Sie stellen eine kontrollierbare Menge an Umweltfak-

toren bereit – im Fall von StarCraft II die Anzahl der Einheiten, ihre Positionen und Kampfstärke – welche genutzt werden kann um beliebig viele Trainingsdaten von kontrollierbarer Schwierigkeit zu erzeugen. Der Zustand des Spiels und der Ausgang des Gefechts sind klar zu bemessen und es gibt konkrete Regeln, die ein Algorithmus lernen muss, um zuverlässige Vorhersagen zu treffen, wie zum Beispiel: *Ein Marine greift mit X Schadenspunkten an*, oder *Wenn die Lebenspunkte einer Einheit auf 0 oder tiefer fallen, stirbt sie*. Die Schwierigkeit für einen Algorithmus liegt darin die richtigen Schlüsse zu ziehen, ohne explizites Wissen über die zugrundeliegenden Regeln des Spiels zu haben. Neben der Eignung als Machine-Learning-Problem, wird ein erfolgreicher Vorhersage-Algorithmus benötigt um künstliche Intelligenzen (KIs) für StarCraft II zu implementieren. Damit KIs das Spiel auf einem hohen Niveau spielen können, müssen sie in der Lage sein, den Ausgang eines Gefechtes vorherzusagen um abzuwägen, ob sie den Kampf eingehen können, oder den Rückzug wählen sollten.

1.1 StarCraft II

StarCraft II ist ein Echtzeit-Strategiespiel entwickelt und veröffentlicht von Blizzard Entertainment. Es ist überwiegend deterministisch[1]. Das Ziel ist es alle feindlichen Einheiten auf einer Karte zu vernichten ohne selbst dabei vernichtet zu werden.

Zu Beginn eines jeden Spiels kann der Spieler zwischen drei Rassen wählen. Jeder Spieler kontrolliert eine Menge an Einheiten U = $\{u_1, u_2, ...u_n\}$. Im Rahmen der Arbeit wird lediglich auf die Gefechte des Spiels Bezug genommen. Von anderen Aspekten des Spiels, wie Basenbau oder Ressourcen-Management wird abstrahiert, um das Szenario einfach zu halten.

Das Spiel wird mit 20 Zuständen pro Sekunde berechnet, läuft jedoch für den menschlichen Spieler wie in Echtzeit ab. Durch die Berechnung der 20 Zustände pro Sekunde, kann auf Zeitschritte abstrahiert werden, wobei in jedem Zeitschritt die gegebenen Befehle den kommenden Zeitschritt beeinflussen. Daher kann der Spielablauf als eine diskrete Serie von Zuständen repräsentiert werden.

Jede Einheit besitzt eine allgemeine Beschreibung die ihr einen Typ $t(u_i)$ zuweist. $t(u_i)$ kann einer von zwei Typen sein $t(u_i) \in$ { Boden, Luft }. Zusätzlich kann jede Einheit mit einer Menge an Attributen $a(u_i)$ versehen werden, die ihre weiteren Eigenschaften spezifizieren. Eine Einheit kann Attributen aus der Menge { Leicht, Gepanzert, Biologisch, Mechanisch } zugeordnet werden.

Einheiten bewegen sich auf einer 2D-Karte und befinden sich zu jedem Zeitpunkt an einer spezifischen Position. Die Karte kann als Matrix mit Dimensionen $H \times B$, $H, B \in \mathbb{N}$ definiert werden, wodurch die Position einer Einheit als zweidimensionaler Punkt $p^t(u_i) = (x, y)$ verstanden werden kann, bei dem t die Spielzeit darstellt und $x \leq B, y \leq H$ mit $x \in [0, B], y \in [0, H]$ gilt.

[1] Die hauptsächlichen Quellen für Zufälligkeiten sind laut DeepMind *Angriffsgeschwindigkeit* und *Aktualierungsreihenfolge*, welche durch einen Random Seed (zu Deutsch etwa: Startwert) initialisiert werden. Diese Zufälligkeiten können durch das manuelle setzen eines festen Random Seeds umgangen werden [20].

Jede Einheit verfügt über ein Attribut Lebenspunkte $hp^t(u_i)$, eine Bewegungsrate $mr(u_i)$ und eine Sichtweite $sr(u_i)$. Lebenspunkte sind von der Spielzeit abhängig und können sich während dem Verlauf des Spiel ändern. Wenn $\exists t$ indem $hp^t(u_i) \leq 0$ gilt, so wird die Einheit for alle folgenden Zeitschritte $t' > t$ als tot behandelt und ist nicht mehr in der Lage ins Spielgeschehen einzugreifen. Die Bewegungsrate bestimmt wie schnell sich eine Einheit über die Spielkarte bewegen kann. Die Sichtweite bestimmt den Radius um eine freundliche Einheit herum in dem die Karte aufgedeckt wird. Ein Spieler kann nur jene gegnerischen Einheiten sehen, welche sich im Sichtbereich von mindestens einer freundlichen Einheit aufhalten. Zusätzlich verfügen Einheiten über weitere Attribute, welche ihre Kampfstärke definieren: Schaden $d(u_i)$, Abklingzeit $cd(u_i)$ und Waffenreichweite $wr(u_i)$. Der Schadenswert wird genutzt um den Schaden zu berechnen, den eine Einheit mit einem einzelnen Schuss verursacht. Abklingzeit bestimmt wie lange eine Einheit warten muss bevor sie einen weiteren Schuss abgeben darf und wird in Sekunden bemessen. Die Waffenreichweite ist der Radius eines Kreises in dessen Fläche die Einheit Angriffe ausführen kann. Der Mittelpunkt des Kreises ist immer die Position der Einheit.

Das Schadensattribut kann durch eine Menge an möglichen Zieltypen $tt(u_i)$ erweitert werden. Eine Einheiten kann zum Beispiel lediglich Bodeneinheiten angreifen, während eine andere nur Lufteinheiten angreifen kann. Daher kann die Menge der angreifbaren Einheiten $tu(u_i)$ definiert werden als Menge aller gegnerischen Einheiten EU für die gilt: $\{x \in EU : t(x) \in tt(u_i)\}$. Einige Einheiten verfügen außerdem über ein Attribut Bonusschaden $bd^a(u_i)$, welches auf den Attributen $a(u_i)$ der attackierten Einheit beruht. Die Angriffe von Einheiten können zusätzlich verbessert werden, indem man bei den Einheiten die Verbesserungen ausbildet. Für jede Angriffsverbesserung erhält die Einheit einen Bonus von $d(u_i) \div 10$ - jedoch mindestens 1 - auf ihren Schaden.

Viele Einheiten sind in der Lage Fähigkeiten $ab(u_i)$ zu nutzen. Die Fähigkeiten unterscheiden sich in ihren Effekten und Nutzungsmöglichkeiten. Einige können aktiv und manuell vom Spieler eingesetzt werden, andere sind passiv oder werden von bestimmten Konditionen im Spiel ausgelöst. Zauber sind eine besondere Form der Fähigkeiten und kosten die Einheiten Energie. Obwohl die intelligente Nutzung von Fähigkeiten und Zaubern in Mensch-gegen-Mensch Gefechten eine signifikante Rolle spielen, werden sie in dieser Arbeit vernachlässigt, da sie den simulierten Armeekommandeuren die Fähigkeit des automatisierten Lernens abverlangen würden.

Eine Einheiten-Beschreibung anhand der StarCraft II Einheit Immortal würde demnach wie folgt aussehen: Ein *Immortal* (folgend kurz I) bekommt die Attribute $a(I) = \{Gepanzert, Mechanisch\}$ zugewiesen. Die Einheit startet bei Zeitschritt 0 mit $hp^0(I) = 200$ Lebenspunkten und erhält eine Bewegungsrate von $mr(I) = 3,15$, sowie eine Sichtweite von $sr(I) = 9$. Der Schaden liegt bei $d(I) = 20$ und die Abklingzeit zwischen jedem Angriff beträgt $cd(I) = 1,04$ Sekunden. Mit seiner Waffe kann der Immortal $wr(I) = 6$ weit angreifen. Die Menge der möglichen Zieltypen ist beim Immortal $tt(I) = \{Boden\}$. Immortal erhalten einen Bonus gegen gepan-

zerte Einheiten, dieser ist spezifiziert als $bd^{Gepanzert}(I) = 30$.

1.2 Kontributionen

In dieser Bachelorarbeit werden die betreffenden theoretischen Grundlagen für diese Problemstellung in Sektion 2 zusammengetragen. Zudem liefert Sektion 1.1 eine formalisierte Darstellung der Spielumgebung von StarCraft II. Verwandte Arbeiten wurden auf mögliche Beiträge zu diesem Thema analysiert und in Sektion 3 zusammengefasst. Es wurde ein Framework geschaffen, welches basierend auf dem in StarCraft II enthaltenen Map-Editor die Generierung beliebig vieler Trainingsdaten sowohl in Form von Spieldaten in Matrix-Repräsentation als auch als Screenshots in gängigen Bildformaten ermöglicht. Das Framework wird in Sektion 5.1 genauer beschrieben. Zudem wurden bestehende Architekturen für die fallspezifischen Eingabematrizen adaptiert. Die genutzten Architekturen werden in Sektion 4 genauer beschrieben und in Sektion 5.4 analysiert und verglichen.

2 Theoretischer Hintergrund

In dieser Sektion werden wichtige Grundlagen definiert und anhand von Beispielen erläutert. Zunächst folgt eine Einführung in Artificial Neural Networks (zu Deutsch: *Künstliche neuronale Netze*) in Sektion 2.1. Daraufhin werden Convolutional Neural Networks (zu Deutsch etwa:*Faltende neuronale Netze*) und ihre allgemeinen Bestandteile näher erklärt (Sektion 2.2).

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) sind im Machine Learning (zu Deutsch: *Maschinelles Lernen*) angewandte Techniken, die von biologischen neuronalen Netzen inspiriert sind.

Machine Learning ist ein Bereich der Informatik, in dem man Probleme durch Algorithmen lösen möchte, welche keinen statisch programmierten Lösungsweg benötigen. Das Ziel von Machine Learning ist das Konstruieren von Algorithmen, die anhand von Datensätzen Schlussfolgerungen ziehen können und bei Eingabe unbekannter Daten diese Schlussfolgerungen zur Vorhersage von Ergebnissen generalisieren können.

ANNs können als gerichtete Graphen verstanden werden. Sie bestehen aus einer Menge an Knoten mit einer Menge an Kanten als Verbindungen zwischen den Knoten, welche die Aktivierungen der Knoten transportieren. Bei ANNs wird zwischen verschiedenen Strukturen unterschieden. In dieser Arbeit werden ausschließlich sogenannte Feed-Forward Neural Networks (zu deutsch etwa: *vorwärts gekoppelte neuronale Netze*) genutzt. In Feed-Forward Neural Networks sind Knoten in Layern (zu Deutsch: *Schichten*) angeordnet, welche untereinander kommunizieren. Ein Feed-Forward Neural Network ist immer ein gerichteter azyklischer Graph in dem die Signale der Neuronen in Kantenrichtung weitergegeben werden.

Neuronen in ANNs haben einen Bias und jede Verbindung zu anderen Neuronen ist mit einem Gewicht belegt. Die Werte beider Eigenschaften sind variabel und werden im Verlauf des Lernprozesses immer wieder angepasst. Außerdem verfügt jedes Neuron über eine Aktivierungsfunktion, welche die Ausgabe des Neurons abhängig vom Eingabewert definiert.

Abbildung 1 zeigt eine einfache Topologie eines ANNs. Eine Eingabe wird vom Input Layer (zu Deutsch: *Eingabeschicht*) in den Hidden Layer (zu Deutsch: *Versteckte Schicht*) gegeben und von diesem in die letzte Schicht, welche als Output Layer (zu Deutsch: *Ausgabeschicht*) funktioniert weitergereicht. Der Hidden Layer kann aus mehreren unterschiedlichen Layern bestehen und dient lediglich als Abstraktion aller Schichten zwischen Input und Output Layer.

Neural Networks (zu Deutsch: *Neuronale Netze*) können als mathematische Modelle verstanden werden, welche eine Funktion $f : X \rightarrow Y$ definieren. In einem Neural Network wendet jedes Neuron eine Funktion auf seine Eingangswerte an, bevor es diese zur nächsten Schicht sendet. Somit kann die Funktion eines Neural

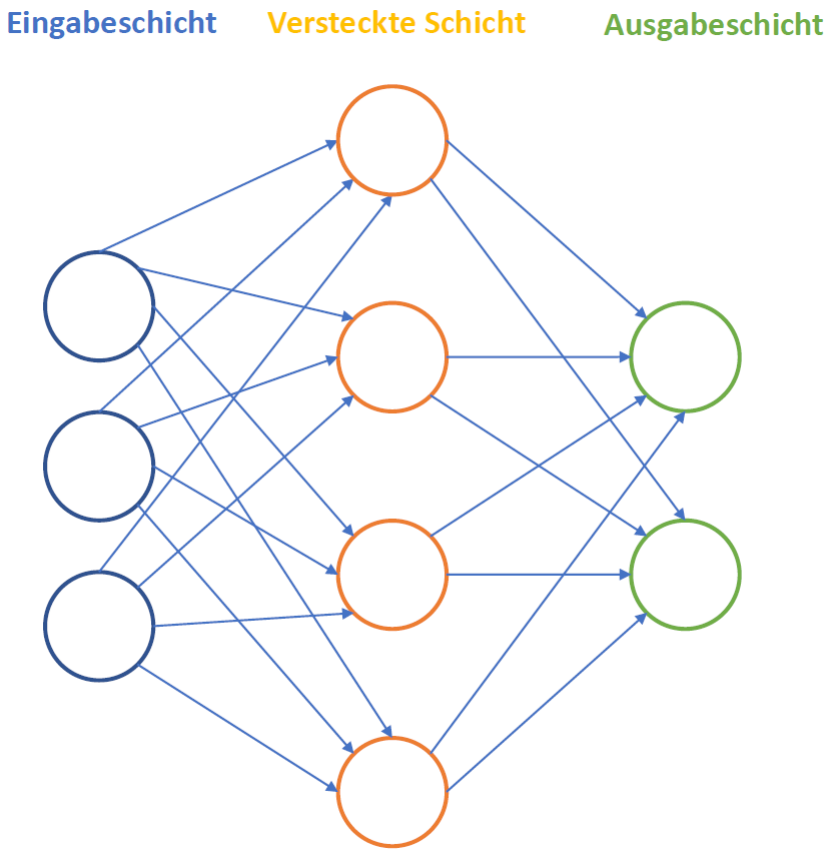

Abbildung 1: Topologie eines einfachen ANNs.

Networks als eine Komposition aus jeder sich im Networks befindlichen Funktion definiert werden.

Bezogen auf Abbildung 2, sieht die zusammengesetzte Funktion des Networks wie folgt aus:

$$f(g_1(h_1(x), h_2(x)), g_2(h_2(x), h_3(x))) \tag{1}$$

Fully-Connected layer Jeder Layer in einem klassischen ANN ist ein Fully-Connected Layer (zu Deutsch etwa: *vollständig verbundene Schicht*). Neuronen in Fully-Connected Layern besitzen Verbindungen zu allen Neuronen des vorrangigen Layers. Jedes Neuron berechnet einen Ausgabewert $F(x)$ unter Verwendung des Eingabewertes **x**, der Gewichte **W**, die den Verbindungen zu dem vorherigen Layer zugewiesen

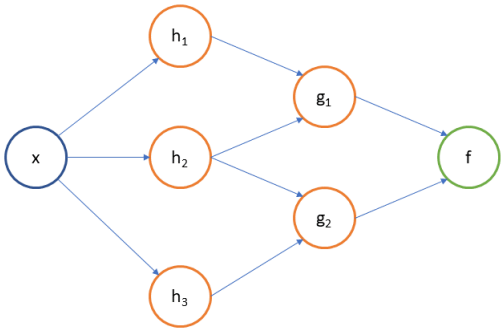

Abbildung 2: Grafische Darstellung der zusammengesetzten Funktion f aus Gleichung 1.

sind, und eines Bias **b**, der dem entsprechenden Neuron zugewiesen ist.

$$\begin{bmatrix} w_{11} & \dots & w_{1m} \\ . & & . \\ . & . & . \\ . & & . \\ w_{n1} & \dots & w_{nm} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_m \end{bmatrix} + \begin{bmatrix} b_1 \\ . \\ . \\ b_n \end{bmatrix} = \begin{bmatrix} y_1 \\ . \\ . \\ y_n \end{bmatrix} \tag{2}$$

Gleichung 2 zeigt die allgemeine Formel nach welcher der Ausgabewert eines Fully-Connected Layers berechnet wird. n steht für die Anzahl der Neuronen im Layer und m ist die Anzahl der Eingabewerte des vorherigen Layers. Die Gleichung kann auch wie folgt in Vektornotation formuliert werden:

$$F(\mathbf{x}) = (\mathbf{W} \cdot \mathbf{x}) + \mathbf{b} \tag{3}$$

In Gleichung 3 sind **W** und **b** lernbare Parameter, welche vom Optimierungsalgorithmus abgepasst werden können. Sie stehen in der Gleichung für die Gewichte (**W**) und den Bias (**b**) des entsprechenden Layers.

Rectified Linear Units Layer In ANNs finden eine Vielzahl von Aktivierungsfunktionen Anwendung. Im Folgenden wir die ReLU-Funktion vorgestellt, da sie in den Modellen, die in dieser Arbeit implementiert werden genutzt wird. Sie ist wie folgt definiert: $relu(x) = max(0, x)$. Die Aktivierungsfunktion wird nach der Verarbeitung eingehender Werte im Neuron auf das Ergebnis angewandt und entfernt negative Werte aus dem Ausgabevektor des Neurons, bevor der Vektor an den nächsten Layer weitergereicht wird [23].

Normalization Layer Das Normalization Layer (zu Deutsch: *Normalisierungsschicht*) wird zur Normalisierung der Aktivierungen des vorherigen Layers genutzt und dient unter anderem der Verhinderung von *Overfitting* (zu Deutsch: Überanpassung). Overfitting tritt dann auf wenn ein Algorithmus die Trainingsdaten zu gut lernt und nicht mehr auf Grundlage dieser die Eingaben generalisieren kann.

