Hochschule Hamm - Lippstadt
Angewandte Informatik und Soziale Medien

Bachelorarbeit

Entwicklung und Evaluation eines neuronalen Netzwerks für die Pfadfindung in Labyrinthen bei vollständig bekannter Geometrie

Eingereicht von:

Luc Westbomke
Westwall 5a
59581 Warstein

Matrikelnummer: 2210049

Abgabedatum: 17. Februar 2025

Erstprüfer: Prof. Dr. Matthias Vögeler
Zweitprüfer: Prof. Dr. Georg Birkenheuer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

A*	A*-Algorithmus
CAE	Convolutional Autoencoder
CNN	Convolutional Neural Network
FCAE	Fully-Convolutional Autoencoder
FCNN	Fully-Connected Neural Network
FFN	Feedforward Neural Network
MSE	Mean Squared Error
NN	Neuronale Netzwerke

Glossar

Batch Normalization Eine Technik zur Normalisierung der Aktivierungen in tiefen neuronalen Netzwerken, die dazu beiträgt, das Vanishing-Gradient-Problem zu reduzieren und das Training zu stabilisieren

GPU Eine spezialisierte Hardware zur parallelen Verarbeitung von grafischen und mathematischen Operationen. GPUs sind besonders nützlich für rechenintensive Aufgaben wie Bildverarbeitung und maschinelles Lernen, da sie zahlreiche Berechnungen simultan ausführen können

Loss Ein Maß dafür, wie weit die Vorhersagen eines Modells von den tatsächlichen Werten abweichen. Ein geringerer Loss-Wert deutet auf eine bessere Übereinstimmung zwischen Modellvorhersagen und Zielwerten hin. Verluste sind entscheidend für die Optimierung von maschinellen Lernmodellen

Vanishing-Gradient-Problem Ein Problem beim Training tiefer neuronaler Netzwerke, bei dem die Gradienten während der Backpropagation so stark abnehmen, dass die frühen Schichten nur noch sehr langsam oder gar nicht mehr lernen

1 Einleitung

Pfadfindungsalgorithmen sind ein fundamentales Element in vielen Bereichen der Informatik, wie etwa in der Robotik, in Navigationssystemen und in Computerspielen. Sie ermöglichen es, den effizientesten Weg zwischen zwei Punkten in einer gegebenen Umgebung zu bestimmen (Hart et al., 1968). Insbesondere bei der Navigation durch komplexe Strukturen wie Labyrinthe stellen sie eine zentrale Herausforderung dar.

Traditionelle Algorithmen wie der A*-Algorithmus (A*) haben sich aufgrund ihrer Effizienz und Genauigkeit weitgehend etabliert (Russell & Norvig, 2016). Mit dem Aufkommen von Deep Learning und Neuronale Netzwerkes (NNs) eröffnen sich jedoch neue Perspektiven für die Pfadfindung. NNs haben die Fähigkeit, komplexe Muster in Daten zu erkennen und könnten somit alternative Ansätze zur Lösung von Pfadfindungsproblemen bieten (LeCun et al., 2015).

Diese Arbeit untersucht den Einsatz von NNs, insbesondere von Fully-Connected Neural Networks (FCNNs) und Fully-Convolutional Autoencoders (FCAEs), für die Pfadfindung in Labyrinthen. Dabei werden bestehende FFN-Modelle aus einer vorangegangenen Studie (Vlad, 2023) herangezogen und mit eigens entwickelten FCAE-Modellen verglichen. Für den Vergleich werden 7×7 große Labyrinthe verwendet, die ebenfalls aus der genannten Studie stammen. Zudem werden die FCAE-Modelle mit dem A*-Algorithmus verglichen, wobei eigene Labyrinthe mittels des Aldous-Broder-Algorithmus generiert wurden.

Um die Reproduzierbarkeit der in dieser Arbeit durchgeführten Experimente zu gewährleisten, wurde der gesamte Quellcode sowie die finale Version der Bachelorarbeit in einem öffentlichen GitHub-Repository[1] bereitgestellt. Dieses enthält die Implementierung der neuronalen Netzwerke, die Algorithmen zum Vergleich sowie die dazugehörigen Datensätze und Analysen.

1.1 Zielsetzung

Das Hauptziel dieser Arbeit ist es, die Leistungsfähigkeit von eigens entwickelten FCAE-Modellen bei der Pfadfindung in Labyrinthen zu analysieren und sie sowohl mit bestehenden FCNN-Modellen (Vlad, 2023) als auch mit dem A*-Algorithmus zu vergleichen. Dabei stehen folgende Fragestellungen im Fokus:

[1] https://github.com/luc-westbomke0/Bachelorarbeit

- Wie genau sind die von den FCAE-Modellen gefundenen Pfade im Vergleich zu denen der FCNN-Modelle und des A*-Algorithmus?
- Wie verhalten sich die Laufzeiten und die Effizienz der FCAE-Modelle im Vergleich zu den FCNN-Modellen und dem A*-Algorithmus?

1.2 Motivation

Die Motivation für diese Untersuchung liegt in der stetigen Weiterentwicklung von künstlicher Intelligenz und Machine Learning. Während traditionelle Algorithmen wie A* auf heuristischen Methoden basieren, bieten NNs die Möglichkeit, aus Daten zu lernen und Muster zu erkennen, die für heuristische Ansätze schwer zugänglich sind (Goodfellow et al., 2016). Dies könnte insbesondere in komplexen oder dynamischen Umgebungen von Vorteil sein.

Die Verwendung von FCAE-Modellen bietet spezifische Vorteile, da sie durch ihre Architektur besonders gut für die Verarbeitung von bildartigen Daten geeignet sind (Masci et al., 2011). Zudem könnten NNs, einmal trainiert, schnellere Entscheidungen treffen und weniger Rechenressourcen benötigen, was in Echtzeitanwendungen entscheidend sein kann (Schmidhuber, 2015).

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in folgende Kapitel:

- **Kapitel 2: Theoretische Grundlagen** – Dieses Kapitel führt in die Pfadfindung ein und beschreibt klassische Algorithmen wie den A*, Grundlagen neuronaler Netze sowie Verfahren zur Labyrinth-Generierung und Evaluationsmetriken.
- **Kapitel 3: Modellierung und Implementierung** – In diesem Kapitel werden die Architektur der FCAE-Modelle und des A*-Algorithmus sowie der Trainingsprozess beschrieben.
- **Kapitel 4: Ergebnisse** – Die experimentellen Ergebnisse der FCAE-Modelle werden präsentiert und mit den FCNN-Modellen und dem A*-Algorithmus verglichen.
- **Kapitel 5: Diskussion** – Die Ergebnisse werden interpretiert, Limitationen der Arbeit identifiziert und Verbesserungsvorschläge formuliert.

- **Kapitel 6: Fazit und Ausblick** – Die zentralen Erkenntnisse der Arbeit werden zusammengefasst und mögliche zukünftige Forschungsrichtungen aufgezeigt.

2 Theoretische Grundlagen

Dieses Kapitel gibt einen Überblick über die wesentlichen theoretischen Grundlagen, die für das Verständnis der Arbeit erforderlich sind. Dazu gehören eine Einführung in die Pfadfindung, eine Beschreibung klassischer Algorithmen, neuronaler Netzwerke sowie Verfahren zur Labyrinth-Generierung und Evaluationsmetriken.

2.1 Einführung in die Pfadfindung

Die Pfadfindung ist ein zentrales Problem in der Informatik und Mathematik, das darauf abzielt, einen optimalen Weg zwischen zwei Punkten in einem definierten Raum zu finden. Sie spielt eine entscheidende Rolle in Bereichen wie Robotik, Navigationssystemen und Computerspielen. Die Pfadfindung ermöglicht es, Aufgaben wie Navigation, Routenplanung und Bewegungssteuerung effizient zu lösen (Cormen et al., 2022).

2.1.1 Definition

Pfadfindung beschreibt die algorithmische Bestimmung des effizientesten Weges zwischen einem definierten Startpunkt und einem Zielpunkt in einer Umgebung. Diese wird häufig als Graph modelliert, wobei Knoten die Positionen und Kanten die möglichen Verbindungen zwischen diesen Positionen darstellen. Ein Pfad kann somit als eine Folge von Kanten bezeichnen (Mussmann & See, n. d.; Russell & Norvig, 2016).

Die Pfadfindung wird in vielen Bereichen eingesetzt, darunter Robotik, Computerspiele und Verkehrsplanung. In der Robotik wird beispielsweise der optimale Weg durch Hindernisse berechnet (Latombe, 1991), während in Computerspielen Nicht-Spieler-Charaktere durch komplexe Umgebungen navigieren (Cui & Shi, 2011).

2.1.2 Problemstellung bei Labyrinthen

Labyrinthe stellen für Pfadfindungsalgorithmen besondere Herausforderungen dar. Um effektive Lösungen zu entwickeln, ist es hilfreich, Labyrinthe als Graphen zu modellieren. In dieser Darstellung wird jedes begehbare Feld des Labyrinths als Knoten $v \in V$ eines Graphen $G = (V, E)$ betrachtet (Sturtevant, 2012). Die möglichen Bewegungen zwischen benachbarten Feldern werden als Kanten $e \in E$ dargestellt, die die entsprechenden Knoten verbinden.

Hindernisse oder Wände im Labyrinth führen dazu, dass bestimmte Kanten zwischen Knoten fehlen, wodurch die Bewegung zwischen diesen Feldern nicht möglich ist (Loomis Jr., 2012).

Diese Graphmodellierung ermöglicht die Anwendung klassischer Pfadfindungsalgorithmen auf Labyrinthe. Allerdings führt die hohe Anzahl von Knoten und Kanten in komplexen Labyrinthen zu einem enormen Suchraum. Dies stellt hohe Anforderungen an die Effizienz der Algorithmen, da der optimale Pfad innerhalb einer akzeptablen Zeit gefunden werden muss (Cui & Shi, 2011).

Ein weiterer Aspekt ist, dass Labyrinthe häufig unvorhersehbare Blockaden und Sackgassen enthalten, was die Suche nach dem optimalen Pfad zusätzlich erschwert (Loizou & Kumar, 2006). Pfadfindungsalgorithmen müssen daher in der Lage sein, solche Hindernisse zu umgehen und alternative Routen zu finden. Dies erfordert intelligente Suchstrategien und gegebenenfalls die Implementierung von Heuristiken, um die Suche effizient zu gestalten (Dechter & Pearl, 1985).

2.2 Klassische Pfadfindungsalgorithmen

Klassische Pfadfindungsalgorithmen basieren auf systematischen Suchstrategien, um den optimalen Pfad in einem Graphen zu finden. Sie unterscheiden sich in ihrer Effizienz, Komplexität und Anwendbarkeit auf verschiedene Problemstellungen (Dechter & Pearl, 1985).

2.2.1 Überblick über klassische Algorithmen

Zu den grundlegenden klassischen Pfadfindungsalgorithmen, die in der Pfadfindung verwendet werden, gehören:

- **Breitensuche (Breadth-First Search, BFS)**: Durchsucht den Graphen in der Breite, indem sie alle Nachbarn eines Knotens untersucht, bevor sie in die Tiefe geht. BFS garantiert die Findung des kürzesten Pfades in ungewichteten Graphen. Die Laufzeit beträgt im Worst-Case $\mathcal{O}(|V| + |E|)$, wobei V die Anzahl der Knoten und E die Anzahl der Kanten ist (Cormen et al., 2022; Moore, 1959).
- **Tiefensuche (Depth-First Search, DFS)**: Durchsucht den Graphen in der Tiefe, indem sie so weit wie möglich entlang eines Zweiges geht, bevor sie zurückkehrt und alternative Pfade erkundet. DFS ist speichereffizienter als BFS, kann aber in tiefen

Graphen ineffizient sein. Die Laufzeit beträgt ebenfalls $\mathcal{O}(|V| + |E|)$ (Cormen et al., 2022; Tarjan, 1972).

- **Dijkstra-Algorithmus**: Ein Algorithmus zur Findung der kürzesten Pfade von einem Startknoten zu allen anderen Knoten in einem Graphen mit nicht-negativen Kantengewichten. Er verwendet eine Prioritätswarteschlange, um die nächste zu untersuchende Kante auszuwählen. Die Laufzeit beträgt $\mathcal{O}(|V|^2)$ für eine naive Implementierung und $\mathcal{O}(|E| + |V| \log |V|)$ bei Verwendung einer binären Heap-Prioritätswarteschlange (Cormen et al., 2022; Dijkstra, 1959).
- **A*-Algorithmus**: Kombiniert die Eigenschaften von Dijkstra mit einer Heuristik, um die Suche effizienter zum Ziel zu leiten. Die durchschnittliche Laufzeit hängt von der verwendeten Heuristik ab. Im besten Fall kann sie sich der Laufzeit $\mathcal{O}(|V|)$ annähern, während sie im ungünstigsten Fall der Laufzeit von Dijkstra entspricht, also $\mathcal{O}(|E| + |V| \log |V|)$ (Hart et al., 1968; Russell & Norvig, 2016).

Diese Algorithmen bilden die Grundlage für viele Anwendungen der Pfadfindung, darunter auch die Lösung von Labyrinthen. Besonders der A*-Algorithmus zeigt durch die Verwendung von Heuristiken eine hohe Effizienz in Szenarien, bei denen der Zielknoten bekannt ist. Die in diesem Abschnitt beschriebenen Ansätze werden in dieser Arbeit als Maßstab für die Leistungsbewertung neuronaler Netzwerke verwendet.

2.2.2 A*

Der A*-Algorithmus ist ein informierter Suchalgorithmus, der häufig in der Pfadfindung eingesetzt wird. Er erweitert den Dijkstra-Algorithmus durch die Einführung einer Heuristikfunktion $h(n)$, die eine Schätzung der Kosten vom aktuellen Knoten n zum Zielknoten liefert (Hart et al., 1968).

Der Algorithmus funktioniert wie folgt:

1. Der Startknoten wird als einziger Knoten in eine Prioritätswarteschlange (Open-List) eingefügt. Die Kostenfunktion jedes Knotens wird mit der Formel:

$$f(n) = g(n) + h(n) \tag{2.1}$$

   berechnet, wobei $g(n)$ die tatsächlichen Kosten vom Startknoten zum aktuellen Knoten n und $h(n)$ die geschätzten Kosten vom aktuellen Knoten n zum Zielknoten sind.

2. In jedem Iterationsschritt wird der Knoten mit den niedrigsten geschätzten Kosten $f(n)$ aus der Open-List entfernt und in die Closed-List aufgenommen.

3. Die Nachbarn des aktuellen Knotens werden untersucht:
   - Falls ein Nachbar bereits in der Closed-List enthalten ist, wird er übersprungen.
   - Falls der Nachbar nicht in der Open-List ist, wird er hinzugefügt und sein Wert für $g(n)$ sowie $h(n)$ berechnet.
   - Falls der Nachbar bereits in der Open-List enthalten ist, wird überprüft, ob ein alternativer Pfad mit niedrigeren Kosten existiert. Falls ja, wird der Wert von $g(n)$ aktualisiert.
4. Dieser Vorgang wird solange wiederholt, bis der Zielknoten aus der Open-List entnommen wird oder die Open-List leer ist (was bedeutet, dass kein Pfad existiert).
5. Nach Erreichen des Zielknotens wird der Pfad durch Rückverfolgung der gespeicherten Vorgängerknoten rekonstruiert.

Die Wahl der Heuristikfunktion $h(n)$ beeinflusst die Effizienz des Algorithmus maßgeblich. Damit der Algorithmus optimale Lösungen garantiert, muss die Heuristik $h(n)$ zulässig und konsistent sein (Pearl, 1984; Russell & Norvig, 2016).

Eine zulässige Heuristik überschätzt niemals die tatsächlichen Kosten zum Ziel, d. h. für alle Knoten n gilt:

$$h(n) \leq h^*(n) \tag{2.2}$$

wobei $h^*(n)$ die tatsächlichen Kosten vom Knoten n zum Ziel darstellt. Dies stellt sicher, dass der Algorithmus keine zu niedrigen Pfade verwirft und somit die optimalen Kosten findet (Pearl, 1984).

Eine konsistente Heuristik (auch monoton steigende Heuristik) erfüllt die zusätzliche Bedingung:

$$h(n) \leq c(n, m) + h(m) \tag{2.3}$$

für alle benachbarten Knoten n und m, wobei $c(n, m)$ die tatsächlichen Kosten des Übergangs von n nach m sind. Eine konsistente Heuristik garantiert, dass die Werte der $f(n)$-Funktion niemals abnehmen, was bedeutet, dass ein einmal abgeschlossener Knoten nicht erneut betrachtet werden muss. Dies macht den Algorithmus effizienter (Dechter & Pearl, 1985; Russell & Norvig, 2016).

Eine häufig verwendete Heuristik in Gitternetzwerken ist die Manhattan-Distanz:

$$h(n) = |x_n - x_{\text{Ziel}}| + |y_n - y_{\text{Ziel}}| \tag{2.4}$$

(Sturtevant, 2012). Die Manhattan-Distanz ist sowohl zulässig als auch konsistent, solange nur Bewegungen in horizontaler oder vertikaler Richtung erlaubt sind (Russell & Norvig, 2016).

Der A*-Algorithmus ist besonders effizient, da er die Suche auf vielversprechende Bereiche des Suchraums konzentriert und dadurch die Anzahl der zu untersuchenden Knoten reduziert (Hart et al., 1968; Russell & Norvig, 2016). In dieser Arbeit wird der A*-Algorithmus als Benchmark verwendet, um die Leistungsfähigkeit der entwickelten FCAE-Modelle zu bewerten.

2.3 Neuronale Netzwerke

Neuronale Netzwerke (NNs) sind Modelle des maschinellen Lernens, die auf der Verarbeitung von Daten durch miteinander verbundene künstliche Neuronen basieren. Sie sind in der Lage, komplexe Muster und Abhängigkeiten in Daten zu erkennen und zu modellieren, wodurch sie in den letzten Jahrzehnten bedeutende Fortschritte in Bereichen wie der Bild- und Spracherkennung, der Robotik und der Entscheidungsfindung ermöglicht haben (Goodfellow et al., 2016).

In dieser Arbeit werden neuronale Netzwerke verwendet, um die Lösung von Labyrinthen zu automatisieren und die Effizienz von klassischen Pfadfindungsalgorithmen mit modernen Architekturen wie FCNNs und FCAEs zu vergleichen. Dieser Abschnitt gibt eine Einführung in die Grundlagen neuronaler Netzwerke und stellt spezifische Architekturen vor, die in dieser Arbeit untersucht werden.

2.3.1 Allgemeine Einführung

NNs bestehen aus einer Vielzahl einfacher Verarbeitungseinheiten, den sogenannten künstlichen Neuronen. Diese Neuronen sind in Schichten organisiert, wodurch das Netzwerk Eingabedaten schrittweise transformieren und komplexe Muster lernen kann (Goodfellow et al., 2016).

Das künstliche Neuron: Verarbeitungseinheit des Netzwerks

Ein künstliches Neuron bildet die grundlegende Recheneinheit eines neuronalen Netzwerks. Es berechnet die gewichtete Summe der Eingaben $x_1, x_2, \ldots, x_n$, addiert einen Bias b, und transformiert diese Summe mittels einer Aktivierungsfunktion f zu einer Ausgabe y:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{2.5}$$

wobei w_i die trainierbaren Gewichte und b der Bias des Neurons sind. Die Aktivierungsfunktion f ist entscheidend, da sie Nichtlinearitäten einführt, die es dem Netzwerk ermöglichen, komplexe Zusammenhänge in den Daten zu lernen (LeCun et al., 2015).

Die Rolle der Aktivierungsfunktion

Die Aktivierungsfunktion f transformiert die Ausgabe eines Neurons und ermöglicht es dem Netzwerk, nicht-lineare Beziehungen zu modellieren. Ohne diese Funktion würde das Netzwerk lediglich lineare Transformationen ausführen, unabhängig von der Anzahl der Schichten. Häufig verwendete Aktivierungsfunktionen sind:

- **ReLU (Rectified Linear Unit):** $f(x) = \max(0, x)$. Diese Funktion ist recheneffizient, reduziert das Problem des Vanishing-Gradient-Problems und ist daher in tiefen neuronalen Netzwerken weit verbreitet.
- **Sigmoid:** $f(x) = \frac{1}{1+e^{-x}}$. Die Sigmoid-Funktion wird oft für Wahrscheinlichkeitsvorhersagen eingesetzt, da die Ausgaben im Bereich $[0, 1]$ liegen.
- **Tanh (Hyperbolischer Tangens):** $f(x) = \tanh(x)$. Diese Funktion wird verwendet, wenn symmetrische Ausgaben im Bereich $[-1, 1]$ benötigt werden.

In tiefen neuronalen Netzwerken kann das Vanishing-Gradient-Problem auftreten, wenn während des Backpropagation-Prozesses die Gradienten in frühen Schichten nahezu null werden, wodurch das Netzwerk nicht mehr effektiv trainiert werden kann (Goodfellow et al., 2016).

Um diesem Problem entgegenzuwirken, wird häufig die ReLU-Aktivierungsfunktion eingesetzt, da sie für positive Werte einen konstanten Gradienten von 1 hat. Zusätzlich wird Batch Normalization verwendet, um die Eingabeverteilungen der Neuronen während des Trainings zu stabilisieren und das Problem des verschwindenden Gradienten weiter zu reduzieren (Ioffe & Szegedy, 2015).

Von Neuronen zu einer Schicht

Eine Schicht in einem neuronalen Netzwerk besteht aus mehreren Neuronen, die parallel arbeiten. Die Transformation einer Eingabe durch die gesamte Schicht kann durch folgende Gleichung beschrieben werden:

$$\mathbf{h}^{(l)} = f\left(\mathbf{W}^{(l)} \cdot \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right), \tag{2.6}$$

wobei:

- $\mathbf{h}^{(l-1)}$: Die Eingaben für die aktuelle Schicht l,
- $\mathbf{W}^{(l)}$: Die Gewichtsmatrix der Schicht l,
- $\mathbf{b}^{(l)}$: Der Bias-Vektor der Schicht l,
- f: Die Aktivierungsfunktion, die Nichtlinearitäten einführt.

