Evolutionary Neural Architecture Search for Sequential Deep Neural Networks on Time Series Data

Evolutionäre Suche von Architekturen Sequenzieller Tiefer Neuronaler Netze auf Zeitreihendaten

Valentin Döring, Leander Masopust

Universitätsbachelorarbeit
zur Erlangung des akademischen Grades

Bachelor of Science
(B. Sc.)

im Studiengang
IT Systems Engineering

eingereicht am 28. Juli 2022 am Fachgebiet
Digital Health - Connected Healthcare
der Digital-Engineering-Fakultät
der Universität Potsdam

Gutachter	Prof. Dr. Bert Arnrich
Betreuer	Orhan Konak

Abstract

Deep Neural Networks have steadily gained importance due to great success in many application areas. The selection of an optimal combination of a data preprocessing technique, model architecture and training parameters is a labor-intensive process. In this bachelor thesis, we propose an algorithm, that automates the architecture search for the task of *Classification of Time Series Data*. Sequential architectures are found, evaluated and improved using evolutionary techniques. Compared to standard architectures in that field, the evolutionary evolved architecture achieves an accuracy improvement of up to 5.82 percent.

Zusammenfassung

Tiefe Neuronale Netze haben durch große Erfolge in vielen Anwendungsbereichen stetig an Bedeutung gewonnnen. Dabei ist die Auswahl einer optimalen Kombination von Datenvorverarbeitungstechnik, Modellarchitektur und Trainingsparametern ein arbeitsaufwendiger Prozess. Für die Problemdomäne *Klassifizierung von Zeitreihendaten* wird in dieser Arbeit ein Algorithmus vorgeschlagen, der die Architekturfindung automatisiert. Mithilfe evolutionärer Techniken werden sequentielle Architekturen gefunden, evaluiert und verbessert. Im Vergleich zu Standardarchitekturen der gleichen Domäne erreicht die evolutionär entwickelte Architektur eine bis zu 5.82 Prozent höhere Genauigkeit.

Inhaltsverzeichnis

Autoren

1. Einleitung: Valentin Döring
2. Voraussetzungen: Leander Masopust
3. Wissenschaftlicher Hintergrund: Leander Masopust
4. Vorgeschlagener Algorithmus
   - 4.1. Programmüberblick: Valentin Döring
   - 4.2. Evolutions-Algorithmus
     - 4.2.1. Umfang der evolutionären Suche: Leander Masopust
     - 4.2.2. Überblick: Valentin Döring
     - 4.2.3. Evolutionäre Techniken: Leander Masopust
     - 4.2.4. Implementierung
       - 4.2.4.1. Einleitung: Valentin Döring
       - 4.2.4.2. Evolutionärer Kontext im ModellGenom: Valentin Döring
       - 4.2.4.3. Mutation des ModellGenoms: Valentin Döring
       - 4.2.4.4. Vom ModellGenom zum Keras-Modell: Leander Masopust
5. Experimente: Leander Masopust
6. Diskussion: Valentin Döring
7. Konklusion: Valentin Döring

1 Einleitung

Bedeutung von Deep Learning Deep Learning spielt eine immer größere Rolle beim Lösen komplexer Probleme wie der Bilderkennung oder der Aktivitätserkennung. In immer mehr Bereichen ist es Machine Learning mit Feature-Extraktion überlegen.
Die benötigte Rechenleistung für das Training ist kein großer Nachteil mehr, weil diese erschwinglich geworden ist. Die Anwendung Tiefer Neuronalen Netze benötigt weniger domänenspezifisches Fachwissen, wodurch mehr Experten zur Verfügung stehen. Darüber hinaus ist ein einmal trainiertes Netz mittlerweile performant genug, um es in Echtzeit-Applikation, zum Beispiel auf dem Smartphone, zu verwenden.

Erklärbarkeit von Deep Learning Die Erklärbarkeit der Netze bleibt hingegen weiterhin eine Herausforderung. Selbst wenn auf Testdaten eine großartige Genauigkeit erzielt wurde, ist die Frage, anhand welcher Merkmale diese Entscheidung getroffen wurde, nicht direkt beantwortbar. Für die einzelnen Architekturkomponenten ist konzeptionell bekannt, welchen Beitrag sie zum Lernen leisten sollen. Trotzdem bleibt unbekannt, was die konkreten Eigenschaften der Daten sind, welche die Komponenten extrahiert haben und damit das Erlernen möglich gemacht haben. Wenn wir wüssten, auf welche Art die Modelle lernen, könnten wir herleiten welche Konfiguration das Lernen optimal unterstützt.

Semantik von Deep Learning Modellen Für die bekannte CNNLSTM - Architektur [Sai+15] haben wir die Intuition, dass Convolutional Schichten besonders gut darin sind, aus eine Fülle roher Eingabedaten sinnvolle Features zu extrahieren. Die Stärke der darauf folgenden LSTM Schichten ist es dann, aufbauend auf dieser Abstraktionsebene, zeitliche Zusammenhänge herzustellen. Es ist naheliegend, dass auf Zeitreihendaten das Einbeziehen zeitlicher Informationen eine sinnvolle Lernstrategie ist.
Über den Verlauf unseres Programms haben wir eine Vielzahl von Modellarchitekturen ausgewertet. Dabei ist uns aufgefallen, dass selbst sehr unterschiedliche Architekturen hohe Genauigkeiten erreichen können. Wenn die vielversprechende CNNLSTM-Architektur in der Evolution von einer ungewöhnlich parametrisierten CNN-Dense-CNN Architektur übertroffen wird, stellt man seine Annahmen über

den Anwendungsbereich der Komponenten in Frage. Daran wird deutlich, wie vage Intuitionen über die Semantik der Modelle sein können.

Schwächen händischer Modelloptimierung Die Optimierung der Modellarchitektur bewegt sich in einem großen Problemraum. Die Schichten können in beliebiger Reihenfolge vorkommen und auf unterschiedliche Arten parametrisiert werden.
Die händische Optimierung kann mit ihrer limitierten Anzahl von Versuchen der Größe des Problemraums nicht gerecht werden. Die Entscheidung, welche Modellarchitektur als nächstes getestet wird, ist daher bewusst oder unbewusst vom einzig vorhandenen Bezugspunkt, nämlich der vermuteten Funktionsweise der Komponenten, beeinflusst. Die Architektur wird nicht in ihrer Reihenfolge vollständig umgedreht. Stattdessen werden beispielsweise an der bekannten CNN-LSTM Struktur kleine Änderungen vorgenommen. Der abgesuchte Problemraum bleibt klein.
Unsere Beobachtung, dass auch sehr unterschiedliche Architekturen hohe Genauigkeiten erzielen können, lässt vermuten, dass der händischen Optimierung viel Potenzial entgeht.
Auch wenn man die Optimierung mit GridSearch oder RandomSearch [BB12] automatisiert, wählen immer noch voreingenommene Menschen die Variablen. Mit jedem weiteren gewählten Parameter erhöht sich die Anzahl der Möglichkeiten um eine Potenz. Also wird hier vorsichtig gewählt. Es sind mehr Versuche möglich, aber der Fokus liegt mehr auf der Parametrisierung anstatt in der Entwicklung von Architektur Innovationen.

Daten- und Modellzentrierte Optimierung Das Paper *The Unreasonable Effectiveness of Data* [HNF09] bestätigt unsere Beobachtung, dass eine Vielzahl verschiedener Modelle gute Genauigkeiten erreichen können. Es wird vorgeschlagen, sich auf Menge und Güte der Trainingsdaten anstatt auf die Modellarchitektur zu konzentrieren. Die Daten haben einen wesentlich größeren Einfluss auf das Ergebnis als die Architektur des Modells. Mit ausreichend, präzisen Daten sinkt die Bedeutung einer guten Architektur.
Es mag der effektivste Hebel sein, aber in der Praxis muss man diesem Fokus die kostenintensive Datenaufnahme entgegengehalten. Weiterhin ist die unbegrenzte Datensammlung, zum Beispiel im medizinischen Bereich, datenschutzrechtlich nicht möglich. Selbst wenn am Ende genug Daten für die erwünschte Genauigkeit gesammelt wurden, verbleibt immer noch Potenzial für die Architekturoptimierung. Auch dann, wenn es prozentual nur wenig ins Gewicht fällt, verbessert die

Architektur das Endergebnis. Es ist demnach in allen Fällen von Interesse, auch der Architektur Aufmerksamkeit zu schenken.

Vergleichbarkeit von Deep Learning Experimenten Um für einen neuen Datensatz eine passende Modell-Architektur zu finden, ist es lohnenswert nach erfolgreichen Beispielen in der gleichen Domäne zu schauen. Für die Aktivitätserkennung stehen beispielsweise einige Versionen der CNNLSTM-Architektur, sowie die etablierte InnoHAR Architektur zur Auswahl.
Eine Herausforderung sind dabei die unterschiedlichen Kriterien, anhand derer diese Architekturen gemessen werden. Es ist entscheidend, wieviele Zeiteinheiten dem Modell für eine Erkennung übergeben werden und welche Sensoren auf welchen Positionen gewählt werden. Falls im Trainingsdatensatz Abschnitte von allen Aufnahmen und Subjekten vorhanden sind, kann sich das Modell Eigenheiten der Daten merken, anstatt die Muster der Bewegung zu generalisieren. Eine Architektur die viele Informationen speichert, kann im Experiment gut abschneiden, aber für die Anwendung auf neue Daten unbrauchbar sein.
Selbst wenn die Kriterien so gewählt sind, dass sie die Generalisierbarkeit bewerten, kommt es dennoch durch unterschiedliche Datenaufteilungen sowie der zufälligen Initialisierung des Netzes zu Schwankungen in der Genauigkeit. Beispiele erfolgreicher Architekturen können bei einem strengen Bewertungsmaßstab schwache Ergebnisse liefern.
Darüber hinaus gibt es für jeden Datensatz durch die Art der Sensoren, die Auswahl der Subjekte, die Umgebung der Aufnahme und die Komplexität der Aktivitäten spezifische Herausforderungen. Es wäre praktisch, eine Architektur zu kennen, die mit allen Datensätzen gleichermaßen gut umgehen kann. In der Praxis zeigt sich jedoch datenspezifisches Optimierungspotenzial.

Wir stellen in dieser Arbeit einen Algorithmus vor, der evolutionär Modellarchitekturen entwickelt. An die Stelle der händischen Optimierung, der aufwendigen Recherche nach dem besten Modell für den eigenen Bewertungsstandard, tritt eine Automatisierung. An die Stelle der voreingenommenen Einschätzung bezüglich der auszuprobierenden Architekturen tritt die unparteiische Bewertung nach der tatsächlichen Leistung der Modelle.

Wir haben auf zwei Datensätzen etablierte, erfolgreiche Architekturen der Aktivitätserkennung mit unserem Algorithmus verglichen. Selbst nach Optimierung der Trainingsparameter aller Modelle setzt sich die ungewöhnliche, evolutionäre Architektur mit fünf beziehungsweise sechs Prozent mehr Genauigkeit gegen alle

anderen Modelle durch.

Die Arbeit ist wie folgt gegliedert. In Kapitel zwei klären wir die Grundlagen Tiefer Neuronaler Netze und genetischer Algorithmen. Danach ordnen wir uns in den aktuellen wissenschaftlichen Kontext ein. Im vierten Kapitel stellen wir unseren Algorithmus im Detail vor. Anschließend wird unser Algorithmus im experimentellen Teil auf die Probe gestellt. In der Diskussion bewerten wir die Ergebnisse. Die Konklusion fasst die Arbeit zusammen und schlägt Ideen für die weitere Forschung vor.

2 Voraussetzungen

Im folgenden Kapitel definieren wir die nötigen Begrifflichkeiten, deren Bedeutung zum Verständnis der restlichen Arbeit nötig sind. Zuerst definieren wir *Tiefe Neuronale Netze* und *evolutionäre Algorithmen*. Im Anschluss wird der Terminus *Evolutionary Neural Architecture Search* als Kombination dieser beiden Themengebiete erläutert.

2.1 Tiefe Neuronale Netze

Um Klassifikationsprobleme hoher Komplexität zu lösen, können Lösungsansätze des Maschinellen Lernens eingesetzt werden. Eine Lösungsteilmenge von Maschinellem Lernen sind Tiefe Neuronale Netze (DNN). Tiefe Neuronale Netze sind durch ihre Netzstruktur, im Folgenden Architektur genannt, dem Aufbau eines biologischen Gehirns nachempfunden. Hintereinandergereihte Neuronenschichten ermöglichen es hierbei mithilfe von Trainingsiterationen, Datenzusammenhänge so zu ermitteln, dass Vorhersagen auf neuen Daten getroffen werden können.

2.1.1 Künstliche Neuronen

Künstliche Neuronen sind die mathematische Repräsentation von biologischen Neuronen. Gegeben eine Menge von Eingabewerten werden ebendiese unterschiedlich gewichtet und mit einem Bias versehen. Damit die Konkatenation von mehreren Neuronen nicht in einer Linearkombination von Funktionen resultiert, wird anschließend eine nicht-lineare Aktivierungsfunktion angewendet.
Sei im folgenden $X \in \mathbb{R}^n$ der Eingabevektor, $W \in \mathbb{R}^n$ der Gewichtungsvektor, b der Bias und φ die Aktivierungsfunktion des Neurons. Dann ergibt sich die in Gleichung 2.1 aufgeführte Formel, durch die der Wert eines Neurons repräsentiert wird.

$$X \mapsto \varphi((\sum_{i=0}^{n-1} w_i * x_i) + b) \tag{2.1}$$

2.1.2 Trainingsprozess

Um das neuronale Netz zu trainieren, werden nach einer Menge von Eingabedatenpunkten (Batchgröße) die Gewichtungen der Neuronenverbindungen angepasst. Dies geschieht durch iterative Rückpropagierung. Rückpropagierung bezeichnet das Anpassen der Gewichte und des Bias von Neuronen und deren Verbindungen. Dies funktioniert durch Berechnung des Fehleranteils jedes Neurons an der Vorhersage milfhile des Produkts partieller Ableitungen.

2.1.3 Neuronale Schichten

Um die Komplexität eines Tiefen Neuronalen Netzes zu bewältigen, werden Netze in Schichten aufgeteilt. Anhand des internen Aufbaus einer Schicht, kann ihr eine vage Semantik zugewiesen werden. Dadurch können spezifische Aufgaben, wie Feature Extraktion oder das Erkennen zeitlicher Abhängigkeiten, bestimmten Schichttypen zugeordnet werden. Im Folgenden definieren wir die für unsere Arbeit wichtigsten Schichttypen.

Dense

Eine Dense Schicht, beispielhaft visualisiert in Abbildung 2.1, ist als das Mapping von n Eingabeneuronen auf m Ausgabeneuronen definiert. Hierbei hat jedes Eingabeneuron eine Verbindung zu jedem Ausgabeneuron.

Dropout

Die Güte von Modellen des Maschinellen Lernens wird an ihrer Fähigkeit zur Generalisierung gemessen. Im Trainingsprozess besteht die Gefahr, dass Modelle datenspezifische Eigenschaften speichern, anstatt allgemeingültige Muster zu erlernen. Diesem Problem, bekannt als Overfitting [Die95], widmet sich unter anderem die Dropout Schicht [Sri+14]. Dropout Schichten mappen n Eingabeneuronen identitär auf $n - x$ Ausgabeneuronen, wobei ebendieses x die parametrisierbare Anzahl von Neuronen ist, deren Wert genullt wird, siehe Abbildung 2.2. Hierdurch muss das Modell bei dem Trainingsprozess auf zufällige Eingabedatenpunkte verzichten und lernt somit auch, mit weniger Informationsgehalt korrekt vorhersagen zu können.

Convolutional

Um lokal zusammenhängende Merkmale aus gegebenen Eingabedaten zu extrahieren, werden Convolutional Schichten [BL97] genutzt. Diese wenden einen *Filter*

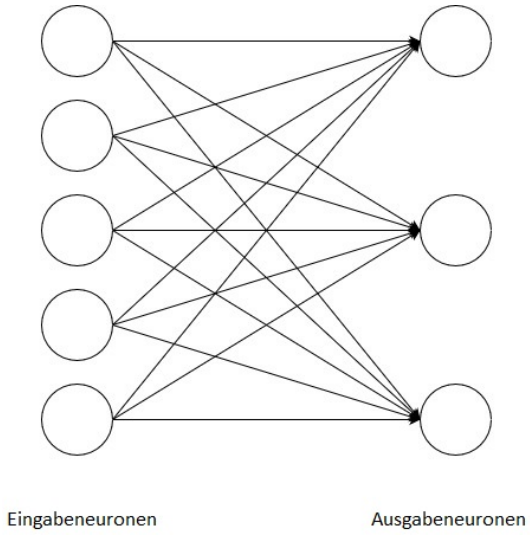


Abbildung 2.1: Dense Schicht mit 5 Eingabeneuronen und 3 Ausgabeneuronen

auf die Eingabedaten über eine - Convolutional1D - oder zwei - Convolutional2D - Dimensionen an und erzeugen dadurch *Featuremaps*, in denen jeweils verschiedene, extrahierte Eingabedatenmerkmale hervorgehoben sind.

