h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbmn
FACHBEREICH MATHEMATIK
UND NATURWISSENSCHAFTEN

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt

Fachbereiche Mathematik und
Naturwissenschaften & Informatik

Automatisierte Konstruktion von künstlichen neuronalen Netzen mithilfe von bestärkendem Lernen

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M. Sc.)
im Studiengang Data Science

vorgelegt von

Kevin Rojczyk

Referent	:	Prof. Dr. Horst Zisgen
Korreferent	:	Prof. Dr. Gunter Grieser

Ausgabedatum	:	1. April 2019
Abgabedatum	:	15. September 2019

Kevin Rojczyk: *Automatisierte Konstruktion von künstlichen neuronalen Netzen mithilfe von bestärkendem Lernen* , © 15. September 2019

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 15. September 2019

Kevin Rojczyk

ABSTRACT

Artificial neural networks (ANN) have many parameters and many degrees of freedom in architecture, making them a powerful method of machine learning, especially in classification. The development of a suitable ANN for an unknown dataset is time-consuming and difficult due to many construction possibilities and required computing capacities. To mitigate this problem, the software *Network Architecture Construction Environment* (NACEnv) was designed as a part of this master thesis using the method of machine learning. It is designed to learn a strategy for efficiently constructing and training a suitable ANN for an unknown dataset. Despite the limitation of the task to *Convolutional Neural Networks* it was not possible to learn a universal strategy. However, with NACEnv it was possible to learn a very simple strategy in a simplified scenario. With NACEnv it was possible to construct a suitable ANN for a dataset, which was previously also used in the training of NACEnv.

KURZFASSUNG

Künstliche neuronale Netze (KNN) besitzen viele Parameter und Freiheitsgrade in der Architektur, was sie zu einer mächtigen Methode des maschinellen Lernens, insbesondere bei der Klassifizierung, macht. Die Entwicklung eines passenden KNNs ist für einen unbekannten Datensatz, aufgrund der vielen Konstruktionsmöglichkeiten und benötigten Rechenkapazitäten, zeitaufwendig und schwierig. Um dieses Problem zu entschärfen, wurde im Rahmen dieser Masterarbeit mithilfe der Methode bestärkendes Lernen die Software *Network Architecture Construction Environment* (NACEnv) entworfen. Diese soll eine Strategie erlernen, um effizient ein passendes KNN für einen unbekannten Datensatz zu konstruieren und zu trainieren. Trotz der Beschränkung der Aufgabenstellung auf *Convolutional Neural Networks* ist es nicht gelungen eine allgemeingültige Strategie zu erlernen. Allerdings gelang es mit NACEnv eine sehr simple Strategie in einem vereinfachten Szenario zu erlernen. Dabei konnte mit NACEnv ein passendes KNN für einen Datensatz konstruiert werden, der zuvor ebenfalls beim Training von NACEnv verwendet wurde.

INHALTSVERZEICHNIS

I MASTERTHESIS
1 EINLEITUNG 2
1.1 Motivation 2
1.2 Ziel der Arbeit 2
1.3 AutoDL Challenge 2019 3
1.4 Gliederung 4
2 GRUNDLAGEN 5
2.1 Künstliche Neuronale Netze und Deep Learning 5
2.1.1 Funktionsweise 6
2.1.2 Training 8
2.1.3 Schwierigkeiten beim Training 11
2.1.4 Convolutional Neural Networks 12
2.2 Bestärkendes Lernen 13
2.2.1 Exploitation und Exploration 14
2.2.2 Off/On-Policy-Verfahren 15
2.2.3 Proximal Policy Optimization 18
3 VERWANDTE ARBEITEN 21
3.1 Network Architecture Search (NAS) 23
3.1.1 NASNet 24
3.1.2 PNAS 25
3.2 MetaQNN 26
3.3 BlockQNN 27
4 EIGENER ANSATZ 30
4.1 Abgrenzung zu anderen Publikationen 32
4.2 Entwicklungsumgebung und Voraussetzungen 33
4.3 Version 1 35
4.3.1 Warum Chainer? 35
4.3.2 Aufbau 36
4.3.3 Evaluierung 38
4.3.4 Schlussfolgerung 39
4.4 Version 2 41
4.4.1 Aufbau 42
4.4.2 Umsetzung der zustandsabhängigen Aktionswahlbeschränkung 46
4.4.3 Vorbereitung für die AutoDL Challenge 47
4.4.4 Wechsel von Chainer zu TensorFlow 49
4.4.5 Evaluierung 51
4.4.6 Schlussfolgerung 54
4.5 Version 3 - NACEnv 54
4.5.1 Aufbau 55
4.5.2 Evaluierung 61

5 MÖGLICHKEITEN ZUR FORTFÜHRUNG 70
5.1 Erste Idee: Pretrain mit SMBO 70
5.2 Zweite Idee: Aussagekräftigere Observation für das Training der KNNs 71
5.3 Dritte Idee: Schichten mithilfe von Konkatenation vergrößern . 71
5.4 Vierte Idee: Numerische Aktionen 72
6 ZUSAMMENFASSUNG 73
6.1 Schwierigkeiten bei Reinforcement Learning 73
6.2 Fazit 75

II APPENDIX
A DATENSÄTZE 78
B BEILIEGENDE DATEIEN 79
C PLOTS VON EVALUATION 9 & EVALUATION 9B 80

LITERATUR 82

ABBILDUNGSVERZEICHNIS

Abbildung 1.3.1 Bearbeitungszeiten der AutoDL Challenge 2019 3
Abbildung 2.1.1 Neuron mit drei Eingangssignalen 6
Abbildung 2.1.2 Die drei am häufigsten genutzten Aktivierungsfunktionen 7
Abbildung 2.1.3 KNN mit einem versteckten FC-Layer 7
Abbildung 2.1.4 Topologie des CNNs LeNet-5 13
Abbildung 3.1.1 Aufbau von NAS 23
Abbildung 3.2.1 Zustandsraums von MetaQNN 27
Abbildung 3.3.1 Vergleich mehrerer alternativen Publikationen 28
Abbildung 4.3.1 Genauigkeit und Komplexitäts-Kosten von Evaluation 1d 40
Abbildung 4.3.2 Komplexitäts-Kosten von Evaluation 1d 41
Abbildung 4.4.1 *Action-State-Machine* von Version 2 43
Abbildung 4.4.2 Umgebung und Observation in Version 2 48
Abbildung 4.4.3 Beispiel für die Punkteberechnung in AutoDL 49
Abbildung 4.5.1 Normalverteilung der Zeitdauer 55
Abbildung 4.5.2 *Action-State-Machine* von NAC 59
Abbildung 4.5.3 Beispiel für Schichten-Injektion 60
Abbildung 4.5.4 Umgebung und Observation in Version 3 62
Abbildung 4.5.5 Aktions-Wahrscheinlichkeiten in Evaluation 5 63
Abbildung 4.5.6 Plots vom Trainingsverlauf von Evaluation 5 64
Abbildung 4.5.7 Trainingsfortschritte bei Evaluation 6 66
Abbildung 4.5.8 Discount-Faktoren in Evaluation 7 67
Abbildung 5.3.1 Beispielbild für das Konkatenieren zweier Schichten . . 72
Abbildung 6.1.1 Probleme bei ML-Verfahren 74

TABELLENVERZEICHNIS

Tabelle 4.0.1 Ein Überblick über die drei implementierten Versionen. 30
Tabelle 4.3.1 Observation in Version 1 38
Tabelle 4.3.2 Levels mit ihren Datensätzen und Kriterien 38
Tabelle 4.4.1 Observation in Version 2 45
Tabelle 4.4.2 PPOs Hyperparameter bei der Parametersuche 53
Tabelle 4.5.1 Observation in Version 3 57

ABKÜRZUNGSVERZEICHNIS

KNN Künstliche neuronale Netze

RL Reinforcement Learning

ML Maschinelles Lernen

PPO Proximal Policy Optimization

Colab Google Colaboratory

A2C Advantage Actor Critic

ASM Action-State-Machine

Observation Repräsentation des Zustandes von der Umgebung

FC-Layer Fully-Connected Layer

ReLU Rectified Linear Unit

Tanh Tangens Hyperbolicus

MSE Mean Squared Error

SGD Stochastic Gradient Descent

CNN Convolutional Neural Network

Conv-Layer Convolution-Layer

FLOPS Floating Point Operations Per Second

GPU Graphics Processing Unit

SMBO Sequential Model-Based Optimization

TPU Tensor Processing Unit

NACEnv Network Architecture Construction Environment

Teil I

MASTERTHESIS

1 EINLEITUNG

Dieses Kapitel dient dazu die Motivation und die Ziele dieser Masterarbeit zu beleuchten.

1.1 MOTIVATION

Im Bereich des maschinellen Lernens (ML) erfreuen sich in den letzten Jahren künstliche neuronale Netze (KNN) großem Interesse. Durch technologische Fortschritte in der Hardware sowie in der Software, können KNNs in immer mehr Disziplinen, andere ML-Verfahren übertrumpfen. Sie gelten als Stand der Technik bei verschiedenen Aufgabenstellungen in den Bereichen Sprach- und Bilderkennung. KNNs können konvexe und nicht-konvexe Probleme lösen und dabei an die Schwierigkeit der Problemstellung angepasst werden, indem die Anzahl der trainierbaren Parameter (Neuronen) und der Aufbau des Netzes geändert wird.

Die vielen Freiheitsgrade der Konfiguration eines KNNs stellen zugleich ein Problem dar, wenn ein neues Netz trainiert werden soll. Es gibt so viele Hyperparameter und Möglichkeiten ein KNN zu konstruieren, dass es normalerweise nicht möglich ist alle Varianten auszuprobieren. Es gibt zwar Vorgehensweisen und Architekturen die sich bewährt haben, allerdings setzen auch diese voraus, dass die KNN-Architekturen und die dazugehörigen Hyperparameter durch Experimente ausprobiert und iterativ an eine spezifische Aufgabenstellung angepasst werden.

Wenn genügend Ressourcen vorhanden sind, wird oft eine automatisierte Hyperparametersuche durchgeführt. Nach einer passenden KNN-Architektur wird aber selten automatisiert gesucht, da es dafür zu viele Konstruktionsmöglichkeiten gibt, als dass dafür die Ressourcen vorhanden sind. So bleibt die Konstruktion einer guten KNN-Architektur oft dem Entwickler überlassen, der mit „gutem Gespür“ und Fachwissen versucht manuell eine passende Architektur zu konstruieren.

1.2 ZIEL DER ARBEIT

In dieser Masterarbeit soll eine Software auf Basis von maschinellem Lernen entwickelt werden, die automatisch eine passende KNN-Architektur für einen beliebigen Datensatz findet. Dazu soll das ML-Verfahren *Reinforcement Learning* (RL) eingesetzt werden, mit dem ein Agent trainiert wird, der eine KNN-Architektur konstruiert und trainiert. Dafür steht dem Agenten kein Vorwissen zur Verfügung. Der Agent muss durch Ausprobieren Strategien und Regeln erlernen, wie eine KNN für einen bestimmten Datensatz aufgebaut sein muss.

Der Agent soll im Resultat möglichst schnell auf iterativerweise, ein passendes KNN für einen gegebenen Datensatz bauen und trainieren.

Weil das Trainieren von KNNs sehr ressourcenaufwändig und das Themenspektrum sehr breit ist, wird der Agent im Rahmen dieser Masterarbeit auf einfacherere Datensätze für die Bildklassifikation fokussiert.

Eine ähnliches Ziel der Masterarbeit wird durch die *AutoDL Challenge 2019* verfolgt, siehe nächstes Unterkapitel 1.3. Eine optionale Teilnahme an der Challenge soll offen gehalten werden, um einen Vergleich der erarbeiteten Ergebnisse der anderen Teilnehmer zu ermöglichen.

1.3 AUTODL CHALLENGE 2019

Die am Anfang des Jahres 2019 angekündigte AutoDL Challlenge ist ein Wettbewerb, welches erstmals im Jahr 2019 ausgetragen wird. Der Aufbau und die Teilnahmebedingungen wurden zeitgleich mit dem Start des Wettbewerbs am 1. Mai veröffentlicht.

Das Ziel des Wettbewerbs ist es, eine Software zu programmieren, die mithilfe von ML-Verfahren automatisiert Klassifizierungen auf unbekannte Datensätze umsetzt. Dafür wurde der Wettbewerb in fünf unabhängige Disziplinen unterteilt, dessen Bearbeitungszeit sich teilweise überschneiden. Folgende Disziplinen gibt es: Bildklassifikation (AutoCV, AutoCV2), linguistische Datenverarbeitung (AutoNLP), Zeitreihenanalyse (AutoSeries) und eine Kombination aus den vorherigen Diziplinen (AutoDL).

Der Zeitverlauf des Wettbewerbs ist in der Abbildung 1.3.1 zu sehen.

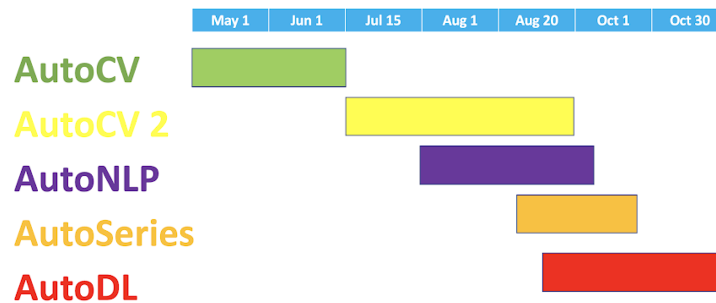


Abbildung 1.3.1: Bearbeitungszeiten der einzelnen Disziplinen der AutoDL Challenge 2019. Bild von der Webseite des Wettbewerbs[1]

Das Interessante am Aufbau des Wettbewerbs ist, dass ähnlich wie bei kaggle[2], der eigene Code direkt auf den Server der Organisatoren hochgeladen werden kann und dort der Code anhand von Datensätzen, die für die Wertung bestimmt aber den Teilnehmer unbekannt sind, evaluiert wird. Der Gewinner einer Disziplin bekommt 2000$ Preisgeld und die Möglichkeit eine Publikation für eine Konferenz einzureichen. Die Organisatoren des Wettbewerbs sind der Verein ChaLearn, der zuvor auch mehrmals die AutoML Challenge veranstaltet hat, Google und das chinesische Start-Up 4Paradigm.

1 https://autodl.chalearn.org
2 https://www.kaggle.com

1.4 GLIEDERUNG

Das nächste Kapitel 2 (Grundlagen) dient der Einführung in die Themenschwerpunkte „künstliche neuronale Netze“ und „bestärkendes Lernen“. Es dient zur Vorbereitung, um die darauf folgenden Kapitel verstehen zu können. In Kapitel 3 (Verwandte Arbeiten) wird der Forschungsstand anhand einer Auswahl von wissenschaftlichen Publikationen reflektiert, die im Zusammenhang mit den Zielen dieser Masterarbeit stehen. Das darauf folgende Kapitel 4 (Eigener Ansatz) widmet sich dem im Rahmen der Masterarbeit selbst entwickelten Ansatz zur Erreichung der Ziele. Das Kapitel 5 (Möglichkeiten zur Fortführung) enthält eigene Ideen zur Erweiterung des vorgestellten Ansatzes, die im Rahmen der Masterarbeit nicht mehr umgesetzt werden konnten. Schlussendlich folgt mit Kapitel 6 eine Zusammenfassung der Schwierigkeiten, die während der Bearbeitung der Masterarbeit aufgetreten sind sowie das Fazit.

2 GRUNDLAGEN

Um eine bessere Nachvollziehbarkeit dieser Arbeit zu ermöglichen, werden in diesem Kapitel die Forschungsschwerpunkte „künstliche neuronale Netze“ (KNN) und „bestärkendes Lernen“/*Reinforcement Learning* (RL) näher beschrieben. Da die genannten Forschungsschwerpunkte komplex und umfangreich sind, wird hier nur das Wissen vermittelt, welches zum Verständnis der Arbeit nötig ist. Es wird vom Leser angenommen, dass dieser grundlegendes Wissen im Bereich maschinelles Lernen besitzt und somit einfache Begrifflichkeiten in dem Kontext nicht erklärt werden müssen.

Dieses Kapitel basiert, wenn nicht anders angegeben, auf folgende Literatur:

FÜR KÜNSTLICHE NEURONALE NETZE: [21][8]

FÜR BESTÄRKENDES LERNEN: [34][30][7]

Die oben aufgelistete Literatur ist auch eine Empfehlung meinerseits für den interessierten Leser, falls dieser sich mit den zwei Schwerpunktthemen breiter oder tiefgehender Auseinandersetzen möchte, als das mit diesem Grundlagen-Kapitel möglich ist.

Es ist dabei zu beachten, dass insbesondere die Themenfelder künstliche neuronale Netze und bestärkendes Lernen aktuell durch hohe Forschungsaktivität geprägt sind und somit der Forschungsstand sich relativ schnell ändert.

Englische Fachbegriffe deren übersetzte Form nicht weitläufig genutzt werden, wurden im Englischem belassen. Damit soll unnötige Verwirrung, insbesondere bei den Lesern die mit den Schwerpunktthemen schon vertraut sind, vermieden werden.

2.1 KÜNSTLICHE NEURONALE NETZE UND DEEP LEARNING

Beginnend in den 1950er Jahren wurde an Datenverarbeitung und maschinelles Lernen auf Basis der Arbeitsweise des Gehirns experimentiert. Das Gehirn ist das lernfähigste System das bekannt ist und damit von besonderem Interesse. Ein Gehirn besteht aus Millionen von Neuronen die miteinander interagieren und bei Lebewesen für die „Datenverarbeitung“ zuständig ist. Die Funktionsweise eines Neurons kann folgendermaßen simplifiziert beschrieben werden: Das Neuron ist aktiv und gibt ein Signal aus, wenn die akkumulierte Stärke der eingehenden Signale von verbundenen Neuronen einen gewissen Schwellwert überschritten hat.

2.1.1 *Funktionsweise*

Künstliche Neuronale Netze, im Folgenden nur KNNs genannt, bestehen aus künstlichen Neuronen die die Funktionsweise echter Neuronen teilweise adaptieren. Weil ein künstliches Neuron nur teilweise einem echten Neuron ähnelt, werden sie in der Fachliteratur manchmal als *Units* bezeichnet. Im restlichen Teil der Arbeit wird der Begriff Neuron verwendet, womit ein künstliches Neuron gemeint ist. Ein Neuron hat typischerweise mehrere andere Neuronen als Eingangssignal. Die jeweiligen Eingangssignale werden gewichtet und zusammen mit dem Neuron eigenem Bias aufsummiert. Das Neuron gibt die Summe als Signal aus, nachdem es mit einer *Aktivierungsfunktion* verrechnet wurde. Die Abbildung 2.1.1 zeigt exemplarisch ein Neuron.

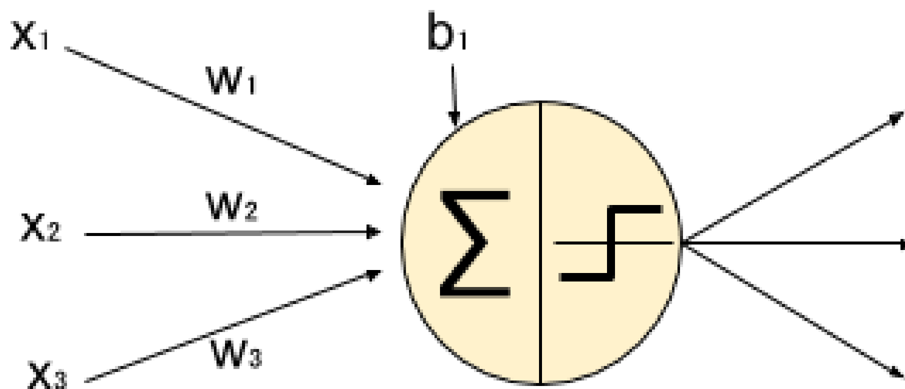


Abbildung 2.1.1: Ein Neuron mit drei Eingangssignalen x_1, x_2, x_3, deren *Gewichte* $w_1, w_2, w3$ und seinem *Bias* b_1. Die Summe wird durch eine *Aktivierungsfunktion* ausgegeben.

Die Aktivierungsfunktion ist maßgeblich für die Fähigkeit eines KNNs wichtig. Üblicherweise ist die Aktivierungsfunktion eine nicht-lineare Abbildung. Nur eine nicht-lineare Aktivierungsfunktion ermöglicht einem KNN auch nicht-lineare Probleme zu lösen. Im Plot 2.1.2 sind drei weitverbreitete Aktivierungsfunktionen dargestellt und geben einen Eindruck über ihre Unterschiede und Gemeinsamkeiten.

Die Neuronen sind bei KNNs normalerweise in Schichten aufgeteilt. In der Regel sind die Ausgangssignale einer Schicht die Eingangssignale der nächsten Schicht. (Es existieren auch komplexere KNNs, für die diese Regel nicht gilt, diese sind aber aus Zeitgründen kein Gegenstand der Untersuchung dieser Arbeit.) Dieser einfacher und am häufigsten verwendeter Aufbau ist als *Feed-Forward Network* bekannt, siehe Abbildung 2.1.3. An den Pfeilen ist zu erkennen, das alle Neuronen der vorherigen Schicht mit allen Neuronen der nächsten Schicht verbunden sind. Solche Schichten werden als *Fully-Connected Layer* (FC-Layer) oder *Dense Layer* bezeichnet. Eine Schicht,

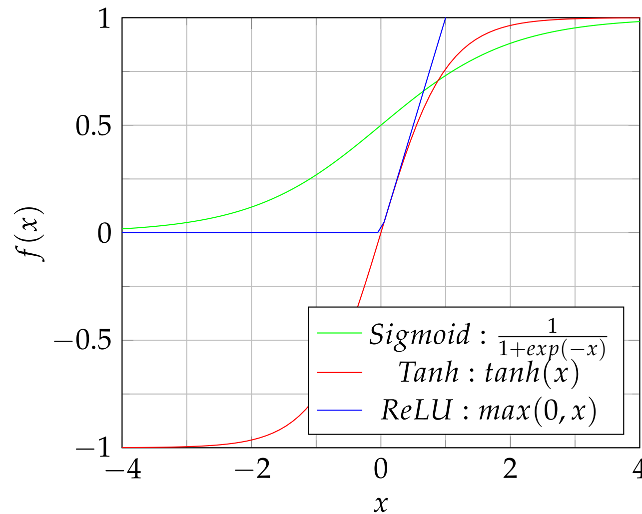


Abbildung 2.1.2: Die drei am häufigsten genutzten Aktivierungsfunktionen für KNNs. Der Abbildungsbereich für Aktivierungsfunktionen ist:
Sigmoid: $[0, 1]$, Tanh: $[-1, 1]$, ReLU: $[0, \infty)$

die weder *Input*/Eingabe noch *Output*/Ausgabe-Schicht ist, wird versteckte Schicht (*hidden Layer*) genannt.

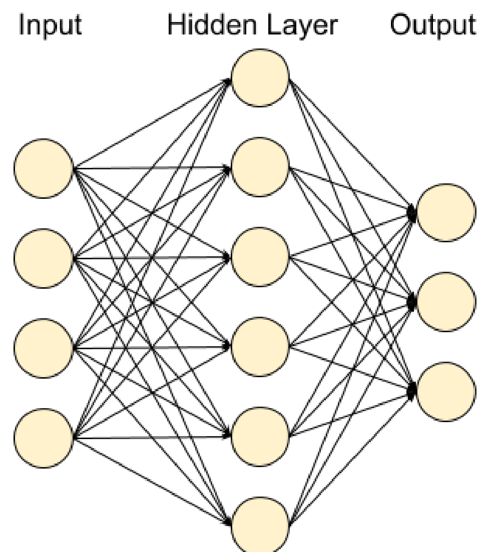


Abbildung 2.1.3: Ein KNN mit einem versteckten FC-Layer, vier Input-Neuronen und drei Output-Neuronen. Der Informationsfluss läuft von links nach rechts.

Beim Feed-Forward Network stellt die erste Schicht die Datenpunkte eines Datenelements aus dem Trainingsdatensatzes dar. Um ein schnelleres Konvergieren des KNNs zu ermöglichen, werden üblicherweise die Datenpunte auf $[0, 1]$ oder $[-1, 1]$, bedingt durch die Wahl der Aktivierungsfunktion, normiert. Die Signale der Neuronen müssen Schicht für Schicht berechnet werden, wobei die letzte Schicht, der *Output-Layer*, das Ergebnis des KNNs darstellt. Der Output-Layer muss für die Aufgabenstellung passend konfigu-

riert sein. Beispiel: Bei einer Klassifikation mit zehn unterschiedlichen Klassen besitzt der Output-Layer zehn Neuronen, im Gegensatz dazu, benötigt der Output-Layer bei einer Regression eines numerischen Wertes lediglich ein Neuron. Um die Interpretierbarkeit der Ausgabe eines Output-Layers bei einer Klassifikation zu erleichtern, wird für den Output-Layer oft ein *Softmax* als Aktivierungsfunktion verwendet.

Definiton 1 (Softmax-Aktivierungsfunktion) *Sei $i \in [1, K]$ je einer Klassen aus K vielen Klassen zugeordnet und $\mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K$ die Output-Neuronen für die jeweiligen Klassen, dann gibt die Softmax-Funktion*

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

die Wahrscheinlichkeit für die Klassenzugehörigkeit i an. Softmax stellt somit eine Wahrscheinlichkeitsverteilung über K verschiedene Ereignisse dar.

2.1.2 *Training*

Das Anpassen der Gewichte eines KNNs an Daten, im Kontext auch Lernen oder Trainieren genannt, ist im Vergleich zur Auswertung eines KNNs komplexer und deutlich rechenintensiver. Im Folgenden wird dazu das *supervised* Lernverfahren *Backpropagation* [25] vorgestellt.

Seit den 1980er Jahren ist diese Methode die bevorzugte Art, um KNNs zu trainieren. Die Backpropagation ermöglichte erstmals effizient KNNs mit mehreren versteckten Schichten zu trainieren. Dies ist insbesondere wichtig, weil die Mächtigkeit eines KNNs mit der Anzahl an Schichten und Neuronen potenziell steigt. KNNs mit besonders vielen Schichten werden auch *Deep Neural Networks* genannt. Sie sind der Grund warum KNNs in vielen Disziplinen des maschinellen Lernens, wie zum Beispiel Bilderkennung [17][23], führend sind. Erst durch die Nutzung von Backpropagation und der Verfügbarkeit der nötigen Hardwareleistung, gehören KNNs zu den wichtigsten Methoden des maschinellen Lernens.

Bei Backpropagation wird der berechnete Fehler der Ausgabe, also die Abweichung vom Zielwert, auch *Loss* genannt. Anhand der partiellen Ableitung wird der Loss auf die Gewichte und den Bias der Neuronen zurück *propagiert*. Die Idee ist, die Gewichte und die Bias von den Neuronen „ein Stück weit" in die entgegengesetzte Richtung des Gradienten abzuändern, so dass das Ergebnis vom KNN sich dem Zielwert annähert. Wie stark die Gewichte und den Bias abgeändert werden, hängt einerseits von einer zu definierenden Lernrate η und anderseits von der partiellen Ableitung $\frac{\partial E}{\partial w_{ij}}$ ab, die eine Aussage trifft wie stark und in welche Richtung (positiv oder negativ) das Gewicht den Fehler beeinflusst.

Definiton 2 (Forward- und Backpropagation) *Sei z_k^l die Summe der mit w_{kj}^l gewichteten Eingangssignale vom k-ten Neuron in Schicht $l \in [1, L]$ und a_k^l der Wert der Summe nach Anwendung einer beliebigen Aktivierungsfunktion σ. Dann ist die Forwardpropagation für die Schichten folgendermaßen definiert:*

$$z_k^l = \sum_j w_{kj}^l a_j^{l-1} + b_k^l \textit{ mit } a_k^l = \sigma(z_k^l)$$

Dabei ist es nötig bei der ersten Schicht mit $l = 1$ zu beginnen und dann eine Schicht nach der anderen auszurechnen. Die Eingabe des KNNs wird durch den Vektor a_j^0 repräsentiert. Die Anzahl der Schichten vom KNN ist durch L gegeben.

Für die Backpropagation benutzen wir die selbe Notation, wie die von der Forwardpropagation. Zusätzlich sei der Loss vom KNN mit E notiert. Dann ist die partielle Ableitung des Neuron j nach dem Loss E:

$$\delta_k^l = \frac{\partial E}{\partial z_k^l} \overset{l=L}{=} \frac{\partial E}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_k^L} = \frac{\partial E}{\partial a_k^L} \sigma' \left(z_k^L \right) \tag{2.1}$$

oder

$$\delta_k^l = \frac{\partial E}{\partial z_k^l} \overset{l \neq L}{=} \left(\sum_m \delta_m^{l+1} w_{mk}^{l+1} \right) \sigma' \left(z_k^l \right) = \left(w^{l+1} \right)_k^T \delta^{l+1} \sigma' \left(z_k^l \right) \tag{2.2}$$

Dabei ist 2.1 die Ableitung für den Output-Layer und 2.2 für alle andere Schichten (hidden Layers).

Die Anpassung des Bias vom k-ten Neuron ist somit:

$$(b_k^l)^{\text{neu}} := (b_k^l)^{\text{alt}} + \Delta b_k \textit{ mit } \Delta b_k = -\eta \frac{\partial E}{\partial b_k} = -\eta \delta_k^l$$

und die Anpassung vom Gewicht für das m-te Eingangssignal des k-ten Neuron:

$$(w_{mk}^l)^{\text{neu}} := (w_{mk}^l)^{\text{alt}} + \Delta w_{mk} \textit{ mit } \Delta w_{mk} = -\eta \frac{\partial E}{\partial w_{mk}^l} = -\eta \delta_k^l a_m^{l-1}$$

Dabei ist η die Lernrate, die meistens in einem Bereich von $[10^{-5}, 1]$ liegt.

Quelle: [21, K. 2]

Die Berechnungen beim Backpropagation lassen sich dabei mit Matrix- und Vektorrechnungen beschleunigen. So ist es möglich, die Anpassung der Gewichte für alle Neuronen in einem Layer l gleichzeitig zu berechnen: $\left(w^{l+1} \right)^T \delta^{l+1} \sigma' \left(z^l \right)$. Insbesondere moderne Grafikkarten (GPU) können diese

Art von Berechnungen effizient und schnell ausführen und sind deshalb in der *Deep-Learning*-Community beliebt.

Es gibt unterschiedliche Möglichkeiten den Loss zu berechnen. Die Methoden werden auch Fehlerfunktionen, Loss-Funktionen oder Kosten-Funktionen genannt. In Definition 3 sind zwei wichtige Loss-Funktionen dargestellt. Zum einen der mittlere quadratischer Fehler (*MSE*), der oft wegen seiner Einfachheit verwendet wird und zum anderen bei der Mehrklassen-Klassifizierung die Kreuzentropie (*Cross-Entropy*).

Definiton 3 (Loss-Funktionen) *Sei $t \in \mathbb{R}$ der Zielwert und $o \in \mathbb{R}$ die beobachtete Ausgabe, dann sind folgendes Loss-Funktionen für $n \in \mathbb{N}$ verschiedene Zielwerte:*

$$MSE = \frac{1}{2}\sum_{i=1}^{n}(t_i - o_i)^2 \qquad \text{(Mittlerer quadratischer Fehler)}$$

$$CE = -\sum_{i=1}^{n}(t_i log(o_i)) \qquad \text{(Kreuzentropie)}$$

Beim Backpropagation-Algorithmus kann der Loss nur von einem Datenelement (*Stochastic Gradient Descent* (SGD)) oder der Durchschnitt vom ganzen Datensatz (*Batch Gradient Descent*) verwendet werden. Das Optimierungsverfahren SGD setzt dabei voraus, dass die Reihenfolge der Datenelemente zufällig ist (*stochastic*). SGD hat gegenüber vom *Batch Gradient Descent* den Vorteil, dass es wahrscheinlicher ein besseres lokales Optimum beziehungsweise ein globales Optimum findet. Meistens wird ein Kompromiss eingesetzt, in dem SGD mit mehreren Datenelemente, den sogenannten Mini-Batches, verwendet wird. Eine Iteration von SGD ist damit schneller berechnet, insbesondere weil durch die Mini-Batches Vektorrecheneinheiten benutzt werden können [21, K. 2].

Im Kontext vom Backpropagation-Algorithmus und SGD ist es wichtig, noch folgendes zu erwähnen:

INITIALISIERUNG: Die Gewichte der Neuronen müssen am Anfang zufällig initialisiert werden. Algorithmen wie Xavier ([6]) haben sich dabei bewährt. Durch ihre Art der zufälligen Initialisierung können KNNs besser konvergieren.

MINI-BATCH: Für gewöhnlich enthält ein Mini-Batch zwischen 32 und 512 Datenelemente. Mini-Batches werden oft einfach nur Batches genannt.

ITERATION: In jeder Iteration wird der Loss eines Mini-Batches berechnet und anhand dessen per Backpropagation das KNN angepasst.

EPOCHE: Wurden alle Mini-Batches angewendet, dann ist eine Epoche abgeschlossen. Normalerweise wird ein KNN anhand mehrerer Epochen trainiert. Nach jeder Epoche werden die Datensätze erneut zufällig auf die Mini-Batches verteilt. Dies ermöglicht dem Lernverfahren schneller und besser zu konvergieren.

