

# **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 April 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.	<input type="checkbox"/>	<input type="checkbox"/>
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	<input type="checkbox"/>	<input type="checkbox"/>
Der Text dieser Arbeit ist unter einer Creative Commons Lizenz (CC BY-SA 4.0) verfügbar.	<input type="checkbox"/>	<input type="checkbox"/>
Der Quellcode ist unter einer GNU General Public License (GPLv3) verfügbar.	<input type="checkbox"/>	<input type="checkbox"/>
Die erhobenen Daten sind unter einer Creative Commons Lizenz (CC BY-SA 4.0) verfügbar.	<input type="checkbox"/>	<input type="checkbox"/>

.....  
(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 Machine Learning (zu Deutsch: Maschinelles Lernen) Algorithmen zur Vorhersage von Gefechts-Ausgängen zweier feindlicher Armeen in dem Echtzeit-Strategiespiel StarCraft II. Das Ziel der Arbeit ist die Untersuchung, ob existierende Image Classification (zu Deutsch: Bildklassifikation) Algorithmen 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 CNN, 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* [17] 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 [19]).

Gefechte in StarCraft II bestehen aus zwei oder mehr Armeen. Jede Armee wird von einem unterschiedlichen 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 angemessenen 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. 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 Umweltfaktoren bereit – im Fall von StarCraft II die Anzahl der Einheiten, ihre Positio-



nen 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 der 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 den 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 müssen.

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 5.2 analysiert und verglichen.

## 1.1 StarCraft II

StarCraft II ist ein Echtzeit-Strategiespiel entwickelt und veröffentlicht von Blizzard Entertainment. Es ist überwiegend deterministisch<sup>1</sup> und 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  $U = \{u_1, u_2, \dots, u_n\}$ .

Das Spiel läuft im Allgemeinen in Echtzeit ab, kann allerdings auf Zeitschritte abstrahiert werden, wo in jedem Zeitschritt die gegebenen Befehle den kommenden Zeitschritt beeinflussen. Daher kann der Spielablauf als eine diskrete Serie von Zuständen repräsentiert werden. Intern läuft das Spiel mit einer Rate von 20 Zuständen pro Sekunde.

Jede Einheit besitzt eine allgemeine Beschreibung die ihr einen Typ  $t(u_i)$  zuweist.  $t(u_i)$  kann einer von drei Typen sein  $t(u_i) \in \{\text{Bodeneinheit, Lufteinheit, kleine Bodeneinheit}\}$ . Zusätzlich kann jede Einheit mit zwei weiteren Attributen  $at(u_i), st(u_i)$

<sup>1</sup>Die hauptsächlichen Quellen für Zufälligkeiten sind laut DeepMind *Angriffsgeschwindigkeit* und *Aktualisierungsreihenfolge*. Diese Zufälligkeiten können durch das manuelle setzen eines Zufallswertes ausgemerzt werden [18] und in die Berechnungen eingearbeitet werden.

versehen werden, die ihre weiteren Eigenschaften spezifizieren. Eine Einheit kann eine leichte oder eine gepanzerte Einheit sein ( $at(u_i)$ ) und außerdem dem biologischen oder mechanischen Typ angehören ( $st(u_i)$ ). Es gibt vereinzelt Ausnahmen in denen Einheiten keinem Typ oder beiden Typen angehören.

Einheiten bewegen sich auf einer 2D Karte und können in Abhängigkeit von der Zeit einer Position zugeordnet werden. 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  $x \leq B, y \leq H$  gilt.

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 für 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 können zum Beispiel lediglich Bodeneinheiten angreifen, während andere nur Lufteinheiten angreifen können. 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  $bd^{at}(u_i)$ , welches auf dem  $at$ -Attribut 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 wür-

den.

## 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. Die Knoten in einem Layer senden somit Signale an Knoten in anderen Layern, können allerdings keine Signale an Knoten des selben Layers senden. Des weiteren zeichnen sich feed-forward neural networks dadurch aus, dass ein Knoten eine Aktivierung immer nur zu dem Layer weitersendet, welcher als sein Nachfolger definiert ist. Es werden keine Aktivierungen an die voranstehenden Layer zurückgesendet.

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.

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

Eingabeschicht      Versteckte Schicht      Ausgabeschicht

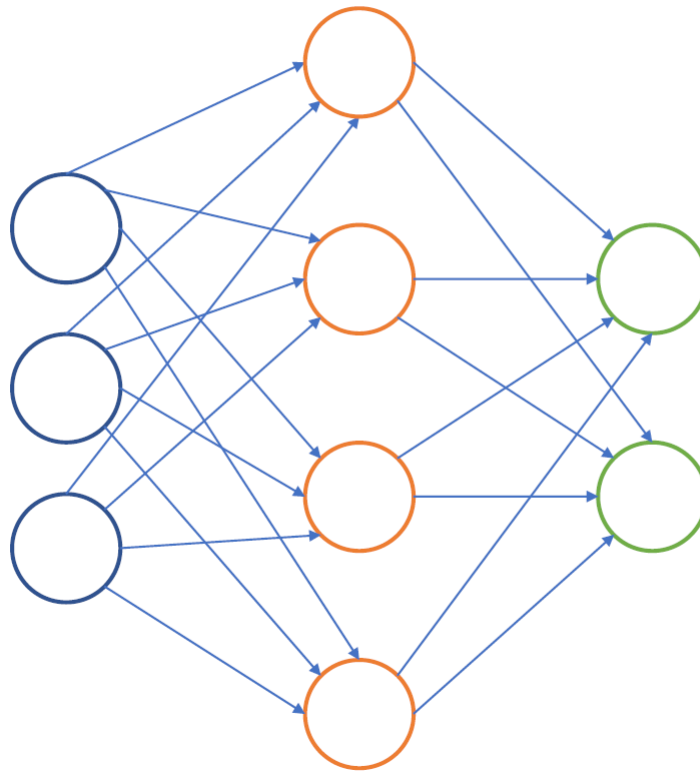


Abbildung 1: Topologie eines einfachen ANNs.

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:

$$f(g_1(h_1(x), h_2(x)), g_2(h_2(x), h_3(x))) \quad (1)$$

**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

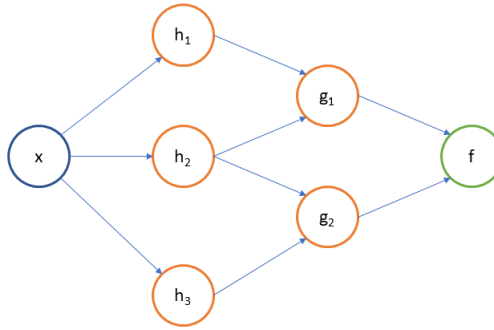


Abbildung 2: Grafische Darstellung der zusammengesetzten Funktion  $f$  aus Gleichung 1.

sind, und eines Bias  $\mathbf{b}$ , der dem entsprechenden Neuron zugewiesen ist.

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

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:

$$F(\mathbf{x}) = (\mathbf{W} \cdot \mathbf{x}) + \mathbf{b}^T \quad (3)$$

In Gleichung 3 sind  $\mathbf{W}$  und  $\mathbf{b}$  lernbare Parameter, welche vom Optimierungsalgorithmus angepasst werden können. Sie stehen in der Gleichung für die Gewichte( $\mathbf{W}$ ) und den Bias( $\mathbf{b}$ ) des entsprechenden Layers.