Max Pooling

Max Pooling Schichten [SMB10] aggregieren das Maximum der Eingabedaten innerhalb eines parametrisierbaren Bereiches über eine oder zwei Dimensionen, indem sich ein Kernel über diese Dimensionen bewegt. Max Pooling Schichten werden häufig im Anschluss an Convolutional Schichten gehängt, um durch die Extraktion der höchsten Werte wichtige Bereiche wie Kanten bei Bildern aus den Featuremaps hervorzuheben. Max Pooling Schichten sind verlustbehaftet, wodurch sie indirekt auch dem Problem des *Overfitting* entgegenwirken kann. Außerdem kann Max Pooling zusätzlich die Datenkomplexität reduzieren.

Batch Normalisierung

Batch Normalisierungs Schichten werden meist zwischen eine andere Schicht und ihre Aktivierungsfunktion geschaltet. Sie normalisieren die Ausgabe der vorherigen Schicht abhängig von dem Durchschnitt und der Standardabweichung des momentanen *Batches* - die Menge der Eingabedaten, nach denen die Gewichtung des Netzes angepasst wird. Batch Normalisierungen wirken dem Problem der *internen*

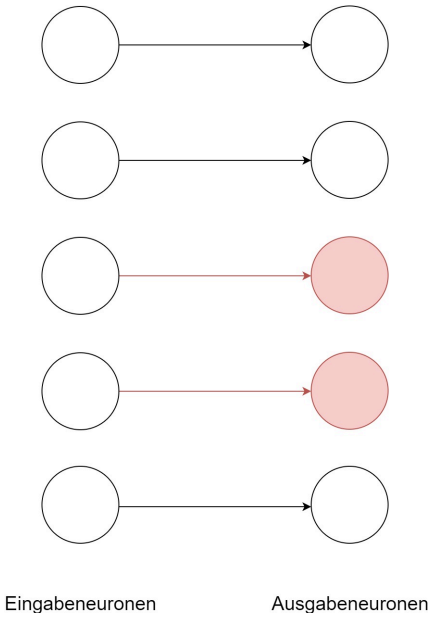


Abbildung 2.2: Dropout Schicht mit 5 Eingabeneuronen, 3 identitär gemappten Ausgabeneuronen und 2 genullten Ausgabeneuronen

Kovarianzverschiebung entgegen, was eine Beschleunigung des Trainingsprozesses zur Folge hat [IS15]. Desweiteren führt die Normalisierung zu ähnlichen Wertebereichen der Ausgaben der vorherigen Schichten. Dadurch weisen die Gradienten beim Trainingsprozess weniger Schwankungen auf und konvergieren somit besser zum Minimum der Verlustfunktion.

Flatten

Eine Möglichkeit, um die Dimensionen der Daten zu reduzieren, ist die Verwendung einer Flatten Schicht. Ein multidimensionale Eingabetensor wird identitär auf eine eindimensionale Menge von Neuronen gemappt.

Long Short Term Memory

Eine große Herausforderung an neuronale Netze für Zeitreihendaten ist das Erkennen von Merkmalen über lange Zeitspannen. In besonders tiefen Netzarchitekturen tritt das *Vanishing Gradient Problem* [Hoc98] auf, was besonders tiefe Netze durch die häufige Multiplikation von partiellen Ableitungen im Rückpropagierungsprozess daran hindert, effizient zu lernen. *Recurrent Neural Nets* (RNNs) [LBE15] können

zeitliche Abhängigkeiten in Daten besonders gut erkennen, sind jedoch durch ihre tiefe Struktur besonders von diesem Problem betroffen. Daher werden häufig *Long Short Term Memory* (LSTM) Schichten genutzt, welche das Vanishing Gradient Problem minimieren, indem die Gradienten in *Memory Cells* über Zeitschritte hinweg gespeichert werden [HS97].

2.2 Genetische Algorithmen

Genetische Algorithmen sind eine Klasse von Optimierungsalgorithmen, die heuristisch einen gegebenen Problemraum durch aus der Biologie abgeleiteten Prinzipien, wie der natürlichen Selektion, Kreuzung und Mutation, explorieren [Mit98]. In einem iterativen Ansatz werden neue Individuen aus Individuen der vorherigen Generation durch Mutation oder Kreuzung erzeugt. Nach dem Prinzip *Survival of the Fittest* werden Individuen, die für die Lösung des Problems geeigneter als andere Individuen sind, bevorzugt überleben, sich häufiger fortpflanzen und somit mehr genetischen Einfluss auf Folgegenerationen nehmen.

Genom

Ein Genom ist die Summe der Eigenschaften eines spezifischen Individuums. Genome müssen durch eine *Fitnessfunktion* auf ihre Eignung als Lösungsindividuum evaluierbar sein.

Generation

Generationen bezeichnen im Kontext von genetischen Algorithmen die Menge von Individuen, welche aus demselben Pool von Elternindividuen entstanden sind. Somit umfasst eine Generation den Evolutionsstand zu einem gegebenen Zeitpunkt.

Selektion

Um die nächste Generation zu erzeugen, müssen Individuen aus der momentanen Generation so ausgewählt werden, dass Eigenschaften, die zu einer guten Fitness geführt haben, weitergetragen werden. Ebendiese Auswahl wird im Folgenden Selektion genannt.

Kreuzung

Nachdem Elternindividuen für die nächste Generation selektiert wurden, müssen neue Individuen erzeugt werden. Ein Ansatz hierfür ist die Kombination der Genome zweier Elternindividuen zu einem Kindindividuum. Diese Kombination zweier Elternindividuen nennt sich Kreuzung.

Mutation

Da durch Kreuzung allein nur bereits existente Attribute verwendet werden, kann keine Attributvarietät hergestellt werden, wodurch es über die Generationen nur

zu lokalen Optima kommt. Daher werden Gene von bestehenden Genomen mit einer festgelegten Wahrscheinlichkeit um eine zufällige Intensität abgeändert. Diese Mutationen ermöglichen je nach Intensitätsgrad sowohl feingranulare, lokale Optimierung als auch Problemraumexploration durch starke Mutation.

2.3 Evolutionary Neural Architecture Search

Die Qualität Tiefer Neuronaler Netze ist von der ausgewählten Architektur abhängig. Um akkurate Architekturen zu finden, gibt es verschiedene Ansätze. Ein Ansatz ist die Nutzung etablierter Netzarchitekturen oder das manuelle Verknüpfen von Schichten, sodass der entstehenden Struktur eine grobe Semantik zugeordnet werden kann. Hierfür benötigt es jedoch tiefgehendes Problemraumverständnis, Einblick in die Struktur der Daten und spezifisches Fachwissen bezüglich Deep Learning.
Ein weiterer Ansatz ist *Neural Architecture Search* (NAS), was als das automatisierte Auffinden einer zum Problemraum passenden Netzarchitektur definiert ist [EMH18]. NAS wird in die drei verschiedenen Teilgebiete *Reinforcement based NAS*, *Gradient based NAS* und *Evolutionary Computation based NAS* eingeteilt. Für letzteren Teilbereich, auch bekannt als *Evolutionary Neural Architecture Search* (ENAS), werden evolutionäre Algorithmen eingesetzt, um einen datenspezifischen, heuristisch optimalen Netzaufbau zu ermitteln [Liu+21]. Der Begriff *Neuroevolution* wird teilweise äquivalent zu ENAS verwendet, wobei Neuroevolution meist auch die evolutionäre Gewichtsanpassung anstelle von Rückpropagierung beinhaltet [Liu+21; Sta12].

3 Wissenschaftlicher Hintergrund

Dieses Kapitel befasst sich mit verschiedenen Arbeiten, die sich bereits mit *Evolutionärer Hyperparameteroptimierung*, *Neuroevolution* und *Evolutionary Neural Architecture Search* beschäftigt haben. Im Folgenden erörtern wir diese und grenzen uns von ihnen ab.

Evolutionäre Hyperparameteroptimierung Singh u. a. [Sin+19] beschäftigen sich mit der Optimierung von Schichtparametern für *Convolutional Neural Nets*. Unter Zuhilfenahme von üblichen Evolutionstechniken, wie Ein-Punkt Kreuzung und zufälliger Mutation, werden die Kernelgrößen von Convolutional Schichten und Max Pooling Schichten angepasst. Auch Cheung und Sable [CS11] nutzen evolutionäre Algorithmen zur Parameteroptimierung von Convolutional Schichten. Spezifischer werden sowohl die Anzahl der Featuremaps der einzelnen Convolutional Schichten als auch die Verbindungen zwischen einzelnen Schichten einer LeNet-5 Modellarchitektur [LeC+89] über Generationen hinweg angepasst. Dahou u. a. [DEZ19] verwenden die evolutionäre Parametrisierung von Convolutional und Dense Schichten zur Sentimentanalyse von arabischen Texten und optimieren hierbei unter anderem die Anzahl der Filter, die Kernelgröße, die Anzahl von Neuronen in einer Dense Schicht und die Dropout Rate einzelner Schichten. Zur Validierung werden zwei verschiedene Fitness Funktionen genutzt, ein zufälliger 80/20 Test-Trainings Split und K-Fold Kreuzvalidierung. Die aus der Evolution resultierenden Modelle erreichen bei der Evaluierung bessere Ergebnisse in den üblichen Metriken, wie Genauigkeit, F1, Präzision und Recall, als andere State-of-the-Art Modelle, wie Combined LSTM [AE17] oder CNN-base [SA19].

In dieser Arbeit verwenden wir ähnliche Mutations-, Kreuzungstechniken und Validierungsfunktionen, beschränken jedoch unsere Evolution nicht nur auf einzelne Parameter in Form von natürlichen Zahlen und vereinzelt reellen Zahlen, sondern erlauben auch die evolutionäre Entwicklung von kategorischen Schichtparametrisierungen oder internen Parameterabhängigkeiten, wie der Stride Parametrisierung von Convolutional Schichten. Desweiteren begrenzen wir uns nicht nur auf die Parametrisierung einzelner Schichten, sondern erlauben auch eine Änderung von Schichttypen und Schichtanzahl, beschränken uns dafür jedoch ausschließlich auf sequentielle Architekturen.

Neuroevolution Eine der ersten Arbeiten, die sich mit der Evolution von Netzstruktur, Modellparametern und Gewichtungen beschäftigte, ist das Paper über *NeuroEvolution of Augmenting Topologies* (NEAT) [SM02]. Der NEAT Algorithmus entwickelt sowohl die Größe des neuronalen Netzes in Form von Neuronen und Neuronenverbindungen als auch Neuronengewichte evolutionär. Varietät in den entstehenden Modellen wird durch das Einführen von *Spezies* bewahrt. NEAT ist eine der ersten Implementierungen, bei denen Individuen durch evolutionäre Komplexifizierung optimiert werden. Ein NEAT Modell besteht nicht wie übliche Tiefe Neuronale Netze aus einer Konkatenation verschiedener Layertypen, sondern entwickelt eine vollkommen eigene Netzstruktur.

Sowohl der in dieser Arbeit vorgeschlagene Algorithmus als auch NEAT befassen sich mit der evolutionären Entwicklung einer Modellarchitektur. Während der NEAT Algorithmus jedoch das Netz auf Neuronenebene entwickelt, setzt unser Algorithmus auf Schichtebene an. Wir machen Gebrauch von ähnlichen evolutionären Techniken wie NEAT, nutzen allerdings zum Training der Gewichte Rückpropagierung, während NEAT auch die Gewichte evolutionär anpasst.

NAS Zoph und Le [ZL16] befassen sich mit der Thematik *Neural Architecture Search* (NAS) mit *Reinforcement Learning*. Hierbei wird die Struktur eines zu optimierenden *Convolutional Neural Networks* (CNNs) schichtweise encodiert und von einem externen *Recurrent Neural Network* (RNN) durch ein *Reward Signal* in Form der Genauigkeit des resultierenden CNNs verbessert. Zusätzlich zur Schichtparameteroptimierung durch das RNN können auch *Skip Connections* hinzugefügt oder entfernt werden. Die zu entwickelten CNN Architekturen sind auf eine vorher festgelegte Größe beschränkt. Mit diesem Algorithmus konnten bessere Ergebnisse in den Bereichen *image classification* und *language modeling* als damalige State-of-the-art Ansätze erzielt werden.
Mithilfe von *Evolutionary Neural Architecture Search* (ENAS) schlagen Real u. a. [Rea+17] einen Ansatz vor, um Modellarchitekturen zur Klassifizierung komplexer Bilder zu entwickeln. Hierzu beschränken sie sich auf die evolutionäre Technik *Mutation*, um neue Individuen zu entwickeln. Im Gegensatz zu anderen Ansätzen bestehen die Genome der Individuen nicht nur aus Schichtparametrisierungen, sondern aus ganzen Convolutional Schichten, die evolutionär zum Modell hinzugefügt oder entfernt werden können. Auch die Filter oder Stride Parametrisierung der Schichten können mutiert und Skip Connections hinzugefügt oder entfernt werden. Zusätzlich wird die Mutation des Trainingsparameters *Lernrate* unterstützt. Alle Modelle der Startpopulation bestehen aus einer einzigen Schicht

ohne Convolution, wodurch eine schlechte Fitness garantiert und Verbesserungen angeregt werden. Der ausgewählte Problemraum der CNN Architekturfindung wird kaum eingeschränkt. Da außerdem tiefe Netzarchitekturen auf den zur Evaluierung genutzten Datensätzen CIFAR-10 und CIFAR-100 [Kri12] bessere Ergebnisse erzielen, ist der Rechenaufwand dieses Algorithmus folglich so hoch, dass der Implementierungsansatz stark auf Parallelisierung ausgelegt ist.

Sowohl der in "Neural Architecture Search with Reinforcement Learning" von Zoph und Le [ZL16] vorgestellte Algorithmus als auch der in dieser Arbeit vorgeschlagene Ansatz findet datenspezifische Modellarchitekturen. In ihrem Algorithmus werden die Modellparametrisierungen pro Iteration jedoch nicht durch evolutionäre Techniken angepasst, sondern es wird gelernt, mithilfe von Reinforcement Learning stetig bessere Modellparameter zu produzieren.
Während sich unsere Arbeit auf Zeitreihendaten spezialisiert, fokussiert sich die Veröffentlichung "Large-Scale Evolution of Image Classifiers" von Real u. a. [Rea+17] auf die Problemdomäne Bilderkennung. Somit gibt es einen von uns abweichenden Architekturfokus durch die ausschließliche Verwendung von Convolutional Schichten als Hauptschichten. Desweiteren wird keine Kreuzung und eine andere Selektionsimplementierung genutzt. Die Gemeinsamkeiten mit unserem Ansatz sind unter Anderem, ganze Schichten als Gene des Genoms zu nutzen und das Hinzufügen oder Entfernen ganzer Schichten zu erlauben und somit variabel in der Modelllänge zu sein. Obwohl wir zusätzlich zu Convolutional Schichten auch Dense und LSTM Schichten nutzen, decken wir durch unsere Beschränkung auf maximal 7 Hauptschichten einen kleineren Problemraum ab.

4 Vorgeschlagener Algorithmus

Unser Algorithmus soll für einen Zeitreihendatensatz eine Architektur für ein Tiefes Neuronales Netz finden, das besonders gut darin ist, Klassifizierungen zu erlernen.
Bevor wir Architekturen anwenden können, müssen Daten geladen und verarbeitet werden. Dieser Schritt hat einen erheblichen Einfluss auf das Ergebnis. In Abschnitt 4.1 geben wir dazu einen Programmüberblick.
In Abschnitt 4.2 widmen wir uns dem eigentlichen Algorithmus. Zuerst definieren wir den Problemraum, in welchem sich der Algorithmus bewegt. Anschließend geben wir einen Überblick über den Ablauf der evolutionären Suche.
Im Unterkapitel *Evolutionäre Techniken* legen wir dar, wie wir Selektion, Mutation und Kreuzung in unserem Algorithmus umgesetzt haben.
Abschließend erläutern wir Details der Implementierung.
Der Quellcode unserer Implementierung ist in diesem Github Repository[1] öffentlich verfügbar.