Weil SGD in einer Iteration immer nur ein Teil des Datensatzes berücksichtigt, kann es nur „schrittweise“ die Gewichte mit dem Backpropagation-Algorithmus anpassen. Die Schrittweite wird dabei durch die zuvor eingeführte Lernrate η vom Backpropagation beeinflusst. Bei SGD ist diese Lernrate statisch. Ausgefeiltere Optimierungsverfahren, die SGD adaptieren, verändern aber die Lernrate zum Beispiel anhand der Anzahl der Neuronen in der Schicht (z.B. AdaGrad, Adam [14]) oder anhand der Größe der Veränderung von der letzten Iteration (z.B. Momentum, Adam). Das simplere SGD wird heutzutage in der Praxis selten verwendet, weil dieser langsamer und oft auch schlechter konvergiert als die genannten Alternativen.

2.1.3 *Schwierigkeiten beim Training*

Der Nachteil bei KNNs ist, dass diese häufig schwerer zu trainieren sind als andere maschinelle Lernverfahren, wie zum Beispiel die *logistische Regression* oder die *Support Vector Machines*. Die Mächtigkeit von KNN-Modellen geht einher mit einer großen Anzahl an Modell-Parametern (Gewichte und Bias der Neuronen) und Hyperparameter wie Lernrate, Topologie des Netzes, Auswahl der Aktivierungsfunktion etc..

Die Modell-Parameter werden durch ein Optimierungsverfahren berechnet und deswegen steigt die benötigte Rechenleistung mit der Anzahl an Modell-Parametern. Die Hyperparameter werden durch eine manuelle Parametersuche gesetzt. Allerdings kann dies auch zum Teil automatisiert werden, zum Beispiel mit einem simplem *Random Search* oder mit einem effizienterem *Bayesian Optimization*, siehe dazu [24]. Aber auch diese Verfahren sind oft durch die benötigte Rechenleistung eingeschränkt.

Es existiert keine perfekte Anleitung inwiefern für die jeweilige Aufgabenstellung die Hyperparameter „richtig“ gesetzt werden. Lediglich Richtlinien für die Parametersuche und Beispiele für gute Kombinationen von Hyperparameter und Topologien von KNNs existieren, die aber größtenteils nur empirisch belegt sind. Die Hyperparameter für ein KNN muss für jeden neuen Datensatz erneut optimiert werden.

Neben diesen grundlegenden Problemen, die auch andere maschinelle Lernverfahren in geringerem Maße haben, besitzen KNNs noch spezielle Probleme die im Zusammenhang mit der Backpropagation stehen:

LOKALES OPTIMUM: Der Backpropagation-Algorithmus in Kombination mit SGD und einer niedrigeren Lernrate neigt dazu nur ein lokales Optimum zu finden. Alternativen zu SGD wie Adam, die die Lernrate dynamisch anpassen, sowie Mini-Batches können hierbei helfen.

VANISHING/EXPLODING GRADIENTS: Die partielle Ableitung eines Gewichts von einem Neuron ist abhängig vom Wert des Gewichts, der Tiefe der Schicht vom Neuron und der Aktivierungsfunktion. Dadurch können sehr niedrige oder auch sehr hohe Wert bei der partiellen Ableitung entstehen. Das KNN lernt dann sehr langsam oder bestehende Informationen im KNN werden überschrieben. Alternativ können andere

Aktivierungsfunktionen genutzt werden, wie zum Beispiel *Rectified Linear Unit* (ReLU), die resistenter sind gegen *Vanishing/Exploding Gradients*.

OVERFITTING: KNNs können bei einer großer Anzahl an Neuronen leicht *overfitten*. Dies kann verhindert werden, in dem der Trainingsprozess abgebrochen wird, wenn es keine weitere Verbesserung auf dem Validierungsdatensatz (*early stopping*) sich ergeben. Eine andere Möglichkeit ist, ein definierten Prozent-Anteil der Schichten beim Training vom KNN zufällig zu deaktivieren (*dropout*, siehe [33]).

2.1.4 *Convolutional Neural Networks*

Für Datesätze mit Bildern oder Videos werden sogenannte *Convolutional Neural Network* (CNN) verwendet, siehe [15]. Dabei wird die räumliche Nähe der Datenpunkte (Pixel) ausgenutzt, um eine bessere Abstraktion der Daten zu bekommen.

Ein CNN ist ein KNN mit *Convolutional-Layer* (Conv-Layer), meistens in Kombination mit *Max-Pooling*:

CONVOLUTIONAL-LAYER: Die Schicht besteht aus einem oder mehreren trainierbaren Faltungs-Operatoren, auch Filter genannt. Für die Filter muss die Anzahl der Filter, die Größe des Filter und die Schrittweite (*stride*) fest definiert werden. Der Filter ist eine Faltungsmatrix, die aus Gewichten besteht, die wie gewöhnliche Neuronen mit Backpropagation trainiert werden. Diese Filter werden anhand des *strides*, je nach Wahl, womöglich auch überlappend, auf alle Eingangssignale (das gesamte Bild) angewendet. Empirisch konnte beobachtet werden, dass in der Regel tiefere Schichten abstraktere Muster erkennen. Zum Beispiel: Im ersten Conv-Layer werden Kanten erkannt und im zweiten Conv-Layer geometrische Muster. Ein zusätzlicher Vorteil liegt darin, dass lediglich die Gewichte in den Filtern trainiert werden müssen, da diese Gewichte mit allen Eingangssignalen geteilt (*shared weights*) werden. Im Vergleich zu einem normalen FC-Layer ergeben sich dadurch weniger zu trainierende Gewichte.

MAX-POOLING: Beim *Pooling* werden die Ausgaben einer vorherigen Schicht, zum Beispiel mit einem 2x2 Raster, zusammengefasst. Ebenso gibt es bei Pooling eine zu definierende Schrittweite/*stride*. Klassischerweise wird das Maximum (Max-Pooling) im Raster als Ausgangssignal ausgegeben. Es sind auch andere Operatoren möglich wie zum Beispiel der Mittelwert vom Raster (Average-Pooling). Pooling dient dazu, die Anzahl an Ausgabesignalen einer Schicht zu verringern um damit die Anzahl an zu trainierenden Gewichten im KNN zu reduzieren.

Ein CNN besteht für gewöhnlich in den ersten Schichten aus mehreren Conv-Layer, mit und ohne Pooling, und endet mit einem oder mehreren

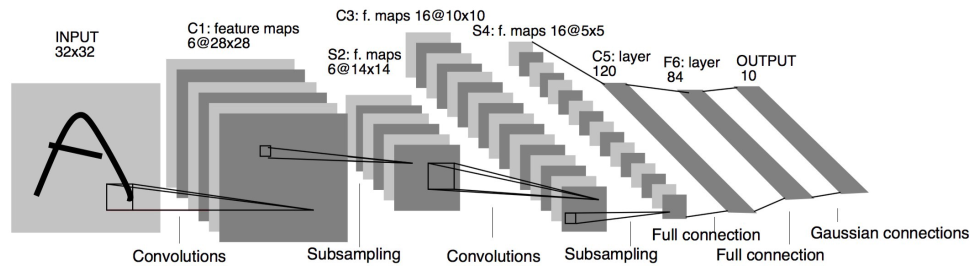


Abbildung 2.1.4: Topologie des CNNs LeNet-5. Die Grafik ist aus dem dazugehörigem Paper [16]. Die *feature maps* sind die Filter und das *subsampling* ist die Pooling-Operation. Das Netz besteht aus zwei Conv-Layern mit Pooling und zwei darauffolgende FC-Layer.

normalen FC-Layern. Die Abbildung 2.1.4 zeigt exemplarisch die Topologie eines CNNs.

CNNs sind maßgeblich verantwortlich, dass KNNs immer wieder neue Rekorde in der Bilderkennung aufstellen [26].

2.2 BESTÄRKENDES LERNEN

Bestärkendes Lernen/*Reinforcement Learning* (RL) ist ein *unsupervised* Machine-Learning-Verfahren, dass seine Lerndaten anhand von Interaktionen mit der „Umgebung" generiert. Die Interaktionen werden „Aktionen" genannt und die agierende Software „Agent". Eine Umgebung kann rein softwaretechnisch sein, wie zum Beispiel ein Computerspiel, oder real, wie zum Beispiel ein Roboter der sich im Raum bewegt. Zum Lernen braucht der Agent eine Bewertung seiner Aktion. Dies passiert mit den sogenannten *Rewards* (Belohnungen). Diese sind üblicherweise reelle Zahlen. Dadurch sind sowohl positive als auch negative Rewards möglich. RL-Agenten entwickeln selbständig eine *Policy* (Strategie), ohne dass ihm dafür vorher vorgezeigt wird, was richtig oder falsch ist. Dem Agenten müssen somit keine Lerndaten zur Verfügung gestellt werden, allerdings dafür eine passende Repräsentation des Zustands der Umgebung (*Observation*) sowie eine Reward-Funktion.

RL hat in den letzten Jahren durch medienwirksame Erfolge an Popularität gewonnen. Das asiatische Brettspiel *Go* [29] und viele alte *Atari Games* [20] konnten mit RL erlernt werden, genauso wie Interaktionen im physischen Raum zum Beispiel mit Roboter-Händen [22]. Zum Teil verdanken RL-Algorithmen ihre Erfolge den Fortschritten der KNNs, da viele moderne RL-Algorithmen Techniken von KNNs adaptieren konnten.

Umgebungen für RL können daran unterschieden werden, ob sie episodisch oder kontinuierlich sind. Episodische Umgebungen haben ein Ende, weil sie nur endlich viele Zeitschritte (*time steps*) ermöglichen. In jedem *time step* muss eine Aktion vom Agent gewählt werden. Kontinuierliche Umgebungen haben in der Theorie unendliche viele *time steps*. Der kumulative Reward $R_t \in \mathbb{R}$, auch *Return* genannt, zum Zeitpunkt $t \in \mathbb{N}$, lässt sich auf folgende zwei Arten formalisieren:

$$R_t = \sum_{k=0}^{\infty} r_{t+k+1} \text{ mit } k \in \mathbb{N} \qquad \textit{(kontinuierliche Variante)}$$

$$R_t = \sum_{k=0}^{T} r_{t+k+1} \text{ mit } k, T \in \mathbb{N} \qquad \textit{(episodische Variante)}$$

Wobei $r_t \in \mathbb{R}$ der Reward von der Aktion zum Zeitpunkt $t \in \mathbb{N}$ ist. Auch wenn für jede Aktion ein Reward ausgegeben werden kann, so ist dieser nicht obligatorisch. Es kann auch ein Reward mit dem neutralen Wert 0 ausgegeben werden. Umgebungen die selten Rewards ausgeben, werden als *sparse rewards environment* bezeichnet. Sie sind häufig schwerer zu erlernen, da es für viele Aktionen keine Rückmeldung gibt, ob sie richtig oder falsch waren.

Oft besteht der Bedarf, dass der Agent in möglichst wenigen *time steps* ein hohen Return erzielt. Damit soll verhindert werden, dass der Agent mehr Aktionen ausführt als nötig. Ein anderer Grund könnte sein, dass ein Reward tatsächlich höherwertig ist, je früher er ausgegeben wird, zum Beispiel bei einem Agenten der auf dem Aktienmarkt spekuliert. Um dieses Verhalten anzulernen gibt es einen sogenannten *Discount*-Faktor $\gamma \in (0, 1)$ aus $\mathbb{R}$. Dieser verkleinert den Reward je mehr *time steps* t vergangen sind. Der *Discount*-Faktor wird oft allgemein auf $0,99$ gesetzt und nur wenn es nötig ist, geringfügig reduziert. Der *Return* mit einem *Discount*-Faktor hat folgende Form:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \qquad \textit{(kontinuierliche Variante mit discount)}$$

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \qquad \textit{(episodische Variante mit discount)}$$

Im Gegensatz zu den *Returns* ohne *Discount*-Faktor, konvergieren die *Returns* mit *Discount*-Faktor immer gegen einen reelen Wert.

2.2.1 *Exploitation und Exploration*

Dadurch, dass RL-Agenten ihre Lerndaten anhand ihrer gewählten Aktion selber generieren, stellt sich bei jeder Wahl der nächsten Aktion die Frage, ob die Umgebung exploriert oder ausgebeutet (*exploitation*) werden soll. Der Agent muss abwägen, ob er eine völlig neue Strategie ausprobiert, oder seine bekannten Strategien verfeinert, um ein möglichst hohen Reward zu erzielen.

Sollte ein Agent nicht mehr explorieren, werden keine grundsätzliche neuen Erfahrungen gemacht und der Agent verharrt in seinem lokalem Optimum. Deshalb ist es für ein RL-Agenten essenziell im Lernmodus auch zu explorieren. Dazu gibt es unterschiedliche Techniken, die alle darauf beru-

hen, auch Aktionen auszuwählen, die nicht als eine optimale Wahl eingeschätzt werden.

Wie stark ein RL-Verfahren exploriert, kann meistens über einen Hyperparameter bestimmt werden. Für gewöhnlich muss dieser für jede Umgebung neu angepasst werden. RL-Agenten die zu wenig explorieren verharren in ihrem lokalem Optimum. RL-Agenten die zu viel explorieren kommen womöglich nicht dazu sich systematisch dem globalem Optimum zu nähern.

2.2.2 *Off/On-Policy-Verfahren*

Es gibt zwei große Gruppierung von RL-Verfahren, die *Off-Policy-* und die *On-Policy*-RL-Verfahren. Eine *Policy*, oft mit π bezeichnet, beschreibt die genaue Vorgehensweise eines Agenten. Beim *On-Policy*-RL-Verfahren gibt die Policy, anhand des Zustandes der Umgebung (*Observation*), die nächste Aktion vor. *Off-Policy*-RL-Verfahren besitzen keine Policy. Sie ermitteln die nächste auszuführende Aktion mit anderen Methoden, zum Beispiel in dem sie die Aktion wählen, die vermutlich zum höchsten Return führt. In den folgenden Unterkapiteln werden kurz zwei bekannte Vertreter der jeweiligen Gruppierungen vorgestellt.

Als Grundlage für diese Unterkapiteln wird an dieser Stelle die (State-)Value-Funktion und Action-Value-Funktion vorgestellt. Es wird dafür angenommen, dass eine Policy π existiert, welches das Verhalten des RL-Agenten steuert.

Die State-Value-Funktion, beziehungsweise oft nur Value-Funktion genannt, gibt den zu erwarteten Return ab Zeitschritt t unter der Policy π, wenn sich die Umgebung im Zustand S befindet, zurück.

Definiton 4 (Value-Funktion) *Sei S ein Zustand der Umgebung, R der Return und t der Zeitschritt. Dann ist Folgendes die Value-Funktion für die Umgebung:*

$$v_\pi(S) = \mathbf{E}_\pi[R_t|S_t = S]$$

Quelle: [34, K. 3.5]

Die Action-Value-Funktion gibt den zu erwarteten Return ab Zeitschritt t unter der Policy π, bei der Wahl der Aktion A im Zustand S, zurück.

Definiton 5 (Action-Value-Funktion) *Sei S ein Zustand der Umgebung, R der Return, A die Aktion und t der Zeitschritt. Dann ist Folgendes die Action-Value-Funktion für die Umgebung:*

$$q_\pi(S, A) = \mathbf{E}_\pi[R_t|S_t = S, A_t = A]$$

Quelle: [34, K. 3.5]

Ein Schätzer für die Value-Funktion wird mit V oder $\hat{v}$ bezeichnet und für die Schätzer eines Action-Value-Funktion wird die Bezeichnungen Q oder $\hat{q}$ benutzt.

2.2.2.1 *Off-Policy: Q-Learning*

Q-Learning ist ein von Watkins im Jahr 1989 vorgestellter Algorithmus, mit dem ein Schätzer für die Action-Value-Funktion $q(S, A)$ trainiert werden kann [37]. Es ist ein *one-step Temporal-Difference-Learning* Algorithmus, das bedeutet der Schätzer $Q(S, A)$ von $q(S, A)$ kann nach jeder Aktionsausführung anhand des Rewards angepasst werden. Dabei wird rekursiv der selbe $Q(S, A)$ genutzt, um den neuen maximal zu erreichenden kumulativen Reward zu berechnen. Dies wird im RL-Kontext *bootstrapping* genannt, da der Schätzer zum Teil mit den eigenen Schätzungen angepasst wird.

Ein Schätzer für Q-Learning wird am Anfang zufällig initialisiert und dann im Verlauf iterativ angepasst:

Definiton 6 (Q-Learning Update) *Seien S und S' Zustände von der Umgebung, A sowie a Aktionen, R der Reward und als Hyperparameter $\alpha \in (0, 1]$ als Schrittlänge und $\gamma \in (0, 1]$ der Discount-Faktor. Dann ist Folgendes die Update-Regel für ein Q-Learning System:*

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Quelle: [34, K. 6.5]

Die Schrittlänge α sollte dabei möglichst klein gewählt werden, weil ein zu hoher Wert eine Konvergenz des Agenten verhindert. Um mit Q-Learning auch explorieren zu können, kann die *ϵ-greedy*-Methode benutzt werden. Dabei wird mit einer Wahrscheinlichkeit von $\frac{\epsilon}{K}$ eine suboptimale Aktion gewählt, wobei K die Anzahl an Wahlmöglichkeiten für Aktionen darstellt. ϵ ist üblicherweise eine kleine Zahl im Bereich $[0.01, 0.2]$.

Es gibt unterschiedliche Möglichkeiten den Schätzer $Q(S, A)$ abzubilden, am Einfachsten mit einer Tabelle. In den letzten Jahren haben Deep-Learning-Ansätze an Beliebtheit gewonnen, wie beispielsweise *Deep Q Network* (DQN)[20], die die *Action-Values* mithilfe von KNNs schätzen.

2.2.2.2 *On-Policy: REINFORCE*

Um direkt eine Policy π zu trainieren, können *Policy-Gradients*-Methoden verwendet werden. Beim Training können gegebenenfalls *Value*-Funktionen genutzt werden. Zur Ermittlung der nächsten Aktion ist aber, im Gegensatz zu den Off-Policy-Verfahren, die Policy zuständig. Für *Policy-Gradients*-Methoden muss die Policy ein parametrisierbares Modell sein, das differenzierbar in Richtung der Parameter ist.

Definiton 7 (Policy-Gradients-Methode) *Sei $\pi(a|s, \boldsymbol{\theta})$ eine parametrisierbare Policy mit dem Parametervektor $\boldsymbol{\theta} \in \mathbb{R}^{d'}$, sodass gilt:*

$$\pi(a|s, \boldsymbol{\theta}) = \Pr\{A_t = a | S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\},$$

dann werden die Parameter der Policy folgenderweise um die Schrittlänge α angepasst:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}$$

Dabei ist $J(\theta)$ eine beliebige Funktion, die die Qualität des Parametervektors zurück gibt. *Quelle: [34, K. 13]*

Wichtig für das Trainieren der Policy ist, dass diese Policy nie deterministisch wird. Das bedeutet also $\pi(a|s, \boldsymbol{\theta}) \in (0, 1)$, ansonsten verliert der Agent die Fähigkeit zu explorieren. Der Vorteil bei dieser On-Policy-Methode ist, dass auch stochastische Policies erlernt werden können. Eine stochastische Policy kann von Vorteil sein, wenn nicht alle relevanten Informationen vorhanden sind, um die optimale Aktion vorhersagen zu können. Dies ist zum Beispiel bei Spielen mit anderen Teilnehmern der Fall, da der Spielzug des Gegner oft nur schwer vorhersagbar ist. Des Weiteren sollte auch der eigene Spielzug schwer vorhersagbar bleiben, was mit einer deterministischen Policy nicht möglich ist.

Ein weiterer Vorteil von On-Policy-Methoden ist, dass es relativ einfach ist sie *supervised* vorzutrainieren. Es kann durch Beispiele gelernt werden, oder womöglich ein bestehendes Regelwerk in die Policy überführt werden.

Im nächsten Abschnitt wird der *Policy-Gradients*-Algorithmus *REINFORCE* von Willams (1992, [38]) vorgestellt. Dieser arbeitet episodisch, das bedeutet erst nach Abschluss einer Episode wird die Policy angepasst.

Definiton 8 (REINFORCE) *Sei A_t die Aktion, S_t der Zustand zum Zeitpunkt t, $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ der Parametervektor und G_t der Return der gesamten Episode. Dann wird der Parametervektor um die Schrittlänge α folgendermaßen angepasst:*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi (A_t | S_t, \boldsymbol{\theta}_t)}{\pi (A_t | S_t, \boldsymbol{\theta}_t)},$$

wobei die Höhe von G_t bestimmt, wie stark die Anpassung des Parametervektors ist.

Quelle: [34, K. 13.3]

Die Policy von REINFORCE wird meistens als mehrschichtiges KNN abgebildet. Somit können die normalen KNN-Frameworks mit Backpropagation genutzt werden. Ein großer Unterschied besteht allerdings beim Loss: Während bei klassischen KNNs der Fehler minimiert wird, wird bei einer Policy der Return G_t maximiert. Statt *Stochastic Gradient Descent* wird also *Stochastic Gradient Ascent* angewendet.

Um die Konvergenzeigenschaften von REINFORCE zu verbessern, wird meistens eine Version mit *baseline*-Vergleich genutzt. Die *baseline* gibt dabei an, wie der Return zuvor abgeschätzt wurde und kann zum Beispiel eine *Value*-Funktion sein. Dadurch kann die Stärke der Anpassung der Policy

davon abhängig gemacht werden, wie groß der Vorteil (*advantage*) der neuen Vorgehensweise wäre.

Definiton 9 (REINFORCE mit baseline) *Seien die Variablen wie zuvor in Definition 8 definiert und $b(s)$ eine beliebige baseline:*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(G_t - b(S_t))\frac{\nabla\pi\left(A_t|S_t, \boldsymbol{\theta}_t\right)}{\pi\left(A_t|S_t, \boldsymbol{\theta}_t\right)},$$

wobei der Unterschied von G_t zu $b(S_t)$ bestimmt, wie stark die Anpassung des Parametervektors ist.

Quelle: [34, K. 13.4]

2.2.3 *Proximal Policy Optimization*

In diesem Unterkapitel wird der RL-Algorithmus *Proximal Policy Optimization* (PPO) vorgestellt. Dies ist auch der Algorithmus, der in dieser Masterarbeit hauptsächlich zum Einsatz kommt.

PPO wurde von Schulman et al. bei OpenAI entwickelt und im Jahr 2017 veröffentlicht [28]. Dieser hat zum Ziel, technisch einfach und trotzdem hinsichtlich der Anzahl an benötigten *time steps* effizient zu sein (*sample efficiency*). Entstanden ist ein Algorithmus, der aktuell zu den leistungsstärksten RL-Algorithmen gehört und die *Advantage-Actor-Critic*-Methode (A2C) nutzt. *Advantage-Actor-Critic* ist eine On-Policy-Methode, wobei auch Off-Policy-Methoden mit verwendet werden.

Der *Actor* ist dabei die Policy, die normalerweise mit REINFORCE trainiert wird und somit für die Entscheidungsfindung zuständig ist. Der *Critic* dient dazu beim Lernen der Policy zu unterstützen und ist im Falle von PPO eine *Value*-Funktion.

Definiton 10 (Advantage-Actor-Critic-Methode) *Seien die Variablen wie in Definition 8 definiert und zusätzlich $\hat{v}\left(S_t, \mathbf{w}\right)$ eine State-Value-Funktion als Criticer sowie R_t der Reward vom aktuellem Schritt.*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(R_t + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\frac{\nabla\pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$$

Dabei ist der Teil $(R_t + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))$ der advantage, der angibt, ob die Aktion A_t bessere oder schlechtere Resultate erzeugt als der Durchschnitt aller Aktion.

Quelle: [34, K. 13.5]

Der Unterschied von REINFORCE mit einer *Value*-Funktion als *baseline* und der Nutzung der *Value*-Funktion bei A2C besteht darin, dass die *baseline* lediglich als relatives Maß der Verbesserung dient. Bei A2C dagegen dient die *Value*-Funktion zum *bootstrapping*, in dem der Value nachfolgender Zustände betrachtet wird. Aus diesem Grund ist A2C auch im Gegensatz zu

REINFORCE keine episodische Methode, sondern kann nach beliebig vielen *time steps* eine Anpassung an der Policy durchführen.

Bei der Entwicklung von PPO wurde das Verhalten vom A2C-Algorithmus *Trust Region Policy* (TRPO) [27] analysiert. Dieser fällt dadurch auf, dass Veränderungen an der Policy beschränkt werden. Das liegt daran, dass A2C-Methoden beim Lernen wegen der Nutzung von *Policy Gradients* sehr instabil sind. Beim verändern der Policy können erlernte Fähigkeiten wieder verloren gehen. Deshalb wird versucht die Anpassung der Policy durch Methoden, wie die Differenzierung zweiter Ordnung (in TRPO) oder Kullback-Leibler-Divergenz (einige PPO Varianten) einzuschränken. Damit sollen zu große, womöglich zerstörerische Veränderungen an der Policy verhindert werden.

Die hier nun vorgestellte und häufigste genutzte Variante von PPO (*clipped*) hat für die Anpassung der Policy eine statische Bandbreite. Änderungen die größer sind werden beschnitten. Dazu wird berechnet, wie stark die Wahrscheinlichkeit sich unterscheidet, eine Aktion A_t im Zustand S_t zu wählen:

$$r_t(\theta) = \frac{\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta_{\text{old}})}$$

Sollte die Wahrscheinlichkeit gleich sein, oder die neuen Policy-Parameter θ identisch zu den alten Policy-Parameter θ_{old}, dann ist der *ratio* $r_t(\theta)$ vom *time step t*: $r_t(\theta) = 1$. Sollte dieser zu stark abweichen, wird dieser anhand des *Clip*-Hyperparameter ϵ beschnitten. Der Hyperparamter ϵ wird meistens mit einem Wert aus dem Bereich $[0.1, 0.3]$ gesetzt. Die beschnittene Anpassung ist somit folgende:

$$L_t^{\text{CLIP}}(\theta) = \min(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t)$$

Dabei ist $\hat{A}_t$ der geschätzte Vorteil (*advantage*) an kumulierten Rewards bezüglich der gewählte Aktion A_t gegenüber einer alternativen Aktion, die von der alten Policy gewählt worden wäre.

Der *Advantage* besteht aus den beobachteten Rewards und den noch zu erwartenen Rewards, welche von der Value-Funktion $\hat{v}$ abgeschätzt werden. PPO macht eine feste Anzahl an T *time steps* bevor eine Anpassung der Policy und der *Value*-Funktion stattfindet und bezieht ein *Discount*-Faktor γ ein. Der *Advantage* ist für die Beobachtung zum Zeitpunkt t somit:

$$\hat{A}_t = R_t + \gamma R_{t+1} + \cdots + \gamma^{T-t+1} R_{T-1} + (\gamma^{T-t}\hat{v}(S_T, w) - \hat{v}(S_t, w)) \quad (2.3)$$

Für die Anpassung der Policy wird noch die quadratische Abweichung der *Value*-Funktion vom beobachteten Value benötigt:

$$L_t^{\text{VF}}(\theta) = \left(\hat{v}_\theta\left(S_t, w\right) - V_t^{\text{targ}}\right)^2 \quad (2.4)$$

und ein Entropie-Term $S[\pi_\theta](S_t)$, der dafür sorgt, dass der RL-Algorithmus nicht irgendwann aufhört zu explorieren. Das geschieht durch ein kleinen

Bonus, den Aktionen bekommen dessen Wahl unwahrscheinlich ist. Die Anpassungsregel für PPO sieht folgendermaßen aus:

Definiton 11 (PPO Policy Update) *Seien die Variablen und Funktionen, wie im vorherigen Abschnitt definiert, dann beschreibt $L_t(\theta)$ den Loss mit einem Batch von T vielen Beobachtung, beziehungsweise T vielen time steps:*

$$L_t(\theta, w) = \hat{\mathbf{E}}_t[L_t^{CLIP}(\theta) - c_{VF} L_t^{VF}(w) + c_E S[\pi_\theta](S_t)], \tag{2.5}$$

wobei c_{VF} und c_E die Hyperparameter/Koeffizienten für die Value-Funktion und die Entropie sind.

Quelle: [28]

Die Umsetzung des Algorithmus in Pseudo-Code kann folgendermaßen aussehen:

Algorithmus 1 : PPO

```
random initialization of θ_0 and w_0
for i=1,2, ... do
    for actor=1,2, ..., N do // parallel execution loop
        for t=1,2, ..., T do
            A_t := Get new action from π(S_t, θ_i)
            S_{t+1}, R_t := Evaluate action A_t on environment
            Â_t := Calculate advantage (2.3)
        end
    end
    θ_{i+1} := Optimize policy with SGD and L_t(θ_i, w_i) (2.5)
    w_{i+1} := Optimize value function with SGD and L_t^VF(w_i) (2.4)
end
```

Eine weitere Darstellung des PPO-Algorithmus ist im Kapitel 4 (Eigener Ansatz), in Abbildung 4.4.2 und 4.5.4, zusehen.

3 VERWANDTE ARBEITEN

In diesem Kapitel werden einige wichtige Publikationen vorgestellt, die als Basis und Inspiration für die hier vorgelegte Masterarbeit dienen. In der folgenden Tabelle 3 ist eine Auswahl der recherchierten Publikationen zusammengefasst. Dabei ist in den Spalten „GN“ und „RL“ vermerkt, ob die Ansätze in den Publikationen generalisierbar (GN) sind und ob sie *Reinforcement Learning* (RL) nutzen. Ein Ansatz gilt im aktuellem Kontext als generalisierbar, wenn das Ergebnis des Ansatzes wiederverwendet werden kann für ein unbekannten Datensatz, an dem zuvor der Ansatz nicht Trainiert worden ist.

Ein Teil der erwähnten Publikationen werden nun in den nächsten Unterkapiteln näher beleuchtet. Pro Unterkapitel wird eine Publikation vorgestellt, wobei jedes Unterkapitel am Ende einen von mir „persönlichen Kommentar“ enthält. Auf diese Weise möchte ich Aspekte der Publikationen hervorheben, die mir besonders aufgefallen sind, oder die ich in Bezug zu meiner Masterarbeit für wichtig halte.

Titel	Veröffentlichung	Autoren	Schwerpunkt	GN?[a]	RL?[b]	Kurz Titel	Kapitel
Neural Architecture Search with Reinforcement Learning	2017	Zoph, Barret Le, Quoc V.	Finden eines guten KNNs für einen Datensatz; CNNs & RNNs werden untersucht; Nutzt REINFORCE mit RNNs	✗[c]	✓[d]	NAS	3.1
Learning Transferable Architectures for Scalable Image Recognition	2018	Zoph, Barret; Vasudevan, Vijay Shlens, Jonathon Le, Quoc V.	Anpassung von NAS auf die Suche einer optimalen CNN-Zelle; REINFORCE wurde durch PPO ersetzt	◗[e]	✓	NASnet	3.1.1
Progressive Neural Architecture Search	2018	Liu, Chenxi Zoph Barret et al.	Suchen einer optimalen CNN-Zelle mithilfe von Sequential Model Base Optimization	◗	✗	PNAS	3.1.2
DeepArchitect: Automatically Designing and Training Deep Architectures	2018	Negrinho, Renato Gordon, Geoff	Suchen einer optimalen KNN-Architektur für einen Datensatz mithilfe der Monte-Carlo-Methode	✗	✗		
Designing Neural Network Architectures using Reinforcement Learning	2017	Baker, Bowen Gupta, Otkrist et al.	Suche einer optimalen CNN-Architektur für einen Datensatz mithilfe von Q-Learning	✗	✓	MetaQNN	3.2
Practical Block-wise Neural Network Architecture Generation	2018	Zhong, Zhao Yan, Junjie et al.	Finden einer optimalen CNN-Zelle mithilfe von Q-Learning	◗	✓	BlockQNN	3.3
Large-Scale Evolution of Image Classifiers	2018	Real, Esteban Le, Quoc et al.	Suche einer optimalen KNN-Architektur für einen Datensatz mithilfe von evolutionären Algorithmen	✗	✗		
Auto-Keras: An Efficient Neural Architecture Search System	2017	Jin, Haifeng Song, Qingquan Hu, Xia	Suche eines optimalen KNNs mithilfe von Architektur-Morphismus und Bayesian Optimization	✗	✗	AutoKeras	

a Ist das trainierte Resultat *generalisierbar*? Kann die *Policy* auch für andere Datensätze angewendet werden?
b Wird *Reinforcement Learning* benutzt?
c Das Zeichen ✗ bedeutet, dass die Eigenschaft kein Bestandteil der Veröffentlichung ist
d Das Zeichen ✓ bedeutet, dass die Eigenschaft ein Bestandteil der Veröffentlichung ist
e Das Zeichen ◗ bedeutet, dass die Eigenschaft teilweise ein Bestandteil der Veröffentlichung ist

3.1 NETWORK ARCHITECTURE SEARCH (NAS)

Im Jahr 2017 veröffentlichen Barret Zoph und Quoc V. Le von Google Brain ihre Arbeit zu „Network Architecture Search with Reinforcement Learning“[41] (NAS). Sie entwickelten einen Agenten, um eine möglichst gute CNN-Architektur für ein Bildklassifikation-Datensatz zu finden. Der Agent muss dafür in jedem *time step* ein vorgegebenes Attribut vom aktuellen Conv-Layer bestimmen. Die Anzahl der Conv-Layers ist dabei vorgegeben und erhöht sich mit der Trainingsdauer vom Agenten. Die Ausgabe der Aktionen vom Agenten kann als einer Art DNA-Sequenz vom CNN interpretiert werden, wo jede Sequenz (Aktion) ein Attribut vom CNN definiert.