Diese mathematische Darstellung erweitert die Berechnung eines einzelnen Neurons auf eine gesamte Schicht. Jede Schicht dient dazu, Merkmale aus den Eingabedaten zu extrahieren und an die nächste Schicht weiterzugeben (LeCun et al., 2015).

2.3.2 Feedforward Neural Network (FFN)

Feedforward Neural Networks (FFNs) sind eine der einfachsten Architekturen neuronaler Netzwerke und basieren auf der schichtweisen Verarbeitung von Eingaben. Informationen fließen dabei ausschließlich in eine Richtung – von der Eingabeschicht über eine oder mehrere versteckte Schichten bis zur Ausgabeschicht. Diese Architektur ist besonders geeignet für Probleme mit festen Eingabe-Ausgabe-Beziehungen, wie z. B. Klassifikation oder Regression (Goodfellow et al., 2016; Hornik, 1991).

Architektur eines FFN

Ein typisches FFN besteht aus drei zentralen Bestandteilen:

- **Eingabeschicht:** Diese Schicht nimmt die Rohdaten des Problems auf, z. B. eine Matrixdarstellung eines Labyrinths. Die Anzahl der Neuronen entspricht der Dimensionalität der Eingabedaten (LeCun et al., 2015).

- **Versteckte Schichten:** Diese führen Transformationen der Eingabedaten durch und lernen zunehmend abstrakte Merkmale, die für die Problemstellung relevant sind (Goodfellow et al., 2016).
- **Ausgabeschicht:** Sie liefert die Vorhersagen des Modells, z. B. den optimalen Pfad durch ein Labyrinth. Die Anzahl der Neuronen entspricht dabei den Anforderungen der Aufgabe (Hornik, 1991).

Einsatz von FFNs in der Pfadfindung

In dieser Arbeit wird ein FFN als Vergleichsmodell für die Evaluation der entwickelten FCAEs verwendet. Insbesondere für kleine Labyrinthe (z. B. 7x7) eignet sich die flache Eingabe eines FFN, da räumliche Beziehungen hier weniger relevant sind (Hornik, 1991; LeCun et al., 2015). Das FFN wurde trainiert, um die Beziehung zwischen ungelösten und gelösten Labyrinthrepräsentationen zu lernen. Trotz seiner einfachen Struktur erfordert das Training großer FFNs oft erhebliche Datenmengen, um zufriedenstellende Ergebnisse zu erzielen (Goodfellow et al., 2016).

Einschränkungen von FFNs in der Pfadfindung

Obwohl FFNs universell einsetzbar sind, zeigen sie spezifische Schwächen, insbesondere bei der Verarbeitung komplexer bildartiger Daten:

- **Fehlende räumliche Kontextinformation:** FFNs betrachten Eingaben unabhängig voneinander, wodurch räumliche Zusammenhänge in Labyrinthen verloren gehen können (LeCun et al., 2015).
- **Skalierungsprobleme:** Mit zunehmender Eingabegröße steigt die Anzahl der Verbindungen exponentiell, was den Rechenaufwand erheblich erhöht (Goodfellow et al., 2016).
- **Überanpassung:** Ohne geeignete Regularisierungstechniken neigen FFNs dazu, sich zu stark an Trainingsdaten anzupassen, was die Generalisierungsfähigkeit einschränkt (Srivastava et al., 2014).

Diese Einschränkungen unterstreichen die Relevanz spezialisierter Architekturen wie Convolutional Neural Network (CNN) oder FCAEs. Diese Architekturen nutzen die räumliche Struktur von Daten effizienter und werden in den folgenden Abschnitten detaillierter erläutert.

2.3.3 Convolutional Neural Network (CNN)

Ein CNN ist eine spezialisierte Architektur neuronaler Netzwerke, die für die Verarbeitung von Daten mit räumlicher Struktur entwickelt wurde. Ursprünglich für Bildverarbeitung konzipiert, haben sich CNNs als äußerst leistungsfähig in Bereichen wie Objekterkennung, medizinischer Bildanalyse und automatischer Sprachverarbeitung erwiesen (LeCun et al., 2015; Schmidhuber, 2015). Im Gegensatz zu klassischen FFNs, die Eingabedaten in einer eindimensionalen Vektorform darstellen, bewahren CNNs die räumlichen Zusammenhänge der Daten durch die Anwendung von Faltungsschichten. Diese Eigenschaft ist besonders vorteilhaft für die Lösung von Labyrinthen, da hier die relative Anordnung von Wänden, Wegen sowie Start- und Zielpunkten entscheidend ist.

Architektur eines CNNs

Die Architektur eines CNN basiert auf mehreren spezialisierten Schichten, die zusammenarbeiten, um Merkmale aus den Eingabedaten zu extrahieren und zu analysieren. Die wichtigsten Komponenten sind:

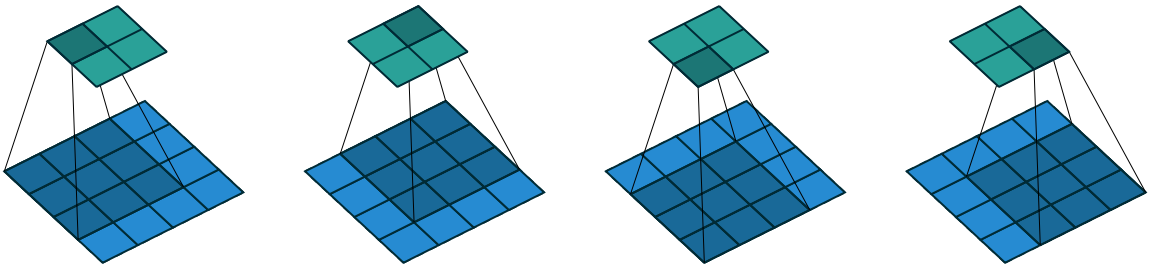
- **Faltungsschicht (Convolutional Layer)**: Extrahiert lokale Merkmale aus den Eingabedaten durch die Anwendung trainierbarer Filter.
- **Pooling-Schicht**: Reduziert die Dimensionalität der Merkmalskarten, um die Berechnungen effizienter zu gestalten.
- **Aktivierungsfunktion**: Führt Nichtlinearitäten ein, um komplexe Zusammenhänge in den Daten zu modellieren.
- **Fully Connected Layer (optional)**: Kann zur finalen Klassifikation oder Regression genutzt werden, wird aber in vollständig faltungsbasierten Architekturen (z. B. FCAEs) vermieden.

Faltungsschicht (Convolutional Layer)

Die Faltungsschicht ist die essenzielle Komponente eines CNN. Sie verwendet trainierbare Filter (*Kernel*), die über die Eingabedaten gleiten und lokale Merkmale extrahieren. Durch diese lokale Verarbeitung bleibt die räumliche Struktur der Daten erhalten (Dumoulin & Visin, 2018). Diese Fähigkeit ist für Anwendungen wie die Labyrinthlösung von großer Bedeutung, da Nachbarschaftsbeziehungen zwischen Gitterzellen berücksichtigt werden.

Abbildung 1 illustriert die Funktionsweise einer Faltung anhand eines 3×3-Kernels. Der Kernel gleitet über die Eingabedaten und berechnet an jeder Position eine gewichtete Summe der Pixelwerte, wobei die blauen Fläche die Eingabe und die türkise Fläche die Ausgabe darstellt.

Abbildung 1: Veranschaulichung der Faltung mit einem 3×3-Kernel



Quelle: Dumoulin und Visin, 2018

In Abbildung 2 ist dargestellt, wie die Faltung numerisch durchgeführt wird. Hier wurde die Kernelgröße 3 gewählt mit der Eingabegröße 5, wodurch die Ausgabe die Größe 3 hat. Die Eingabe wird mit den trainierbaren Parametern des Kernels multipliziert und summiert. Dadurch entsteht ein Feld der Ausgabe. In der Folge gleitet der Kernel weiter über die Eingabe. Initialisiert werden die Wert, wie andere Parameter in Neuronalen Netzwerken (Goodfellow et al., 2016).

Padding: Erhalt der Ausgangsdimension

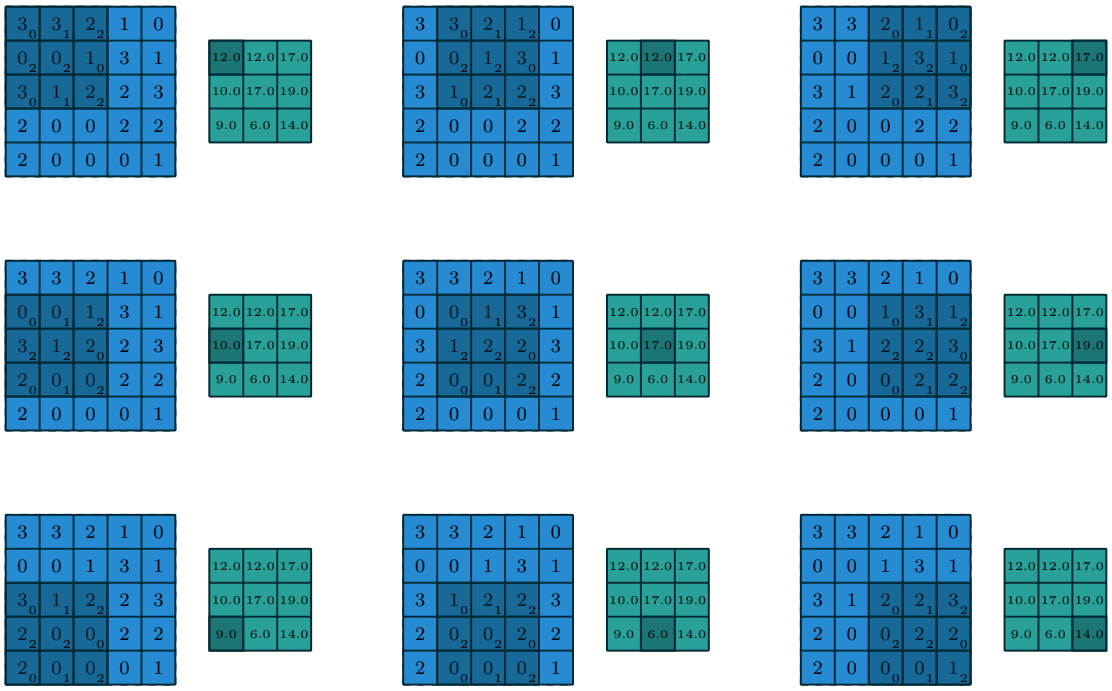
Da sich die Dimension der Eingabe durch die Faltung reduziert, wird oft Padding genutzt. Dabei werden zusätzliche Pixel an den Bildrändern eingefügt, um die ursprüngliche Dimension beizubehalten (Dumoulin & Visin, 2018). Abbildung 3 zeigt diesen Effekt.

Stride: Anpassung der Schrittweite

Der Stride bestimmt, um wie viele Pixel der Filter pro Schritt verschoben wird:

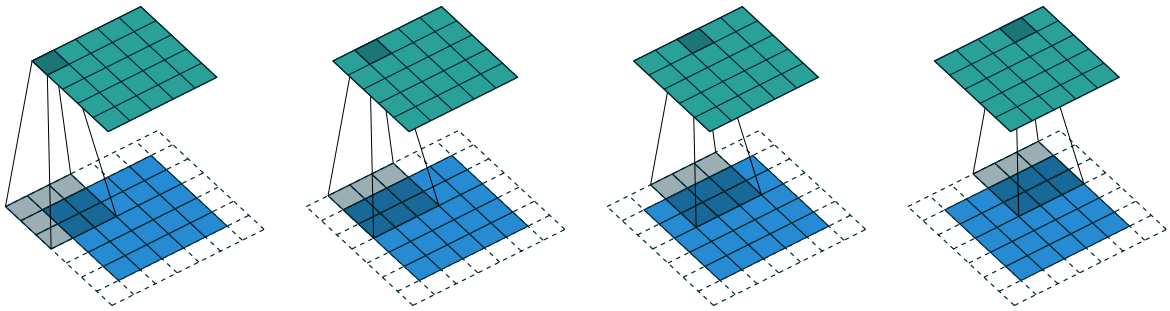
- Stride $= 1$: Der Kernel bewegt sich Pixel für Pixel.
- Stride > 1: Der Kernel springt mehrere Pixel weiter, wodurch die Ausgabegröße reduziert wird, was in Abbildung 4 gezeigt wird.

Abbildung 2: Numerische Darstellung der Faltung mit einem 3×3-Kernel



Quelle: Dumoulin und Visin, 2018

Abbildung 3: Padding bei der Faltung

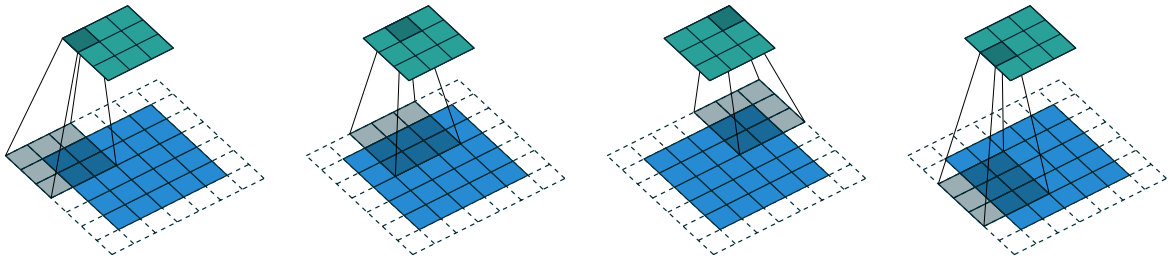


Quelle: Dumoulin und Visin, 2018

Einsatz von CNNs in der Pfadfindung

In dieser Arbeit werden CNNs zur Lösung von Labyrinthen eingesetzt, indem sie die räumliche Struktur der Eingabedaten nutzen, um Pfade vorherzusagen. Während klassische Algorithmen wie A* deterministische Suchverfahren nutzen, lernen CNNs aus Trainingsdaten, Muster zu erkennen und effiziente Lösungen zu approximieren (LeCun et al., 2015; Schmidhuber, 2015).

Abbildung 4: Einfluss der Schrittweite (Stride) 2 auf die Faltung



Quelle: Dumoulin und Visin, 2018

2.3.4 Autoencoder und Fully-Convolutional Autoencoder (FCAE)

Autoencoder sind neuronale Netzwerke, die darauf ausgelegt sind, Eingabedaten zu komprimieren und anschließend möglichst originalgetreu zu rekonstruieren. Dies geschieht durch eine Reduzierung der Daten auf eine latente Darstellung, aus der die ursprünglichen Informationen wiederhergestellt werden. Sie bestehen aus zwei Hauptkomponenten:

- **Encoder:** Der Encoder transformiert die Eingabe in eine komprimierte Darstellung im sogenannten Latent Space, indem er relevante Merkmale extrahiert.
- **Decoder:** Der Decoder rekonstruiert aus dieser latenten Darstellung die ursprünglichen Eingabedaten.

Autoencoder werden häufig in Bereichen wie Merkmalsextraktion, Datenkomprimierung und Rauschunterdrückung eingesetzt (Goodfellow et al., 2016; Masci et al., 2011). Die Qualität der Rekonstruktion wird durch eine Verlustfunktion bewertet, die den Unterschied zwischen Eingabe und rekonstruierten Daten minimiert, häufig unter Verwendung des Mean Squared Error (MSE).

Convolutional Autoencoder (CAE)

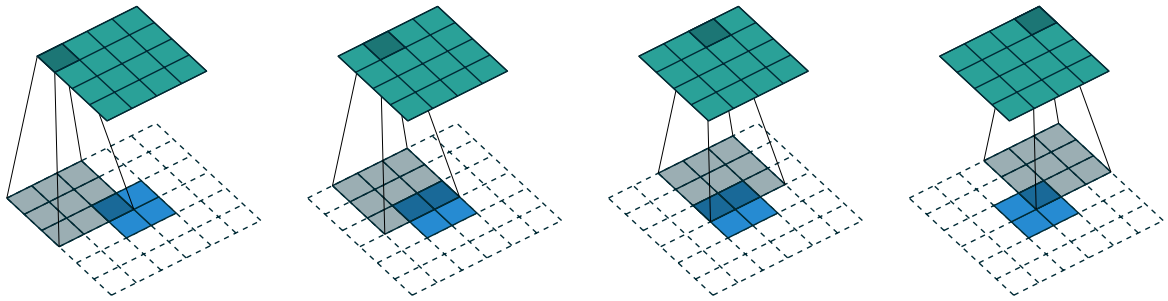
CAEs erweitern klassische Autoencoder, indem sie Faltungsschichten (Convolutional Layers) zur Verarbeitung von Eingabedaten verwenden. Diese Architektur eignet sich besonders für bildartige Daten, da Faltungsschichten räumliche Merkmale wie Kanten, Muster und Strukturen extrahieren (LeCun et al., 2015; Masci et al., 2011). Der Encoder eines CAE besteht aus mehreren Faltungsschichten, die die Eingaben schrittweise komprimieren, während der Decoder die ursprünglichen Daten rekonstruiert.

Transponierte Faltung (Transposed Convolution)

Die transponierte Faltung ist eine zentrale Operation im Decoder von CAEs. Sie dient dazu, die durch die Faltungsschichten im Encoder verkleinerten Merkmale wieder auf die ursprüngliche Eingabegröße zu rekonstruieren und kann also als "umgekehrte" Faltung betrachtet werden (Dumoulin & Visin, 2018).

Abbildung 5 veranschaulicht die Funktionsweise einer transponierten Faltung. Dabei wird ein kleiner Eingabetensor schrittweise vergrößert, indem trainierbare Filter auf die Eingabe angewendet werden. Im Gegensatz zu einfachen Upsampling-Methoden lernt die transponierte Faltung direkt aus den Daten, welche Merkmale rekonstruiert werden sollen (Masci et al., 2011).

Abbildung 5: Veranschaulichung der transponierten Faltung mit einem 3×3-Kernel



Quelle: Dumoulin und Visin, 2018

Einfluss von Stride auf die transponierte Faltung

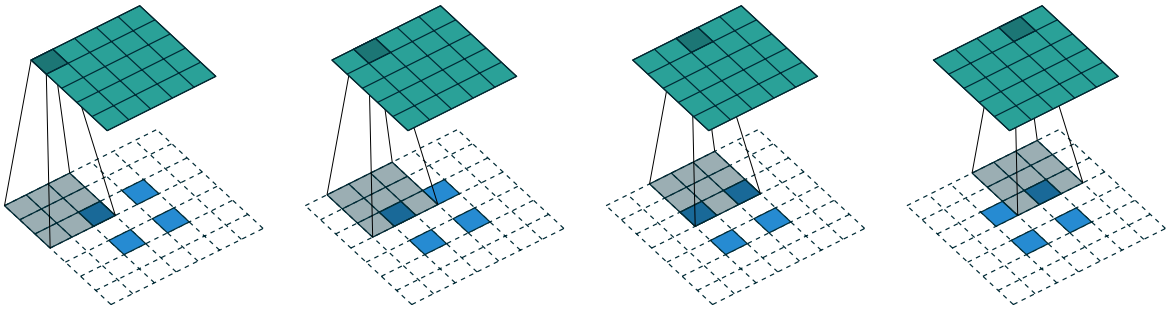
Ein wesentlicher Unterschied zur normalen Faltung ist der Einfluss des Strides. Während eine größere Schrittweite in einer herkömmlichen Faltung die Ausgabe verkleinert, führt eine größere Schrittweite in der transponierten Faltung zu einer Vergrößerung der Ausgabegröße.

Abbildung 6 zeigt diesen Effekt:

Fully-Convolutional Autoencoder (FCAE)

Fully-Convolutional Autoencoders (FCAEs) stellen eine Variante von CAEs dar. Sie lassen vollständig verbundene Schichten (Fully Connected Layers) vollständig aus. Dadurch bleiben die räumlichen Beziehungen in den Daten durchgängig erhalten, was für bildartige Eingaben vorteilhaft ist (Masci et al., 2011).

Abbildung 6: Einfluss der Schrittweite (Stride) auf die transponierte Faltung



Quelle: Dumoulin und Visin, 2018

Vorteile von FCAEs gegenüber klassischen CAEs

Der wesentliche Unterschied zwischen FCAEs und CAEs besteht darin, dass die latente Darstellung in einem FCAE weiterhin eine mehrdimensionale Form besitzt, anstatt in einen eindimensionalen Vektor umgewandelt zu werden. Dies bringt mehrere Vorteile:

- **Erhalt der räumlichen Struktur:** Durch den Verzicht auf vollständig verbundene Schichten bleibt die räumliche Beziehung der Daten während des gesamten Prozesses erhalten.
- **Reduzierte Parameteranzahl:** Die Anzahl der trainierbaren Parameter wird erheblich gesenkt, was Speicherbedarf und Trainingszeit verringert.
- **Verbesserte Skalierbarkeit:** FCAEs können effizient auf größere Eingaben angewendet werden, da keine feste Vektorlänge erzwungen wird.

Die Architektur eines FCAE besteht ausschließlich aus Faltungsschichten und transponierten Faltungsschichten. Der Encoder extrahiert Merkmale und komprimiert die Eingabedaten in einer strukturierten, mehrdimensionalen latenten Darstellung. Der Decoder rekonstruiert die ursprüngliche Eingabe, indem er diese Darstellung schrittweise zurückwandelt. Dadurch eignen sich FCAEs besonders für bildartige Daten wie Labyrinthe, da die räumliche Struktur durchgehend erhalten bleibt (Dumoulin & Visin, 2018; Masci et al., 2011).

Relevanz für die Arbeit

Die Anwendung von FCAEs in dieser Arbeit bietet entscheidende Vorteile für die Pfadfindung in Labyrinthen:

- **Direkte Verarbeitung von Labyrinthdaten:** Die Labyrinthe können direkt als Bilddaten behandelt werden, ohne dass eine aufwändige Vorverarbeitung erforderlich ist.

- **Erhalt der räumlichen Beziehungen:** Durch die Nutzung reiner Faltungsschichten bleibt die Struktur der Labyrinthe erhalten, was für die korrekte Identifikation von Wegen entscheidend ist.

Da klassische Algorithmen wie A* deterministisch arbeiten und exakte Lösungen berechnen, während FCAEs auf Mustern aus Trainingsdaten basieren, wird in dieser Arbeit untersucht, ob FCAEs als Alternative zur klassischen Pfadfindung dienen können.