**Rectified Linear Units Layer** In ANNs finden eine Vielzahl von Aktivierungsfunktionen Anwendung. Im Folgenden wird 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 das nächste Layer weitergereicht wird [21].

**Normalization Layer** Das Normalization Layer (zu Deutsch: *Normalisierungsschicht*) wird zur Normalisierung der Aktivierungen des vorherigen Layers genutzt. 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].

**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  $(\mathbf{x}, \mathbf{y})$  einer Eingabe  $\mathbf{x}$  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 [21]:

$$z = \frac{1}{2} \|(\mathbf{x}, \mathbf{y}) - \hat{y}\|^2 \quad (4)$$

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

$$softmax(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad (5)$$

Eine vollständige Funktion für das in Abbildung 1 gezeigte ANN wäre:

$$f(\mathbf{x}) = softmax(\mathbf{W}_2 \cdot relu(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

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  $4 \times 3$  Matrix 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  $4 \times 1$ . 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  $2 \times 4$  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  $2 \times 1$ .

## 2.2 Convolutional Neural Networks

Ein Convolutional Neural Network (CNN) ist ein tiefes, feed-forward ANN. 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 befindet sich der Hidden Layer, in dem die Informationen aus den Eingabedaten hauptsächlich verarbeitet werden. Der Hidden Layer kann 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.

**Convolutional layers** Das Convolutional Layer ist einer der Kernbestandteile eines CNNs. Der Layer verfügt über einen Filter, bemessen durch einen Tensor des selben Ranges wie die Eingabedaten, welcher sogenannte Feature (zu Deutsch: *Merkmale*) in den Eingabedaten detektiert. Filter sind durch ihre rezeptiven Felder charakterisiert, welche die Größe der zeitgleich untersuchten Areale bestimmen [9]. Ein Beispiel für das rezeptive Feld eines Neurons könnte eine  $M \times N$  große Fläche seine, welche kleinere oder gleich große Abmessungen haben sollte wie die Dimensionen der Eingabedaten. Da die Tiefe des Filters  $D_f$  äquivalent zu der Tiefe der Eingabedaten  $D_i$  ist, ergibt sich  $D_f = D_i$ . Eingabewerte mit Dimensionen  $10 \times 10 \times D_i$  würde demnach in einem  $M \times N \times D_i$  dimensionierten Filter mit  $M, N \leq 10$  resultieren.

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. Aufbauend auf dem Beispiel im vorherigen Absatz ergibt sich eine Ausgabedimension von  $6 \times 6$  bei einem  $10 \times 10$  Eingabewert und einem  $5 \times 5$  Filter. Im Allgemeinen resultiert eine Eingabe mit Bemessung  $H^l \times W^l \times D^l$  und einem Filter der Größe  $H \times W \times D^l \times D$  in einer Ausgabe der Dimension  $(H^l - H + 1) \times (W^l - W + 1) \times D$ . In dieser Notation verweist  $l$  auf den vorherigen Layer [21].

Sollte eine Ausgabe mit den selben Dimensionen wie die Eingabe 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 der Eingabematrix 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 der Matrix verschoben wird. Wenn jede mögliche Position auf der Matrix



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 [7] beschrieben werden. In der Formel ist der stride-Parameter  $s = 1$  und es findet kein padding statt. Die Notation folgt der bereits etablierten Notation und  $H^{l+1}$ ,  $W^{l+1}$  und  $D^{l+1}$  beziehen sich auf Höhe, Breite und Tiefe der Ausgabematrix.

$$y \in \mathbb{R}^{H^{l+1} \times W^{l+1} \times D^{l+1}} \quad (6)$$

mit  $H^{l+1} = H^l - H + 1$ ,  $W^{l+1} = W^l - W + 1$ , and  $D^{l+1} = D$  [21].

$$y_{i',j',d_i} = \sum_{i=0}^H \sum_{j=0}^W \sum_{d=0}^{D^l} f_{i,j,d,d_i} \cdot x_{i'+i,j'+j,d}^l \quad (7)$$

$f$  bezeichnet die Menge aller Filter und  $x^l$  ist die Ausgabe des vorherigen Layers. Die Gleichung 7 wird über die komplette Tiefe  $D^l$  der Eingabe wiederholt ( $0 \leq d_i \leq D^l$ ). 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 [21].

Im Bezug auf hoch dimensionierte Eingabewerte können fully-connected neural networks räumliche Abhängigkeiten nicht nutzen, da jedes Neuron vom jedem Neuron des vorherigen Layers Informationen sammelt und so immer alle Aktivierungen verarbeitet. Um räumliche Informationen auszuwerten nutzen CNNs ein Pattern zur Erfassung lokaler Zusammenhänge. Aufgrund dieses Patterns, ist jedes Neuron nur mit einer kleinen Menge der Eingabegröße verbunden. Hier wird nun das rezeptive Feld benötigt, da es den Hyperparameter betitelt, welcher die räumliche Bemessung des Sichtfeldes eines jeden Neurons abbildet.

**Pooling layers** Pooling Layer (zu Deutsch etwa: *bündelnde Schicht*) werden auch als downsampling Layer bezeichnet. Ihre Aufgabe besteht darin die Ausgabewerte der vorherigen Layers in einen Wert zu kombinieren und somit die Dimensionen der weitergereichten Daten zu reduzieren. Abbildung 3 zeigt eine bildliche Erklärung des maxpool Layers.

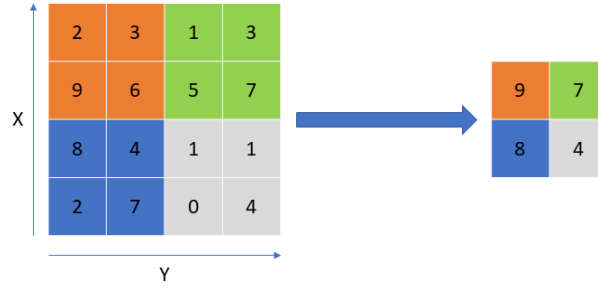


Abbildung 3: Einfache Veranschaulichung des maxpool Layers mit  $2 \times 2$  stride-Parameter.

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 der Eingabe in Abhängigkeit von seinen Parametern. So reduziert ein  $2 \times$  pooling Layer eine  $8 \times 8$  dimensionierte Eingabe auf die Dimension  $4 \times 4$ , da es die Eingabematrix in sich nicht überlappende  $2 \times 2$  Quadrate unterteilt und für jedes Quadrat einen Wert errechnet, welcher in die Ausgabematrix 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 [21]:

$$H^{l+1} = \frac{H^l}{H}, W^{l+1} = \frac{W^l}{W}, D^{l+1} = D^l. \quad (8)$$

Ein maxpooling Layer kann wie folgt definiert werden:

$$y_{i',j',d} = \max_{0 \leq i < H, 0 \leq j < W} x_{i',H+i,j',W+j,d}^l \quad (9)$$

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

In der Vergangenheit wurden diverse Arbeiten mit Bezug auf die Vorhersage von Kampfszenarien verfasst. Kilmer [8] 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.