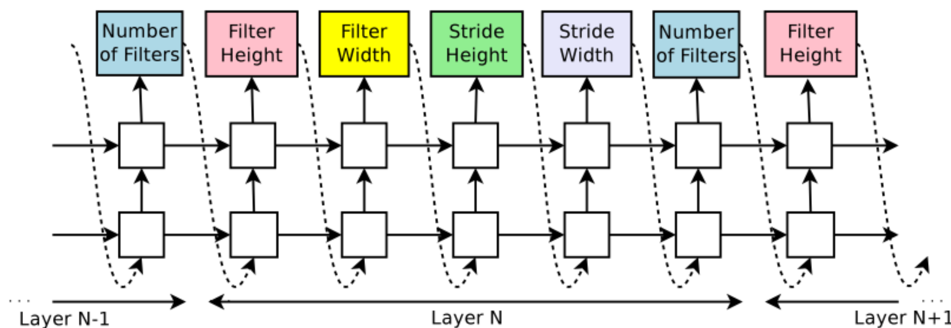


Abbildung 3.1.1: Zwei LSTM-Layer die sequentiell die Conv-Layers parametrisieren. Bild entnommen aus der original Publikation [41].

Nach dem der Agent alle Attribute der CNN-Architektur bestimmt hat, wird das CNN trainiert und die Genauigkeit auf dem Validierungsdatensatz (*validation accuracy*) als Reward für die Episode ausgegeben. Der Agent wird mit dem *Policy-Gradients*-Algorithmus REINFORCE mit *baseline* (siehe Definition 9) trainiert, wobei als *baseline* ein gleitender, exponentieller Mittelwert der vorherigen Genauigkeiten auf dem Validierungsdatensatz genutzt wird. Dies passiert nach jeder Episode, das heißt, nach jeder trainierten CNN-Architektur. Das Trainieren der CNN-Architekturen wurde dabei auf mehreren Rechnern parallel ausgeführt und das Trainieren des Agenten geschah asynchron. Das Policy-Modell vom Agenten ist ein RNN[1], bestehend aus zwei LSTM-Layer[2]. Somit hat der Agent bei seiner Entscheidungsfindung die Möglichkeit Informationen zu seinen letzten getroffenen Aktionen heranzuziehen.

Nach dem erste Tests erfolgreich waren, wurden dem Agenten weitere Fähigkeiten implementiert. Diese sollen dem Agenten ermöglichen CNN-Architekturen zu erstellen, die qualitativ mit den bekannten Standard-Lösungen mithalten können. Die Autoren der Publikation implementierten Pooling sowie die Möglichkeit Verzweigungen im CNN einzubauen (*branching layers*). Dadurch können Daten auf mehreren Pfaden durch das CNN propagiert

1 *Rekurrentes Neuronale Netze* sind KNNs bei denen die Signale der Schichten auch rückgekoppelt werden und somit Informationen aus den vorherigen Iterationen speichern

2 *Long short-term memory* ist ein Schicht-Typ welches in RNNs verwendet wird und besonders gut geeignet ist für Netze mit vielen Schichten

werden. Das setzten sie um, in dem jede Schicht ein Anker-Attribut besitzt, dass spezifiziert, welche vorherige Schicht als Eingangssignal für die aktuelle Schicht genutzt werden soll.

Schichten die von keiner anderen Schicht als Eingangssignal genutzt werden, dienen als Eingangssignal für den Output-Layer. Aufgrund ihrer unterschiedlichen Dimensionen werden Conv-Layer, die inkompatibel zueinander sind, durch das Ersetzen fehlender Neuronen durch Null'er, angeglichen (*padding with zeros*).

Auf das Manipulieren von *batchsize*, *learning rate* so wie auf andere Schicht-Typen, außer Conv-Layers, wurde verzichtet. Die Autoren der Abhandlung erwähnten aber ausdrücklich, dass durch eine Anpassung des Agenten, beziehungsweise der Umgebung, dies möglich wäre.

Für die Experimente nutzten die Autoren den Datensatz CIFAR-10 (siehe Appendix A). Für das Training wurden, wie heutzutage üblich, die Bilder zufällig in Orientierung und Ausschnitt angepasst (*augmentation*). Als Aktivierungsfunktion wurde ReLU genutzt und jede CNN-Architektur wurde 50 Epochen lang trainiert. Die Auswahl an Aktionen für die Attribute waren alle, mit ungefähr 5 verschiedenen Optionen, relativ klein. So konnte sich zum Beispiel der Agent für die Filter-Höhe eines Conv-Layer lediglich eine von folgende Optionen wählen: $[1, 3, 5, 7]$.

Beim Trainieren des Agenten wurde am Anfang mit sechs Schichten begonnen, wobei die Anzahl nach jeweils 1.600 trainierten CNNs um zwei Schichten erhöht wurde. Nach 12.800 trainierten CNNs wurde das beste Ergebnis mit einer Genauigkeit von 96,35% erreicht. Dies ist ungefähr ein Wert, der auch von Menschen konstruierte CNNs, zum Zeitpunkt der Veröffentlichung von NAS, erreichen.

Zusätzlich zu erwähnen ist, dass auch Experimente zur Konstruktion von Zellen für RNNs ein Bestandteil der Publikation sind. Statt nach einer Architektur aus Schichten wurden dabei nur nach einer Architektur von mathematischen Operationen gesucht, um eine möglichst gute RNN-Zelle zu entwickeln. Dies gelang auch ähnlich erfolgreich. Weil RNNs kein Bestandteil dieser Arbeit sind, wird an dieser Stelle nicht weiter darauf eingegangen und dem interessiertem Leser auf die ursprüngliche Veröffentlichung verwiesen.

Persönlicher Kommentar: Der Agent ist bei seiner Konfigurationsmöglichkeiten der Architektur stark eingeschränkt, auch wenn die branching layers eine interessante und mächtige Fähigkeit ist. Der RL Agent wird hier ausschließlich zur Parametersuche genutzt und total fixiert auf nur einen Datensatz. Die erlernte Policy ist somit auf andere Datensätze nicht anwendbar.

3.1.1 *NASNet*

Aufsetzend auf der vorherigen Veröffentlichung [41] haben Barret Zoph und weitere Google Brain Mitarbeiter versucht, wiederverwendbare CNN-Zellen zu finden [42]. Es wird also nicht eine ganze CNN-Architektur vom Agenten

konfiguriert, sondern lediglich eine einzelne Zelle, die dann mehrfach im CNN zur Anwendung kommt. Dieses Prinzip, eine Zelle zu entwickeln und diese dann mehrfach zu verwenden, wird auch in andern bekannten CNN-Architekturen (ResNet[9], Inception[36]) benutzt.

Der grundlegende Gedanke bei dieser Veröffentlichung war, das Training des Agenten zu beschleunigen, in dem mit dem Agenten auf einem einfacherem, kleinerem Datensatz eine gute CNN-Zelle gesucht, um diese Zelle später auf einem größeren, komplexeren Datensatz einzusetzen. Die CNN-Zelle sollte in soweit generalisierbar sein, dass diese auch für andere, ähnliche Datensätze genutzt werden kann. In ihren Experimenten nutzten die Autoren den CIFAR-10 Datensatz zum Trainieren des Agenten und setzten dann die beste gefundene CNN-Zelle mit mehrfacher Verwendung für den Datensatz ImageNet (siehe Appendix A) in einem CNN ein. Neu ist bei dem veröffentlichten Ansatz zudem, dass statt wie ursprünglich REINFORCE, *Proximal Policy Optimization* (Unterkapitel 2.2.3) verwendet wurde.

Auf dem CIFAR-10 konnten die Autoren damit eine Genauigkeit von 97,6% erreichen. Mit der gefundenen CNN-Zelle konnten sie wiederum auf dem ImageNet eine Genauigkeit von 82% erreichen. Dieses Resultat entspricht dem Stand der Technik. Durch die Verkleinerung des Aktions-/Suchraums auf die Konfiguration einer CNN-Zelle, anstatt eines ganzen CNNs, konnte das Training vom Agent um das siebenfache verschnellert werden. Dennoch benötigten 500 GPUs vier Tage lang dafür und trainierten dabei 20.000 verschieden CNNs.

Persönlicher Kommentar: Interessanter wäre es in diesem Kontext gewesen, wenn das Trainieren des Agenten auf dem Zieldatensatz stattgefunden hätte. Es wäre zum Beispiel möglich für das Training des Agenten nur einen Teil der Daten zu nutzen. Vermutlich ist dies die praktikablere Variante, da in der Realität nicht in allen Situationen ein alternativer, vereinfachter Datensatz zum Zieldatensatz zu Verfügung steht.

3.1.2 *PNAS*

Chenxi Liu, Barret Zoph et al. veröffentlichen mit „Progressive Neural Architecture Search“[18] eine alternative Umsetzung von dem NASNet-Ansatz [41]. Statt *Reinforcement Learning* nutzen sie die heuristische Methode *Sequential Model-Based Optimization* (SMBO). Dabei wird aufgrund der Performance mit simplen CNN-Zellen begonnen und im Verlauf der Suche werden immer komplexere CNN-Zellen evaluiert. Zur Entscheidungsfindung, welche CNN-Zellen-Architektur als nächstes trainiert werden soll, wird parallel zur Suche ein Schätzer trainiert, der die potenzielle Qualität einer CNN-Zelle vorhersagt. Die Ergebnisse der trainierten CNN-Architekturen dienen als Feedback für den Schätzer. Damit der Schätzer möglichst genaue Schätzungen über die Qualität vorhersagen kann, wurden nur CNN-Zellen betrachtet, die sich nicht zu stark von den bisher trainierten Zellen unterscheiden. In je-

dem Schritt wird eine feste Anzahl an neuen CNN-Zellen trainiert, dabei sind es die, die vom Schätzer die höchste vorhergesagte Qualität haben.

Die produzierten Resultate sind ähnlich gut, wie die von der ursprünglichen Umsetzung des NASNets. Allerdings ist PNAS fünfmal effizienter (*sample efficiency*) und achtmal schneller (*compute time*) als die RL-Variante.

Persönlicher Kommentar: Der verwendete Schätzer hat Ähnlichkeiten zu der Value-Funktion von PPO im NASNet. Des Weiteren ist es bemerkenswert, dass keine Methode implementiert wurde um bei PNAS eine Exploration zu erzwingen. Schlussendlich erscheint der Ansatz mit SMBO tatsächlich effektiver zu sein. Was wichtig ist, wenn man bedenkt, dass die Ansätze mit RL eine Policy trainieren die nur für ein Datensatz passt. Für ein neuen Datensatz muss die Policy verworfen und eine neue Policy trainiert werden.

3.2 METAQNN

MetaQNN ist ein Algorithmus von Bowen Baker et al. vom MIT aus dem Jahr 2017 [1]. Der Algorithmus arbeitet auf der Basis von Q-Learning (siehe 2.2.2.1) mit *experience replay* und hat zum Ziel, eine gute CNN-Architektur für einen bestimmten Datensatz zu finden. Dazu kann der Agent in jedem *time step* entscheiden, ob dieser eine Schicht inklusive seiner Parameter hinzufügt oder den Konstruktionsvorgang terminiert und damit die konstruierte CNN-Architektur trainiert. Auch hier wird die Genauigkeit auf dem Validierungsdatensatz dem Agenten als Reward übermittelt.

Es wurde versucht den Aktionsraum möglichst eng zu fassen. Der Agent kann als Schicht-Typ ein Conv-Layer, ein Max-Pooling-Layer oder ein normales Fully-Connected-Layer wählen. Dabei wählt dieser im selbem Schritt die Parametrisierung der Schichten, wobei diese fast alle kategorisch und nicht numerisch sind, siehe „Parameter Values“ in Abbildung 3.1.1. Der Agent kann nur eine vorgegebene maximale Anzahl an Schichten hinzufügen. Um zusätzlich den Agenten einzuschränken, haben die Autoren Eigenschaften von typischen Architekturen bekannter CNNs implementiert. So dürfen nur maximal zwei FC-Layer verwendet werden und eine Schicht darf nur gleich viele oder weniger Neuronen besitzen, als die vorherige Schicht. Darüber hinaus ist es dem Agenten nicht erlaubt zwei Pooling-Layer dem CNN hintereinander hinzuzufügen, weil diese zu einem Pooling-Layer zusammengefasst werden können. Um zu verhindern, dass Conv-Layer und Pooling-Layer die Eingabe so stark verkleinern, dass die Daten auf 1x1-Format reduziert werden, wird die Parametrisierung, abhängig von der Ausgabegröße (*R-size*) der letzten Schicht, eingeschränkt.

Die Repräsentation des Zustands der Umgebung des Agenten (*Observation*), ist die Beschreibung der letzten hinzugefügten Schicht. Die Struktur ist in Abbildung 3.1.1 zu sehen.

Vor dem Training der CNN-Architekturen wird nach jeder zweiten Schicht ein Dropout-Layer hinzugefügt und nach einem festen Schema parametrisiert. Jede Architektur wird mit *Adam* 20 Epochen lang trainiert. Sollte die

Layer Type	Layer Parameters	Parameter Values
Convolution (C)	$i \sim$ Layer depth $f \sim$ Receptive field size $\ell \sim$ Stride $d \sim$ # receptive fields $n \sim$ Representation size	< 12 Square. $\in \{1, 3, 5\}$ Square. Always equal to 1 $\in \{64, 128, 256, 512\}$ $\in \{(\infty, 8], (8, 4], (4, 1]\}$
Pooling (P)	$i \sim$ Layer depth $(f, \ell) \sim$ (Receptive field size, Strides) $n \sim$ Representation size	< 12 Square. $\in \{(5, 3), (3, 2), (2, 2)\}$ $\in \{(\infty, 8], (8, 4]$ and $(4, 1]\}$
Fully Connected (FC)	$i \sim$ Layer depth $n \sim$ # consecutive FC layers $d \sim$ # neurons	< 12 < 3 $\in \{512, 256, 128\}$
Termination State	$s \sim$ Previous State $t \sim$ Type	Global Avg. Pooling/Softmax

Abbildung 3.2.1: Beschreibung des Zustandsraums, abhängig vom Layer-Typ. Tabelle ist entnommen aus dem original Paper [1].

Architektur nach der ersten Epoche nicht konvergieren, wird das Training mit einer um den Faktor $0,4$ reduzierten Lernrate wiederholt. Dies wird bis zu fünf Mal ausprobiert. Zur Exploration nutzt MetaQNN den *ϵ-greedy* Algorithmus, wobei das ϵ am Anfang des Agenten-Trainings sehr hoch ist und mit der Anzahl an erprobten CNN-Architekturen immer weiter sinkt. MetaQNN exploriert also am Anfang sehr stark und im Verlaufe des Agenten-Trainings *exploitet* es immer mehr. Um zu vermeiden, dass die selbe CNN-Architektur mehrmals trainiert wird, speichert sich das MetaQNN die Ergebnisse aller trainierten CNN-Architekturen ab und kann somit im Wiederholungsfall ohne Training direkt den Reward zurückgeben.

Für ihre Experimente nutzten die Autoren die Datensätze *Street View House Numbers* (SVHN), CIFAR-10 und MNIST (siehe Appendix A). Dafür benötigten sie pro Datensatz mit 10 GPUs acht bis zehn Tage. Die Ergebnisse sind vergleichbar gut, wie die von bekannten CNN-Architekturen, die auf den selben technischen Fähigkeiten (Conv-Layer, Max-Pooling, etc.) basieren. Der CIFAR-10-Datensatz konnte mit einer Fehlerrate von 6,92% die technisch ähnliche Architektur *VGGnet*[31], die eine Fehlerrate von 7,25% hat, geringfügig unterbieten.

Persönlicher Kommentar: Das Ziel der Publikation ist weniger die bekannten Bestmarken oder NAS in der Genauigkeit zu übertreffen, als eine technische Alternative aufzuzeigen, die im Vergleich zur NAS Publikation weniger Rechenleistung benötigt. Das Ausprobieren verschiedener Lernraten ist ein interessanter Ansatz.

3.3 BLOCKQNN

In der Veröffentlichung von Zhong et al. vom Jahr 2018 wird eine Abwandlung von MetaQNN [1] vorgestellt, die den Namen BlockQNN hat [40]. Wie MetaQNN arbeitet BlockQNN auch mit dem Q-Learning Algorithmus in Kombination mit der *ϵ-greedy* Methode. In BlockQNN wird aber nicht nach einer optimalen CNN-Architektur gesucht, sondern nur nach einer optimalen CNN-Zelle. Die Situation verhält sich hier ähnlich wie bei den vorgestell-

ten Algorithmen NAS (Kapitel 3.1) und NASnet (Unterkapitel 3.1.1). Ein Vergleich der unterschiedlichen Ansätze ist in Abbildung 3.3.1 zu sehen.

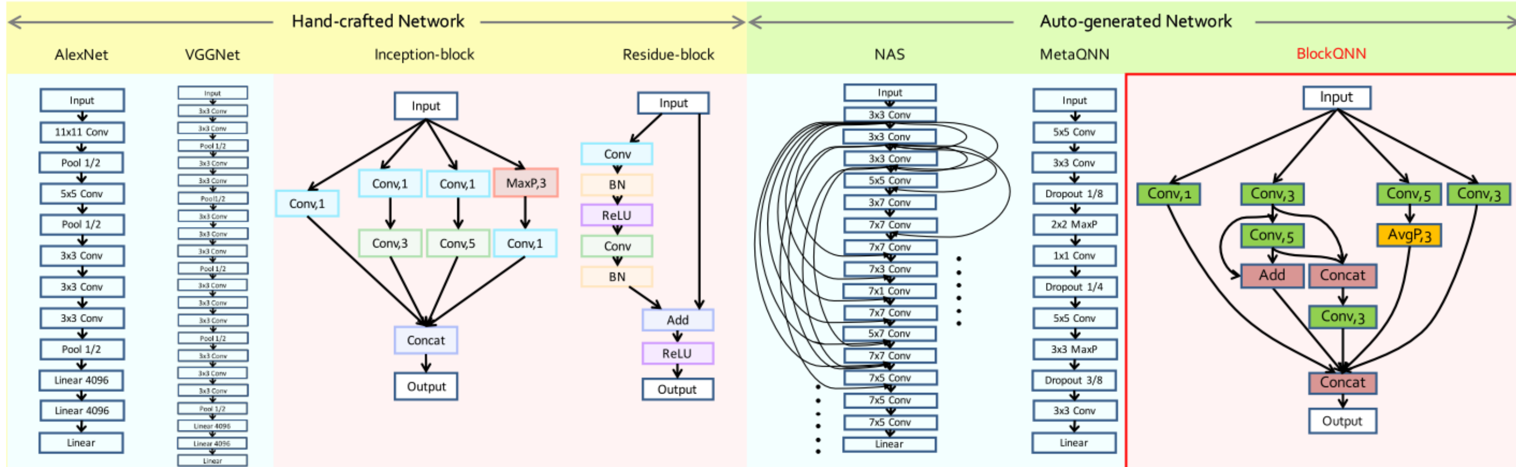


Abbildung 3.3.1: Gegenüberstellung von manuell und automatisch generierten erstellten CNN-Architekturen (*hand-crafted*). Architekturen mit blauem Hintergrund sind vollständige CNNs und Architekturen mit rotem Hintergrund sind CNN-Zellen (*blocks*). Die Grafik ist entnommen aus der original Publikation [40]

Als Aktion hat der Agent die Möglichkeit, die Konstruktion zu terminieren um das Training der CNN-Architektur zu starten oder eine Schicht hinzuzufügen, wobei angegeben werden muss welche vorherige Schicht als Eingangssignal verwendet werden soll (Stichwort: *branching layer*). Folgende Schicht-Typen stehen dabei zur Auswahl: Max-Pooling, Average-Pooling, Elemental-Add[3], Concat[4] und Conv-Layer. Der Conv-Layer selbst ist dabei als *Pre-activation Convolutional Cell* (PCC, [10]) definiert. Dies bedeutet, dass dieser aus einem gewöhnlichen Conv-Layer mit *Batch Normalization* besteht und eine ReLU-Aktivierungsfunktion verwendet. Dies ist eine bewährte Kombination von vielen CNN-Architekturen und soll den Suchraum für die CNN-Zellen verkleinern. Jede Aktion beziehungsweise Schicht besteht aus einem sogenannten *Network Structure Code* (NSC), welcher ein numerisches Fünfer-Tupel ist und die Parametrisierung einer Schicht darstellt. In den vorherigen vorgestellten Veröffentlichungen, ist die Beschreibung der Schichten ähnlich umgesetzt. Die letzte Aktion, beziehungsweise Schicht, ist zugleich auch die aktuelle Observation, welche dem RL-Agenten übergeben wird.

Damit der Agent schneller konvergiert, wird der *Return* gleichmäßig auf alle Aktionen als Reward in der Episode verteilt (*shaped intermediate reward*). Normalerweise haben sonst alle Aktionen einen Reward von Null und die letzte Aktion *Termination/Training* gibt die Genauigkeit auf dem Validierungsdatensatz als Reward zurück. Um Trainingszeit zu sparen, wird mit *early-stopping* das Training einer Zelle verfrüht abgebrochen. Voraussetzung für den Abbruch ist, dass die Genauigkeit innerhalb eines gewissen Zeit-

3 Eingangssignale von zwei Schichten miteinander addiert
4 Eingangssingnale von zwei Schichten bilden eine neue Schicht

raums sich nicht verbessert hat. Dabei wurde von den Autoren beobachtet, dass die Abweichung der beobachteten Genauigkeit zur echten Genauigkeit ohne *early stopping* in negativer Korrelation zur „Dichte" der Zelle und Berechnungskomplexität der Zelle in FLOPS steht. Dies wurde in der Reward-Funktion mit einbezogen, um eine möglichst genaue Abschätzung der Genauigkeit zu erreichen.

Bei den vorgestellten Resultaten wurde eine maximale Anzahl von 23 Schichten verwendet. Es wurde auf dem CIFAR-100 Datensatz trainiert und dabei der ϵ-Wert für die Exploration im Verlauf des Agenten-Trainings stetig von 1.0 auf 0.1 reduziert. Der *Discount*-Faktor war mit dem Wert 1 deaktiviert.

Es wurden 11.392 Zellen innerhalb von 3 Tagen mit 32 GPUs evaluiert. Unter mehrfacher Verwendung der gefundenen Zelle im CNN (2 bis 5-fach), konnte auf dem CIFAR-10 Datensatz eine Fehlerrate von 3,54% und auf dem CIFAR-100 18% erreicht werden. Somit wurde eine Genauigkeit erreicht, die ähnlich gut ist, wie die von NAS und NASnet. Die selbe Zelle wurde mit 3-facher Ausführung in einem CNN auf dem Datensatz ImageNet trainiert. Dabei konnte mit einer Fehlerrate von 22,6% ein sehr gutes Ergebnis erzielt werden, was die Generalisierbarkeit der Zelle beweisen soll.

Persönlicher Kommentar: Auffallend ist, dass direkt auf dem komplexeren CIFAR-100 trainiert wird, anstatt auf dem einfacherem CIFAR-10, wie bei den anderen Publikationen. Interessant ist die Einbeziehung der Berechnungskomplexität und der Dichte einer Zelle für die Reward-Berechnung. Die Publikation ist ziemlich ausführlich, aber leider fehlte eine genauere Beschreibung der eingesetzten early-stopping-Methode. Des Weiteren ist die Aussage zur Generalisierbarkeit vermutlich ziemlich eng zu sehen, weil die CIFAR und ImageNet-Datensätze sehr ähnlich sind von den Daten und von der Aufgabenstellung.

4 EIGENER ANSATZ

Dieses Kapitel ist dem selbst entwickeltem Ansatz gewidmet. Die Entwicklung ist in drei Versionen unterteilt, die aufeinander aufbauen. Die ersten zwei Versionen dienen als Vorstufe des eigenen Ansatzes und als technisches *Proof-of-Concept*, um das Verhalten der Umgebung mit dem RL-Algorithmus zu untersuchen. Dadurch wird eine Verbesserung des Vorgehens während der Bearbeitung der Masterarbeit ermöglicht, um sich iterativ der Umsetzung des eigenen Ansatzes zu nähern. Erst die dritte Version besitzt die technischen Möglichkeiten das anvisierte Ziel dieser Arbeit (siehe 1.2) zu erreichen. Diese ist die finale Version und entspricht dem angedachten eigenen Ansatz. Für jede Version existiert ein Unterkapitel in dem der Zweck, der Aufbau und die Evaluierung beschrieben ist. Ein Übersicht dazu ist in der Tabelle 4.0.1 zu sehen.

Version	Ziele	Kurze Beschreibung
Version 1	Ein gutes KNN für ein Level finden. Level wird anhand der Größe eines Datums wiedererkannt. Lernverhalten von PPO & Umgebung analysieren.	KNNs werden mit Chainer trainiert. Aktionen bestehen aus numerischen Dreier-Tupeln.
Version 2	Ein gutes KNN für ein Level finden. Level wird anhand der Größe eines Datums wiedererkannt. Lernverhalten von PPO, Umgebung und ASM[1]analysieren.	KNNs werden mit Tensorflow statt Chainer trainiert. Aktionen besteht nur aus einem numerischen Wert. Einführung der ASM mit Aktionswahlbeschränkung. (Quadrierte Genauigkeit als Reward.)
Version 3 NACEnv	Ein Regelwerk erlernen um gute KNNs zu konstruieren. Regelwerk soll vom Trainingsverhalten abhängig sein.	KNNs werden mit Tensorflow trainiert. LevelManager reskaliert zufällig Bilddatensätze. ASM optimiert und vereinfacht. Schichten-Injektion implementiert. Mehrmaliges Trainieren des KNNs möglich. Quadrierte Genauigkeit als Reward. Rewards sind Verbesserungen der Genauigkeit oder ansonsten 0.

Tabelle 4.0.1: Ein Überblick über die drei implementierten Versionen.

Verwendet wird für den selbst entwickelten Ansatz der *On-Policy*-RL-Algorithmus *Proximal Policy Optimization*, in der zuvor vorgestellten Variante *clipped* mit

1 Die *Action-State-Machine* (ASM) wird im Unterkapitel 4.4 von Version 2 eingeführt.

einer statischen Bandbreite (erklärt im Unterkapitel 2.2.3). Die Arbeitshypothese lautet: das Konstruieren einer guten KNN ist ähnlich wie das Spielen eines Aufbau- oder Strategie-Computerspiels. Ein Einsatzfeld in dem sich PPO schon bewährt hat. In Computerspielen müssen zum Beispiel Gebäude oder Armeen iterativ aufgebaut und verbessert werden um eine möglichst hohe Punktzahl zu erlangen und um schlussendlich die Partie zu gewinnen. Bei KNNs werden stattdessen Schichten iterativ hinzugefügt und angepasst, um eine möglichst hohe Genauigkeit zu erreichen. Dies passiert in beiden Szenarien, beim Computerspiel wie auch beim konstruieren eines KNNs, auf Basis des Feedbacks von der Umgebung. Beispiele hierfür sind welchen wirtschaftlichen Ertrag ein Gebäude erwirtschaftet, oder analog, wie sich die Genauigkeit vom KNN nach der Anpassung eines Parameters ändert. Die richtigen Schlussfolgerungen anhand der Observation von der Umgebung zu ziehen, ist dabei die Voraussetzung für ein guten RL-Agenten, weil sich der Zustand einer Umgebung ständig verändert und die Observationen essenzielle Informationen dazu enthalten. In dieser Masterarbeit sind die Observationen der aktuelle Zustand vom KNN sowie Meta-Daten des Trainings-Datensatz. Dieser kann größtenteils nur indirekt untersucht werden, in dem ein KNN anhand des Datensatzes trainiert wird und das Verhalten des KNN analysiert wird. Die Datensätze können, als weitere Analogie, als die *Levels* eines Computerspiels gesehen werden.

Die Umgebung ist somit die Software die alle Funktionalitäten dem Agenten zur Verfügung stellt, um ein KNN zu konstruieren und zu trainieren. Die Observation wiederum repräsentiert ein Teil des inneren Zustands der Umgebung. Ein RL-Agent interagiert mit der Umgebung anhand der Aktionen und bekommt die Observation und ein Reward als Feedback. Weil die Umgebung und der RL-Agent voneinander unabhängig sind, können statt PPO auch andere RL-Algorithmen verwendet werden, das wurde aber aus zeitlichen Gründen nicht ausprobiert.

Die vom RL-Agenten erlernte Policy soll eine Strategie oder ein Regelwerk repräsentieren, in dem Schrittweise für einen unbekannten Datensatz ein KNN konstruiert wird. Es ahmt auf diese Weise einen Menschen nach, der ebenfalls basierend auf Regeln und bekannten *Best-Practices* iterativ ein KNN konstruiert und trainiert. Solche Regeln sind zum Beispiel:

INITIALANZAHL DER NEURONEN IN EINER SCHICHT: Die Anzahl an Neuronen der nachfolgenden Schicht ist kleiner oder gleich groß, wie die des Vorgängers.

PASSENDE LERNRATE: Es ist sinnvoll mit einer niedrigen Lernrate, z.B. $0,0001$, und dem Trainieren von 20 Epochen zu beginnen. Sinkt der Loss vom KNN, erhöhe die Lernrate um eine Zehnerpotenz. Wiederhole das Trainieren und das erhöhen der Lernrate solange bis der Loss nicht mehr sinkt. Die letzte Lernrate bei der der Loss sinkt, ist die passende Lernrate für das KNN ([21], Kapitel 3).

BILDKLASSIFIKATION: Conv-Layer funktionieren besonders gut in Kombination mit ReLU-Aktivierungsfunktionen, *Batch Normalization* und Max-Pooling ([10]).

LERNRATE UND BATCHSIZE: Statt die Lernrate zu reduzieren, kann auch die Batchsize erhöht werden ([32]).

CNN UND BATCHSIZE: Eine zu große Batchsize verschlechtert die Genauigkeit. Es ist sinnvoll probeweise mit einer Batchsize von 32 zu starten und gegebenenfalls iterativ zu erhöhen auf 64, 128 und 256 bis sich die Genauigkeit verschlechtert ([19]).

Diese Regeln sind oft nur, wenn überhaupt, empirisch bewiesen. Manchmal widersprechen sich diese Regeln gegenseitig, beziehungsweise sind nur unter bestimmten (womöglich unbekannte) Zusammenhängen, zum Beispiel in Verwendung mit einer bestimmten Aktivierungsfunktion, gültig. Entstanden sind viele solcher Regeln durch simples Ausprobieren und Beobachten. Genau auf die selbe Art soll der RL-Agent sein Regelwerk erlernen.

Im Rahmen dieser Masterarbeit wurde der entwickelte Ansatz nur auf Datensätze für Bildklassifikation trainiert. Allerdings erlaubt die Konzeption vom Ansatz auch das Training mit anderen Datensatz-Typen, womit eine noch größere Generalisierung des erlernten Regelwerks möglich ist.

4.1 ABGRENZUNG ZU ANDEREN PUBLIKATIONEN

Dass der Agent ein allgemeines Regelwerk erlernt unterscheidet diesen Ansatz von den anderen Ansätzen die zuvor in Kapitel 3 vorgestellt wurden. Während die anderen Ansätze nur eine Parametersuche für einen bestimmten Datensatz umsetzen, wird in dem hier vorgestelltem Ansatz ein allgemeines Regelwerk gesucht, dass auch auf unbekannte Datensätze angewendet werden kann. Dieser Ansatz ist deshalb stark vom Feedback seiner Umgebung abhängig, um eine sinnvolle Regel zu erlernen beziehungsweise anwenden zu können. Für die anderen Ansätze ist der Zustand der Umgebung (Konfiguration der letzten Schicht oder ähnliches) vermutlich für den Erfolg weniger ausschlaggebend und es wäre somit interessant diese Ansätze zu wiederholen, in dem die Umgebung schlicht auf die Nummer der aktuellen Schicht reduziert wird. Ähnliches wurde im Endeffekt auch mit PNAS (Unterkapitel 3.1.2) gezeigt, das statt mit RL, auf Basis von *Sequential Model-Based Optimization*, umgesetzt wurde und besser performt als die erwähnten Alternativen.

Im Hinblick darauf, dass Methoden wie *Sequential Model-Based Optimization* oder *Bayesian Optimization* scheinbar besser performen, stellt sich die Frage ob RL überhaupt für dieses Szenario die optimale Methode ist. RL-Algorithmen besitzen eher eine schlechte *sample efficiency* (siehe 6.1) und benötigen deshalb viele Beispiele an denen sie lernen können.

Bei dem hier vorgestelltem Ansatz dagegen, dient jedes Training eines KNNs auf einen Datensatz dazu, das allgemeine Regelwerk zu verbessern.