4.1 Programmüberblick

Daten laden

Unsere Pipeline, visualisiert in Abbildung 4.1, kann mit Zeitreihendaten aus jedem Kontext arbeiten. Wir erwarten eine CSV Datei, welche für jeden aufgenommenen Datenpunkt folgende Einträge zur Verfügung stellt:

- **Messdaten:** Die zu klassifizierenden Daten. Zum Beispiel Beschleunigungs- oder Rotationsdaten.
- **Art der Aktivität:** Diese Angabe ordnet dem Datenpunkt ein Label zu, beispielsweise eine Aktivität.
- **Kontext:** Zum Beispiel die Nummer der Aufnahme oder die Nummer des aufgenommenen Subjekts. Mit dieser Angabe ist es möglich, eine Teilung in Test- und Trainingsdatensatz vorzunehmen. Mithilfe dieser Angabe kann

1 https://github.com/Deep-Learning-as-a-Service/EvA

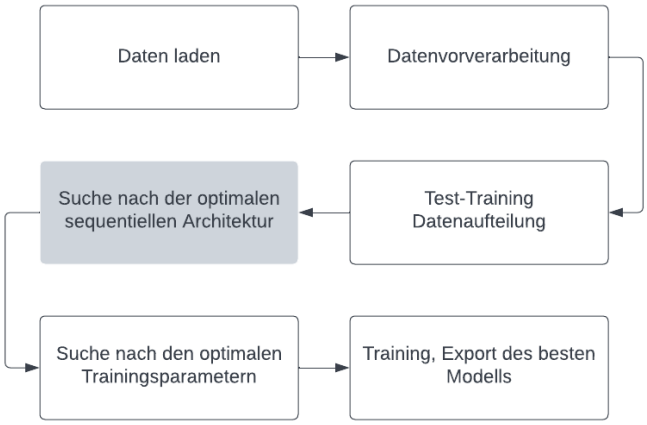


Abbildung 4.1: Flussdiagramm des implementierten Algorithmus

validiert werden, ob das Modell auch kontextunabhängig den Sachverhalt erlernen kann.

Datenvorverarbeitung

1. **Fenster zuschneiden** Tiefe Neuronale Netze haben eine definierte Eingabegröße, Zeitreihendaten können jedoch beliebig lang sein. Darum schneiden wir diese in Abschnitte, welche die Eingabegröße erfüllen. Dieser Prozess wird auch *windowing* genannt.

2. **Lineare Interpolation** In Zeitreihendaten gibt es immer wieder fehlende Datenpunkte, welche durch Übertragungsfehler verursacht werden. Die fehlenden Datenpunkte werden annähernd rekonstruiert, indem wir ihnen auf Basis angrenzender Messwerte einen Zwischenwert zuweisen.

Test-Training Datenaufteilung

Wir teilen die Daten in fünf gleichgroße Teile mit möglichst gleichverteilten Subjekten. Unsere Fitnessfunktion verwendet diese, um in fünf Auswertungsschritten eine möglichst präzise Aussage über die Güte eines Modells zu machen. Wir vertiefen die Test-Training Datenaufteilung in Abschnitt 4.2.3.

Suche nach der optimalen sequentiellen Architektur

In diesem Schritt sucht der von uns vorgeschlagene, evolutionäre Algorithmus ein Modell, welches auf den gegebenen Daten eine möglichst hohe Genauigkeit erzielt. Die konkrete Umsetzung des Algorithmus beschreiben wir in Abschnitt 4.2.

Suche der optimalen Trainings-Parameter

Für die beste gefundene Modell-Architektur versuchen wir hier die besten Trainings-Parameter zu finden, um die Genauigkeit des Modells weiter zu verbessern. Dafür verwenden wir die Bibliothek Optuna [Aki+19].

Training, Export des besten Modells

Zum Abschluss trainieren wir unsere beste Konfiguration auf allen uns zur Verfügung stehenden Daten. Das daraus exportierte TensorFlow-Modell [Aba+16] wird als Ergebnis unserer Pipeline ausgegeben.

4.2 Evolutions-Algorithmus

4.2.1 Umfang der evolutionären Suche

Da die Suche nach einer optimalen Modellarchitektur eine umfangreiche Parametermenge umfasst, haben wir den Problemraum eingeschränkt. Wir befassen uns ausschließlich mit sequentiellen Modellarchitekturen mit 2-7 Schichten. Ein enstehendes Modell darf nicht mehr als 30 Millionen trainierbare Parameter beinhalten. Desweiteren werden nur die Hauptschichten Convolutional2D, Dense und LSTM genutzt. Für jede dieser Schichten ist eine Parametermenge definiert. Die Wertebereiche, Standardwerte und Verteilungen der einzelnen Parameter je Schicht sind in Tabelle 4.1 zu finden. Die kategorischen Werte des *Stride* Parameters der Convolutional2D Schichten sind hierbei prozentual auf die Kernelgröße derselben Schicht bezogen. Die Parameter *Max Pooling* und *Batch Normalisierung* der Convolutional2D Schicht und *Dropout* der Dense Schicht sind eigenständige Schichten (Nebenschichten), die je nach Wert an die Hauptschicht angehängt werden.

Parametername	*Wertebereich*	*Standardwerte*	*Verteilungsfunktion*	*Verteilungs-eigenschaften*
Dense Schicht				
Units	16 - 1024	32, 64, 128, 256, 512	Normalverteilung	Durchschnitt: 514, Standardabweichung: 2000
Dropout	0.0, 0.1, 0.15, 0.2, 0.25, 0.3	0.0, 0.1	Kategorische Verteilung	0.0: 50%, andere: 10%
LSTM Schicht				
Units	2 - 512	8, 64, 256	Normalverteilung	Durchschnitt: 256, Standardabweichung: 300
Dropout	0.0, 0.1, 0.15, 0.2, 0.25, 0.3	0.0, 0.1	Kategorische Verteilung	0.0: 50%, andere: 10%
Convolutional2D Schicht				
Filter	16 - 256	32, 64, 128	Normalverteilung	Durchschnitt: 64, Standardabweichung: 200
Kernelgröße	(x,y) mit $1 \leq x, y \leq 10$	(1,1), (3,3), (5,5)	Normalverteilung	Durchschnitt: 3, Standardabweichung: 8
Stride	(x,y) mit x,y ∈ {1, 25%, 50%, 75%, 100%}	(1,1), (100%, 1)	Kategorische Verteilung	1: 60%, andere: 10%
Dropout	0.0, 0.1, 0.15, 0.2, 0.25, 0.3	0.0, 0.1	Kategorische Verteilung	0.0: 50%, andere: 10%
Max Pooling	None, (2,2), (4,4)	None, (2,2)	Kategorische Verteilung	None: 50%, (2,2): 30%, (4,4): 20%
Batch Normalisierung	True, False	True, False	Kategorische Verteilung	True: 20%, False: 80%

Tabelle 4.1: Umfang der ausgewählten Parameter und Schichten

4.2.2 Überblick

Als Eingabe bekommt der Algorithmus, siehe Algorithmus 1, eine Evolutionskonfiguration. Aus dieser geht die Anzahl der Generationen und die Anzahl an Individuen pro Generation hervor. Wir bestimmen darin, welche Evolutionstechniken in welchem Verhältnis angewandt werden sollen, um aus einer alten Generation eine neue Generation zu erzeugen.
Der Algorithmus bekommt einen Datensatz, anhand dessen er die Fitness seiner Individuen bewertet.
Über die Generationen hinweg speichert der Algorithmus das beste bisher gesehene Indiviuum. Nach der letzten Generation wird er dieses Individuum als Ergebnis der Evolution ausgeben.

Algorithmus 1: Sequentielle Evolution

Input: In der Evolutionskonfiguration sind die Anzahl an Generationen *n_generations* und die Anzahl an Individuen pro Generation *pop_size* definiert. Darüber hinaus ist der Datensatz gegeben.
Output: Das beste gefundene Individuum

```
1  P ← Initialisiere eine Generation aus zufälligen Individuen der Größe pop_size;
2  best_individual ← Initialisiere zufällig ein Individum als bestes Individuum P[0]
3  while i < n_generations do
4      for p ∈ P do
5          p_model ← Wandle Individuum p in ein ausführbares Modell der Keras Bibliothek
             um
6          Berechne Fitnesswert von p_model auf Basis der Daten
7          p ← speichere die Fitness
8      P ← Sortiere die Population absteigend nach Fitness
9      if Fitness von P[0] > Fitness von best_individual then
10         best_individual ← P[0];
11     P ← Erzeuge mit der alten Generation P und best_individual eine neue Generation der
         Größe pop_size. Anwendung der Evolutions-Techniken entsprechend der
         Evolutionskonfiguration
12 return best_individual
```

4.2.3 Evolutionäre Techniken

Aus den evaluierten Individuen der letzten Generation und dem bisher besten Individuum sollen die Individuen der nächsten Generation erstellt werden. Es wird entschieden, welche Individuen weiterentwickelt oder verworfen werden, ähnlich wie in der biologischen Selektion. Erfolgreiche Individuen dürfen bevorzugt ihr Erbmaterial an die nächste Generation weitergeben.

Für die Kreation neuer Individuen kommen die Evolutionstechniken Mutation und Kreuzung zum Einsatz. Neben der Weiterentwicklung von Individuen der letzten Generation werden in jeder Generation auch Weiterentwicklungen des global besten Individuums der Evolution (*Feinabstimmung des besten Individuums*), sowie zufällige Individuen inkludiert. Bei der Wahl der Zusammensetzung verschiedener Kreationsarten ist es wichtig, eine Balance zwischen Mikro- und Makrooptimierung zu finden. Auf der einen Seite ist es sinnvoll, an den besten Individuen kleine Veränderungen vorzunehmen, um eine höhere Genauigkeit zu erzielen, auf der anderen Seite gilt es zu vermeiden, dass die neue Generation zu gleichartig ist und der Algorithmus somit in einem lokalen Optimum verbleibt. Es besteht die Chance, dass eine gänzlich unterschiedliche Modellstruktur existiert, die eine höhere Genauigkeit erzielt.
Im Folgenden wird erläutert, wie die evolutionären Techniken dafür umgesetzt wurden.

Selektion

Gegeben eine Menge von Genomen der letzten Generation ist nun eine Teilmenge ebendieser Genome so auszuwählen, dass Genome mit höherer Fitness bevorzugt werden. Diese ausgewählten Genome werden im Folgenden als *Elterngenome* bezeichnet. Um die Diversität der Individuen nicht zu stark einzuschränken, sollen auch Genome mit geringerer Fitness beibehalten werden können, jedoch mit geringerer Häufigkeit. Daher wird folgender Algorithmus zur Auswahl der Elterngenome genutzt.

Sei X die Menge aller Genome der letzten Generation mit $|X| = n, n \in \mathbb{N}$, woraus $2 \leq k \leq n, k \in \mathbb{N}$ Elterngenome ermittelt werden sollen. Zuerst werden alle Fitnesswerte der Genome so normalisiert, dass $\sum_{i=0}^{n} \text{fitness}(X_i) = 1$. Anschließend wird eine zufällige Zahl $rand \in [0, 1]$, $rand \in \mathbb{R}$ generiert und das erste Genom, dessen normalisierte Fitness - addiert zu den vorher betrachteten, normalisierten Fitnesswerten - größer-gleich $rand$ ist, wird ausgewählt. Solange wir weitere

Elterngenome benötigen, wiederholen wir den Algorithmus, beginnend bei der Fitnessnormalisierung.
Somit werden Genome mit höherer Fitness bevorzugt ausgewählt, Genome mit geringerer Fitness können jedoch mit einer geringeren Wahrscheinlichkeit auch selektiert werden.

Kreuzung

Nach dem Ermitteln der Elterngenome für die Nachfolgegeneration müssen nun neue Individuen aus ebendiesen Elterngenomen erzeugt werden. Eine Variante hierfür ist die Kreuzung zweier Elterngenome. Für den vorgeschlagenen Algorithmus sind die folgenden zwei Kreuzungsalgorithmen implementiert.

Mittelpunkt Kreuzung Der erste Kreuzungsalgorithmus ist die Ein-Punkt Kreuzung [Sys89], für diesen Algorithmus expliziter die Mittelpunkt Kreuzung.
Seien zwei Elterngenome E_1, E_2 mit $|E_1| = n_1, |E_2| = n_2$ gegeben, wobei $n_1, n_2 \in [2, 7]$ hierbei der Anzahl der Schichten der jeweiligen Genome entspricht. Es wird nun der jeweilige Schnittpunkt beider Elterngenome ermittelt, expliziter die Mittelpunkte $m_1 = \lfloor \frac{n_1}{2} \rfloor, m_2 = \lfloor \frac{n_2}{2} \rfloor$. Anschließend wird mittels einer Pseudozufallszahl bestimmt, welches der Elterngenome für die erste, respektive zweite Hälfte des Kindgenoms K übernommen wird, siehe Abbildung 4.2.

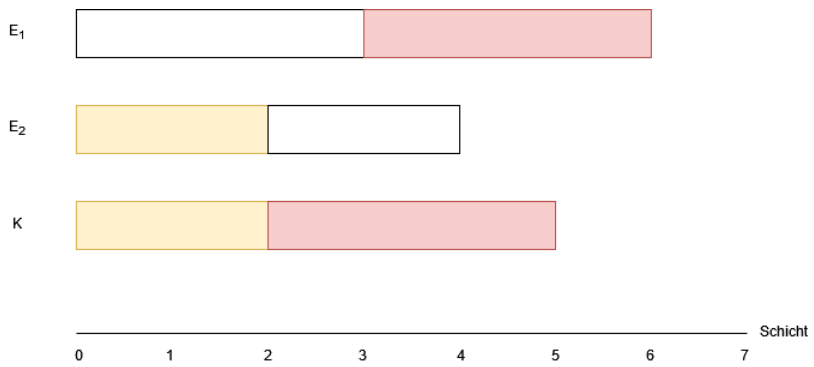


Abbildung 4.2: Mittelpunkt Kreuzung von Elterngenomen E_1 und E_2 zu resultierendem Kindgenom K. Die farblich markierten Teilbereiche sind die Hälften der Elterngenome, aus denen sich K zusammensetzt .

Uniforme Kreuzung Der zweite Kreuzungsalgorithmus, der in dieser Arbeit Verwendung findet, ist die Uniforme Kreuzung [Sys89], visualisiert in Abbildung 4.3. Seien auch hier zwei Elterngenome E_1, E_2 mit $|E_1| = n_1, |E_2| = n_2$ gegeben, wobei $n_1, n_2 \in [2, 7]$ als Anzahl der Schichten der jeweiligen Genome definiert ist. Bis inklusive Schicht $n_{min} = min(n_1, n_2)$ wird nun pseudozufällig zwischen den Schichten der Elterngenome ausgewählt, welche Schicht dem Kindgenom K zugewiesen wird. Exzessive Schichten werden pseudozufällig vom Kindgenom übernommen oder verworfen.

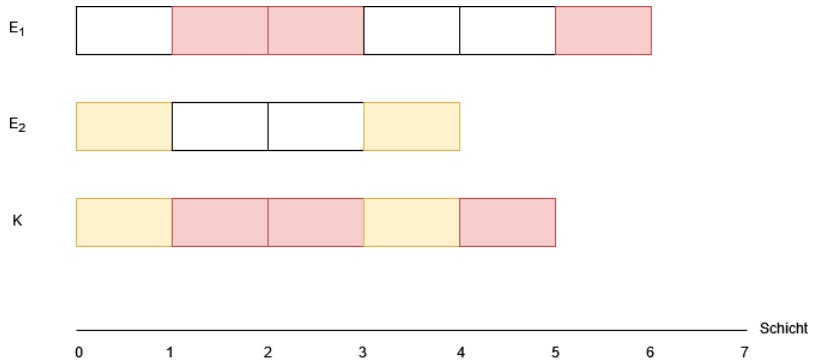


Abbildung 4.3: Uniforme Kreuzung von Elterngenomen E_1 und E_2 zu resultierendem Kindgenom K. Die farblich markierten Schichten sind die pseudozufällig ausgewählten Teile, die von dem Kindgenom übernommen werden.

Mutation

Eine weitere Methodik, um neue Kindgenome aus der vorherigen Generation zu erzeugen, ist Mutation. Gegeben ein Elterngenom, kann dieses nun mit gegebener Intensität mutiert werden und somit in einem neuen Genom resultieren, das entsprechend der Mutationsintensität Ähnlichkeiten mit seinem Elterngenom aufweist. Es wird zwischen den Intensitäten *leicht*, *mittel*, *hoch* und *gesamt* unterschieden. Zu jeder Intensität gibt es eine festgelegte Wahrscheinlichkeitsverteilung für die Anwendung von Mutationstechniken. Die Mutation eines Elterngenoms zu einem Kindgenom verläuft somit, grafisch in Abbildung 4.4 dargestellt, wiefolgt.

Strukturmutation Gegeben ein zu mutierendes Elterngenom und die Intensität der Mutation wird mit einer spezifischen Wahrscheinlichkeitsverteilung der gegebenen Intensität eine Strukturveränderung des Genoms angewendet. Unterschieden

wird hier zwischen der *Entfernung einer zufälligen Schicht*, dem *Hinzufügen einer zufälligen Schicht an zufälliger Position* und *keiner Strukturveränderung*.