Nachfolgend wird ausschließlich Batch Normalization (BN) genutzt, da es sowohl für Fully-Connected Layer als auch für Convolutional Layer genutzt werden kann und die Trainingsgeschwindigkeit des Netzwerks beschleunigt [7]. Angewandt auf einen Mini-Batch B mit den Eingaben $B = \{x_1...m\}$ berechnet Batch Normalization für eine n-dimensionale Eingabe $x = (x^{(1)}...x^{(n)}) \in B$ zunächst den Mittelwert und die Varianz des Batches:

$$mean(B) = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{4}$$

$$var^2(B) = \frac{1}{m}\sum_{i=1}^{m}(x_i - mean(B))^2 \tag{5}$$

Mit beiden Werten wird die Eingabe dann Normalisiert, wobei ϵ als Konstante zur Varianz addiert wird um numerische Stabilität sicher zu stellen:

$$\hat{x}_i = \frac{x_i - mean(B)}{\sqrt{var^2(B) + \epsilon}} \tag{6}$$

Zuletzt wird die Eingabe skaliert und verschoben. γ und β sind lernbare Parameter, welche im Batch Normalization Layer eingeführt werden:

$$y_i = \gamma \times \hat{x}_i + \beta \tag{7}$$

Loss Function Die Loss Function (zu Deutsch: *Verlustfunktion*) evaluiert wie genau ein Neural Network die ground-truth (zu Deutsch etwa: Grundwahrheit) der Trainingsdaten vorhersagt. Ihre Aufgabe besteht darin den Abstand zwischen der ground-truth **(x, y)** einer Eingabe **x** und der Vorhersage des ANNs $\hat{y}$. Neural Networks versuchen Gewichte und Biase so anzupassen, dass dieser Wert minimiert wird. Ein Beispiel für eine Loss Function z wäre [23]:

$$z = \frac{1}{2}\|\mathbf{(x,\ y)} - \hat{\mathbf{y}}\|^2 \tag{8}$$

Softmax-Funktion Die Softmax-Funktion ist eine Variante der logisitschen Funktion. Die Funktion erhält einen Vektor **z** mit reellen Werten und einer beliebigen Dimension K und verrechnet dessen Werte, so dass sich jeder Wert des Vektors **z** im Bereich (0, 1] befindet und die Summe aller Werte 1 ergibt. Die Softmax-Funktion wird nach folgender Formel berechnet [3]:

$$softmax(\mathbf{z})_j = \frac{\mathrm{e}^{z_j}}{\sum_{k=1}^{K} \mathrm{e}^{z_k}} \tag{9}$$

Eine vollständige Funktion für das in Abbildung 1 gezeigte ANN wird in Gleichung 10 ausformuliert.

$$f(\mathbf{x}) = softmax(\mathbf{W_2} \cdot relu(\mathbf{W_1} \cdot \mathbf{x} + \mathbf{b_1}) + \mathbf{b_2}) \tag{10}$$

Der Eingabevektor $\mathbf{x}$ ist in $\mathbb{R}^3$. Zuerst wird $\mathbf{x}$ vom Hidden Layer verarbeitet. In diesem Beispiel ist der Hidden Layer Fully-Connected, so dass die Formel für Fully-Connected Layer 2 mit $\mathbf{W_1}$ und $\mathbf{b_1}$ angewendet werden kann. $\mathbf{W_1}$ ist eine 4×3 Matrize und $\mathbf{b_1}$ ist in $\mathbb{R}^4$. Das Resultat wird mit einer Aktivierungsfunktion – in diesem Fall der ReLU-Funktion – verrechnet und hat danach die Dimension 4×1. Daraufhin wird das Resultat an den Output Layer weitergegeben, welcher auch ein Fully-Connected Layer ist. Auch hier wird Gleichung 2 für Fully-Connected mit $\mathbf{W_2}$ als 2×4 Matrix und $\mathbf{b_2}$ als Vektor in $\mathbb{R}^2$ angewandt. Zuletzt wird die Softmax-Funktion auf das Ergebnis des Output Layers angewandt. Die Dimension des Ausgabevektors ist 2×1.

2.2 Convolutional Neural Networks

Ein Convolutional Neural Network (CNN) ist eine Spezialisierung des allgemeineren ANN-Begriffs. Es findet häufig Anwendung um Aufgaben im Zusammenhang mit Bildern, wie zum Beispiel Bildklassifikationen, zu lösen. Im Vergleich zu ANNs benötigen CNNs im Allgemeinen weniger Gewichte, durch ihre Eigenschaft der lokalen Konnektivität.

Ähnlich wie ANNs bestehen CNNs jeweils aus einem Input und einem Output Layer. Zwischen diesen beiden Layern befinden sich ein oder mehrere Hidden Layer, welche die Informationen aus den Eingabedaten verarbeiten. Die Hidden Layer können aus verschiedenen Layer-Typen zusammengesetzt werden, welche beliebig untereinander kombiniert werden können. Einige dieser Layer sind zum Beispiel: Convolutional, Pooling, Fully-Connected und Normalization Layer. In der Praxis sind ideale Layerzusammenstellungen stark an die Problemstellung gebunden und werden empirisch bestimmt.

Tensor Ein Tensor wird im Kontext von Python und Tensorflow als ein mehrdimensionales Array verstanden. In der Arbeit werden drei- und vierdimensionale Tensoren verwendet. Abbildung 2.2 zeigt eine visuelle Erklärung der Begriffe, die im Bezug auf Tensoren folgend genutzt werden. Breite beschreibt die Ausdehnung des Tensors auf der x-Achse, Höhe die Ausdehnung auf der y-Achse und die Tiefe die Ausdehnung auf der z-Achse.

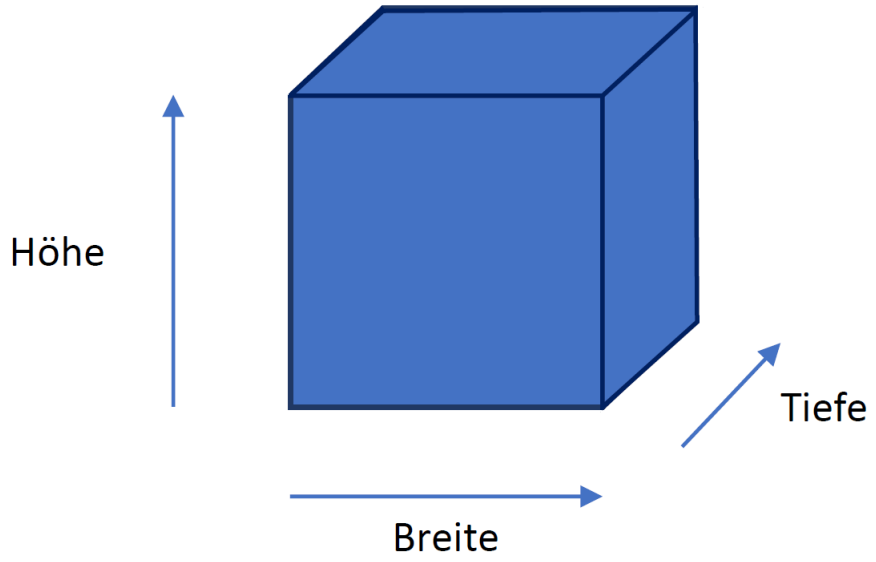

Abbildung 3: Visualisierung eines Tensors des 3. Ranges mit Zuordnung der Begrifflichkeiten.

Convolutional Layer Ein Convolutional Layer ist einer der Kernbestandteile eines CNNs. Der Layer verfügt über eine bestimmte Anzahl an Filtern, welche durch einen Tensor des selben Ranges wie die Eingabedaten bemessen werden. Filter sind durch ihre Reception Fields (zu Deutsch: rezeptive Felder) charakterisiert, welche die Größe der zeitgleich untersuchten Areale in den Eingabedaten bestimmen [11]. Ein Beispiel für das Reception Field eines Neurons könnte eine $M \times N$ große Fläche sein, welche kleinere oder gleich große Abmessungen haben sollte wie die Dimensionen der Eingabedaten. Das Reception Field spannt somit eine Matrix auf in der die Gewichte jedes Filters enthalten sind. In Abbildung 2.2 spannt das Reception Field für einen Filter eine 2×2 Matrix, deren Gewichte alle 1 sind. Während der Filter die Eingabe verarbeitet, wird er über die Eingabematrix geschoben und multipliziert die Werte in dem untersuchten Bereich mit den Werten seines Filters. Die Produkte werden zu einem einzigen Werte aufsummiert, welcher dann den Wert für das Areal widerspiegelt. Wenn der Prozess endet, ist das Ergebnis eine etwas kleinere Matrix, in welcher die aufsummierten Ergebnisse für jede mögliche Position des Filters in den Eingabedaten erfasst sind. Sind in einem Layer mehrere Filter, so verfügt jeder Filter über eine individuelle Gewichtsmatrix mit gleichen Dimensionen. Jeder dieser Filter resultiert jeweils in einer Ausgabematrix. Bei zwei-dimensionalen $H \times W$ Eingabedaten – mit H = Höhe und W = Breite – werden die resultierenden Ausgabematrizen in der dritten Dimension D (Tiefe) angeordnet, wobei D = die Anzahl der verwendeten Filter ist. Die Ausgabe entspricht somit einem Tensor des Ranges 3.

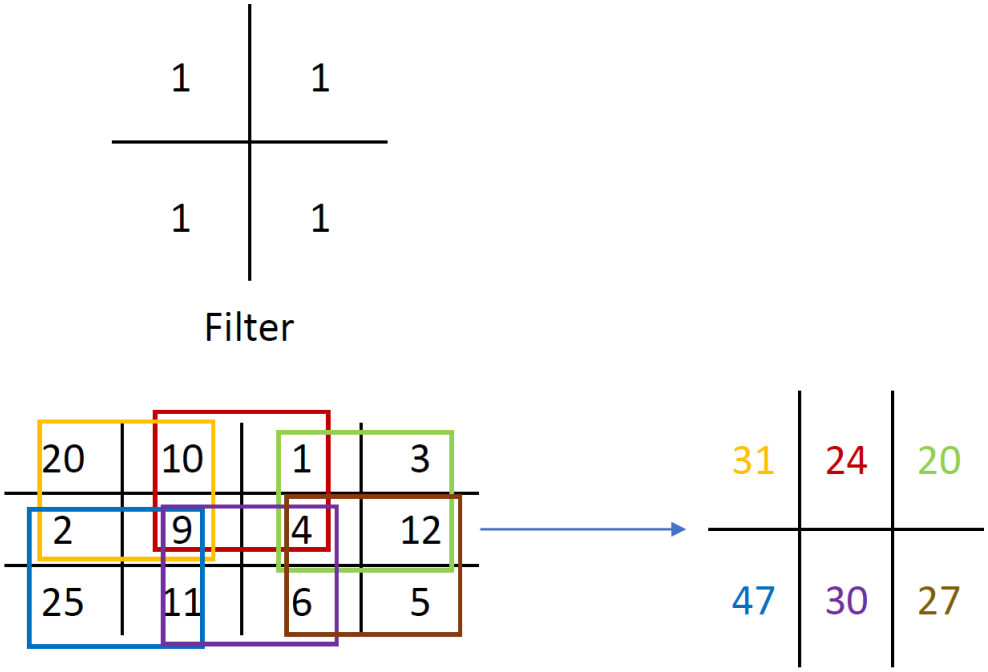

Abbildung 4: Ablauf einer Convolution mit einem 2×2 Filter und einer 3×4 großen Eingabematrix.

Nimmt man eine Matrix der Dimension 10×10 und führt auf dieser eine Convolution mit 5 Filtern und einem 5×5 Reception Field aus, so ist die Ausgabe des Convolution Layers ein Tensor des dritten Ranges mit Dimensionen $6 \times 6 \times 5$.

Im Allgemeinen resultiert die Eingabe eines Tensors mit Bemessung $H^l \times W^l \times D^l$ und einem Filter der Größe $H \times W \times D^l \times D$ in einem Ausgabetensor der Dimension $(H^l - H + 1) \times (W^l - W + 1) \times D$. In dieser Notation verweist l auf den vorherigen Layer [23].

Sollte eine Ausgabetensor mit den selben Dimensionen wie der Eingabetensor benötigt werden, so ist es möglich *padding* (zu Deutsch in etwa: *Polsterung*) zu nutzen, um der Matrix Spalten und Zeilen nach Bedarf hinzuzufügen. Um die Dimensionen des Eingabetensors zu erhalten, müssen $\lfloor \frac{H-1}{2} \rfloor$ Zeilen oberhalb der ersten Zeile und $\lfloor \frac{H}{2} \rfloor$ unterhalb der letzten Zeile hinzugefügt werden. Außerdem müssen $\lfloor \frac{W-1}{2} \rfloor$ Spalten vor der ersten Spalte und $\lfloor \frac{W}{2} \rfloor$ Spalten nach der letzten Spalte eingefügt werden. Der Notation der vorangegangenen Absätze folgend, beziehen sich W und H auf die vertikale und horizontale Dimension des Filters.

Zusätzlich kann ein *stride*-Parameter (zu Deutsch: *Schritt-Parameter*) s definiert werden. Dieser Parameter bestimmt um wie viele Stellen der Filter nach jeder Rechnung auf dem Tensor verschoben wird. Wenn jede mögliche Position auf dem Tensor evaluiert werden soll, beträgt der Wert des stride-Parameters $s = 1$. Für alle $s > 1$ wird der Filter $s - 1$ Positionen entlang beider Axen überspringen.

Der Vorgang des Abgehens der Eingabewerte kann durch eine mathematische Formel [12] beschrieben werden. In der Formel ist der stride-Parameter $s = 1$ und es findet kein padding statt. H^{l+1}, W^{l+1} und D^{l+1} beziehen sich auf Höhe, Breite und Tiefe des Ausgabetensors.

$$y \in \mathbb{R}^{H^{l+1} \times W^{l+1} \times D^{l+1}} \tag{11}$$

mit $H^{l+1} = H^l - H + 1$, $W^{l+1} = W^l - W + 1$, and $D^{l+1} = D$ [23].

$$y_{i',j',di} = \sum_{i=0}^{H} \sum_{j=0}^{W} \sum_{d=0}^{D^l} f_{i,j,d,di} \cdot x^l_{i'+i,j'+j,d} \tag{12}$$

f ist ein Tensor des vierten Ranges mit den Indizes $0 \leq i < H, 0 \leq j < W, 0 \leq d^l < D^l und 0 \leq d < D$ und stellt die Menge der Filter und deren Gewichte dar. x^l ist die Ausgabe des vorherigen Layers. Die Gleichung 12 wird über die komplette Tiefe D^l der Eingabe wiederholt ($0 \leq di \leq D^l$). Sie wird lediglich auf Positionen (i', j') angewandt, welche die Bedingungen $0 \leq i' < H^{l+1}$ und $0 \leq j' < W^{l+1}$ erfüllen [23].

Pooling layers Pooling Layer (zu Deutsch etwa: *bündelnde Schicht*) werden auch als Downsampling Layer bezeichnet. Ihre Aufgabe besteht darin die Ausgabewerte des vorherigen Layers in einen Wert zu kombinieren und somit die Dimensionen der weitergereichten Daten zu reduzieren. Abbildung 5 zeigt eine bildliche Erklärung des Maxpool Layers.

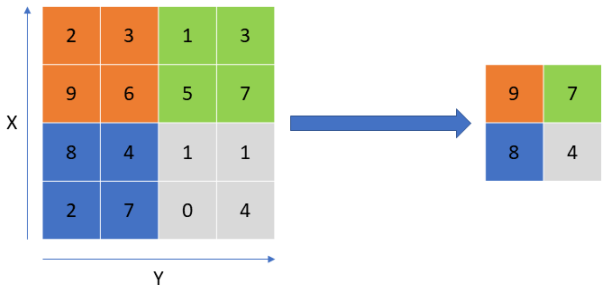

Abbildung 5: Einfache Veranschaulichung des Maxpool Layers mit 2×2 stride-Parameter.

Der Maxpool Layer fußt auf der Idee, dass die exakte Position eines Merkmals nicht so wichtig ist, wie seine Position in Relation zu anderen Merkmalen. Abhängig von der gestellten Aufgabe könnte die Lage des Merkmals auch gar nicht relevant sein und die wichtige Information, die dieses Merkmal enthält ist lediglich seine Existenz. Ein Pooling Layer reduziert die Dimensionen des Eingabetensors in Abhängigkeit von seinen Parametern. So reduziert ein 2×2 Pooling Layer eine 8×8 dimensionierte Eingabe auf die Dimension 4×4, da es die Eingabematrix in sich

nicht überlappende 2×2 Quadrate unterteilt und für jedes Quadrat einen Wert errechnet, welcher in die Ausgabematrrix einfließt. Die Inhalte der Ausgabe hängen von dem Typ des Layers ab, so errechnet ein Maxpooling Layer immer die größten Werte beobachteten Bereichs, ein Average Pooling Layer wiederum berechnet den Durchschnitt aller Werte in dem relevanten Bereich.

Der Pooling Layer arbeitet auf der kompletten Tiefe der Eingabe und reduziert diese daher nicht, sondern berechnet jede Matrix unabhängig von den anderen. Die Eingabe wird somit lediglich in Breite und Höhe reduziert. Obwohl es auch andere Varianten der Pooling Layer gibt, wie zum Beispiel Average Pooling oder L2-norm Pooling, wird hauptsächlich Maxpooling in den bekanntesten Architekturen verwendet.

Der Notation folgend resultiert ein Pooling Layer der Größe $(H \times W)$ in einer Ausgabe $H^{l+1} \times W^{l+1} \times D^{l+1}$. Unter der Annahme, dass H und W, H^l und W^l teilen, gilt [23]:

$$H^{l+1} = \frac{H^l}{H}, W^{l+1} = \frac{W^l}{W}, D^{l+1} = D^l. \tag{13}$$

Ein Maxpooling Layer kann wie folgt definiert werden:

$$y_{i',j',d} = \max_{0 \leq i < H, 0 \leq j < W} x^l_{i' \cdot H + i, j' \times W + j, d} \tag{14}$$

es gilt $0 \leq i' < H^{l+1}, 0 \leq j' < W^{l+1}$, und $0 \leq d < D^{l+1} = D^l$.

3 Verwandte Arbeiten

Die Computerspiele StarCraft und StarCraft II waren schon mehrfach Anschauungsobjekt im Bezug auf Schlachtensimulation und der Vorhersage von Gefechtsausgängen. Robertson und Watson [12] fasst Literatur zum Thema StarCraft zusammen und arbeitet Forschungsfelder oder Aufgaben heraus, welche einen positiven Einfluss auf die Forschung im Bereich der künstlichen Intelligenz (KI) haben könnten. Robertson kommt zu dem Schluss, dass Machine Learning bei der Verbesserung bestehender KIs an Einfluss gewinnen wird und dass ein klarer Trend zu Machine-Learning-Verfahren zu erkennen ist.

Verwandte Arbeiten existieren sowohl für die Spielumgebung von StarCraft [1, 2, 14, 16, 17, 22], als auch für die Spielumgebung von StarCraft II [5, 13]. Die Arbeiten, die sich mit der Vorhersage von Gefechts-Ausgängen beschäftigen nutzen vor allem simulationsbasierte Ansätze [1, 2, 5] um optimale Handlungsfolgen vorherzusagen und diese im Wettkampf gegen existierende Skript zu testen, oder verwenden Machine Learning um die Ergebnisse der Gefechte vorherzusagen [14, 16, 17].

Stanescu u. a. [17] stellt ein Modell vor, auf dessen Basis ein Lernalgorithmus die offensiven und defensiven Feature Values (zu Deutsch: *Eigenschaftswerte*) jeder Einheit erlernt. Feature Values können zum Beispiel in der Offensive der Schaden einer Einheit und in der Defensive die Lebenspunkte sein. Das Paper nennt als weiteres Beispiel noch Angriffsreichweite als offensives und Bewegungsrate als defensives Feature. Diese Feature Values werden nachfolgend für jede Einheit der Spieler berechnet. Anschließend werden die Feature Values beider Spieler aggregiert:

$$Agg = (O_A/D_B - O_B/D_A) \tag{15}$$

Mit O als offensiven und D als defensiven Feature Value. Durch eine Sigmoid-Funktion wird anschließend die Wahrscheinlichkeit, das Spieler A Spieler B schlägt bestimmt. Da sich die Arbeit auf die Zusammensetzung von Armeen beschränkt, vernachlässigt es einige Faktoren, welche bei einer ganzheitlichen Betrachtung von Gefechten eine Rolle spielen. So wird Terrain und die Vor- und Nachteile die es mit sich bringt, außer Acht gelassen, fliegende Einheiten, sowie Einheiten, die Fähigkeiten nutzen werden ignoriert und Einheitenverbesserungen sind in dem Modell auch nicht existent.