Die optimale Policy ist das eigentliche gesuchte Ergebnis und nicht die optimale KNN-Architektur für einen Datensatz. Dieser ist lediglich ein Produkt der erlernten Policy.

Ein Vorteil der anderen Ansätze gegenüber dem hier vorgestelltem Ansatz ist, dass ihre Agenten nicht vollständig zu Ende konvergieren müssen. Wenn diese eine optimale KNN-Architektur gefunden haben, müssen sie die dafür nötigen Schritte nicht erlernen. Alleine das Finden der Architektur genügt, um deren definiertes Ziel zu erreichen. Die trainierten Agenten müssen die Architektur nicht reproduzieren können. Dementgegen muss der Ansatz von dieser Masterarbeit konvergieren, um aus den Beobachtungen Regeln zu erlernen. Weil das anhand verschiedener Datensätze und KNN-Architekturen passiert, hat der Ansatz auch Überschneidungen zu den Themenfeldern *Meta-Learning* und *Transfer-Learning*.

4.2 ENTWICKLUNGSUMGEBUNG UND VORAUSSETZUNGEN

Die umgesetzten Versionen vom Ansatz sind auf CNNs beschränkt um die Komplexität der Aufgabe einzuschränken. Mithilfe anderer Datensätze und Erweiterung der technischen Fähigkeiten der Umgebung, lassen sich auch andere KNN-Typen, wie RNNs, Autoencoder etc., erlernen. Bei der Umsetzung des Ansatzes wurde Python 3.5 verwendet. Für den RL-Teil wurde das Framework *Stable Baseline*[2] genutzt, ein Fork vom bekannterem *OpenAI Baselines*[3] Framework. Der Code vom *Stable Baseline* erscheint aufgeräumter und die Dokumentation vollständiger als der von *OpenAI Baselines*. Zum Trainieren der KNNs wurde das Framework Chainer[4] oder Tensorflow[5] benutzt. Wann welches KNN-Framework genutzt wurde und warum, wird in den folgenden Unterkapiteln erklärt (siehe 4.4.4).

Um KNNs effizient und schnell zu trainieren, werden High-End-Grafikkarten benötigt. Deswegen wurden die Cloud-Dienste *Google Colaboratory*[6] (Colab) und *Google AI Platform*[7] benutzt. Auf der AI Platform standen 300$ zur Verfügung, die jedem Nutzer am Anfang kostenlos als Gutschrift zur Verfügung gestellt wird. Der Colab-Dienst dagegen kann kostenlos genutzt werden. Der Unterschied zwischen den zwei Diensten liegt darin, dass bei Colab nur ein Jupyter-Notebook zur Verfügung steht und nur eine Server-Instanz mit Grafikkarte zeitgleich genutzt werden kann. Die Hardware der Instanzen besteht dabei aus zwei CPU-Kernen und aus einer *Nvidia Tesla K80* GPU, oder der doppelt so schnellen *Nvidia Tesla T4*. Welche Grafikkarte einem zur Verfügung gestellt wird ist zufällig und kann nicht direkt beeinflusst werden. Je intensiver Colab genutzt wird, desto wahrscheinlicher ist es eine K80 zugewiesen zu bekommen. Zum Schluss der Masterarbeit wurde sogar zeitweise gar keine GPU mehr zur Verfügung gestellt, weil zwischenzeit-

2 https://stable-baselines.readthedocs.io
3 https://github.com/openai/baselines
4 https://chainer.org
5 https://tensorflow.org
6 https://colab.research.google.com
7 https://cloud.google.com/products/ai/

lich Google eine maximale Nutzungsdauer pro Benutzer eingebaut hat. Ein weiterer Nachteil von Colab ist, dass die Server-Instanz abgeschaltet wird, sobald der Browser geschlossen wird oder aber spätestens nach 12 Stunden Laufzeit. Deswegen ist ein regelmäßiges Zwischenspeichern der Resultate (Policy, Logs) auf einen externen Speicherort, wie zum Beispiel dem *Google Drive*, nötig.

Die AI Platform, die ein Dienst der *Google Cloud Platform* (GCP) ist, ist zwar kostenpflichtig, hat aber den Vorteil, dass die Anzahl der Server-Instanzen und die Ausstattung dieser Instanzen beliebig gewählt werden kann. Im Rahmen dieser Masterarbeit wurde hauptsächlich die kleinste, wählbare GPU-Instanz mit 4 CPU-Kernen und der Nvidia K80 Grafikkarte genutzt. Die Hardwareleistung dieser Instanz kann am ehesten von den umgesetzten Versionen des Ansatzes vollständig ausgeschöpft werden und ist somit am kosteneffizientesten.

Um den Trainingsfortschritt des Agenten zu verfolgen, wurde ein Logging-Modul in Python entwickelt, das Einträge in eine vorgegebene Tabelle in *Google Sheet* hinzufügt. Der Vorteil ist, dass die Logging-Daten in der Cloud gesichert und immer aktuell sind. Es müssen nicht unterschiedlich Logging-Prozeduren implementiert werden, wenn der Ansatz auf Colab, AI Platform oder lokal ausgeführt wird. Des Weiteren ist es möglich, einfache Analysen und Grafiken anhand der Log-Einträge zu erstellen, die in Echtzeit aktualisiert werden. Der Nachteil ist, dass es eine obere Schranke gibt, wie viele Interaktionen mit der API von *Google Sheet* pro Minute getätigt werden können, was ein Caching der Logging-Einträge erforderlich macht. Darüber hinaus werden alle Einträge in den Browser geladen und im Browser ausgewertet. Das führt dazu, dass die Performance schon nach wenigen Tausend Einträgen einbricht.

In dieser Masterarbeit kam der PPO-Algorithmus von *Stable Baseline*[8] zum Einsatz. Wenn nicht anders angegeben, wurden die Hyperparameter von PPO bei ihren Standardwerten belassen. Folgende Hyperparameter sind Relevant:

GAMMA = 0.99: Discount-Faktor (γ).

ENTROPY = 0.01: Entropy-Koeffizient für die Loss-Berechnung (c_E).

N-STEPS = 128: Anzahl an Aktionen bevor die Policy verbessert wird.

LERNRATE = 0.00025: Lernrate für die Backpropagation bei der Policy (α).

NOPTEPOCHS = 4: Anzahl der Epochen beim Trainieren der Policy.

CLIPRANGE = 0.2: Spannweite für Veränderungen bei der Policy (ϵ).

Die in Klammern angegebenen Variablen gleichen denen aus der Vorstellung von PPO im Unterkapitel 2.2.3.

8 https://stable-baselines.readthedocs.io/en/master/modules/ppo2.html

Die Policy und die Value-Funktionen bestehen beide aus jeweils zwei FC-Layers mit 64 Neuronen und der Aktivierungsfunktion Tanh. Diese Konfiguration wurde als Beispiel vom Framework vorgegeben. Es wurde übernommen weil es ein einfaches KNN ist, aber trotzdem mächtig genug ist, für die Aufgaben in dieser Masterarbeit.

Die der Masterarbeit beiliegenden Dateien enthalten die Protokolle von den Trainings der Agenten und alle Implementationen die in den folgenden Unterkapiteln umgesetzt worden sind. Eine Beschreibung der Ordnerstruktur befindet sich im Appendix B.

4.3 VERSION 1

In Version 1 wurden erste Erfahrungen gesammelt, wie eine sinnvolle Beschreibung der Umgebung (*Observation*) aussehen muss, damit diese effektiv vom RL-Agenten genutzt werden kann. Die Aktionen sind als numerische Dreier-Tupel implementiert, wobei der erste Wert im Tupel die Aktionsart und die anderen zwei Werte die Parameter bilden. Einem Computerspiel nachempfunden, gibt es aufsteigende Levels und jedes Level repräsentiert ein Datensatz. Erst wenn der RL-Agent eine vordefinierte Genauigkeit bei einem Level erreicht hat, wird das nächste Level verwendet. Bedingt durch den Aufbau der Version 1 wurde erwartet, dass der Agent, basierend auf der Größe eines Datums aus dem Datensatz, lediglich das Hinzufügen einer bestimmten Anzahl an Schichten erlernt.

Die Größe eines Datums vom Datensatz, dient in dieser Version zur Identifikation eines Datensatzes. Ein generalisiertes Regelwerk zum Bauen einer KNN-Architektur für ein beliebiges, unbekannte Datensatz ist in dieser Version nicht das Ziel.

Zum Trainieren der KNNs wurde das Framework Chainer verwendet.

4.3.1 *Warum Chainer?*

Um die KNNs zu trainieren, wurde nach einer Recherche für das Framework Chainer entschieden. Chainer ist im Vergleich zu anderen Frameworks, wie TensorFlow, PyTorch oder DeepLearning4J, eher weniger bekannt [3]. Entwickelt wurde Chainer vom japanischem Start-Up *Preferred Networks*[9].

Chainer führt bei der Performance in vielen Benchmarks, insbesondere wenn es um die Ausschöpfung der GPU-Leistung geht. Des Weiteren erscheint der Code von Chainer bei der Anwendung übersichtlicher und von der Struktur klarer zu sein, als bei anderen Frameworks. Ein anderer Vorteil ist, dass Chainer beim Trainieren und beim Propagieren sehr flexibel ist und sich dadurch auch exotischere Möglichkeiten, wie zum Beispiel das einbauen einer if-Anweisung in ein KNN, ergeben. Diese Flexibilität wurde aber im Rahmen dieser Masterarbeit nicht genutzt.

Gegen den Einsatz von Chainer spricht, dass es durch die seltene Nutzung des Frameworks nur eine kleine Nutzergemeinschaft existierst und damit

9 https://www.preferred-networks.jp/en/

weniger Hilfestellungen im Internet zu finden sind. Allerdings war dies im Rahmen der Masterarbeit nie ein Problem, weil die offizielle Dokumentation als Hilfestellung ausgereicht hat oder ein Problem so allgemein war, dass die Fragestellung unabhängig vom genutzten Framework gelöst werden konnte.

Nichts desto trotz muss hier aber darauf Hingewiesen werden, dass zum Beispiel TensorFlow von der Funktionalität her mächtiger ist als Chainer. TensorFlow dient nicht nur dazu KNNs zu trainieren, sondern allgemein als Machine-Learning-Framework mit dem unterschiedliche mathematische Operationen auf große Datenmengen angewendet werden können.

4.3.2 *Aufbau*

Der Agent muss in jedem *time step* eine Aktion, bestehend aus drei numerischen Werten, bestimmen. Eine Aktion hat damit die Form:

$$\text{Aktion} := (\text{Aktions-Typ}, \text{Parameter 1}, \text{Parameter 2})$$

Der erste Wert im Tupel bestimmt die Art der Aktion. Die nachfolgenden zwei Werte parametrisieren die gewählte Aktion. Nicht alle Aktionen haben zwei Parameter. Die Aktion TRAIN_EPOCH zum Beispiel, besitzt gar keine Parameter, somit werden die Werte von Parameter 1 und Parameter 2 bei der Ausführung der Aktion von der Umgebung ignoriert. Alle Werte im Aktions-Tupel sind vom Daten-Typ INTEGER. In der Version 1 sind folgende fünf Aktionen implementiert:

Aktions-Typ	Beschreibung	Parameter 1	Parameter 2
TRAIN_EPOCH	Trainiere die KNN-Architektur	-	-
ADD_BOTTOM_LAYER	Hinzufüge eine Schicht	Anzahl Neuronen	-
REMOVE_BOTTOM_LAYER	Entfernt eine Schicht	Schicht-Nr.	-
CHOOSE_LAYER	Aktiviere eine Schicht für die Modifikation	Schicht-Nr.	-
CHANGE_LAYER_PARAM	Modifiziere ein Attribut von der aktiven Schicht	Schicht-Param.	Param.-Wert

Dem Agenten ist ein statischer Wertebereich für die Aktionen vorgegeben:

$$\text{Aktion} \in ([0,4], [0, MAX_LAYER], [0,100])$$

Dabei ist MAX_LAYER ein Konfigurationsparameter und in dieser Version auf den Wert 10 gesetzt. Der statische Wertebereich führt dazu, dass der Agent auch Werte für Parameter 1 und Parameter 2 wählen kann, die ungültig sind. Somit ist es für den Agenten möglich, die Aktion $(\text{REMOVE_BOTTOM_LAYER}, 5, 0)$ zu wählen und damit zu versuchen, die fünfte Schicht in der Architektur zu löschen, selbst wenn weniger als fünf Schichten in der Architektur vorhanden sind. Solche ungültigen Aktionen werden von der Umgebung abgefangen und nicht ausgeführt, aber mit einem negativen Reward von -1 bestraft. Aktionen die nicht abgefangen werden konnten und dadurch einen

Systemfehler ausgelöst haben, werden mit einem Reward von -100 bestraft. Dadurch soll der Agent nur gültige Aktionen erlernen.

Die komplexeste Aktion bezüglich der Parametrisierung ist die Aktion CHANGE_LAYER_PARAM, mit dem eine bestehende Schicht verändert werden kann. Bei dieser Aktion bestimmt der erster Parameter, welche Eigenschaft verändert werden soll und der zweite Parameter bestimmt den neuen Wert der Eigenschaft. Zuvor muss aber mit der Aktion CHOOSE_LAYER die Schicht gewählt werden, die verändert werden soll. Folgende Aktionen zum Verändern der Eigenschaft sind vorhanden:

Eigenschaft (Param. 1)	**Beschreibung**	**Wertebereich (Param. 2)**
0: ActivationFunction	Aktivierungsfunktion	{SIGMOID, TANH, ReLU}
1: Stride	Stride vom Conv-Layer	$[0, 100]$
2: Padding	Padding vom Conv-Layer	$[0, 100]$
3: LayerType	Schicht-Typ	{Fully-Connected, Convolution}

Nach dem eine oder mehrere Aktionen vom Typ CHANGE_LAYER_PARAM ausgeführt worden sind, muss aus technischen Gründen die Aktion CHOOSE_LAYER mit dem Wert 0 folgen. Diese Aktion signalisiert der Umgebung, dass die Veränderung der Schicht abgeschlossen ist und ermöglicht der Umgebung intern die Schicht neu aufzubauen. Sollte diese Regel missachtet werden, wird beim Aufruf von TRAIN_EPOCH das Training der Architektur nicht ausgeführt und der Agent mit einem negativen Reward von -1 bestraft.

Dem Agenten steht eine ausführliche Repräsentation des Zustands der Umgebung (*Observation*) zur Verfügung. Damit und in Kombination der Rewards sollte es möglich sein, eine Policy zu erlernen die gültige und sinnvolle Aktionen vorschlägt. Der Aufbau der Observation ist in Tabelle 4.3.1 zu sehen.

Durch die Aktion TRAIN_EPOCH wird eine Episode beendet und die konstruierte KNN-Architektur trainiert. Das KNN wird 50 Epochen lang mit *MomentumSGD* trainiert. Die Genauigkeit des Validierungs-Datensatzes wird dem Agenten als Reward zurückgegeben. Nach der Episode wird die Umgebung zurückgesetzt und das aktuelle Level geladen.

Das aktuelle Level bestimmt der *LevelManager*. Die Umgebung bekommt ständig den selben Datensatz, bis die Kriterien für einen Aufstieg des Levels erfüllt sind. Als Kriterien dienen dafür die Anzahl an trainierten Architekturen im aktuellem Level und die bisher beste erreichte Genauigkeit im Level, siehe Tabelle 4.3.2.

Der *LevelManager* enthält auch eine simple Methode zur Abschätzung der Komplexität einer KNN-Architektur. Dabei wird die Anzahl Neuronen in der Architektur betrachtet:

$$np.sum(observation_state['layers - units'])/(MAX_LAYER * MAX_UNITS)$$

$$= \sum_i |\text{layer}_i| / (10 * 1000)$$

Name	Form	Wertebereich	Beschreibung
layers-units	(MAX_LAYER,1)	$[0, MAX_UNITS] \subset$ INT32	Anzahl der Neuronen je Schicht
layers-activation-f.	(MAX_LAYER,1)	ActivationFuncType$\in$ INT8	Aktivierungsfunktion je Schicht
layers-type	(MAX_LAYER,1)	LayerType $\in$ INT8	Schicht-Typ je Schicht
layers-param1	(MAX_LAYER,1)	$[0, \infty] \subset$ INT8	Stride von Conv-Layer je Schicht
layers-param2	(MAX_LAYER,1)	$[0, \infty] \subset$ INT8	Padding von Conv-Layer je Schicht
active-layer	*DISKRET*	$[0, MAX_LAYER] \subset$ INT32	Schicht die gerade manipuliert wird
layer-count	*DISKRET*	$[0, MAX_LAYER] \subset$ INT32	Anzahl an Schichten im KNN
dataset-type	*DISKRET*	$[0, 10] \subset$ INT32	Typ vom Datensatz (deaktiviert[10])
dataset-input-units	*DISKRET*	$[0, MAX_UNITS] \subset$ INT32	Größe eines Datums
dataset-output-units	*DISKRET*	$[0, MAX_UNITS] \subset$ INT32	Größe des Zielwerts
lever-target-accuracy	(1,1)	$[0, 1] \subset$ FLOAT32	Min. Genauigkeit für das nächste Level
last-accuracy	*DISKRET*	$[0, 100] \subset$ INT32	Genauigkeit vom letzten Training (deaktiviert)
complexity-cost	(1,1)	$[0, 1] \subset$ FLOAT32	Aktuelle Komplexität vom KNN-Architektur

Tabelle 4.3.1: Die Observation in Version 1.

Nr.	Datensatz	Min. Episoden	Min. Genauigkeit
0	InvertSignal	100	99,9%
1	SimpleDigits2	500	98%
2	SimpleDigits10	500	95%
3	MNIST	10	95%
4	CIFAR10	10	95%
5	CIFAR100	10	95%

Tabelle 4.3.2: Die Levels mit ihren Datensätzen und Kriterien für ein Level aufstieg. Mehr Informationen zu den Datensätzen selbst sind im Appendix A zu finde.

Die Metrik hat den Namen *Complexity-Cost* und war ursprünglich dafür gedacht, den Agenten für zu große KNN-Architekturen zu bestrafen. Allerdings wurde die Metrik für diesen Zweck nicht gebraucht, weil keine zu großen KNN-Architekturen während der Experimente entstanden sind und andere Probleme schwerwiegender waren. Stattdessen wurde die Metrik zur Verhaltens-Analyse des Agenten genutzt.

Die Umgebung ist in `envs/foo_env.py` implementiert.
Der *LevelManager* befindet in der Datei `envs/datasets_levels.py`.

4.3.3 *Evaluierung*

Die Evaluierung wurde mit den Konfigurationsparametern MAX_LAYER = 10 und MAX_UNITS = 1000 ausgeführt. Erst nach der Auswertung mehrerer Experimente ist dabei ein technischer Fehler aufgefallen: Der Agent konnte

nie eine Schicht mit mehr als 10 Neuronen wählen, weil der Wertebereich des Parameters 1 per Definition am MAX_LAYER geknüpft ist.

Die hier nun gezeigten Ergebnisse sind aber unabhängig vom erwähntem Fehler.

Bei den Trainings wurde beobachtet, dass der Agent es nicht erlernen konnte, Schichten der KNN-Architektur hinzuzufügen. Bei der deterministischen Anwendung der Policy[11] konnte sogar beobachtet werden wie nur ungültige Aktionen gewählt worden sind. Erst bei der stochastischen Anwendung der Policy, wurde mit einer geringen Wahrscheinlichkeit die Aktion TRAIN_EPOCH ausgewählt. Dies geschah aber dann zufällig, bedingt durch die stochastische Anwendung der Policy.

Stellvertretend für die Evaluation von Version 1 soll hier nun die „Evaluation 1d" genauer vorgestellt werden. Bei diesem Experiment wurden innerhalb von sieben Stunden 74.000 Episoden durchgeführt, ergo 74.000 KNN-Architekturen trainiert. Der PPO-Algorithmus erlaubt es mehrere Agenten samt Umgebung parallel auszuführen und dies wurde hier auch genutzt. Zwei Agenten mit jeweils eigenen Umgebungen und *LevelManagers* trainierten KNN-Architekturen unabhängig voneinander, benutzten und lernten aber die selbe Policy.

Bei Evaluation 1d konnten die zwei Agenten problemlos Level 0 und Level 1 bewältigen. Die Kriterien für das nächste Level wurden mit Anhieb erreicht. Dadurch dass jeder Agent eine eigene Instanz vom *LevelManager* besitzt, wurden auf diese Art im Endeffekt die Kriterien bei der Mindestanzahl an Episoden immer doppelt erreicht. Es wurden damit insgesamt 200 Episoden am InvertSignal Datensatz trainiert, statt nur 100 Episoden.

Im Level 2 sind beide Agenten stecken geblieben (Datensatz SimpleDigits10). Sie konnten nur eine Genauigkeit von maximal 79% erzielen und damit das Kriterium zum Aufstieg (Genauigkeit von mindestens 95%) nicht erreichen. Im Diagramm 4.3.1 ist zu erkennen, wie die Agenten im Level zwei stecken geblieben sind. Die unterschiedlichen Genauigkeiten kommen unter anderem auch dadurch zustande, dass die Gewichte und die Bias standardmäßig zufällig initialisiert werden.

Eine Genauigkeit von 95% im Level 2 kann leicht mit dem Hinzufügen einer weiteren Schicht erreicht werden. Allerdings lässt sich am Wert *Complexity-Cost* im Diagramm 4.3.1 und insbesondere im Diagramm 4.3.2 erkennen, dass die Agenten relativ früh aufgehört haben bei der Exploration Schichten hinzuzufügen.

Die anderen Experimente mit verschiedenen Variationen vom Entropy-Koeffizienten und Discount-Faktor ergaben keine Unterschiede im Lern-Verhalten.

4.3.4 *Schlussfolgerung*

Es konnte mit PPO keine sinnvolle Policy erlernt werden. Das könnte unter anderem daran liegen, dass einerseits die Observation eine hohe Varianz hat und zugleich schwer interpretierbar ist, weil diskrete Attribute durch Integer

11 weil PPO eine stochastische Policy trainiert, ist auch eine stochastische Anwendung möglich

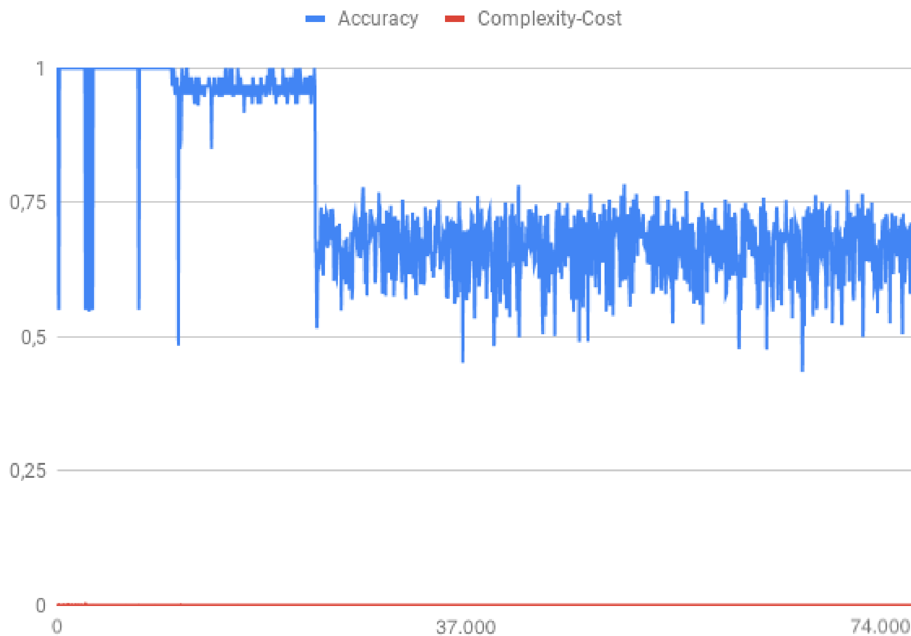


Abbildung 4.3.1: Evaluation 1d: Genauigkeit und Complexity-Cost im Zeitverlauf von 74.000 Episoden. Es ist gut zu erkennen wie die Genauigkeit mit jedem Level-Aufstieg abnimmt und im Level 2 stagniert.

repräsentiert werden (anstatt durch ein One-Hot-Vektor wie üblicherweise bei KNNs). Anderseits ist die Wahlmöglichkeit mit $5 * 11 * 101 = 5555$ verschiedenen Aktionen ziemlich groß. Dabei ist ein Großteil dieser Aktionen auch noch ungültig und führt zu einer Bestrafung des Agenten, was erklären würde warum der Agent früh aufhört zu explorieren.

Ein weiterer Grund, warum der Agent früh aufhört Schichten in die KNN-Architektur aufzunehmen ist, dass in den ersten zwei Levels die besten Ergebnisse erreicht werden können, wenn keine weiteren Schichten hinzugefügt werden. Innerhalb den ersten 1200 Episoden wird somit die Policy konditioniert, keine Schichten hinzuzufügen. Ab Level 2 würde sich das Hinzufügen einer Schicht lohnen, wurde aber bei Evaluation 1d lediglich in nur drei von 72.800 Episoden vom Agenten versucht. Es kann gesagt werden, dass die Policy auf die Level 0 und Level 1 *overfittet* worden ist und es dem Agenten sehr schwer fällt aus diesem lokalem Optimum zu entkommen.

Nachteilig ist vermutlich auch, dass die Aktionen aus optionalen Parametern bestehen. Ungenutzte Parameter werden zwar von der Umgebung ignoriert, aber nicht vom PPO, beziehungsweise vom Backpropagation-Algorithmus der PPO zum Einsatz kommt. Dieser Umstand sollte kein Hindernis sein, um eine sinnvolle Policy zu erlernen, dennoch ist es nicht klar wie stark das Training für den PPO-Algorithmus dadurch erschwert wird. Immerhin exploriert der Algorithmus dann auch über Parametern, die von der Umgebung ignoriert werden.

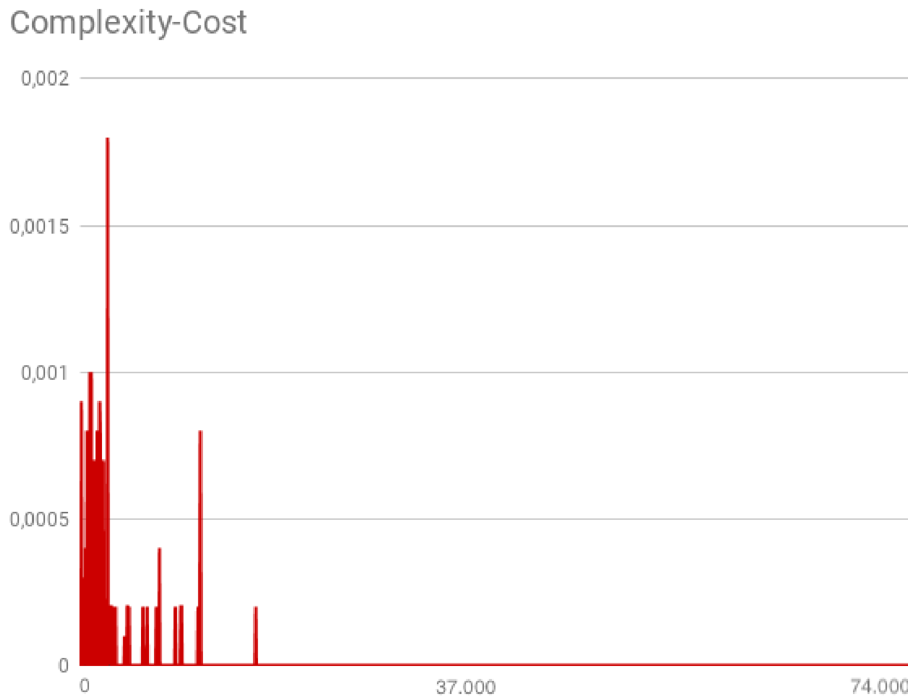


Abbildung 4.3.2: Evaluation 1d: Complexity-Cost im Zeitverlauf von 74.000 Episoden. Es ist zu erkennen, wie nach wenigen hundert Episoden alle KNN-Architekturen keine zusätzliche Schichten hinzugefügt bekommen.

4.4 VERSION 2

Basierend auf den Erfahrungen von Version 1 wurde versucht, den Wertebereich und die Observation zu verbessern, damit der PPO-Algorithmus einfacher eine Policy erlernen kann. Das Lernziel für den Agenten ist das selbe, wie in Version 1: Die Policy soll anhand der Input-Neuronen den Datensatz wiedererkennen und eine passende KNN-Architektur aufbauen. Eine Generalisierung der Policy ist in Version 2 noch nicht vorgesehen. Auch die Version 2 ist somit nur als Vorstufe zur finalen Version gedacht.

Als grundlegende Veränderung in Version 2 ist der Wechsel von den Aktionen in numerischen Dreier-Tupel zu lediglich einem numerischen Wert und die damit einhergehende dynamische Einschränkung des Wertebereichs (Unterkapitel 4.4.2) hervorzuheben. Dadurch kann der Agent nur noch gültige Aktionen auswählen.

Darüber hinaus wurde die technische Vorbereitung getroffen, um an der *AutoDL Challenge* teilzunehmen, was einen Wechsel des KNN-Frameworks von Chainer auf Tensorflow zur Folge hatte (Unterkapitel 4.4.4).

Weitere Optimierungen wurden implementiert, um das Lernen einer sinnvollen Policy zu beschleunigen. Diese werden im folgenden Unterkapitel vorgestellt.

4.4.1 *Aufbau*

Die wichtigste Neuerung ist die Veränderung der wählbaren Aktionen für den Agenten. Der Wertebereich der Aktionen besteht aus ganzzahligen Werten aus dem Bereich $[0, MAX_UNITS]$, wobei der Konfigurationsparameter MAX_UNITS die maximale Anzahl an Neuronen pro Schicht beschreibt. Normalerweise ist der Konfigurationsparameter MAX_UNITS auf 200 gesetzt, was zu 201 verschiedenen Aktionen führt und damit eine starke Reduzierung gegenüber den 5555 von Version 1 ist. Die Vereinfachung der Aktionen von Dreier-Tupels auf einen einzigen numerische Wert führt aber zu einer komplexeren Logik bei der Anwendung der Aktionen. Ähnlich wie bei den Aktionen zum Ändern von Schicht-Attributen in Version 1 (Aktionen CHOOSE_LAYER und CHANGE_LAYER_PARAM), sind nun mehrere Aktionen beziehungsweise Zeitschritte nötig, um eine Änderung an der KNN-Architektur vorzunehmen. Die Aktionen haben einen eigenen Zustandsraum und je nach Zustand hat eine gewählte Aktion eine völlig andere Bedeutung. Dieser Zustandsraum für die Aktionen hat in dieser Masterarbeit die Bezeichnung *Action-State-Machine* (ASM) und dessen Zustand ist auch ein Teil der Observation. Dieser besteht aus einem Attribut state-action der die aktuelle Operation angibt sowie einem untergeordneten Attribut state-param, welcher den Zustand/Fortschritt der aktuellen Operation angibt. Eine Darstellung der *Action-State-Machine* für Version 2 befindet sich auf der Seite 43. Der Vorteil ist nun, dass der Wertebereich für die Aktionen anhand des Zustandes der ASM beschränkt werden kann. Wie die konkrete Umsetzung dieser Beschränkung aussieht, ist in Unterkapitel 4.4.2 beschrieben.