Schichtmutation Anschließend wird über die Schichten des Genoms iteriert und mit einer weiteren spezifischen Wahrscheinlichkeitsverteilung der gegebenen Intensität eine Schichtveränderung durchgeführt. Die möglichen Veränderungsvarianten sind in diesem Schritt *Änderung des Schichttypen*, beispielsweise von Dense zu LSTM, *keine Schichtveränderung* und *Parametermutation der Schicht*.

Parametermutation Die Parametermutation hat wiederum eine eigene spezifische Wahrscheinlichkeitsverteilung der gegebenen Intensität. In diesem Schritt wird nun für jeden Parameter der momentanen Schicht entschieden, ob ebendieser Parameter *leicht*, *mittel*, *hoch* oder *gesamt* mutiert wird. Jeder Parameter bewegt sich in einem bestimmten Intervall von zugelassenen Werten, siehe Tabelle 4.1. Einerseits sollen neuartige, unbekannte Modellarchitekturen zugelassen werden, somit sollen die Intervalle möglichst groß sein. Andererseits soll der Problemraum nicht zu groß sein, um Optima in angemessener Zeit auffinden zu können. Wenn der Großteil der produzierten Genome keine Chance auf eine hohe Genauigkeit hat, ist möglicherweise kein zufriedenstellendes Optimum zu erreichen. Um sich dieser Balance differenziert anzunähern, sind Verteilungsfunktionen implementiert. Während die Parameterintervalle groß bleiben, werden neue Mutationswerte entsprechend einer definierten Normalverteilung vergeben. Mit Erwartungswert und Varianz können konventionelle Parametrisierungen mit hoher Frequenz gewählt werden, damit dieser Teil des Suchraums bevorzugt abgearbeitet wird. Gleichzeitig soll nicht ausgeschlossen werden, dass auch ungewöhnliche Extremwerte vorkommen können, da diese möglicherweise einen relevanten Beitrag zur Evolution leisten. Desweiteren ist der maximale Abstand zwischen originalem und mutiertem Wert abhängig von der gegebenen Mutationsintentität. Für kategorische Parameter wird eine kategorische Verteilungsfunktion genutzt, die jedem möglichen Wert des Wertebereichs eine Gewichtung zuordnet. Diese Gewichtung entspricht der Wahrscheinlichkeit, bei einer Parametermutation ausgewählt zu werden. Da für kategorische Parameter kein maximaler Abstand von Original- zu Mutationswert definiert werden kann, werden kategorische Parameter nur zu einer von der Mutationsintensität abhängigen Wahrscheinlichkeit mutiert.

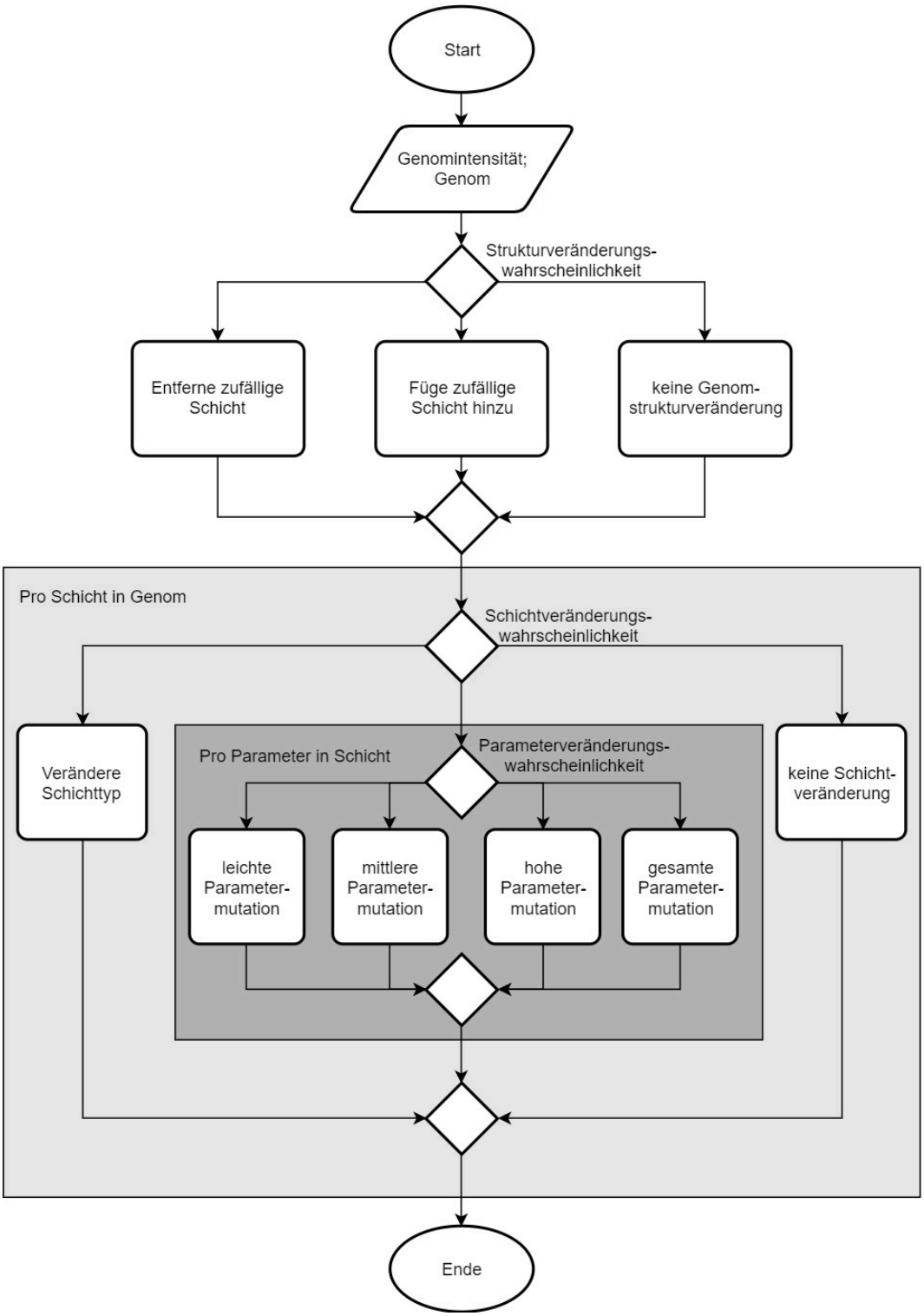


Abbildung 4.4: Flussdiagramm der Genommutation mit gegebener Intensität. Die Struktur-, Schicht- und Parameterveränderungswahrscheinlichkeiten sind abhängig von der Intensität, mit der das Genom mutiert werden soll

Fitness

Um einschätzen zu können, welche Genome einem guten oder schlechten Lösungsansatz entsprechen, müssen wir eine Funktion definieren, anhand derer Individuen der Evolution bewertet werden. Wir haben uns für eine *5-Fold Kreuzvalidierung* (5-Fold Cross Validation) entschieden, visualisiert in Abbildung 4.5, da diese Vorgehensweise auf den in dieser Arbeit genutzten Daten eine angemessene Balance zwischen Schwankungsausgleichung von Test- und Trainingsdatenaufteilungen und Zeitaufwand bringt. In der Grafik sind die 5 Folds visualisiert, wobei eine Säule den gesamten Datensatz darstellt. Der grüne Teil eines Folds ist der Trainingsdatensatz; der Testdatensatz des Folds ist rot eingefärbt.

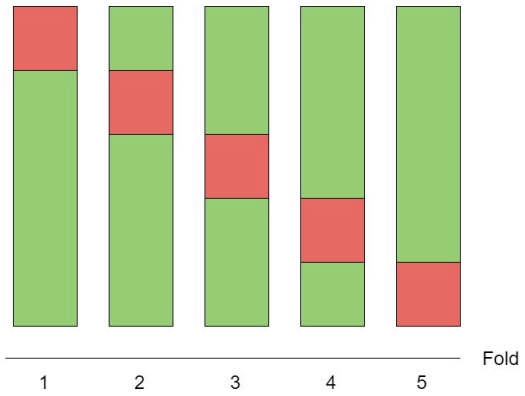


Abbildung 4.5: 5-Fold Cross Validation Datenaufteilung
grün: Trainingsdaten, rot: Testdaten

Desweiteren wird die Aufteilung auf Aufnahme Ebene durchgeführt (Leave-Recording-Out). Dadurch wollen wir ein möglichst realistisches Szenario erzeugen, in dem eine neue Aufnahme bewertet werden soll. Das Modell kann somit nicht auf spezifische Wertebereiche von Aufnahmen oder andere aufnahmespezifische Eigenheiten overfitten.
Da das Training auf personenspezifischen Merkmalen gewollt ist, verwenden wir die *StratifiedKFold* Klasse der Scikit-Learn Bibliothek [Ped+12] auf Subjekten. Hierdurch entstehen die 5 Folds so, dass möglichst in jeder der 5 Aufteilungen alle Subjekte sowohl im Test- als auch Trainingsdatensatz vorkommen.

Wir trainieren somit 5 Instanzen des zu bewertenden Modells auf je einem Fold mit jeweils 5 Epochen, einer Lernrate von 0.001 und einer Batchgröße von 32 und geben die durchschnittliche Genauigkeit der 5 Modelle als endgültigen

Fitnesswert zurück.

Wird eine Modellarchitektur von der Fitnessfunktion bewertet, dann wird der resultierende Fitnesswert über den gesamten Evolutionsverlauf gespeichert. Tritt dieselbe Modellarchitektur erneut auf, kann der bereits berechnete Fitnesswert abgefragt und verwendet werden, was zu einer erhöhten Laufzeiteffizienz führt.

4.2.4 Implementierung

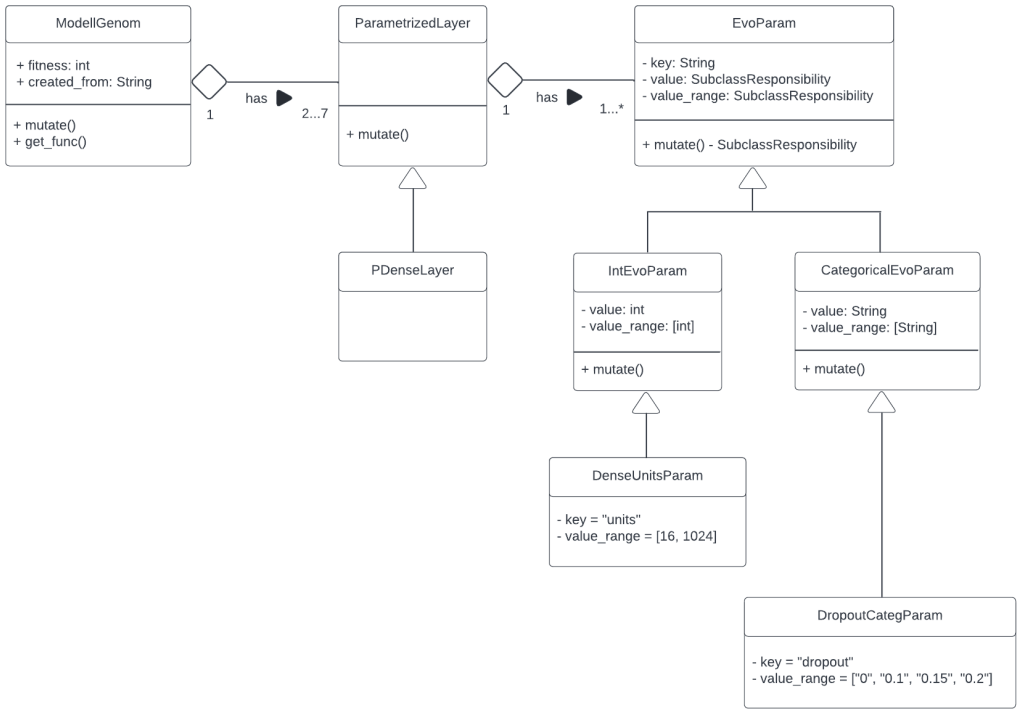


Abbildung 4.6: UML-Diagramm eines Ausschnitts der Softwarearchitektur

Anforderungen Für die Implementierung des Evolutions-Algorithmus brauchen wir ein Datenformat, mit dem wir Modelle im Code beschreiben. Diese Darstellung muss alle Informationen enthalten, um daraus ein ausführbares Modell zu kreieren. Es soll auf einfache Weise möglich sein, die evolutionären Techniken Mutation und Kreuzung anzuwenden. Darüber hinaus soll der evolutionäre Kontext erfasst werden. Das meint die Technik, mit der das Modell erstellt wurde und die Genauigkeit die es erreicht hat.

Klasse ModellGenom Naheliegend wäre, die Modellklasse der Keras Bibliothek zu verwenden. Schließlich wird spätestens für das Training der Architektur ein Keras-Modell benötigt. Einmal kompiliert, ist die Konfiguration eines Keras-Modells gesetzt. Es ist umständlich, die Schichten und Parameter aus diesem Format auszulesen, um neue Modelle zu kreieren. Stattdessen haben wir uns entschieden, eine

ModellGenom Klasse zu schreiben, welche eine eigene Darstellung der Modellparameter besitzt. Erst für die Anwendung wird das ModellGenom in ein Keras-Modell übersetzt. So haben wir die Freiheit, Kontextinformationen der Evolution zu speichern und eine generische Darstellung von Schichten und deren Parametern zu bauen, welche die Evolutionstechniken unterstützen. In Abbildung 4.6 haben wir diesen Ausschnitt unserer Softwarearchitektur als UML-Diagramm dargestellt.

Evolutionärer Kontext im ModellGenom

Instanzen der ModellGenom Klasse haben eine Instanzvariable *fitness*, welche die Güte des Genoms speichert. Sie besitzt außerdem eine Variable *created_from*, die auf die evolutionäre Technik hinweist, mit der das Genom gebaut wurde.
Die evolutionäre Herkunft des Modells wird bei der Initialisierung gespeichert. Diese Angabe ist vor allem für die Analyse der Evolution interessant. Die Zuordnung von Evolutiontechnik zur Güte der Genome gibt Aufschluss darüber, wie erfolgreich die Techniken waren.

Beschreibung des UML Diagramms Jedes ModellGenom hat eine Liste von Instanzen der Klasse *ParametrizedLayer*. Diese beschreibt sequentiell das Vorkommen der Hauptschichten im Modell, welche wir im Unterabschnitt 4.2.1 bereits eingeführt haben. Jede Schicht hat wiederum selbst eine Liste von Instanzen der Klasse *EvoParam*, welche die Parameter der jeweiligen Schicht beschreiben. Beispielsweise bei der Dense Schicht kann das der Units Parameter und der Dropout Parameter sein. Jeder Parameter hat einen *key* und einen *value* Wert. Der Units Parameter wird mit seinem Schlüssel direkt in einen Parameter der Dense Schicht aus der Keras-Bibliothek übersetzt. Der Dropout Parameter hingegen hat kein Äquivalent in der Dense Schicht der Keras-Bibliothek. Für diesen Parameter setzen wir nach der Hauptschicht eine Dropout Keras Schicht. Unser Algorithmus versucht gute Kombinationen und Parametrisierungen von Schichten zu finden. Dropout Schichten stehen dabei nicht allein für sich. Sie lernen nicht, sondern sind erst in Zusammenhang mit lernenden Schichten hilfreich. Weitere Beispiele für Schichtparameter wären die Aktivierungsschicht oder die Batch Normalisierungs Schicht. Mit unserer Architektur haben wir die Möglichkeit, zwischen Haupt- und Nebenschichten zu unterscheiden, unabhängig von der Keras-Bibliothek.

Mutation des ModellGenoms

Ablauf Mutieren wir ein Genom, dann wird zunächst die strukturelle Mutation der Schichten umgesetzt. Dies geschieht im ModellGenom, weil dieser Zugriff

auf die vollständige Liste hat. Anschließend werden die Parameter der Schichten mutiert. Im ModellGenom wird von der konkreten Implementierung abstrahiert. Die Aufgabe wird an die Schichten selbst weitergegeben. Diese entscheiden sich entweder dafür, keine Mutation durchzuführen oder die Aufgabe an ihre Parameter weiterzugeben. Auch Parameter wiederum haben die Möglichkeit die Mutation zu ignorieren oder diese entsprechend ihres Datentyps auf ihrem aktuellen Wert umzusetzen

Parametermutation Die Mutationsimplementierungen sind innerhalb eines Datentyps jeweils gleich. Deshalb haben wir Klassen für die Parameterdatentypen angelegt und lassen die Schichtparameter davon erben.
Die *value_range* eines Integers, wie dem Units Parameter, enthält ein Minimum und ein Maximum. In der Mutation wird einem Integer im Intervall um seinen aktuellen Wert entsprechend der Wahrscheinlichkeitverteilung ein neuer Wert zugewiesen.
Die *value_range* eines Categoricals wie Dropout enthält eine Liste möglicher Werte. In der Mutation wird dem Categorical entsprechend einer diskreten Wahrscheinlichkeitsverteilung ein neuer Wert zugewiesen.