Stanescu et al. Stanescu, Barriga und Buro [16] bauen ihren Vorhersage-Algorithmus auf dem Gesetz von Lanchester (auch Lanchester's Square Law genannt) auf, welches er in seinem Buch *Aircraft in Warfare – The Dawn of the Fourth Arm* von 1916 formuliert. Hierbei handelt es sich um ein Modell, welches versucht die Verluste in militärischen Gefechtssituationen einzuschätzen. Basierend auf der Einheitenzusammensetzung wird die *Armee-Effektivität* α_{avg} für beide Spieler errechnet. In der Gleichung wird der Wert für Armee A berechnet. A bezieht sich auf die Anzahl der Einheiten und α_i ist die Effektivität einer Einheit und zeitgleich die lernbare Variable, die im Verlauf des Trainings immer weiter angepasst wird. α_i wird für jede Einheit zu Beginn des Trainings mit dem Produkt aus Schaden und Lebenspunkten

der Einheit initialisiert.

$$\alpha_{avg} = \frac{\sum_{i=1}^{A} alpha_i}{A} \tag{16}$$

Der Algorithmus ersetzt unter Turnierbedingungen den simulationsbasierten Entscheidungsalgorithmus des UAlbertaBots im Bezug auf die Entscheidung ein Gefecht einzugehen oder sich zurückzuziehen. Das Modell zieht Interaktionen zwischen Einheiten, wie z.B. das Heilen durch Medics nicht in Betracht und ignoriert Einheitenpositionierung.

Sánchez-Ruiz [14] beschränkt sich auf die Vorhersage der Gefechte und vergleicht die Präzision verschiedenener Machine-Learning-Algorithmen. In den Experimenten kommen die Klassifizierungs-Verfahren Linear Discriminant Analysis (LDA, zu Deutsch: Lineare Diskriminanzanalyse), Quadratic Discriminant Analysis (QDA, zu Deutsch: Quadratische Diskriminanzanalyse) Support Vector Machines (SVM, zu Deutsch etwa: Stützvektormaschine), k-Nearest Neighbour (kNN, zu Deutsch: k nächste Nachbarn) und Weighted k-Nearest Neighbour (kkNN, zu Deutsch: gewichtete k nächste Nachbarn) zur Anwendung. Es werden im Allgemeinen vier Feature (zu Deutsch: *Merkmale*) zur Evaluierung definiert: Die Anzahl der Einheiten, ihr durchschnittliches Leben und ihre relative Position, sowie Distanz zur gegnerischen Armee. Die relative Position wird angenähert, indem um jede Armee ein minimales Rechteck gezogen wird, dessen Fläche bildet steht stellvertretend für die Streuung der Einheiten. Die Distanz zur feindlichen Armee wird bestimmt durch die Entfernung der Mittelpunkte beider Rechtecke. Die Vordefinierten Features beziehen sich nicht auf Merkmale der einzelnen Einheiten, sondern stellen die Armeen in den Fokus. Es werden daher einheitenspezifische Merkmale wie Schadenswerte der Einheiten, Rüstung oder Reichweite ignoriert.

Alle drei Arbeiten erreichen mit ihren Algorithmen die 90%-Marke in der Vorhersage von Gefechtsausgängen, nutzen die Ergebnisse allerdings für unterschiedliche Weiterverwendungen. Stanescu u. a. [17] versucht weiterführend für jede Einheitenzusammensetzung aus 2 unterschiedlichen Typen, die Armee zu finden, welche diese Zusammensetzung schlägt. Stanescu, Barriga und Buro [16] vergleicht das Entscheidungsverhalten des Algorithmus mit gängigen Implementationen in StarCraftBots und Sánchez-Ruiz [14] evaluiert die Änderung der Accuracy im Verlauf des Gefechts und erreicht zwar mit allen vorgestellten Klassifizierungsverfahren die 90%-Marke, jedoch lediglich kkNN liegt auch kurz nach Beginn des Gefechtes bei 90%, während die restlichen Verfahren erst nach ca. 60% der Gefechts-Zeit eine Accuracy von 90% erreichen. Für einen Entscheidungsalgorithmus wäre die Arbeit von Stanescu, Barriga und Buro [16] die beste Wahl, weil sie als einziges bereits geschädigte Einheiten berücksichtigen kann und mit der vorgestellten Methode eine Accuracy von 93% erreicht.

Churchill, Saffidine und Buro [2] entwickelt einen Such-Algorithmus, der basierend auf einen Spiel-Zustand optimale Entscheidungen für eine KI treffen soll. Da die Suche nach einer optimalen Entscheidung oftmals langwierig sein kann, wird

die maximale Dauer der Evaluierung in der aktiven Anwendung in einer KI auf 5ms beschränkt. Trotz dieser Einschränkung schafft es der Algorithmus einfache KI-Skripte zu schlagen. Das Modell des Algorithmus modelliert den Zustand des Spiels und jeder einzelnen Einheit und errechnet für jeden Zeitschritt eine Menge an Legal Moves (zu Deutsch: *legale/erlaubte Züge*). Das Modell unterliegt einigen Limitierungen, die seine Nutzbarkeit einschränken. So werden Lebenspunkt-Regeneration, Einheitenkollision und Reisezeit von Projektilen nicht in die Evaluierung einbezogen.

Portfolio greedy search [1] ist ein Such-Algorithmus, der – ähnlich wie Churchill, Saffidine und Buro [2] – versucht Handlungsfolgen zu konstruieren, welche auf Gefechts-Szenarien mit bis zu 50 Einheiten pro Seite angewandt werden können. Im Zuge der Entwicklung konnten Churchill und Buro ebenfalls SparCraft vorstellen. Ein System mit dem Gefechte abstrakt modelliert werden können.

Wender und Watson haben vorgestellt, wie Reinforcment Learning (zu Deutsch: *Bestärkendes Lernen*) in StarCraft Anwendung finden kann [22]. Der von ihnen vorgestellte Algorithmus sollte das Mikromanagement der Einheiten erlernen und schlussendlich geskriptete Abfolgen ablösen. In ihrem Paper zeigten sie, dass Reinforcement Learning ein geeignetes Verfahren für diese Domäne ist.

Helmke, Kreymer und Wiegand [5] entwickelt Näherungsmodelle basierend auf eingespeisten Grundkonstellationen für das Spiel StarCraft II. In dem Paper werden 4 Annäherungsmethoden vorgestellt, welche unterschiedliche Eigenschaften der beiden Armeen in Betracht ziehen. Es werden Lebenspunkte, Schaden pro Sekunde, Fernkampfschaden, Rüstung, Attribute, Boni und Bonus Schaden pro Sekunde als Eigenschaften einer Einheit festgelegt und in unterschiedlichen Zusammensetzungen gemeinsam evaluiert. Einheiten-Fähigkeiten, sowie Einheitenverbesserungen vernachlässigt werden vernachlässigt. Da einige Einheiten nur Einheiten eines speziellen Typs angreifen können führt das zu Unschärfen in der Vorhersage. Des weiteren wird die Position der Einheiten nicht mit einbezogen.

Samvelyan u. a. [13] befasst sich mit den Problemen von Multi-Agent Reinforcement Learning (MARL). In dem Paper werden sogenannte Benchmark-Probleme für MARL eingeführt. Außerdem veröffentlichen die Verfasser mit *PyMARL* ihr Framework (zu Deutsch etwa: *Gerüst*) für die Erstellung und Analyse von tiefen MARL-Algorithmen.

Kilmer [9] diskutiert in seinem Artikel die Verwendung von ANNs zur Vorhersage von Gefechtsausgängen in Militärszenarien. Es werden mehrere Anwendungsfälle skizziert in denen ANNs Beiträge leisten können, wie u.a. das Ersetzen bestehender Simulationsverfahren durch ANN-basierte Modelle, oder aber auch das Optimieren bestehender Verfahren durch Erkenntnisse aus ANN-Simulationen.

4 Methodologie

In diesem Abschnitt werden die verwendeten Architekturen vorgestellt. Alle in dieser Bachelorarbeit verwendeten Architekturen, wurden von ihren Entwicklern für eine Anwendung im Bereich der zweidimensionalen Bildklassifizierung implementiert. Die Eingabedaten liegen in diesem Fall allerdings als vierdimensionaler Tensor vor. Dadurch mussten in den problemspezifischen Implementationen dieser Arbeit Anpassungen an Filtereinstellungen vorgenommen werden, ohne die grundlegende Struktur der Architekturen zu verändern. In den vorgestellten Modellen wird die Tiefe in den Convolutional Layern ignoriert und jede der 8 Schichten wird einzeln von den Filtern bearbeitet, daher verfügen alle Pooling und Convolutional Filter über Dimension des Schemas [1,X,X], wobei gilt $X \in \mathbb{N}$, $X \geq 1$. Erst in den Fully-Connected Layern werden alle Schichten zusammen betrachtet.

Da die Architekturen über stark variierende Parameterzahlen verfügen und durch sehr tiefe Strukturen mitunter zu Umfangreich für die vorhandenen Rechenressourcen sind, wurden die Parameter einiger Architekturen angepasst, sodass alle Architekturen einen ähnlichen Umfang – gemessen an der Parameteranzahl – haben. Dieser Umstand schafft gleichzeitig eine bessere Vergleichbarkeit der Strukturen aller Architekturen, da Abweichungen durch wesentlich höhere Parameterzahlen ausgeschlossen werden können.

Viele gängige Implementationen der vorgestellten Architekturen setzen zudem Global Average Pooling vor den Fully-Connected Layern ein. Da in dieser Arbeit jedoch zwei gegeneinander arbeitende Faktoren in einem Bild miteinander verglichen werden, wird die Matrix lediglich auf 2×2 durch Average-Pooling reduziert. Infolgedessen erhält der Fully-Connected Layer 2 Werte pro Armee mit denen er weiter rechnen kann.

4.1 Problemstellung

Diese Arbeit nimmt sich der Frage an, wie gut gängige Bildklassifizierungsalgorithmen auf einer neuen Domäne funktionieren. Hierzu wird untersucht, ob die Architekturen die Ausgänge von StarCraft II Gefechten vorhersagen können.

Die Ausgabe des Netzwerkes stellt einen Vektor mit drei Elementen in Form eines Arrays dar. Die Werte des Vektors beziehen sich auf die Ausgänge der Gefechte, wobei der erste Werte für den Sieg des Teams Minerals steht, der zweite für einen Sieg der Vespene und der dritte Wert steht für ein Unentschieden beider Teams.

Die Eingabedaten unterscheiden sich stark von den ursprünglich für die Architekturen angedachten Eingabedaten. Während alle Architekturen in ihrem eigentlichen Aufgabenfeld Bilder als dreidimensionalen Tensor (Tiefe $\times$ Höhe $\times$ Breite) erhalten – wobei die Tiefe die 3 Farbkanäle enthält – muss für die Verarbeitung der Gefechte eine weitere Dimension hinzugefügt werden. Es entsteht ein Tensor des vierten Ranges, welcher für jeden der acht Feature Layer einen dreidimensionalen Tensor mit den Dimensionen 84×84 enthält. Somit wird jedes Gefecht von einem

Tensor mit den Dimensionen $8 \times 84 \times 84$ abgebildet. In dem Anwendungsfall dieser Arbeit werden Feature Layer genutzt, um jeweils einen bestimmten Aspekt des Spielzustandes zu Beginn der Simulation darzustellen. Die einzelnen Feature Layer enthalten folgende Informationen:

Der erste Layer enthält das *Player-Relative*-Feature (zu Deutsch etwa: Spieler bezogen). Dieser Layer enthält die Informationen dazu, welche Einheit zu dem betreffenden Spieler gehört. Die Werte geben dem Spieler an, ob es sich um eine freundliche, feindliche oder neutrale Einheit handelt.

Als zweiten Layer wird das *Unit-Type*-Feature (zu Deutsch: Einheiten-Typ) verwendet. Dieser Layer enthält die eindeutigen IDs der Einheiten-Typen, sodass alle Einheiten auf der Karte klar zu identifizieren sind.

Die Layer drei bis fünf enthalten die Lebenspunkte, Energie und Schildpunkte der Einheiten. Die Energie ist relevant um Fähigkeiten zu nutzen, da diese Energie kosten. Das ist in dieser Version allerdings noch nicht relevant, da die Einheiten in den Simulation lediglich Angreifen und auf den Einsatz von Fähigkeiten verzichten. Die Layer geben Aufschluss darüber wie die Lebens- und Schildpunkte auf der Karte verteilt sind.

Im sechsten Layer befinden sich die Informationen über die Spieler IDs. Während Player-Relative den Zustand der Einheiten aus Sicht eines Spieler wiedergibt, kann mit den Informationen aus dem *Player-ID* Feature Layer auch ein außenstehender Beobachter die Einheiten eindeutig zuordnen.

Der siebte Layer enthält die *Height Map* (zu Deutsch: Höhenkarte), welche die Struktur der Karte abbildet. Der Layer ist besonders relevant, bei Karten mit Höhenunterschieden, da fliegende Einheiten diese leicht überbrücken können. Bodeneinheiten müssen diese entweder umlaufen, oder haben genug Angriffsreichweite um trotzdem angreifen zu können.

Zuletzt wird als achter Layer die *Unit Density* (zu Deutsch: Einheitendichte) genutzt. Der Layer enthält für jeden Pixel der Karte Informationen darüber, wie viele Einheiten sich gleichzeitig darauf befinden.

Da diese acht Layer den Neural Networks ohne Kontext als ein Tensor übergeben werden, stellt sich die Frage, ob die Neural Networks mit dem unterschiedlichen Informationsgehalt der Layer arbeiten können und z.B. Abhängigkeiten ableiten können, oder bestimmten Layern besondere Wichtigkeit zuordnen. Zusätzlich stellt sich die Frage, ob Neural Network mit den Unit-Ids effektiv lernen können. Hierzu werden die Daten der Feature Layer gespiegelt, sodass Sieger- und Verlierer-Seite tauschen. Im Anschluss werden zweite identische Netze trainiert. Bei einem Versuch werden die Label mit rotiert, sodass das Netz die Frage beantworten muss, *Welche Spielfeldseite gewinnt*. In dem anderen Durchlauf bleiben die Label gleich, wodurch das Netz die Frage *Welcher Spieler gewinnt?* beantworten muss. Die einhergehende Vergrößerung der Gefechte, bringt gleichzeitig den Vorteil der größeren Datenmenge mit.

4.2 Baseline

Um die Effektivität der Architekturen zu überprüfen wurde eine Baseline implementiert, welche nach einer festen Implementierung die Gefechts-Ausgänge berechnet. Die Baseline ist eine eigene Implementation und folgt keinem existierenden Ansatz. Der Algorithmus soll die Gefechte anhand der verwendeten Einheiten möglichst akkurat vorhersagen und einen Vergleichswert für die betrachteten Architekturen bieten. Grundlegend werden Offensiv- und Defensivwerte für jede Armee errechnet und gegeneinander abgewogen.

Die Offensive Kampfkraft der Armeen wird durch einen *Powervalue* dargestellt. Der Powervalue errechnet sich aus dem Angriffswert folgend Damage-per-second (DPS, zu Deutsch: Schaden pro Sekunde) zusammen mit sämtlichen Boni, welche durch die unterschiedlichen Einheitenklassen hinzukommen. Die Berechnung des Powervalues einer einzelnen Einheit wird in Algorithmus 1 veranschaulicht. Die Prozedur Powervalue wird für jede Einheit einer Armee ausgeführt und erhält die Spezifika der Einheit wie auch die Spezifika der gegnerischen Armee als Eingabewerte. Der Bonusschaden *bonus* wird durch den Grundbonus, den die Einheit gegen einen speziellen Typ erhält und den Anteil der relevanten Einheiten in der gegnerischen Armee berechnet und auf den Grundschaden der Einheit addiert. Die Berechnung der Bonusschäden bringt keine nennenswerte Steigerung der Accuracy (zu Deutsch: Genauigkeit), wurde aber der Vollständigkeit halber implementiert.

Algorithmus 1 Berechnung des Powervalues für jede Einheit.

```
1:  procedure POWERVALUE(unit, enemyUnits)
    ▷ jeder Bonustyp beinhaltet einen Wert type, als Identifikator und einen Wert
    damage, der den Bonusschaden spezifiziert.
2:      bonus ← 0
3:      for all bt ∈ unit.bonustype do
4:          if Number of Enemies with bt.type > 0 then
            ▷ berechne anteilig den Bonusschaden der aktuellen Einheit
5:              factor ← Number of Enemies with bt.type ÷ Number of Enemies
6:              bonus ← bonus + bt.damage × factor
7:          end if
8:      end for
9:      return unit.basedamage + bonus
10: end procedure
```

Da Einheiten immer nur eine bestimmte Art von Einheiten angreifen können, errechnet der Algorithmus 2 insgesamt 6 Werte. Für jedes der beiden Teams wird der Wert für Schaden an Bodeneinheiten und der Wert für Schaden an Lufteinheiten errechnet. Zusätzlich wird ein Powervalue in Relation zur Einheiten Verteilung Luft-Boden errechnet:

Um die Überlebensfähigkeit der Armeen zu bestimmen werden die Hitpoints (zu Deutsch: Lebenspunkte) der Einheiten sowie deren Schilde in Algorithmus 3 her-

Algorithmus 2 Berechnung der Powervalues nach Angriffs-Typen und des gewichteten Powervalues

```
1: procedure ARMYPOWERVALUES(units, enemyUnits)
2:     powerGround ← 0
3:     powerAir ← 0
4:     for all unit ∈ units do
5:         if unit.attacktype =′ Ground′ then
6:             powerGround ← powerGround + POWERVALUE(unit, enemyUnits)
7:         else
8:             powerAir ← powerAir + POWERVALUE(unit, enemyUnits)
9:         end if
10:    end for
11:    return powerGround, powerAir
12: end procedure
13: powerGround, powerAir = ARMYPOWERVALUES(units, enemyUnits)
14: groundShare = Number of Enemies Type Ground ÷ Number of Enemies
15: airShare = Number of Enemies Type Air ÷ Number of Enemies
16: weightedPower ← powerGround ∗ groundShare + powerAir ∗ airShare
```

angezogen. Auch hier wird bei beiden Armeen zwischen Luft und Bodeneinheiten unterschieden. Somit wird eine Vergleichbarkeit zwischen dem Durchhaltevermögen eigener Luft- bzw. Bodentruppen und Luft- bzw. Boden-DPS der Gegenseite geschaffen.

Die beiden letzten Charakteristika bzgl. beider Armeen sind Aufklärung und Unsichtbarkeit. Sie werden durch boolsche Werte ausgedrückt, welche dann Wahr sind wenn mindestens eine Einheit in der Armee über die entsprechende Fähigkeit verfügt. Die Werte fließen in den Wert *fightable* mit ein, welcher bestimmt, ob eine Armee überhaupt in der Lage ist alle Einheiten der anderen Seite zu attackieren. Fightable ist dann Wahr, wenn die Armee alle in der Gegnerarmee enthaltenen Einheitentypen (Luft/Boden) angreifen kann. Sollte der Gegner über Unsichtbare Einheiten verfügen, so ist fightable genau dann Wahr, wenn die Armee über Einheiten verfügt, die aufklären können.