Da die Generierung reeller Daten aufwendig ist und solche Simulationen immer in Echtzeit ablaufen müssen, werden vermehrt Computersimulationen genutzt um Kampfszenarien zu erzeugen. Die Computerspiele StarCraft und StarCraft II waren schon mehrfach Anschauungsobjekt im Bezug auf Schlachtensimulation und der Vorhersage von Gefechtsausgängen. Robertson und Watson [10] 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. Churchill und Buro [1], Churchill, Saffidine und Buro [2], Sánchez-Ruiz-Granados [12], Stanescu, Barriga und Buro [14], Stanescu u. a. [15] und Wender und Watson [20] nutzen für ihre Beiträge die Spielumgebung von StarCraft, während Helmke, Kreymer und Wiegand [5] und Samvelyan u. a. [11] ihre Methoden in der Umgebung von StarCraft II entwickeln und testen. Die Arbeiten von Churchill und Buro [1], Churchill, Saffidine und Buro [2] und Helmke, Kreymer und Wiegand [5] beschäftigen sich mit der Vorhersage von Gefechtsausgängen unter Verwendung von Simulationsverfahren. Sánchez-Ruiz-Granados [12], Stanescu, Barriga und Buro [14] und Stanescu u. a. [15] nutzen unterschiedliche Machine Learning Verfahren um Gefechte vorherzusagen.

Stanescu u. a. [15] stellt ein Modell vor, auf dessen Basis ein Lernalgorithmus die offensiven und defensiven Feature Values (zu Deutsch: *Eigenschaftswerte*) jeder Einheit erlernt. Das Modell basiert auf offensiven und defensiven Merkmalen jeder Einheit, welche zu offensiven und defensiven Eigenschaften einer Armee verrechnet werden. 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.

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 Trefferpunktregeneration, Einheitenkollision und Reisezeit von Projektilen nicht in die Evaluierung einbezogen.

Stanescu et al. Stanescu, Barriga und Buro [14] 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 Lanchester wird ein Algorithmus vorgestellt, welcher durch vergangene Gefechte seine Parameter anpassen kann. 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-Granados [12] beschränkt sich auf die Vorhersage der Gefechte und vergleicht die Präzision der verschiedenen Machine Learning Algorithmen. 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 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.

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 Reinforcement Learning (zu Deutsch: *Bestärkendes Lernen*) in StarCraft Anwendung finden kann [20]. 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 jedoch Einheiten-Fähigkeiten, sowie Einheitenverbesserungen vernachlässigt. Außerdem werden Einheitentypen,

wie nicht modelliert. 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. [11] 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.

## 4 Architekturen

In diesem Abschnitt werden die verwendeten Architekturen vorgestellt.

### 4.1 Problemstellung

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 mit Abmessung  $8 \times 84 \times 84 \times 1$  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  $2 \times 2$  durch Average-Pooling reduziert. Infolgedessen erhält der Fully-Connected Layer 2 Werte pro Armee mit denen er weiter rechnen kann.

### 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 berücksichtigt sowohl die Offensive Kampfkraft der beiden Armeen, als auch die defensiven Eigenschaften jeder Armee.

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. So sind manche Einheiten beispielsweise effektiver gegen Einheiten welche dem mechanischen Typ angehören. 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, enemies)
  ▷ jeder Bonustyp beinhaltet einen Wert type, als Identifikator und einen Wert
  damage, der den Bonusschaden spezifiziert.
2:   for all bt ∈ unit.bonustype do
3:     if Anzahl der Gegner mit Bonustyp bt.type > 0 then
      ▷ berechne anteilig den Bonusschaden der aktuellen Einheit
4:       factor ← Anzahl Gegner mit bt.type ÷ Gesamtzahl Gegner
5:       bonus ← bonus + bt.damage × factor
6:     end if
7:   end for
8:   return unit.basedamage + bonus
9: 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 herangezogen. 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

---

**Algorithmus 2** Berechnung der Powervalue nach Angriffs-Typen und des gewichteten Powervalue

---

```
1: procedure ARMYPOWERVALUES(army, opposingarmy)
2:   for all unit  $\in$  army do
3:     if unit.attacktype = 'Ground' then
4:       powervalue.ground  $\leftarrow$  powervalue.ground +
        POWERVALUE(unit, opposingarmy)
5:     else
6:       powervalue.air  $\leftarrow$  powervalue.air +
        POWERVALUE(unit, opposingarmy)
7:     end if
8:   end for
    $\triangleright$  powervalue enthält die Werte für ground und air
9:   return powervalue
10: end procedure
11: powervalue = ARMYPOWERVALUES(army, opposingarmy)
12: groundshare = Anzahl Gegner Typ Boden  $\div$  Gesamtzahl Gegner
13: airshare = Anzahl Gegner Typ Luft  $\div$  Gesamtzahl Gegner
14: weightedpowervalue  $\leftarrow$  powervalue.ground * groundshare + powervalue.air *
    airshare
```

---

---

**Algorithmus 3** Berechnung der Überlebensfähigkeit nach Typen

---

```
1: procedure SURVIVABILITY(units)
2:   for all unit  $\in$  units do
3:     if unit.type = 'Ground' then
4:       defense.ground  $\leftarrow$  defense.ground + unit.hitpoints + unit.shield
5:     end if
6:     if unit.type = 'Air' then
7:       defense.air  $\leftarrow$  defense.air + unit.hitpoints + unit.shield
8:     end if
9:   end for
10:  return defense
11: 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

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 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 zeitli-

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

chen 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(armya, armyb)
2:   if  $\neg \text{armya.fightable} \wedge \neg \text{armyb.fightable}$  then
3:     return 'Remis'
4:   end if
5:   if  $\text{armya.fightable} \wedge \neg \text{armyb.fightable}$  then
6:     if  $\text{armya.powervalue.ground} * K \geq \text{armyb.defense.ground} \wedge$   

 $\text{armya.powervalue.air} * K \geq \text{armyb.defense.air}$  then
7:       return 'Team A siegt'
8:     else
9:        $\triangleright$  Team A schafft Sieg nicht innerhalb der Zeit
10:      return 'Remis'
11:    end if
12:  end if
13:  if  $\neg \text{armya.fightable} \wedge \text{armyb.fightable}$  then
14:    if  $\text{armyb.powervalue.ground} * K \geq \text{armya.defense.ground} \wedge$   

 $\text{armyb.powervalue.air} * K \geq \text{armya.defense.air}$  then
15:      return 'Team B siegt'
16:    else
17:       $\triangleright$  Team B schafft Sieg nicht innerhalb der Zeit
18:      return 'Remis'
19:    end if
20:  end if
21:   $\text{armyvaluea} \leftarrow \text{armya.weightedpowervalue} + \text{armya.defense.ground} +$   

 $\text{armya.defense.air}$ 
22:   $\text{armyvalueb} \leftarrow \text{armyb.weightedpowervalue} + \text{armyb.defense.ground} +$   

 $\text{armyb.defense.air}$ 
23:  if  $\text{armyvaluea} > \text{armyvalueb}$  then
24:    return 'Team A siegt'
25:  else
26:    return 'Team B siegt'
27:  end if
28: end procedure
```