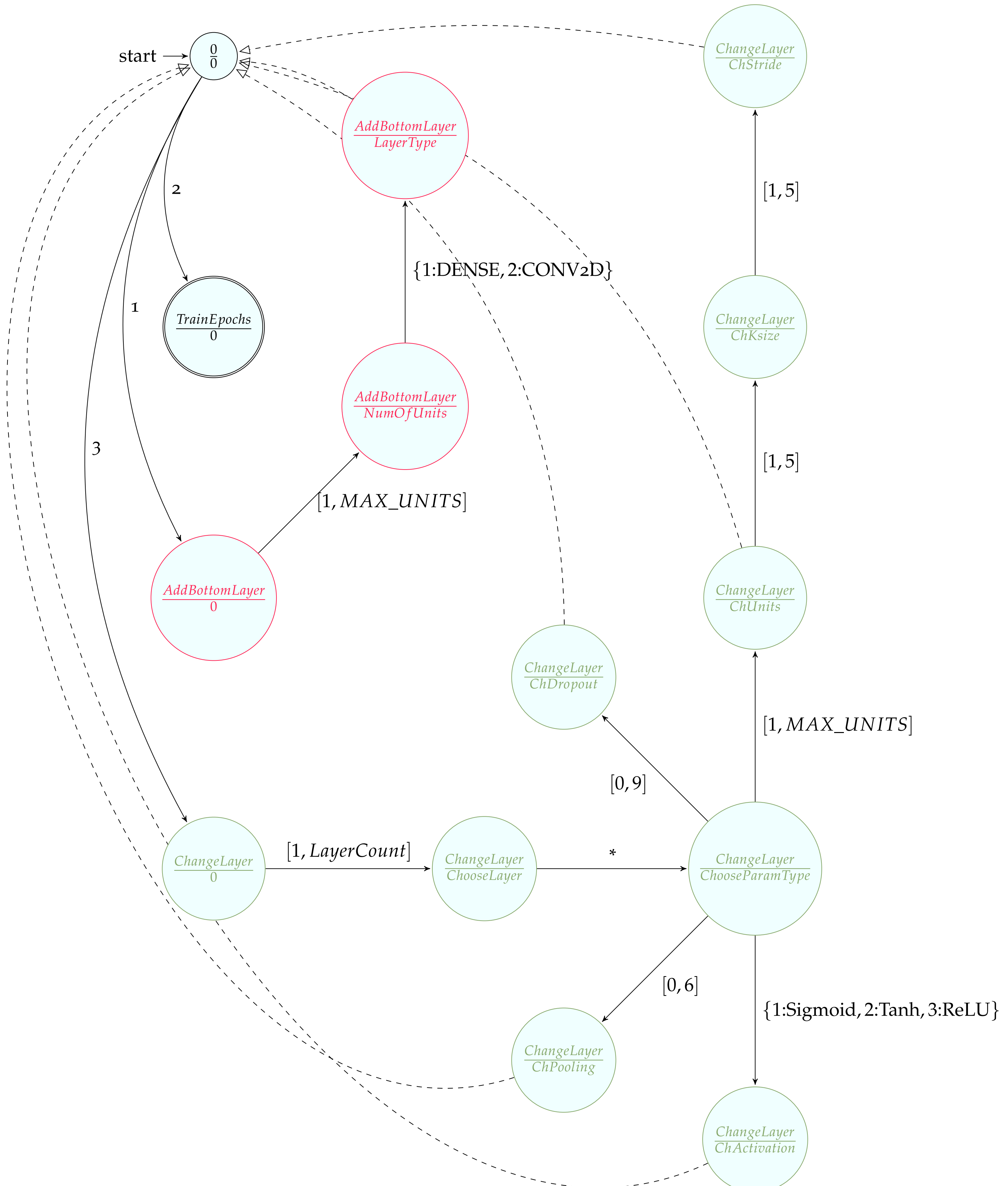


Abbildung 4.4.1: *Action-State-Machine* von Version 2.

Das Schema der Zustände entspricht: $\frac{\text{state-action}}{\text{state-param}}$. Die Zahlen/Wertebereiche auf den Pfeilen sind die Aktionen.

Aktionen mit $*$ verändern den state-param der ASM.

Gestrichelte Pfeile beenden innerhalb der selben Aktion eine Option und reseten die ASM.

Mithilfe der genannten Änderungen existieren nun keine ungenutzten Parameter beziehungsweise Aktions-Werte mehr, die von der Umgebung ignoriert werden müssen und die zuvor das Erlernen der Policy erschwert haben. Hinzu kommt, dass die zustandsabhängige Aktionswahlbeschränkung nochmals den Suchraum für den Agenten einschränkt. Für das Hinzufügen einer neuen Schicht

$$\text{ADD_BOTTOM_LAYER} \rightarrow [1, \text{MAX_UNITS}] \rightarrow \{\text{DENSE}, \text{CONV2D}\}$$

existieren nur $1 * 200 * 2 = 400$ verschiedene Möglichkeiten. Zur Verdeutlichung des Prinzips des ASMs: Folgende drei Aktionen müssen nacheinander gewählt werden, um ein FC-Layer mit 42 Neuronen der KNN-Architektur hinzuzufügen:

$$1 \rightarrow 42 \rightarrow 1$$

Lediglich die neuen ChangeLayer Aktionen können den Suchraum des Agenten auf ein sechsstelligen Bereich vergrößern, allerdings wächst dieser linear zu der Anzahl an vorhandenen Schichten in der KNN-Architektur.

Dadurch dass Aktionen wie ChangeLayer mehrere Zeitschritte benötigen und der Agent mehrere Aktionen hintereinander wählen muss, wird nun zur eindeutiger Identifizerung der Begriff *Option* eingeführt. Der Begriff ist entnommen aus [35], in der eine Option als eine Temporale-Abstraktion von einer Folge von Aktionen definiert ist. Ab Version 2 definieren nun die Zustände AddBottomLayer, ChangeLayer und TrainEpochs aus dem ASM-Attribut state-action die aktuell aktive Option. Die erste Aktion einer Option aktiviert die Option, soll bedeuten, dass der ASM-Attribut state-action gesetzt wird. Darauf folgende Aktionen parametrisieren die Option, bis die Option vollständig parametrisiert ist und die ASM zurück gesetzt wird. In der ASM von Abbildung 4.4.1 ist jede Option anhand einer eigenen Farbe (Rot, Grün, Schwarz) kenntlich gemacht.

In Version 2 wurde auch die Observation von der Umgebung verbessert. Die diskreten Attribute wurden in booleschen Vektoren der Größe n transformiert, wobei n die Anzahl an unterschiedlichen Werten eines Attributs angibt (*One Hot Encoding*). Des Weiteren sind nun die meisten numerischen Attribute auf $[0, 1]$ normalisiert. Dies soll das Konvergieren der Policy beschleunigen und ist eine typische Maßnahme bei KNNs, die mit Backpropagation trainiert werden. Weil das *Stable Baseline* Framework diese Funktionen nicht selbst anbietet, wurde diese auf konforme Art im Rahmen der Masterarbeit nachimplementiert (Datei `util/dict_helper.py`). In der folgenden Tabelle 4.4.1 wird die neue Observation dargestellt.

An der Tabelle ist auch zu erkennen, dass zwei neue Funktionen der Umgebung implementiert worden sind: *Max-Pooling* und *Dropout*. Beides wurde implementiert, um das erreichen besserer Genauigkeiten zu ermöglichen.

Durch den Wechsel auf TensorFlow können nicht mehr mehrere Agenten parallel miteinander arbeiten (Siehe Unterkapitel 4.4.4). Es wurde somit beim Trainieren der Policy nur ein Agent benutzt. Beim Trainieren der KNN-Architekturen wurde der Konfigurationsparameter MAX_EPOCH auf 500

Name	Form	Wertebereich	Beschreibung
layers-units	*N* (MAX_LAYER,1)	$[0, MAX_UNITS] \subset$ INT32	Anzahl der Neuronen je Schicht
layers-activation-f.	*N* (MAX_LAYER,1)	ActivationFuncType$\in$ INT8	Aktivierungsfunktions je Schicht
layers-type	*N* (MAX_LAYER,1)	LayerType$\in$ INT8	Schicht-Typ je Schicht
layers-param1	*N* (MAX_LAYER,1)	$[0, \infty] \subset$ INT8	Stride von Conv-Layer je Schicht
layers-param2	*N* (MAX_LAYER,1)	$[0, \infty] \subset$ INT8	Padding von Conv-Layer je Schicht
layers-dropout	(MAX_LAYER,1)	$[0, \infty] \subset$ INT8	Prozentualer Dropout je Schicht
layers-pooling	(MAX_LAYER,1)	$[0, \infty] \subset$ INT8	Breite & Höhe des Max-Pooling-Filters
active-layer	*OneHotEncoding*	$[0, MAX_LAYER] \subset$ INT32	Schicht die gerade manipuliert wird
layer-count	*OneHotEncoding*	$[0, MAX_LAYER] \subset$ INT32	Bisherige Anzahl an Schichten im KNN
dataset-type	*OneHotEncoding*	$[0, 10] \subset$ INT32	Typ vom Datensatz (deaktiviert)
dataset-dims	*OneHotEncoding*	$[0, 10] \subset$ INT32	Anzahl der Dimensionen eines Datums
dataset-input-units	(1, 1)	$[0, MAX_UNITS] \subset$ INT32	Größe eines Datums
dataset-output-units	(1, 1)	$[0, MAX_UNITS] \subset$ INT32	Größe des Zielwerts
lever-target-accuracy	(1,1)	$[0, 1] \subset$ FLOAT32	Min. Genauigkeit für das nächste Level
last-accuracy	*(1, 1)*	$[0, 100000] \subset$ INT32	Genauigkeit vom letzten Training
complexity-cost	(1,1)	$[0, 1] \subset$ FLOAT32	Aktuelle Komplexität vom KNN-Architektur
state-action	*OneHotEncoding*	$[0, 7] \subset$ INT32	Aktueller Zustand der ASM
state-param	*OneHotEncoding*	$[0, 4] \subset$ INT32	Aktueller Zustand 2. Ordnung der ASM
action-space	(2, 1)	$[0, \infty] \subset$ FLOAT32	Min. & Max. für die nächste Aktion

Tabelle 4.4.1: Die Observation in Version 2. Das N in der Spalte „Form“ gibt an, dass der Wertebereich normalisiert wird. Die farblichen Markierungen kennzeichnen Änderungen gegenüber der Version 1: grün = neu, gelb = verändert und rot = entfernt.

erhöht, aber dafür auch eine *Early-Stopping*-Methode implementiert. Wenn beim Training der KNN-Architektur zehn Epochen lang keine Verbesserung der Genauigkeit auf dem Validierungsdatensatz erreicht werden kann, dann beendet die *Early-Stopping*-Methode das Training. Der Agent bekommt die Genauigkeit der Epoche als Reward übermittelt, welches am besten war. Die maximale Epochen-Anzahl wurde bei den Experimenten nur in den seltensten Fällen erreicht.

Zusätzlich werden in Version 2 die Ergebnisse (Rewards) der KNN-Architekturen in einem Cache zwischengespeichert. Anhand der Observation werden KNN-Architekturen wiedererkannt und der Reward dem Agenten direkt übermittelt. Weil der Cache auf maximal 32 Einträge beschränkt ist, müssen Einträge auch wieder gelöscht werden. Dies passiert anhand des *Last-Recently-Used*-Verfahren (LRU), nach dem immer der am längsten nicht verwendete Eintrag gelöscht wird.

Um das in Version 1 vermutete *overfitting* auf die ersten beiden Level zu verhindern, wurden zwei weitere Level-Strategien im LevelManager integriert. Der eine ist der *Parallel Mode*, bei dem jeder Agent ein Level zugewiesen bekommt und damit ständig den selben Datensatz verwendet. Wenn also fünf Levels existieren, müssen auch fünf Agenten erstellt werden, damit alle Levels genutzt werden. Eine weitere Level-Strategie ist der *Random Mode*, in dem ein Agent nach jeder Episode zufällig ein Level zugewiesen bekommt. Diese Level-Strategie ist inspiriert vom Trainieren von KNNs mit Mini-Batches. Bei KNNs werden die Dateneinträge eines Datensatzes nach jeder Epoche erneut zufällig auf die Mini-Batches verteilt, was beim Konvergieren der KNNs hilft. Die ursprüngliche Level-Strategie aus Version 1 ist weiterhin Bestandteil des Level-Managers unter dem Namen *Ascending Mode*.

Die Umgebung ist in `envs/foo_env2.py` implementiert.

4.4.2 *Umsetzung der zustandsabhängigen Aktionswahlbeschränkung*

Bei der Umsetzung der zustandsabhängigen Aktionswahlbeschränkung wurde darauf geachtet, mit dem *Stable Baseline* Framework eine möglichst kompatible Implementierung zu programmieren. Damit bleibt die Möglichkeit bestehen, den RL-Algorithmus PPO durch einen anderen *On-Policy*-RL-Algorithmus zu ersetzen. Im Internet existierte keine vergleichbare Implementierung, um die Aufgabenstellung zu lösen. Dies machte eine eigene Implementation im Rahmen der Masterarbeit erforderlich.

Für die Entscheidungsfindung bei einer aus einem KNN bestehenden Policy, wie bei PPO, benutzt *Stable Baseline* eine Softmax-Aktivierungsfunktion in der letzten Schicht der Policy. Weil bestimmte RL-Algorithmen, wie zum Beispiel PPO, eine stochastische Policy erlernen, sind die ausgegeben Wahrscheinlichkeiten von der Softmax-Aktivierungsfunktion die Wahrscheinlichkeit, mit der eine Aktion als nächstes gewählt wird. Dabei wird der *Gumbel-Max Trick* [4] angewendet, um effizient eine zufällige Wahl anhand der Wahrscheinlichkeiten zu bestimmen.

Zum Einschränken der wählbaren Aktionen werden die Eingangssignale von der Softmax-Aktivierungsfunktion manipuliert. Jedem Eingangssignal ist eine Aktion zugeordnet und je höher der Wert des Eingangssignals ist, desto höher ist die Wahrscheinlichkeit für die Aktion gewählt zu werden. Die Eingangssignale von Aktionen die nicht gewählt werden sollen, bekommen einen hohen negativen Wert (-42) zugewiesen. Sollte dennoch zufällig eine Aktion gewählt werden, die eigentlich nicht wählbar ist, dann fängt die Umgebung diese Aktion ab und bricht die laufende Episode ab. Allerdings wurde diese Situation bei den Trainings im Rahmen der Masterarbeit nie beobachtet.

Um die Implementierung zu vereinfachen, kann lediglich ein zusammenhängender Wertebereich für die Aktionen bestimmt werden. Dieser basiert auf dem Minimum und Maximum des Wertebereichs. Diese einfache Implementierung reicht für diese Masterarbeit völlig aus. Das Minimum und Maximum wird von der *Action-State-Machine* bestimmt und ist ein Teil der Observation. Aus der Observation wird dann das Minimum und Maximum entnommen, um für die Policy die zuvor beschriebene Manipulation der Eingangssignale vorzunehmen.

Die Implementierung ist in der Datei `util/custom_policies.py` zu finden. Eine Übersicht über die Umgebung in Version 2 und dem Zusammenspiel mit der Policy ist in der Abbildung 4.4.2 zu sehen.

4.4.3 *Vorbereitung für die AutoDL Challenge*

In Version 2 wurden Vorbereitungen getroffen, um an der *AutoDL Challenge* (vorgestellt in Unterkapitel 1.3) teilzunehmen. Die wichtigste Vorbereitung war der Wechsel zu Tensorflow. Der Wechsel war nötig, weil alle Einreichungen für die Challenge auf einem Server der Veranstalter ausgeführt werden. Die technische Voraussetzung dafür ist aber, die Nutzung von Tensorflow oder PyTorch. Diese Umstellung wird im folgendem Unterkapitel 4.4.4 näher beleuchtet.

Eine weitere Hürde ist, dass keine Pakete installiert werden können und somit alle genutzten Bibliotheken und Frameworks ein Teil der Einreichung sein müssen. Dies stellte sich als Herausforderung da, weil die Bibliotheken wiederum abhängig von anderen Bibliotheken sind und so weiter. Alle Bibliotheken in die Einreichung zu kopieren, die bei einem neuen System installiert werden, war keine Lösung, weil die Abgabe mit über 700 MB zu groß war. Es durften also nur Bibliotheken in die Einreichung kopiert werden, die auf dem Server der Challenge nicht vorhanden waren. Ein manuelles zusammenstellen der benötigten Bibliotheks-Dateien war wegen der großen Anzahl zu aufwendig. Die Lösung für das Problem war, mithilfe einer virtuellen Maschine die System-Umgebung vom Server nachzustellen, die Bibliotheken dort mit den normalen Python-Tools nachzuinstallieren und die neu erstellten Dateien zu extrahieren. Dabei war das Aufsetzen der virtuellen Maschine durch die Verfügbarkeit eines digitalen Abbilds des Servers vom Challenge-Veranstalter relativ einfach.

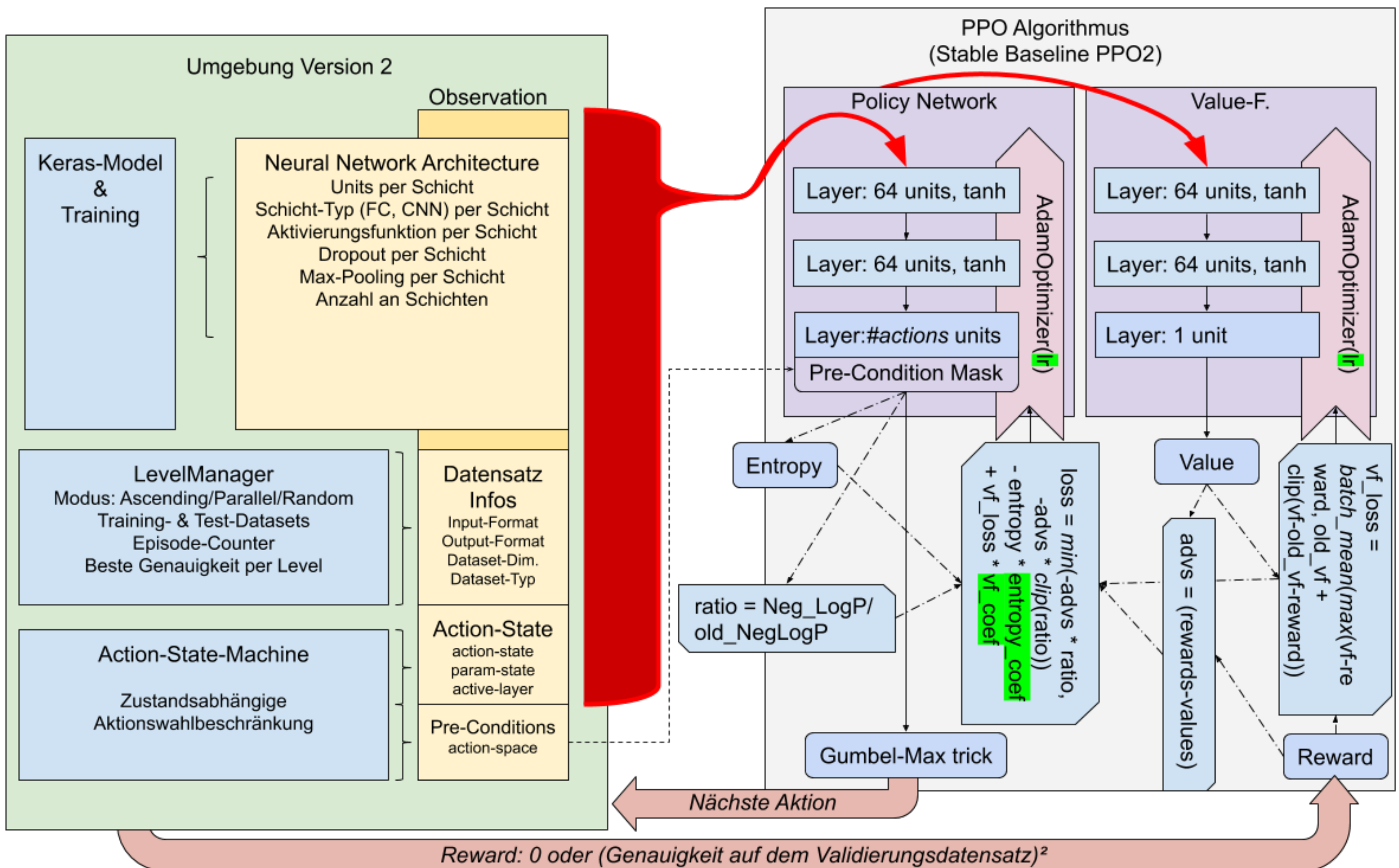


Abbildung 4.4.2: Überblick über die Umgebung und Observation in Version 2 sowie PPO. Im Kasten „PPO Algorithmus“ befindet sich im unterem Teil auch teilweise der Code für die Berechnung vom Loss zum Trainieren des Agenten. Hellgrüne Variable sind Hyperparameter. Hellgrüne Variable sind Hyperparameter. Die Pfeile deuten den Informationsfluss im System an.

Extrahiert wurden 15 Bibliotheken, die in der Laufzeitumgebung vom Server der Challenge fehlten. Nach dem manuellen Bereinigen der Bibliotheken von den Bestandteilen, die nicht gebraucht wurden, wie zum Beispiel ungenutzte RL-Algorithmen in *Stable Baselines*, waren die extrahierten Bibliotheken nur noch 30 MB groß.

Eine weitere Änderung war bezüglich des Lernziels vom Agenten nötig. Bei der Challenge kann der eingereichte Algorithmus während der Laufzeit mehrmals seine Genauigkeit überprüfen lassen. Je früher der Algorithmus eine hohe Genauigkeit erzielt, desto höher ist die resultierende Punktzahl. Siehe dazu die Beispielgrafik 4.4.3.

Um eine möglichst hohe Punktzahl zu erreichen, wurde für die Umgebung in Version 3 ein Konzept erarbeitet, das die zeitliche Komponente beachtet. Weil Version 2 noch nicht als Einreichung für die Challenge diente, wurden diese Änderungen erst in Version 3 umgesetzt.

12 https://autodl.lri.fr/competitions/3

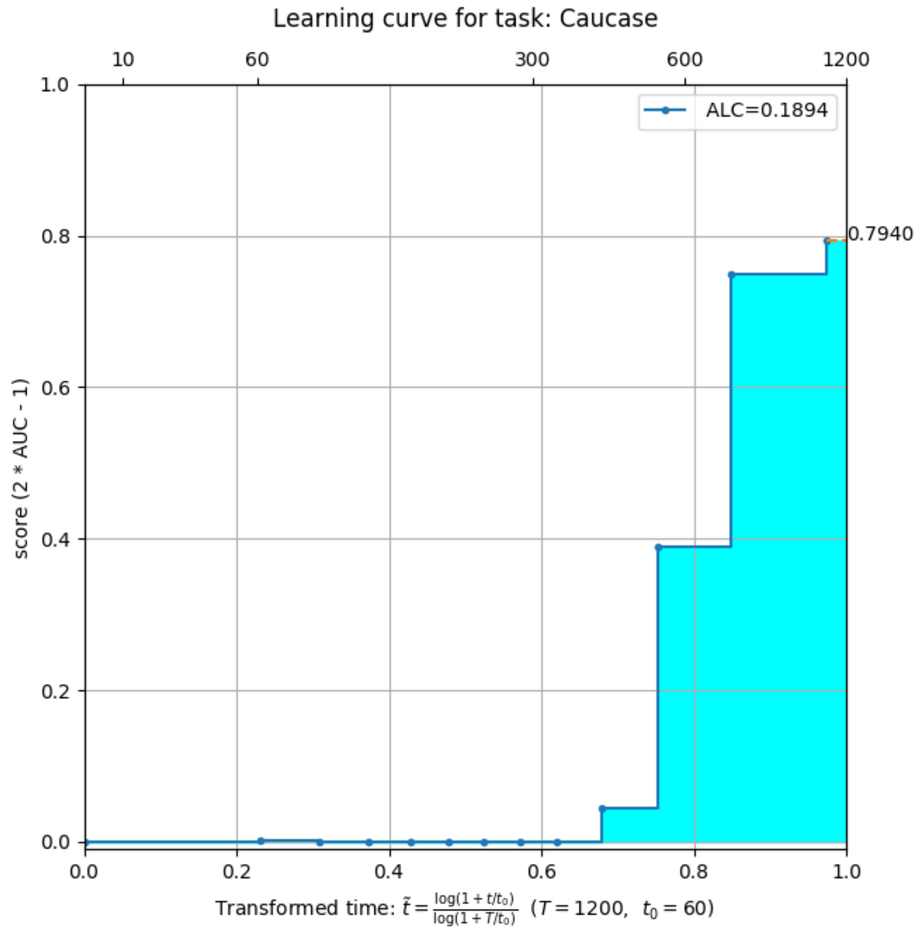


Abbildung 4.4.3: Die erreichte Genauigkeit (Y-Achse) von einem exemplarischen Algorithmus im Zeitverlauf (X-Achse). Die blaue Fläche repräsentiert die erreichte Punktzahl. Bild ist entnommen von der offiziellen Webseite der Challenge[12].

4.4.4 *Wechsel von Chainer zu TensorFlow*

Um eine Teilnahme an der AutoDL Challenge zu ermöglichen, war ein Wechsel auf TensorFlow nötig. Zwar bestand theoretisch die Möglichkeit auch Chainer über Python-Befehle zu installieren, allerdings wurde dieser Aufwand anfangs höher eingeschätzt, als ein Wechsel auf TensorFlow. Die Befürchtung war, dass insbesondere die Anbindung der Grafikkarte kompliziert sein würde und eine zeitaufwendige Installation der passenden Treiber nötig wäre. Weil keine gesonderte Installationsphase bei der AutoDL Challenge vorgesehen war, würde die verbrachte Zeit mit der Installation negative Auswirkungen auf die erreichbare Punktzahl ergeben. Trotz des Mehraufwandes wurde der Wechsel auf TensorFlow durchgeführt, weil die Möglichkeit, diese Arbeit gegen die von anderen direkt zu vergleichen, eine attraktive Ergänzung für diese Masterarbeit wäre.

Ein Vorteil beim Wechsel auf TensorFlow ist, dass die *Tensor Processing Unit* (TPU) von Google genutzt werden können. Die auf Colab kostenlos

zur Verfügung stehenden Tensor-Prozessoren beschleunigen, im Vergleich zu der Nvidia K80 Grafikkarte, das Trainieren von KNNs um das zwei- bis 20-fache. Allerdings konnten diese im Rahmen der Masterarbeit nicht genutzt werden, weil es technische Komplikationen in Verbindung mit der Arbeitsweise von der Umgebung gab.