Vom ModellGenom zum Keras-Modell

Damit ein ModellGenom zu einem kompilierbaren und ausführbaren Modell der Keras Bibliothek werden kann, sind folgende Schritte nötig.

Sicherstellung der Kompilierbarkeit Bevor ein ModellGenom die Schritte zur Übersetzung in ein Keras Modell gehen kann, muss überprüft werden, ob es überhaupt möglich ist, das Keras Modell zu erstellen. Hierzu wird die Klasse *ModelChecker* genutzt, die folgende Probleme behebt:

1. **Null-Dimension durch Convolutional2D Schichten** Wenn Convolutional2D Schichten hintereinander geschaltet werden, deren kombinierte Kernels und Strides dazu führen würden, dass eine Ausgabedatendimension null ist, kann das resultierende Keras Modell nicht erstellt werden.
   Lösungsansatz: Für die Lösung dieses Problems werden die Ausgabedimensionen aller Schichten des ModellGenoms simuliert. Solange eine der Schichten eine Nulldimension als Ausgabe hat, wird eine zufällige Convolutional2D Schicht durch eine andere Schicht, LSTM oder Dense, ersetzt.

2. **Memory Overflow** Sollte das ModellGenom viele komplexe Schichten mit hoher Parametrisierung haben, beispielsweise sieben Dense Schichten mit je

1024 Units, wird die Kreation des Keras Modells auf den meisten GPUs durch einen Speicherüberlauf (Memory Overflow) abbrechen.
Lösungsansatz: Auch für dieses Problem werden die Ausgabedimensionen aller Schichten und somit die Anzahl an benötigten Neuronen berechnet. Solange diese Zahl größer als 2^{30} ist, wird die Schicht mit der höchsten Komplexität, definiert durch die höchste *Units* Parametrisierung einer Dense oder LSTM Schicht, vor der betroffenen Schicht ausgewählt; diese Schicht erhält anschließend einen gefünftelten *Units* Parameter Wert.

3. **Hohe Komplexität** Um die benötigte Zeit der Evolution möglichst effizient zu nutzen, sollen keine Modelle mit mehr als 30 Millionen trainierbaren Parametern erstellt werden, da deren Training zu zeitaufwendig ist.
Lösungsansatz: Da die Probleme *Null-Dimension durch Convolutional2D Schichten* und *Memory Overflow* zu diesem Zeitpunkt bereits gelöst wurden, ist eine Konvertierung des ModellGenoms zu einem Keras Modell bereits möglich. Somit kann das entsprechende Keras Modell erstellt und anschließend die Anzahl trainierbarer Parameter durch Aufruf der Methode *count_params()* bestimmt werden. Solange diese Anzahl über 30 Millionen liegt, wird auch hier die komplexeste LSTM oder Dense Schicht gesucht und anschließend der Wert des *Units* Parameters gefünftelt.

Nachdem der *ModelChecker* die notwendigen Änderungen an dem ModellGenom vorgenommen hat, kann das ModellGenom in ein Keras Modell übersetzt werden.

Übersetzung zum Keras Modell Hierfür wird zuerst die *Input* Schicht der Keras Bibliothek vorgeschaltet, um die Eingabedimensionen festzulegen. Im Anschluss wird eine *Normalization* Schicht der Keras Bibliothek hinzugefügt, um die Eingabedatenwerte pro Feature zur Laufzeit zu normalisieren. Anschließend werden die Schichten des ModellGenoms in die entsprechenden Keras Schichten mit korrekter Parametrisierung übersetzt. Jede Modell Schicht bekommt hierfür die Dimensionen der Eingabedaten und es wird intern entschieden, ob die Anzahl der Dimensionen geändert werden muss. Hierfür werden die verlustfreien, dimensionsumformenden Operationen *Hinzufügen einer Channel Dimension* und textitEntfernen der Channel Dimension verwendet. Für den Übergang von Convolutional zu LSTM Schicht muss beispielsweise die Channeldimension in die Featuredimension aufgelöst werden.

Anschließend kann die Funktion *get_func()* auf der Schicht aufgerufen werden. Diese Funktion übersetzt die Schicht der Klasse *ParametrizedLayer* inklusive

aller zugehörigen Parameter der Klasse *EvoParam* in eine Keras Schicht. Dense und Convolutional2D Schichten haben des Weiteren *After Layer Params*. Diese Parameter sind keine Schichtparameter, sondern eigenständige Schichten, im weiteren Verlauf Nebenschichten genannt. Alle Hauptschichten - *Dense, LSTM, Convolutional2D* - haben außerdem eine eigene Aktivierungsschicht in Form einer *Rectified Linear Unit* (ReLU) [NH10]. Das Modell wird somit mit dieser Aktivierungsschicht und den Nebenschichten, wie *Batch Normalisierung* oder *Max Pooling*, ergänzt.

Nachdem alle Schichten zum Modell hinzugefügt wurden, wird eine *Flatten* Schicht gefolgt von einer Dense Schicht mit *Softmax* Aktivierungsfunktion angehangen, um die Ausgabe in Form von Wahrscheinlichkeiten pro Ausgabeklasse zu ermöglichen. Das Modell wird anschließend mit dem *RMSProp* Optimizer der Keras Bibliothek, einer Lernrate von 0.001 und der Verlustfunktion *Categorical Crossentropy* kompiliert.

5 Experimente

In dem folgenden Abschnitt werden wir die Ergebnisse unseres Algorithmus und die qualitative Einordnung der resultierenden Modelle auf verschiedenen Datensätzen aufzeigen.
Zur Ausführung der folgenden Experimente wurde eine Nvidia A40 48 GB GDDR6 mit 16 GB RAM und 8 CPU-Kernen genutzt. Alle Experimente wurden mit Python 3.9.0 und TensorFlow 2.7.0 ausgeführt.

5.1 Definition Modellarchitekturschreibweise

Um verschiedene Modellarchitekturen tabellarisch darzustellen, bietet sich die Nutzung der folgenden, abkürzenden Schreibweise an.
Eine Zeile entspricht einer Schicht; die Reihenfolge der Schichten des Modells ist von oben nach unten zu lesen. Da nur sequentielle Modelle betrachtet werden, ist diese Schreibweise eindeutig.
Die für den vorgestellten Algorithmus ausgewählten Schichten mit einer beispielhaften Parametrisierung, für die Übersicht unseres Umfangs siehe Tabelle 4.1, werden folgendermaßen geschrieben:

Convolutional2D

C(f=13, k=(2,2), s=(1,1), d=0.1, bn=True, mp=(2,2))

ist eine Convolutional2D Schicht mit 13 Filtern, (2,2) Kernel, (1,1) Stride, 0.1 Dropout und anschließender Batch Normalisierung und (2,2) Max Pooling Schicht.

LSTM

L(u=338, d=0.2)

ist eine LSTM Schicht mit 338 Units und 0.2 Dropout.

Dense

L(u=8, d=0.0)

ist eine Dense Schicht mit 8 Units ohne anschließender Dropout Schicht.

5.2 Datensätze

5.2.1 Opportunity

Der Opportunity Datensatz [Cha+13] besteht aus 20 Aufnahmesequenzen von 4 Subjekten, die mit mehreren Abstraktionsleveln von Aktivitäten des täglichen Lebens (ADL) gelabelt sind. Für unser Experiment beschränken wir uns auf 6 Bewegungsaktivitäten des höchsten Abstraktionslevels, aufgenommen in 30 Hz: *null*, *relaxing*, *coffee time*, *early morning*, *cleanup*, *sandwich time*. Zur Verfügung stehen hierzu 23 verschiedene am Körper des Subjekts angebrachte Sensoren, 12 Objektsensoren und 13 Umgebungssensoren. Die Positionen der IMU Sensoren sind in Abbildung 5.1 auf der linken Seite visualisiert; die folgenden Experimente beschränken sich ausschließlich auf die Daten der Sensorenteilmenge *RLA, LLA, BACK, R-SHOE, L-SHOE*. Somit belaufen sich die Dimensionen der Gesamtdaten auf 13958 Aktivitätsfenster mit je 90 Datenpunkten und 51 Features.

5.2.2 Pflegeaktivitäten

Zum Evaluieren der Möglichkeit, Pflegeaktivitäten mithilfe von Tiefen Neuronalen Netzen vorherzusagen, haben wir einen eigenen Datensatz mit 5 am Körper angebrachten Intertialsensoren - Rumpf, beide Handgelenke, beide Fußgelenke - aufgenommen. Eine Visualisierung der Sensorpositionen lässt sich in Abbildung 5.1 im rechten Teil auffinden. Der Datensatz besteht aus 60 Hz Aufnahmen von 10 Individuen, die 14 Pflegeaktivitäten nachstellen, unter anderem *bett machen*, *medikamente stellen*, *umkleiden* und *gesamtwaschen im bett*. Nach der Aufteilung der Datenpunkte in Aktivitätsfenster betragen die Dimensionen der gesamten Daten 32931 Aktivitätsfenster mit 90 Datenpunkten und 70 Features.

5.3 Vergleichsmodelle

Um die Güte der evolutionären Modelle einschätzen zu können, vergleichen wir die beste gefundene Architektur mit den folgenden drei Modellen, die jeweils entweder Standardmodelle sind oder explizit für die Aktivitätserkennung entwickelt wurden.

CNNLSTM Das erste Modell entspricht der von Sainath u. a. [Sai+15] entwickelte Architektur *CNNLSTM*. CNNLSTM ist eine sequentielle Standardarchitektur für

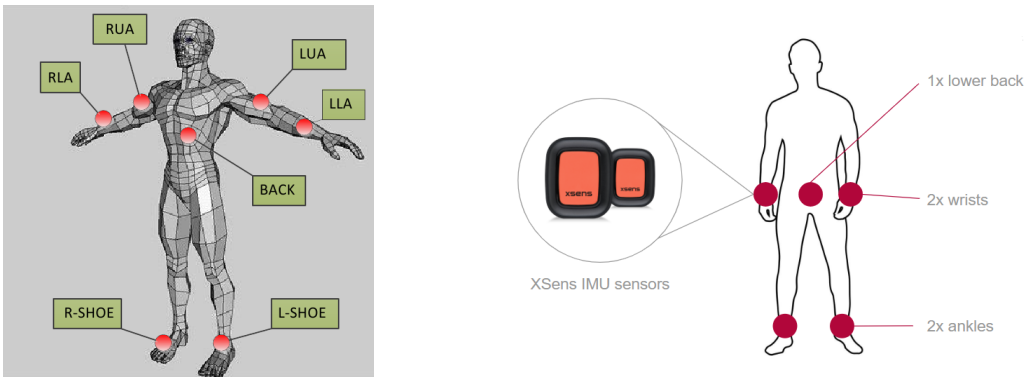


Abbildung 5.1: Sensorpositionen des Opportunity (l) [Bild aus Cha+13, S. 2] und Pflegeaktivitäten (r) Datensatzes. Für die folgenden Experimente auf dem Opportunity Datensatz wurde nur die Sensorenteilmenge *RLA, LLA, BACK, R-SHOE, L-SHOE* verwendet.

die Domäne *Zeitreihendaten*, wie der Verarbeitung natürlicher Sprache oder auch Aktivitätserkennung.

ResNet *Residual Neural Networks*, oder auch *ResNet* [He+15] ist eine nicht-sequentielle Modellarchitektur, deren Hauptbestandteil in *Residual Blöcken* liegt, welche es zusätzlich zu normalen Schichten erlauben, Neuronenverbindungen zu überspringen. ResNets gehören auch zu den Standardmodellen der Tiefen Neuronalen Netze.

InnoHAR Die von Xu u. a. [Xu+19] publizierte Modellarchitektur *InnoHAR* wurde speziell zur Erkennung menschlicher Aktivitäten auf Basis von Sensorendaten entwickelt. Die nicht-sequentielle Architektur erlaubt sowohl räumliche als auch zeitliche Feature Extraktion durch *Inception Blöcke* in Verbindung mit LSTMs und Gated Recurrent Units (GRUs). InnoHAR ist eine State-of-the-Art Architektur im Bereich Aktivtätserkennung.

5.4 Setup

Für die Hyperparameter des evolutionären Algorithmus, der Datenaufbereitung und der Fitnessfunktion, anhand derer Individuen der Evolution bewertet werden, wurde die in Tabelle 5.1 aufgelistete Auswahl getroffen. Aufgrund des zeitlichen Aufwands des in dieser Arbeit präsentierten Algorithmus beschränken sich die Experimente auf genau ein Setup.

Datenaufbereitungsparameter	
Interpolationstechnik	Lineare Interpolation
Aktivitätsfenstergröße	90 Datenpunkte (3s Opportunity, 1.5s Pflegeaktivitäten)
Aktivitätsfenster Überlappung	50%
Normalisierungstechnik	Normalisierung mit Normalverteilung pro Feature mit Durchschnitt 0 und Standardabweichung 1
Evolutionsparameter	
Generationen	140
Individuen pro Generation	8
Techniken pro Generation	1 leichte Mutation 1 mittlere Mutation 1 hohe Mutation 1 gesamte Mutation 1 Feinabstimmung des besten Individuums 1 uniforme Kreuzung 1 Mittelpunkt Kreuzung 1 zufälliges Modell
Initiale Modelle	8 zufällige Modelle mit Standard Schichtparametrisierung
Fitnessparameter	
Fitness	5-Fold Cross Validation mit Leave-Recording-Out
Fitness Metrik	Durchschnittsgenauigkeit aller Folds
Trainingsepochen pro Modell	5
Lernrate	0.001
Batchgröße	32

Tabelle 5.1: Ausgewählte Hyperparameter der Experimente

5.5 Evolutionsverlauf

Im folgenden Abschnitt werden die Ergebnisse des Algorithmus auf den beiden Datensätzen Opportunity und Pflegeaktivitäten dargestellt. Die Parameter wurden wie in Abschnitt 5.4 beschrieben gewählt.

Opportunity

Der visualisierte Verlauf der Evolution lässt sich in Abbildung 5.2 betrachten. Der orangene Graph zeigt jeweils die höchste Fitness eines Individuums einer Generation, während der blaue Graph die höchste Accuracy eines Individuums bis zur derzeitigen Generation darstellt. Die benötigte Zeit für den Evolutions-

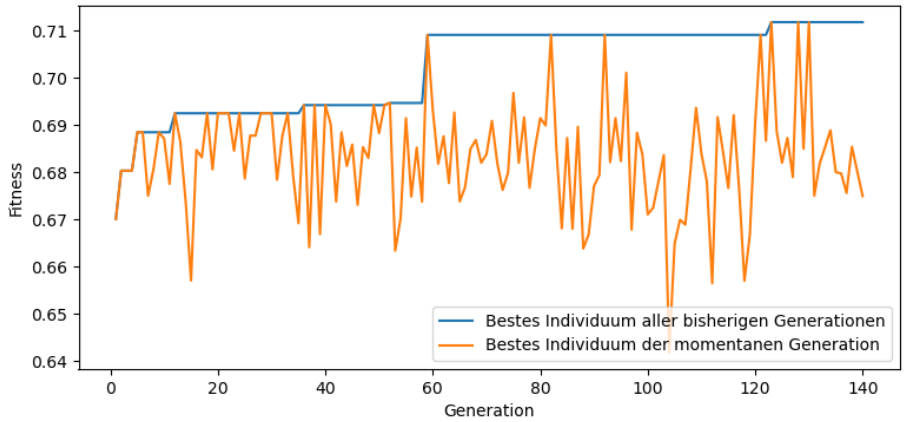


Abbildung 5.2: Evolutionsverlauf der besten Individuen auf dem Opportunity Datensatz

prozess beträgt 47 Stunden und 49 Minuten. Im Verlauf der Evolution gibt es 7 Modellverbesserungen, siehe Tabelle 5.2. $\Delta Fitness$ ist hierbei und im Folgenden als Verbesserung der Fitness im Vergleich zum vorherigen besten Individuum definiert. Das beste Individuum des Evolutionsprozesses erreicht einen Fitnesswert von 0.7118.