2.3.5 Trainingsprozesse und Optimierung

Das Training eines neuronalen Netzwerks, einschließlich FFNs, CNNs und FCAEs, basiert auf der iterativen Anpassung von Gewichten und Bias-Werten, um eine optimale Übereinstimmung zwischen den Vorhersagen des Netzwerks und den Zielwerten (Labels) zu erreichen. Ziel des Trainings ist die Minimierung einer Verlustfunktion, die den Fehler zwischen den Vorhersagen und den Zielwerten quantifiziert (Goodfellow et al., 2016; LeCun et al., 2015).

Grundlegender Trainingsprozess

Der Trainingsprozess eines neuronalen Netzwerks umfasst die folgenden Schritte:

1. **Forward Propagation:** Die Eingabedaten durchlaufen die verschiedenen Schichten des Netzwerks, um die Ausgaben (Vorhersagen) zu berechnen.
2. **Fehlerberechnung:** Eine Verlustfunktion, wie der Mean Squared Error (MSE) oder die Cross-Entropy, berechnet den Unterschied zwischen den Vorhersagen und den Zielwerten.
3. **Backward Propagation:** Mithilfe des Backpropagation-Algorithmus werden die Gradienten der Verlustfunktion in Bezug auf die Gewichte und Bias-Werte berechnet (Rumelhart et al., 1986).
4. **Aktualisierung der Parameter:** Die Gewichte und Bias-Werte werden durch Optimierungsverfahren wie Stochastic Gradient Descent (SGD) oder Adam angepasst (Kingma & Ba, 2014).

Dieser Prozess wird iterativ über mehrere Epochen wiederholt, bis das Netzwerk eine zufriedenstellende Leistung auf den Trainingsdaten erreicht (Goodfellow et al., 2016).

Optimierungsverfahren

Die Wahl des Optimierungsverfahrens beeinflusst maßgeblich die Geschwindigkeit und Qualität des Trainingsprozesses. Zu den gängigen Optimierungsverfahren gehören:

- **Stochastic Gradient Descent (SGD):** Ein grundlegendes Verfahren, das die Parameter in Richtung des negativen Gradienten der Verlustfunktion aktualisiert. Obwohl SGD einfach und effizient ist, kann es bei komplexen Optimierungsproblemen langsam konvergieren.
- **Adam (Adaptive Moment Estimation):** Eine Erweiterung von SGD, die adaptiv lernratenbasierte Updates verwendet. Adam kombiniert die Vorteile von Momentum und adaptiven Lernraten, was zu einer schnelleren und stabileren Konvergenz führt (Kingma & Ba, 2014).

Verlustfunktionen

Die Wahl der Verlustfunktion hängt von der Aufgabenstellung ab:

- **Mean Squared Error (MSE):** Diese Verlustfunktion wird häufig bei Regressionsproblemen eingesetzt, da sie den quadratischen Fehler zwischen den Vorhersagen und den Zielwerten misst.
- **Cross-Entropy Loss:** Geeignet für Klassifikationsprobleme, da sie die Wahrscheinlichkeit der korrekten Klasse maximiert (Goodfellow et al., 2016).

Für die FCAEs dieser Arbeit wird der MSE verwendet, um die Qualität der Rekonstruktion von Labyrinthdaten zu bewerten.

Regularisierungstechniken

Um Überanpassung (Overfitting) an die Trainingsdaten zu vermeiden, werden verschiedene Regularisierungstechniken eingesetzt:

- **Dropout:** Eine zufällige Deaktivierung von Neuronen während des Trainings, um die Abhängigkeit des Netzwerks von einzelnen Neuronen zu reduzieren (Srivastava et al., 2014).
- **L2-Regularisierung:** Fügt der Verlustfunktion eine Strafe für große Gewichtswerte hinzu, um die Modellkomplexität zu verringern.

Hyperparameter-Tuning

Das Training eines neuronalen Netzwerks erfordert die Abstimmung verschiedener Hyperparameter, darunter:

- **Lernrate:** Bestimmt die Größe der Parameter-Updates in jeder Iteration.
- **Batchgröße:** Die Anzahl der Trainingsbeispiele, die in einer Iteration verarbeitet werden.
- **Anzahl der Epochen:** Die Anzahl der vollständigen Durchläufe durch den Trainingsdatensatz.

In dieser Arbeit wurden gängige Standardwerte für die Hyperparameter gewählt, da kein umfangreiches Hyperparameter-Tuning durchgeführt wurde (Goodfellow et al., 2016).

Validierung und Testen

Während des Trainingsprozesses wird die Leistung des Netzwerks regelmäßig auf einem separaten Validierungsdatensatz überprüft, um sicherzustellen, dass das Modell nicht überangepasst wird. Nach Abschluss des Trainings wird das Modell auf einem Testdatensatz evaluiert, um seine Generalisierungsfähigkeit zu beurteilen (Goodfellow et al., 2016).

Zusammenfassung der Hyperparameter

In dieser Arbeit wurde Adam als Optimierungsverfahren gewählt, da es durch seine adaptiven Lernraten und Momentum-basierte Updates eine effiziente und stabile Konvergenz ermöglicht. Die Verlustfunktion MSE wurde verwendet, um die Rekonstruktion der Labyrinthe zu bewerten. Zudem wurden Regularisierungstechniken wie Dropout und L2-Regularisierung eingesetzt, um die Generalisierungsfähigkeit der Modelle sicherzustellen (Kingma & Ba, 2014; Srivastava et al., 2014).

2.4 Labyrinth-Generierung und Datensätze

Labyrinthe bilden die zentrale Datenbasis für die Evaluierung der in dieser Arbeit untersuchten Algorithmen und neuronalen Netzwerke. Die Struktur und Komplexität der Labyrinthe spielen eine entscheidende Rolle für die Vergleichbarkeit und Aussagekraft der Experimente, da sie die Leistungsfähigkeit der getesteten Modelle und Algorithmen unmittelbar beeinflussen.

Alle Labyrinthe werden als zweidimensionale Tensoren dargestellt, um eine effiziente Verarbeitung durch neuronale Netzwerke zu gewährleisten. Diese Darstellung ermöglicht es, die räumliche Struktur der Daten direkt zu nutzen, was insbesondere für CNNs und FCAEs von Vorteil ist.

2.4.1 Verfahren zur Labyrinth-Generierung

In dieser Arbeit wurden zwei unterschiedliche Ansätze zur Generierung von Labyrinthen verwendet, die sich hinsichtlich ihrer Komplexität und Anwendungszwecke unterscheiden:

Einfache 7x7-Labyrinthe durch zufällige Platzierung von Hindernissen

Für die 7×7-Labyrinthe wurden Hindernisse zufällig auf einem Gitter platziert, während Start- und Endpunkte manuell definiert wurden (Vlad, 2023). Diese einfachen Labyrinthe zeichnen sich durch ihre geringe Komplexität aus und eignen sich besonders für die Evaluation von Grundfähigkeiten der neuronalen Netzwerke. Sie wurden verwendet, um FCNNs und FCAEs auf kleinen und überschaubaren Datenstrukturen zu vergleichen.

Komplexere Labyrinthe durch den Aldous-Broder-Algorithmus

Für größere Labyrinthe in den Größen 10×10 bis 50×50 wurde der Aldous-Broder-Algorithmus verwendet. Dieser Algorithmus basiert auf einem Random Walk und erzeugt sogenannte perfekte Labyrinthe, bei denen alle Zellen durch genau einen Pfad verbunden sind, ohne Zyklen zu enthalten. Diese Eigenschaften machen die generierten Labyrinthe eindeutig lösbar, was für die Evaluierung von Pfadfindungsalgorithmen von entscheidender Bedeutung ist (Grabrovsek, 2019; Loizou & Kumar, 2006; Loomis Jr., 2012).

Der Aldous-Broder-Algorithmus funktioniert wie folgt:

1. Eine zufällige Startzelle im Gitter wird ausgewählt.
2. Der Algorithmus bewegt sich zufällig zu einer benachbarten Zelle.
3. Wenn die Nachbarzelle noch nicht besucht wurde, wird sie mit der aktuellen Zelle verbunden, indem die Wand zwischen den beiden entfernt wird. Die Nachbarzelle wird als besucht markiert.
4. Dieser Vorgang wird so lange wiederholt, bis alle Zellen im Labyrinth besucht wurden.

Das Ergebnis ist ein Spannbaum, der alle Zellen des Gitters miteinander verbindet, ohne dass Zyklen entstehen. Dies garantiert die Lösbarkeit des Labyrinths und sorgt dafür, dass es genau einen Pfad zwischen zwei beliebigen Punkten gibt. Die gleichmäßige Verteilung des Algorithmus stellt sicher, dass alle möglichen Spannbäume mit gleicher Wahrscheinlichkeit generiert werden (Loizou & Kumar, 2006; Loomis Jr., 2012).

Vergleich der Verfahren

Die zufällige Platzierung von Hindernissen erzeugt einfache und weniger anspruchsvolle Labyrinthe, die ideal für die Evaluation von Netzwerken auf kleinen Eingabedaten sind. Im Gegensatz dazu erlaubt der Aldous-Broder-Algorithmus die Generierung komplexerer Labyrinthe, die eine größere Herausforderung darstellen. Diese perfekt generierten Labyrinthe ermöglichen es, die Robustheit und Skalierbarkeit der Algorithmen und neuronalen Netzwerke unter realistischeren Bedingungen zu testen.

2.4.2 Struktur und Format der Datensätze

Die in dieser Arbeit verwendeten Labyrinthe werden in Form von zweidimensionalen Tensoren dargestellt, um eine effiziente Verarbeitung in neuronalen Netzwerken zu ermöglichen.

Kodierung der Datensätze

Sowohl die bestehenden 7×7-Labyrinthe als auch die neu generierten 10×10 bis 50×50 großen Labyrinthe haben die gleiche Kodierung, die von Vlad, 2023 übernommen wurde. Die Kodierung der Tensoren erfolgt wie folgt:

- 0: Freie Wege.
- 1: Hindernisse.
- 3: Start- und Endpunkt.

Die zugehörigen Lösungen, die ebenfalls als Tensoren vorliegen, enthalten folgende Kodierungen:

- 0: Freie Wege.
- 2: Optimaler Pfad.
- 3: Start- und Endpunkt.

Relevanz der Datenstruktur

Die klar definierte Datenstruktur ermöglicht eine eindeutige Trennung zwischen Eingabedaten (ungelöste Labyrinthe) und Ausgabedaten (optimale Lösungen). Diese Struktur erleichtert das Training der neuronalen Netzwerke, da die Modelle die zugrunde liegenden Muster direkt lernen können. Darüber hinaus stellt die konsistente Kodierung sicher, dass unterschiedliche Ansätze – von FCAEs und FCNNs bis hin zum A*-Algorithmus – auf denselben Datensätzen evaluiert werden können, wodurch eine direkte Vergleichbarkeit der Ergebnisse gewährleistet ist. Die Kombination aus einfachen und komplexen Labyrinthen ermöglicht eine umfassende Bewertung der Leistung der untersuchten Modelle.

2.5 Evaluationsmetriken für Pfadfindungsalgorithmen

Die Evaluierung der in dieser Arbeit untersuchten Ansätze basiert auf einer Reihe von Metriken, die verschiedene Aspekte der Leistungsfähigkeit bewerten. Diese Metriken wurden so gewählt, dass sie die Genauigkeit, Effizienz und Ressourcennutzung der Modelle und Algorithmen abdecken (Cui & Shi, 2011; Russell & Norvig, 2016). Im Folgenden werden die verwendeten Metriken definiert und deren Anwendung im Kontext dieser Arbeit beschrieben.

2.5.1 Definition und Relevanz der Metriken

Pfadgenauigkeit:

Die Pfadgenauigkeit misst den Anteil der von einem Ansatz berechneten Pfade, die exakt mit der optimalen Lösung übereinstimmen. Sie gibt direkt Aufschluss über die Fähigkeit eines Modells, die zugrunde liegenden Muster der Trainingsdaten zu lernen und auf neue Daten anzuwenden. Die Berechnung erfolgt durch den Vergleich des modellierten Pfads mit der optimalen Lösung (Hart et al., 1968; Russell & Norvig, 2016).

Inferenzzeit:

Die Inferenzzeit bezeichnet die durchschnittliche Dauer, die ein Modell benötigt, um eine Lösung für ein Labyrinth zu berechnen. Sie dient als Indikator für die Effizienz eines Modells und dessen Eignung für zeitkritische Anwendungen wie Robotik oder Echtzeitspiele (Cui & Shi, 2011; Loizou & Kumar, 2006).

Speicherbedarf:

Der Speicherbedarf beschreibt die während der Berechnung genutzten Ressourcen, sei es GPU-Speicher (bei neuronalen Netzwerken) oder RAM (bei klassischen Algorithmen). Diese Metrik ist entscheidend für die Skalierbarkeit und für Anwendungen auf ressourcenbeschränkten Geräten (Grabrovsek, 2019; LeCun et al., 2015).

Parameteranzahl:

Die Parameteranzahl gibt die Anzahl der trainierbaren Gewichte und Bias-Werte eines Modells an. Sie dient als Maß für die Komplexität und beeinflusst direkt den Speicherbedarf und die Berechnungszeit des Netzwerks (Goodfellow et al., 2016).

Modellgröße:

Die Modellgröße beschreibt die Speichergröße des trainierten Modells und wird in Megabyte (MB) gemessen. Sie ist ein wichtiger Faktor für die Anwendbarkeit in mobilen oder eingebetteten Systemen (Schmidhuber, 2015).

2.5.2 Metriken im Vergleich neuronaler Netzwerke

Die oben definierten Metriken werden verwendet, um die verschiedenen Architekturen neuronaler Netzwerke (z. B. FCNN und FCAE) untereinander zu vergleichen. Dabei liegt der Fokus auf folgenden Aspekten:

- Die **Pfadgenauigkeit** bewertet die Fähigkeit des Modells, korrekte und präzise Labyrinthlösungen zu berechnen.
- Die **Inferenzzeit** zeigt, wie effizient ein Modell arbeitet und wie gut es sich für zeitkritische Anwendungen eignet (Cui & Shi, 2011; Russell & Norvig, 2016).
- Der **Speicherbedarf** auf der GPU wird analysiert, um die Skalierbarkeit der Modelle, insbesondere bei größeren Labyrinthen, zu bewerten (LeCun et al., 2015).
- Die **Parameteranzahl** und die **Modellgröße** geben Aufschluss über die Komplexität und den Speicherbedarf des Netzwerks (Goodfellow et al., 2016).

2.5.3 Metriken im Vergleich mit dem A*-Algorithmus

Der Vergleich zwischen neuronalen Netzwerken und dem A*-Algorithmus basiert auf denselben Metriken, jedoch mit unterschiedlichem Fokus:

- Die **Pfadgenauigkeit** prüft, wie gut neuronale Netzwerke die vom A*-Algorithmus berechneten optimalen Lösungen reproduzieren können (Hart et al., 1968).
- Die **Inferenzzeit** wird genutzt, um die Effizienz von GPU-basierten Netzwerken und dem CPU-basierten A*-Algorithmus bei wachsenden Labyrinthgrößen zu vergleichen (Loizou & Kumar, 2006).
- Der **Speicherbedarf** wird zwischen GPU (für neuronale Netzwerke) und Hauptspeicher (für A*) gegenübergestellt, um die Ressourceneffizienz der Ansätze zu analysieren (LeCun et al., 2015).

Zusammenfassung der Metriken:

Die hier definierten Metriken bieten eine umfassende Grundlage zur Bewertung der Modelle und Algorithmen. Während neuronale Netzwerke das Potenzial zeigen, den A*-Algorithmus in einigen Szenarien zu ersetzen, bieten die Effizienzmetriken Einblicke in ihre Anwendbarkeit in realen Umgebungen (Cui & Shi, 2011).

3 Modellierung und Implementierung

Dieses Kapitel beschreibt die konkrete Umsetzung der Modelle und Algorithmen, die in dieser Arbeit untersucht wurden. Dabei wird zunächst die Erstellung der verwendeten Datensätze erläutert, gefolgt von einer detaillierten Darstellung der Architektur der neuronalen Netzwerke. Anschließend wird auf die Trainingsprozesse der Modelle eingegangen, bevor die Implementierung des A*-Algorithmus beschrieben wird.

3.1 Labyrinth-Generierung und -Daten

Für die Experimente in dieser Arbeit wurden zwei unterschiedliche Datensätze verwendet. Diese unterscheiden sich sowohl in der Methode ihrer Generierung als auch in ihrem Einsatzzweck. Während die bestehenden 7×7-Labyrinthe vor allem für den Vergleich zwischen FCNNs und FCAEs genutzt wurden, dienen die eigens generierten größeren Labyrinthe dem Vergleich zwischen FCAEs und dem A*-Algorithmus.

3.1.1 Datensätze für den Vergleich zwischen FCNN und FCAE

Die 7×7-Labyrinthe stammen aus bestehenden Datensätzen, die von Vlad, 2023 bereitgestellt wurden. Dieser Datensatz diente als Grundlage für den Vergleich zwischen dem bestehenden FCNN und dem in dieser Arbeit entwickelten FCAEs. Die Hauptmerkmale der Labyrinthe und deren Kodierung wurden bereits in Abschnitt 2.4.2 erläutert.

Für die Experimente wurden die Labyrinthe sowohl in ungelöster als auch in gelöster Form bereitgestellt. Die Lösungen wurden mithilfe des A*-Algorithmus generiert, um den optimalen Pfad zu berechnen. Das FCAE wurde darauf trainiert, die ungelösten Labyrinthe direkt in ihre Lösungen zu überführen, während das FCNN auf dieselbe Aufgabe optimiert wurde. Dabei wurden die Start- und Endpunkte beibehalten, die Hindernisstruktur jedoch entfernt.

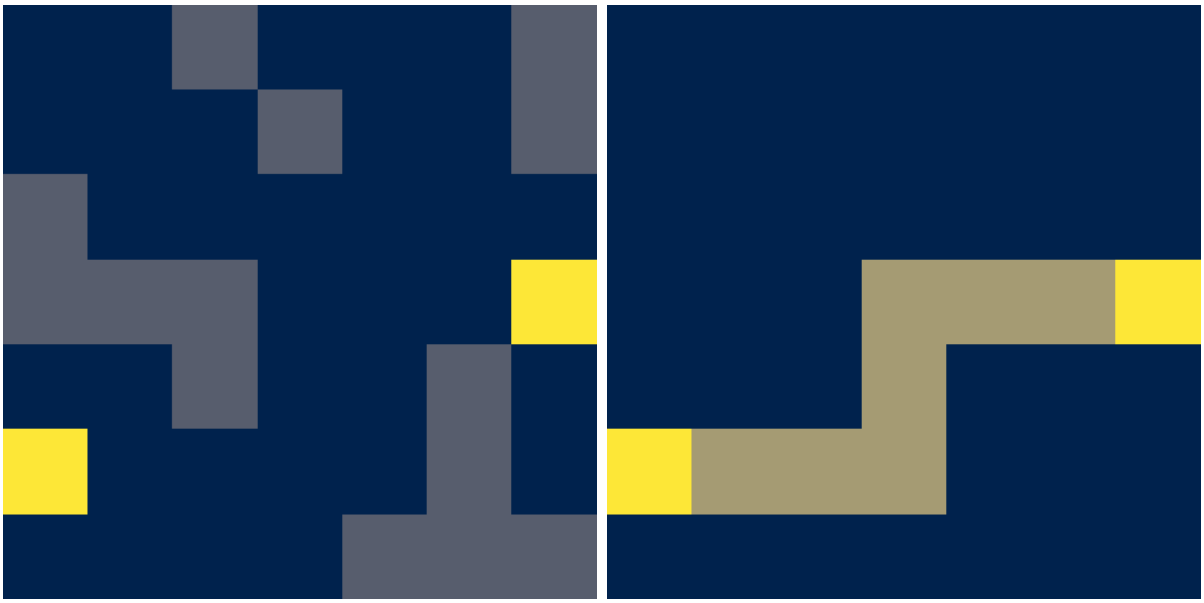
Der Datensatz setzt sich aus drei Hauptpartitionen zusammen:

- **Trainingsdatensatz**: Bestehend aus 63.052 Labyrinthen (`X.dat.npy`, `Y.dat.npy`). Dieser Datensatz wurde für das Training des Modells genutzt.
- **Validierungsdatensatz**: Enthält 12.566 Labyrinthe (`X.dat_smol.npy`, `Y.dat_smol.npy`). Er wurde zur Überwachung der Modellleistung während des Trainings verwendet.

- **Testdatensatz**: Umfasst 31.455 Labyrinthe (`X.dat_test.npy`, `Y.dat_test.npy`). Dieser wurde für die finale Evaluierung des Modells herangezogen.

Eine Veranschaulichung dieser Labyrinthe ist in Abbildung 7 dargestellt.

Abbildung 7: Veranschaulichung eines ungelösten und gelösten 7x7-Labyrinth



3.1.2 Datensätze für den Vergleich zwischen FCAE und A*

Für den Vergleich zwischen FCAE und dem klassischen A*-Algorithmus wurden größere Labyrinthe in den Größen 10×10, 20×20, 30×30, 40×40 und 50×50 generiert. Die Generierung erfolgte mithilfe des Aldous-Broder-Algorithmus, der eine gleichmäßige Verteilung der Pfade gewährleistet. Jedes Labyrinth enthält genau einen durchgehenden Pfad von einem Start- zu einem Endpunkt. Details zur Funktionsweise dieses Algorithmus wurden in Abschnitt 2.4.1 ausführlich beschrieben.

Der für das Training der FCAE-Modelle verwendete Datensatz umfasst insgesamt 100.000 Labyrinthe pro Labyrinthgröße. Die Daten wurden in 80 % Trainingsdaten und 20 % Testdaten aufgeteilt. Die Entscheidung, die Datensatzgröße konstant zu halten, wurde aufgrund von Speicher- und Rechenkapazitäten getroffen. Größere Labyrinthe erfordern nicht nur eine höhere Modellkapazität, sondern führen auch zu einem Anstieg des Speicherbedarfs und der Rechenzeit, insbesondere während des Trainingsprozesses. Diese Skalierungsprobleme sind in Absatz 3.3.3 näher erläutert.