Nach dem der Code umgeschrieben war, ist rückblickend die Anwendung von Chainer einfacher als von TensorFlow. Zum Beispiel muss bei TensorFlow das Format der Eingangssignale explizit angepasst werden, wenn ein FC-Layer (1D) auf ein Conv-Layer (2D) folgt. Dies passiert in Chainer anscheinend implizit. Siehe Code-Beispiel 1.

```
# Chainer Version
import chainer
...

model = Sequential(
    L.Convolution2D(in_channels=None, out_channels=32, ksize=3),
    F.relu,
    L.Convolution2D(in_channels=None, out_channels=64, ksize=3),
    F.relu,
    partial(F.max_pooling_2d, ksize=2)),
    L.Linear(None, 128),
    F.relu,
    L.Linear(None, num_classes),
    F.softmax,
)

# Tensorflow Version (mit Keras)
from tensorflow import keras as K
...

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                                input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten()) # Explizite Anpassung des Formats
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

Listing 1: Zwei identische CNNs. Ersteres in Verwendung mit Chainer und zweites in Verwendung mit TensorFlow und Keras.

Viel schlimmer und zeitaufwendiger waren die Memory-Management-Probleme mit TensorFlow. In Gegensatz zu Chainer gibt es bei TensorFlow ein komplexes System aus *Sessions* und *Dataflow-Graphs*, die zum Trainieren von KNNs nötig sind. In den Dataflow-Graphs sind die Rechenoperationen für das Training gespeichert und eine Session konfiguriert auf welcher Hardwa-

re ein Dataflow-Graph ausgeführt wird. Dabei stellte sich heraus, dass bei unsachgemäßer Anwendung, der Hauptspeicher und/oder der Arbeitsspeicher der Grafikkarte voll läuft (*Out Of Memory*). Dies ist ein schwer zu analysierendes Problem, weil dieser erst nach dem Trainieren von mehreren KNN-Architekturen ersichtlich wird.

Zur Lösung des Problems war es nötig zwischen den Trainings der einzelnen KNN-Architekturen, auf eine sehr genaue Trennung der Sessions und der Dataflow-Graphs zu achten. Dabei ist es wichtig, dass die Datensätze jedes Mal erneut von der Festplatte geladen werden und als Python-Objekte neu initialisiert werden. Ansonsten verwenden mehrere Graphen ein und denselben Datensatz(-Objekt), was dazu führt, dass die neuen Graphen als Unter-Graphen am alten Graphen angehängt werden. Der Datensatz verbleibt damit im Speicher und wird auf diese Weise wiederverwendet, was Ressourcen einspart. Allerdings sind die KNN-Architekturen auch ein Teil der Graphen und verbleiben somit auch im Speicher. Je mehr KNN-Architekturen vom Agenten trainiert worden sind, desto mehr veraltete KNN-Architekturen sammeln sich im Speicher an.

Ein weiteres spezielles Memory-Management-Problem tauchte auf, als die KNN-Architekturen auf Grafikkarten trainiert wurden. Wieder ergab sich das Problem, dass der Speicher der Grafikkarte voll läuft. Die Lösung war, nur noch einen Agenten, anstatt mehrere parallel, zu verwenden. TensorFlow (Version 1.14) unterstützt nicht die parallele Ausführung mehrerer Sessions, was zu diesem Memory-Management-Bug führt. Die serielle Ausführung der Agenten/Sessions löst dieses Problem.

Nach dem die technische Probleme mit TensorFlow gelöst waren, muss die ursprüngliche Aufwandseinschätzung für Chainer und TensorFlow infrage gestellt werden. Auch weil mittlerweile klar ist, dass sowohl TensorFlow als auch Chainer die Bibliothek *cuDNN* nutzen, um von der Grafik-Hardware zu abstrahieren. Treiber oder Systembibliotheken hätten vermutlich nicht installiert werden müssen, um Chainer auf dem Server von der AutoDL Challenge zu nutzen. Allerdings kann nicht zweifelsfrei behauptet werden, dass die Memory-Management-Probleme nicht auch bei Chainer in einem späterem Entwicklungsstadium aufgetaucht wären.

4.4.5 *Evaluierung*

Bei der Evaluierung von Version 2 wurden verschiedene Experimente durchgeführt mit insgesamt 110.000 Episoden, die auf Colab trainiert wurden. Allerdings muss beachtet werden, dass bei vielen dieser Episoden nicht wirklich ein KNN-Architektur trainiert worden ist, sondern bloß das fertige Ergebnis aus dem Cache benutzt wurde. Weitere Episoden wurden auf der AI Platform für die Hyperparametersuche trainiert, siehe 4.4.5.1.

Die Resultate von der Evaluation waren ähnlich zu diesen von Version 1. Der Agent erlernt weiterhin nicht die Nutzung der Fähigkeit, eine oder mehrere Schichten in die KNN-Architektur hinzuzufügen. Die Policy gab in der

deterministischen Ausführung ständig nur die Aktion „2“ vor, was die Option TrainEpochs entspricht.

Exemplarisch wird hier ein Experiment vorgestellt, dessen Log-Datei den Namen „Log Version 2 AI 4 Loops Optimized“ hat. Bei diesem Experiment wurde ein Agent 7,5 Stunden lang trainiert und hat dabei 9600 Episoden ausgeführt. Der LevelManager befand sich im Parallel Mode und weil die ersten vier Levels aktiviert waren, wurden jeweils 2400 Episoden pro Datensatz ausgeführt. Welche Datensätze es sind, können aus der Tabelle von Appendix A entnommen werden (Der Datensatz *InvertSignal* wird ab Version 2 nicht mehr genutzt, weil es zu einfach ist). Level 0 (*SimpleDigits2*) war wiedermal kein Problem für den Agenten, weil dieser Datensatz so einfach ist. Das Hinzufügen von Schichten führt fast immer zur einer negativen Wirkung auf die Genauigkeit.

Bei den Levels 1 (*SimpleDigits10*), 2 (*FashionMNIST*) und 3 (*CIFAR10*) können dagegen das Hinzufügen von Schichten in die KNN-Architektur die Genauigkeit verbessern. Bei Level 1 kann somit die Genauigkeit von 73% auf über 80% erhöht werden. Bei Level 2 ergibt sich eine Genauigkeit ohne Schichten von durchschnittlich 84,7%. Die beste Architektur in der Log-Datei erreichte eine Genauigkeit von 89%, mithilfe von einem FC-Layer mit 146 Neuronen. Allerdings wurde vom Agenten auch nur 16 KNN-Architekturen gefunden, die eine bessere Genauigkeit als 84,7% ergaben. 24 Architekturen mit einem oder mehreren FC- oder Conv-Layern ergaben eine schlechtere Genauigkeit. Durch eine zu kleine Anzahl an Neuronen oder durch eine zu hohe Komplexität konvergierten diese KNN-Architekturen nur schlecht oder überhaupt nicht.

Beim Level 3 verhielt sich die Situation ähnlich, der Unterschied bei den Genauigkeiten war nur deutlich größer. Ohne Schichten lag die Genauigkeit bei 40%, mit einem oder mehreren FC- oder Conv-Layern wurden Genauigkeiten von 60% erreicht, beim Training des Agenten.

Vermutlich konnten auch viele KNN-Architekturen auch nicht oder schlecht konvergieren, weil die Lernrate oder die Batchsize unpassend war. Auch die geringen Unterschiede in der Genauigkeit beim Level 2 könnten ein Problem darstellen, weil bei PPO die Stärke mit der eine Policy angepasst wird zum Teil vom *Advantage* abhängt.

Schlussendlich stellte sich die Frage, ob die Hyperparameter von der PPO selbst unpassend gesetzt waren. Deswegen wurden mehrere Hyperparameter-Suchen durchgeführt, siehe nächstes Unterkapitel 4.4.5.1.

4.4.5.1 *Hyperparameter Optimization*

Weil der Agent in Version 1 und Version 2 nichts oder nur monoton die Aktion „2“ erlernt hat, sollte überprüft werden, ob eine Anpassung der Hyperparameter von der PPO bei der Anpassung der Policy behilflich ist. Dafür wurde der Dienst *HyperTune* auf der AI Platform von Google genutzt. Dieser dient zur Parametersuche mithilfe der bayesianischen Optimierung.

Es wurden folgende Parameter von PPO untersucht, siehe Tabelle 4.4.2.

Parameter	Beschreibung	Suchbereich	Skala
Ent-Coef	Entropie-Koeffizient für die Loss-Berechnung	$[10^{-9}, 0.1] \subset$ DOUBLE	log
N-Steps	Anzahl an Aktionen bevor die Policy verbessert wird	$[16, 2048] \subset$ INT	linear
LR	Die Lernrate für die Backpropagation bei der Policy	$[10^{-5}, 1.0] \subset$ DOUBLE	log
Lam	Verhältnis von Bias und Varianz für die Advantage-Abschätzung	$[0.8, 1.0] \subset$ DOUBLE	linear
NOptEpochs	Anzahl der Epochen beim trainieren der Policy	$[1, 48] \subset$ INT	log
Cliprange	Spannweite für Veränderungen bei der Policy	$[0.1, 0.8] \subset$ DOUBLE	linear

Tabelle 4.4.2: PPOs Hyperparameter die bei der Parametersuche benutzt wurden. Spalte „Skala" beschreibt ob der Suchbereich logarithmisch oder linear ist.

Es wurden zwei größere Parametersuchen durchgeführt, die jeweils zehn Stunden gedauert haben. Jede Parametersuche trainierte dabei, mit unterschiedlichen Parametern, parallel drei Instanzen von Agenten. Jede Instanz lief auf einem Server mit vier CPUs und einer Nvidia K80 Grafikkarte und dauert circa zwei Stunden. 15 Experimente wurden pro Parametersuche somit durchgeführt.

Die Rewards der Umgebungen wurden für die Parametersuche anhand der Erfahrungen von Version 2 modifiziert. Statt einfach die Genauigkeit der trainierten KNN-Architektur als Reward zurück zugeben, wurde der Wert von den Genauigkeiten quadriert. Damit lag der maximal zu erreichender Reward bei $100 * 100 = 10.000$ Dadurch sollen kleine Verbesserungen bei hohen Genauigkeiten stärker belohnt werden, weil Verbesserungen von hohen Genauigkeiten schwieriger zu erreichen sind, als Verbesserungen bei einer ursprünglich niedrigen Genauigkeit.

Zur Bewertung der gewählten Parameter bekommt *HyperTune* die Summe der Rewards vom Level 2 und 3. Wobei der Reward von jedem Level, aus dem Durchschnitt von vier Wiederholungen gebildet wird, um die Varianz zu verringern. Je höher die Summe, desto besser die gewählten Parameter. Der Unterschied zwischen den zwei Parametersuchen lag darin, ob die Policy stochastisch oder deterministisch für die Entscheidungsfindung der nächsten Aktion angewendet werden soll. Dabei ergaben sich auch große Unterschiede in der Summe der Rewards bei den zwei Parametersuchen.

Bei der Parametersuche der deterministischen Ausführung ergaben alle Experimente die selben Summen an Rewards. Sie lag bei ungefähr 4400. Es konnte also kein Unterschied zwischen den gewählten Parametern gemessen werden.

Bei der Parametersuche der stochastischen Ausführung ergab sich wiederum deutliche Unterschiede bei den Summen der Rewards. Sie lag in einem

Bereich von 2400 bis 4700. Allerdings konnten diese Werte nicht mehr reproduziert werden. Es stellte sich heraus, dass die Varianz der Rewards zu hoch ist und das Feedback somit nicht brauchbar.

Die erlernten Policies von der stochastischen als auch von der deterministischen Ausführung waren von der Qualität, wie die zuvor auf Colab trainierten Policies. Womöglich wurde zu wenig Rechenzeit den einzelnen Experimenten zur Verfügung gestellt. Allerdings konnte nicht noch mehr Rechenzeit in die Parametersuche für Version 2 investiert werden, weil der Rest des Budgets für Version 3 bestimmt war.

4.4.6 *Schlussfolgerung*

Es scheint, dass die Umgebung immer noch zu komplex ist, um eine sinnvolle Policy zu erlernen. Dass die Policy unabhängig von der Observation ständig die Aktion „2“ vorhersagt, muss auch daran liegen, dass der Agent die Informationen von der ASM nicht verwerten kann.

Ein weiteres Problem ist vermutlich die Vergleichsweise geringe Anzahl an Episoden bei den Trainings. Bei den vorgestellten, verwandten Arbeiten in Kapitel 3 wurden verschiedene Architekturen im fünfstelligen Bereich trainiert. Bei den Agenten aus Kapitel 2.2, die Atari Games [20] zu spielen erlernt haben, waren mehrere Millionen *time steps* nötig. Weil in den Training-Logs zu beobachten ist, dass der Agent in Version 2 gute KNN-Architekturen findet, aber die nötigen Aktionen dazu sich nicht merken kann, wurde hier ein Problem mit der *sample efficiency* geschlussfolgert. Dieses Problem wird im Unterkapitel 6.1 intensiver behandelt.

4.5 VERSION 3 - NACENV

Dies ist die finale Version und die eigentliche Umsetzung des Ansatzes, die den Namen *Network Architecture Construction Environment* (NACEnv) trägt. Im Gegensatz zu den vorherigen Versionen, ist das Ziel dieser Version, einen Agenten zu trainieren, der die Voraussetzungen der AutoDL Challenge meistert und damit auch mit unbekannten Datensätzen arbeiten kann. Aus zeitlichen Gründen ist die NACEnv weiterhin auf CNNs und Bildklassifikationsdatensätzen beschränkt. Allerdings lässt sich die NACEnv einfach um weitere Fähigkeiten und Levels erweitern.

Die wichtigste Neuerung gegenüber der vorherigen Versionen ist, dass der Agent wiederholt eine vorgegebene Anzahl an Epochen auf einem KNN trainieren kann und zwischen den Trainings die Möglichkeit hat, die KNN-Architektur zu verändern. Ähnlich wie ein Mensch vorgehen würde, soll sich der Agent iterativ dem Optimum annähern. Ohne Vorwissen zum Datensatz, soll der Agent durch gezieltes Ausprobieren und Beobachten die Veränderungen der Genauigkeit, eine gute KNN-Architektur finden und trainieren können.

Bedingt dadurch, dass auch die Voraussetzungen geschaffen werden sollen, um an der AutoDL Challenge teilzunehmen, gibt es ab dieser Version

zwei anstatt nur ein Optimierungsziel. Vorher war nur die erzielte Genauigkeit auf dem Validierungsdatensatz relevant, ab dieser Version kommt eine zeitliche Komponente hinzu. Der Agent hat eine begrenzte Zeit, in der dieser ein KNN konstruieren und trainieren kann. Je früher dieser eine hohe Genauigkeit erreichen kann, um so besser. Siehe dazu gegebenenfalls das Unterkapitel 4.4.3 zur AutoDL Challenge.

4.5.1 *Aufbau*

Um dem Agenten ein „Zeitgefühl" antrainieren zu können, ist die Laufzeit einer Episode beschränkt. Dafür wird beim Training eine zufällige Zeitdauer aus einer Normalverteilung mit dem Erwartungswert von 6 Minuten und der Standardabweichung von 1 ermittelt. Abbildung 4.5.1 zeigt diese Verteilung. Bei der AutoDL Challenge liegt die Zeitdauer für jeden Datensatz fest bei 20 Minuten, aber zum Trainieren des Agenten wird vorerst eine kürzere Zeitdauer benutzt. Die vorhandene Zeitdauer wird für die Episode in der Observation im Attribut time-left dem Agenten transparent gemacht und nach jedem TrainEpochs aktualisiert. Auch wird dem Agenten mit dem Attribut time-last-epoch die benötigte Zeit vom letzten TrainEpochs zur Verfügung gestellt, um gegebenenfalls das Training zeitlich zu optimieren, beispielsweise durch eine Anpassung der Batchsize. TrainEpochs trainiert die KNN-Architektur mit dem Adam-Optimizer 10 Epochen lang (Konfigurationsparameter EPOCHS_PER_TRAIN) und falls innerhalb von fünf Anwendungen von TrainEpochs sich die Genauigkeit vom KNN nicht verbessert, wird die Episode beendet (*early-stopping*). Falls TrainEpochs die maximale Zeitdauer überschreitet, wird das Ergebnis vom letzten KNN-Training ausgegeben. Die zufällige Zeitdauer soll dem Agenten zum Beispiel die Möglichkeit geben, Zusammenhänge zwischen der Anzahl der Neuronen einer Schicht und die benötigte Trainingszeit der KNN-Architektur zu erlernen.

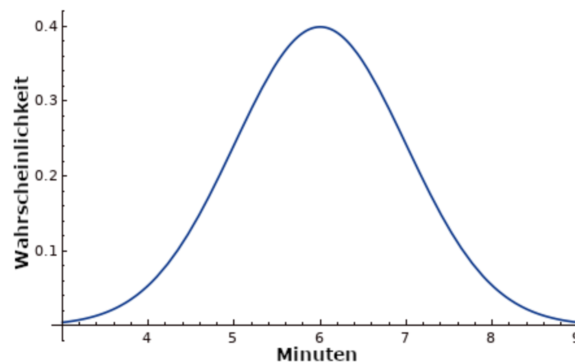


Abbildung 4.5.1: Verteilung der Zeitdauer für die Episoden mit $\mathcal{N}(6\,\text{Minuten}, 1)$. X-Achse: Zeitdauer. Y-Achse: Wahrscheinlichkeit, dass die Zeitdauer gewählt wird.

Im Gegensatz zu den Versionen 1 und 2, soll der Agent in dieser Version anhand der Attribute dataset-input-units und dataset-output-units keine Rückschlüsse mehr bilden können, welcher Datensatz verwendet wird. Deshalb wurde das Attribut dataset-output-units entfernt und die Größe eines belie-

bigen Datums (die Auflösung der Bilder) wird in jeder Episode auf eine zufällige Größe reskaliert. Dabei kann die Auflösung der Bilder eines Datensatzes von der Hälfte bis zum Doppeltem der ursprünglichen Auflösung betragen. Die Auflösung eines Datums kann der Agent anhand des Attributes dataset-input-units abschätzen. Die Datensätze sind weiterhin auf den Bereich $[0, 1]$ normalisiert und zur optimalen Ausführung auf einer GPU, im *NCHW*-Format, formatiert[13]

Um die Perfomance beim Trainieren der KNN-Architekturen zu erhöhen, wurde der Datentyp der KNNs und den Datensätze von float32 auf float16 umgestellt. Dadurch sind die Datensätze nur noch halb so groß, was zu kürzeren Ladezeiten führt. Darüber hinaus unterstützen neuere Grafikkarten, wie die Nvidia T4, Berechnungen auf Basis von float16. Dadurch werden die KNN-Architekturen in der Umgebung NACEnv doppelt so schnell trainiert und ein Wechsel von der Nvidia-Grafikkarte K80 auf die etwa doppelt so teure T4 ist sinnvoll. Der Nachteil dabei ist, dass durch die geringere Genauigkeit des Datentyps float16, beim Trainieren eines KNNs eine geringfügig schlechtere Genauigkeit erreicht wird. Dies ist aber für das Trainieren des Agenten nicht weiter wichtig, weil der relative Unterschied der Genauigkeiten bei den KNN-Architekturen bestehen bleibt.

Der Aufbau der neuen Observation kann aus der Tabelle 4.5.1 entnommen werden.

Der Aktionsraum wurde im NACEnv weiter verkleinert und optimiert, siehe Abbildung 4.5.2. So können zum Beispiel die Anzahl an Neuronen bei der Option AddLayer nur noch mit dem Faktor 16 bestimmt werden. Der Wahlbereich für die Neuronen bei der Option AddLayer liegt bei $[1, 32]$. Das bedeutet, eine Schicht kann minimal $1 * 16 = 16$ Neuronen und maximal $32 * 16 = 512$ Neuronen besitzen.

In Kombination mit den zusätzlichen Parametern der Conv-Layers, ergeben sich bei der Option AddLayer insgesamt $32 + 32 * 6 * 7 = 32 + 1344$ verschiedene Möglichkeiten eine Schicht hinzuzufügen.

Beim nachträglichem Ändern einer Schicht mit der Option ChangeLayer, existieren nur noch $10 + 3 = 13$ Möglichkeiten pro existierende Schicht in der Architektur.

Das Trainieren einer KNN-Architektur mit TrainEpochs wurde um die Möglichkeit ausgebaut, die Lernrate und die Batchsize zu parametrisieren. Das führt zu $9 * 32 = 288$ verschiedenen Möglichkeiten der Parametrisierung der Option TrainEpochs. Dabei sei angemerkt, dass die Wahl der Batchsize ebenfalls mit dem Faktor 16 verrechnet wird. Die Batchsize kann damit im Endeffekt eine maximale Größe von 512 erreichen. Die verwendete Lernrate berechnet sich aus 10^{-x}, wobei x die gewählte Aktion ist. Wegen diesen neuen Parametern und der Möglichkeit, mehrmals die Aktion TrainEpochs auf eine KNN-Architektur anzuwenden, enthält die Observation folgende

13 NCHW oder auch *channels first* ist die optimale Anordnung der Daten für die Verarbeitung auf einer GPU. NCHW entspricht einem mehrdimensionalem Array mit der Struktur [Batchsize, Channel, Height, Width]. Der *Channel* kann dabei mehrfarbig sein z.B. ein Dreier-Tupel für RGB oder Monochrom. Die Alternative ist NHWC/*channels last* und ist auf CPUs performanter.

Name	Form	Wertebereich	Beschreibung
layers-units	N(MAX_LAYER,1)	$[0, MAX_UNITS] \subset$ INT32	Anzahl der Neuronen je Schicht
layers-activation-f.	N(MAX_LAYER,1)	ActivationFuncType$\in$ INT8	Aktivierungsfunktion je Schicht
layers-type	N(MAX_LAYER,1)	LayerType$\in$ INT8	Schicht-Typ je Schicht
layers-param1	(MAX_LAYER,1)	$[0, \infty] \subset$ INT8	Stride von Conv-Layer je Schicht
layers-param2	(MAX_LAYER,1)	$[0, \infty] \subset$ INT8	Padding von Conv-Layer je Schicht
layers-ksize	N(MAX_LAYER,1)	$[2, 8] \subset$ INT8	Filter-Größe vom Conv-Layer je Schicht
layers-dropout	(MAX_LAYER,1)	$[0, 9] \subset$ INT8	Prozentualer Dropout je Schicht
layers-pooling	(MAX_LAYER,1)	$[0, 6] \subset$ INT8	Breite & Höhe des Max-Pooling-Filter
active-layer	*OneHotEncoding*	$[0, MAX_LAYER] \subset$ INT32	Schicht die gerade manipuliert wird
layer-count	N(1,1)	$[0, MAX_LAYER] \subset$ INT32	Bisherige Anzahl an Schichten im KNN
dataset-type	*OneHotEncoding*	$[0, 4] \subset$ INT32	Typ vom Datensatz (deaktiviert)
dataset-dims	*OneHotEncoding*	$[0, 5] \subset$ INT32	Anzahl der Dimensionen eines Datums
dataset-input-units	N (1,1)	$[0, MAX_UNITS] \subset$ INT32	Größe eines Datums
dataset-output-units	(1, 1)	$[0, MAX_UNITS] \subset$ INT32	Größe des Zielwerts
last-accuracy	(1,1)	$[0, 100000] \subset$ INT32	Genauigkeit vom letzten Training
best-accuracy	(1,1)	$[0, 1] \subset$ FLOAT32	Die beste erreichte Genauigkeit in der Episode
last-lr	(1,1)	$[0, 1] \subset$ FLOAT32	Lernrate vom letzten TrainEpochs
time-left	N(1,1)	$[0, 20] \subset$ FLOAT32	Übrige Zeitdauer in Minuten
time-last-epoch	N(1,1)	$[0, 20] \subset$ FLOAT32	Trainingszeit vom letzten TrainEpochs
last-batchsize	N(1,1)	$[1, 32] \subset$ FLOAT32	Batchsize vom letzten TrainEpochs
complexity-cost	(1,1)	$[0, 1] \subset$ FLOAT32	Aktuelle Komplexität vom KNN-Architektur
state-action	*OneHotEncoding*	$[0, 7] \subset$ INT32	Aktueller Zustand der ASM
state-param	*OneHotEncoding*	$[0, 4] \subset$ INT32	Aktueller Zustand 2. Ordnung der ASM
action-space	(2,1)	$[0, \infty] \subset$ FLOAT32	Min. & Max. für die nächste Aktion

Tabelle 4.5.1: Die Observation in Version 3. Das N in der Spalte „Form" gibt an, dass der Wertebereich normalisiert wird. Die farblichen Markierungen kennzeichnen Änderungen gegenüber der Version 2: grün = neu, gelb = verändert und rot = entfernt.

zusätzliche Attribute zum Training: last-lr, last-batchsize, last-accuracy und best-accuracy (Siehe Tabelle 4.5.1). Die Option TrainEpochs gibt weiterhin die quadrierte Genauigkeit als Reward zurück, allerdings nur den positiven Betrag der Verbesserung zur bisher höchsten quadrierten Genauigkeit. Eine Verschlechterung der Genauigkeiten wird mit einem neutralen Reward 0 bewertet, damit der Agent es einfacher hat, beim Trainieren der KNNs ein lokales Optimum zu überspringen.

Allgemein ist der Aktionswahlbereich deutlich verkleinert worden, vom Bereich $[0, 200]$ mit $\mathsf{MAX_UNITS} = 200$ in Version 2 auf den Bereich $[0, 32]$ mit $\mathsf{MAX_UNITS} = 512$. Dies ermöglicht ein effizienteres Trainieren der Policy.

Trotz der Vereinfachung der Optionen, ist die Umgebung komplexer geworden, da nun in einer Episode mehrfach die Option TrainEpochs verwendet werden muss und die KNN-Architekturen zwischen TrainEpochs verändert werden können.

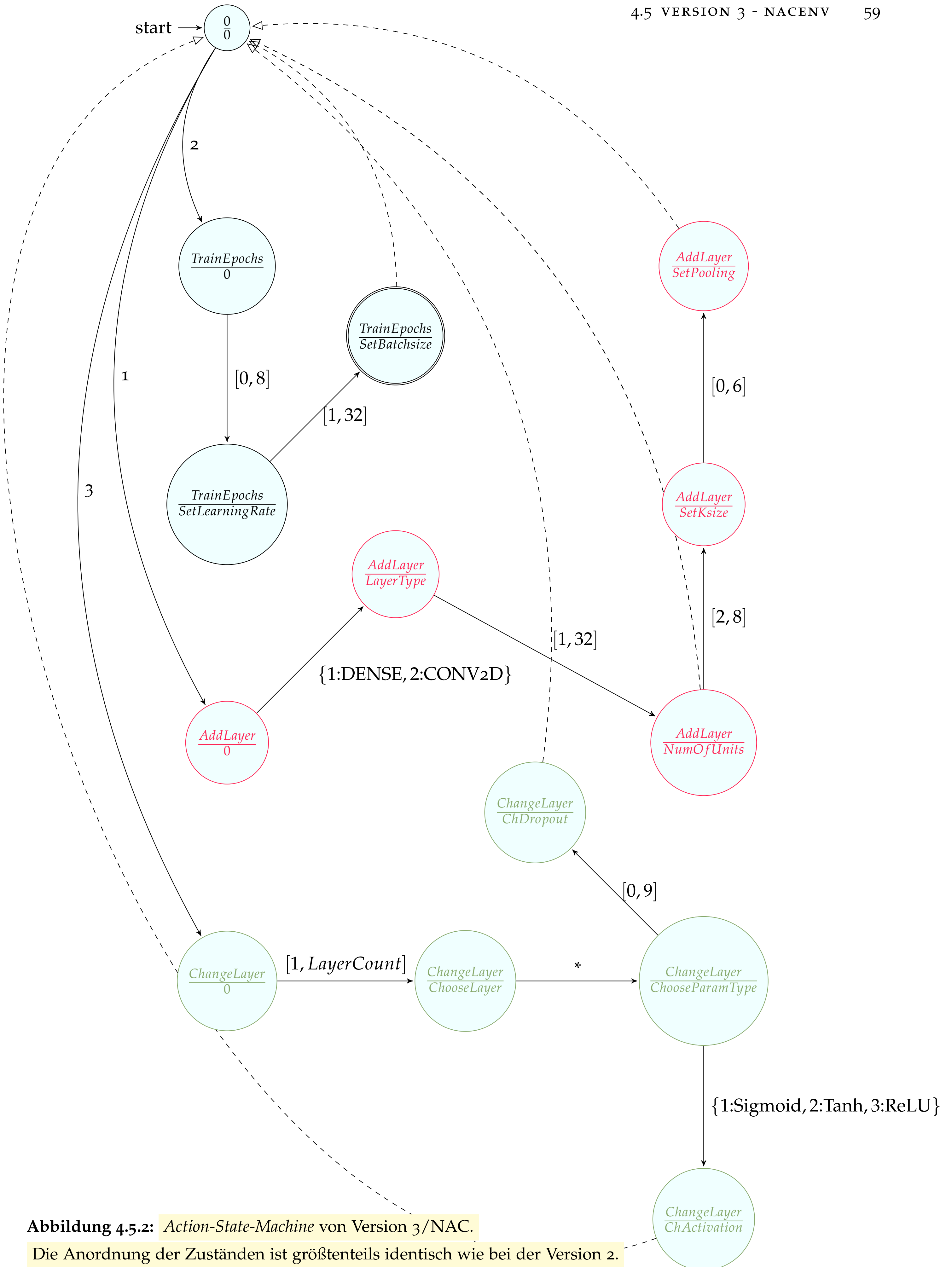


Abbildung 4.5.2: *Action-State-Machine* von Version 3/NAC.
Die Anordnung der Zuständen ist größtenteils identisch wie bei der Version 2.
Siehe dazu Seite 43 zum Vergleich.

Zum Hinzufügen neuer Schichten wurde eine Methode implementiert, die den Namen *Schichten-Injektion* bekommen hat. Die zwei Hauptmerkmale davon sind, dass eine neue Schicht nur zwischen einem Conv-Layer und einem FC-Layer hinzugefügt werden kann und dass möglichst viele trainierte Gewichte, nach dem eine neue Schicht hinzugefügt wurde, weiter genutzt werden können.

Weil eine neue Schicht immer zwischen einem Conv-Layer und einem FC-Layer hinzugefügt wird, ist es egal von welchem Typ die neue Schicht ist. Die optimale Reihenfolge, zuerst die Conv-Layers und darauffolgend die FC-Layers, wird auf diese Weise immer eingehalten. Beim Hinzufügen einer oder mehreren Schichten gehen die Gewichte für eine bestehende Schicht verloren, weil sie vom Format ungültig ist. Die Gewichte für die neue Schicht wiederum sind noch nicht trainiert, siehe Abbildung 4.5.3 zur exemplarischen Illustration der Arbeitsweise der Schichten-Injektion.

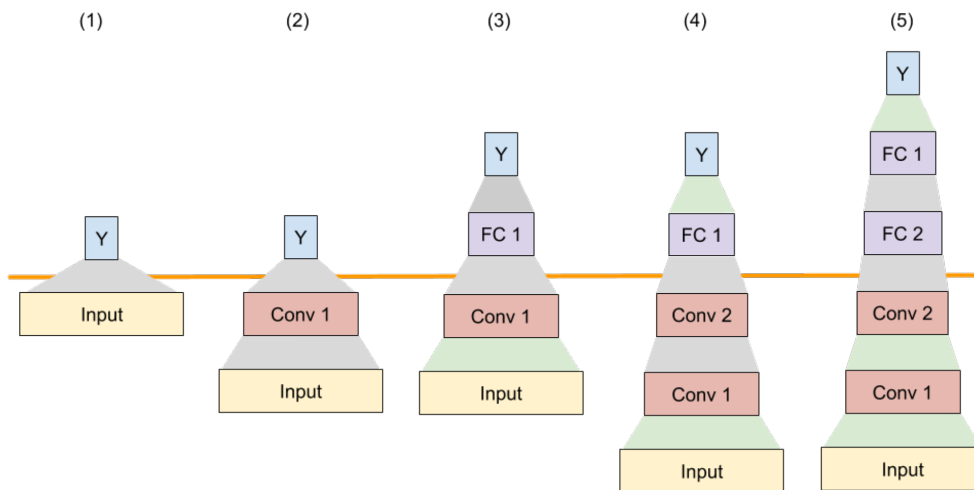


Abbildung 4.5.3: Beispielhafte Injektion von Schichten in einer leeren KNN-Architektur. Die Abbildung stellt fünf Phasen eines KNNs dar. Nach jeder Phase wird *eine* neue Schicht hinzugefügt. Vor dem Hinzufügen wird jedes mal die Option *TrainEpochs* ausgeführt, so dass alle Gewichte im KNN antrainiert sind. Die Kästen repräsentieren die Schichten und je breiter diese sind, desto mehr Neuronen enthalten diese. An der Schicht „Input“ liegt ein Datum an und die Schicht „Y“ ist der Output-Layer. „FC“ steht für einen Fully-Connected-Layer und „Conv“ für ein Convolution-Layer. Die Trapeze zwischen den Schichten deuten die dazugehörigen Gewichte an. Die Gewichte die grün gefärbt sind konnten aus der letzten Phase übernommen werden. Die horizontale, orangene Linie in der Mitte zeigt an, an welcher Stelle in der KNN-Architektur die nächste Schicht injiziert wird. Beschreibung der Phasen: (1) Leeres KNN; (2) Ein Conv-Layer wurde hinzugefügt; (3) Ein FC-Layer wurde hinzugefügt; (4) Ein weiterer Conv-Layer wurde hinzugefügt; (5) Ein weiterer FC-Layer wurde hinzugefügt.

Das kontinuierliche Hinzufügen von Schichten zwischen Conv- und FC-Layern hat auch den Vorteil, dass eher jüngere Gewichte verloren gehen, die im Vergleich zu den anderen Schichten weniger oft trainiert worden sind.

Es sollen möglichst viele Gewichte wiederverwendet werden, damit möglichst wenig Trainingszeit beim verändern der KNN-Architektur verloren

geht. Deshalb kann der Agent auch keine Schichten aus einer KNN-Architektur entfernen.

Technisch umgesetzt wurde die Schichten-Injektion in dem alle Schichten einen eindeutigen Namen haben und die dazugehörigen Gewichte anhand dieser Namen zwischengespeichert werden. Nach dem Hinzufügen einer oder mehrerer neuen Schichten in die KNN-Architektur, werden die Gewichte anhand der Namen der Schichten wiederhergestellt. Weil es für die neuen Schichten keine Gewichte mit dessen Namen existieren und damit nicht wiederhergestellt werden können, behalten diese ihre zufällige Initialwerte.

Die Schichten-Injektion selbst, wie auch das Ziel, möglichst wenige Gewichte bei einer Änderung der KNN-Architektur zu verlieren, führte dazu, dass die Option ChangeLayer in der dritten Version der Umgebung die Anzahl an Neuronen einer Schicht nicht mehr ändern kann. Es ist zugleich auch der Grund, warum die trainierten KNNs nicht mehr, wie in Version 2, zwischengespeichert werden können, um die Performance vom Training des Agenten zu erhöhen. Das Zwischenspeichern der Rewards von der Option TrainEpochs ergibt nur Sinn, wenn es eine Bindung mit der KNN-Architektur und den Gewichten der Schichten gibt. Aber insbesondere die Gewichte haben durch die Kombination von zufälliger Initialisierung und den unterschiedlichen parametrisierten TrainEpochs eine so hohe Varianz, dass es sehr unwahrscheinlich ist, dass die selbe KNN-Architektur mit exakt den selben Gewichten zweimal trainiert wird.

Weil das Verstehen der *Action-State-Machine* für den Agenten fundamental ist, sind *skip-connections* für die Observations-Attribute state-action und state-param in allen Schichten der Policy und der Value-Funktion integriert. Das bedeutet, jede Schicht im Agenten bekommt zusätzlich als Eingangssignal zum regulärem Eingangssignal der Schicht die zwei erwähnten Attribute. Damit hat jede Schicht im Agenten einen direkten Zugriff auf den Stand der ASM. In der Abbildung 4.5.4 sind die *skip-connections* im *Policy Network* und in der Value-Funktion zu sehen.

4.5.2 *Evaluierung*

Bei den Trainings der Agenten ist früh aufgefallen, dass die *skip-connections* tatsächlich dem Agenten helfen die ASM zu interpretieren. Diese Tatsache konnte auch verifiziert werden, in dem eine Gegenprobe gemacht wurde, in dem die Agenten nochmals ohne *skip-connections* trainierte wurden.

Exemplarisch wird hier die „Evaluation 5“ vorgestellt, in dem ein Agent zehn Stunden lang mit einer Nvidia T4 Grafikkarte auf der Google AI Platform trainierte. Der Modus vom LevelManager war auf *random* gesetzt, der Entropie-Koeffizient von 0.1 auf 0.3 leicht erhöht und, um das Training etwas zu vereinfachen, wurde die Trainings-Zeitdauer für die KNN-Architekturen fest auf 5 Minuten gesetzt und das zufällige Reskalieren der Bilder deaktiviert. Beim Training in Evaluation 5 wurden 190 KNN-Architekturen erprobt.

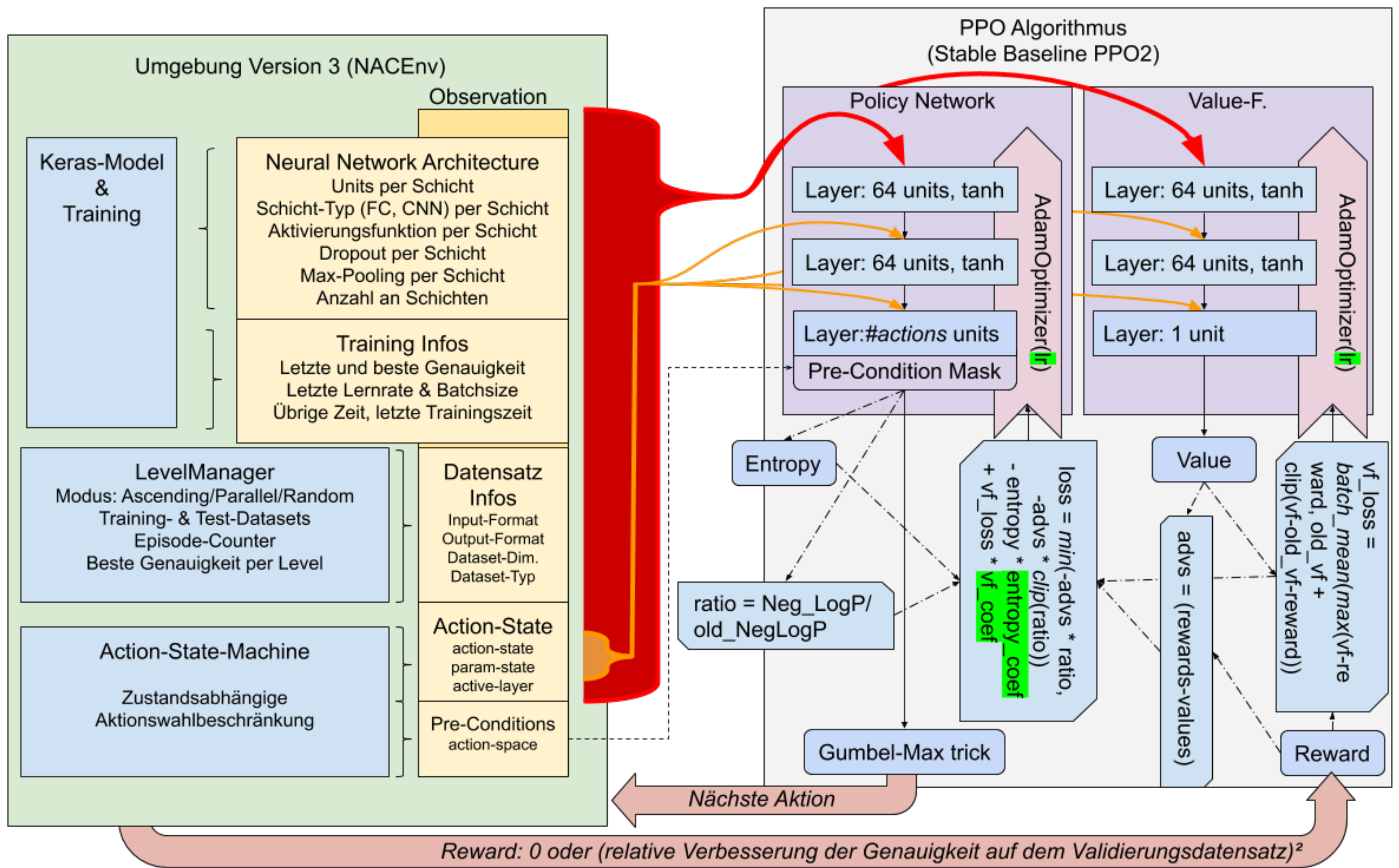


Abbildung 4.5.4: Ein Überblick über die Umgebung und Observation in Version 3 sowie vom PPO-Algorithmus. Im Kasten „PPO Algorithmus“ befindet sich im unterem Teil auch teilweise der Code für die Berechnung vom Loss zum Trainieren des Agenten. Hellgrüne Variablen sind Hyperparameter. Die Pfeile deuten den Informationsfluss im System an. Die goldenen Pfeile sind die *skip-connections* für die ASM.

Bei der deterministischen Ausführung der trainierten Policy, wird ständig folgende Folge von Aktionen ausgegeben:

$$2 \rightarrow 2 \rightarrow 5$$

Diese Aktionen haben folgende Bedeutung (Siehe auch die Abbildung 4.5.2 vom ASM):

$$\text{TrainEpochs} \rightarrow \text{Lernrate} := 0,001 \rightarrow \text{Batchsize} := 80$$

Die Policy hat also weiterhin nicht erlernt, neue Schichten hinzuzufügen. Es reagiert weiterhin nicht auf die restlichen Attributen in der Observation. Ersichtlich wird dies auch bei der Betrachtung der Aktions-Wahrscheinlichkeiten in Abbildung 4.5.5, die unabhängig vom Level und den restlichen Attribute der Observation jedes mal gleich sind.