Generation	Technik	Architektur	$\Delta Fitness$
2	hohe Mutation	C(f=64, k=(3,3), s=(1,1), d=0.0, bn=True) D(u=334, d=0.3)	1.03%
5	Feinabstimmung des besten Individuums	C(f=39, k=(3,3), s=(1,1), d=0.0, bn=True) D(u=334, d=0.3)	0.81%
12	Feinabstimmung des besten Individuums	C(f=45, k=(3,3), s=(1,1), d=0.0, bn=True) D(u=381, d=0.3)	0.40%
36	Feinabstimmung des besten Individuums	C(f=45, k=(3,3), s=(1,1), d=0.0, bn=True) D(u=476, d=0.3)	0.17%
52	Mittelpunkt Kreuzung	C(f=39, k=(4,2), s=(1,1), d=0.0, bn=True) C(f=128, k=(1,1), s=(1,1), d=0.0, bn=True) D(u=64, d=0.3) L(u=8, d=0.0)	0.04%
59	Feinabstimmung des besten Individuums	C(f=43, k=(4,2), s=(1,1), d=0.0, bn=True) C(f=140, k=(1,1), s=(1,1), d=0.0, bn=True) D(u=177, d=0.0) L(u=8, d=0.0)	1.44%
123	Feinabstimmung des besten Individuums	C(f=16, k=(4,2), s=(1,1), d=0.0, bn=True) C(f=147, k=(1,1), s=(1,1), d=0.0, bn=True) D(u=194, d=0.0) L(u=68, d=0.0)	0.27%

Tabelle 5.2: Bestmodellverbesserungen im Verlauf der Evolution auf dem Opportunity Datensatz

Pflegeaktivitäten

Die Evolution auf dem Pflegeaktivitäten Datensatz verläuft wie in Abbildung 5.3 visualisiert. Über 140 Generationen gibt es 4 Verbesserungen, welche in Tabelle 5.3 aufgeschlüsselt sind. Die insgesamt benötigte Laufzeit für den Evolutionsprozess beträgt 68 Stunden und 20 Minuten. Das beste Individuum erreicht einen Fitnesswert von 0.5224.

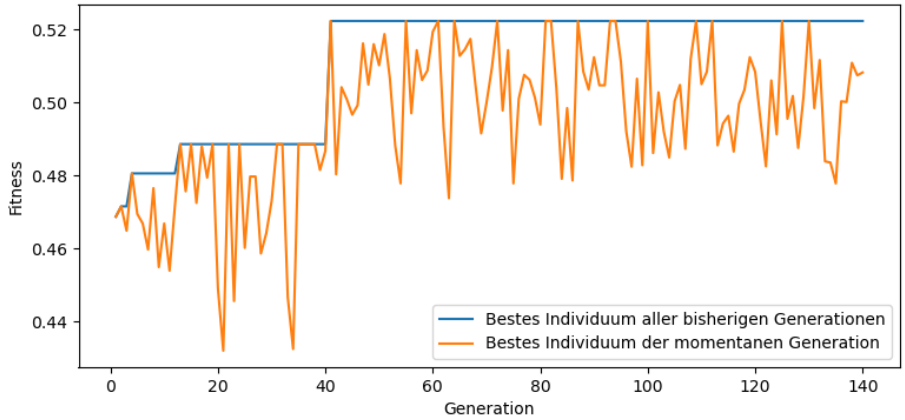


Abbildung 5.3: Evolutionsverlauf der besten Individuen auf dem Pflegeaktivitäten Datensatz

Generation	Technik	Architektur	$\Delta Fitness$
2	uniforme Kreuzung	C(f=64, k=(3,3), s=(1,1), d=0.1, bn=True) L(u=256,d=0.1)	0.28%
4	hohe Mutation	C(f=65, k=(3,3), s=(1,1), d=0.1,bn=True) L(u=182,d=0.0) L(u=223,d=0.1) D(u=34,d=0.0)	0.90%
13	gesamte Mutation	C(f=37, k=(5,5), s=(1,2), d=0.0, bn=True, mp=(2,2)) L(u=355, d=0.15) D(u=180, d=0.0)	0.80%
41	leichte Mutation	C(f=25, k=(9,1), s=(2,1), d=0.0, bn=True) L(u=98, d=0.1) D(u=612, d=0.0)	3.37%

Tabelle 5.3: Bestmodellverbesserungen im Verlauf der Evolution auf dem Pflegeaktivitäten Datensatz

5.6 Modellvergleich

Um die Qualität der Modellfindung mithilfe des vorgeschlagenen Algorithmus in Relation zur Nutzung von Standardmodellen zu stellen, haben wir das jeweils beste Modell der Evolutionen mit Modellen der Architekturen *CNNLSTM*, *ResNet* und *InnoHAR* verglichen. Alle Modelle wurden auf einem 5-Fold der Daten mit der Standardhyperparametrisierung von 5 Epochen, einer Lernrate von 0.001 und einer Batchgröße von 32 trainiert. Die einzelnen Folds wurden auf Aufnahme Ebene durchgeführt (Leave-Recording-Out), um beispielsweise das Merken von Wertebereichen oder anderen aufnahmespezifischen Eigenheiten zu verhindern und somit eine realistische Aussage über die Genauigkeiten der Modelle treffen zu können.

Da jedoch die Wahl von Hyperparametern eine wichtige Rolle für die Qualität eines Modells spielt, haben wir desweiteren anschließend die Hyperparameter *Epochen*, *Lernrate* und *Batchgröße* mithilfe von Optuna [Aki+19] optimiert. Hierfür wurde jedes Modell 20 Iterationen auf demselben 5-Fold der jeweiligen Daten mithilfe der Fitness Funktion bewertet und oben genannte Hyperparameter verbessert.

Die Ergebnisse dieser beiden Experimente sind für den Opportunity Datensatz in Abbildung 5.4 und für den Pflegedatensatz in Abbildung 5.5 aufzufinden. Die Architekturen der beiden Modelle *Unser bestes Individuum* sind in Tabelle 5.2 für Opportunity und in Tabelle 5.3 für den Pflegeaktivitäten Datensatz in der letzten Spalte aufgelistet. Die von Optuna optimierten Hyperparameter der einzelnen Modelle pro Datensatz sind in Tabelle 5.4 aufzufinden.

Zur Verifizierung ist der Quellcode zu diesen Experimenten hier[2] für Opportunity respektive hier[3] für den Pflegedatensatz zu finden. Durch die partiell zufällige Gewichtsinitialisierung der Netze und einer durch den Trainingsprozess entstehenden Parallelisierungzufälligkeit bei Nutzung einer GPU kommt es jedoch zu leichten Abweichungen.

2 https://github.com/Deep-Learning-as-a-Service/EvA/blob/master/src/experiments/five_fold_opp_bachelorarbeit_experiment.py
3 https://github.com/Deep-Learning-as-a-Service/EvA/blob/master/src/experiments/five_fold_lab_bachelorarbeit_experiment.py

Modell	*Epochen*	*Lernrate*	*Batchgröße*
Opportunity			
CNNLSTM	27	0.00139	52
ResNet	14	0.00096	22
InnoHAR	5	0.00101	49
Unser bestes Individuum	22	0.00026	23
Pflegeaktivitäten			
CNNLSTM	5	0.00117	25
ResNet	6	0.00127	22
InnoHAR	5	0.001	32
Unser bestes Individuum	5	0.00155	30

Tabelle 5.4: Von Optuna optimierte Hyperparameter

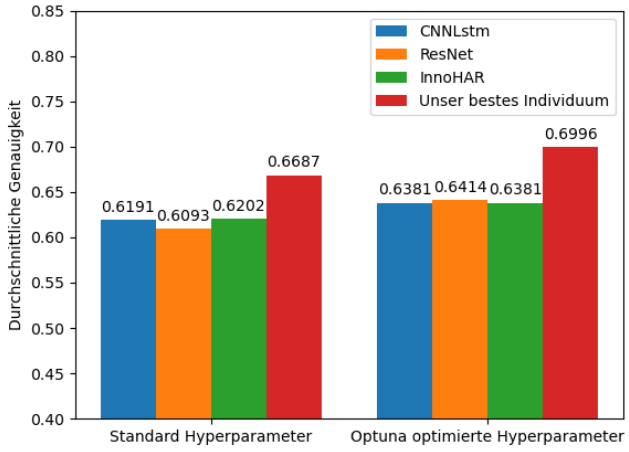


Abbildung 5.4: Opportunity Datensatz - Modellvergleich

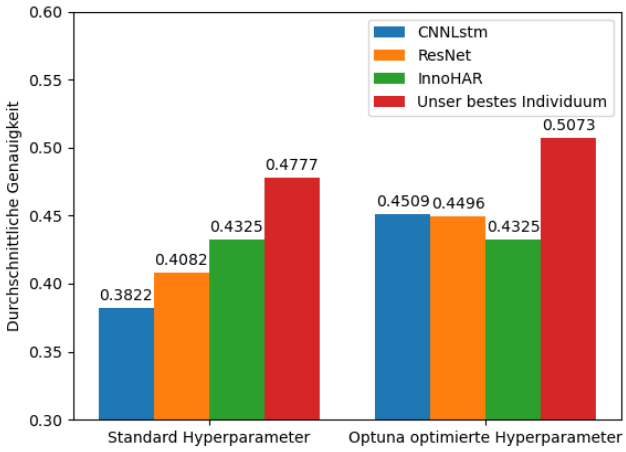


Abbildung 5.5: Pflegeaktivitäten Datensatz - Modellvergleich

6 Diskussion

Für beide Datensätze hat unser Algorithmus selbständig in jeweils weniger als drei Tagen Modellarchitekturen gefunden. Unser Programm schlägt die händische Suche im Kriterium Aufwand deutlich.
Um zu zeigen, dass unser Algorithmus eine sinnvolle Alternative ist, begründen wir, warum unser Experiment repräsentativ ist und bewerten unsere Ergebnisse.
Abschließend gehen wir darauf ein, welche evolutionären Techniken welchen Beitrag zum Ergebnis geliefert haben.

6.1 Repräsentativität des Experiments

Wahl der Datensätze Wir haben den Opportunity Datensatz gewählt, weil dieser von vielen Papern im Bereich der Aktivitätserkennung als Maßstab verwendet wird. Den Pflegeaktivitäten-Datensatz nutzen wir, weil dieser ein herausforderndes Beispiel für die Erkennung einer großen Zahl komplexer Aktivitäten darstellt.
Unser Bewertungsmaßstab ist streng gewählt, um die Fähigkeit einer Architektur zur Generalisierung zu testen.
Wir teilen die Daten bereits auf Aufnahme-Ebene, statt auf Window-Ebene. So stellen wir sicher, dass das Modell nicht auf Eigenheiten verschiedener Aufnahmen overfitten kann. Um die Schwankungen, die durch unterschiedliche Datenaufteilungen und die zufälligen Initialisierung verursacht werden, auszugleichen, bedienen wir uns eines K-Folds mit fünf Faltungen.
Unser Setup haben wir vor dem ersten Experiment festgelegt und nicht mehr geändert, um dem Evolutionsalgorithmus nicht absichtlich eine günstige Konfiguration zu geben.

Niedrige Fitness Wir erreichen im Vergleich zu anderen Opportunity Aktivitätserkennungs-Maßstäben insgesamt eine relativ niedrige Fitness. Dies ist mit dem strengen Bewertungsmaßstab der Datenaufteilung, der Verwendung eine Untermenge der zur Verfügung stehenden Sensoren, sowie der klein gewählten Fenstergröße zu erklären.
Auch auf den Pflegeaktivitätsdaten stellen wir unsere Architektur vor eine große Herausforderung. Bei den 14 zu erkennenden Tätigkeiten handelt es sich um kom-

plexe, zusammengesetzte Aktivitäten, wie *Gesamtwaschen im Bett* oder *Medikamente stellen*. Anhand einer Fenstergröße von 1.5 Sekunden ist es anspruchsvoll die richtige Klassifizierung zu finden.

Wahl der Vergleichsmodelle Für die Einschätzung unseres Algorithmus vergleichen wir die Genauigkeit des evolutionären Modells mit der von etablierten, erfolgreichen Architekturen im Bereich der Aktivitätserkennung. Um sicherzustellen, dass keiner Architektur Potential durch ungünstig gewählte Trainingsparameter entgeht, führen wir eine automatisierte Optimierung der Parameter mit der Bibliothek Optuna durch.

Reproduzierbarkeit des Experiments Die Verwendung einer GPU ist für unseren Algorithmus unerlässlich, weil sonst die Ausführung deutlich länger als drei Tage in Anspruch nehmen würde. Mit GPU-Unterstützung ist es uns nicht möglich die Zufälligkeit der Initialisierung der Modelle im Training mit einem Seed zu fixieren. Daher unterliegt selbst unser strenger Bewertungsmaßstab kleinen Schwankungen bis zu zwei Prozent. In der Folge ist die im Experiment beschriebene Evolution nicht identisch reproduzierbar.
Wir sind jedoch in mehreren Ausführungen der Evolution mit Abweichungen von bis zu einem Prozent auf ähnlich hohe Genauigkeiten gekommen. Der Quellcode unseres Algorithmus ist frei verfügbar auf Github.

6.2 Analyse der Ergebnisse

6.2.1 Fitness

Auf dem Opportunity Datensatz waren wir ohne Optimierung der Trainingsparameter 4.85 Prozent, mit Optimierung 5.82 Prozent besser als das beste Vergleichsmodell. Auf dem Pflegeaktivitäten-Datensatz waren wir ohne Optimierung der Trainingsparameter 4.52 Prozent, mit Optimierung 5.64 Prozent besser als das beste Vergleichsmodell, siehe Tabelle 6.1.
Unser Ziel war es, mit den besten optimierten Vergleichsarchitekturen mithalten zu können. Es ist bemerkenswert, dass wir in beiden Fällen deutlich besser abschneiden. Damit bringt unser Algorithmus nicht nur die Zeitersparnis gegenüber der händischen Optimierung und Recherche, sondern er sollte zudem wegen seiner Ergebnisse in Erwägung gezogen werden.

Datensatz	Architektur	$\Delta Fitness$ ohne Optuna Optimierung	$\Delta Fitness$ mit Optuna Optimierung
Oppportunity	C(f=16, k=(4,2), s=(1,1), d=0.0, bn=True) C(f=147, k=(1,1), s=(1,1), d=0.0,bn=True) D(u=194, d=0.0) L(u=68, d=0.0)	4.85%	5.82%
Pflegeaktivitäten	C(f=25, k=(9,1), s=(2,1), d=0.0, bn=True) L(u=98, d=0.1) D(u=612, d=0.0)	4.52%	5.64%

Tabelle 6.1: Ergebnis der evolutionären Architektur gegenüber dem besten Vergleichsmodell

6.2.2 Architektur

Für den Opportunity Datensatz hat der evolutionäre Algorithmus folgende CNN-Dense-LSTM Architektur entwickelt.

C(f=16, k=(4,2), s=(1,1), d=0.0, bn=True)
C(f=147, k=(1,1), s=(1,1), d=0.0, bn=True)
D(u=194, d=0.0)
L(u=68, d=0.0)

Die evolutionäre Architektur erweitert die klassische CNNLSTM Architektur um eine Dense Schicht. Dabei fällt die zweite Convolutional Schicht mit einer ungewöhnlichen Parametrisierung auf. Mit Kernelgröße (1, 1) und Stridegröße (1, 1) gibt es keine klassische Filterung mehr. Stattdessen werden Feature-Maps generiert, die lediglich eine Skalierung der Eingabedaten pro Channel umsetzen. Die Convolutional Schichten besitzen kein Max Pooling, werden aber immer um eine Batch Normalisierung erweitert.

Für den Pflegeaktivitäten-Datensatz hat der evolutionäre Algorithmus folgende Architektur entwickelt.

C(f=25, k=(9,1), s=(2,1), d=0.0, bn=True)
L(u=98, d=0.1)
D(u=612, d=0.0)

Bemerkenswert an dieser Architektur ist die abschließende Dense Schicht mit einer großen Anzahl von Units. Das Modell kommt mit einer einzigen Convolutional Schicht ungewöhnlicher Kernelgröße aus und erweitert diese erneut um eine Batch Normalisierungsschicht.

6.3 Analyse der Evolutionären Techniken

In diesem Unterkapitel vergleichen wir zunächst die evolutionären Techniken mit der zufälligen Suche. Wir entwickeln Metriken, die aufzeigen, welchen Beitrag die einzelnen Techniken zur Optimierung geleistet haben. Daraus leiten wir ab, welche Rollen sie für die Evolution gespielt haben.
Schließlich analysieren wir den Verbesserungsverlauf der beiden Evolutionsexperimente und stellen diese in Zusammenhang mit den Erkenntnissen über die Rollen in der Evolution.

6.3.1 Zufällige Suche

Um zu zeigen, dass der Erfolg des Algorithmus auf evolutionäre Techniken zurückzuführen ist und nicht mit der zufälligen Suche austauschbar ist, betrachten wir den Beitrag der Zufallsindividuen in der Evolution.
Die zufälligen Startindividuen wurden in beiden Experimenten bereits bei der ersten Anwendung unserer Techniken abgelöst. Keines der Zufallsindividuen wurde zum beste Modell der ersten Generationen.