---

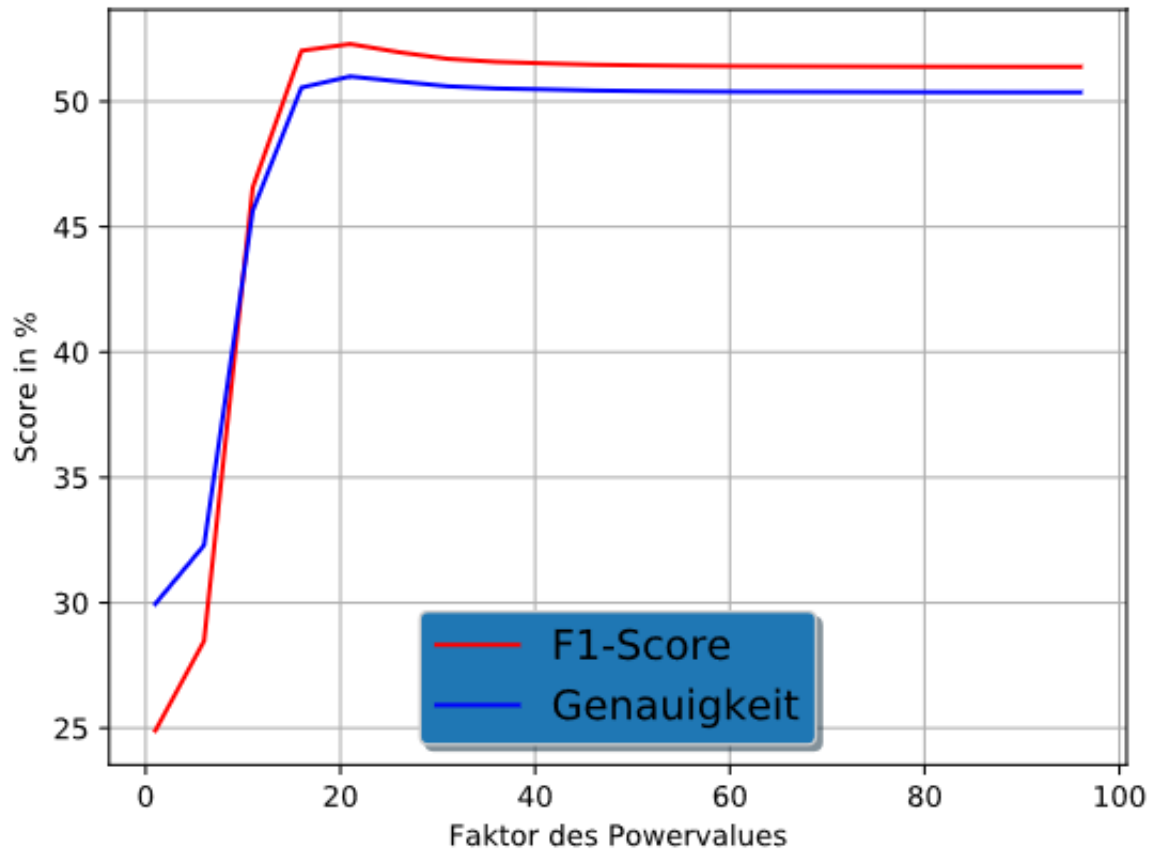


Abbildung 4: Accuracy und F1-Score für  $K \in [6, 96]$ .

### 4.3 All-Convolutional Net

Das All-Convolutional Net von Springenberg et al. [13] 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 5 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.

# All-Convolutional Neural Network

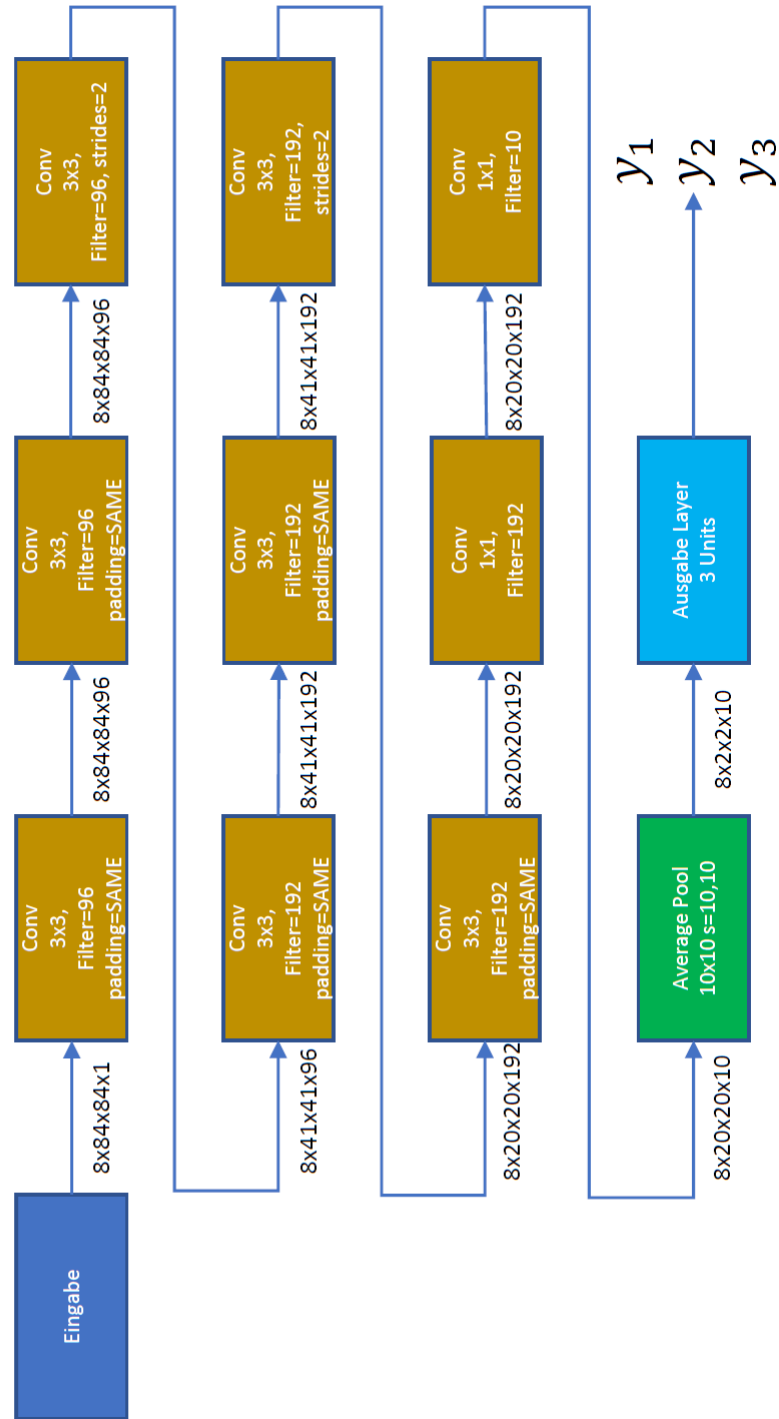
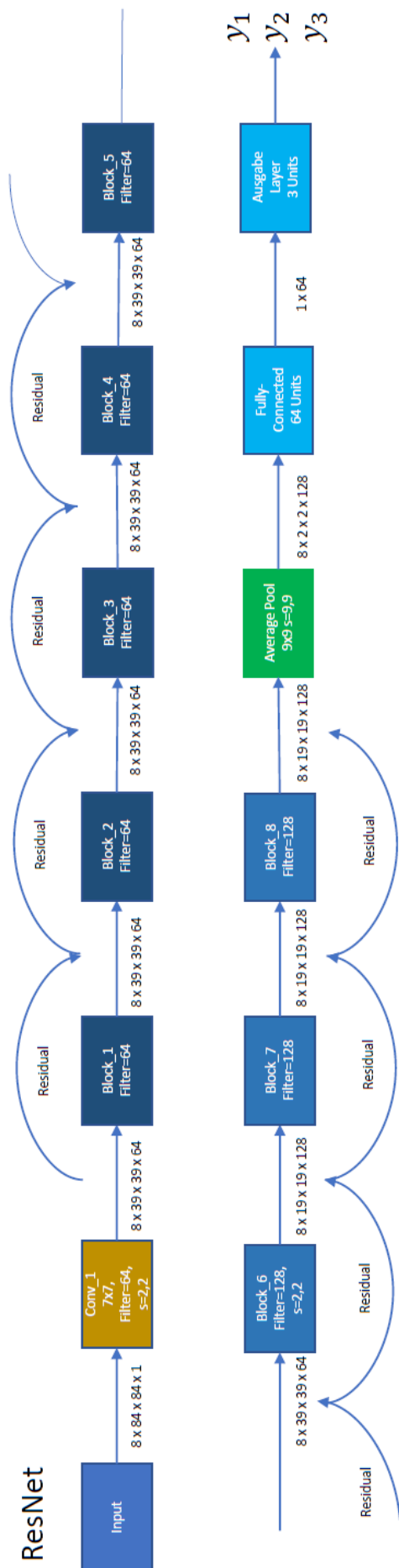