```
[2] [[1.3463818e-19 3.5624364e-01 6.4375639e-01 1.6575317e-19 1.7834757e-19
  1.9349994e-19 1.1678887e-19 9.1593826e-20 7.1801009e-20 7.3679168e-20
  5.6761580e-20 7.1260447e-20 9.7540543e-20 6.7948971e-20 5.6555605e-20
  6.7617737e-20 8.6244888e-20 6.3985251e-20 6.8347756e-20 7.7713321e-20
  5.9637396e-20 7.1889524e-20 5.0011350e-20 5.1863307e-20 8.6349248e-20
  6.9135743e-20 6.6582630e-20 5.5331753e-20 5.1233276e-20 5.4654169e-20
  7.7630067e-20 8.1685242e-20 8.5656679e-20]]
[2] [[1.02885231e-01 1.42160535e-01 2.35480726e-01 9.75307748e-02
  1.24368966e-01 1.56849146e-01 6.50892481e-02 4.51521799e-02
  3.04831602e-02 1.92401710e-20 1.25022542e-20 1.83637925e-20
  2.98152211e-20 1.64698009e-20 1.24986781e-20 1.67740854e-20
  2.50431322e-20 1.59968507e-20 1.69523380e-20 2.21586516e-20
  1.37098950e-20 1.76637214e-20 1.02255518e-20 1.05822306e-20
  2.55166155e-20 1.79981657e-20 1.58032468e-20 1.17773523e-20
  1.10290350e-20 1.15248684e-20 2.06746298e-20 2.21567080e-20
  2.42933882e-20]]
[5] [[7.07663935e-20 1.09997883e-01 1.27604842e-01 4.79619987e-02
  8.13487470e-02 1.63479716e-01 3.75023000e-02 2.45225951e-02
  1.30921910e-02 1.67549662e-02 9.03518405e-03 1.78749003e-02
  3.46118882e-02 1.36221200e-02 9.63065214e-03 1.45303011e-02
  2.91422568e-02 1.60738043e-02 1.46907065e-02 2.70314552e-02
  1.12439422e-02 1.41734695e-02 6.94567431e-03 6.81186887e-03
  2.98406091e-02 1.85350236e-02 1.23617277e-02 8.13458953e-03
  8.42209626e-03 7.69728376e-03 1.99912265e-02 2.17731483e-02
  2.55608112e-02]]
```

Abbildung 4.5.5: Die drei Aktion $2 \rightarrow 2 \rightarrow 5$ (Geklammerte Ziffer am linken Rand) mit ihren jeweiligen Aktions-Wahrscheinlichkeiten in dem die Aktionen gewählt worden sind. Das erste Element im Array gibt die Wahrscheinlichkeit für die Aktion 0, das zweite Element die Wahrscheinlichkeit für die Aktion 1 u.s.w. an. Das Screenshot ist entnommen aus dem Jupyter-Notebook VERSION3 EVALUATION 5.IPYNB.

Anderseits ist an den Aktions-Wahrscheinlichkeiten abzulesen, dass das Hinzufügen einer Schicht (Aktion 1) mit einer Wahrscheinlichkeit von 36% bei der stochastischen Ausführung der Policy gewählt wird. Der Agent schließt es also nicht aus, dass das Hinzufügen einer Schicht eine gute Wahl sein kann.

Bei vorherigen Trainings wurde ein ähnliches Verhalten, wie bei Evaluation 5 beobachtet. Die Policies gaben ständig die Folge von Aktion in der Form von $2 \rightarrow 2 \rightarrow x$ oder $2 \rightarrow 3 \rightarrow x$, mit einem beliebig x größer als 2, aus.

Auch konnte diesmal wieder in der Log-Datei abgelesen werden, dass der Agent KNN-Architekturen mit zusätzlichen Schichten findet, die eine bessere Genauigkeit ergeben. Dass der Agent nicht erlernen konnte, weitere Schichten hinzuzufügen, könnte daran liegen, dass der Agent noch nicht fertig konvergiert hat, siehe Abbildung 4.5.6.

Evaluation 6

Als nächstes wurde versucht eine Policy zu trainieren, die auf einem einfachem Datensatz erlernt, eine Schicht hinzuzufügen. Diese Policy soll als Grundlage für weitere, komplexere Szenarien mit mehreren Levels dienen. Die Evaluation trägt den Namen „Evaluation 6“ und ist in der Log-Datei LOG VERSION 3 EVAL 6.1 protokolliert. In Evaluation 6 wurde ein Agent mit 2000 Episoden nur auf dem Level 1 trainiert. Level 1 hat einen einfachen,

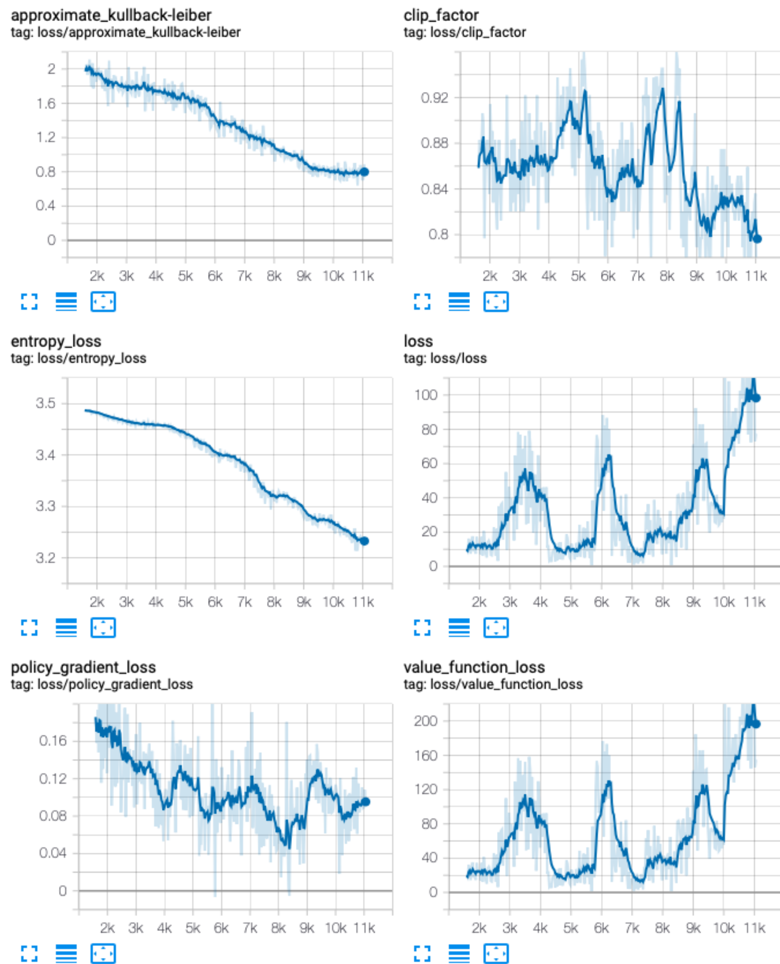


Abbildung 4.5.6: Plots vom Trainingsverlauf von Evaluation 5. Der Agent scheint noch nicht vollständig zu Ende konvergiert zu haben: Die Unterschiede zwischen der neuen und der alten Policy sind noch am sinken (Plots *Approx. Kullback-Leibler Divergenz*, *clip_factor* und *entropy_loss*). Die Value-Funktion (*value_function_loss*) scheint noch kein guter Schätzer zu sein und auch die Policy (*policy_gradient_loss*) ist noch am verändern.

kleineren Datensatz, aber gerade komplex genug, dass die Genauigkeit von einer weiteren Schicht profitieren würde. Um das Training des Agenten zu vereinfachen, ist die Umgebung auf maximal fünf Schichten für eine KNN-Architektur beschränkt, die maximale Trainingszeit wurde von 5 auf 2 Minuten verkürzt und das zufällige Reskalieren der Datensätze deaktiviert.

Nach dem Training von 1000 Episoden (9 Stunden), hat die Policy noch nicht erlernt eine Schicht hinzuzufügen. In der deterministischen Ausführung der Policy wurde ständig nur die Sequenz von Aktionen $2 \rightarrow 2 \rightarrow 1$ ausgegeben. Allerdings kann eine Veränderung der Aktions-Wahrscheinlichkeiten zwischen dem ersten Aufruf der Option TrainEpochs und den Restlichen beobachtet werden. Beim ersten Mal lag die Wahrscheinlichkeit für die Option AddLayer bei 20%:

$$[2] \quad [[9.85311696E^{-21}, 2.00505257E^{-01}, 7.99494803E^{-01}, 5.02948851E^{-20}, ..]]$$

Ab dem zweiten Mal erhöhte sich die Wahrscheinlichkeit für die Option AddLayer auf 24%:

$$[2] \quad [[2.36806284E^{-20}, 2.42524788E^{-01}, 7.57475257E^{-01}, 1.08104816E^{-19}, ..]]$$

Dieser Unterschied ist insofern bemerkenswert, weil zum ersten Mal im Rahmen dieser Masterarbeit eine Reaktion der Aktion-Wahrscheinlichkeiten auf die Observation beobachtet werden konnte.

Anhand der Log-Datei ist abzulesen, dass der Agent in 380 Episoden Schichten hinzugefügt hat. Allerdings wurden oft zu viele Schichten (mehr als eine) hinzugefügt und damit eine unterdurchschnittliche Genauigkeit erreicht.

Nach dem Training von insgesamt 2000 Episoden (insgesamt 19 Stunden Trainingszeit) hat der Agent erlernt, der KNN-Architektur exakt ein Conv-Layer hinzuzufügen und danach die Architektur zu trainieren. Die erste von der Policy vorgegebene Option ist:

$$1 \rightarrow 2 \rightarrow 6 \rightarrow 6 \rightarrow 2$$

Das entspricht folgenden Anweisungen:

$$\text{AddLayer} \rightarrow \text{LayerType} := \text{Conv2D} \rightarrow \text{NumOfUnits} := 6 \rightarrow \text{Ksize} := 6 \rightarrow \text{Pooling} := 2$$

Die Wahrscheinlichkeit, am Anfang der Episode die Option AddLayer zu wählen, lag mit 96% ziemlich hoch. Nach dem Hinzufügen des Conv-Layers trainierte der Agent die KNN-Architektur und zwar mit folgenden Optionen („$2x$" entspricht einer zweifachen Wiederholung):

$$2x(2 \rightarrow 4 \rightarrow 1), (2 \rightarrow 4 \rightarrow 13), (2 \rightarrow 4 \rightarrow 1), 6x(2 \rightarrow 4 \rightarrow 13)$$

Das bedeutet, der Agent trainiert ständig mit der selben Lernrate (4), aber variiert die Batchsize. Am Anfang gibt die Policy eine niedrige Batchsize vor (1) und gegen Ende nur noch eine höhere (13). Das mag etwas merkwürdig erscheinen, weil üblicherweise die Batchsize fix ist und die Lernrate im Verlaufe des Trainings eines KNNs gesenkt wird, aber tatsächlich ist das Verhalten der Policy die effizientere Variante. Die ICLR Publikation [32] von 2018 bestätigt das Verhalten, die Lernrate zu fixieren und die Batchsize zu vergrößern, als die effizientere Variante ein KNN zu trainieren.

Die Batchsize mit dem Parameterwert 1 entspricht einem Batch mit 16 Elementen. Das erscheint etwas klein, allerdings besteht der Trainingsdatensatz nur aus 300 Beispielen und somit ist der Batchsize gerechtfertigt.

Das Hinzufügen *eines* Conv-Layers in eine KNN-Architektur konnte der Agent anscheinend erst am Ende der 2000 Episoden erlernen. Dies kann von der Abbildung 4.5.7 abgleitet werden. Alle Aktionen die in der deterministischen Ausführung vorgegeben wurden, haben eine Wahrscheinlichkeit von über 95% gewählt zu werden. Eine Ausnahme stellen die Aktionen zur Parametrisierung von der Lernrate und dem Batchsize. Dessen Wahrscheinlich-

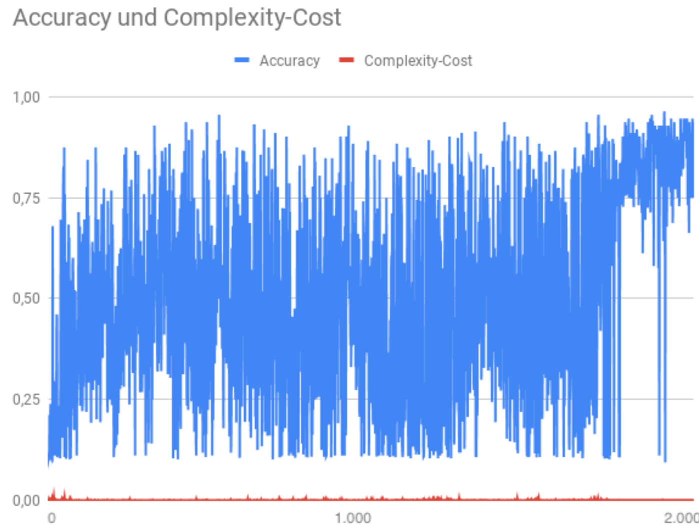


Abbildung 4.5.7: Evaluation 6: Genauigkeit und Complexity-Cost beim Trainieren auf 2000 Episoden.

keiten sind nicht ganz so groß, liegen aber alle bei über 25%. Bei diesen zwei Parameter würde also der Agent vermehrt üben, wenn der Agent weitere Episoden auf der Policy trainiert.

Bei der deterministischen Ausführung der Policy erreichen die trainierten KNNs von Level 0 eine Genauigkeit von 100% und von Level 1 eine Genauigkeit von 88%.

Um zu prüfen wie stabil der Lernverhalten vom Szenario Evaluation 6 ist, wurde das Training des Agenten wiederholt (Log Version 3 Eval 6b). Das Setting und die Hyperparameter waren identisch zu Evaluation 6. Nach 2000 Episoden konnte ein ähnlich gutes Ergebnis in Evaluation 6b erreicht werden wie in Evaluation 6. Der Unterschied besteht lediglich darin, dass Evaluation 6b erlernt hat ein FC-Layer hinzuzufügen, statt eines Conv-Layers. Das Lernverhalten kann damit als relativ stabil interpretiert werden. Dies ist nicht selbstverständlich, siehe Unterkapitel 6.1.

Evaluation 7

In Evaluation 7 wurde versucht, unter Wiederverwendung der trainierten Policy von Evaluation 6, eine Policy zu trainieren die auf unterschiedlichen Levels auch unterschiedlich reagiert. Dabei soll das Wiederverwenden der Policy aus Evaluation 6 helfen die Trainingszeit einzusparen.

Der Agent wurde mit 2500 Episoden auf den Levels 0 und 1 trainiert. Weil die Datensätze zu einfach sind, wurde erstmals in Version 3 der Discount-Faktor von $0,99$ auf $0,975$ gesenkt. Damit sollen die Rewards die früher erzielt werden höher gewichtet werden, eine Eigenschaft die auch ein Teil der Bewertung bei der AutoDL Challenge ist. Wie stark die Dämpfung des Discount-Faktors ist, wird durch die Abbildung 4.5.8 ersichtlich. Des Weite-

ren wurde erwartet, dass der Agent erlernte, auf Schichten bei Level 0 zu verzichten, weil dieses Level noch einfacher ist als Level 1.

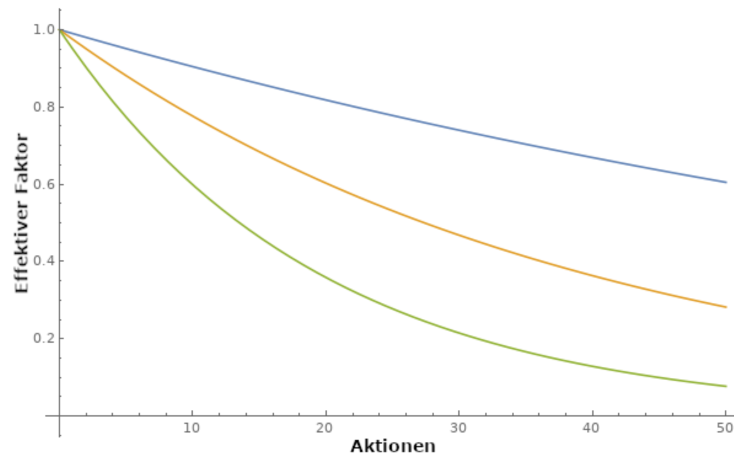


Abbildung 4.5.8: Die Reduzierung der Rewards von drei unterschiedlichen Discount-Faktoren. Folgende Discount-Faktoren sind dargestellt: blau = 0,99; gelb = 0,975; grün = 0,95. Der Reward von der x-ten Aktion wird mit dem dazugehörigen effektivem Faktor multipliziert, um den reduzierten Reward zu bekommen.

Es wurde der Wert $0,975$ für den Discount-Faktor genutzt, weil die Policy von Evaluation 6 beim Level 1 36 Aktionen benötigt um eine Episode abzuschließen. Bei einem Discount-Faktor von $0,975$ ist der Reward von der 36-ten Aktion um ungefähr 50% reduziert.

An dieser Stelle sollte angemerkt werden, dass der Discount-Faktor im NACEnv eigentlich von der vergangenen Zeit abhängen sollte und nicht von der Anzahl der Aktionen. Dies würde eher dem Szenario der AutoDL Challenge entsprechen. Allerdings wurde dies aus Zeitmangel in der Masterarbeit nicht umgesetzt.

Die Ergebnisse von Evaluation 7 zeigen, dass der Agent, Level 0 und Level 1 unterschiedlich zu behandeln, nicht erlernen konnte. Das liegt vermutlich daran, dass Level 0 und Level 1 zu ähnlich sind und es dadurch für den Agenten einfacher ist, eine Strategie zu finden die für beide Levels besonders gut funktioniert.

Allerdings hat sich die Policy in soweit verändert, dass nun statt eines Conv-Layers ein FC-Layer vom Agenten verwendet wird. Dies steht vermutlich mit dem Discount-Faktor in Zusammenhang, weil ein FC-Layer zwei Parameter weniger besitzt als ein Conv-Layer. Durch die Reduzierung der Anzahl an Aktionen, kann der Agent einen höheren Return erzielen.

Evaluation 8

Weil Level 0 und 1 zu ähnlich sind, wurde beim Training von Evaluation 8 noch das Level 2 mitverwendet. Es wurde die selbe Konfiguration wie in der Evaluation 7 angewendet. Aber auch bei der Evaluation 8 konnte der Agent nach 1600 Episoden nicht erlernen, die Levels unterschiedlich zu behandeln. Das könnte daran liegen, dass 1600 Episoden zu wenig sind.

Evaluation 9

In der letzten Evaluation „Evaluation 9“ wurden alle Funktionen von NACEnv, beziehungsweise alle Konfigurationsparameter mit ihren Standardwerten genutzt. Die Datensätze wurden zufällig skaliert und die Trainingszeit aus einer Normalverteilung zufällig gewählt, siehe Beschreibung von NACEnv im vorherigen Unterkapitel 4.5.1. Es wurden alle Levels für das Experiment verwendet.

Um das Training des NACEnv zu beschleunigen, wurde die vortrainierte Policy aus Evaluation 6 verwendet. Das Training ist in der Datei LOG VERSION 3 EVAL 9 protokolliert, umfasst 900 Episoden und dauerte 62 Stunden. Benutzt wurde für das Training die schnellere Nvidia T4 Grafikkarte.

Bei der Evaluation der trainierten Policy zeigte sich, dass zum Teil unlogische Aktionen bestimmt worden sind. Bei allen Levels hinzufügte der Agent zu Beginn einen Conv-Layer mit folgender Sequenz hinzu:

$$1 \rightarrow 2 \rightarrow 6 \rightarrow 6 \rightarrow 2$$

Das entspricht der selben Parametrisierung wie die in Evaluation 6. Danach trainiert der Agent mit TrainEpochs ein- oder mehrmals das KNN:

$$2 \rightarrow 4 \rightarrow 6 \rightarrow 10$$

Bei Level 0 ist dabei die Besonderheit zu beobachten, dass nach diesen Sequenzen zusätzlich neun FC-Layer hinzugefügt wurden und danach weiter trainiert wurde. Das Vorgehen ist kontraproduktiv und verschlechtert die Genauigkeit des KNNs. Der Agent beginnt aber womöglich zu erlernen, dass es Sinn ergeben kann, auf die erzielte Genauigkeit beim TrainEpochs mit einem Verändern des KNNs zu reagieren. Ein Verhalten das eigentlich vom Agent erwartet wird.

Bei den Levels 1, 2 und 3 wird nur ein Conv-Layer verwendet und die resultierende Genauigkeit ist akzeptabel. Allerdings hat diese Strategie immer noch eine starke Ähnlichkeit zu der ursprünglichen Strategie von Evaluation 6.

Die selbe Strategie verwendet der Agent auch für die Levels 4 und 5. Weil diese Strategie, und insbesondere die Parameter für die Option TrainEpochs, unpassend sind, ist die resultierende Genauigkeit des KNNs sehr schlecht.

Wegen der hohen Komplexität des Experiments und des relativ kurzen Trainings mit 900 Episoden, kann davon ausgegangen werden, dass die Policy nicht zu Ende konvergiert ist. Auch die Trainings-Plots von Evaluation 9, zu sehen in Appendix C, legen den Schluss nahe. Ein längeres Training war nicht möglich, weil kein Guthaben mehr für die AI Platform übrig war und die Bearbeitungszeit der Masterarbeit sich dem Ende nahte.

Parallel wurde die Evaluation 9b durchgeführt, das sich zu Evaluation 9 darin unterscheidet, dass ein Zwischenstand der Policy von Evaluation 6 benutzt wurde. Es ist der Zwischenstand der Policy nach 1000 Episoden und damit noch nicht vollständig konvergiert. Die finale Policy von Evaluation 6 wurde an insgesamt 2000 Episoden trainiert.

Die Hoffnung war, dass eine nicht vollständige konvergierte Policy es dem Agenten vereinfacht, eine passende Policy für das neue Szenario in der Evaluation 9/9b zu erlernen. Der Agent sollte es einfacher haben, aus dem lokalem Optimum der vortrainierten Policy zu entkommen. Dies konnte allerdings nach dem Training mit 500 Episoden nicht beobachtet werden. Für weitere Episoden fehlten die Zeit und die Rechenkapazitäten.

Die Schlussfolgerungen aus den Evaluationsergebnissen von NACEnv sind ein Teil vom Fazit, siehe Kapitel 6.2.

5 MÖGLICHKEITEN ZUR FORTFÜHRUNG

Dieses Kapitel ist den Ideen gewidmet, die während der Bearbeitung der Masterarbeit entstanden sind, aber aus zeitlichen Gründen nicht umgesetzt worden sind. Trivial und kein Bestandteil dieses Kapitels sind Ideen, die lediglich darauf beruhen die Umgebung um Fähigkeiten, wie zum Beispiel der Nutzung von weiteren Aktivierungsfunktionen, zu erweitern. Die Ideen in diesem Kapitel beschäftigen sich mit der grundlegenden Frage, wie das Lernverhalten des Agenten optimiert werden kann.

5.1 ERSTE IDEE: PRETRAIN MIT SMBO

RL-Algorithmen generieren beim Lernen anhand der Umgebung ihre Beispiele zufällig. Das führt dazu, dass sich Beispiele wiederholen und dass bei der Generierung keine Priorisierung statt findet. Es wird also vom Agenten nicht immer die KNN-Architektur konstruiert, die das höchste Potential hat die bisher beste Genauigkeit zu übertreffen. Dies bedeutet auch, selbst wenn sich ein generiertes Beispiel als besonders gut erweist, dann wird nicht zwangsweise als Nächstes ein ähnliches Beispiel generiert.

In der Publikation von PNAS (Unterkapitel 3.1.2) wird gezeigt, wie mithilfe von *Sequential Model-Based Optimization* (SMBO) die Beispiele fünfmal effizienter verwertet beziehungsweise generiert werden. Die Idee wäre nun, eine Policy *supervised* vorzutrainieren mithilfe von Daten, die aus einem SMBO-Prozess entstehen. Wenn die Policy vortrainiert ist, kann anschließen mit einem On-Policy-RL-Algorithmus weiter trainiert und die Policy optimiert werden. Der Vorteil bei diesem Szenario ist, dass die aufwendige Generierung der Beispiele, also das Konstruieren und das Trainieren der KNN-Architekturen, am Anfang mit SMBO umgesetzt wird und die daraus erzeugten Resultate (Observation + Rewards) gespeichert werden. Mit diesen offline Resultaten kann dann effizient (*supervised*) die Policy trainiert werden und dies kann gegebenenfalls mehrmals passieren, um zum Beispiel verschiedene Hyperparameter auszuprobieren.

Im Gegensatz zu PNAS ist die Umgebung von NACEnv allerdings komplexer. Insbesondere weil bei NACEnv der Agent iterativ durch das Trainieren der KNN-Architektur herausfinden muss, welche KNN-Architektur geeignet ist. Dieses iterative Vorgehen müsste beim SMBO-Prozess simuliert werden, damit die Policy sinnvoll trainiert werden kann. Zum Beispiel in dem im SMBO-Prozess nach jeder Änderung der KNN-Architektur pauschal einmal die Option TrainEpochs ausführt.

5.2 zweite idee: aussagekräftigere observation für das training der knns

Das einzige Feedback bezüglich des Lernverhaltens bei der Nutzung der Option TrainEpochs sind die Observations-Attribute last-accuracy und best-accuracy. Womöglich sind auch die Attribute last-lr, time-last-epoch, last-batchsize und complexity-cost bei der Einschätzung des Lernverhaltens behilflich. Weil der Agent so gut wie keine Informationen über den Datensatz besitzt, ist es um so wichtiger, Informationen aus der Option TrainEpochs zur Verfügung zu stellen, weil es die einzige Methode ist mit der der Agent (indirekt) den Datensatz erforschen kann.

Die Observation könnte zur Verbesserung um die statistischen Informationen von den Gewichten der Schichten ergänzt werden. Das könnten die Werte der Gewichte oder die Werte der Gradienten der Gewichte aus dem letzten Training sein. Damit die Observation weiterhin eine feste Dimension hat und nicht zu viele neue Attribute hinzugefügt werden müssen, können die erwähnten Werte für jede Schicht statisch zusammengefasst werden. Es könnte zum Beispiel je Schicht der Mittelwert, die Varianz oder mehrere Quantile berechnet werden.

5.3 dritte idee: schichten mithilfe von konkatenation vergrössern

Die vorgestellte Version von NACEnv hat den Nachteil, dass die Anzahl an Neuronen je Schicht beim Hinzufügen der Schicht bestimmt werden müssen und nicht nachträglich geändert werden können. Sollte die Anzahl an Neuronen zu klein sein, hat das negative Effekte auf die Genauigkeit.

Um dieses Problem zu lösen, könnte eine weitere Option in die Umgebung implementiert werden, die dem Agenten erlaubt eine neue Schicht hinzuzufügen und diese an einer bestehenden Ziel-Schicht „seitlich“ anzubinden. Die neue Schicht würde die selben Eingangssignale wie die der Ziel-Schicht bekommen. Die Ausgangssignale der neuen Schicht sowie die der Ziel-Schicht werden miteinander konkateniert und der nachfolgenden Schicht übergeben. Dies kann auf eine Weise umgesetzt werden, in dem bestehende Gewichte des KNNs nicht verloren gehen. Damit würde der grundlegende Gedanke der Schichten-Injektion, möglichst wenig trainierte Gewichte bei einer Veränderung der KNN-Architektur zu verlieren, bewahrt bleiben.

Das Potential dieser Erweiterung könnte in Verbindung mit der zweiten Idee noch besser genutzt werden, weil dann der Agent womöglich besser einschätzen könnte, ob eine bestimmte Schicht zu wenige Neuronen besitzt. An diesem Punkt angekommen, wäre es auch eine Überlegung wert, eine Option zu implementieren die konkatenierte Schichten wieder löschen kann. Denn nach der Publikation [5] besitzen große KNNs bedingt durch zufällig schlecht gewählte Initialwerte (*lottery ticket hypothesis*) zum Teil auch Gewich-

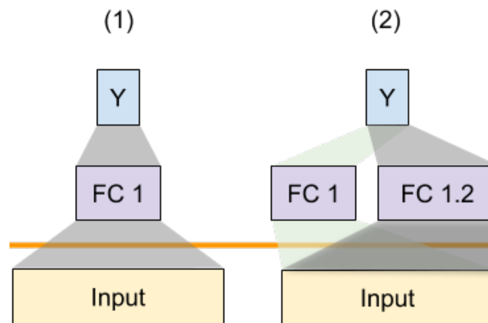


Abbildung 5.3.1: Beispielbild für das Konkatenieren zweier Schichten. In der zweiten Phase wurde der FC-Layer 1.2 zu dem FC-Layer 1 hinzugefügt. Beispielbild ist angelehnt an Abbildung 4.5.3.

te die „Tod“ sind. Diese könnten dann vom Agenten gezielt entfernt werden und damit die Performance des KNNs erhöhen.

5.4 VIERTE IDEE: NUMERISCHE AKTIONEN

Alle umgesetzten Version in dieser Masterarbeit haben einen kategoriellen Aktionswahlbereich. Allerdings gibt es Aktionen beziehungsweise Parameter von Optionen, die von Natur aus vom numerischem Typ sind. Zum Beispiel ist der Parameter Batchsize vom numerischem Typ und es ist anzunehmen, dass zwei nah einander liegende Werte ähnliche Ergebnisse beim Trainieren liefern und zwei sehr unterschiedliche Werte wiederum unterschiedliche Ergebnisse. Wenn angenommen werden kann, dass es nur einen optimalen Wert für einen Parameter zu einem bestimmten Stand der Observation gibt, dann kann der kategorielle Aktionswahlbereich durch einen auf einer Normalverteilung basierenden Aktionswahlbereich ersetzt werden. Beim Trainieren der Policy würde nicht die Wahrscheinlichkeit einer Aktion verändert werden, sondern der Erwartungswert und die Varianz der zugrundeliegenden Normalverteilung. Die Normalverteilung würde diskretisiert werden und die Wahrscheinlichkeitsdichte die diskreten Elemente wäre die Wahrscheinlichkeit für eine Aktion. Dies alles sollte dazu führen, dass die Policy schneller konvergiert, da die Verteilung die Wahrscheinlichkeiten der numerischen Parameter besser repräsentieren kann.

Dies könnte als direkter Bestandteil der KNN von der Policy mit Tensorflow implementiert werden, so dass weiterhin das Trainieren mit Backpropagation funktionieren würde. Dafür müsste die beschriebene diskretisierte Normalverteilung direkt vor dem Softmax-Layer eingesetzt werden und eine Schicht mit zwei Neuronen würde den Erwartungswert und die Varianz repräsentieren. Damit die Wahrscheinlichkeiten für die Aktionen sowohl kategoriell als auch über die diskretisierte Normalverteilung bestimmt werden können, kann mit dem Tensorflow-Operator TF.COND() zwischen den zwei Varianten in der Policy, abhängig vom Stand des ASMs, umgeschaltet werden.