Darüber hinaus haben wir in jeder Generation ein Individuum zufällig erzeugt. Jedoch konnte sich im Verbesserungsverlauf kein Individuumm dieser Erzeugungsart durchsetzen. Aus Tabelle 6.2 geht hervor, dass die zufälligen Individuen im Vergleich zu den evolutionären Architekturen eine niedrigere Durchschnitts-Fitness besitzen.

Evolutionäre Technik	Durchschnitts-Fitness der Architekturen
Feinabstimmung des besten Individuums	67.94%
Leichte Mutation	60.90%
Mittlere Mutation	61.78%
Hohe Mutation	61.73%
Gesamte Mutation	62.01%
Uniforme Kreuzung	60.85%
Mittelpunkt Kreuzung	63.23%
Zufälliges Individuum	57.96%

Tabelle 6.2: Durchschnittsgenauigkeit der von den evolutionären Techniken produzierten Architekturen

6.3.2 Beitrag der Evolutionären Techniken

Um zu messen, welche Technik welchen Beitrag zur Optimierung geleistet hat, haben wir jedem neuen Individuum die Differenz-Fitness zu seinen Eltern-Genomen zugewiesen. Bei der Mutation gibt es genau ein Elterngenom, welches den gewünschten Bezugswert liefert, bei der Kreuzung berechnen wir die Differenz zum Mittelwert der Genauigkeiten der Eltern.
Auf diese Weise können wir bestimmen, wie oft eine Technik Individuen verbessert oder verschlechtert hat. In Abbildung 6.1 stellen wir diese Metrik für jede Evolutionstechnik in einem Violin-Plot dar. In Abbildung 6.2 haben wir den gleichen Plot mit einer Beschränkung der y-Achse auf eine Amplitude von -0,1 bis +0,1 erstellt. Als Datengrundlage haben wir 140 Generationen Evolution auf dem Datensatz Opportunity genutzt. Jede Technik wurde in jeder Generation einmal ausgeführt. Demnach hat jede Technik 140 Datenpunkte. Wir gehen davon aus, dass diese Samplegröße reicht, um erste Schlussfolgerungen zu ziehen. Dabei sei beachtet, dass wir eine Verfälschung durch Zufall nicht ausschließen können.

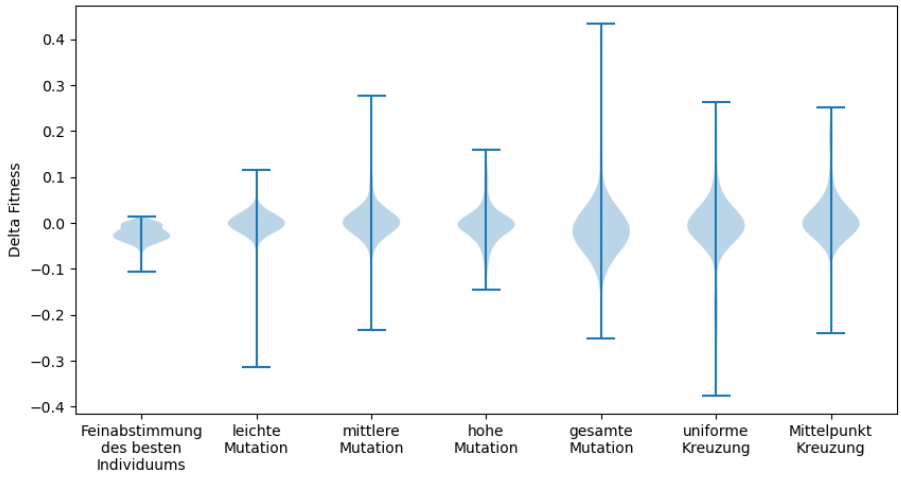


Abbildung 6.1: Violin-Plot der Differenzen zwischen der Fitness der Elterngenome und den evolutionär entwickelten Modellen

Veränderungsamplitude der Techniken

Zunächst fällt auf, dass sich der Großteil der Datenpunkte im Intervall zwischen -0,1 und +0,1 befindet. Angesichts des großen Änderungspotenzials, mit welchem wir Techniken, wie die *Gesamte Mutation* und die *Uniforme Kreuzung* ausgestattet haben, überrascht uns die kleine Amplitude. Gleichzeitig erinnert es uns daran, dass wir über den Verlauf der Evolutionen viele neuartige Strukturen gesehen haben, die unerwarteter Weise kompetetive Genauigkeiten hatten. Entsprechend unserer Beobachtung ist die Menge an Modellarchitekturen, die gute Ergebnisse erreichen können, groß.
Darüber hinaus zeigt die kleine Amplitude die enge Abhängigkeit der Kindgenome von ihren Elterngenomen.

Verhältnis von Verbesserung zu Verschlechterung

Die Plots zeigen, dass bei den meisten Techniken das Verhältnis von Individuen-Verbesserung und Verschlechterung ausgeglichen ist.
Bei der Technik *Feinabstimmung des besten Individuums* gibt es hingegen deutlich mehr Verschlechterung. Das liegt daran, dass diese Technik nur auf dem aktuell besten Individuen angewendet wird, anstatt auf einer Gruppe von Elterngenomen, in denen auch Genome niedriger Fitness vorhanden sind. Feinabstimmungs-Verbesserungen bedeuten immer eine Verbesserung des aktuellen Optimums der

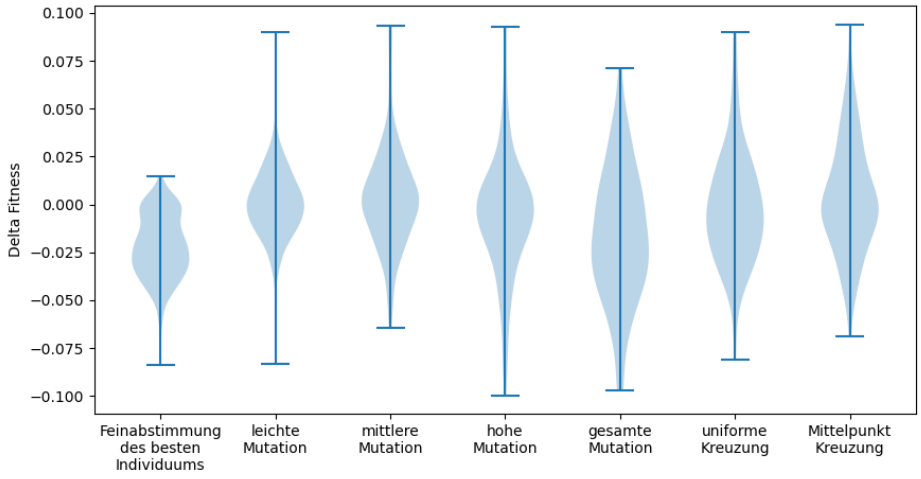


Abbildung 6.2: Violin-Plot der Differenzen zwischen der Fitness der Elterngenome und den evolutionär entwickelten Modellen im Intervall -0,1 bis +0,1

Evolution. Sie kommen seltener vor, sind aber von größerem Wert.
Dementsprechend leisten alle Techniken einen zufriedenstellenden Beitrag zur Evolution.

6.3.3 Rollenerfüllung der Evolutionären Techniken

Das Ziel unserer evolutionären Techniken ist es, auf der einen Seite durch Feinabstimmung bereits gute Individuen weiter zu verbessern, auf der anderen Seite offen für Innovationen zu sein. Bei Auswahl und Implementierung der Techniken war es uns wichtig, eine Balance der beiden Interessen herzustellen. Wir wollen weder in lokalen Optima hängen bleiben, noch zu viel schlechte Zufallsarchitekturen produzieren.
Auf der kontinuierlichen Skala von Feinabstimmung bis Innovation weisen wir jeder Technik eine Position zu. Einer Technik mit Fokus auf Innovation erlauben wir, große Schwankungen in der Fitness der Individuen zu haben. Dafür erwarten wir Verbesserungssprünge mit neuartigen Strukturen, die uns aus lokalen Optima herausholen. Eine Feinabstimmungs-Technik hingegen soll eine kleine Fitness-Amplitude haben, um sich gezielt dem nächsten lokalen Optimum zu nähern. Wenn wir uns bereits in nächster Nähe zum besten Optimum befinden, hilft es uns nicht, mit völlig neuartigen Strukturen den ganzen Problemraum abzutasten.
Mit dem Setzen von Intervallgrenzen und Wahrscheinlichkeiten können wir di-

rekt beeinflussen, wie stark Genome durch eine Technik verändert werden. Eine Technik, die wir mit dem Fokus Feinabstimmungen entwerfen, lassen wir kleine Veränderungen am Genom vornehmen, während wir einer Technik für Innovation großes Veränderungspotenzial geben. Dahinter steht die Intuition, dass es eine Korrelation zwischen Stärke der Parameterveränderung an der Modellarchitektur und Veränderung der Fitness gibt. Da jedoch in manchen Fällen schon eine kleine Veränderung an bestimmten Parametern zu einer großen Veränderung der Fitness führen kann, bleibt es bei der Erwartung einer schwachen Korrelation. Solange die Erklärbarkeit der Modelle eine Herausforderung darstellt, müssen uns experimentelle Beobachtungen als Werkzeug dienen, um sich der Balance zwischen Feinabstimmung und Innovation anzunähern.

Veränderungsamplitude Wie präzise unsere Wahl der Veränderungspotenziale wirklich war und ob diese sich wie gewünscht ergänzen, können wir nun am Violin-Plot in Abbildung 6.1 ablesen.
Die Amplituden der Techniken sind durch wenige Ausreißer ähnlich groß. Es lassen sich keine direkten Zusammenhänge mit den intendierten Rollen der Techniken herstellen.
Eine mögliche Begründung für diese Beobachtung ist, dass selbst eine kleine oder mittlere Mutation Modelle unkompilierbar machen kann. In diesem Fall werden mit Unterstützung des ModellCheckers zufällig Schichten ausgetauscht oder Parameterwerte gefünftelt. Das enstehende Individuum ordnen wir dennoch der ursprünglichen Technik zu.
Darüber hinaus kann dies auch ein Beispiel für ein kleine Änderung mit großem Effekt sein, welche untermauert, dass es keine direkte Korrelation zwischen Parameter- und Fitnessveränderung gibt.
Für die Einschätzung der Rollenerfüllung der Techniken nehmen wir die Ausreißer aus unsere Betrachtung heraus und fokussieren uns auf die Form die Dichtefunktionen in Abbildung 6.2.

Interpretation der Dichtefunktionen Mutationen gibt es in den Ausführungen leicht, mittel, hoch und gesamt. Entsprechend der Benennung hoffen wir auf kleine bis große Schwankungen in der Fitness. Vergleichen wir die Dichtefunktionen der vier Techniken in Abbildung 6.2, können wir den gewünschten Effekt beobachten. Mit zunehmendem Veränderungspotenzial ist die Streuung der Datenpunkte größer.
Die *Feinabstimmung des besten Individums* ist eine *leichte Mutation* auf dem bisher besten Individiuum der Evolution. Die hohe Dichte um den Wert null

erklären wir damit, dass die Feinabstimmung öfter Individuen erstellt, welche im Verlauf der Evolution bereits ausgewertet wurden. Dies kommt dadurch zustande, dass diese Technik wiederholt Änderungen auf dem immer selben Individuum vornimmt. Wenn ein Duplikat auftritt führen wir nicht noch einmal eine Evaluation durch, sondern weisen die bereits bekannte Fitness zu. Als Resultat erhält die Feinabstimmung in diesen Fällen einen Differenzwert von null. Darüber hinaus ist, wie bereits erwähnt, das Verhältnis zwischen Verbesserung und Verschlechterung unausgeglichen, weil diese Technik ausschließlich mit den besten Individuen arbeitet.
Die Kreuzungen gehören, wie erwartet, im Maßstab der Mutationen zu den Techniken mit größerem Streuungsgrad. Eine Neukombination zweier völlig unterschiedlicher Architekturen ist ein Eingriff mit hohem Veränderungspotenzial.

In der Summe deckt die Implementierung der Techniken ein breites Spektrum von Feinabstimmung bis Innovation ab. Damit ist die Voraussetzung dafür, dass sich die Techniken über den Verlauf der Evolution in ihren Rollen ergänzen, erfüllt.

6.3.4 Analyse des Verbesserungsverlaufs

Opportunity Daten

Diversität der Generationen Der orangene Graph in Abbildung 5.2 visualisiert für jede Generation die Fitness des besten Individuums. Es fällt auf, dass dieser Wert über die Generationen hinweg unverändert großen Schwankungen um bis zu sechs Prozent ausgesetzt ist.
Diese Beobachtung ist mit einer zentralen Entscheidung begründbar, die wir für unseren Algorithmus getroffen haben. Würden wir über die Generationen hinweg die leistungsstärksten Individuen überleben lassen und immer wieder als Erbmaterial für Folggenerationen verwenden, würde sich der orangene Graph nie nach unten bewegen.
Wir merken uns zu jedem Zeitpunkt der Evolution das beste, bisher gesehen Individuum. In den Generationen lassen wir hingegen zu, dass aus einer guten, alten Generation durch die evolutionären Techniken eine schlechtere, neue Generation entsteht. Selbst wenn ein Indiivuum ausgewählt ist, um sein Erbmaterial an die Folgeneration weiterzugeben, wird es dennoch nach Ablauf dieser Generation nicht ein weiteres Mal betrachtet.
Die Auswahl der Eltern für die Erzeugung einer Generation hat großen Einfluss auf das Aussehen der Folgegeneration. Durch die Anwendung evolutionärer Techniken mit geringem Änderungspotential besteht die Wahrscheinlichkeit, dass sich die

besten Individuen einer Generation ähneln. Wenn wir zulassen würden, dass starke Individuen über mehrere Generationen hinweg ihr Erbmaterial weitergeben dürfen, ist die Gefahr groß, dass viele Generationen in Struktur und Parametrisierungen gleichartig sind. Angesichts des großen Problemraums würden wir schnell in lokalen Optima hängenbleiben.
Wir akzeptieren, dass sich Generationen verschlechtern und haben dafür mehr Diversität in den Generationen.

Verlauf der besten Individuen Der blaue Graph zeigt zu jedem Zeitpunkt die Fitness des besten bekannten Indiivuums der Evolution an. In Tabelle 5.2 können wir sehen, welche Techniken für die Verbesserungen verantwortlich waren. Darüber hinaus sehen wir, welche Architekturen die hohen Genauigkeiten erreicht haben.
Die erste Generation besteht ausschließlich aus zufälligen Individuen. Schon nach dem ersten Evolutionschritt hat die *hohe Mutation* ein neues bestes Individuum hervorgebracht. Gerade in den frühen Generationen war zu erwarten, dass Techniken mit Schwerpunkt auf Innovation gute Chancen haben, beste Individuen zu produzieren. Feinabstimmungen stehen in enger Abhängigkeit zum Erbgut, welches ihnen zur Verfügung steht. Im frühen Stadium ist die Menge qualitativ hochwertiger Individuen noch gering. Bemerkenswerterweise kommen im Anschluss drei Verbesserungen durch *Feinabstimmung des besten Individuums* zustande. Obwohl wir uns noch in einer frühen Phase der Evolution befinden, ist der Einfluss einer Technik mit Fokus auf Feinabstimmung dominant. Es könnte sich zum einen um eine zufällige Erscheinung handeln, zum anderen könnte man diese Beobachtung mit dem hohen Startniveau der Evolution begründen.
In Generation 52 schlägt die *Mittelpunkt Kreuzung* ein besseres Individuum vor. Daraus resultiert zwar nur eine minimale Fitness-Verbesserung, dafür kommt eine innovative, neue Modellstruktur an die Spitze.
Ab diesem Zeitpunkt arbeitet die Feinabstimmung auf einer neuen, besten Architektur. In Generation 59 macht die Evolution den größten Verbesserungssprung. Auf der innovativen Architektur hat die *Feinabstimmung des besten Individuums* durch drei kleine Veränderungen ein deutlich besseres Modell produziert. Dieser Verlauf ist ein gutes Beispiel dafür, wie sich die Rollen der Techniken ergänzen. Die Feinabstimmung liefert zwar konstant Beiträge zur Evolution, jedoch sind die von ihr produzierten Individuen kaum divers.

Welche Technik letztendlich die ausschlaggebenden Beiträge leisten wird, ist schwer vorherzusehen. Auch wenn eine Technik nicht direkt für ein neues, bestes Indivi-

duum verantwortlich ist, ist es trotzdem möglich, dass sie wichtige Vorlagen dafür geleistet hat. Die Eltern der Individuen haben großen Einfluss auf Struktur und Parametrisierung der Kinder.