Abbildung 5: Architektur des All-Convolutional CNN.

## 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 Residual Neural Network shortcut connections eingefügt, welche ausschließlich als Identitätsfunktion dienen.

Abbildung 6 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 5 listet die Anzahl der Parameter auf.



Block 1 bis 5, 7, 8

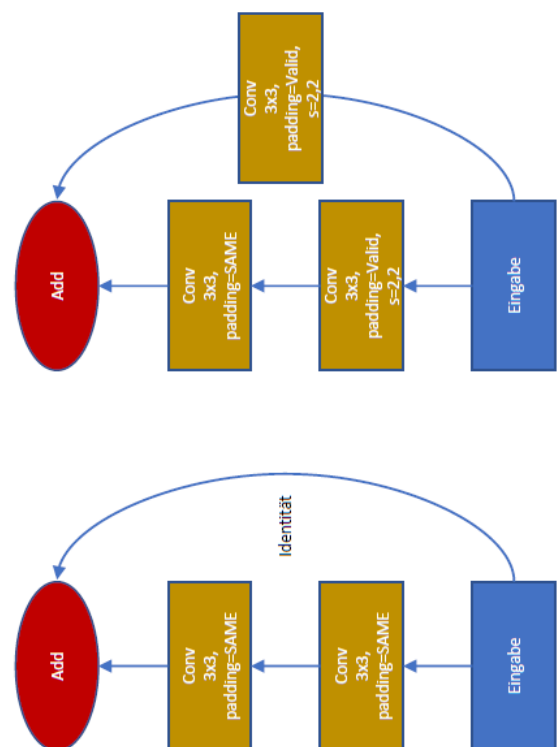


Abbildung 6: ResNet mit acht Residual Blöcken.



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 [16]. Sie führt drei verschiedene Module, sowie zwei Reduction-Blöcke als Bausteine des Netzes ein. Die Zusammensetzung der Bausteine ist in den folgenden Abbildungen 7, 8, 9 und 10 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 [16].

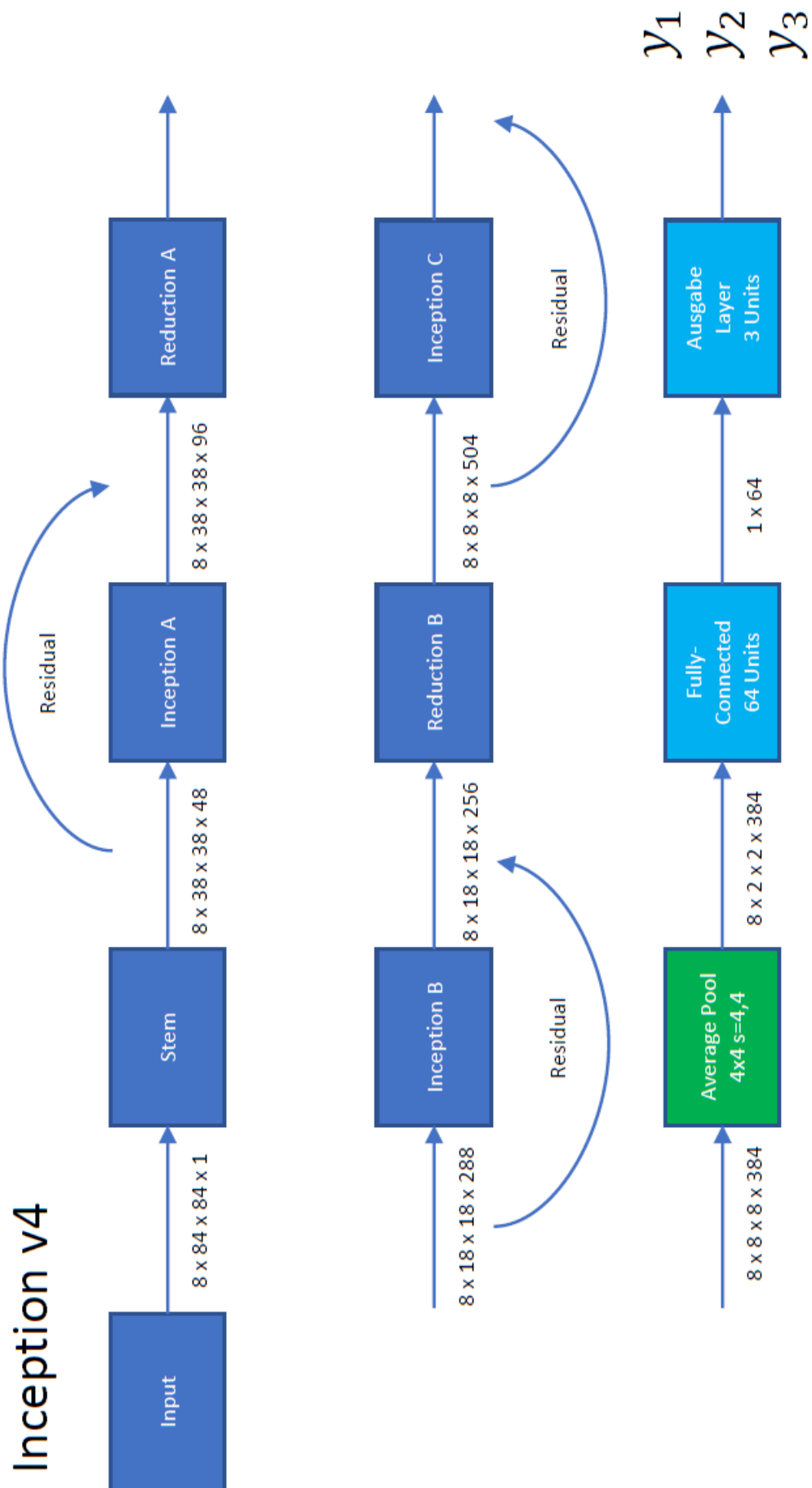


Abbildung 7: Übersicht der Inception V4-Architektur.