6 ZUSAMMENFASSUNG

Im folgenden Unterkapitel 6.1 werden allgemein die Schwierigkeiten von RL-Algorithmen beschrieben, die zum großen Teil auch Auswirkungen auf die Bearbeitung der Masterarbeit hatten. In dem darauf folgenden Unterkapitel 6.2 wird ein Fazit über die erreichten Resultate der Masterarbeit gezogen.

6.1 SCHWIERIGKEITEN BEI REINFORCEMENT LEARNING

Bei Reinforcement Learning bestehen viele Schwierigkeiten, um ein funktionierenden Agenten zu trainieren. Bemerkbar macht sich das insbesondere dann, wenn reale Probleme gelöst werden müssen und keine akademischen Beispielaufgaben. Vieler dieser Schwierigkeiten sind Probleme, die Grundsätzlich für alle Methoden des maschinellen Lernens gelten. Einige Probleme kommen nur in Verbindung mit Reinforcement Learning oder Deep Reinforcement Learning vor. Die folgenden Abschnitte sind weitestgehend Zusammenfassungen eines Beitrags von Alex Irpan, RL-Forscher bei Google Brain, aus dem Jahr 2018 [13]. Trotz einer positiven Haltung zum RL und die Erforschung dessen, bezichtigt er der Technologie aktuell noch keinen besonders hohen Reifegrad zu besitzen (Zitat: „Unfortunately, it doesn't really work yet.“ [13]).

RL besitzt grundlegend die selben Schwierigkeiten, wie alle Verfahren für maschinelles Lernen. Die Hyperparameter müssen richtig gesetzt werden und davon womöglich sehr viele, ohne oft vorher zu Wissen, was der „richtige Wert“ für die Hyperparameter ist. Dies ist ein Problem das auch besonders *Deep Learning* mit KNNs betrifft und diese Masterarbeit mit NACEnv zu mindestens zum Teil lösen sollte. Darüber hinaus stellt sich oft die Frage, ob der verwendete Datensatz, oder die Observation bei RL, die nötigen Informationen überhaupt enthält, damit das genutzte ML-Verfahren ein funktionierendes Modell erlernen kann. Schlussendlich kommt die Schwierigkeit dazu, dass eine fehlerhafte Programmierung der Algorithmen oft nicht vom Compiler des Codes entdeckt werden kann, was sich durch schlechtes oder Nicht-Konvergieren erkenntlich macht. Diese Fehlerbeschreibung wiederum gilt für alle zuvor erwähnten Schwierigkeiten, was die Identifizierung der Ursache eines berstenden Problems erschwert. Die folgende Abbildung 6.1.1 des KI-Wissenschaftlers Zayd Enam vom *Stanford AI Lab* versucht zu verdeutlichen, dass der Raum an Problemen mehrdimensional ist und für ein gut konvergierendes Verfahren *alle* Schwierigkeiten optimal gelöst werden müssen.

Durch den Einsatz von Reinforcement Learning kommen weitere Problemfelder hinzu. So besitzen heutzutage die meisten RL-Algorithmen eine nied-

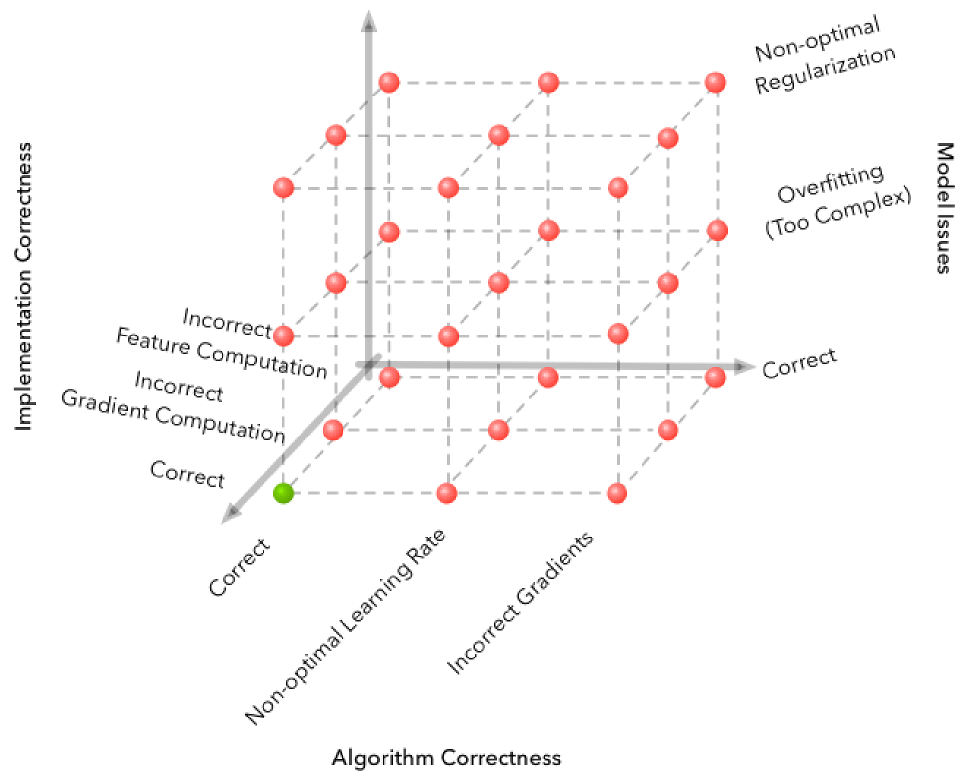


Abbildung 6.1.1: Beispielhafter Raum der Probleme bei ML-Verfahren, entnommen aus [39]. Das Bild hat einfachheitshalber nur drei Dimensionen, aber der Autor des Bildes merkt in der Originalpublikation an, dass es noch mehr Dimensionen/Kategorien an Problemen gibt.

rige *sample efficiency*. Die sample efficiency beschreibt, wie effektiv ein RL-Algorithmus von seinen gegeben beziehungsweise von der Umgebung generierten Beispielen (Obeservation + Aktion + Reward) lernen kann. Damit ein RL-Algorithmus ein beliebiges Atari-Spiel erlernt, benötigt dieser durchschnittlich 18 Millionen bis 70 Millionen Frames/Beispiele [12][2]. Das sind ungefähr 83 bis 324 Stunden, die dann ein Agent braucht um ein Atari-Spiel auf dem Niveau eines Menschen zu spielen. Ein Mensch benötigt bei diesen einfachen Spielen üblicherweise nur wenige Minuten. Bei der vorgestellten Publikation *NAS* (Unterkapitel 3.1) waren 12.800 Beispiele nötig um das Ziel zu erreichen, wobei 22.400 GPU-Stunden gebraucht worden sind[1]. Dabei sei an der Stelle außer Acht gelassen, ob bei NAS die Policy vollständig konvergierte oder nur zu einer Art geführten Parameter-Suche genutzt wurde (Diskutiert in Unterkapitel 4.1). Auch in dieser Masterarbeit wurde bei der Evaluierung anhand der Logs ständig beobachtet, wie bei der Exploration gute KNN-Architekturen entdeckt worden sind. Allerdings konnten die

1 Die Angabe ist aus der verwandten Publikation *NASNet*[42] ist nicht ganz eindeutig, weil angeblich 800 GPUs 28 Tage lang genutzt wurden. Dies würde aber in 537.600 GPU-Stunden resultieren und damit auf 42 Stunden je trainiertes KNN statt 1,75 Stunden je KNN bei 22.400 GPU-Stunden.

Agenten die dafür benötigten Aktionen meistens nicht Erlernen und damit das Ergebnis nicht wiederholen.

Eine weitere Schwierigkeit ist, dass RL-Agenten schnell in lokale Optima konvergieren und nur schwer diese verlassen können. Die Policy oder Value-Funktion hat eine Strategie so stark verinnerlicht, dass ein Abändern der Vorgehensweise des Agenten viele „Gegenbeispiele" benötigt. Durch das Anpassen eines speziellen Hyperparameter kann meistens die Exploration der Umgebung erhöht und damit das Verlassen des lokalen Optimums vereinfacht werden. Wird aber zu stark exploriert, lernt der Agent womöglich gar keine sinnvolle Strategie. Es ist also ein Kompromiss zwischen *exploration* und *exploitation* zu finden, was wiederum bisher ein ungelöstes Problem ist.

Probleme bereitet auch die Instabilität von RL-Algorithmen. Ob und wie schnell ein Agent konvergiert, hängt unter anderem von den zufällig gewählten Initialwerten der Modell-Parameter ab. Dies kann auch bei der zufälligen Initialisierung der Gewichte für KNNs beobachtet werden. Allerdings macht sich diese Schwierigkeit dort nicht so stark bemerkbar [5]. Bei RL-Algorithmen dagegen besteht je nach Situation eine Wahrscheinlichkeit von 30%, dass der Agent wegen ungünstig zufällig gewählten Initialwerten nicht konvergiert [13]. Und selbst wenn der Agent konvergiert, wird womöglich nur langsam oder nur ein lokales Optimum erreicht. [11].

Theoretisch lassen sich alle diese Probleme lösen, indem noch mehr Rechenleistung aufgewendet wird, um mehr Beispiele von der Umgebung zu generieren oder verschiedene (Hyper-)Parameter auszuprobieren. Alternativ können womöglich andere Techniken die selbe Aufgabenstellung effizienter lösen (wie zum Beispiel bei PNAS, Unterkapitel 3.1.2) oder ein Agent wird supervised vortrainiert und danach lediglich mithilfe von RL-Algorithmen feinoptimiert.

6.2 FAZIT

Während der Bearbeitung der Masterarbeit ist aufgefallen, dass die Aufgabenstellung noch schwerer ist als ursprünglich angenommen. Das Ziel, ein Agenten zu trainieren, der ein KNN für einen unbekannten Datensatz konstruiert und trainiert, konnte nicht erreicht werden. Es wurde zwar nicht erwartet, dass am Ende der Masterarbeit ein Agent zur Verfügung steht, der ein perfektes KNN konstruiert, aber dass der Agent zumindest versuchen würde für jeden Datensatz eine eigene passende KNN-Architektur zu konstruieren. Die in diesem Fall zusammenhängenden Schwierigkeiten wurden zuvor in Unterkapitel 6.1 ausführlich erläutert.

Beim Wettbewerb AutoDL wurden einige Einreichung hochgeladen, diese dienten allerdings nur zu technischen Testzwecken. Die Resultate sind nicht verwertbar, weil der Agent nicht erlernen konnte, mit einem unbekannten Datensatz zu arbeiten.

Mit der schlussendlich entwickelten Umgebung NACEnv wurde zumindest gezeigt, dass der Agent für einen bekannten Datensatz ein KNN zu kon-

struieren (Evaluation 6 in Unterkapitel 4.5.2) erlernen konnte. Dafür musste die Umgebung insofern stark vereinfacht werden, dass der Agent nur auf einem einzigen kleinen Datensatz experimentieren konnte. Das hatte den Vorteil, dass die Trainingszeit des Agenten mit 19 Stunden auf nur einer GPU relativ kurz war. Mit deutlich mehr Trainingszeit und gegebenenfalls einer weiteren Optimierung des NACEnv kann der Agent womöglich doch den Umgang mit unbekannten Datensätzen erlernen.

In Evaluation 6 konnte ebenfalls beobachtet werden, wie der Agent selbständig eine neue Strategie erlernen konnte. Der Agent fixiert beim Trainieren des KNNs die Lernrate und erhöht nach einigen Epochen die Batchsize. Diese Strategie wurde erst kürzlich in der Publikation [32] als eine effiziente Variante zum Trainieren eines KNNs vorgestellt. Dass der Agent diese ungewöhnliche Strategie erlernen konnte, unterstreicht das Potenzial vom Ansatz, welcher in dieser Masterarbeit vorgestellt wurde. Es kann automatisiert nach neuen, noch unbekannten Strategien und Regeln zum Konstruieren und zum Trainieren von KNNs gesucht werden, die manuell nur sehr aufwendig zu entdecken sind.

Der iterative Vorgang der Bearbeitung der Masterarbeit, der in den drei dokumentierten Versionen der Umgebung mündete, ermöglichte eine systematische Annäherung an das Ziel der Masterarbeit. Insbesondere konnte damit auch aufgezeigt werden, welche Methoden nicht oder nur schlecht funktionieren. Es wurden damit Erfahrungen gesammelt, die in die endgültige Version der Umgebung NACEnv einflossen.

Ein wichtiges Produkt dieser Arbeitsweise ist das Ergebnis, dass der naive Ansatz eine Umgebung zu entwickeln mit Aktionen die mehrere Parameter besitzen, wie in Version 1, für diese Masterarbeit nicht zweckmäßig ist. Statt dessen wurde in Version 2 mit der neu entwickelten *Action-State-Machine* (siehe Unterkapitel 4.4.2) eine Aktion mit nur einem Parameter, inklusive einer dynamischen Aktions-Wahlbeschränkung, eingeführt. Dies verhilft dem Agenten die Umgebung effizienter zu erlernen.

Neu ist zudem die implementierte Schichten-Injektion-Methode, die ein einfaches und effizientes Einfügen von neuen Schichten in ein bestehendes CNN ermöglicht (siehe Unterkapitel 4.5.3).

Des Weiteren wurden im Kapitel 5 mehrere Ideen vorgestellt, mit dem das Potential von NACEnv noch gesteigert werden kann.

Teil II

APPENDIX

A DATENSÄTZE

Auf dieser Seite befindet sich eine Tabelle zu allen erwähnten Datensätzen in dieser Masterarbeit. Alle Datensätze die in der Tabelle aufgelistet sind, wurden für den eigenen Ansatz im LevelManager verwendet. Die einzige Ausnahme stellt hier der Datensatz *ImageNet* dar. Weitere Informationen zu den Datensätzen sind auf der Webseite von *TensorFlow Datasets*[1] zu finden. Informationen zu den *SimpleDigits*-Datensätzen sind auf uci.edu[2] vorhanden.

Ob und welchem Level ein Datensatz zugeordnet ist, kann aus der Spalte „V1" für Version 1 und „V2/3" für Version 2 und Version 3 abgelesen werden. Alle Datensätze sind Klassifizierungsdatensätze. Bis auf den selbst generierten Datensatz *InvertSignal*, sind alle Datensätze Bilddatensätze.

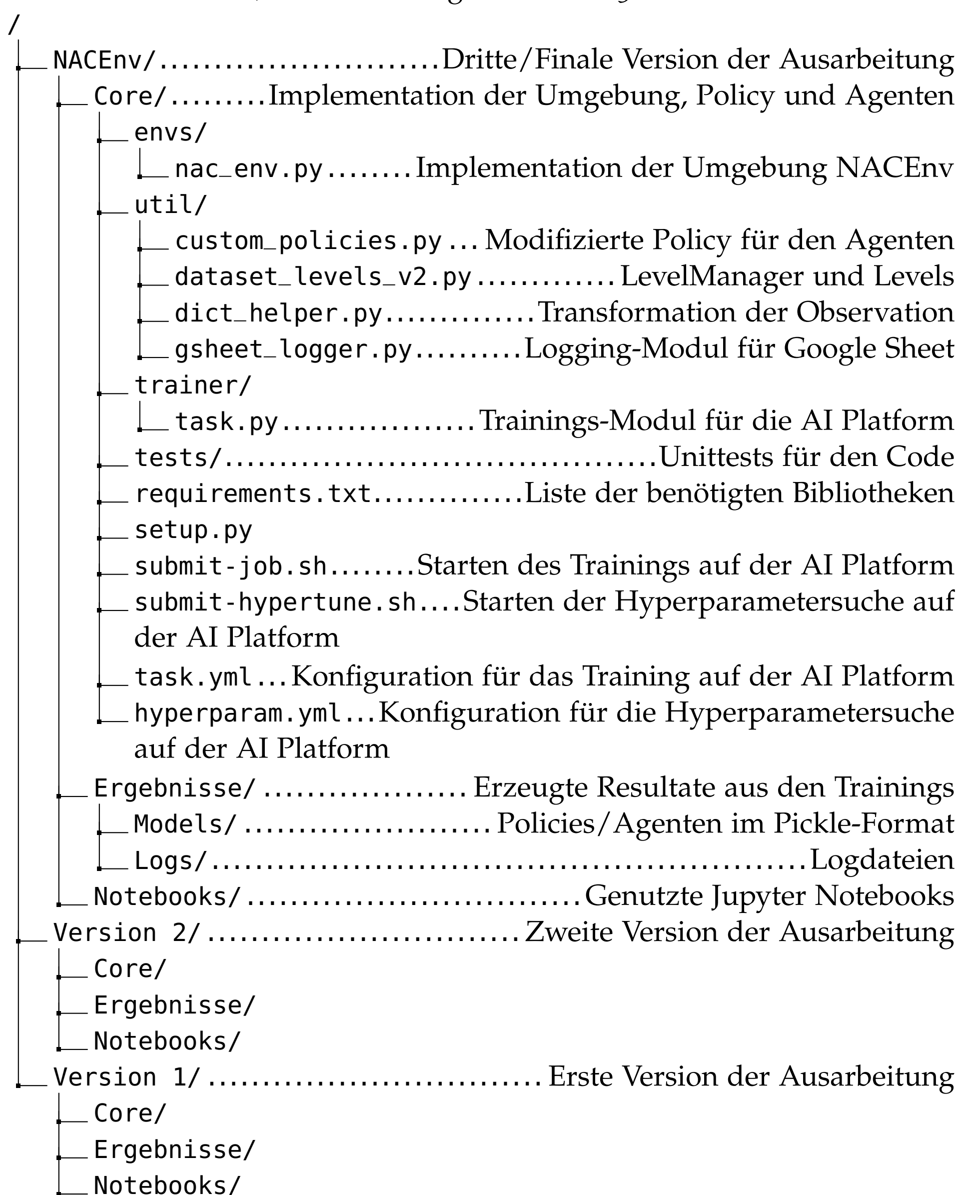
Name	**Beschreibung**	**(# Training, # Test)[a]**	**Format[b]**	**# Klassen**	**V1**	**V2/V3**
InvertSignal	Selbst generierter Datensatz mit 0er und 1er. Das Ziel ist, den Wert zu invertieren.	(400, 100)	(1)	2	0	-
SimpleDigits2	Bilder von handschriftlichen Zahlen 0 und 1	(300, 100)	(1, 8, 8)	2	1	0
SimpleDigits10	Bilder von handschriftlichen Zahlen von 0 bis 9	(300, 100)	(1, 8, 8)	10	2	1
FashionMNIST	Bilder von unterschiedlichen Kleidungsstücken	(60.000, 10.000)	(1, 28, 28)	10	3	2
CIFAR10	Bilder aus 10 unterschiedlichen Kategorien, wie z.B. Flugzeuge, Katzen	(50.000, 10.000)	(3, 32, 32)	10	4	3
MNIST	Bilder von handschriftlichen Zahlen	(60.000, 10.000)	(1, 28, 28)	10	-	4
CIFAR100	Bilder aus 100 unterschiedlichen Kategorien, wie z.B. Flugzeuge, Katzen	(50.000, 10.000)	(3, 32, 32)	100	-	5
ImageNet	Datensatz vom *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC)	(1.281.167, 50.000)	(3, ?, ?)	1.000	-	-

a Anzahl der Trainings- und Test-Elemente in den Datensätzen
b Format der Datenelemente. Die erste Stelle beschreibt für gewöhnlich den Farbraum der Bilder. 1 entspricht Monochrom. 3 entspricht RGB.

1 https://www.tensorflow.org/datasets/datasets
2 https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

B BEILIEGENDE DATEIEN

In diesem Kapitel wird kurz die beiliegende Ordnerstruktur samt Dateien beschrieben. Auf eine ausführlichere Beschreibung von Version 1 und Version 2 wurde verzichtet, weil sie analog zu Version 3 strukturiert ist.

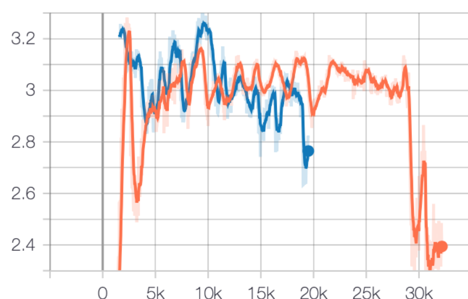


C PLOTS VON EVALUATION 9 & EVALUATION 9B

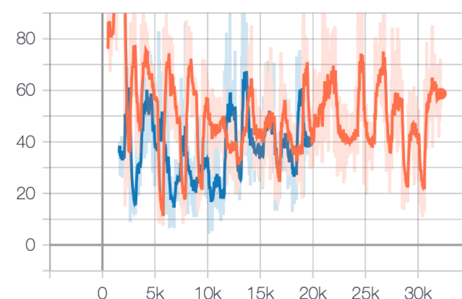
Im Folgenden sind die Trainings-Plots von den Experimenten Evaluation 9 und Evaluation 9b, basierend auf Version 3, zu sehen. Beschrieben sind die Experimente im Unterkapitel 4.5.2.

Die orangenen Linien kennzeichnen den Trainingsverlauf der Evaluation 9 und die blauen Linien die der Evaluation 9b. Die Linien sind zu einem gewissen Grad geglättet, damit die Trends besser erkennbar sind. Der wahre Trainingsverlauf ist in der jeweiligen Farbe halb-transparent dargestellt. Erstellt wurden die Plots mithilfe von Tensorboard.

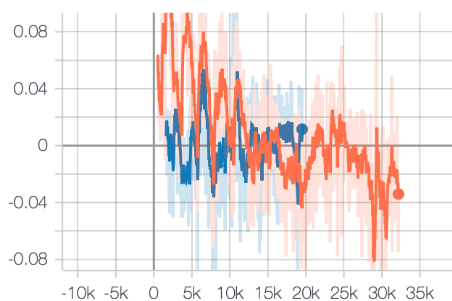
entropy_loss
tag: loss/entropy_loss



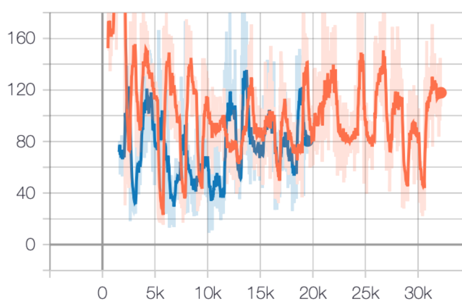
loss
tag: loss/loss



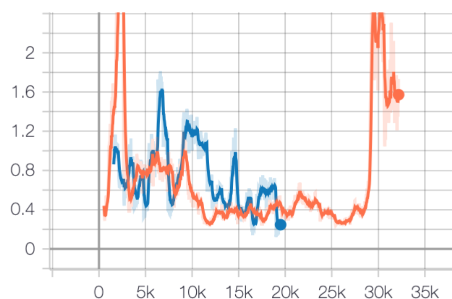
policy_gradient_loss
tag: loss/policy_gradient_loss



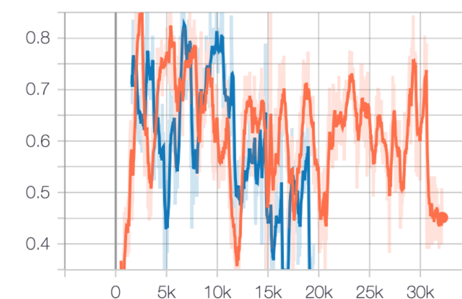
value_function_loss
tag: loss/value_function_loss



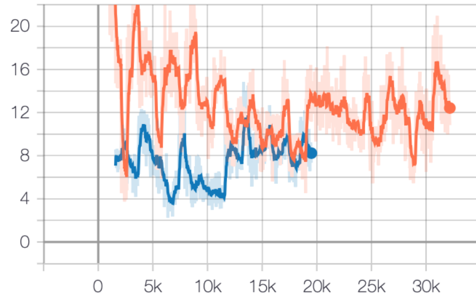
approximate_kullback-leiber
tag: loss/approximate_kullback-leiber



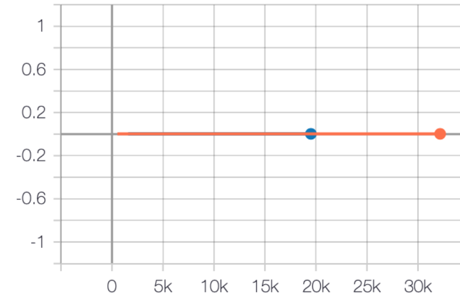
clip_factor
tag: loss/clip_factor



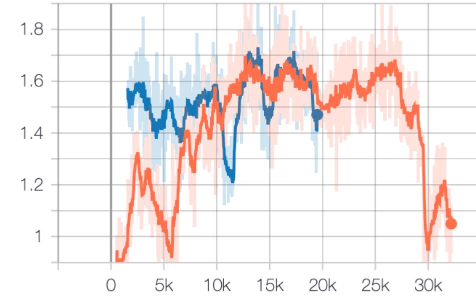
discounted_rewards
tag: input_info/discounted_rewards



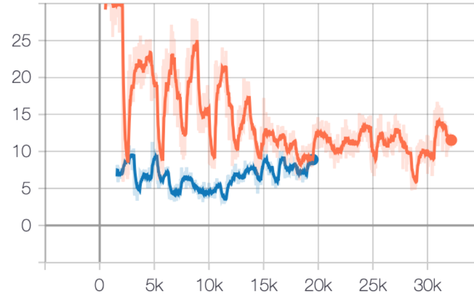
learning_rate
tag: input_info/learning_rate



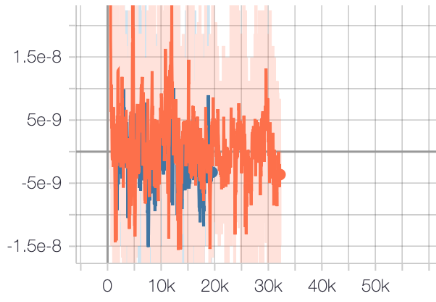
old_neglog_action_probabilty
tag: input_info/old_neglog_action_probabilty



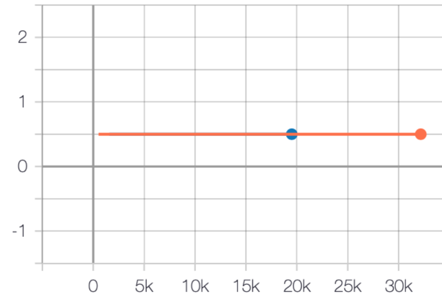
old_value_pred
tag: input_info/old_value_pred



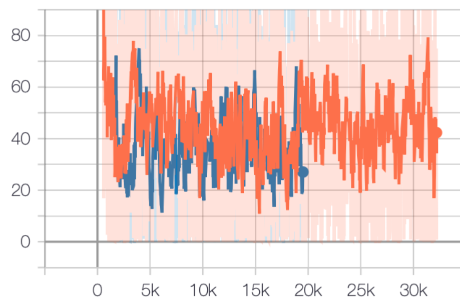
advantage
tag: input_info/advantage



clip_range
tag: input_info/clip_range



episode_reward



LITERATUR

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik und Ramesh Raskar. "Designing Neural Network Architectures using Reinforcement Learning". In: *ICLR* (2017). URL: http://arxiv.org/abs/1611.02167.

[2] Marc G. Bellemare, Will Dabney und Rémi Munos. "A distributional perspective on reinforcement learning". In: *34th International Conference on Machine Learning, ICML 2017*. Bd. 1. Juli 2017, S. 693–711. ISBN: 9781510855144. URL: http://arxiv.org/abs/1707.06887.

[3] Dan Clark. *Top 16 Open Source Deep Learning Libraries and Platforms*. 2018. URL: https://www.kdnuggets.com/2018/04/top-16-open-source-deep-learning-libraries.html.

[4] Amid Fish. *The Humble Gumbel Distribution*. 2018. URL: http://amid.fish/humble-gumbel.

[5] Jonathan Frankle und Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks". In: *ICLR 2019* (März 2018). URL: http://arxiv.org/abs/1803.03635.

[6] Xavier Glorot und Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)* 9 (2010), S. 249–256. ISSN: 15324435. DOI: 10.1.1.207.2059. URL: http://machinelearning.wustl.edu/mlpapers/paper_files/AISTATS2010_GlorotB10.pdf.

[7] Josh Greaves. *Everything You Need to Know to Get Started in Reinforcement Learning*. 2017. URL: https://joshgreaves.com/reinforcement-learning/introduction-to-reinforcement-learning/.

[8] Erik Hallström. *Backpropagation from the beginning*. 2016. URL: https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Bd. 2016-Decem. Dez. 2016, S. 770–778. ISBN: 9781467388504. DOI: 10.1109/CVPR.2016.90. URL: http://arxiv.org/abs/1512.03385.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun. "Identity mappings in deep residual networks". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Bd. 9908 LNCS. März 2016, S. 630–645. ISBN: 9783319464923. DOI: 10.1007/978-3-319-46493-0{_}38. URL: http://arxiv.org/abs/1603.05027.

[11] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup und David Meger. "Deep reinforcement learning that matters". In: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*. Sep. 2018, S. 3207–3214. ISBN: 9781577358008. URL: http://arxiv.org/abs/1709.06560.

[12] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar und David Silver. "Rainbow: Combining improvements in deep reinforcement learning". In: *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*. Okt. 2018, S. 3215–3222. ISBN: 9781577358008. URL: http://arxiv.org/abs/1710.02298.

[13] Alex Irpan. *Deep Reinforcement Learning Doesn't Work Yet*. 2018. URL: https://www.alexirpan.com/2018/02/14/rl-hard.html.

[14] Diederik P. Kingma und Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *ICLR 2015* (Dez. 2014). URL: http://arxiv.org/abs/1412.6980.

[15] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard und L. D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* (2008). ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

[16] Yann LeCun, Léon Bottou, Yoshua Bengio und Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* (1998). ISSN: 00189219. DOI: 10.1109/5.726791.

[17] Y Lecun u. a. "Comparison of Learning Algorithms for Handwritten Digit Recognition". In: *INTERNATIONAL CONFERENCE ON ARTIFICIAL NEURAL NETWORKS*. 1995.

[18] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang und Kevin Murphy. "Progressive Neural Architecture Search". In: *ECCV 2018* (Dez. 2017). URL: http://arxiv.org/abs/1712.00559.

[19] Dmytro Mishkin, Nikolay Sergievskiy und Jiri Matas. "Systematic evaluation of convolution neural network advances on the Imagenet". In: *Computer Vision and Image Understanding* 161 (Juni 2017), S. 11–19. ISSN: 1090235X. DOI: 10.1016/j.cviu.2017.05.007. URL: http://arxiv.org/abs/1606.02228http://dx.doi.org/10.1016/j.cviu.2017.05.007.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra und Martin Riedmiller. "Playing Atari with Deep Reinforcement Learning". In: *Deepmind* (Dez. 2013). URL: http://arxiv.org/abs/1312.5602.

[21] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2016. URL: http://neuralnetworksanddeeplearning.com/.

[22] OpenAI u. a. "Learning Dexterous In-Hand Manipulation". In: *CoRR* (Aug. 2018). URL: http://arxiv.org/abs/1808.00177.

[23] *Review of Deep Learning Algorithms for Image Classification*. 2019. URL: https://medium.com/zylapp/review-of-deep-learning-algorithms-for-image-classification-5fdbca4a05e2.

[24] Kevin Rojczyk. "Automatische Optimierung von System-Konfigurationen in Java mit Fokus auf Apache Spark". Bachelorthesis. Hochschule Darmstadt, 2017, S. 58.

[25] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Okt. 1986), S. 533–536. ISSN: 00280836. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[26] Olga Russakovsky u. a. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (Dez. 2015), S. 211–252. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0816-y. URL: http://link.springer.com/10.1007/s11263-015-0816-y.

[27] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan und Pieter Abbeel. "Trust region policy optimization". In: *32nd International Conference on Machine Learning, ICML 2015*. Bd. 3. Feb. 2015, S. 1889–1897. ISBN: 9781510810587. URL: http://arxiv.org/abs/1502.05477.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford und Oleg Klimov. "Proximal Policy Optimization Algorithms". In: *OpenAI* (Juli 2017). URL: http://arxiv.org/abs/1707.06347.

[29] David Silver u. a. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *Deepmind* (Dez. 2017). URL: http://arxiv.org/abs/1712.01815.

[30] Thomas Simonini. *An intro to Advantage Actor Critic methods: let's play Sonic the Hedgehog!* 2018. URL: https://www.freecodecamp.org/news/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d/.

[31] Karen Simonyan und Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *ICLR 2015* (Sep. 2014). URL: http://arxiv.org/abs/1409.1556.

[32] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying und Quoc V. Le. "Don't Decay the Learning Rate, Increase the Batch Size". In: *ICLR 2018* (Nov. 2017). URL: http://arxiv.org/abs/1711.00489.

[33] Ilya Sutskever, Geoffrey Hinton, Alex Krizhevsky und Ruslan R Salakhutdinov. "Dropout : A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* (2014).

[34] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning : An Introduction*. Second edi. Massachusetts, United States: MIT Press Ltd, 2018, S. 526. ISBN: 9780262039246. URL: http://incompleteideas.net/book/the-book-2nd.html.

[35] Richard S. Sutton, Doina Precup und Satinder Singh. “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning”. In: *Artificial Intelligence* 112.1-2 (Aug. 1999), S. 181–211. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(99)00052-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370299000521.

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke und Andrew Rabinovich. “Going deeper with convolutions”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Bd. 07-12-June. Sep. 2015, S. 1–9. ISBN: 9781467369640. DOI: 10.1109/CVPR.2015.7298594. URL: http://arxiv.org/abs/1409.4842.

[37] Christopher Watkins. “Learning from delayed rewards”. In: *Robotics and Autonomous Systems* 15.4 (1989), S. 233–235. ISSN: 09218890. DOI: 10.1016/0921-8890(95)00026-c.

[38] Ronald J. Williams und Ronald J. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 8.3-4 (Mai 1992), S. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: http://link.springer.com/10.1007/BF00992696.

[39] S. Zayd Enam. *Zayd’s Blog – Why is machine learning 'hard'?* 2016. URL: http://ai.stanford.edu/~zayd/why-is-machine-learning-hard.html.

[40] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao und Cheng-Lin Liu. “Practical Block-wise Neural Network Architecture Generation”. In: *CVPR* (2018). URL: http://arxiv.org/abs/1708.05552.

[41] Barret Zoph und Quoc V. Le. “Neural Architecture Search with Reinforcement Learning”. In: *ICLR 2017* (Nov. 2017). URL: http://arxiv.org/abs/1611.01578https://ai.google/research/pubs/pub45826.

[42] Barret Zoph, Vijay Vasudevan, Jonathon Shlens und Quoc V. Le. “Learning Transferable Architectures for Scalable Image Recognition”. In: *CVPR* (2018). URL: http://arxiv.org/abs/1707.07012.