Als Entscheider-Funktion wurden zunächst lediglich die beiden Team_Werte aus der Prozedur **Decider** (Algorithmus 4) Zeile 19 folgend verglichen. In Tabelle 1 wird der aus den berechneten Labeln und mit Scikit-Learn erstellte Classification-Report veranschaulicht. Es ist sowohl eine F1-Score von 0.67 abzulesen. Der Nachteil der Methode ist, dass Remis als Ausgang nicht in Betracht gezogen werden.

Um Remis zu evaluieren müssen zunächst die Gründe für ein Remis erörtert werden. Als naive Grundlage wird zunächst die Annahme herangezogen, dass ein Remis dann auftritt, wenn beide Armeen sich nicht attackieren können. Die Prozedur **Decider** wird und den Zeilen 2-3 modifiziert, indem die Fightable-Variablen beider Teams abgefragt werden. Für den Fall dass beide Seiten ein Falsch zurückgeben, gibt der Entscheider eine 2 für Remis zurück. Tabelle 2 zeigt den Classification-Report

Algorithmus 3 Berechnung der Überlebensfähigkeit nach Typen

```
1: procedure SURVIVABILITY(units)
2:     defenseGround ← 0
3:     defenseAir ← 0
4:     for all unit ∈ units do
5:         if unit.type =′ Ground′ then
6:             defenseGround ← defenseGround + unit.hitpoints + unit.shield
7:         end if
8:         if unit.type =′ Air′ then
9:             defenseAir ← defenseAir + unit.hitpoints + unit.shield
10:        end if
11:    end for
12:    return defenseGround, defenseAir
13: end procedure
```

Tabelle 1: Classification-Report ohne Remis.

	precision	recall	f1-score	support
Minerals	0,71	0,74	0,72	56.027
Vespene	0,61	0,58	0,59	39.921
avg / total	0,67	0,67	0,67	95.948

Tabelle 2: Classification-Report mit naiver Remis-Evaluation.

	precision	recall	f1-score	support
Minerals	0,66	0,43	0,52	56.027
Vespene	0,47	0,54	0,50	39.921
Remis	0,32	0,50	0,39	25.032
avg / total	0,53	0,48	0,49	120.980

Tabelle 3: Classification-Report mit kompletter Remis-Evaluation.

	precision	recall	f1-score	support
Minerals	0,70	0,48	0,57	56.027
Vespene	0,54	0,51	0,53	39.921
Remis	0,32	0,58	0,41	25.032
avg / total	0,57	0,51	0,52	120.980

dieses Ansatzes. precision, recall und f1-score sind für die Siege beider Seiten stark gesunken und die Accuracy der Auswertung liegt bei etwa 47,91%.

Bei genauer Beobachtung der Gefechte fällt in der Folge auf, dass Gefechte nicht nur Unentschieden ausgehen, wenn beide Teams nicht angreifen können, sondern auch wenn beide Teams sich durch töten einiger Einheiten eines bestimmten Typs in ein Remis kämpfen. Ein weiterer Fall für ein Remis ist durch die zeitliche Einschränkung der Replay-Generierung gegeben. Sollte ein Team nur über wenige Einheiten eines bestimmten Typs verfügen, so kann es sein, dass diese Einheiten fightable zu Wahr evaluieren lassen. Da es aber zu wenige Einheiten sind, schaffen sie es nicht die relevanten Gegnereinheiten in der vorgegebenen Zeit zu töten und der Generator bricht das Gefecht mit einem Remis als Ergebnis ab. Daher wird der Entscheider um die Zeilen 5 bis 17 erweitert. Hier wird die Fähigkeit bestimmt eine Armee in gegebener Zeit zu vernichten. Die Gleichung tritt in Kraft, sobald eine Armee – durch die Zusammensetzung der beiden Armeen – nicht komplett zerstört werden kann. In diesem Fall bestimmt der Algorithmus, ob diese Armee im Rahmen der zeitlichen Vorgaben den Sieg erringen kann, sprich genug Schaden verursachen kann um die andere Armee zeitnah zu zerstören. Die Konstante K in der Berechnung der Schadenswerte in der Zeilen 6 und 13 wurde durch empirisch durch Ausprobieren ermittelt. Abbildung 4.2 zeigt den F1-Score und die Accuracy unter Berücksichtigung verschiedener Werte für die Konstante. Das Maximum wird beider Werte liegt bei ca. 21. Der Classification-Report für Konstante=21 ist in Tabelle 3 dargestellt. Die Accuracy für diese Variante liegt bei ca. 51% und bringt somit einen Zugewinn von 3,09 Prozentpunkten.

Algorithmus 4 Entscheidungsprozedur

```
1: procedure DECIDER(teamA, teamB)
2:     if ¬teamA.fightable ∧ ¬teamB.fightable then
3:         return 'Remis'
4:     end if
5:     if teamA.fightable ∧ ¬teamB.fightable then
6:         if teamA.powerGround ∗ K ≥ teamB.defenseGround ∧
   teamA.powerAir ∗ K ≥ teamB.defenseAir then
7:             return 'Team A siegt'
8:         else
           ▷ Team A schafft Sieg nicht innerhalb der Zeit
9:             return 'Remis'
10:        end if
11:    end if
12:    if ¬teamA.fightable ∧ teamB.fightable then
13:        if teamB.powerGround ∗ K ≥ teamA.defenseGround ∧
   teamB.powerAir ∗ K ≥ teamA.defenseAir then
14:            return 'Team B siegt'
15:        else
           ▷ Team B schafft Sieg nicht innerhalb der Zeit
16:            return 'Remis'
17:        end if
18:    end if
19:    teamAPower ← teamA.weightedPower + teamA.defenseGround +
   teamA.defenseAir
20:    teamBPower ← teamB.weightedPower + teamB.defenseGround +
   teamB.defenseAir
21:    if teamAPower > teamBPower then
22:        return 'Team A siegt'
23:    else
24:        return 'Team B siegt'
25:    end if
26: end procedure
```

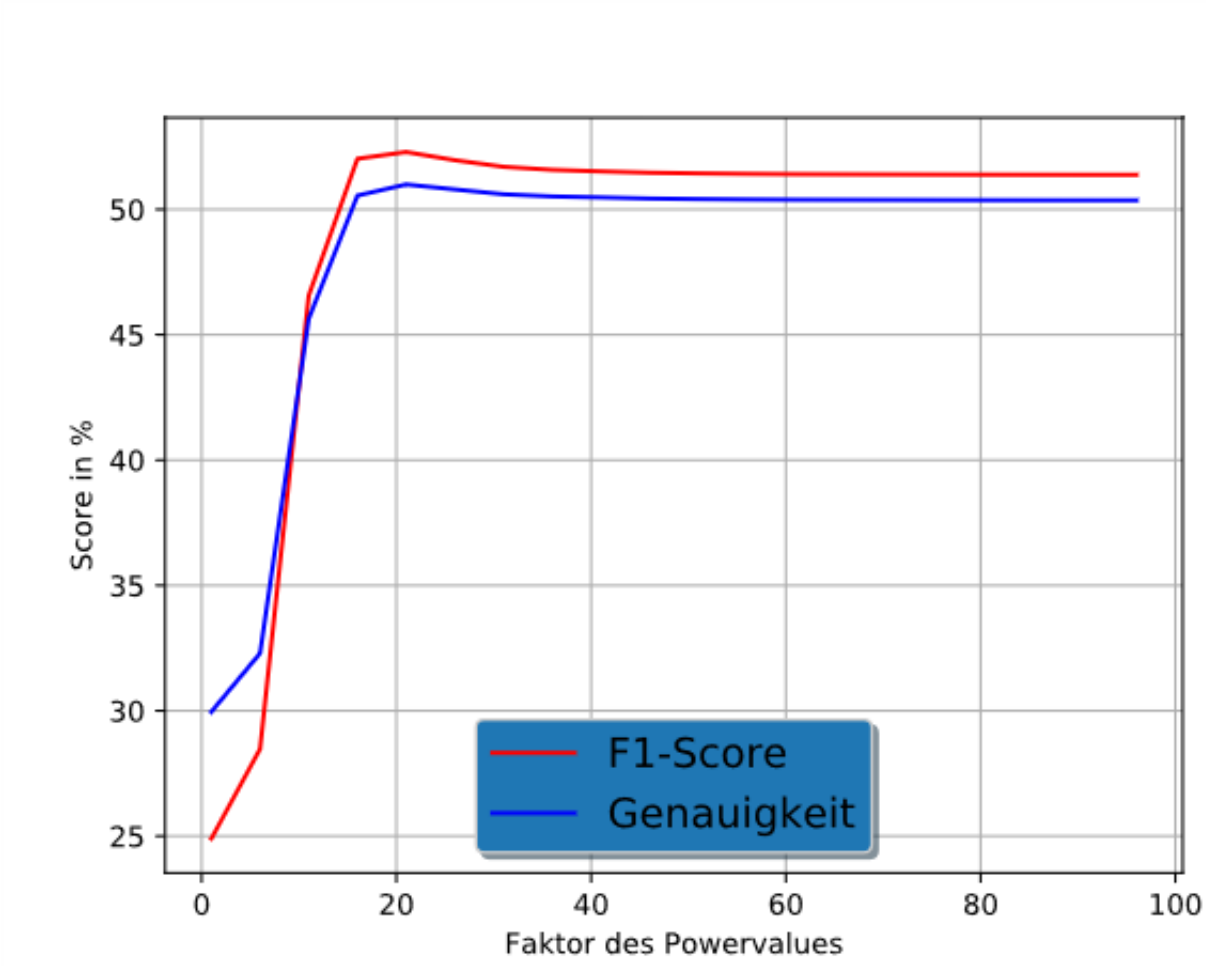

Abbildung 6: Accuracy und F1-Score für $K \in [6, 96]$.

4.3 All-Convolutional Net

Das All-Convolutional Net von Springenberg et al. [15] ist eine CNN-Architektur, die gänzlich auf den Gebrauch von Max-Pooling Layern verzichtet. Die Reduzierung der Dimensionen wird durch den Einsatz von *Convolutional Layern* mit einem entsprechenden stride-Parameter vollzogen. In Experimenten basierend auf dem CIFAR-10 Datensatz erreichten die vorgestellten Architekturen eine Fehlerrate von weniger als 10% und zeigten im Vergleich zu Modellen mit Max-Pooling äquivalente oder sogar bessere Fehlerraten. Tabelle 4 zeigt die Parameterzahlen der in dieser Bachelorarbeit genutzten Architektur, welche in Abbildung 7 dargestellt ist. Die Tabelle listet in der Layer-Spalte zuerst die Filtergröße, dann die Filteranzahl und nachfolgend gegebenenfalls den stride- und den padding-Parameter mit auf. Soll-

Tabelle 4: All Convolutional Neural Network - Architektur.

Layer	Ausgabedimension	Parameter
Eingabe	8 x 84 x 84 x 1	
3x3 Conv, 96	8 x 84 x 84 x 96	1.152
3x3 Conv, 96	8 x 84 x 84 x 96	83.232
3x3 Conv, 96/s=2 V	8 x 41 x 41 x 96	83.232
3x3 Conv, 192	8 x 41 x 41 x 192	166.464
3x3 Conv, 192	8 x 41 x 41 x 192	332.352
3x3 Conv, 192/s=2 V	8 x 20 x 20 x 192	332.352
3x3 Conv, 192	8 x 20 x 20 x 192	332.352
1x1 Conv, 192	8 x 20 x 20 x 192	37.440
1x1 Conv, 10	8 x 20 x 20 x 10	1.950
10x10 Average Pooling	8 x 2 x 2 x 10	
Output Layer (FCL), Units=3		963
Summe Variablen		1.371.489

ten diese nicht aufgeführt sein, gilt *stride=(1,1,1)* und *padding='same'*. Die Ausgabedimension ist im Format *channels last* (zu Deutsch: *Kanäle zuletzt*) angegeben.

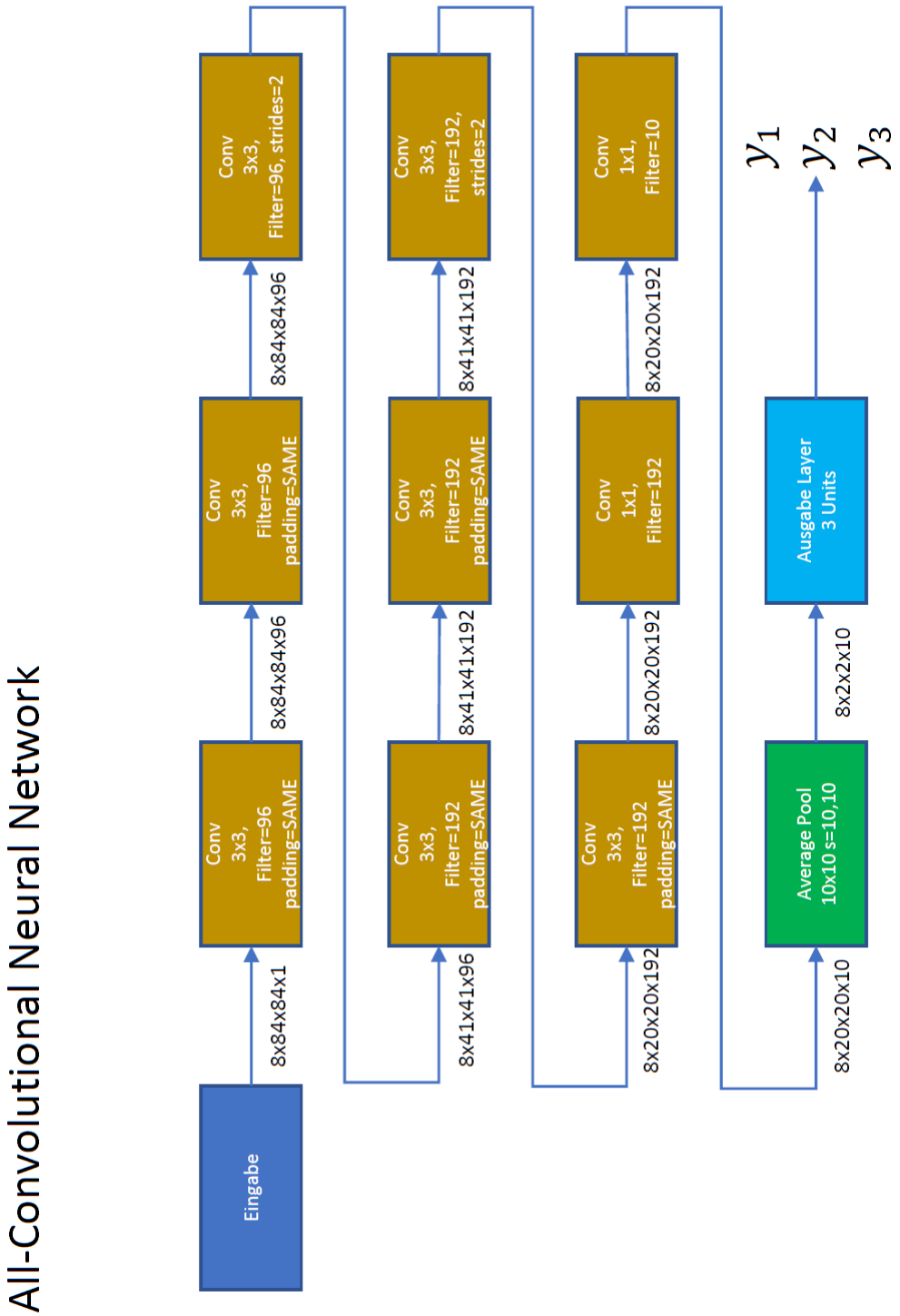

Abbildung 7: Architektur des All-Convolutional CNN.

4.4 ResNet

ResNet ist eine von He u. a. [4] vorgestellte Architektur, welche sich Shortcut Connections (zu deutsch etwa: *abkürzende Verbindungen*) zu nutze macht. Mit zunehmender Tiefe von CNNs wurde bei einigen Problemstellungen festgestellt, dass tiefere CNNs an Genauigkeit verlieren können, was im Allgemeinen der intuitiven Einschätzung widersprach, da man sich von dem Einsatz zusätzlicher Layer eine steigende Präzision erwartete. Da eine Evaluierung der Gradienten des entsprechenden CNNs das Vanishing Gradient Problem (zu deutsch etwa: *verschwindende Gradienten Problem*) ausschloss, wurde die Vermutung aufgestellt, dass die Layer im CNN leichter sogenannte *zero mappings* ($f(x) = 0$) lernen und daher nicht die Identitätsfunktion ($f(x) = x$). Als mögliche Lösung werden im Deep Resiual Neural Network shortcut connections eingefügt, welche ausschließlich als Identitätsfunktion dienen.

Abbildung 8 zeigt die Implementation eines ResNets mit acht Residual Blöcken. Die Eingabe wird zunächst von einem Convolutional Layer mit 7x7 Filtergröße und Stride=2 verarbeitet um die Dimensionen der Eingabe zu reduzieren. Nachfolgend werden diverse Residual Blöcke hintereinander gereiht. Die Residual Blöcke bestehen aus zwei Convolutional Layern mit einer Filtergröße von 3x3 und werden von einer Residual Connection überbrückt. Diese bildet in allen Blöcken mit Ausnahme von Block 6 die Identität der Eingabematrizen ab und wird nach den beiden Convolutional Layern hinzu addiert. Da Block 6 die Dimensionen in Höhe und Breite der Eingabe durch einen Stride-Parameter=2 reduziert und zugleich die Anzahl der Kanaldimensionen auf 128 erhöht, muss die Residual Connection mittels eines Convolutional Layers die gleichen Berechnungen vollziehen. Daher besteht in diesem Fall die Residual Connection aus einem Convolutional Layer mit Filtergröße 3x3 und Stride=2. Nach den Residual Blöcken werden die Daten wie bei Inception V4 mittels Average Pooling auf die Dimensionen 8x2x2x128 (Kanalanzahl) reduziert und danach von einem FCL mit 64 Einheiten weiterverarbeitet. Der Ausgabelayer wird von einem FCL mit drei Einheiten gebildet. Tabelle 7 listet die Anzahl der Parameter auf.