Abbildung 7 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 [16] 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 8 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 9) 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 10) 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. Es folgen Inception-B (Abbildung 9) und Reduction-B (Abbildung 10), 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 9). 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.

# STEM

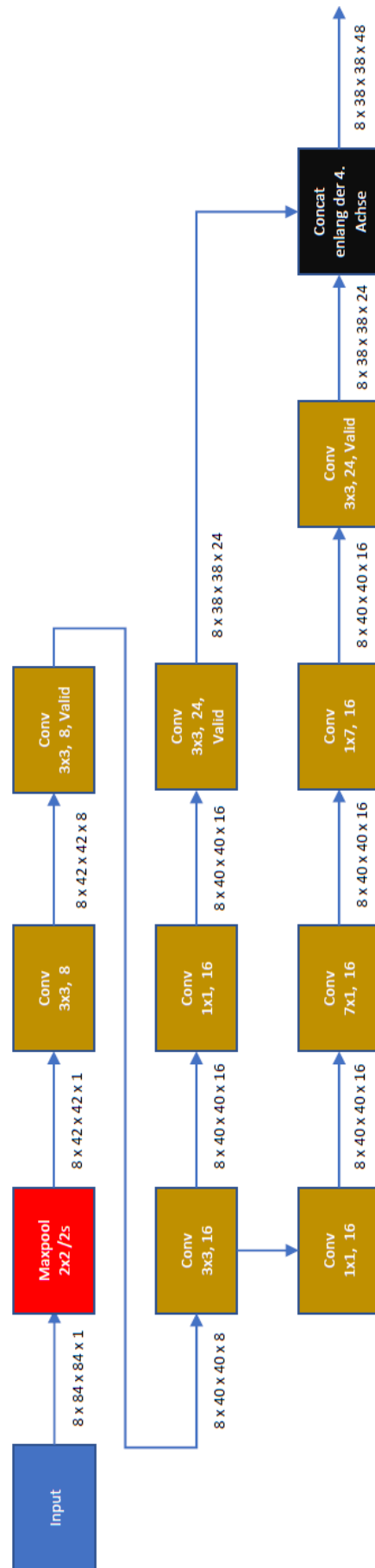


Abbildung 8: Aufbau des STEM-Moduls.

Der STEM-Block aus Abbildung 8 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 [16] 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  $8 \times 38 \times 38 \times 24$  eine einzige Ausgabe mit  $8 \times 38 \times 38 \times 48$  entsteht. Diese wird an das Inception-A-Modul weitergegeben.

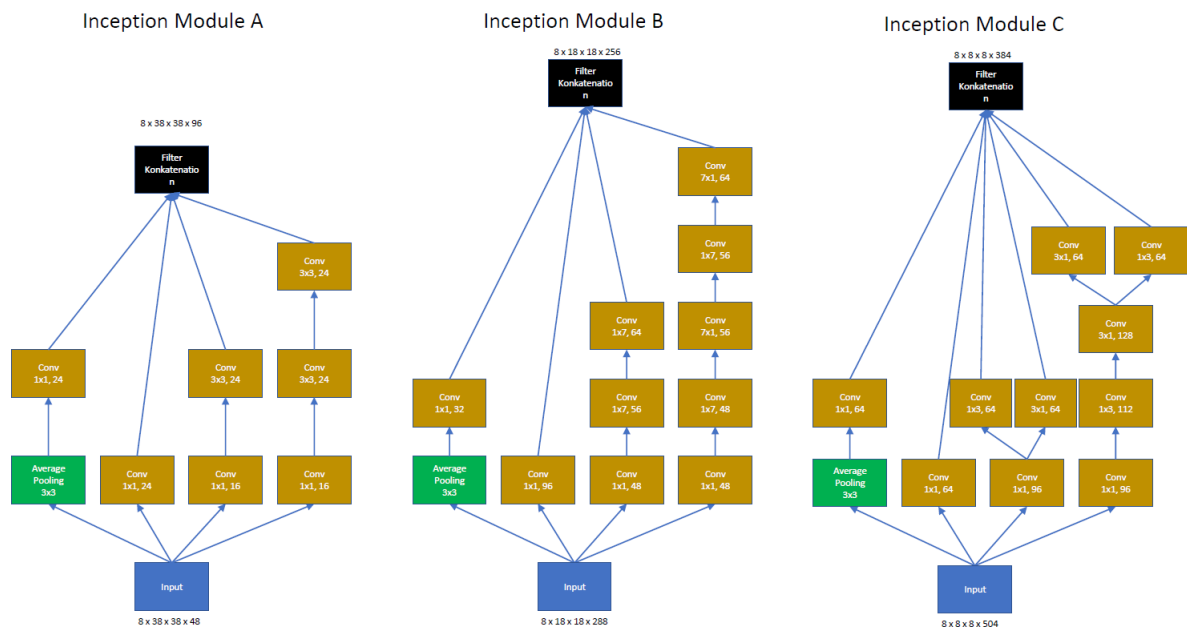


Abbildung 9: Übersicht der Inception V4-Module A,B und C.

Die Inception-Module (Abbildung 9) 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 unterscheiden.

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.

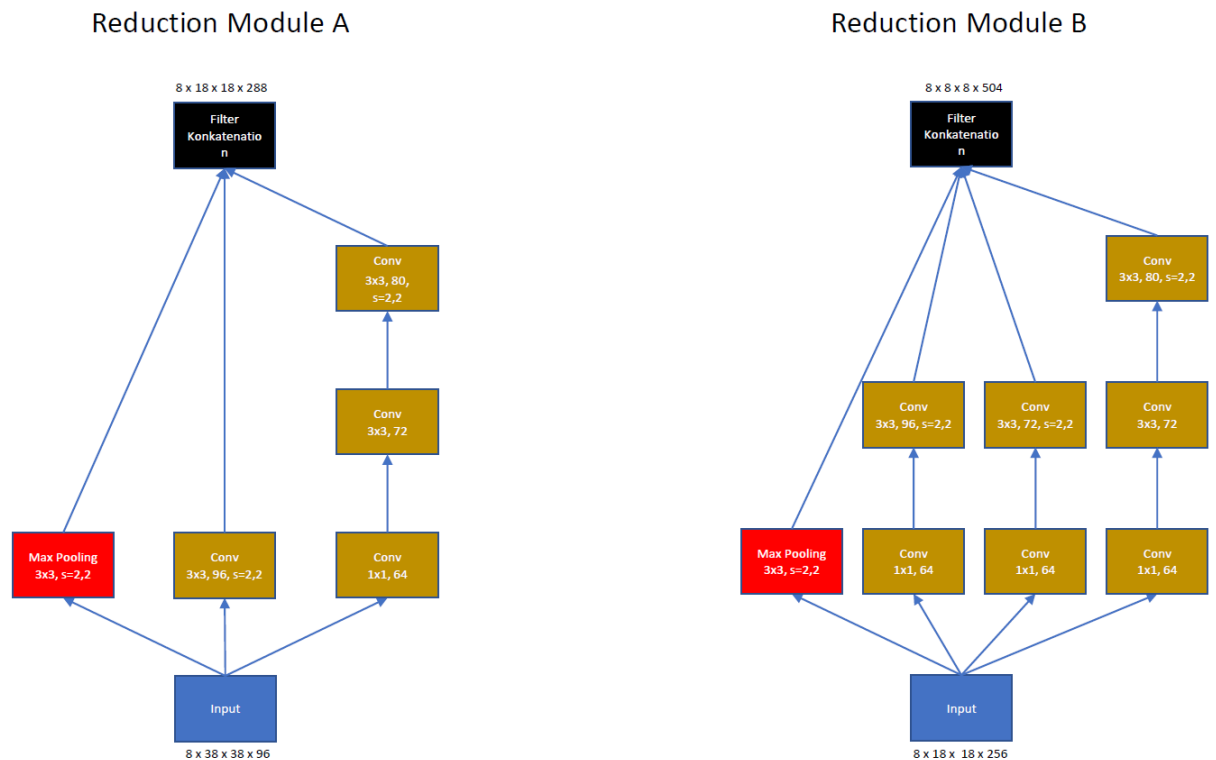


Abbildung 10: Übersicht der Inception V4-Reduktions-Module A und B.