Für den späteren Vergleich zwischen den trainierten FCAE-Modellen und dem A*-Algorithmus wurde aus den gesamten 100.000 Labyrinthen pro Größe zusätzlich eine zufällige Stichprobe

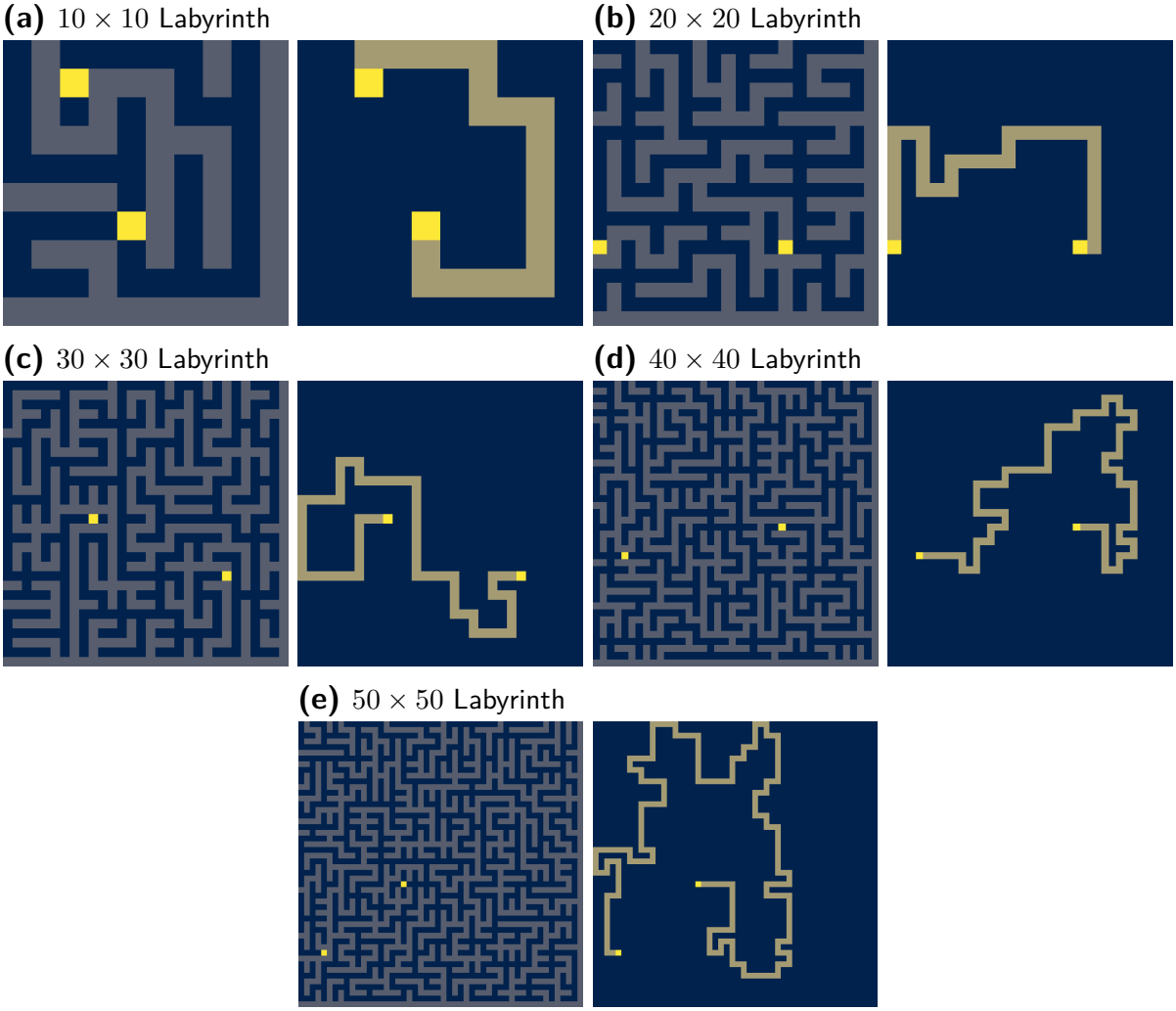
von 20.000 Labyrinthen ausgewählt. Diese unabhängigen Evaluierungsdatensätze stellten sicher, dass die Modelle nicht nur auf den Testdaten aus der Trainingsphase geprüft wurden, sondern auch auf einer zufällig gewählten Teilmenge, die eine realistische Leistungseinschätzung erlaubt.

Theoretisch würde eine Erhöhung der Anzahl der Trainingslabyrinthe insbesondere für größere Labyrinthgrößen die Generalisierungsfähigkeit der Modelle weiter verbessern. Mehr Daten ermöglichen eine robustere Repräsentation der unterschiedlichen Strukturen und Wege innerhalb der Labyrinthe, was zu einer genaueren Pfadvorhersage führen kann. Gleichzeitig würde dies jedoch die Rechenanforderungen weiter erhöhen, was in der Praxis durch die verfügbare Hardware limitiert wird.

Die größeren Labyrinthgrößen ermöglichen dennoch eine detaillierte Analyse der Skalierbarkeit und Effizienz der FCAE-Modelle im Vergleich zum A*-Algorithmus. Zusätzlich erlauben sie eine Untersuchung der Grenzen der Modellarchitekturen, insbesondere in Bezug auf Genauigkeit, Inferenzzeit und Speicherbedarf.

Eine Darstellung aller Labyrinthgrößen mit Lösung zeigt Abbildung 8

Abbildung 8: Veranschaulichung der Labyrinthe 10×10 bis 50×50 mit Lösung



Eigene Darstellung

3.2 Architektur der Neuronalen Netze

Im Folgenden werden die Architekturen des FCNN sowie der FCAEs detailliert beschrieben.

3.2.1 Architektur des FCNN-Modells

Das in dieser Arbeit verwendete FCNN wurde von Vlad, 2023 entwickelt und dient als Vergleichsmodell für die Bewertung der neu entwickelten FCAEs. Um die Labyrinthe im Modell zu verarbeiten, werden die 7×7-Labyrinthe zunächst in einen flachen Vektor der Größe 1×49

umgewandelt. Die Ausgabe des Modells wird ebenfalls in dieser flachen Form zurückgegeben, bevor sie in die ursprüngliche Matrixform zurücktransformiert wird.

Übersicht der Varianten des FCNN

In der Originalquelle wurden mehrere Varianten des FCNN entwickelt, die sich hinsichtlich der Parameteranzahl, der verwendeten Aktivierungsfunktionen und der erzielten Genauigkeit unterscheiden. Tabelle 1 bietet eine Übersicht über diese Varianten und deren Leistung.

Tabelle 1: Übersicht über verschiedene FCNN-Architekturen und ihre Leistung

Parameteranzahl	Bezeichnung	Genauigkeit (%)
9.209	Small / ReLU	38,42
11.659	+2L / ReLU	40,67
11.843	Sigmoid Pre out / ReLU	56,11
11.843	Trained Thick & Sparse Mazes / ReLU	54,30
27.089	Sigmoid Out + Dropout Regularization	66,95
30.559	More Layers / Linear	85,47
61.644	More Layers V2 / Linear	88,26
59.094	More Layers V3 / Linear	90,31
1.519.074	Large NN (30ep) / Linear	97,55
1.519.074	Large NN V2 + L2 (300ep) / Linear	97,93

Quelle: Vlad, 2023

Die Tabelle zeigt, dass Modelle mit einer höheren Anzahl an Parametern in der Regel eine bessere Genauigkeit erzielen. Besonders bemerkenswert ist der Einfluss zusätzlicher Schichten und Regularisierungstechniken wie L2-Regularisierung und Dropout auf die Leistung der Modelle. Während einfache Architekturen wie das Small / ReLU-Modell mit nur 9.209 Parametern eine Genauigkeit von 38,42

Erklärung der Modellbezeichnungen

Die Bezeichnungen der Modelle geben Hinweise auf ihre Architektur:

- Small / ReLU: Ein kleines Modell mit einer einfachen Architektur, das die ReLU-Aktivierungsfunktion verwendet.
- +2L / ReLU: Erweiterung des Small-Modells um zwei zusätzliche Schichten.

- Sigmoid Pre out / ReLU: Verwendung der Sigmoid-Aktivierungsfunktion vor der Ausgabeschicht, während in den versteckten Schichten ReLU genutzt wird.
- Trained Thick & Sparse Mazes / ReLU: Ein Modell, das speziell auf unterschiedlich dichten Labyrinthen trainiert wurde.
- Sigmoid Out + Dropout Regularization: Nutzung der Sigmoid-Aktivierungsfunktion in der Ausgabeschicht sowie Dropout.
- More Layers / Linear: Ein Modell mit mehr versteckten Schichten, das eine lineare Aktivierungsfunktion in der Ausgabeschicht verwendet, anstelle von ReLU verwendet.
- More Layers V2 / Linear: Eine erweiterte Version des vorherigen Modells mit einer optimierten Schichtstruktur.
- More Layers V3 / Linear: Eine weitere Version mit zusätzlichen Anpassungen in der Architektur.
- Large NN (30ep) / Linear: Ein sehr großes Modell mit einer linearen Aktivierungsfunktion, das über 30 Epochen trainiert wurde.
- Large NN V2 + L2 (300ep) / Linear: Eine optimierte Version des großen Modells, die über 300 Epochen trainiert wurde und L2-Regularisierung nutzt.

Beispielarchitektur: Small / ReLU

Als Beispiel für ein einfaches FCNN-Modell wird die Architektur des ersten Eintrags in Tabelle 1 genauer betrachtet:

- Eingabeschicht: Besteht aus 49 Neuronen, die das flache Eingabeformat des 7×7-Labyrinths repräsentieren.
- Versteckte Schichten: Das Modell enthält 6 vollständig verbundene Schichten, die 45, 40, 25, 20, 25 und 40 Neuronen umfassen, wovon jede Schicht die ReLU-Aktivierungsfunktion verwendet.
- Ausgabeschicht: 49 Neuronen, die das transformierte Labyrinth wieder in der flachen Darstellung ausgeben.

Mit nur 9.209 Parametern bleibt die Modellkapazität begrenzt, was sich in einer relativ niedrigen Genauigkeit von 38,42 % widerspiegelt.

Architektur des Vergleichsmodells

Das in dieser Arbeit verwendete Vergleichsmodell Large NN V2 + L2 (300ep) stellt die leistungsstärkste Variante der in Tabelle 1 aufgeführten FCNNs dar. Es verfügt über 1.519.074 trainierbare Parameter und erreicht eine Genauigkeit von 97,93 %. Die Architektur dieses Modells setzt sich wie folgt zusammen:

- Eingabeschicht: Diese verarbeitet den flachen Eingabevektor der Größe 1×49, der aus einem Labyrinth generiert wurde.
- Versteckte Schichten: Das Modell umfasst fünf vollständig verbundene Schichten mit 1235, 768, 532, 149 und 98 Neuronen. Jede Schicht verwendet die ReLU-Aktivierungsfunktion, um nicht-lineare Muster zu lernen.
- Ausgabeschicht: Diese gibt ebenfalls einen flachen Vektor der Größe 1×49 aus, der anschließend zurück in die ursprüngliche Matrixform 7×7 transformiert wird.

Die große Anzahl an Parametern erlaubt es dem Modell, deutlich komplexere Abhängigkeiten und Muster innerhalb der Labyrinthstruktur zu erfassen. Durch die L2-Regularisierung wird zudem Overfitting reduziert, was zur hohen Genauigkeit beiträgt.

Zusammenfassung

Die Analyse der verschiedenen FCNN-Architekturen zeigt, dass die Modellgenauigkeit durch eine Erhöhung der Parameteranzahl und den Einsatz von Regularisierungstechniken erheblich gesteigert werden kann. Während das Small / ReLU-Modell nur eine Genauigkeit von 38,42 % erreicht, erzielt das Large NN V2 + L2 (300ep)-Modell mit 1.519.074 Parametern eine Genauigkeit von 97,93

3.2.2 Architektur der FCAE-Modelle

Alle in dieser Arbeit verwendeten FCAEs folgen einer Encoder-Decoder-Architektur. Der Encoder reduziert die Dimensionen der Eingabedaten schrittweise durch Faltungsschichten, während der Decoder diese Reduktion mit transponierten Faltungen umkehrt. Batch-Normalisierung und Dropout werden integriert, um die Stabilität des Trainings zu erhöhen und Überanpassung zu vermeiden.

Der Hauptunterschied zwischen den Modellen liegt in der Anzahl der Faltungsschichten, die an die Größe der Labyrinthe angepasst wird. Ziel ist es stets, sicherzustellen, dass die rekonstruierten Ausgaben dieselben Dimensionen wie die Eingaben aufweisen.

Erfahrungen zur Architekturwahl und Parameteroptimierung

Während der Entwicklung der FCAE-Modelle wurden verschiedene Architekturvarianten getestet, um eine optimale Balance zwischen Modellkomplexität, Inferenzzeit und Genauigkeit zu erreichen. Ein zentrales Experiment bestand in der Variation der Anzahl der Faltungs- und transponierten Faltungsschichten.

Ein Problem, das während der Architekturentwicklung auftrat, waren sogenannte Checkerboard-Artefakte in den rekonstruierten Labyrinthen. Diese Artefakte entstehen insbesondere in den transponierten Faltungsschichten des Decoders und äußern sich durch regelmäßige, ungewollte Muster in der Ausgabe. Die Ursachen dieser Muster werden in Odena et al., 2016 detailliert beschrieben.

Durch systematische Experimente konnte festgestellt werden, dass die Intensität dieser Artefakte maßgeblich von der Größe des Latent-Space abhängt. Sobald die Größe der repräsentierten Merkmalskarten im Latent-Space unter 5×5 fällt, nehmen die Artefakte signifikant zu. Dies liegt vermutlich daran, dass zu wenige räumliche Informationen für die Rekonstruktion vorhanden sind, was zu instabilen Übergängen in den transponierten Faltungen führt.

Ein weiterer wichtiger Faktor ist das Verhältnis der Kernel-Größe zur Stride-Größe in den transponierten Faltungsschichten. Wie in Odena et al., 2016 erläutert, treten Artefakte verstärkt auf, wenn die Kernel-Größe nicht durch die Stride-Größe teilbar ist. In diesem Fall entstehen asymmetrische Überlappungen zwischen den rekonstruierten Pixeln, wodurch sich ungleichmäßige Muster in der Ausgabe ergeben. Um diesem Problem entgegenzuwirken, wurden in der finalen Architektur Kernel- und Stride-Werte so gewählt, dass diese Bedingung erfüllt ist, wodurch die Artefakte deutlich reduziert werden konnten.

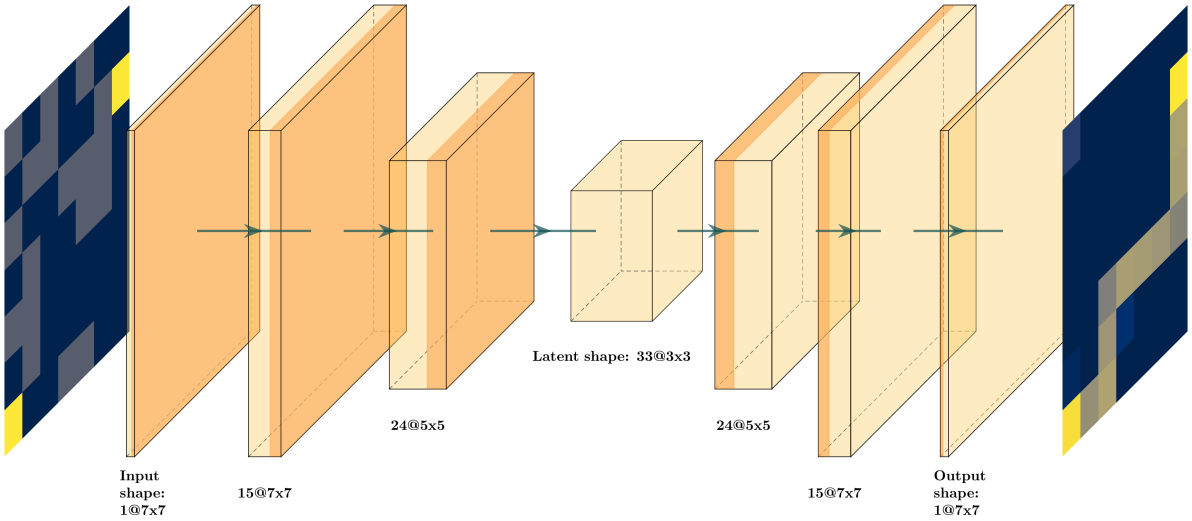
Die final gewählten Architekturparameter für die einzelnen Labyrinthgrößen ergaben sich aus diesen Experimenten und stellen einen Kompromiss zwischen Modellkomplexität und Leistung dar. Die genauen Spezifikationen der final verwendeten Modelle werden nachfolgend dargestellt.

Architektur des FCAE-Modells für 7×7 Labyrinthe

Das FCAE-Modell für 7×7-Labyrinthe dient dem direkten Vergleich mit dem FCNN. Es besteht aus drei Faltungsschichten im Encoder und drei transponierten Faltungsschichten im Decoder. Eine Normalisierung der Eingaben wurde nicht durchgeführt, da diese auch beim Vergleichsmodell nicht angewendet wurde. Eine genauere Erklärung folgt in Absatz 3.3.1.

Bei der Entwicklung dieses Modells lag der Fokus auf einer möglichst geringen Anzahl an Parametern, um ein kompaktes und effizient trainierbares Modell zu erhalten, das dennoch mit dem FCNN konkurrieren kann. Diese Designentscheidung reduzierte die Modellkomplexität auf 21.340 trainierbare Parameter. Eine schematische Darstellung ist in Abbildung 9 zu sehen.

Abbildung 9: Architektur des FCAE-Modells für 7×7 Labyrinthe



Eigene Darstellung

Architektur des FCAE-Modells für 10×10 Labyrinthe

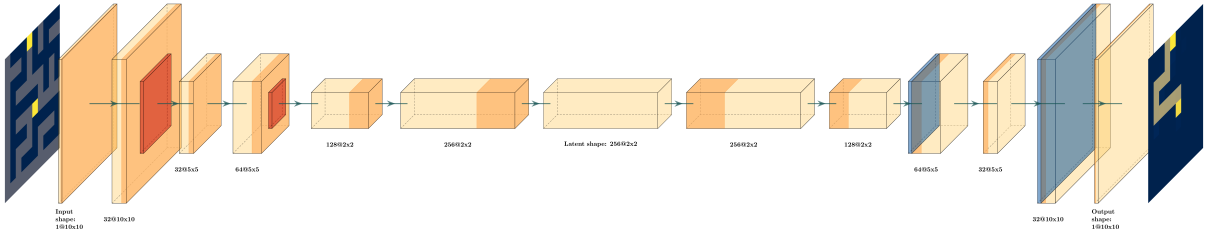
Um die steigende Komplexität der Eingaben zu bewältigen, wurde das Modell für 10×10-Labyrinthe um zusätzliche Schichten erweitert. Der Encoder besteht aus sechs Faltungsschichten, der Decoder aus sechs transponierten Faltungsschichten. Durch die Verwendung von Output-Paddings wird sichergestellt, dass die rekonstruierten Labyrinthe exakt dieselbe Größe wie die Eingaben haben.

Das Modell verfügt über 1.884.481 trainierbare Parameter. Die Architektur ist in Abbildung 10 visualisiert.

Architektur des FCAE-Modells für 20×20 Labyrinthe

Die Architektur für 20×20-Labyrinthe baut auf dem 10×10-Modell auf und erweitert es um eine zusätzliche Faltungs- und transponierte Faltungsschicht im Encoder bzw. Decoder. Dies ermöglicht eine präzisere Verarbeitung komplexer Strukturen.

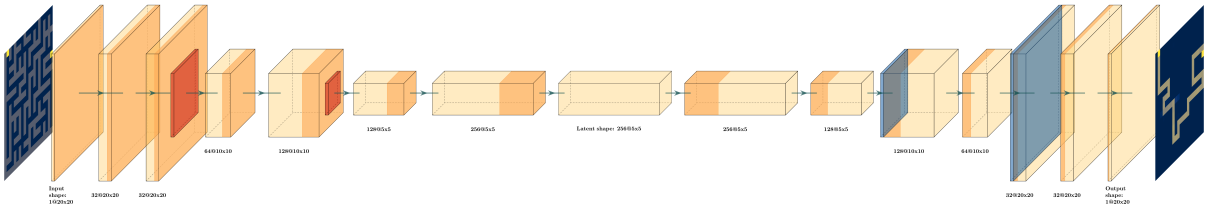
Abbildung 10: Architektur des FCAE-Modells für 10×10 Labyrinthe



Eigene Darstellung

Das Modell umfasst 2.088.001 trainierbare Parameter. Eine Visualisierung der Architektur ist in Abbildung 11 zu sehen.

Abbildung 11: Architektur des FCAE-Modells für 20×20 Labyrinthe



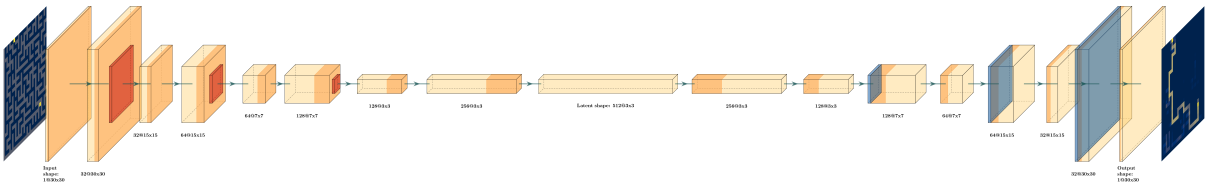
Eigene Darstellung

Architektur der FCAE-Modelle für 30×30, 40×40 und 50×50 Labyrinthe

Für größere Labyrinthe (30×30, 40×40 und 50×50) wurde eine einheitliche Architektur entwickelt. Der Encoder umfasst acht Faltungsschichten, während der Decoder ebenfalls aus acht transponierten Faltungsschichten besteht. Dropout wird in den ersten beiden Schichten und Batch-Normalisierung in allen Schicht verwendet, um Stabilität und Generalisierung zu gewährleisten.

Die Modelle verfügen über 3.311.809 trainierbare Parameter. Abbildung 12 zeigt die schematische Darstellung der Architektur, die für alle drei Größen identisch ist.