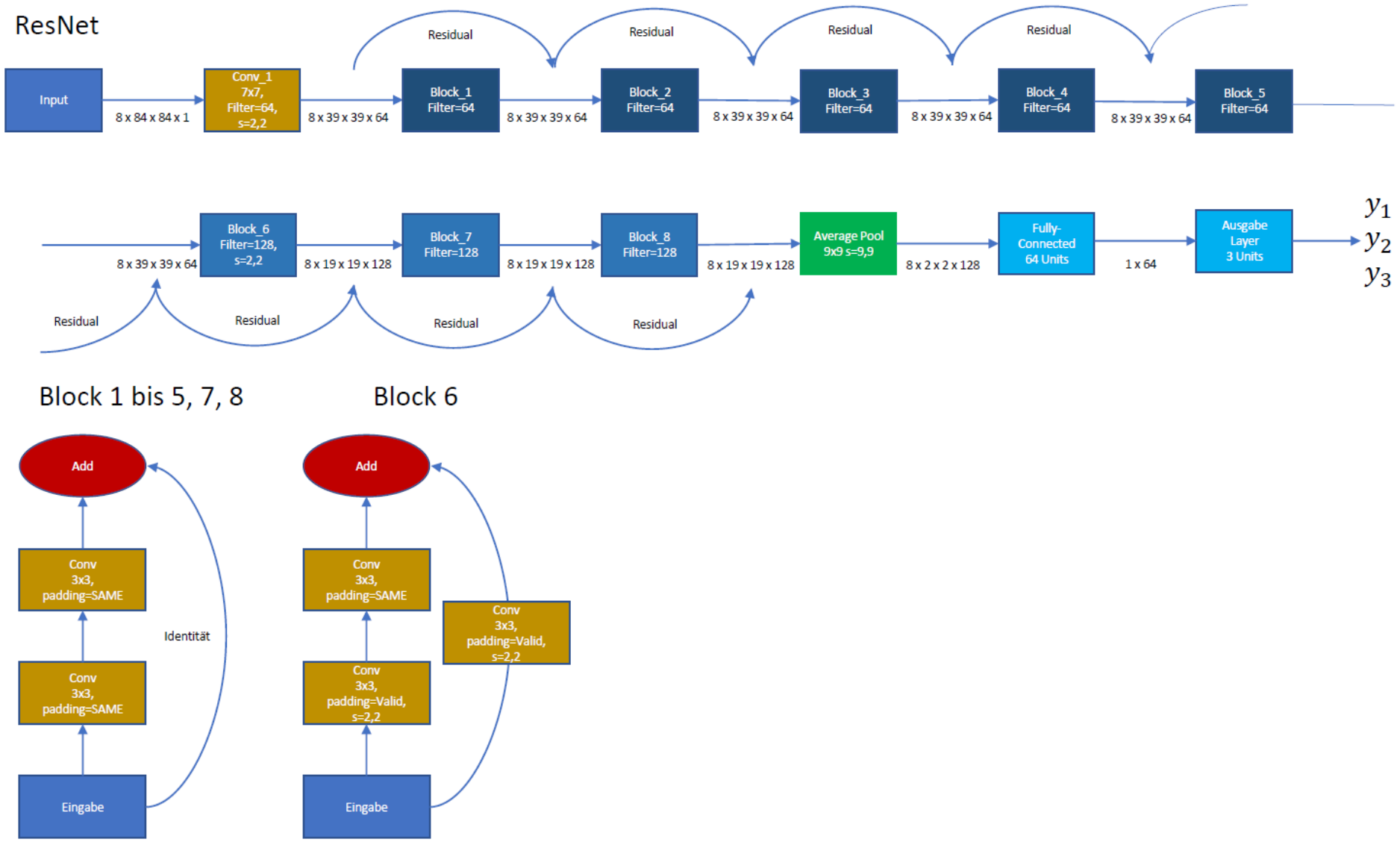

Abbildung 8: ResNet mit acht Residual Blöcken.

Tabelle 5: ResNet: Auflistung der Variablenzahl.

Block	Parameter
7x7 Conv	3.328
Block 1	73.984
Block 2	73.984
Block 3	73.984
Block 4	73.984
Block 5	73.984
Block 6	295.552
Block 7	295.424
Block 8	295.680
FCL Units=64	262.208
Ausgabe Layer	195
Summe Variablen	1.522.307

4.5 Inception V4

Inception V4 ist eine CNN-Architektur vorgestellt von Szegedy, Ioffe und Vanhoucke [18]. Sie führt drei verschiedene Module, sowie zwei Reduction-Blöcke als Bausteine des Netzes ein. Die Zusammensetzung der Bausteine ist in den folgenden Abbildungen 9, 10, 11 und 12 dargestellt.

Im Vergleich zu früheren Versionen der Inception-Architekturen konnte die Trainingsgeschwindigkeit durch das Nutzen von Residual Connections drastisch erhöht werden. Zudem erreichten die neuen Versionen der Architektur verglichen mit Inception-v3 und ResNet-151 bessere Leistungen in Tests auf dem ILSVRC 2012 Datensatz [18].

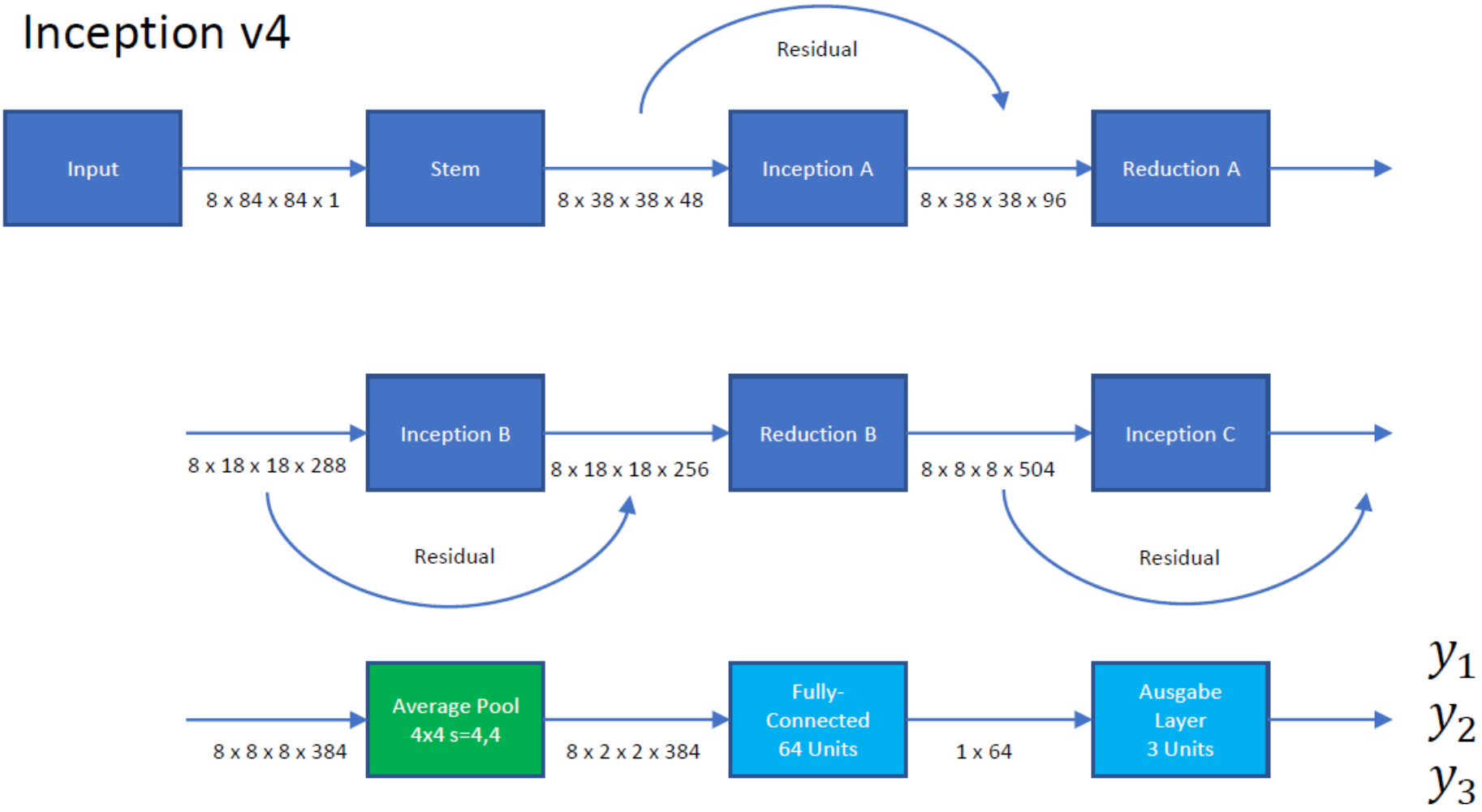

Abbildung 9: Übersicht der Inception V4-Architektur.

Abbildung 9 zeigt den allgemeinen Aufbau der Inception V4 Netzwerkes. Filter sind mit der Filtergröße und Filteranzahl, sowie gegebenenfalls Stride (s) und Padding (Vaild) aufgeführt. Sofern Stride und Padding fehlen, gilt Stride=1 und Padding=*same*. Die Dimensionen sind an den Pfeilen zwischen den Layern aufgeführt und nach dem Format *channels last* formatiert. Im Vergleich zu der Vorlage von Szegedy, Ioffe und Vanhoucke [18] wurde Filteranzahl eines jeden Layers durch 4 geteilt um die Parameterzahl in dem Bereich der anderen Architekturen zu halten. Zur Normalisierung wird im Allgemeinen Batch Normalization genutzt, welches nach der Konkatenation in jedem Block zur Anwendung kommt.

Die Eingabe wird zunächst vom STEM-Block verarbeitet, welcher in Abbildung 10 detailliert erläutert wird. Er dient im Netzwerk als Eingabelayer und dient der ersten Verarbeitung der Daten sowie der Reduktion der Bilddimensionen durch Pooling und entsprechende Padding-Einstellungen. Daraufhin werden die Daten an das erste Inception-Modul (Inception-A Abbildung 11) übergeben. Das Inception-A-Modul wird von einer Residual Connection überbrückt. Das Verfahren ist der ResNet-Architektur aus Sektion 4.4 entlehnt. In dieser Architektur wird die Residual Connection mittels eines 1x1 Convolutional Filters auf die passende Anzahl an Kanälen dimensioniert. Das Inception-A-Modul in Verbindung mit der Residual Connection kann beliebig oft hintereinander gereiht werden, um die gewünschte Tiefe des Netzwerks zu erlangen, da sich die Dimensionen mit fortlaufender Aneinanderreihung nicht verändern. Unter Berücksichtigung der Parameterzahl des Netzwerks, verfügt das Netzwerk jedoch nur über ein Inception-A-Modul, bevor es in das Reduction-A-Modul (Abbildung 12) übergeht. Im Reduction-A-Modul werden die Dimensionen der Eingabedaten entlang der zweiten und dritten Achse reduziert während sich die Anzahl der Kanäle von 96 auf 288 erhöht. Er folgen Inception-B (Abbildung 11) und Reduction-B (Abbildung 12), welche nach dem selben Muster verfahren. Das Inception-Modul verarbeitet die Daten auf vier verschiedenen Pfaden und übergibt diese an das Reduction-Modul um erneut die Dimensionen der Höhe und Breite auf 8x8 zu reduzieren. Auch das Inception-B-Modul wird von einer Residual Connection überbrückt und kann zusammen mit dieser Verbindung beliebig oft in Reihe geschaltet werden. Den Abschluss der Convolutional Layer bildet das Inception-C-Modul (Abbildung 11). Nachfolgend wird mittels Average Pooling die Eingabe auf 8x2x2x384 reduziert und als Vektor an den FCL übergeben. Die Auswertung zu den drei Klassen findet in dem zweiten FCL statt, welcher lediglich aus drei Einheiten besteht.

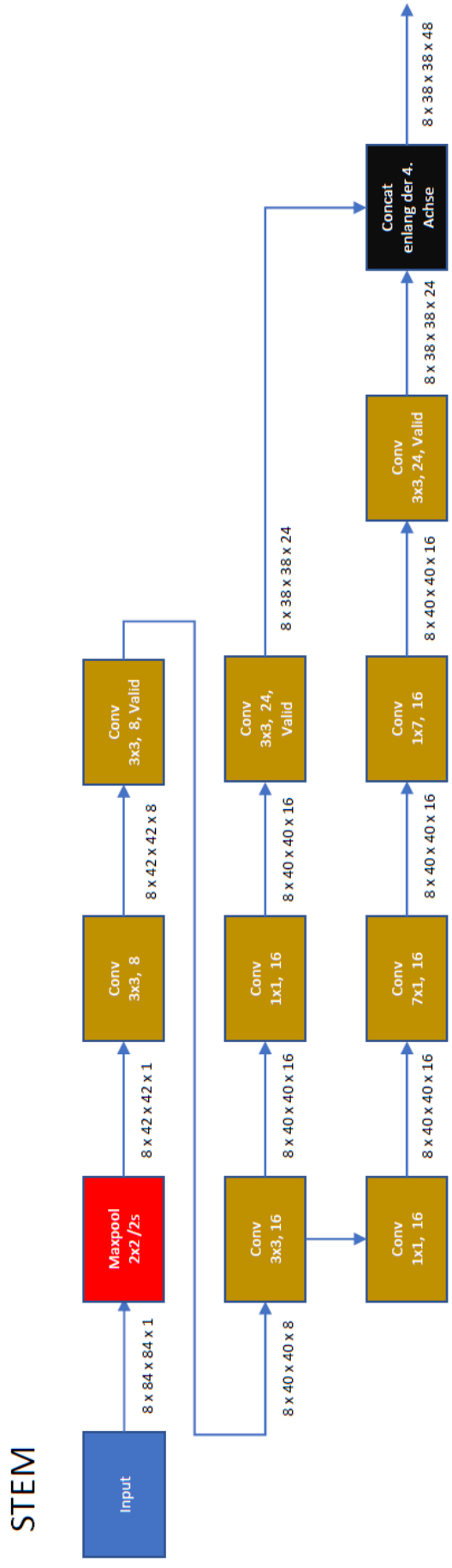

Abbildung 10: Aufbau des STEM-Moduls.

Der STEM-Block aus Abbildung 10 bildet die Eingabeschicht der Architektur, welcher die ersten Berechnungen vollzieht und gleichzeitig der ersten Reduzierung der Bilddimensionen dient. In dieser Version sind nicht alle dimensionsreduzierenden Elemente des STEM-Blocks aus Szegedy, Ioffe und Vanhoucke [18] enthalten, da die Eingabedimension der Daten dieser Arbeit wesentlich geringer ist, als die Bildgröße des ILSVRC 2012 Datensatzes (84x84 im Vergleich zu 299x299), somit ist eine weitere Reduktion der Daten vor Einführung des ersten Inception-Moduls nicht notwendig. Nach anfänglicher Verarbeitung durch einige Convolutional Layer, werden die Daten auf zwei Wegen weiterverarbeitet. Der erste Weg arbeitet mit quadratischen Filtergrößen, während der zweite Weg mit rechteckigen Filtergrößen in Reihe arbeitet. Beide Wege werden vor der Weitergabe an das nächste Modul entlang der Kanalachse (4. Achse) konkateniert, sodass aus den beiden Ausgaben mit Dimensionen 8x38x38x24 eine einzige Ausgabe mit 8x38x38x48 entsteht. Diese wird an das Inception-A-Modul weitergegeben.

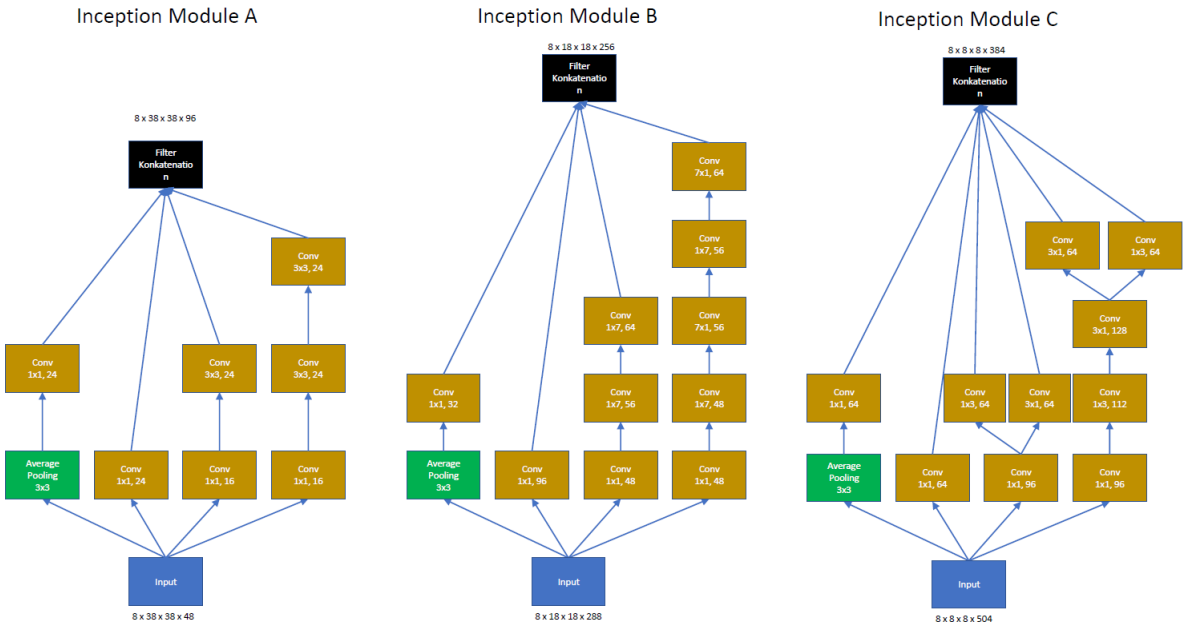

Abbildung 11: Übersicht der Inception V4-Module A,B und C.

Die Inception-Module (Abbildung 11) sind speziell für ihre Eingabegrößen angepasst. Alle Module spalten sich aus den Eingabedaten heraus in 4 Wege ab. In Modul C spalten sich zwei Wege erneut auf. Jedes Modul nutzt 1x1 Convolutional Layer um die Dimensionen der Eingabe vor der Verarbeitung oder Weitergabe zu reduzieren. Diese Layer findet man auf den drei rechten Wegen an erster Stelle und auf dem linken Weg nach dem Average Pooling in allen Modulen. Die Unterschiede zwischen den Modulen findet man hauptsächlich auf der rechten Seite aller Module, da sich die Convolutional Layer links lediglich durch die Filteranzahl un-

terscheiden. Auf der rechten Seite der Module lässt sich erkennen, dass Modul A auf symmetrische Filter setzt. Die Module B und C arbeiten jedoch mit asymmetrischen Filtern, welche meist entgegengesetzt angeordnet sind (3x1 folgt auf 1x3). Die rechten Wege in Modul B sind drei, bzw. fünf Layer tief und mit Filtern der Maße 1x7 und 7x1. In Modul C hingegen spalten sich die Wege nach einem und drei Layern auf und die Filter arbeiten mit 3x1 und 1x3 Layouts. Alle Module konkatenieren schlussendlich ihre vier bzw. sechs Verarbeitungspfade entlang der Kanalachse, bevor die Daten den nächsten Layer erreichen.