Zur Reduzierung der Bilddimensionen Höhe und Breite nutzt Inception Reduction-Module (Abbildung 10). 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 bzgl. 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 Squeeze-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 11 dargestellt – aus vier zusätzlichen Elementen. Zunächst wird Global Average Pooling auf die Matrizen angewandt. Die Dimensionen der Ausgabe entsprechen in der Folge  $1 \times 1 \times AD$ , 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.



# Squeeze-and-Excitation Modifikation

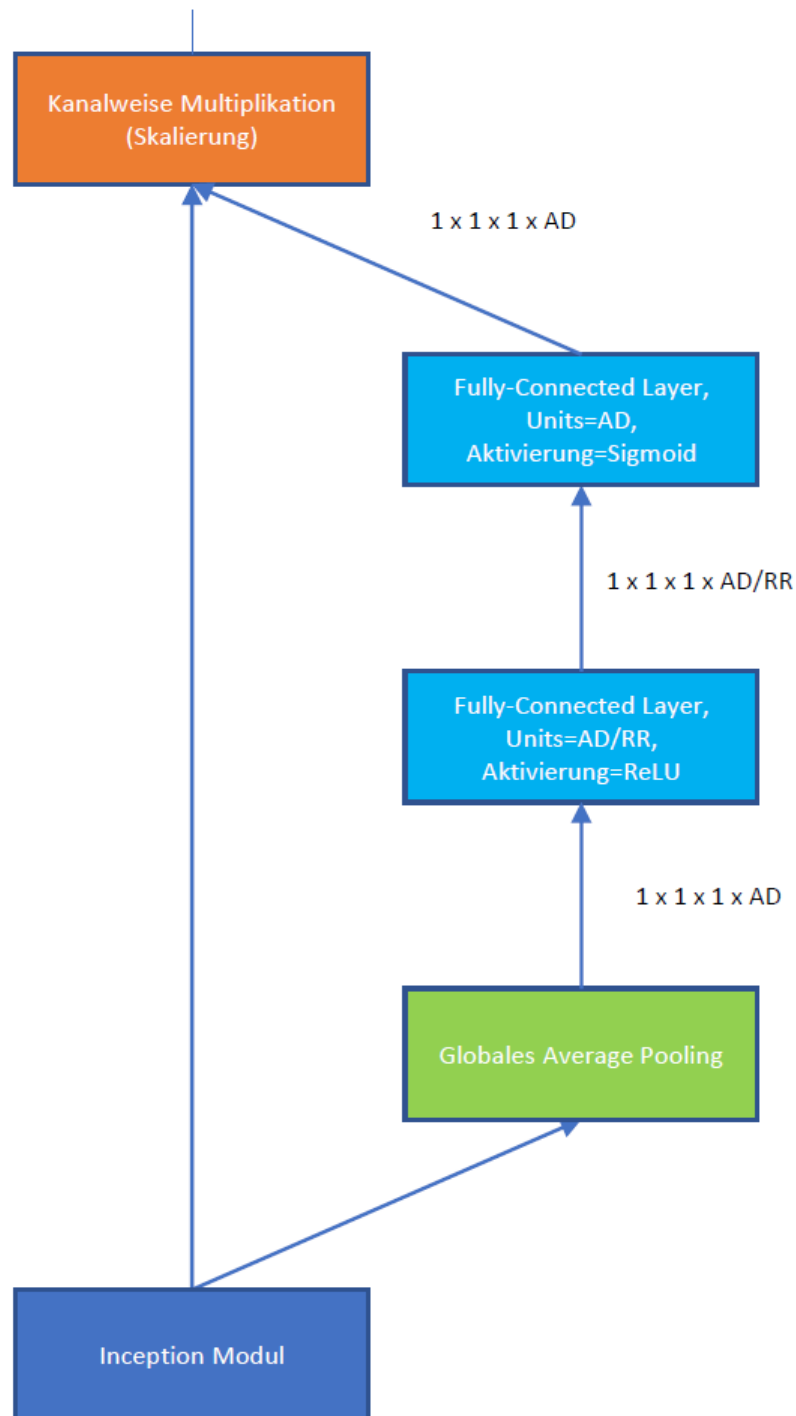


Abbildung 11: Der Squeeze-and-Excitation-Block.

## 5 Evaluation

### 5.1 Datengenerierung

Der Vorgang der Datengenerierung wurde nach dem in Abbildung 12 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.