Abbildung 12: Architektur der FCAE-Modelle für 30×30, 40×40 und 50×50 Labyrinthe



Eigene Darstellung

3.3 Trainingsprozesse und Optimierung

In diesem Kapitel werden die grundlegenden Prozesse und Methoden beschrieben, die für das Training und die Optimierung der verwendeten Modelle eingesetzt wurden. Der Fokus liegt dabei auf den spezifischen Herausforderungen und Lösungen, die sich aus der Modellarchitektur sowie den Eigenschaften der Trainingsdaten ergeben.

3.3.1 Trainingsdaten und Vorverarbeitung

Die Trainingsdaten wurden bereits in Abschnitt 3.1.1 ausführlich beschrieben. Die wichtigsten Aspekte der Datensätze werden hier kurz zusammengefasst, um deren Relevanz für die Trainingsprozesse zu verdeutlichen.

Datensatzübersicht

Für das Training der Modelle wurden Labyrinthe unterschiedlicher Größen verwendet. Die Labyrinthe der Größe 7×7 basierten auf einem extern generierten Datensatz, der bereits im Vergleichsmodell FCNN verwendet wurde. Für größere Labyrinthgrößen (10×10, 20×20, 30×30, 40×40 und 50×50) wurden die Datensätze selbst erstellt, wie in Abschnitt 3.1.2 erläutert. Jeder Datensatz enthält sowohl das ungelöste Labyrinth (Eingabe) als auch die optimale Lösung (Ausgabe).

Vorverarbeitung der Labyrinthe

Die Vorverarbeitung der Daten war abhängig von der Modellarchitektur und der Größe der Labyrinthe. Für das 7×7 FCNN-Modell war eine Umwandlung der zweidimensionalen Darstellung in ein eindimensionales Format von 1×49 notwendig, um der Struktur des FCNN zu entsprechen. Für das 7×7 FCAE-Modell wurde die Eingabe hingegen in ihrer ursprünglichen Form belassen.

Bei den Modellen für größere Labyrinthgrößen (10×10 bis 50×50) wurde die Eingabe normalisiert. Hierbei wurden die Werte in den Eingabematrizen so skaliert, dass sie im Bereich von $[0, 1]$ lagen. Die Ausgaben, welche die gelösten Labyrinthe repräsentieren, wurden nicht normalisiert, da deren Werte semantische Bedeutung haben (z. B. 2 für den optimalen Pfad, 3 für die Start- und Endpunkte). Ein Padding der Eingabedaten war nicht erforderlich, da jedes FCAE-Modell auf eine spezifische Größe der Labyrinthe trainiert wurde.

3.3.2 Trainingsprozess

Der Trainingsprozess der FCAE-Modelle wurde darauf ausgelegt, eine stabile und effiziente Konvergenz zu gewährleisten. Im Folgenden werden die wichtigsten Schritte und Konfigurationen des Trainingsprozesses beschrieben. Diese Werte wurden basierend auf ihrer weit verbreiteten Anwendung und ihren stabilen Ergebnissen in ähnlichen Szenarien gewählt (Zhu et al., 2019)

Optimierungsalgorithmus

Für das Training der FCAE-Modelle wurde der Adam-Optimierer verwendet, ein in der Literatur häufig eingesetztes Optimierungsverfahren für neuronale Netzwerke (Kingma & Ba, 2014). Der Optimierer kombiniert die Vorteile von Adaptive Gradient Descent und Momentum und ist besonders robust gegenüber kleinen Änderungen der Lernrate. Für alle Modelle wurde eine feste Lernrate von $\alpha = 10^{-4}$ und ein Gewichtungszerfall (L2) von 10^{-2} genutzt.

Verlustfunktion

Als Verlustfunktion wurde der Mean Squared Error (MSE) Loss eingesetzt. Diese Funktion bewertet den Durchschnitt der quadrierten Abweichungen zwischen den vorhergesagten und den tatsächlichen Werten. Der MSE eignet sich besonders gut für die Rekonstruktion von Labyrinthlösungen, da Abweichungen in der Ausgabe gegenüber der Referenz bestraft werden. Dies fördert präzisere Vorhersagen der optimalen Pfade.

Trainingskonfiguration

Die Trainingsprozesse wurden mit einer Batch-Größe von 64 durchgeführt. Die Anzahl der Trainingsdurchläufe (Epochen) betrug je nach Modell zwischen 50 und 200.

Regularisierungsmechanismen

Zur Vermeidung von Overfitting wurden in den FCAE-Modellen nach der ersten und zweiten Faltungsschicht eine Dropout-Schicht mit einer Wahrscheinlichkeit von 10 % bis 20 % eingefügt. Diese reduzieren die Wahrscheinlichkeit, dass das Modell übermäßig auf spezifische Trainingsdaten angepasst wird, und verbessern die Generalisierungsfähigkeit.

Verwendete Hardware

Das Training wurde auf einer Nvidia RTX 2070 GPU mit 8 GB RAM durchgeführt, unterstützt von einem AMD Ryzen 5 2600X Prozessor und 16 GB Hauptspeicher.

3.3.3 Validierung und Evaluierung während des Trainings

Um die Modellleistung nach und während des Trainings zu überwachen und mögliche Überanpassungen an die Trainingsdaten zu vermeiden, wurde ein umfassender Validierungsprozess implementiert. Die wichtigsten Schritte und Methoden sind im Folgenden beschrieben.

Validierung der Labyrinthlösungen

Da die vom Modell vorhergesagten Werte nicht exakt (d.h. genau 2 für den Pfad und 3 für die Start- und Endpunkte) sind, musste eine Methode entwickelt werden, um aus der kontinuierlichen Ausgabe eine diskrete Pfadstruktur zu rekonstruieren. Die Implementierung in Anhang 6.2 stammt, wie das FCNN-Modell, von Vlad, 2023, jedoch wurde eine Begrenzung der maximalen Pfadlänge entfernt. Die Validierung der vorhergesagten Lösungen erfolgt dabei in mehreren Schritten:

1. **Identifikation der Start- und Endpunkte:** Die Start- und Endpunkte des Labyrinths werden durch einen Schwellwert identifiziert. Dabei wird überprüft, welche Werte im vorhergesagten Labyrinth innerhalb eines bestimmten Toleranzbereichs um den erwarteten Start- bzw. Endpunktwert liegen. Liegt kein Punkt innerhalb dieses Bereichs, gilt das Labyrinth als ungültig.
2. **Rekonstruktion des Pfades:** Beginnend am Startpunkt wird iterativ der nächste Schritt im Pfad gewählt. Hierzu wird das Nachbarfeld mit dem höchsten vorhergesagten Wert ausgewählt, das zudem nicht bereits besucht wurde. Dieser Prozess wird solange fortgeführt, bis entweder das Endziel erreicht wird oder keine weiteren Schritte möglich sind.
3. **Validierung der Lösung:** Der rekonstruierte Pfad wird abschließend mit der bekannten optimalen Lösung verglichen. Hierbei wird überprüft, ob der extrahierte Pfad tatsächlich einen durchgängigen Weg zwischen Start- und Zielpunkt bildet, der sich an den vorhergesagten Wahrscheinlichkeiten orientiert.

Der Algorithmus zur Pfadrekonstruktion nutzt dabei eine Helligkeits-basierte Strategie, bei der für jeden Schritt das benachbarte Feld mit dem höchsten Vorhersagewert ausgewählt wird. Dies stellt sicher, dass der Netzwerk-Output sinnvoll interpretiert wird und kleinere numerische Ungenauigkeiten in den Vorhersagen nicht zu falschen Pfaden führen.

Durch diese Methode wird es möglich, auch kontinuierliche Modellvorhersagen robust zu evaluieren, ohne strikte binäre Werte zu verlangen. Das Verfahren stellt sicher, dass das neuronale Netzwerk mit klassischen Algorithmen wie A* verglichen werden kann, ohne dass numerische Rundungsfehler zu falschen Bewertungen der Modellleistung führen.

Überwachungsmechanismen während des Trainings

Während des Trainings wurde der Verlustverlauf regelmäßig überwacht, um sicherzustellen, dass das Modell stabil konvergiert. Um Überanpassungen (Overfitting) zu vermeiden, wurde der Trainingsprozess gestoppt, wenn der Validierungsverlust über mehrere Epochen hinweg stagnierte oder anstieg. Obwohl kein formales Early Stopping implementiert wurde, diente diese Überwachung als qualitativer Indikator für eine potenzielle Überanpassung.

Visualisierung des Trainingsverlaufs

Zur Analyse des Trainingsprozesses wurden die Verlaufsdaten des Verlusts über alle Epochen hinweg aufgezeichnet und grafisch dargestellt. Abbildung 13 zeigt die Trainings- und Validierungsverluste für alle FCAE-Modelle (7x7 bis 50x50). Die Verlustkurven verdeutlichen, dass die Modelle stabil konvergieren und keine Anzeichen von Überanpassung zeigen.

Abbildung 13 zeigt, dass bei allen Modellen der Verlust konvergiert, wobei der Validierungsverlust mit dem Trainingsverlust übereinstimmt. Dies deutet auf eine erfolgreiche und stabile Trainingsdynamik hin.

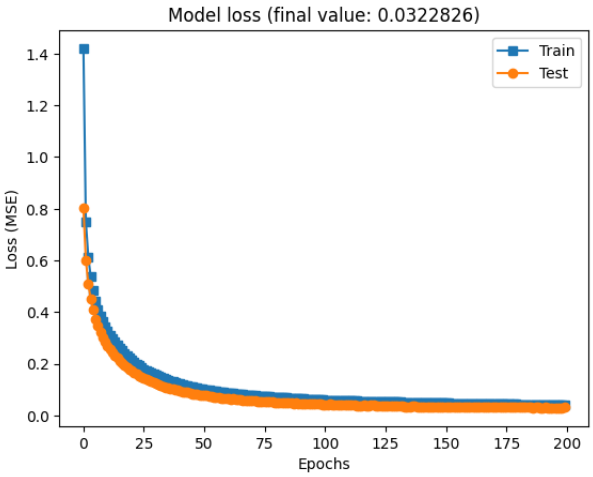
Skalierungsherausforderungen bei größeren Modellen

Mit zunehmender Labyrinthgröße wuchsen auch die Speicheranforderungen der Datensätze erheblich. Während die Datensätze für die 10×10 Labyrinthe mit jeweils 100.000 ungelösten und gelösten Labyrinthen eine Größe von je 76 MB erreichten, stieg die Größe der Datensätze für die 50×50 Labyrinthe auf 1,86 GB pro Datei. Die Analyse der Dateigrößen zeigt, dass die Zunahme der Speicheranforderungen näherungsweise quadratisch zur Labyrinthgröße ist.

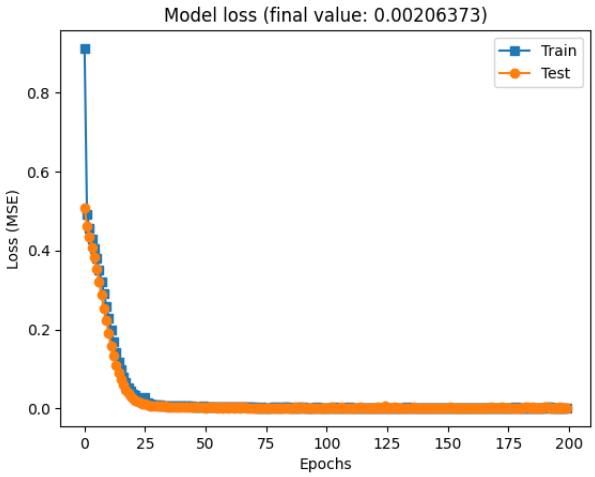
Die begrenzten Ressourcen einer Nvidia RTX 2070 GPU mit 8 GB Speicher führten schließlich dazu, dass beim Training des 50x50 FCAE-Modells der Speicher vollständig ausgelastet war.

Abbildung 13: Trainings- und Validierungsverlust der FCAE-Modelle

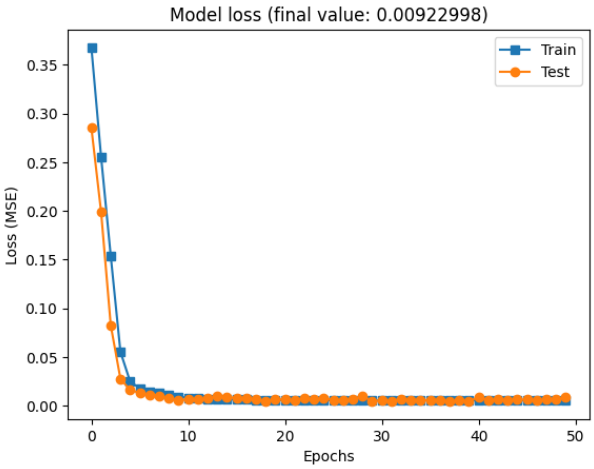
(a) 7x7 FCAE-Modell



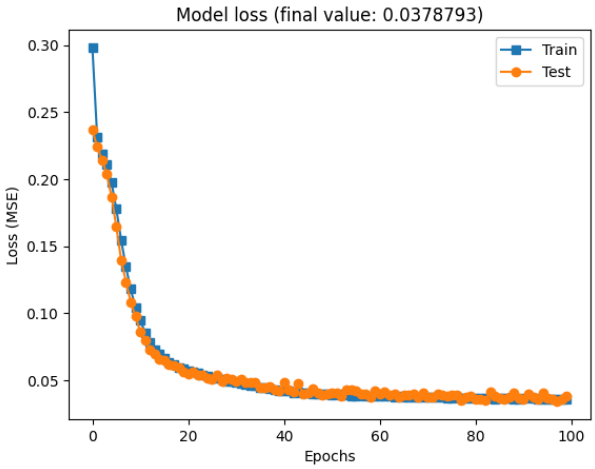
(b) 10x10 FCAE-Modell



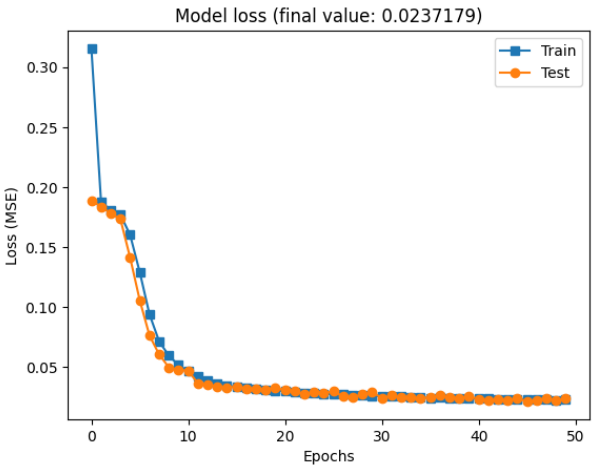
(c) 20x20 FCAE-Modell



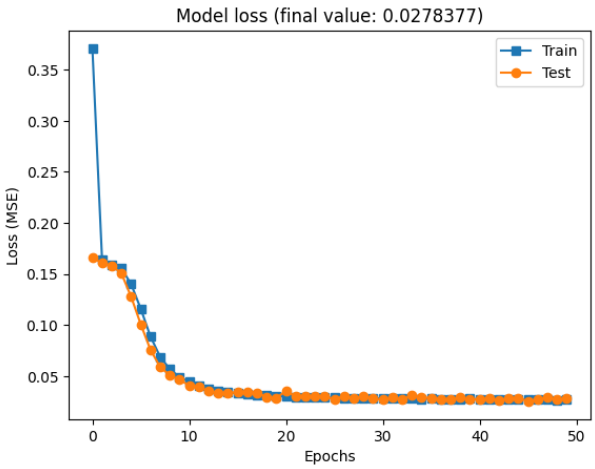
(d) 30x30 FCAE-Modell



(e) 40x40 FCAE-Modell



(f) 50x50 FCAE-Modell



Eigene Darstellung

Aus diesem Grund wurde die Entwicklung und Evaluierung der Modelle nach dieser Labyrinthgröße eingestellt, da die Hardwareanforderungen für größere Modelle nicht mehr erfüllt werden konnten.

Zusätzliche Überlegungen

Die Konsistenz zwischen Trainings- und Validierungsverlust über alle Modelle hinweg deutet auf einen robusten Trainingsprozess hin. Die Speicheranforderungen und die Trainingszeit stiegen jedoch mit zunehmender Labyrinthgröße erheblich an. Dies verdeutlicht die Grenzen des aktuellen Hardware-Setups und zeigt potenzielle Verbesserungsansätze durch den Einsatz leistungsfähigerer GPUs oder effizienterer Modellarchitekturen.

3.4 Implementierung des A*-Algorithmus

Die Implementierung des A*-Algorithmus basiert auf den in Unterabschnitt 2.2.2 beschriebenen theoretischen Grundlagen und der Umsetzung von Vlad, 2023. Ziel der Implementierung ist es, den optimalen Pfad zwischen den Start- und Endpunkten in den gegebenen Labyrinthen zu finden, wobei Hindernisse (Wände) umgangen werden müssen.

Anpassungen an die Labyrinthstruktur

In den Datensätzen dieser Arbeit werden die Start- und Endpunkte durch den Wert 3, freie Wege durch den Wert 0 und Hindernisse durch den Wert 1 repräsentiert. Die Implementierung wurde so angepasst, dass die Start- und Endpunkte im Labyrinth korrekt erkannt und in den Suchprozess integriert werden. Hindernisse werden dabei vollständig berücksichtigt, sodass der Algorithmus ausschließlich auf den freien Wegen operiert.

Effizienz und Spezifika der Implementierung

Die Implementierung verwendet eine Prioritätswarteschlange, um die effizienteste Reihenfolge für die Knotenbesuche zu bestimmen. Die heuristische Funktion basiert auf der Manhattan-Distanz, da sie für rasterbasierte Strukturen wie Labyrinthe besonders geeignet ist. Der Algorithmus wurde so gestaltet, dass er sowohl kleine Labyrinthe (z. B. 7x7) als auch größere Strukturen (z. B. 50x50) zuverlässig lösen kann.

Quellcode und Referenz

Der vollständige Quellcode des A*-Algorithmus ist im Anhang unter Anhang 6.2 dokumentiert. Die Implementierung umfasst die Suche nach dem kürzesten Pfad, die Entfernung der Hindernisse aus dem Labyrinth und die Markierung des gefundenen Pfads im Labyrinth (mit dem Wert 2).

Die Implementierung des A*-Algorithmus diente als Benchmark für den Vergleich mit den entwickelten neuronalen Netzwerken und liefert somit eine wichtige Referenz zur Bewertung der Leistungsfähigkeit der Modelle. Eine detaillierte Analyse der Ergebnisse wird in Kapitel 4 vorgestellt.

4 Ergebnisse

In diesem Kapitel werden die experimentellen Ergebnisse der entwickelten FCAEs vorgestellt. Dabei erfolgt eine Gegenüberstellung mit einem FCNN sowie dem klassischen A*-Algorithmus.

4.1 Vergleich von FCNN und FCAE

In diesem Unterkapitel wird ein umfassender Vergleich zwischen dem FCNN und dem FCAE durchgeführt. Der Fokus liegt auf den Metriken Pfadgenauigkeit, Inferenzzeit, Speichernutzung, Parameteranzahl und Modellgröße. Die Analysen basieren auf experimentellen Messungen mit unterschiedlichen Batchgrößen, die eine detaillierte Bewertung der Skalierbarkeit und Effizienz der Modelle ermöglichen.

Pfadgenauigkeit

Die Ergebnisse zeigen, dass das FCNN in allen getesteten Batchgrößen durchweg eine leicht höhere Pfadgenauigkeit aufweist als das FCAE. Mit steigender Batchgröße nimmt jedoch die Pfadgenauigkeit bei beiden Modellen leicht ab. Diese Entwicklung ist in Abbildung 14 dargestellt.

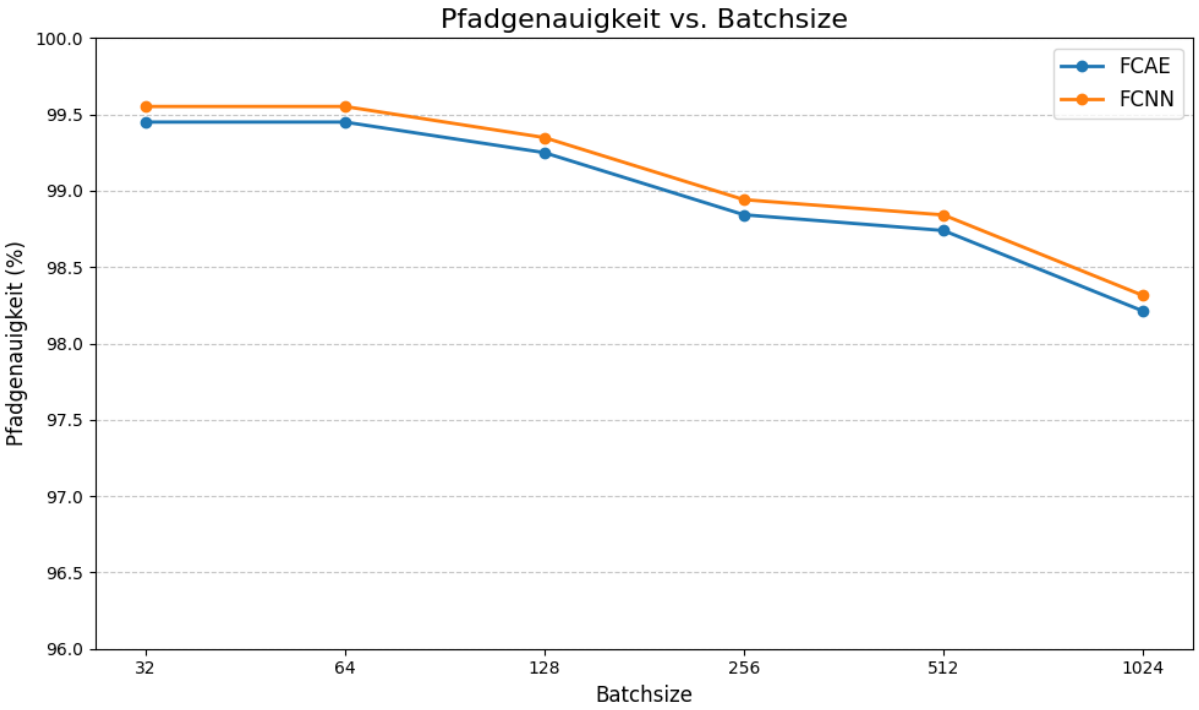
Die Pfadgenauigkeit des FCAE für 7×7 Labyrinthe fällt zudem geringer aus als die des 10×10 Modells, wie in Abschnitt 4.2 ersichtlich wird. Dies lässt sich darauf zurückführen, dass das 7×7 Modell mit einer deutlich geringeren Anzahl an Parametern konzipiert wurde, um eine möglichst kompakte Modellstruktur zu gewährleisten. Diese Reduzierung der Modellkomplexität begrenzt die Fähigkeit des Modells, feinere Strukturen und Variationen in den Labyrinthdaten zu erfassen, wodurch sich eine etwas geringere Pfadgenauigkeit ergibt.