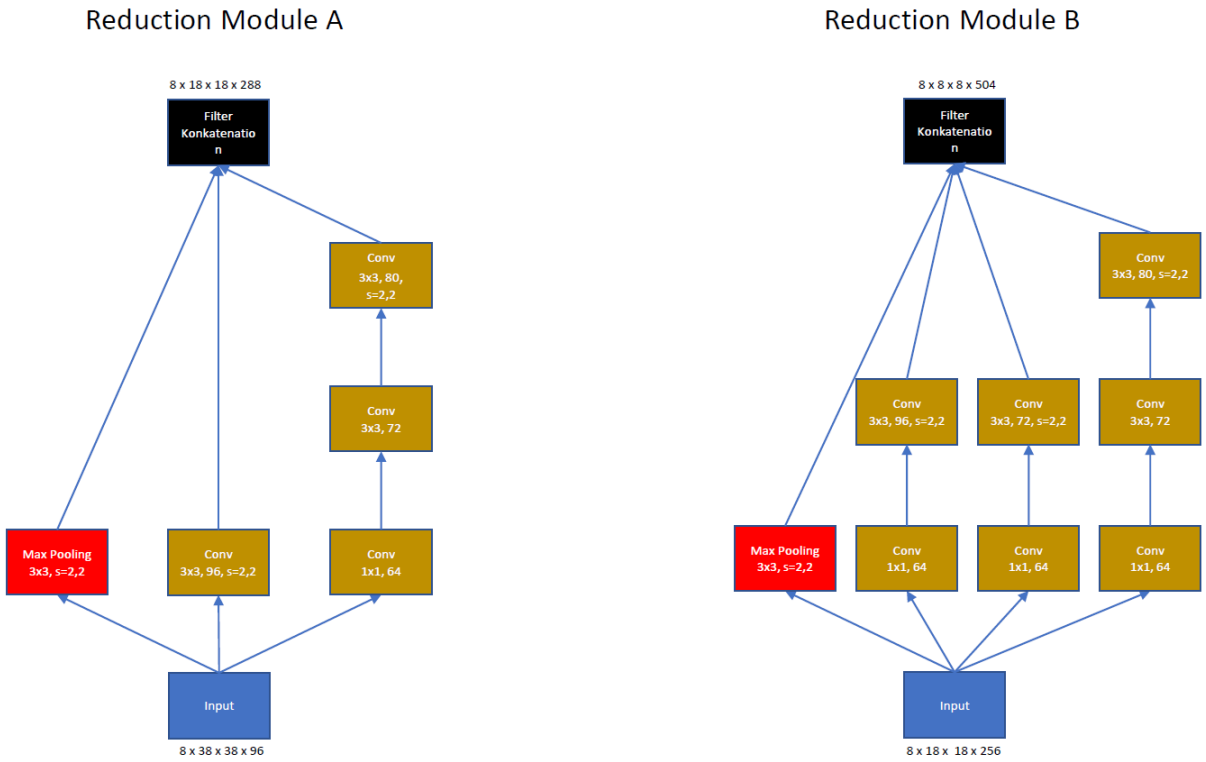

Abbildung 12: Übersicht der Inception V4-Reduktions-Module A und B.

Zur Reduzierung der Bilddimensionen Höhe und Breite nutzt Inception Reduction-Module (Abbildung 12). Beide Module nutzen sowohl Max-Pooling als auch Convolutional Layer mit Stride=2 um die Dimensionen der Eingabedaten entsprechend zu reduzieren. Reduction-A nutzt lediglich auf der rechten Seite des Moduls einen 1x1 Filter um die Dimensionen der Daten zu reduzieren. Bei Reduction-B kommt diese Methode bei den drei rechten Wegen zum Einsatz. In beiden Modulen wird mit 3x3 Filtern gearbeitet. Diese Filter reduzieren die Größe der Eingabedaten bzlg. Höhe H^i und Breite W^i wie folgt: $W^o = W^i/2 - 1$ und $H^o = H^i/2 - 1$. W^o und H^o stehen hier entsprechend für die Breite und Höhe der Ausgabe. Auch in den Reduction-Modulen werden die Ergebnisse der einzelnen Berechnungspfade zuletzt entlang

Tabelle 6: Inception V4 und V4-SE: Auflistung der Variablenzahl.

Block	Parameter V4	Parameter V4-SE
STEM	13.048	13.048
Inception A	20.984	21.008
Reduction A	182.144	182.144
Inception B	265.464	224.568
Reduction B	240.752	240.752
Inception C	518.064	398.352
Fully-Connected	786.496	786.496
Output Layer	195	195
Summe Variablen	2.027.147	1.866.563

der Kanalachse konkateniert.

Tabelle 6 zeigt die einzelnen Module der Inception Architektur und ihre Variablenzahlen. In den Zahlen für Inception A, B und C sind die Variablen der Residual Network Verbindungen mit eingerechnet.

4.6 Inception mit Squeeze-and-Excitation-Layer

Hu, Shen und Sun [6] hat Squezze-and-Excitation Layer eingeführt, um den Aspekt der Kanal Beziehungen zu untersuchen. Die Grundidee folgt dem Gedanken, dass unter den einzelnen Kanälen der Convolutional Layer Beziehungen modelliert werden können, welche es dem Netzwerk ermöglichen Feature Recalibration (zu Deutsch: *Merkmal-Neukalibrierung*) auszuüben. Dieser Mechanismus ermöglicht es dem Netzwerk einzelne gewinnbringende Feature stärker zu gewichten und schwache Feature zu unterdrücken. Squeeze-and-Excitation Layer werden in der Arbeit von Hu, Shen und Sun [6] nicht als eigenständige Bausteine eines Netzwerkes verstanden, sondern dienen als Erweiterung bestehender Architekturen. Daher wird in dieser Bachelorarbeit das Inception-Netzwerk, welches in Sektion 4.5 eingeführt wird, um diesen Layer erweitert. Grafik zeigt nach welchem Muster Squeeze-and-Excitation in Inception eingeführt werden kann. In der aktuellen Architektur werden die Squeeze-and-Excitation Layer jeweils nach den Inception Modulen angewendet.

Squeeze-and-Excitation Layer bestehen – wie in Abbildung 13 dargestellt – aus vier zusätzlichen Elementen. Zunächst wird Global Average Pooling auf die Matrizen angewandt. Die Dimensionen der Ausgabe entsprechen in der Folge 1x1x1xAD, wobei AD für Ausgabedimension steht und gleich der Anzahl der Kanäle der Inception-Ausgabe ist. Als nächstes wird ein FCL genutzt für den ein neuer Hyperparameter Reduction-Ratio (RR, zu Deutsch: *Reduktionsverhältnis*) eingeführt wird. Dieser Hyperparameter erlaubt es die Kapazität und Berechnungskosten der Squeeze-and-

Excitation Layer zu beeinflussen, je höher der Wert gewählt ist, desto weniger Parameter fügen die Squeeze-and-Excitation Layer dem Netzwerk hinzu. Auf den FCL folgt eine ReLU-Aktivierung und dann erneut ein FCL, bei dem die Anzahl der Einheiten der AD entspricht. Das Ergebnis wird in die Ausgabe des Inception-Moduls integriert, indem die Werte des FCL kanalweise mit der Ausgabe multipliziert werden. Die exakte Anzahl der Parameter dieser Version sind in der zweiten Spalte von Tabelle 6 aufgelistet, wobei die Parameter der Squeeze-and-Excitation Layer in die Parameterzahl, der drei Inception Module eingerechnet sind.

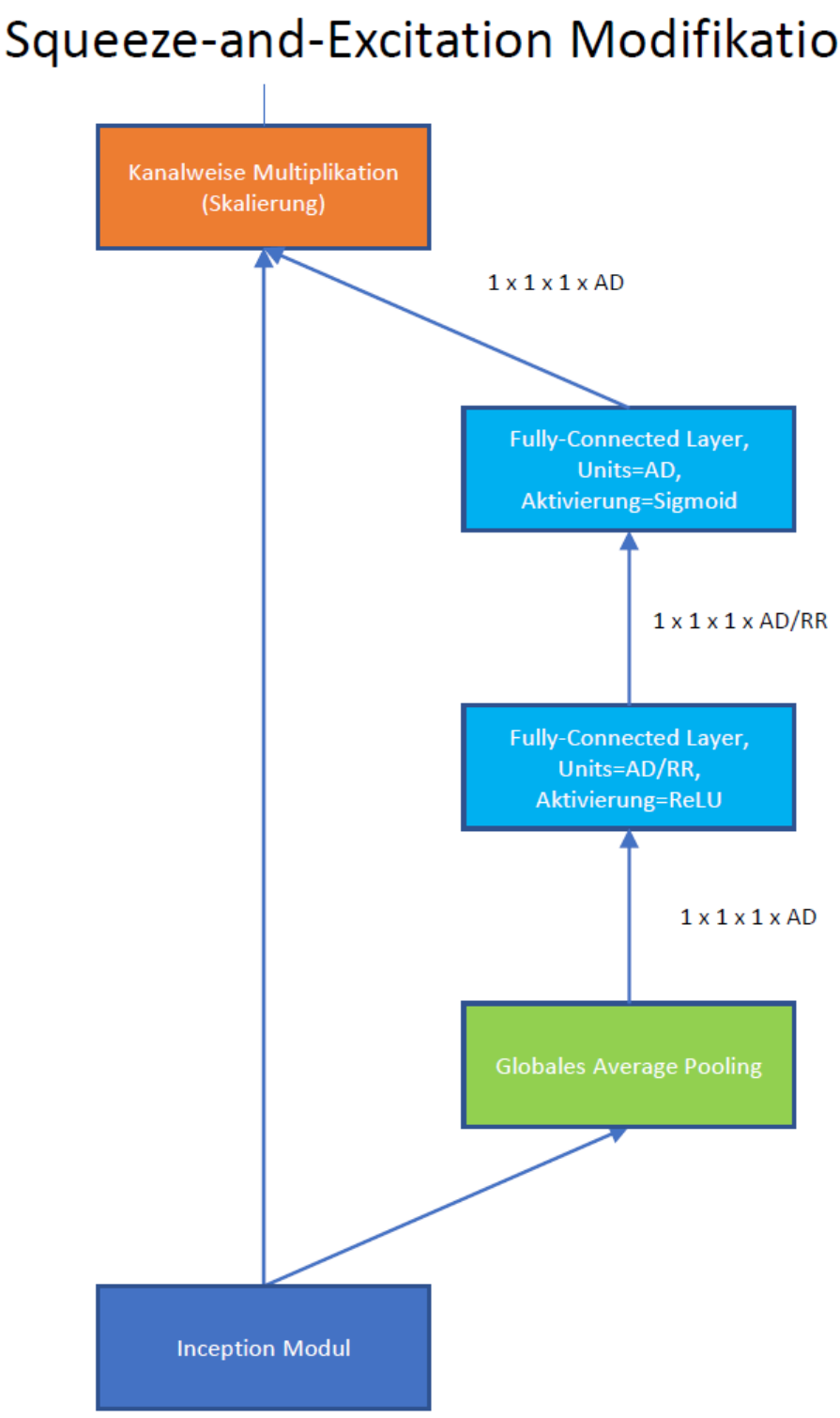

Abbildung 13: Der Squeeze-and-Excitation-Block.

5 Evaluation

5.1 Datengenerierung

Der Vorgang der Datengenerierung wurde nach dem in Abbildung 14 aufgezeigten Schema implementiert. Durch die Verwendung einer Custom Map (zu Deutsch: *benutzerdefinierten Karte*), eine durch den in StarCraft II mitgelieferten Editor modifizierte Spielfläche, können die beiden Armeen mit zufällig zusammengestellten Einheiten-Konstellationen generiert werden. Zu Beginn des Gefechts werden beide Armeen am gegenseitigen Ende der Karte erzeugt. Bevor die Armeen den Kampf beginnen, wird der Spiel-Zustand (Spieler IDs, Höhenkarte, Einheitentyp, etc.) in Form von mehreren Matrix-Repräsentationen gespeichert. Sobald eine der beiden Seiten keine Einheiten mehr hat gilt die Schlacht als beendet und die Armee, welche noch über Einheiten verfügt wird als Sieger betrachtet. Das Ergebnis wird als Integer-Wert gespeichert und im späteren Verlauf des Trainings zu einem One-Hot-Vektor transformiert.

Die Armeen werden zufällig generiert, der Generierungsprozess läuft wie folgt ab: Zunächst wird für beide Armeen eine Rasse gewürfelt. Die Rasse bestimmt welche Einheiten die Armee erzeugen darf. Im Folgenden wird für jede Einheit durch einen Zufallszahlen-Generator die Anzahl der maximal zu erzeugenden Einheiten ermittelt. Die Anzahl der Einheit, welche insgesamt von einer Armee erzeugt werden dürfen ist auf eine zufällige Grenze zwischen 10 und 50 begrenzt. Es werden daher alle Werte der einzelnen Einheiten so reduziert, dass die Gesamtzahl der Einheiten in einer Armee diese Grenze nicht überschreitet. Da bei zu großen Unterschieden in der Armeegröße die Klassifizierung zu einer trivialen Aufgabe wird, wurde zusätzlich eine Versorgungs-Grenze implementiert. In StarCraft II ist die Versorgungsgrenzer ein Wert, welcher die Größe der Armee eines Spieler begrenzt. Diese Grenze wird in dem Kontext der Datengenerierung genutzt um nur jene Gefechte zu erzeugen, in denen der Unterschied der Einheiten-Versorgung (folgend Supply-Differenz) beider Armeen kleiner ist als ein definierter Grenzwert (in diesem Fall 5).

Abbildung 15 zeigt den Generierungsprozess in Gänze. Die Custom Map aus StarCraft II generiert ein Gefecht mit passender Differenz an Einheiten-Versorgung und lässt dieses nach speichern des Spielzustands austragen. Dieser Vorgang wird vor dem Beenden der Custom Map 25 Mal ausgeführt. Mit Beenden der Custom Map werden alle Vorgänge auf der Karte, als Replay (zu Deutsch: *Wiederholung*) gespeichert. Ein Replay enthält somit 25 Gefechte, welche alle einzeln ausgewertet werden.

Die Entscheidung mehrere Gefechte auf einmal zu generieren wurde getroffen, da die Custom Map bei jedem Start das Spiel neu laden muss und somit der zeitliche Overhead bei Einzelgenerierung der Replays wesentlich größer wäre. Die Anzahl der in einem Replay generierten Gefechte wurde auf 25 gesetzt, da bei zu vielen Iterationen der Gefechtsgenerierung die Replay-Datei zu groß wurde und das Spiel beim Schließen der Custom Map abstürzte. Der Absturz resultierte in jedem Fall in einem Datenverlust. Bei 25 Gefechten pro Replay liegt die Absturzrate derzeit

immer noch bei ca. 50%, jedoch resultierte das weitere Senken der Gefechte pro Replay nicht in einem stabileren Generationsprozess.

Die Generierung der Gefechte findet auf der schnellsten Spieleinstellung statt, damit in gleicher Zeit mehr Gefechte simuliert werden können. Dadurch ist es aktuell möglich in einem Zeitraum von acht Stunden ca. 250 valide (500 Generierungen inklusive der Abstürze) Replays zu produzieren, welche nach der Feature Extraktion (Zeitaufwand erneut ca. drei bis vier Stunden) in 6250 Samples resultieren.

Abbildung 14: Übersicht des Generierungsprozesses der Trainingsdaten.

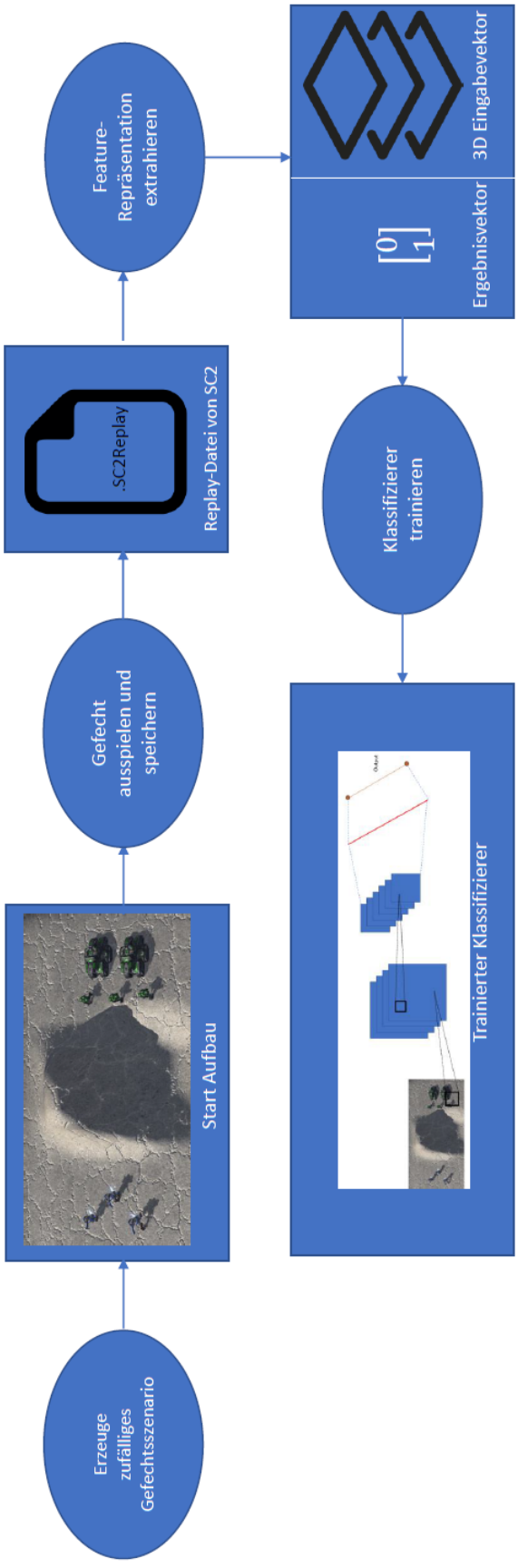

Abbildung 15: Übersicht des Generierungsprozesses der Trainingsdaten.

Die Abbildungen 16, 17 und 18 zeigen die Zusammensetzung der simulierten Gefechte auf. Abbildung 16 zeigt die absolute Anzahl der Gefechte unter Berücksichtigung der Supply-Differenz. Die X-Achse trägt die Supply-Differenz in 0,5er-Schritten auf und auf der Y-Achse befindet sich die absolute Anzahl der entsprechenden Gefechte. Um die reelle Differenz der Gefechte zu erhalten muss man den Wert der X-Achse durch zwei teilen. Es sind drei klare Abstufungen zu erkennen. Die meisten Gefechte wurden mit einer Supply-Differenz $sd \leq 5$ generiert. Das zweite Plateau bilden die Gefechte $5 < sd \leq 10$ und das dritte Plateau wird von den Gefechten mit $10 < sd \leq 15$ gebildet. Die Bildung der Plateau ist durch den Generierungsprozess zu erklären. Es wurden mittels drei unterschiedlicher Custom Maps Daten generiert, welche in Versionen unterteilt wurden. Die erste Version lässt die Generierung von Gefecht mit einer Differenz von bis zu 5 zu. Version 2 lässt eine Differenz von 10 zu und Version 3 dementsprechend von 15. Da engere Gefechte nicht ausgeschlossen wurden, ist die Anzahl der engen Differenzen folglich höher. Die starken Einbrücke in jedem zweiten Balken werden von der Rasse *Zerg* hervorgerufen, da diese die einzige Rasse sind deren Einheiten eine Supply-Anforderung von 0,5 haben können. Somit sind auch Differenzen mit X,5 nur möglich bei einer Beteiligung der Zerg an dem Gefecht. Auch wenn die Versionen nur Gefechte mit einer Differenz von maximal 15 zulassen, wurden dennoch Gefechte mit teilweise deutlich höheren Differenzen generiert. Dieser Umstand ist der Tatsache geschuldet, dass die Custom Maps während der Generierung bei einer zu großen Supply-Differenz die Armeen neu würfeln. Wenn dieser Vorgang zu oft wiederholt werden muss, kommt es zu einem Absturz der Custom Map, weshalb nach 8 Wiederholungen das Gefecht ohne Berücksichtigung der Supply-Differenz generiert wird um einem Absturz zu verhindern. Da die Supplies auch gespeichert werden, können diese Gefechte beim Training der Modelle problemlos herausgefiltert werden.