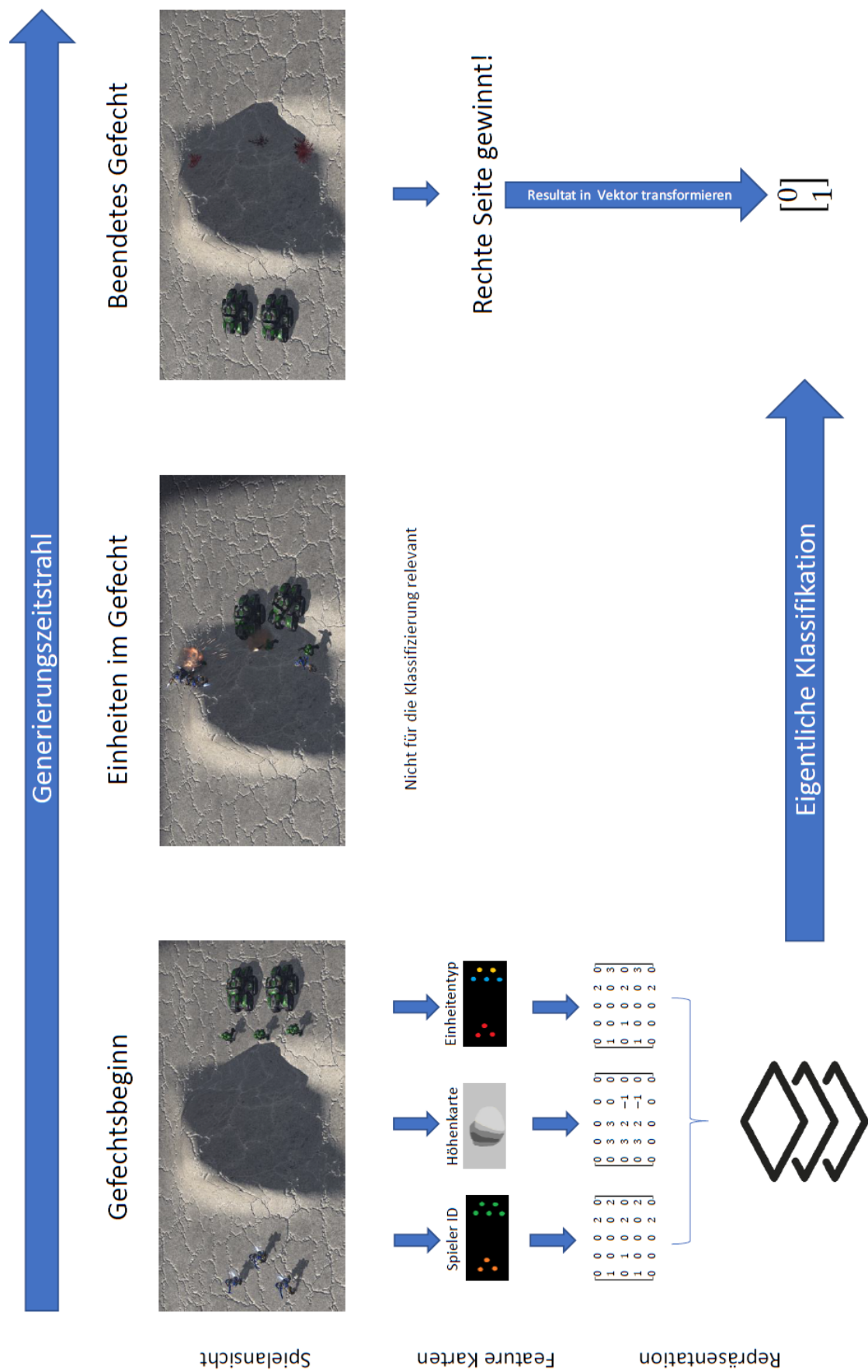
Die Armeen werden zufällig generiert, der Generierungsprozess läuft wie folgt ab: Zunächst wird für beide Armeen eine Rasse gewürfelt. Die Rasse bestimmt welche Einheiten die Armee erzeugen darf. Im Folgenden wird für jede Einheit durch einen Zufallszahlen-Generator die Anzahl der maximal zu erzeugenden 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 beider Armeen kleiner ist als ein definierter Grenzwert (in diesem Fall 5).

Abbildung 13 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.



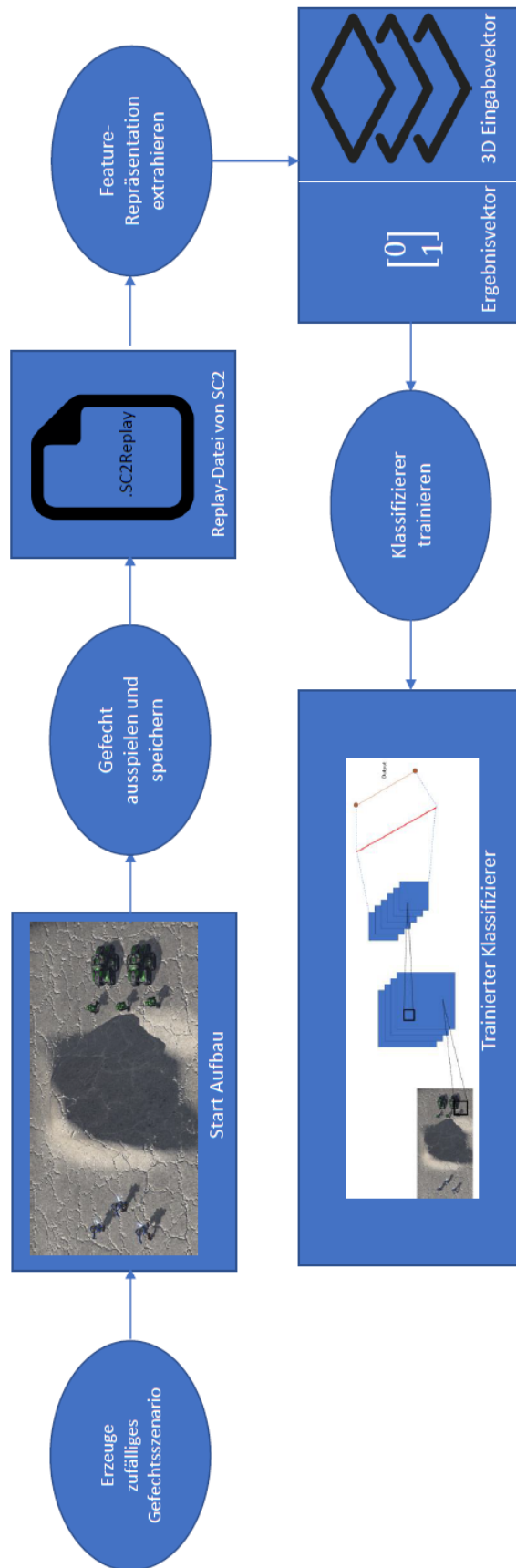


Abbildung 13: Übersicht des Generierungsprozesses der Trainingsdaten

## **5.2 Resultate**

## **5.3 Auswertung**

## **6 Konklusion**

### **6.1 Fazit und Ausblick**

## Literatur

- [1] D. Churchill und M. Buro. "Portfolio greedy search and simulation for large-scale combat in starcraft". In: *2013 IEEE Conference on Computational Intelligence in Games (CIG)*. Aug. 2013, S. 1–8.
- [2] David Churchill, Abdallah Saffidine und Michael Buro. "Fast Heuristic Search for RTS Game Combat Scenarios". In: 2012.
- [3] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [4] Kaiming He u. a. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Juni 2016).
- [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] R. A. Kilmer. "Applications of Artificial Neural Networks to combat simulations". In: *Mathematical and Computer Modelling* 23.1 (1996), S. 91–99.
- [9] Keiron O'Shea und Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *CoRR abs/1511.08458* (2015).
- [10] Glen Robertson und Ian Watson. "A Review of Real-Time Strategy Game AI". In: *AI magazine* (2014).
- [11] Mikayel Samvelyan u. a. *The StarCraft Multi-Agent Challenge*. 2019. arXiv: 1902.04043 [cs.LG].
- [12] Antonio A. Sánchez-Ruiz-Granados. "Predicting the Outcome of Small Battles in StarCraft". In: *ICCBR*. 2015.
- [13] Jost Tobias Springenberg u. a. "Striving for Simplicity: The All Convolutional Net". In: *CoRR abs/1412.6806* (2014).
- [14] Marius Stanescu, Nicolas Barriga und Michael Buro. "Using Lanchester Attrition Laws for Combat Prediction in StarCraft". In: 2015.
- [15] Marius Stanescu u. a. "Predicting Army Combat Outcomes in StarCraft". In: 2013.
- [16] 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.



- [17] Christian Szegedy u. a. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014).
- [18] Oriol Vinyals u. a. "StarCraft II: A New Challenge for Reinforcement Learning". In: *CoRR* abs/1708.04782 (2017).
- [19] 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).
- [20] S. Wender und I. 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)*. Sep. 2012, S. 402–408.
- [21] Jianxin Wu. "Introduction to Convolutional Neural Networks". In: *National Key Lab for Novel Software Technology Nanjing University, China*. 2017, S. 5.