Visueller Vergleich der Lösungen

Um die Unterschiede in der Qualität der Lösungen zwischen den Modellen zu verdeutlichen, zeigt Abbildung 15 eine visuelle Darstellung der Netzwerkausgaben.

Die Abbildungen verdeutlichen, dass das FCNN in der Regel klarere und zusammenhängendere Pfade erzeugt, während das FCAE tendenziell weichere Übergänge in der Pfadprognose

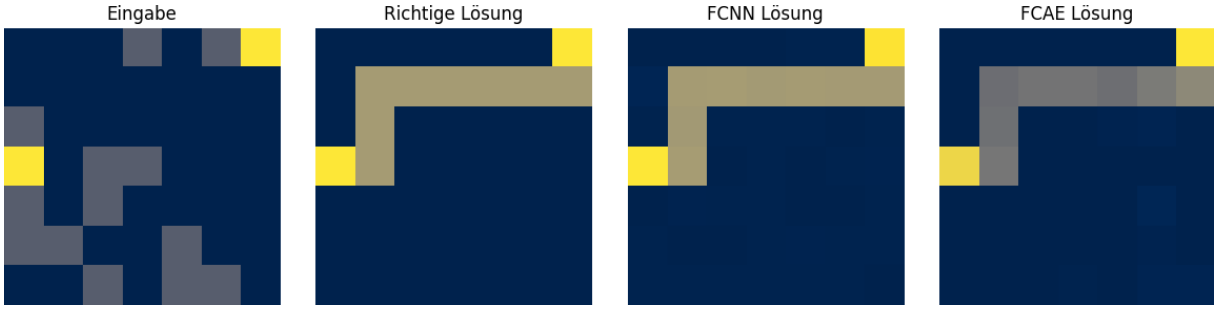
Abbildung 14: Pfadgenauigkeit bei unterschiedlichen Batchgrößen



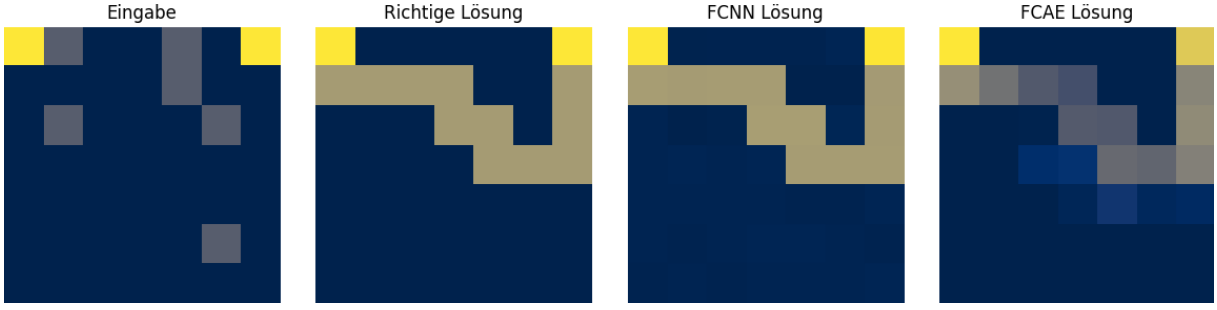
Eigene Darstellung

Abbildung 15: Visueller Vergleich der Lösungen von FCNN und FCAE

(a) Beispiel 1: Labyrinth mit einfachem Pfad



(b) Beispiel 2: Labyrinth mit komplexerer Pfadstruktur



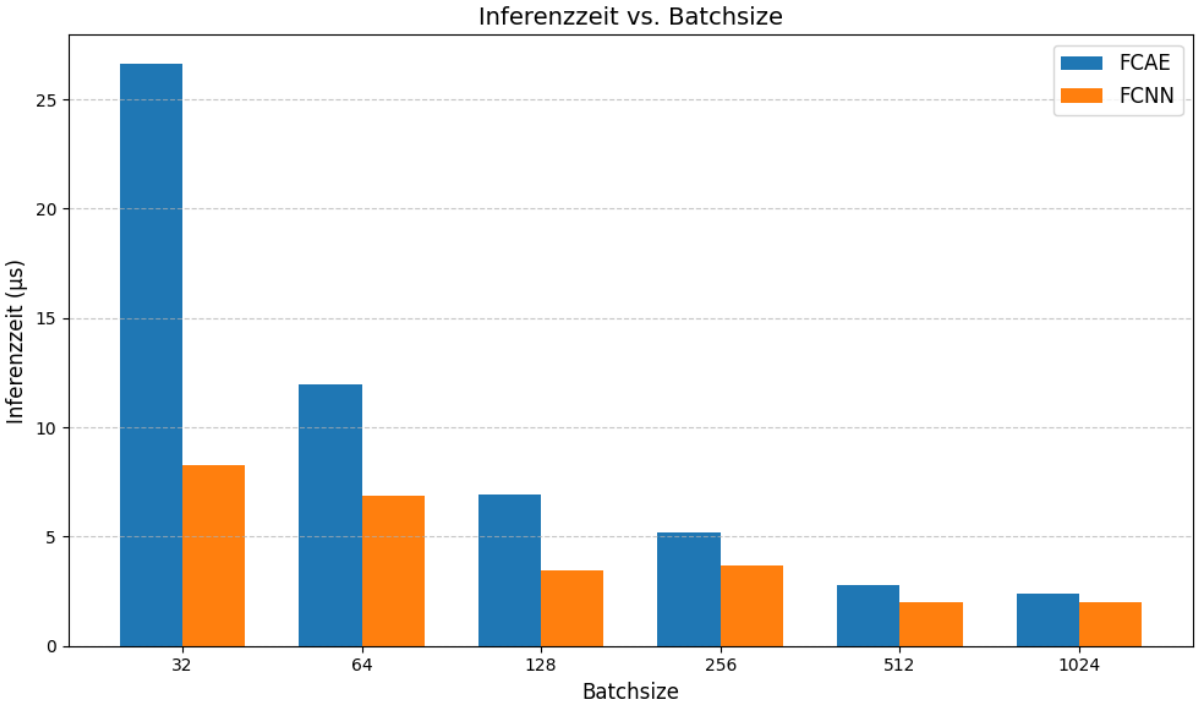
Eigene Darstellung

zeigt. Dies kann darauf zurückgeführt werden, dass das FCAE die Labyrinthe als Bilddaten verarbeitet und damit möglicherweise weniger deterministische Strukturen lernt als das FCNN.

Inferenzzeit

Abbildung 16 zeigt die Entwicklung der Inferenzzeit in Abhängigkeit von der Batchgröße. Das FCNN überzeugt hier durch eine deutlich geringere Inferenzzeit, insbesondere bei kleineren Batchgrößen. Das FCAE benötigt aufgrund seiner komplexeren Architektur mehr Rechenzeit, bleibt jedoch in einem vertretbaren Rahmen.

Abbildung 16: Inferenzzeit bei unterschiedlichen Batchgrößen

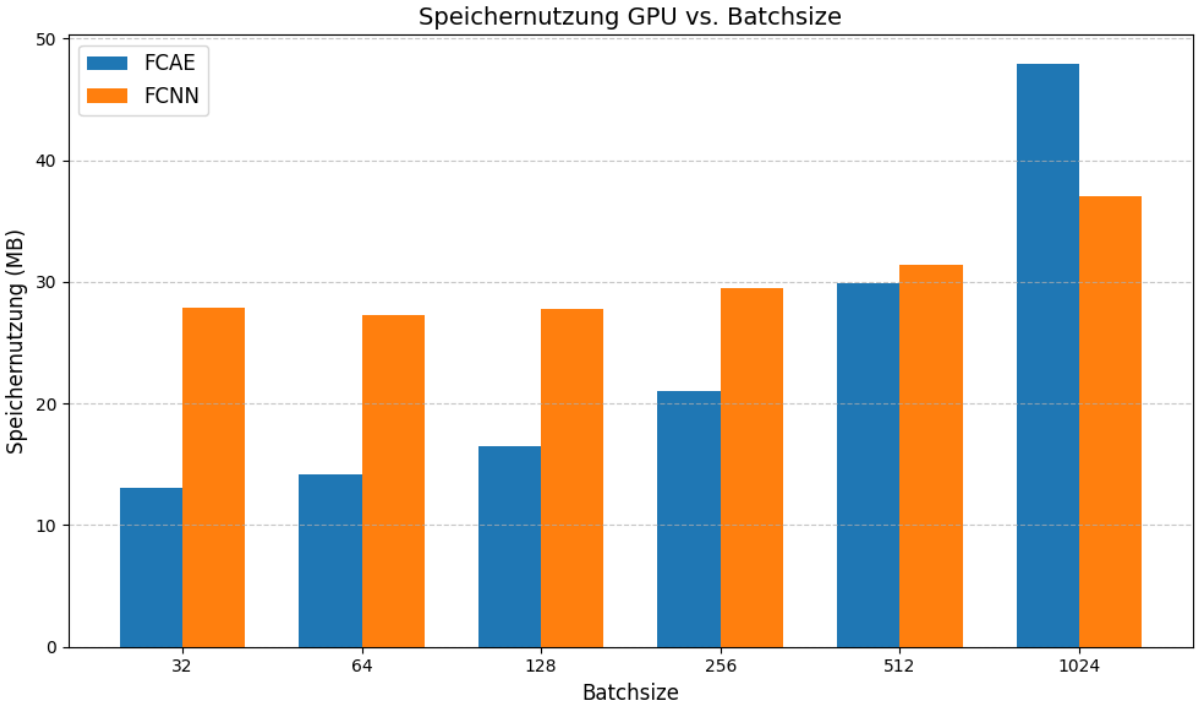


Eigene Darstellung

Speichernutzung auf der GPU

Die Speichernutzung auf der GPU ist ein wichtiger Faktor für die Skalierbarkeit der Modelle. In Abbildung 17 wird deutlich, dass das FCAE bei kleineren Batchgrößen weniger Speicher benötigt als das FCNN. Mit zunehmender Batchgröße steigt der Speicherverbrauch des FCAE jedoch überproportional, was seine Effizienz bei sehr großen Datenmengen einschränkt. Das FCNN zeigt hingegen eine gleichmäßigere Entwicklung des Speicherverbrauchs.

Abbildung 17: Speichernutzung auf der GPU bei unterschiedlichen Batchgrößen



Eigene Darstellung

Parameteranzahl und Modellgröße

Das FCAE weist eine deutlich geringere Anzahl an Parametern (21,340) im Vergleich zum FCNN (1,519,074) auf. Dies schlägt sich auch in der Modellgröße nieder: Das FCAE hat eine kompakte Größe von 81 KB, während das FCNN 5,79 MB umfasst.

Zusammenfassung der Ergebnisse

Der Vergleich zwischen FCNN und FCAE zeigt deutliche Unterschiede in den Leistungsmerkmalen. Das FCNN überzeugt durch schnelle Inferenzzeiten und eine gleichmäßige Speicherverbrauchsskala, während das FCAE durch eine kompakte Modellgröße und geringere Speicheranforderungen bei kleinen Batchgrößen punktet. Die visuelle Analyse der vorhergesagten Pfade zeigt, dass das FCNN klarere und konsistentere Ergebnisse liefert, während das FCAE eher weichere Vorhersagen mit leichteren Abweichungen trifft. Die Wahl des Modells sollte auf Basis der spezifischen Anforderungen erfolgen: Für Szenarien mit hoher Geschwindigkeit und Präzision ist das FCNN die bessere Wahl, während das FCAE in speicherbeschränkten Umgebungen Vorteile bietet.

4.2 Vergleich von FCAE und A*

In diesem Kapitel werden die Ergebnisse des Vergleichs zwischen den Fully Convolutional Autoencoder (FCAE)-Modellen und dem A*-Algorithmus vorgestellt. Die Testreihen wurden für verschiedene Labyrinthgrößen von 10×10 bis 50×50 durchgeführt. Die betrachteten Metriken umfassen die Pfadgenauigkeit, die Inferenzzeit und die Speichernutzung.

Ein zentraler Aspekt des Vergleichs zwischen dem FCAE-Modell und dem A*-Algorithmus ist die Qualität der generierten Pfade. In Abbildung 18 sind beispielhafte Labyrinthlösungen dargestellt. Die Abbildungen zeigen jeweils die Eingabe (links), die von A* generierte optimale Lösung (Mitte) und die durch das FCAE-Modell erzeugte Pfadvorhersage (rechts).

Pfadgenauigkeit

Die Pfadgenauigkeit der getesteten Modelle ist in Tabelle 2 dargestellt. Während der A*-Algorithmus für alle Labyrinthgrößen eine konstante Genauigkeit von 100% aufweist, zeigen die FCAE-Modelle eine abnehmende Genauigkeit mit zunehmender Labyrinthgröße.

Für kleine Labyrinthe von 10×10 liegt die Pfadgenauigkeit der FCAE-Modelle bei 99,99%. Mit zunehmender Labyrinthgröße verringert sich dieser Wert auf 99,14% für 20×20-Labyrinthe und 89,88% für 30×30-Labyrinthe. Für größere Labyrinthe beträgt die Genauigkeit 83,98% bei 40×40 und reduziert sich weiter auf 75,50% für 50×50 Labyrinthe.

Tabelle 2: Pfadgenauigkeit (%) von FCAE und A* für verschiedene Labyrinthgrößen

Labyrinthgröße	FCAE	A*
10x10	99,99	100,00
20x20	99,14	100,00
30x30	89,88	100,00
40x40	83,98	100,00
50x50	75,50	100,00

Eigene Darstellung

Inferenzzeit

Die durchschnittlichen Inferenzzeiten der FCAE-Modelle und des A*-Algorithmus sind in Abbildung 19 veranschaulicht. Die Inferenzzeit des A*-Algorithmus nimmt mit der Labyrinth-

größe zu, von 465,96 µs bei 10×10 Labyrinthen auf 14.877,36 µs für 50×50 Labyrinthe.

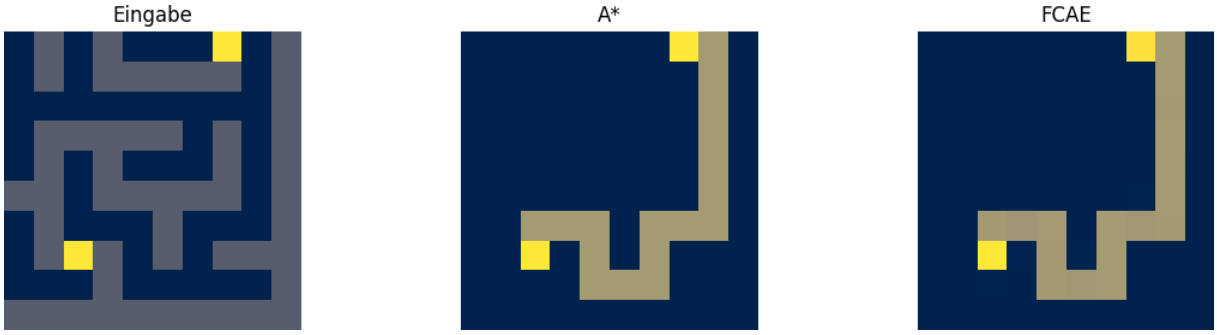
Die beobachtete Entwicklung der Laufzeit des A*-Algorithmus entspricht den theoretischen Erwartungen. Da A* den Graphen schrittweise durchsucht und dabei alle erreichbaren Knoten speichert, wächst die Laufzeit in etwa quadratisch mit der Anzahl der Zellen im Labyrinth. Dies folgt aus der Tatsache, dass A* im schlimmsten Fall jeden Knoten im Suchraum besuchen muss, was einer Zeitkomplexität von $O(n^2)$ für ein $n \times n$-Labyrinth entspricht (Hart et al., 1968; Russell & Norvig, 2016). Die gemessenen Werte bestätigen diesen theoretischen Verlauf, da die Laufzeit von etwa 466 µs bei 10×10-Labyrinthen auf etwa 14.877 µs bei 50×50-Labyrinthen ansteigt, was ungefähr einem quadratischen Wachstum entspricht.

Im Gegensatz dazu zeigt sich für die FCAE-Modelle eine deutlich geringere Inferenzzeit. Für 10×10-Labyrinthe beträgt die mittlere Inferenzzeit auf der GPU im Batch-Modus 70,39 µs und im Single-Load-Modus 43,97 µs. Bei 50×50-Labyrinthen steigen diese Werte auf 121,60 µs bzw. 119,17 µs, bleiben jedoch weiterhin signifikant unter der des A*-Algorithmus. Die FCAE-Modelle profitieren hierbei von der parallelen Berechnung auf der GPU, wodurch die Zeitkomplexität nahezu konstant bleibt und große Labyrinthe effizient verarbeitet werden können.

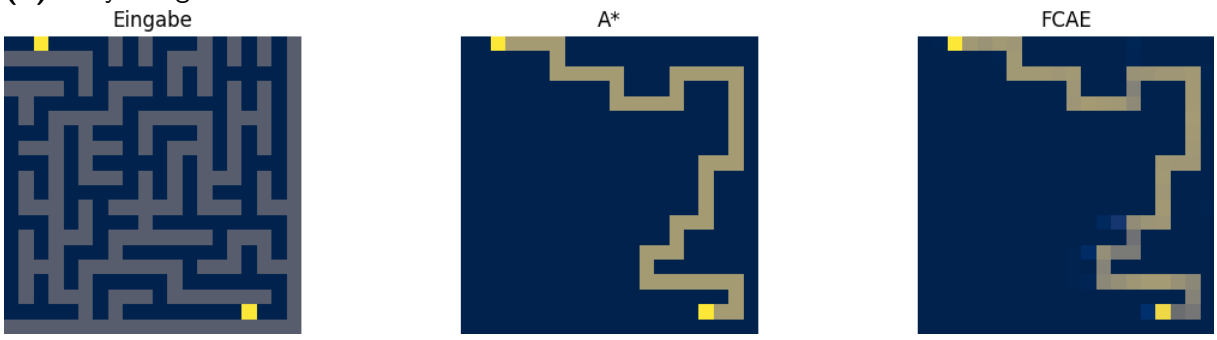
Diese Unterschiede verdeutlichen die Skalierungsvorteile der neuronalen Netzwerke gegenüber klassischen Suchalgorithmen wie A*, insbesondere bei großen Problemgrößen. Während A* mit wachsender Labyrinthgröße eine exponentiell steigende Laufzeit aufweist, bleibt die Inferenzzeit des FCAE-Modells annähernd konstant, wodurch es sich für Szenarien eignet, in denen eine schnelle Pfadfindung erforderlich ist.