Pflegeaktivitätsdaten

Die Evolution auf den Pflegeaktivitätsdaten zeigt ein anderes Bild des Verbesserungsverlaufs. Die Technik *Feinabstimmung des besten Individuums* ist diesmal für keine einzige Verbesserung verantwortlich.
Bis Generation 13 sorgen Techniken mit Fokus auf Innovation für neue Architekturen. Das können wir erneut damit begründen, dass gerade in den frühen Generationen noch viele Architektur-Innovationen kompetitiv sind. Dennoch bleibt unklar, warum die Technik *Feinabstimmung des besten Individuums* keinen Einfluss nehmen konnte. Abschließend findet eine Verbesserung durch die *leichte Mutation* statt. Diese nimmt am aktuell besten Modell eine kleine Änderung vor und erzielt damit einen sehr großen Verbesserungssprung.

Die kleine Anzahl an Verbesserungen kann darin begründet sein, dass in Generation 41 zufällig bereits ein sehr gutes Modell getroffen wurde, welches kaum noch verbessert werden kann. Die Evolution ist trotzdem durch einen interessanten Prozess der Vorarbeit gegangen. Mit dieser Architektur werden im Vergleich zu den etablierten Modellen wesentlich höhere Genauigkeiten erreicht.

7 Konklusion

Wir haben einen Algorithmus implementiert, welcher Architekturen Tiefer Neuronaler Netze für Zeitreihendaten entwickeln kann.
Dafür haben wir eine umfangreiche Softwarearchitektur gebaut. Sie definiert ein ModellGenom, welches, mehrere Abstraktionebenen über der Keras-Bibliothek, evolutionäre Mechanismen unterstützt.
Für die Evolution haben wir uns an bekannten, evolutionären Techniken orientiert und diese im eigenen Kontext interpretiert. In unserem umfangreichen Problemraum beliebiger Schichtkombinationen und Parametrisierungen besteht die Gefahr in lokalen Optima hängen zu bleiben oder nur schlechte Zufalls-Architekturen zu produzieren.
Mit Auswahl und Implementierung der Techniken hält unser Algorithmus die Balance zwischen Weiterentwicklung der besten Individuen und Offenheit für Innovation. Während wir einerseits Zufall mit Verteilungsfunktionen beeinflussen und Minmal-Mutationen auf besten Individuen vornehmen, geben wir andererseits auch neuartigen Strukturen trotz schlechter Fitness eine Chance und bauen durch Kreuzung innovative Neukombinationen.
Im Ergebnis erhalten wir ein produktives Zusammenspiel der Techniken, welches zu stetigen Verbesserungen führt.

Wir konnten anhand zweier Datensätze zeigen, dass unser Algorithmus fähig ist, Architekturen zu entwickeln, die besser als etablierte Architekturen im Bereich der Aktivitätserkennung abschneiden.

7.1 Ausblick

Größe des Problemraums Unser Algorithmus beschränkt sich auf eine Auswahl von Schichten, auf eine Modelltiefe und die sequentielle Struktur der Modelle. Bekannte, erfolgreiche Architekturen, wie ResNet und InnoHAR, könnten von unserem Algorithmus nie kreiert werden, weil sie außerhalb des Suchbereichs liegen. Es wäre vielversprechend, unseren Algorithmus dahingehend zu erweitern und tiefe, mehrspurige Modelle mit einer größeren Auswahl von Schichten zuzulassen. Auch in der Parametrisierung der Schichten setzen wir an vielen Stellen Standardwerte,

welche optimiert möglicherweise höhere Genauigkeiten erzielen könnten.
Darüber hinaus hat die Datenvorverarbeitung einen großen Einfluss auf die Güte eines Modells. Die Art, wie die Aktivitäts-Fenster zurechtgeschnitten werden oder wie die Interpolation umgesetzt wird, könnten unter anderem ebenfalls Teil der Optimierung sein.

Adaptive Evolution Die Evolution selbst definiert zudem eine Menge eigener Parameter. Neben Populationsgröße und Generationszahl hat die Auswahl der Evolutionstechniken in jeder Generation einen großen Effekt auf den Verlauf der Optimierung. Es gibt Techniken, die mit höherer Wahrscheinlichkeit in den frühen Generationen Verbesserung bringen, und andere, die mit kleinen Veränderungen eher in späten Generationen von Nutzen sind. In unterschiedlichen Phasen der Evolution könnten unterschiedliche Techniken bevorzugt werden [MB07].
Die Evolutionstechniken selbst definieren Wahrscheinlichkeiten, welche bestimmen, wieviel sie an einem Modell ändern. Diese sind lediglich intuitiv gesetzt. Es würde eine Menge weitere Experimente erfordern, um deren Einfluss auf die Evolution zu messen. Auch hier wären Verbesserungen denkbar.

Innovationsschutz mithilfe einer Distanzmetrik Da der Problembereich der Architektursuche groß ist, ist die Optimierung immer in Gefahr sich in lokalen Optima zu verfangen. Wenn die produzierten Modelle zu gleichartig sind, entgeht der Evolution möglicherweise das Optimum mit einer andersartigen Architektur. Unser Algorithmus achtet bereits darauf, dass nicht nur die besten Modelle einer Generation ihr Erbmaterial weitergeben dürfen. Mit einer gewissen Wahrscheinlichkeit werden auch schlechte Modelle ausgewählt, um die nächste Generation zu erzeugen. In der evolutionären Optimierung gibt es jedoch einen Ansatz, der diese Aufgabe noch präziser angeht. Um gezielt Architektur-Innovationen, die zunächst nicht gut abschneiden, vor der Selektion zu beschützen, können mithilfe einer Distanzmetrik Modellunterschiede quantifizierbar gemacht werden.
Mit dieser Metrik kann zudem das Änderungspotenzial von Evolutionstechniken klarer definiert werden. Im Paper *A Study of Fitness Landscapes for Neuroevolution* [RSV20] wurde bereits für Convolutionale Tiefe Neuronale Netze eine grammatikbasierte Distanzmetrik entworfen.

Adaptive Fitnessfunktion Die Dauer der Evolution hängt stark davon ab, wie viel Zeit dafür eingeräumt wird, das Modell zu bewerten. Je mehr Zeit man sich nimmt, Modelle auf verschiedene Teilungen der Daten zu trainieren, um einen realistischen Mittelwert zu erhalten, desto länger wird die Optimierung

brauchen. Unser Algorithmus benötigt auf dem Opportunity Datensatz mit GPU Unterstützung in etwa zwei Tage. Um auf diese Dauer zu kommen, sind wir den Kompromiss eingegangen, fünf Teilungen vorzunehmen und jeweils nur fünf Epochen zu trainieren. Bei vielen Modellen sieht man schon in frühen Epochen, ob die Architektur effektiv lernt. Es gibt jedoch auch Modelle, die hohe Genauigkeiten haben können, diese aber erst nach einer deutlich größeren Anzahl von Epochen erreichen. Diese Modelle werden durch diese Entscheidung benachteiligt. Ihre Architekturen können sich bei unserem Algorithmus nicht durchsetzen. Würde man je nach Modellgröße oder Trainingsverlauf die Epochenzahl anpassen, könnte man diesen Nachteil besser ausgleichen. Auch die Wahl von Lernrate und Batchgröße ist bei uns während der Architektur-Optimierung immer gleichbleibend. Es ist möglich, dass der getrennten Optimierung von Architektur- und Trainingsparametern viele Modelle entgehen. Modelle, die nur in der Gesamtheit ihrer Parameter betrachtet, zu einer überragenden Fitness kommen, werden von unserem Ansatz übersehen.

In der Zukunft wäre es außerdem spannend, evolutionäre Architektursuche in vielen anderen Bereichen des maschinellen Lernens zu verwenden.

Literatur

[Aba+16] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu und Xiaoqiang Zheng. **TensorFlow: A system for large-scale machine learning**. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf (siehe S. 19).

[AE17] Sadam Al-Azani und El-Sayed El-Alfy. **Hybrid Deep Learning for Sentiment Polarity Determination of Arabic Microblogs**. In: Okt. 2017, 491–500. ISBN: 978-3-319-70095-3. DOI: 10.1007/978-3-319-70096-0_51 (siehe S. 13).

[Aki+19] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta und Masanori Koyama. **Optuna: A Next-generation Hyperparameter Optimization Framework**. In: Juli 2019, 2623–2631. ISBN: 978-1-4503-6201-6. DOI: 10.1145/3292500.3330701 (siehe S. 19, 42).

[BB12] James Bergstra und Y. Bengio. **Random Search for Hyper-Parameter Optimization**. *The Journal of Machine Learning Research* 13 (März 2012), 281–305 (siehe S. 2).

[BL97] Y. Bengio und Yann Lecun. **Convolutional Networks for Images, Speech, and Time-Series** (Nov. 1997) (siehe S. 6).

[Cha+13] Ricardo Chavarriaga, Hesam Sagha, Alberto Calatroni, Sundara Tejaswi Digumarti, Jose del R. Millan, Daniel Roggen und Gerhard Tröster. **The Opportunity challenge: A benchmark database for on-body sensor-based activity recognition**. *Pattern Recognition Letters* 23 (Jan. 2013), 2033–2042. DOI: 10.1016/j.patrec.2012.12.014 (siehe S. 36, 37).

[CS11] Brian Cheung und Carl Sable. **Hybrid Evolution of Convolutional Networks**. In: *2011 10th International Conference on Machine Learning and Applications and Workshops*. Bd. 1. 2011, 293–297. DOI: 10.1109/ICMLA.2011.73 (siehe S. 13).

[DEZ19] Abdelghani Dahou, Mohamed Elsayed Abd Elaziz und Junwei Zhou. **Arabic Sentiment Classification Using Convolutional Neural Network and Differential Evolution Algorithm**. *Computational Intelligence and Neuroscience* 2019 (Feb. 2019), 1–16. DOI: 10.1155/2019/2537689 (siehe S. 13).

[Die95] Tom Dietterich. **Overfitting and Undercomputing in Machine Learning**. *ACM Comput. Surv.* 27:3 (Sep. 1995), 326–327. ISSN: 0360-0300. DOI: 10.1145/212094.212114. URL: https://doi.org/10.1145/212094.212114 (siehe S. 6).

[EMH18] Thomas Elsken, Jan Hendrik Metzen und Frank Hutter. **Neural Architecture Search: A Survey** (2018). DOI: 10.48550/ARXIV.1808.05377. URL: https://arxiv.org/abs/1808.05377 (siehe S. 12).

[He+15] Kaiming He, Xiangyu Zhang, Shaoqing Ren und Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. DOI: 10.48550/ARXIV.1512.03385. URL: https://arxiv.org/abs/1512.03385 (siehe S. 37).

[HNF09] Alon Halevy, Peter Norvig und Nanediri Fernando. **The Unreasonable Effectiveness of Data**. *Intelligent Systems, IEEE* 24 (Mai 2009), 8–12. DOI: 10.1109/MIS.2009.36 (siehe S. 2).

[Hoc98] Sepp Hochreiter. **The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions**. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 06:02 (1998), 107–116. DOI: 10.1142/S0218488598000094. eprint: https://doi.org/10.1142/S0218488598000094. URL: https://doi.org/10.1142/S0218488598000094 (siehe S. 8).

[HS97] Sepp Hochreiter und Jürgen Schmidhuber. **Long Short-term Memory**. *Neural computation* 9 (Dez. 1997), 1735–80. DOI: 10.1162/neco.1997.9.8.1735 (siehe S. 9).

[IS15] Sergey Ioffe und Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167 (siehe S. 8).

[Kri12] Alex Krizhevsky. **Learning Multiple Layers of Features from Tiny Images**. *University of Toronto* (Mai 2012) (siehe S. 15).

[LBE15] Zachary C. Lipton, John Berkowitz und Charles Elkan. *A Critical Review of Recurrent Neural Networks for Sequence Learning*. 2015. DOI: 10.48550/ARXIV.1506.00019. URL: https://arxiv.org/abs/1506.00019 (siehe S. 8).

[LeC+89] 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**. *Neural Computation* 1:4 (Dez. 1989), 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. eprint: https://direct.mit.edu/neco/article-pdf/1/4/541/811941/neco.1989.1.4.541.pdf. URL: https://doi.org/10.1162/neco.1989.1.4.541 (siehe S. 13).

[Liu+21] Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G. Yen und Kay Chen Tan. **A Survey on Evolutionary Neural Architecture Search**. *IEEE Transactions on Neural Networks and Learning Systems* (2021), 1–21. DOI: 10.1109/tnnls.2021.3100554. URL: https://doi.org/10.1109%2Ftnnls.2021.3100554 (siehe S. 12).

[MB07] Silja Meyer-Nieberg und Hans-Georg Beyer, 47–75. In: Bd. 54. Apr. 2007. ISBN: 978-3-540-69431-1. DOI: 10.1007/978-3-540-69432-8_3 (siehe S. 58).

[Mit98] Melanie Mitchell. **An Introduction to Genetic Algorithms**. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262631857 (siehe S. 10).

[NH10] Vinod Nair und Geoffrey Hinton. **Rectified Linear Units Improve Restricted Boltzmann Machines Vinod Nair**. In: Bd. 27. Juni 2010, 807–814 (siehe S. 33).

[Ped+12] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, Edouard Duchesnay und Gilles Louppe. **Scikit-learn: Machine Learning in Python**. *Journal of Machine Learning Research* 12 (Jan. 2012) (siehe S. 27).

[Rea+17] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le und Alexey Kurakin. **Large-Scale Evolution of Image Classifiers**. In: *Proceedings of the 34th International Conference on Machine Learning*. Hrsg. von Doina Precup und Yee Whye Teh. Bd. 70. Proceedings of Machine Learning Research. PMLR, Juni 2017, 2902–2911. URL: https://proceedings.mlr.press/v70/real17a.html (siehe S. 14, 15).

[RSV20] Nuno Rodrigues, Sara Silva und Leonardo Vanneschi. **A Study of Fitness Landscapes for Neuroevolution**. In: Juli 2020. DOI: 10.1109/CEC48606.2020.9185783 (siehe S. 58).

[SA19] Dima Suleiman und Arafat Awajan. **Comparative Study of Word Embeddings Models and Their Usage in Arabic Language Applications** (Apr. 2019). DOI: 10.1109/ACIT.2018.8672674 (siehe S. 13).

[Sai+15] Tara Sainath, Oriol Vinyals, Andrew Senior und Hasim Sak. **Convolutional, Long Short-Term Memory, Fully Connected Deep Neural Networks**. In: *ICASSP*. 2015 (siehe S. 1, 36).

[Sin+19] Animesh Singh, Sandip Saha, Ritesh Sarkhel, Mahantapas Kundu, Mita Nasipuri und Nibaran Das. *A Genetic Algorithm based Kernel-size Selection Approach for a Multi-column Convolutional Neural Network*. 2019. DOI: 10.48550/ARXIV.1912.12405. URL: https://arxiv.org/abs/1912.12405 (siehe S. 13).

[SM02] Kenneth O. Stanley und Risto Miikkulainen. **Evolving Neural Networks through Augmenting Topologies**. *Evolutionary Computation* 10:2 (2002), 99–127. DOI: 10.1162/106365602320169811 (siehe S. 14).

[SMB10] Dominik Scherer, Andreas Müller und Sven Behnke. **Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition**. In: Jan. 2010, 92–101. ISBN: 978-3-642-15824-7. DOI: 10.1007/978-3-642-15825-4_10 (siehe S. 7).

[Sri+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever und Ruslan Salakhutdinov. **Dropout: A Simple Way to Prevent Neural Networks from Overfitting**. *Journal of Machine Learning Research* 15:56 (2014), 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (siehe S. 6).

[Sta12] Kenneth Stanley. **Neuroevolution: Evolving neural networks**. *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Companion* (Juli 2012), 805–826. DOI: 10.1145/2330784.2330917 (siehe S. 12).

[Sys89] Gilbert Syswerda. **Uniform Crossover in Genetic Algorithms**. In: Jan. 1989 (siehe S. 23, 24).

[Xu+19] Cheng Xu, Duo Chai, Jie He, Xiaotong Zhang und Shihong Duan. **InnoHAR: A Deep Neural Network for Complex Human Activity Recognition**. *IEEE Access* PP (Jan. 2019), 1–1. DOI: 10.1109/ACCESS.2018.2890675 (siehe S. 37).

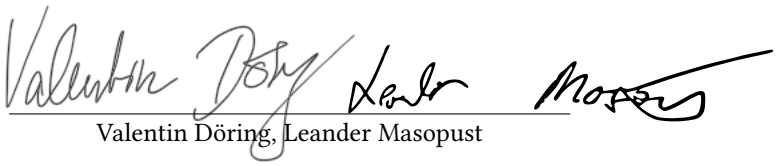
[ZL16] Barret Zoph und Quoc V. Le. *Neural Architecture Search with Reinforcement Learning*. 2016. DOI: 10.48550/ARXIV.1611.01578. URL: https://arxiv.org/abs/1611.01578 (siehe S. 14, 15).

Abbildungsverzeichnis

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen sind als solche kenntlich gemacht.

Potsdam, 28. Juli 2022



Valentin Döring, Leander Masopust