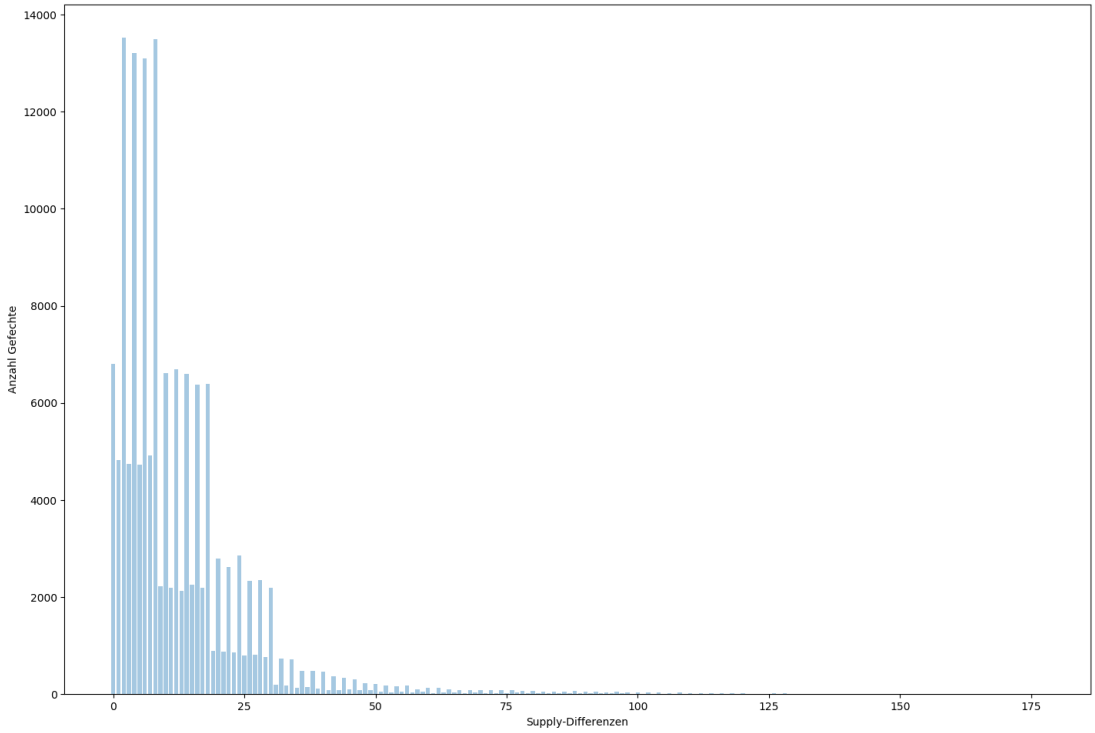

Abbildung 16: Verteilung der generierten Gefechte nach Supply-Differenz.

Abbildung 17 zeigt die Anzahl der Gefechte, sowie die Anzahl der Siege insgesamt und nach gegnerischer Rasse. Die drei Balkengruppen sind nach Rassen aufgeteilt und zeigen für jede Rasse jeweils die Anzahl der Gefechte, Siege, Siegen gegen Zerg, Protoss und Terran. Die Anzahl der Gefechte ist für alle Rassen in etwa gleich und liegt zwischen 81.000 (Zerg) und 85.500 (Terran). Die Siege sind auch in ähnlichen Dimensionen, wobei Protoss und Terran mit jeweils 38.000 Siegen sehr nah beieinander liegen und Zerg mit 44.000 etwas darüber. Die Siege nach Rassen weisen im allgemeinen eine gleichmäßige Verteilung auf. Lediglich beim Aufeinandertreffen von Protoss und Terran scheinen die Protoss einen Vorteil zu haben, da die Terraner nur einen Bruchteil der Siege gegen Protoss haben. Protoss gewann 15.719 mal gegen Terran, Terran hingegen nur 4263 mal gegen Protoss.

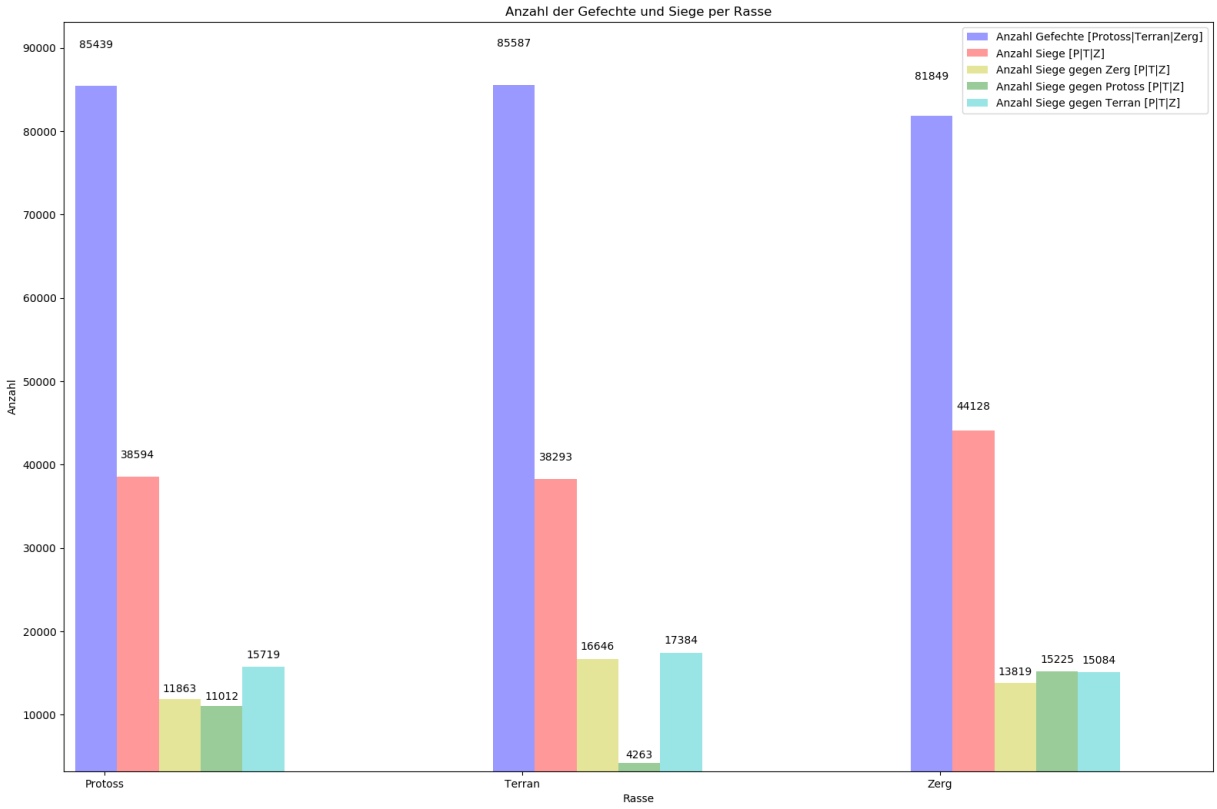

Abbildung 17: Verteilung der Siege nach Rasse und Gegnerrasse.

In Abbildung 18 werden die Siege nach Supply-Differenz und Rasse grob unterteilt analysiert. Die Balkengruppen auf der X-Achse bilden die unterschiedlichen Supply-Differenzen ab. Auf der Y-Achse sind die Siege in absoluten Zahlen aufgetragen. *Sieg bei kleiner Differenz* umfasst alle Siege, bei denen der Sieger einen Supply-Vorteil sv von $0 < sv \leq 5$ hatte. *Sieg trotz Nachteil* beinhaltet alle Siege, bei denen der Sieger weniger Supply genutzt hat als der Verlierer. *Siege mit hoher Differenz* zählt alle Siege mit einer Differenz größer 5 auf, bei denen der Sieger im Vorteil war und *Siege bei gleichem Supply* umfasst alle Siege bei denen der Supply exakt gleich war. Anhand der Verteilung der Siege lässt sich ablesen, dass Zerg bei einem Supply-Vorteil zum Sieg tendieren, während Protoss die Besten darin sind, aus einem Nachteil einen Sieg zu machen. Bei gleichem Supply sind die Zerg am schwächsten und Protoss am erfolgreichsten.

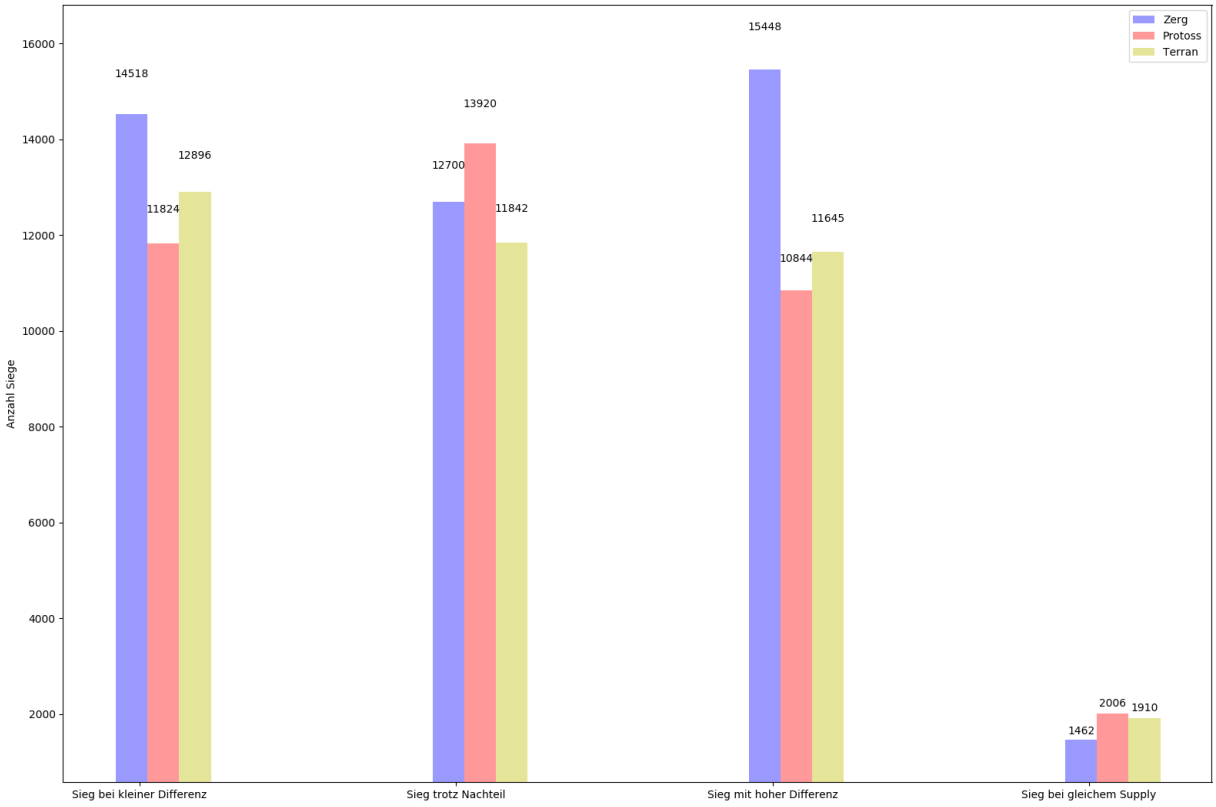

Abbildung 18: Verteilung der Siege nach Supply-Differenz und Rasse.

5.2 Evaluationsmetriken

Die Leistungsfähigkeit der einzelnen Architekturen wird Anhand verschiedener Metriken bestimmt. Die TOP-1 error rate (zu Deutsch: Top-1 Fehlerrate) wird in der aktuellen Forschung häufig zum Vergleich unterschiedlicher Machine Learning Methoden verwendet. Bei der Klassifizierung von Datensätzen mit einer Vielzahl an Klassen wird im Regelfall die TOP-5 zusätzlich verwendet. Da sich das Klassifizierungsproblem in dieser Arbeit auf drei Klassen beschränkt, entfällt diese Möglichkeit. Die TOP-1 error rate beschreibt bei wie vielen Klassifizierungen das Neural Network die falsche Klasse vorhergesagt hat und steht im direkten Verhältnis zur Accuracy, welche beschreibt wie viele Klassen korrekt vorhergesagt wurden. Es gilt also:

$$TOP\text{-}1 = Vorhersagen_{Falsch} \div Vorhersagen_{Gesamt} \tag{17}$$

$$Accuracy = Vorhersagen_{Wahr} \div Vorhersagen_{Gesamt} \tag{18}$$

$$Accuracy = 1 \ - \ TOP\text{-}1 \tag{19}$$

Neben der TOP-1 error rate wird die Leistung der Architekturen auch mit den Metriken Precision (zu Deutsch: Präzision), Recall (zu Deutsch etwa: Widerruf) und

F1-Score bemessen. Die für diese Metriken relevanten Teile der Klassifizierung sind in den Abbildungen 19 und 20 dargestellt. Abbildung 19 zeigt die Verteilung der Daten nach der Vorhersage durch den Klassifizierungsalgorithmus. Im Kreis berfinden sich alle Datenpunkte, welche der Algorithmus als *Wahr* vorhergesagt hat. Außerhalb finden sich alle Punkte wieder, die als *Falsch* klassifiziert wurden. Alle Punkte auf der linken Hälfte des Bildes sind eigentlich *Wahr* und somit relevant für die Klassifizierung, während sich auf der rechten Seite ausschließlich falsche und somit irrelevante Datenpunkte befinden. Daher gelten alle Punkte auf der linken Seite außerhalb des Kreises als (*False negative*) und alle Punkte innerhalb des Kreises als (*True positive*). Auf der rechten Seite gilt dementsprechend, dass die Daten im Kreis (*False positive*) und die Punkte außerhalb (*True negative*) sind. Abbildung 20 zeigt nun die daraus resultierende Berechnung der unterschiedlichen Metriken. Precision errechnet sich aus dem Verhältnis der *True positive* Daten zu der Gesamtheit der positiven Daten $True\,positive + False\,positive$. Die Metrik spiegelt daher wieder sicher ein Algorithmus darin ist Daten richtig zu evaluieren. Die Metrik vernachlässigt jedoch wie hoch der Anteil der positiven Daten ist, der gefunden wird. Beispielsweise könnte ein Algorithmus eine Klasse lediglich ein Mal als richtig bewerten. Liegt er damit richtig so wäre der Precision-Wert 1, da ein *True positive* zusammen mit keinem *False positive* $1 \div (1 + 0) = 1$ ergibt. Daher wird als zweite Metrik der Recall herangezogen. Dieser errechnet sich aus dem Verhältnis der Anzahl an *True postive* Datenpunkten zu der Anzahl der relevanten Daten $True\,positive + False\,negative$ und veranschaulicht somit wie gut der Algorithmus die richtigen Klassen wiedererkennt. Der Abbildung der Accuracy dient zur Veranschaulichung der Unterschiede der drei Metriken, da in jedem Fall unterschiedliche Datenmengen herangezogen werden. Der F1-Score dient der Zusammenfassung beider Metriken (Precision und Recall), indem er das gewichtete harmonische Mittel abbildet. Der F1-Score errechnet sich wie in Gleichung 20 beschrieben für $\alpha = 1$ (Gleichung 21).

$$F_\alpha = (1 + \alpha^2) \cdot (Precision \cdot Recall) \div (\alpha^2 \cdot Precision + Recall) \tag{20}$$

$$F_1 = 2 \cdot (Precision \cdot Recall) \div (Precision + Recall) \tag{21}$$

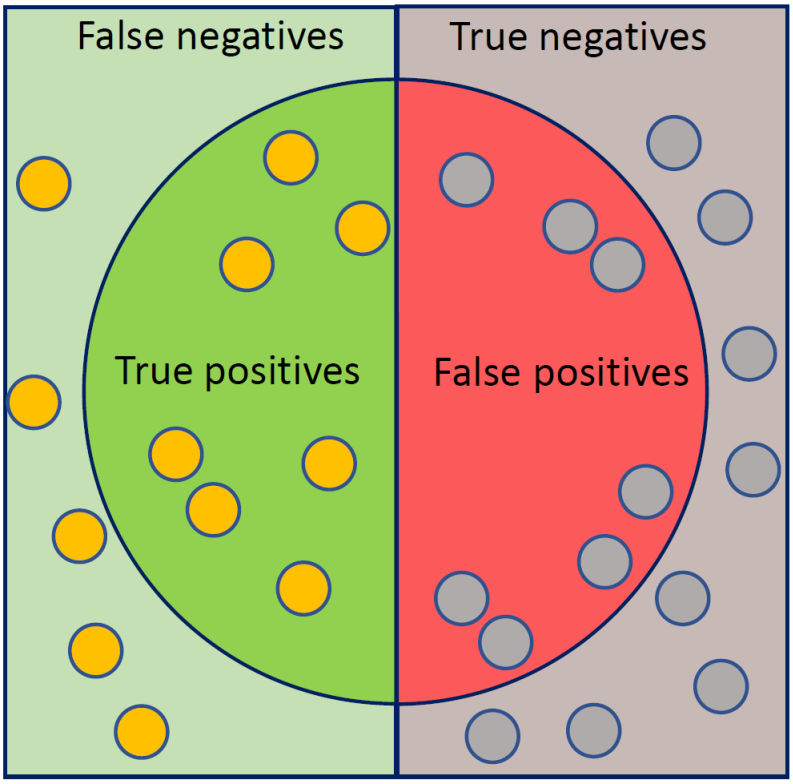

Abbildung 19: Verteilung der relevanten Elemente als Grundlage für Precision und Recall.

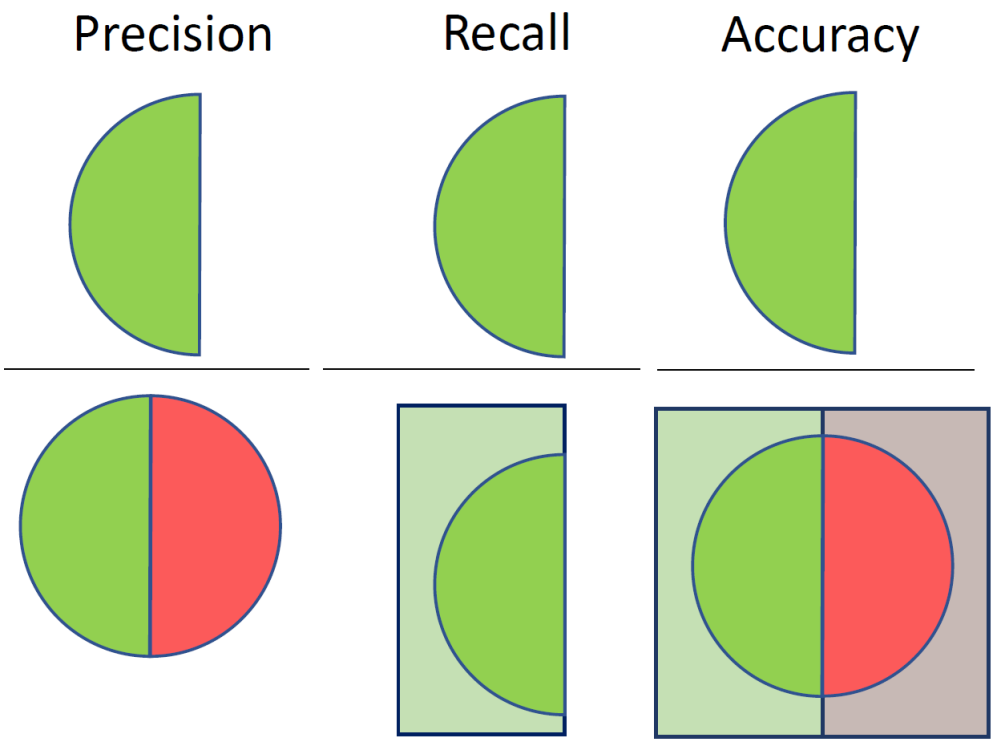

Abbildung 20: Schematische Darstellung der Errechnung von Precision, Recall und Accuracy.