Abbildung 18: Visueller Vergleich zwischen A* und FCAE für verschiedene Labyrinthgrößen

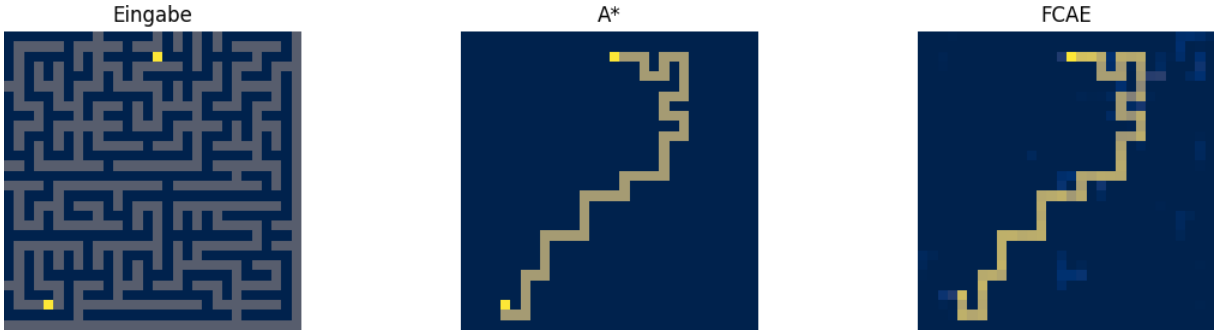
(a) Labyrinthgröße 10x10



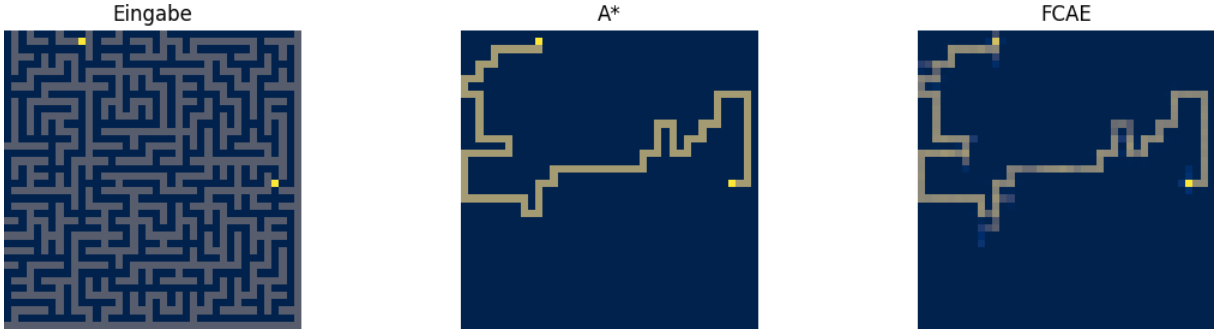
(b) Labyrinthgröße 20x20



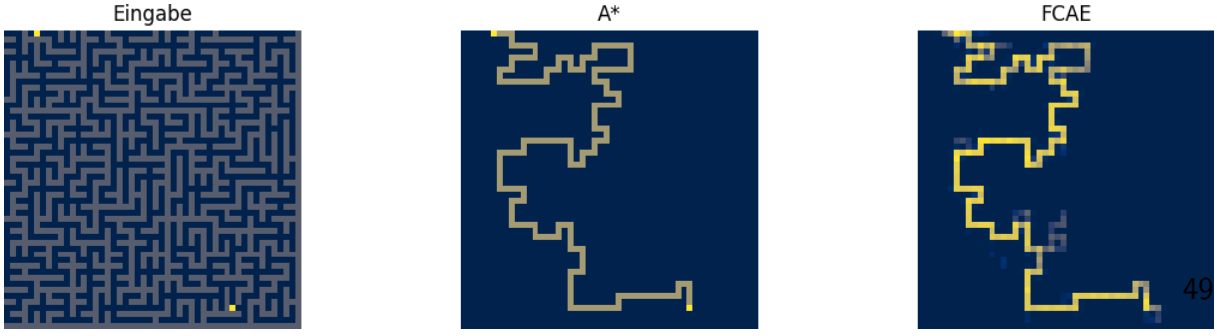
(c) Labyrinthgröße 30x30



(d) Labyrinthgröße 40x40



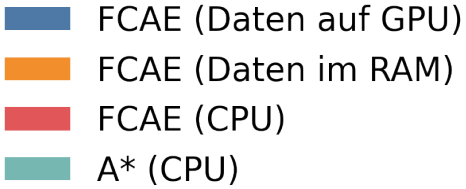
(e) Labyrinthgröße 50x50



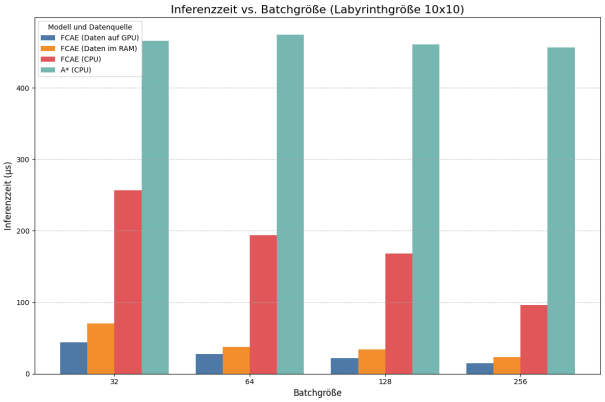
Eigene Darstellung

Abbildung 19: Inferenzzeit von FCAE und A* für verschiedene Labyrinthgrößen

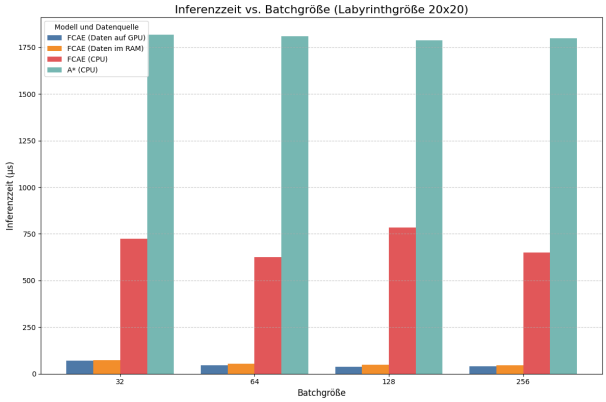
(a) Legende



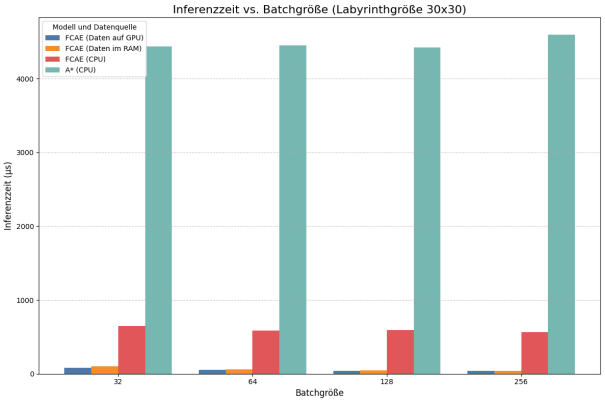
(b) 10x10 Labyrinth



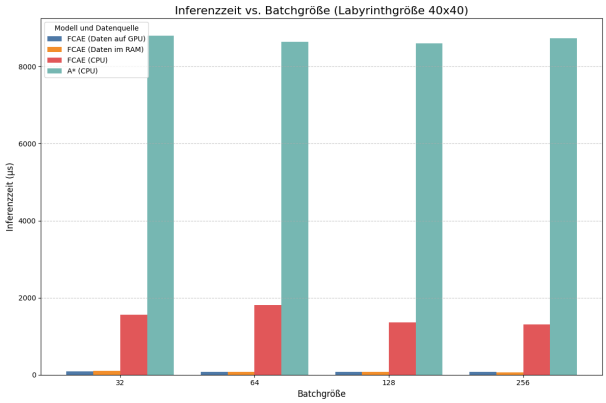
(c) 20x20 Labyrinth



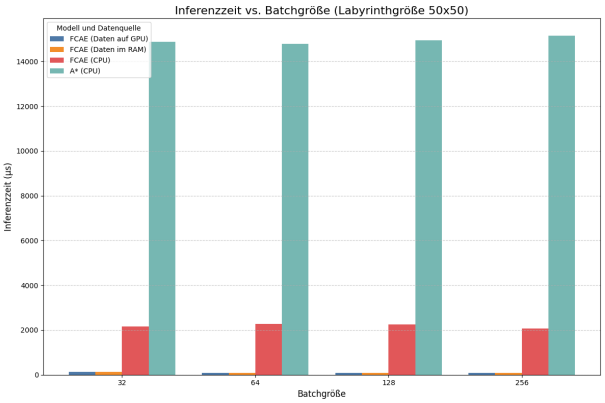
(d) 30x30 Labyrinth



(e) 40x40 Labyrinth



(f) 50x50 Labyrinth



Eigene Darstellung

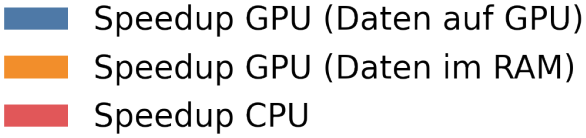
Speedup von FCAE gegenüber A*

Der Speedup der FCAE-Modelle gegenüber A* ist in Abbildung 20 dargestellt. Die Berechnung des Speedups basiert auf dem Verhältnis der Inferenzzeiten von A* zur FCAE-Inferenzzeit.

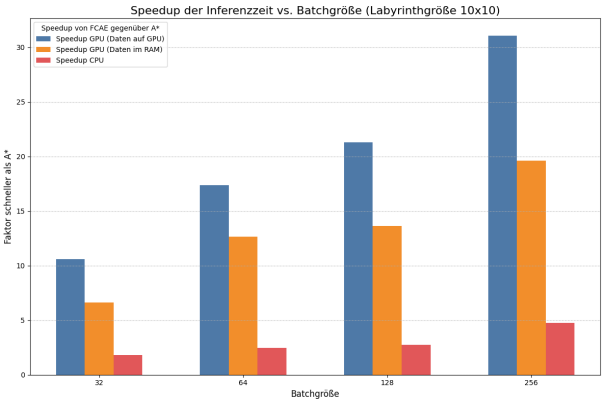
Für 10×10-Labyrinthe erreicht FCAE einen Speedup-Faktor von 31,07 im GPU-Modus bei einer Batchgröße von 256. Mit zunehmender Labyrinthgröße erhöht sich dieser Wert weiter. Für 50×50-Labyrinthe beträgt der höchste Speedup 176,24.

Abbildung 20: Speedup der FCAE-Inferenzzeit gegenüber A*

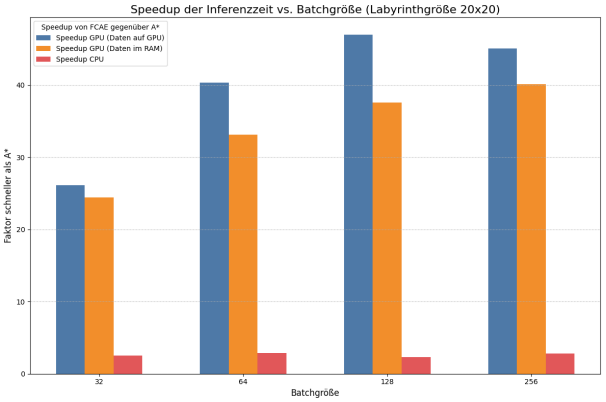
(a) Legende



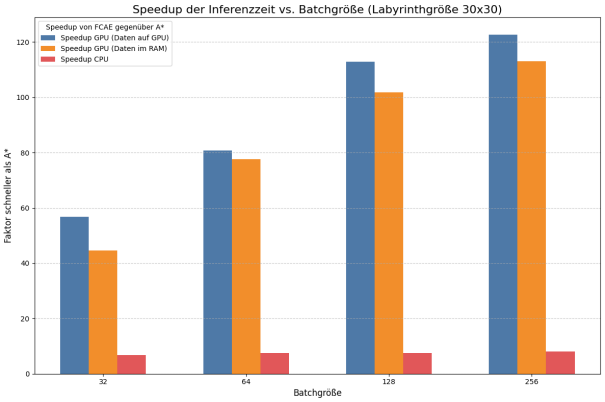
(b) 10x10 Labyrinth



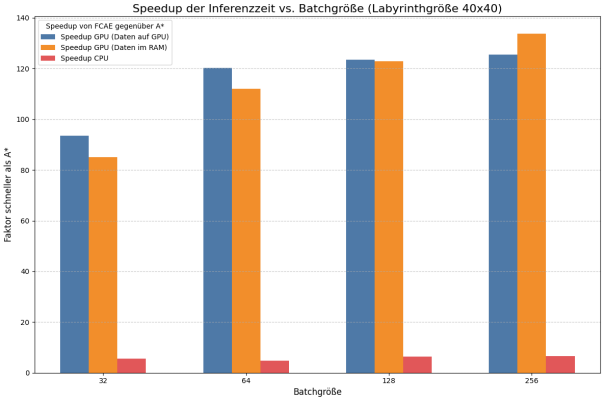
(c) 20x20 Labyrinth



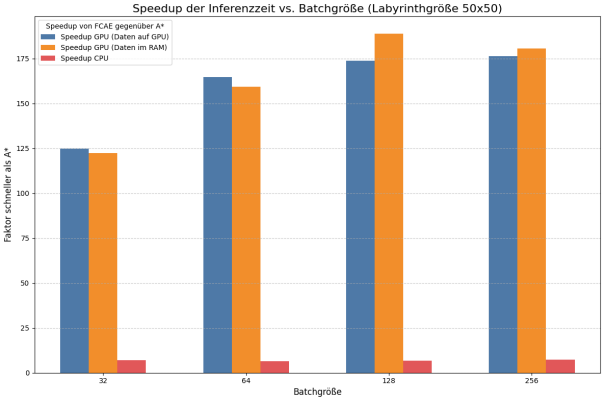
(d) 30x30 Labyrinth



(e) 40x40 Labyrinth



(f) 50x50 Labyrinth



Eigene Darstellung

Speichernutzung

Die Speichernutzung der Modelle ist in Tabelle 3 dokumentiert. Während die FCAE-Modelle eine steigende Speichernutzung mit zunehmender Labyrinthgröße aufweisen, bleibt die Spei-

chernutzung von A* konstant unter 1 MB. Die FCAE-Speichernutzung auf der GPU steigt von 14,21 MB für 10×10-Labyrinthe auf 239,70 MB für 50×50-Labyrinthe.

Tabelle 3: Speichernutzung (in MB) für FCAE und A*

Labyrinthgröße	FCAE (GPU)	A* (CPU)
10x10	14,21	0,11
20x20	39,22	0,17
30x30	71,29	0,25
40x40	125,45	0,42
50x50	239,70	0,49

Eigene Darstellung

5 Diskussion

In diesem Kapitel werden die Ergebnisse aus Kapitel 4 im Detail interpretiert und deren Bedeutung im Kontext der Zielsetzungen dieser Arbeit analysiert. Dabei liegt der Fokus darauf, die Leistungsfähigkeit der Fully-Convolutional Autoencoder (FCAEs) bei der Lösung von Labyrinthen zu bewerten und mit dem klassischen A*-Algorithmus zu vergleichen. Besonderes Augenmerk wird auf die Aspekte der Pfadgenauigkeit, Inferenzzeit und des Speicherverbrauchs gelegt. Darüber hinaus werden die Grenzen der verwendeten Methoden beleuchtet und mögliche Verbesserungen für zukünftige Arbeiten diskutiert.

5.1 Interpretation der Ergebnisse

5.1.1 Vergleich von FCNN und FCAE

Der Vergleich zwischen dem Fully Connected Neural Network (FCNN) und dem Fully Convolutional Autoencoder (FCAE) basiert auf Experimenten mit 7×7-Labyrinthen. Die Analyse umfasst die Metriken Pfadgenauigkeit, Inferenzzeit und Speichernutzung, um die Unterschiede in der Leistungsfähigkeit beider Modellarchitekturen zu bewerten.

Pfadgenauigkeit

Die Messungen zeigen, dass das FCNN über alle Batchgrößen hinweg eine leicht höhere Pfadgenauigkeit aufweist als das FCAE. Bei einer Batchgröße von 32 beträgt die Genauigkeit des FCNN 99,55%, während das FCAE eine Genauigkeit von 99,45% erreicht. Die Differenz der Pfadgenauigkeit des FCNN zu dem angegebenen Wert von Vlad, 2023 kann an der angepassten Implementation des Verfahrens zur Validierung des berechneten Pfads liegen, wie in Absatz 3.3.3 erklärt. Mit zunehmender Batchgröße sinkt die Genauigkeit bei beiden Modellen leicht, wobei das FCNN stets einen kleinen Vorteil behält. Bei einer Batchgröße von 1024 beträgt die Genauigkeit des FCNN 98,31%, während das FCAE auf 98,21% fällt.

Inferenzzeit

In Bezug auf die Inferenzzeit zeigt das FCNN eine signifikant schnellere Verarbeitung als das FCAE. Bei einer Batchgröße von 32 beträgt die durchschnittliche Inferenzzeit des FCNN 8,25

µs, während das FCAE mit 26,62 µs etwa dreimal langsamer ist. Dieser Trend bleibt über alle Batchgrößen hinweg erhalten: Bei einer Batchgröße von 1024 benötigt das FCNN 1,99 µs, während das FCAE 2,37 µs benötigt.

Dieser Unterschied resultiert aus der geringeren Anzahl an Rechenschritten im FCNN. Während das FCNN eine direkte Matrix-Multiplikation durchführt, müssen beim FCAE mehrere Faltungsschichten durchlaufen werden, was zu einer höheren Berechnungszeit führt. Dennoch bleibt das FCAE auch bei großen Batchgrößen in einem effizienten Rechenbereich und profitiert insbesondere bei größeren Eingaben von seiner Architektur.

Speichernutzung

Die Speichernutzung zeigt deutliche Unterschiede zwischen den beiden Architekturen. Das FCAE ist wesentlich kompakter und benötigt deutlich weniger Speicher. Die Modellgröße des FCAE beträgt nur 81 KB, während das FCNN mit 5,79 MB mehr als 70-mal größer ist. Diese Reduktion ergibt sich aus der deutlich geringeren Anzahl an Parametern: Das FCAE besitzt 21.340 Parameter, während das FCNN 1.519.074 Parameter umfasst.

In der GPU-Speichernutzung zeigt sich ein ähnliches Muster. Während das FCNN für eine Batchgröße von 32 27,85 MB Speicher belegt, benötigt das FCAE lediglich 13,09 MB. Mit zunehmender Batchgröße steigt der Speicherbedarf beider Modelle, wobei das FCNN stets deutlich mehr Speicher verbraucht. Bei einer Batchgröße von 1024 liegt der Speicherbedarf des FCNN bei 37,05 MB, während das FCAE mit 47,92 MB leicht darüber liegt. Dies zeigt, dass das FCAE mit zunehmender Batchgröße eine höhere Speichernutzung entwickelt, jedoch im Vergleich zur Modellgröße weiterhin effizient bleibt.

Zusammenfassung

Zusammenfassend bietet das FCNN eine höhere Pfadgenauigkeit und eine kürzere Inferenzzeit, benötigt jedoch deutlich mehr Speicher als das FCAE. Das FCAE überzeugt insbesondere durch seine kompakte Modellgröße, die es für Anwendungen mit begrenztem Speicher attraktiv macht. Während das FCNN mit über 1,5 Millionen Parametern eine enorme Rechenleistung für die Pfadfindung aufbringt, erreicht das FCAE mit nur 21.340 Parametern eine nahezu vergleichbare Genauigkeit. Dies entspricht weniger als 1,5% der Parameteranzahl des FCNN, was die Effizienz der Faltungsarchitektur unterstreicht.

Allerdings wurde der Vergleich zwischen FCNN und FCAE nur für 7×7-Labyrinthe durchgeführt. Die Frage, wie sich das FCAE bei größeren Labyrinthen im Vergleich zu klassischen Suchalgorithmen wie A* verhält, bleibt dabei unbeantwortet. In den folgenden Ab-

schnitten wird daher untersucht, wie das FCAE im direkten Vergleich mit A* auf größeren Labyrinthen von 10×10 bis 50×50 abschneidet. Besondere Aufmerksamkeit gilt dabei der Skalierbarkeit des parametereffizienten FCAE im Vergleich zum deterministischen A*-Algorithmus.

5.1.2 Vergleich von FCAE und A*

Pfadgenauigkeit

Die Ergebnisse zeigen, dass der A*-Algorithmus in allen Labyrinthgrößen eine Pfadgenauigkeit von 100% erreicht hat. Dies war zu erwarten, da A* als deterministischer Algorithmus konzipiert ist und unter den gegebenen Bedingungen stets optimale Lösungen findet. Im Vergleich dazu erzielten die FCAE-Modelle abhängig von der Größe der Labyrinthe eine variierende Pfadgenauigkeit. Während das Modell für die 10×10-Labyrinthe eine nahezu perfekte Genauigkeit von 99,99% erreichte, nahm diese mit zunehmender Labyrinthgröße sukzessive ab. Für die 50×50-Labyrinthe betrug die Pfadgenauigkeit nur noch 75,28%.

Die abnehmende Genauigkeit bei größeren Labyrinthen lässt sich auf mehrere Faktoren zurückführen. Einerseits wird die Komplexität der Labyrinthe mit zunehmender Größe deutlich höher, was die Fähigkeit des Modells, die zugrundeliegenden Muster zu erkennen und präzise Pfade vorherzusagen, einschränken könnte. Andererseits könnte die begrenzte Kapazität der Modelle, insbesondere im Hinblick auf die Anzahl der Parameter, ein limitierender Faktor sein. Ein weiterer möglicher Grund ist die begrenzte Anzahl an Trainingsdaten für die größeren Labyrinthgrößen, was die Generalisierungsfähigkeit der Modelle negativ beeinflussen könnte.

Zusammenfassend lässt sich festhalten, dass die FCAE-Modelle insbesondere bei kleineren Labyrinthgrößen eine hohe Leistungsfähigkeit zeigen, bei zunehmender Größe jedoch an ihre Grenzen stoßen. Dieses Ergebnis unterstreicht die Herausforderungen, neuronale Netzwerke für stark skalierende Probleme wie die Pfadfindung in großen Labyrinthen einzusetzen.

Inferenzzeit

Die Inferenzzeit zeigt signifikante Unterschiede zwischen dem A*-Algorithmus und den FCAE-Modellen. Der A*-Algorithmus weist mit zunehmender Labyrinthgröße eine erhebliche Skalierung der Inferenzzeit auf. Bei 10×10-Labyrinthen beträgt die Inferenzzeit etwa 460 bis 470

Mikrosekunden pro Labyrinth, während sie bei 50×50-Labyrinthen auf durchschnittlich 14.900 bis 15.200 Mikrosekunden ansteigt.

Die FCAE-Modelle demonstrieren eine deutlich effizientere Inferenzzeit. Wenn die Daten bereits auf der GPU liegen benötigt das FCAE-Modell für 10×10-Labyrinthe etwa 44 Mikrosekunden. Für 50×50-Labyrinthe steigt die Inferenzzeit auf etwa 119 Mikrosekunden. Im Vergleich zum A*-Algorithmus zeigt sich somit eine drastische Reduzierung der Rechenzeit, insbesondere bei größeren Labyrinthen.

Ein bemerkenswerter Vorteil der FCAE-Modelle liegt in ihrer Skalierungseffizienz. Während die Inferenzzeit des A*-Algorithmus mit zunehmender Labyrinthgröße deutlich ansteigt, zeigt die Inferenzzeit der FCAE-Modelle eine vergleichsweise moderate Zunahme. Diese Eigenschaft macht die FCAE-Modelle besonders geeignet für Anwendungen, bei denen eine schnelle Pfadfindung in großen und komplexen Labyrinthen erforderlich ist.

Zusätzlich zeigt die Fähigkeit der FCAE-Modelle, auf der GPU parallel zu arbeiten, klare Vorteile in Szenarien, in denen eine große Anzahl von Labyrinthen gleichzeitig verarbeitet werden muss. Die deutlich kürzeren Rechenzeiten pro Labyrinth verdeutlichen die Stärke von neuronalen Netzwerken wie FCAE im Vergleich zu klassischen Algorithmen wie A*.

Die Effizienzsteigerung der FCAE-Modelle gegenüber A* wird durch den Speedup-Faktor weiter quantifiziert. Dieser zeigt, um welchen Faktor FCAE schneller ist als A*. Für kleine Labyrinthe mit 10×10 Feldern beträgt der maximale Speedup bei einer Batchgröße von 256 etwa 31,07. Mit wachsender Labyrinthgröße steigt der Vorteil der FCAE-Modelle weiter an. So erreicht das Modell bei 50×50-Labyrinthen einen maximalen Speedup von 176,24. Dies verdeutlicht die zunehmende Parallelisierungseffizienz der FCAE-Modelle, insbesondere bei großen Eingabegrößen. Während A* mit zunehmender Problemgröße stark an Rechenzeit zunimmt, bleibt die Inferenzzeit der FCAE-Modelle nahezu konstant mit nur moderaten Steigerungen.

Speichernutzung

Die Speichernutzung während der Inferenz zeigt klare Unterschiede zwischen den FCAE-Modellen und dem A*-Algorithmus.

Die GPU-Speichernutzung der FCAE-Modelle nimmt mit zunehmender Labyrinthgröße erheblich zu. Für 10×10-Labyrinthe beträgt die Speichernutzung etwa 19 MB, während sie

bei 50×50-Labyrinthen auf über 239 MB ansteigt. Dieser Anstieg ist auf die Komplexität der Modelle und die größeren Eingabedimensionen zurückzuführen, die sowohl während der Vorwärts- als auch der Rückwärtspropagation eine höhere Speicherauslastung erfordern.

Im Vergleich dazu zeigt der A*-Algorithmus eine deutlich geringere Speichernutzung. Auf der CPU benötigt A* für 10×10-Labyrinthe etwa 0.11 MB und für 50×50-Labyrinthe etwa 0.5 MB. Dies spiegelt die relative Einfachheit des Algorithmus wider, da dieser lediglich den Zustand des Suchraums und die offenen Knoten im Arbeitsspeicher halten muss.

Ein wichtiger Vorteil der FCAE-Modelle ist jedoch, dass sie durch die parallele Verarbeitung auf der GPU optimiert sind und trotz höherem Speicherverbrauch signifikante Zeitvorteile bieten. Dies ist besonders relevant in Szenarien, in denen die Inferenzzeit kritischer ist als der Speicherverbrauch. Auf der anderen Seite bleibt der A*-Algorithmus durch seine geringe Speichernutzung eine effiziente Wahl für Systeme mit begrenzten Hardware-Ressourcen.

Die Ergebnisse zeigen somit ein klares Abwägungsverhältnis zwischen Speichernutzung und Inferenzzeit. Die FCAE-Modelle erfordern zwar mehr Speicher, bieten jedoch eine massiv reduzierte Inferenzzeit, insbesondere bei größeren Labyrinthen. Der A*-Algorithmus ist hingegen speichereffizienter, zeigt jedoch erhebliche Nachteile in der Laufzeit bei großen Labyrinthen.

5.2 Limitierungen und mögliche Verbesserungen

Trotz der vielversprechenden Ergebnisse der FCAE-Modelle und ihrer beeindruckenden Leistung bei kleineren Labyrinthen bestehen mehrere Limitierungen, die sowohl die Ergebnisse dieser Arbeit als auch die allgemeine Anwendbarkeit der Modelle beeinflussen. Im Folgenden werden die zentralen Einschränkungen und mögliche Ansätze zur Verbesserung beschrieben.

Speicher- und Rechenressourcen

Eine der zentralen Limitierungen der FCAE-Modelle liegt in den hohen Anforderungen an Speicher und Rechenleistung, insbesondere bei größeren Labyrinthen. Während das 7x7-Modell problemlos auf der eingesetzten GPU mit 8 GB Speicher trainiert werden konnte, stiegen die Anforderungen bei den 50×50-Modellen so stark an, dass die Trainingskapazität der Hardware erschöpft war. Dies schränkt die Skalierbarkeit der FCAE-Modelle erheblich ein und macht ihre Anwendung auf noch größere Labyrinthe oder ressourcenbeschränkte Umgebungen schwierig.

Generalisierbarkeit

Ein weiterer Schwachpunkt der FCAE-Modelle ist ihre begrenzte Generalisierbarkeit. Jedes Modell wurde speziell auf eine feste Labyrinthgröße trainiert und kann nicht ohne Weiteres auf Labyrinthe anderer Größen angewendet werden. Dies steht im Gegensatz zu klassischen Algorithmen wie A*, die unabhängig von der Eingabegröße arbeiten.

Pfadgenauigkeit bei größeren Labyrinthen

Die Abnahme der Pfadgenauigkeit bei größeren Labyrinthen stellt eine weitere Herausforderung dar. Während die FCAE-Modelle bei kleinen Labyrinthen nahezu perfekte Ergebnisse lieferten, fiel die Genauigkeit bei 50×50-Labyrinthen signifikant ab. Dies deutet darauf hin, dass die Modelle Schwierigkeiten haben, die Muster in größeren und komplexeren Umgebungen zu erfassen.

Um die Pfadgenauigkeit zu verbessern, könnten fortschrittliche Verlustfunktionen, wie etwa gewichtete MSE, eingesetzt werden, die stärker auf kritische Bereiche wie Start- und Endpunkte oder Engpässe fokussieren. Alternativ könnten hybride Ansätze entwickelt werden, bei denen FCAE-Modelle für die initiale Lösungsfindung verwendet werden, während ein klassischer Algorithmus wie A* zur Korrektur von Fehlern dient.

Limitierte Trainingsdaten

Die Trainingsdaten der FCAE-Modelle beschränkten sich auf Labyrinthe mit zufällig generierten Hindernissen und optimalen Pfaden. Dies könnte die Modelle in ihrer Fähigkeit einschränken, komplexere oder realistischere Labyrinthe zu lösen, die möglicherweise andere strukturelle Merkmale aufweisen.

Eine Erweiterung des Datensatzes könnte die Generalisierung und Leistungsfähigkeit der Modelle verbessern. Beispielsweise könnten realitätsnahe Szenarien mit dynamischen Hindernissen oder mehrdimensionalen Labyrinthen in die Trainingsdaten integriert werden, um die Robustheit und Flexibilität der Modelle zu erhöhen.

Fehlende Praxisanwendung

Obwohl die FCAE-Modelle in dieser Arbeit vielversprechende Ergebnisse erzielten, wurde ihre Leistung ausschließlich auf synthetischen Daten getestet. Dies limitiert die Aussagekraft der Ergebnisse in Bezug auf reale Anwendungen, wie beispielsweise in der Roboternavigation oder der Verkehrsplanung.

Zukünftige Arbeiten könnten sich darauf konzentrieren, die Modelle in praxisnahen Szenarien zu evaluieren, um deren Nutzen und Einschränkungen in realen Umgebungen zu analysieren. Dies könnte auch die Integration von Sensordaten oder die Anpassung an dynamische Umgebungen umfassen.

Zusammenfassung der Limitierungen

Zusammenfassend lässt sich sagen, dass die FCAE-Modelle trotz ihrer Vorteile, wie der hohen Inferenzgeschwindigkeit und ihrer Fähigkeit, komplexe Muster zu lernen, durch Speicheranforderungen, Generalisierungsprobleme und eine eingeschränkte Pfadgenauigkeit bei großen Labyrinthen limitiert sind. Diese Limitierungen bieten jedoch zahlreiche Ansatzpunkte für zukünftige Verbesserungen, die die praktische Anwendbarkeit und Effizienz der Modelle weiter steigern könnten.

6 Fazit und Ausblick

6.1 Zusammenfassung der Ergebnisse

Diese Arbeit untersuchte die Leistungsfähigkeit von FCAE-Modellen für die Lösung von Labyrinthen im Vergleich zu klassischen Pfadfindungsalgorithmen wie A*. Ziel war es, die Eignung von FCAE-Modellen für die Pfadfindung zu bewerten und deren Effizienz, Skalierbarkeit und Genauigkeit zu analysieren.

Die Ergebnisse zeigten, dass FCAE-Modell im Vergleich zu einem FCNN bei 7×7-Labyrinthen, bei der Inferenzzeit in Kombination mit kleineren Batchgrößen unterlegen ist, aufgrund der aufwendigeren Berechnungen. Darüber hinaus wurden FCAE-Modelle für Labyrinthe unterschiedlicher Größen (10×10 bis 50×50) entwickelt und getestet. Diese Modelle zeigten eine hohe Pfadgenauigkeit bei kleineren Labyrinthen, jedoch abnehmende Leistungen bei zunehmender Größe. Der A*-Algorithmus erreichte hingegen konsistent eine Pfadgenauigkeit von 100%, wies jedoch längere Inferenzzeiten auf, insbesondere bei größeren Labyrinthen.

Die Skalierbarkeit der FCAE-Modelle wurde durch den exponentiellen Anstieg des GPU-Speicherbedarfs begrenzt, was ihre Anwendung bei sehr großen Labyrinthen einschränkte. Im Vergleich dazu benötigte A* deutlich weniger Speicher und erwies sich als robuster in ressourcenbegrenzten Umgebungen.

Skalierbarkeit und Effizienz der Modelle

Die Analyse der Skalierbarkeit und Effizienz verdeutlichte zentrale Unterschiede zwischen FCAE-Modellen und dem A*-Algorithmus. FCAE-Modelle profitieren von ihrer Fähigkeit, einmal trainiert, sehr schnell auf der GPU inferieren zu können. Insbesondere bei größeren Labyrinthen konnte eine deutliche Reduktion der Inferenzzeit im Vergleich zu A* erzielt werden.

Mit zunehmender Labyrinthgröße stieg jedoch die Speicheranforderungen der FCAE-Modelle an, wodurch die Trainingsfähigkeit bei 50×50-Labyrinthen aufgrund der 8-GB-GPU-Speicherbegrenzung erschöpft war. Dies zeigt, dass die aktuelle Architektur ohne Optimierungen nicht für Labyrinthe jenseits dieser Größenordnung skalierbar ist.

A* zeigte im Gegensatz dazu eine konstante Speichernutzung bei niedrigen Anforderungen, konnte jedoch bei der Inferenzzeit nicht mit den FCAE-Modellen konkurrieren. Diese Eigen-

schaften machen A* robuster in ressourcenbeschränkten Umgebungen, während FCAE-Modelle besser für Szenarien mit Echtzeitanforderungen geeignet sind.

Stärken und Schwächen der FCAE-Modelle

Die Stärke der FCAE-Modelle liegt in ihrer Fähigkeit, komplexe Muster zu erlernen und dadurch effiziente Pfadfindungslösungen mit minimalen Laufzeiten bereitzustellen. Zudem bieten sie den Vorteil, dass sie für eine Vielzahl von Anwendungsfällen angepasst und optimiert werden können, indem die Architektur gezielt modifiziert wird.

6.2 Ausblick

Die Ergebnisse dieser Arbeit bieten mehrere Ansatzpunkte für zukünftige Forschung und Entwicklungen:

- **Optimierung der FCAE-Architektur:** Die Einführung effizienterer Architekturen, beispielsweise durch die Integration von Lightweight-Layern oder die Verwendung von Quantisierungstechniken, könnte den Speicherbedarf und die Rechenzeit der Modelle signifikant reduzieren.
- **Generalisierung der Modelle:** Die Entwicklung von FCAE-Modellen, die auf Labyrinthe unterschiedlicher Größen angewendet werden können, wäre ein wichtiger Fortschritt.
- **Hybride Ansätze:** Die Kombination von FCAE-Modellen mit klassischen Algorithmen wie A* könnte deren jeweilige Stärken vereinen. Beispielsweise könnte A* verwendet werden, um Engstellen oder schwer zugängliche Bereiche in einem Labyrinth zu lösen, während FCAE-Modelle die restlichen Bereiche effizient verarbeiten.
- **Einsatz in der Praxis:** Eine weitere spannende Forschungsrichtung wäre die Anwendung der Modelle in realen Szenarien, wie der Roboternavigation oder der Planung von Lieferketten, um deren Praxistauglichkeit zu evaluieren.

Die Arbeit zeigt, dass FCAE-Modelle ein vielversprechender Ansatz für die Pfadfindung sind, jedoch weitere Optimierungen und Forschungen notwendig sind, um ihr volles Potenzial auszuschöpfen.

Literaturverzeichnis

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4. Aufl.). MIT Press.

Cui, X., & Shi, H. (2011). A*-based Pathfinding in Modern Computer Games. *International Journal of Computer Science and Network Security*, *11*(1).

Dechter, R., & Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *Journal of the ACM (JACM)*, *32*(3).

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*.

Dumoulin, V., & Visin, F. (2018). A guide to convolution arithmetic for deep learning. <https://arxiv.org/abs/1603.07285>.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Grabrovsek, P. (2019). Analysis of Maze Generating Algorithms. *IPSI Transactions on Internet Research*.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, *4*(2).

Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. <https://arxiv.org/abs/1502.03167>.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Latombe, J.-C. (1991). *Robot Motion Planning*. Springer.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature*, *521*, 436–44. https://doi.org/10.1038/nature14539

Loizou, S. G., & Kumar, V. (2006). Maze-solving agents: From human to robots. *IEEE Robotics & Automation Magazine*, *13*(3).

Loomis Jr., H. H. (2012). Maze solving using robot control and computer vision. *Journal of Computing Sciences in Colleges*, *28*(2).

Masci, J., Meier, U., Cireșan, D., & Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. *International Conference on Artificial Neural Networks*. https://doi.org/10.1007/978-3-642-21735-7_7

Moore, E. F. (1959). The shortest path through a maze. *Proceedings of an International Symposium on the Theory of Switching*.

Mussmann, S., & See, A. (n. d.). Graph Search Algorithms. <https://cs.stanford.edu/people/abisee/gs.pdf>.

Odena, A., Dumoulin, V., & Olah, C. (2016). Deconvolution and Checkerboard Artifacts. *Distill*. https://doi.org/10.23915/distill.00003

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*.

Russell, S., & Norvig, P. (2016). *Artificial Intelligence A Modern Approach, Global Edition* (3. Aufl.). Pearson Deutschland. <https://elibrary.pearson.de/book/99.150005/9781292153971>.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1).

Sturtevant, N. R. (2012). Grid-based pathfinding on the 15-puzzle. *Proceedings of the Fifth Annual Symposium on Combinatorial Search*.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*(2).

Vlad. (2023). Maze Solver Neural Net [Zugriff am 28. November 2024]. <https://github.com/Vlad/Maze-Solver-Neural-Net>.

Zhu, W., Yeh, W.-C., Chen, J., Chen, D., Li, A., & Lin, Y. (2019). Evolutionary Convolutional Neural Networks Using ABC. *ICMLC '19: Proceedings of the 2019 11th International Conference on Machine Learning and Computing*, 156–162. https://doi.org/10.1145/3318299.3318301

Anhang

Anhang 1: FCAE-Architekturen

fcae.py

```
import torch
import torch.nn as nn

class FCAE_7x7(nn.Module):
    def __init__(self):
        super(FCAE_7x7, self).__init__()

        self.encoder = nn.Sequential(
            nn.Conv2d(1, 15, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(15),
            nn.Dropout2d(p=0.2),

            nn.Conv2d(15, 24, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.BatchNorm2d(24),
            nn.Dropout2d(p=0.2),

            nn.Conv2d(24, 33, kernel_size=3, stride=1, padding=0),
            nn.ReLU(),
            nn.BatchNorm2d(33),
        )

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(33, 24, kernel_size=3, stride=1, padding
                =0),
            nn.ReLU(),
            nn.BatchNorm2d(24),

            nn.ConvTranspose2d(24, 15, kernel_size=3, stride=1, padding
                =0),
            nn.ReLU(),
            nn.BatchNorm2d(15),

            nn.ConvTranspose2d(15, 1, kernel_size=3, stride=1, padding=1)
                ,
```

```
34              nn.ReLU(),
35          )
36
37      def forward(self, x: torch.Tensor):
38          x = self.encoder(x)
39          x = self.decoder(x)
40          return x
41
42
43  class FCAE_10x10(nn.Module):
44      def __init__(self):
45          super(FCAE_10x10, self).__init__()
46
47          self.encoder = nn.Sequential(
48              nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
49              nn.ReLU(),
50              nn.BatchNorm2d(32),
51              nn.Dropout2d(p=0.1),
52
53              nn.Conv2d(32, 32, kernel_size=2, stride=2, padding=0),
54              nn.ReLU(),
55              nn.BatchNorm2d(32),
56              nn.Dropout2d(p=0.1),
57
58              nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
59              nn.ReLU(),
60              nn.BatchNorm2d(64),
61
62              nn.Conv2d(64, 128, kernel_size=2, stride=2, padding=0),
63              nn.ReLU(),
64              nn.BatchNorm2d(128),
65
66              nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
67              nn.ReLU(),
68              nn.BatchNorm2d(128),
69
70              nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
71              nn.ReLU(),
72              nn.BatchNorm2d(256),
73          )
74
75          self.decoder = nn.Sequential(
76              nn.ConvTranspose2d(256, 128, kernel_size=3, stride=1, padding
                    =1),
77              nn.ReLU(),
```

```
78              nn.BatchNorm2d(128),
79
80              nn.ConvTranspose2d(128, 128, kernel_size=3, stride=1, padding
                    =1),
81              nn.ReLU(),
82              nn.BatchNorm2d(128),
83
84              nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2, padding
                    =0, output_padding=1),
85              nn.ReLU(),
86              nn.BatchNorm2d(64),
87
88              nn.ConvTranspose2d(64, 32, kernel_size=3, stride=1, padding
                    =1),
89              nn.ReLU(),
90              nn.BatchNorm2d(32),
91
92              nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2, padding
                    =0),
93              nn.ReLU(),
94              nn.BatchNorm2d(32),
95
96              nn.ConvTranspose2d(32, 1, kernel_size=3, stride=1, padding=1)
                    ,
97              nn.ReLU(),
98          )
99
100     def forward(self, x: torch.Tensor):
101         x = self.encoder(x)
102         x = self.decoder(x)
103         return x
```

Anhang 2: A*-Implementierung

a_star.py

```
1  import numpy as np
2
3  def a_star(maze):
4      rows, cols = maze.shape
5      gates = np.argwhere(maze == 3)
6      if len(gates) < 2:
```

```
7           return maze
8
9       start = tuple(gates[0])
10      end = tuple(gates[1])
11
12      def heuristic(a, b):
13          return abs(a[0] - b[0]) + abs(a[1] - b[1])
14
15      directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
16
17      open_set = [start]
18      came_from = {}
19      g_score = {start: 0}
20      f_score = {start: heuristic(start, end)}
21
22      while open_set:
23          current = min(open_set, key=lambda x: f_score.get(x, float("inf")
                ))
24          open_set.remove(current)
25
26          if current == end:
27              solved_maze = np.where(maze == 1, 0, maze)
28              x, y = current
29              while current in came_from:
30                  x, y = current
31                  if solved_maze[x, y] == 0:
32                      solved_maze[x, y] = 2
33                  current = came_from[current]
34              return solved_maze
35
36          for dx, dy in directions:
37              neighbor = (current[0] + dx, current[1] + dy)
38
39              if (
40                  0 <= neighbor[0] < rows
41                  and 0 <= neighbor[1] < cols
42                  and maze[neighbor] != 1
43              ):
44                  tentative_g_score = g_score[current] + 1
45
46                  if tentative_g_score < g_score.get(neighbor, float("inf")
                        ):
47                      came_from[neighbor] = current
48                      g_score[neighbor] = tentative_g_score
```

```
49                     f_score[neighbor] = tentative_g_score + heuristic(
                           neighbor, end)
50                     if neighbor not in open_set:
51                         open_set.append(neighbor)
52
53     return maze
```

Anhang 3: Pfadevaluierung

evaluate.py

```
1  import torch
2
3  def evaluate_found_path(answer: torch.Tensor, Y_test: torch.Tensor):
4      start = find_gate(Y_test, 0.8, True, 3)
5      end = find_gate(Y_test, 0.8, False, 3)
6
7      if start is None or end is None:
8          return False
9
10     current = (start[0], start[1])
11     visited = set()
12     visited.add(current)
13
14     max_moves = (
15         answer.shape[0] * answer.shape[1]
16     )
17
18     for i in range(max_moves):
19         brightest_neighbour = find_brightest_neighbour(answer, current,
               visited, Y_test)
20         if brightest_neighbour is None:
21             return False
22         current = (brightest_neighbour[0], brightest_neighbour[1])
23         visited.add(current)
24         if current == end:
25             return True
26     return False
27
28 def find_gate(answer: torch.Tensor, threshold: float, start: bool,
       gate_value: float = 1.0):
29     min_value = gate_value - threshold
```

```
30      max_value = gate_value + threshold
31
32      gate_candidates = ((answer >= min_value) & (answer <= max_value)).
            nonzero(as_tuple=True)
33
34      if len(gate_candidates[0]) == 0:
35          return None
36
37      if start:
38          return gate_candidates[0][0].item(), gate_candidates[1][0].item()
39      else:
40          return gate_candidates[0][-1].item(), gate_candidates[1][-1].item
                ()
41
42  def find_brightest_neighbour(
43      answer: torch.Tensor,
44      position: tuple[int, int],
45      visited: set[tuple[int, int]],
46      Y_test: torch.Tensor,
47  ):
48      rows, cols = answer.shape
49
50      ind_up = (max(position[0] - 1, 0), position[1])
51      ind_down = (min(position[0] + 1, rows - 1), position[1])
52      ind_left = (position[0], max(position[1] - 1, 0))
53      ind_right = (position[0], min(position[1] + 1, cols - 1))
54
55      value_up = answer[ind_up[0]][ind_up[1]] if ind_up not in visited else
            -1
56      value_down = answer[ind_down[0]][ind_down[1]] if ind_down not in
            visited else -1
57      value_left = answer[ind_left[0]][ind_left[1]] if ind_left not in
            visited else -1
58      value_right = answer[ind_right[0]][ind_right[1]] if ind_right not in
            visited else -1
59
60      max_value = max(value_up, value_down, value_left, value_right)
61      if max_value == -1:
62          return None
63
64      if max_value == value_up and Y_test[ind_up[0]][ind_up[1]] != 1.0:
65          return ind_up
66      elif max_value == value_down and Y_test[ind_down[0]][ind_down[1]] !=
            1.0:
67          return ind_down
```

```
68      elif max_value == value_left and Y_test[ind_left[0]][ind_left[1]] !=
            1.0:
69          return ind_left
70      elif max_value == value_right and Y_test[ind_right[0]][ind_right[1]]
            != 1.0:
71          return ind_right
72
73      return None
```

Eidesstattliche Versicherung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Literatur und Hilfsmittel angefertigt habe. Wörtlich übernommene Sätze und Satzteile sind als Zitate belegt, andere Anlehnungen hinsichtlich Aussage und Umfang unter Quellenangabe kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und ist auch noch nicht veröffentlicht.

Ort, Datum: Warstein, 17.02.25 Unterschrift: Luc Westbomke