5.3 Trainingsmethodik

CNNs können sich diverse Hyperparameter zu nutze machen um den Trainingserfolg zu erhöhen. Außerdem kann mit der Auswahl des Optimierungsalgorithmus (folgend Optimizer) auch Einfluss auf das Lernverhalten der Neural Networks genommen werden. Alle Architekturen verwenden den Adam-Optimizer, welcher in der gängigen Literatur immer wieder Anwendung findet. Der Adam-Optimizer ist ein Algorithmus der die Vorteile von AdaGrad und RMSProp kombiniert [10], indem er jedem Parameter eine Learning Rate (zu Deutsch: Lernrate) zuweist und diese im Verlauf des Trainings anpasst. Der wichtigste Hyperparameter in dieser Arbeit ist die Learning Rate. Sie bestimmt den Faktor, der bei der Errechnung der Gewichtsanpassungen durch Backpropagation genutzt wird und somit wie stark der Optimizer die Gewichte verändern kann. Die Learning Rate wurde je nach Modell und Aufbau etwas variiert und liegt aber stets zwischen 0,01 und 0,001. Im Laufe des Trainings werden alle Layer mittels der sogenannten *Kernel Regularization* (zu Deutsch etwa: Kern-Normalisierung) normalisiert. Kernel Regularization entspricht einem Hyperparameter, der für jeden Layer eines Neural Networks definiert wird. Dieser Hyperparameter ist ein festgelegter Wert auf die Loss Function addiert wird und die damit die Anpassung der Gewichte beeinflusst. Er soll unter anderem Over-

fitting verhindern. Für alle Netze wird in jedem Layer die Regularizierung mittels L2 regularization mit dem Wert $1 \times e^{-4}$. Für alle K Die Inception-SE-Architektur fordert zudem mit der Reduction Ratio (Reduktionsverhältnis) einen weiteren Hyperparameter. Dieser steuert den Zuwachs an Parametern durch die Fully-Connected Layer im Squeeze-and-Excitation Block der Architektur und sein Wert liegt in dieser bei 4. Das bedeutet, dass in dem ersten Fully-Connected Layer die Anzahl der Channel durch 4 geteilt wird um die Dimensionen der Ausgabe zu reduzieren und somit die Anzahl der benötigten Parameter zu verkleinern.

In dem Trainingsvorgang wird Data Augmentation (zu Deutsch: Datenvermehrung) verwendet. Dieser Vorgang verfolgt zwei Ziele. Zum einen soll die Anzahl der verfügbaren Daten erhöht werden, zum anderen sollen die Architekturen die Stärke der Armeen unabhängig von ihrer Lage auf dem Schlachtfeld evaluieren können. Beides wird angegangen indem alle Eingabematrizen in vier Versionen jeweils 90 Grad rotiert in den Datensatz eingespeist werden (folgend: Rotated-Variante). So muss die Architektur das Gefecht in vier Richtungen analysieren und ist gezwungen den Werten der Armeen eine höhere Aufmerksamkeit zu schenken. Als zweite Augmentierungsmethode wurden die Matrizen nur einmal dupliziert und um 180 Grad rotiert (folgend: Mirrored-Variante). Zusätzlich wurden die Label getauscht. Neben der Datenvermehrung, soll so überprüft werden, ob sich die Architekturen bei der Vorhersage der Gewinnerseite anders verhalten, oder ob ähnliche Ergebnisse zu erwarten sind.

Im Zuge der Evaluierung aller Architekturen wurde jede Architektur mit beiden Augmentierungsvarianten und einer Version ohne Augmentierung (folgend: NoAug-Variante) analysiert.

Die vorgestellte ResNet-Architektur wurde mit jeweils 90.000 Datensätzen für die Rotated-Variante und 30.000 Datensätzen für die Mirrored-Variante und die NoAug-Variante trainiert. Die NoAug-Variante wurde mit einer Learning Rate von 0,001 trainiert, die Rotated-Variante mit 0,01 und die Mirrored-Variante mit 0,005. Abbildung 21 zeigt den Verlauf der Verlustfunktion während des Trainings für alle drei Varianten. Sowohl bei der Rotated- als auch bei der Mirrored-Variante findet man einen plötzlichen Anstieg der Verlustwerte im Verlauf des Trainings. Dieser Ausschlag geht damit einher, dass das Netz ab einem bestimmten Zeitpunkt alle Daten zur selben Klasse evaluiert. Da die Mirrored-Variante vor dem Ausschlag die höchste Accuracy aufweist, wird das Modell aus Epoche 17 zur weiteren Auswertung genutzt.

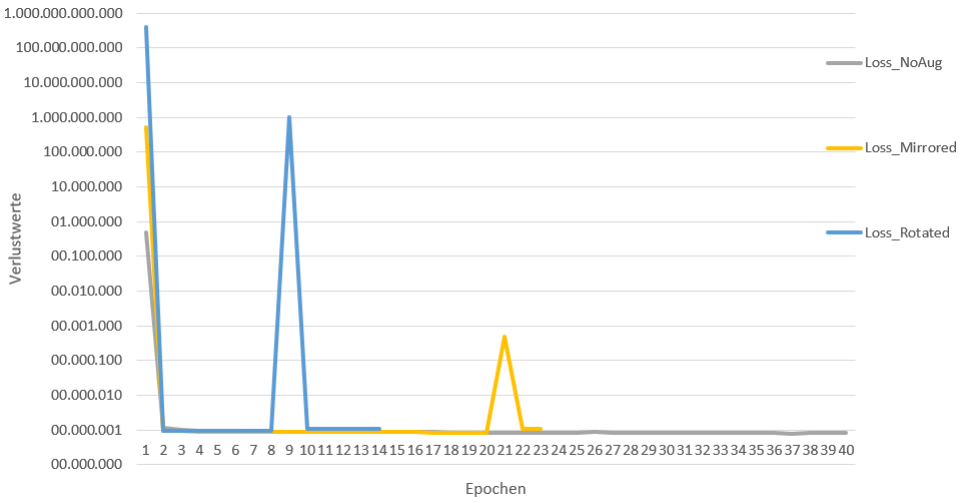

Abbildung 21: Verlustwerte der drei ResNet-Varianten.

Das All-Convolutional Net wurde in allen Varianten mit einer Learning Rate von 0,005 trainiert. Die Rotated-Variante und die Mirrored-Variante wurden mit 90.000 Datensätzen trainiert, die NoAug-Variante wurde aus Zeitgründen mit 30.000 Datensätzen. In Abbildung 22 werden die Verlustwerte aller Versionen dargestellt. Es zeigt sich ein fortlaufender Abstieg der Verlustwerte, was andeutet das, alle Varianten auf den Trainingsdaten gute Ergebnisse erzielen. Zieht man mit Abbildung 23 allerdings die Accuracy der Testdaten mit in Betracht, so zeigt sich dass alle Varianten nach der 10 Epoche zum Overfitten neigen. Es wird die Mirrored-Variante, zur weiteren Evaluierung herangezogen, da sie auf den Testdaten die beste Accuracy aufwies.

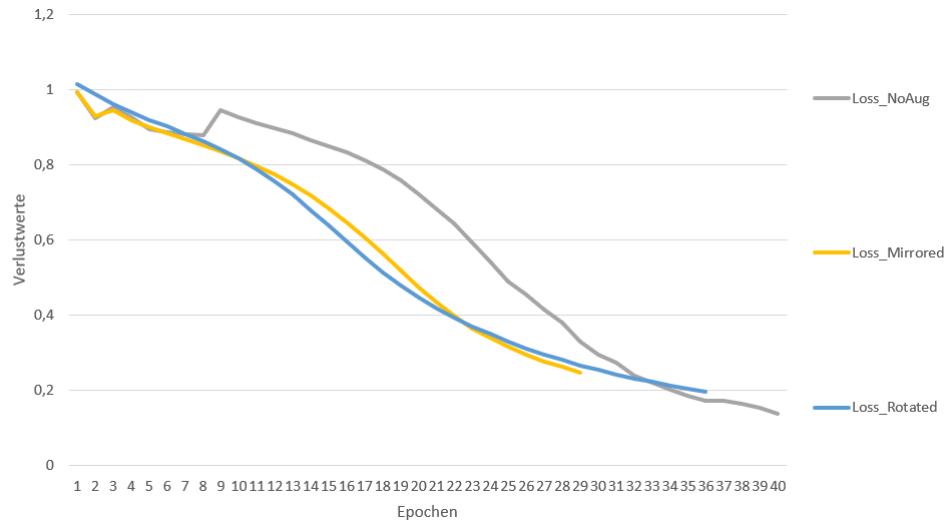

Abbildung 22: Entwicklung der Verlustwerte der All-Convolutional Net-Varianten während des Trainings.

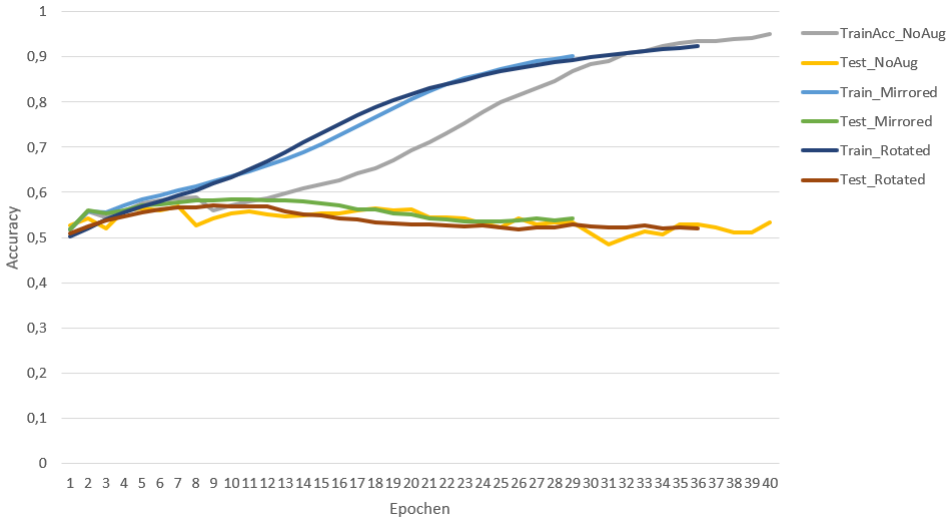

Abbildung 23: Entwicklung der Accuracy aller All-Convolutional Net-Varianten für Test- und Trainingsdaten.

Inception V4 Inception V4 wurde mit der Learning Rate 0,001 trainiert. Auf der Abbildung **??**TBD** sind die Werte der Accuracy aller Versionen eingetragen. Die X-Achse zeigt die Epochen und die Y-Achse stellt die Accuracy-Werte dar. In der Abbildung sind zwei Durchläufe ohne Augmentierung in blau (a) und orange (b) enthalten, sowie ein Durchgang mit 90 Grad Rotation in **TBD** (c) und ein Durchgang

Tabelle 7: Auflistung der Accuracy und des F1-Score auf den Testdaten für alle Architekturen.

Architektur	Accuracy	F1-Score
All-Convolutional Net	0,585	0,567
ResNet	0,611	0,601
Inception V4		
Inception V4 SE		

mit 180 Grad Rotation in **TBD** (d). Version (a) wurde mit 90.000 Datensätzen trainiert, während (b) lediglich mit 60.000 Datensätzen trainiert wurde. Die Versionen (a) und (b) zeigen einen ähnlichen Verlauf und tendieren zum Overfitten. (a) hat im Testdatensatz sein Maximum in Epoche 10 bei 0,5739. (b) findet sein Maximum eine Epoche später in Epoche 11 mit einem Wert von 0,5690. Beide Durchgänge folgen dem bei All-Convolutional Nets beschriebenen Muster des Overfittens, indem bei beiden Netzen nach erreichen der Maxima Test- und Trainings-Accuracy auseinandergehen. Während sich die Trainings-Accuracy erneut dem Wert 1 annähert schwankt die Test-Accuracy in beiden Fällen zwischen 0,50 und 0,53.

TBD Bild mit den Accuracies von Inception V4 **TBD**

Inception mit Squeeze-and-Excitation-Layer

5.4 Resultate

6 Konklusion

TBD FAZIT **TBD**

Ausblick Ein Großteil der Trainingsvorgänge neigte zum Overfitting ab der 60%-Grenze. In einer Fortführung der Arbeit kann mit einem größeren Datensatz und mehr Rechenleistung untersucht werden, ob die 60%-Grenze ohne Overfitting übertreten werden kann.

Der stärkere Einsatz von dreidimensionalen Convolutions, kann in weiteren Versuchen untersucht werden. In den gelaufenen Trainings waren alle Convolutions mit einem Filter der Dimension mit Tiefe 1 konfiguriert. Eine Untersuchung, ob eine Filtertiefe > 1 eine Steigerung der Accuracy liefert steht noch aus. Aufbauend auf variablen Größen der Filtertiefe könnte eine Unterteilung nach semantischen Gemeinsamkeiten in kleinere Blöcke eine Steigerung der Performance mit sich bringen, indem man gezielt z.B. alle Feature Layer die Einheitenwerte betreffen zusammen evaluiert. So könnten die Layer Einheiten-Typ, Lebenspunkte und Schild zusammen verarbeitet werden und Layer, welche die Einheiten einem Spieler zuordnen im späteren Verlauf separat verrechnet werden.

In den Untersuchungen von Sánchez-Ruiz [14] zeigt dieser klar eine starke Verbesserung der Vorhersagegenauigkeit mit fortschreitendem Verlauf des Gefechts auf. Eine Untersuchung der Gefechte auf Basis einer Zeitreihe der ersten 5-10 Sekunden, könnte z.B. mit einem Long short-term memory network (LSTM-Network, zu Deutsch: langes Kurzzeitgedächtnis Netzwerk) analysiert werden und damit die Accuracy der Vorhersage verbessern.

Aktuell beziehen sich die Vorhersagen noch auf den Bildschirm, welcher dem Spieler zur Verfügung steht und finden basierend auf Daten von einem neutralen Beobachter statt, der alle Einheiten – auch die unsichtbaren – sehen kann. Als nächsten Schritt kann die Minimap einbezogen werden, welche einen Überblick über das gesamte Sichtfeld der Spieler liefert. Zusätzlich kann die Rolle eines Spieler eingenommen werden, der nur die Einheiten in seinem Sichtfeld sieht. Außerdem können Höhenunterschiede in die Custom Map eingebaut werden, welche die Ausgangssituation des Gefechts verändern.

Einhergehend mit einer Erweiterung der Modelle kann die Performance mit der Implementation in einen Bot evaluiert und mit gängigen und bereits implementierten Entscheidungsalgorithmen verglichen werden. Unter Einbindung von Bots oder Scripts können die Modelle zusätzlich um den Einsatz von Fähigkeiten und das Nutzen von Micromanagement erweitert werden.

Literatur

[1] David Churchill und Michael Buro. "Portfolio greedy search and simulation for large-scale combat in starcraft". In: *2013 IEEE Conference on Computational Inteligence in Games (CIG), Niagara Falls, ON, Canada, August 11-13, 2013*. 2013, S. 1–8.

[2] David Churchill, Abdallah Saffidine und Michael Buro. "Fast Heuristic Search for RTS Game Combat Scenarios". In: *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-12, Stanford, California, USA, October 8-12, 2012*. 2012.

[3] Ian J. Goodfellow, Yoshua Bengio und Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[4] Kaiming He u. a. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385.

[5] Ian Helmke, Daniel Kreymer und Karl Wiegand. "Approximation Models of Combat in StarCraft 2". In: *CoRR* abs/1403.1521 (2014). arXiv: 1403.1521.

[6] Jie Hu, Li Shen und Gang Sun. "Squeeze-and-Excitation Networks". In: *CoRR* abs/1709.01507 (2017).

[7] Sergey Ioffe und Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 2015, S. 448–456.

[8] Nadir Jeevanjee. *An Introduction to Tensors and Group Theory for Physicists*. Birkhäuser, Boston, 2011.

[9] R. A. Kilmer. "Applications of Artificial Neural Networks to combat simulations". In: *Mathematical and Computer Modelling* 23.1 (1996), S. 91–99.

[10] Diederik P. Kingma und Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015.

[11] Keiron O'Shea und Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *CoRR* abs/1511.08458 (2015).

[12] Glen Robertson und Ian D. Watson. "A Review of Real-Time Strategy Game AI". In: *AI Magazine* 35.4 (2014), S. 75–104.

[13] Mikayel Samvelyan u. a. "The StarCraft Multi-Agent Challenge". In: *CoRR* abs/1902.04043 (2019). arXiv: 1902.04043.

[14] Antonio A. Sánchez-Ruiz. "Predicting the Outcome of Small Battles in StarCraft". In: *Workshop Proceedings from The Twenty-Third International Conference on Case-Based Reasoning (ICCBR 2015), Frankfurt, Germany, September 28-30, 2015.* 2015, S. 33–42.

[15] Jost Tobias Springenberg u. a. “Striving for Simplicity: The All Convolutional Net”. In: *CoRR* abs/1412.6806 (2014).

[16] Marius Adrian Stanescu, Nicolas A. Barriga und Michael Buro. “Using Lanchester Attrition Laws for Combat Prediction in StarCraft”. In: *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2015, November 14-18, 2015, University of California, Santa Cruz, CA, USA.* 2015, S. 86–92.

[17] Marius Stanescu u. a. “Predicting Army Combat Outcomes in StarCraft”. In: *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-13, Boston, Massachusetts, USA, October 14-18, 2013*. 2013.

[18] Christian Szegedy, Sergey Ioffe und Vincent Vanhoucke. “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”. In: *CoRR* abs/1602.07261 (2016). arXiv: 1602.07261.

[19] Christian Szegedy u. a. “Going Deeper with Convolutions”. In: *CoRR* abs/1409.4842 (2014).

[20] Oriol Vinyals u. a. “StarCraft II: A New Challenge for Reinforcement Learning”. In: *CoRR* abs/1708.04782 (2017).

[21] Izhar Wallach, Michael Dzamba und Abraham Heifets. “AtomNet: A Deep Convolutional Neural Network for Bioactivity Prediction in Structure-based Drug Discovery”. In: *CoRR* abs/1510.02855 (2015).

[22] Stefan Wender und Ian D. Watson. “Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft: Broodwar”. In: *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012, Granada, Spain, September 11-14, 2012*. 2012, S. 402–408.

[23] Jianxin Wu. “Introduction to Convolutional Neural Networks”. In: *National Key Lab for Novel Software Technology Nanjing University, China.* 2017, S. 5.