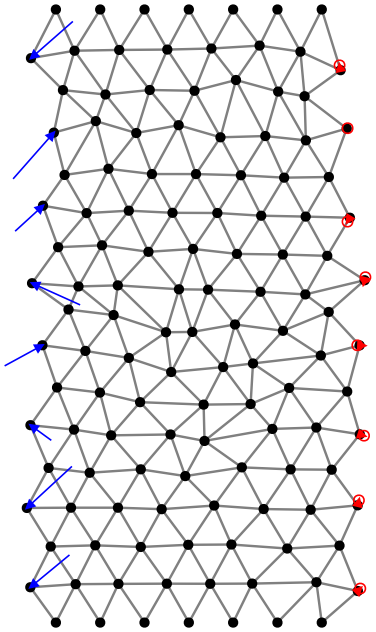
Analyse der Optimierungsverfahren mechanischer neuronaler Netzwerke



Alexander Reimer **Matteo Friedrich**

Gymnasium Eversten Oldenburg

Betreuer: Herr Dr. Glade

Inhaltsverzeichnis

Zusammenfassung

Wir wollen uns mit dem neuen, noch vergleichsweise wenig erforschten Bereich der *mechanical neural networks*, kurz MNNs, beschäftigen. MNNs sind programmierbare Materialien, welchen verschiedene Verhaltensweisen, wie zum Beispiel ein bestimmtes Verformungsverhalten, antrainiert werden können. Sie bestehen aus Massepunkten (genannt Neuronen), welche durch Federn miteinander verbunden werden. Ihr Verhalten ergibt sich durch die Steifheiten der Federn. Die grundlegende Annahme von MNNs ist, dass diese Federkonstanten in zukünftigen Materialien einzeln angepasst werden können. In Analogie zu künstlichen neuronalen Netzwerken wäre es dann prinzipiell möglich, durch eine geeignet gewählte Konfiguration an Federkonstanten verschiedene Verhaltensweisen auf externe Kräfte anzutrainieren. Während sich die bisherige Forschung auf die technische, physische Implementation dieser Netzwerke fokussiert hat, wollen wir das Trainingsverfahren optimieren. Dazu haben wir bereits eine Simulation eines MNNs umgesetzt, die bisher verwendeten Algorithmen (evolutionäres Lernen und Pattern Search) selbst implementiert, sowie mit neuen Parametern ausprobiert und verglichen. Dafür haben wir uns jedoch auf die Anwendung dieser Algorithmen in Simulationsrechnungen beschränkt. Die Ergebnisse sollten dennoch einen guten Startpunkt für reale MNNs bieten. Der von uns entwickelte Code ist die erste öffentlich verfügbare Implementation eines MNNs. Unsere Ergebnisse zeigen, dass MNNs mehrere komplexe Verhaltensweisen lernen können. Diese intelligenten Materialien eröffnen vielfältige zukünftige technologische Anwendungsmöglichkeiten.

1 Einleitung

Inspiration für dieses Projekt ist ein Artikel von 2022 mit dem Titel „Mechanical neural networks: Architected materials that learn behaviors“ von Lee *et al.* (2022) [3]. Es ist der erste uns bekannte veröffentlichte Artikel, der sogenannte *mechanical neural networks*, kurz MNNs, beschreibt – ein neuer Forschungsbereich, in dem neuronale Netzwerke in der physischen Welt umgesetzt werden (s. Abb. 1). Im Gegensatz zu bisherigen mathematischen und elektrischen Implementationen handelt es sich hier um eine mechanische, bei der Federn mit variabler Steifheit miteinander verbunden werden. Abhängig von den Härten der Federn (Federkonstanten) weist das Material unterschiedliche Verhaltensweisen bei Krafteinwirkung auf. Ein MNN kann also als anpassbares und durch Sensoren sogar lernfähiges Material verwendet werden, welches seinen Bedingungen oder gewünschten Verwendungszwecken angepasst werden kann. Dazu kann das Material sowohl in einer Simulation als auch physisch, durch Sensoren in den Federn, trainiert werden.

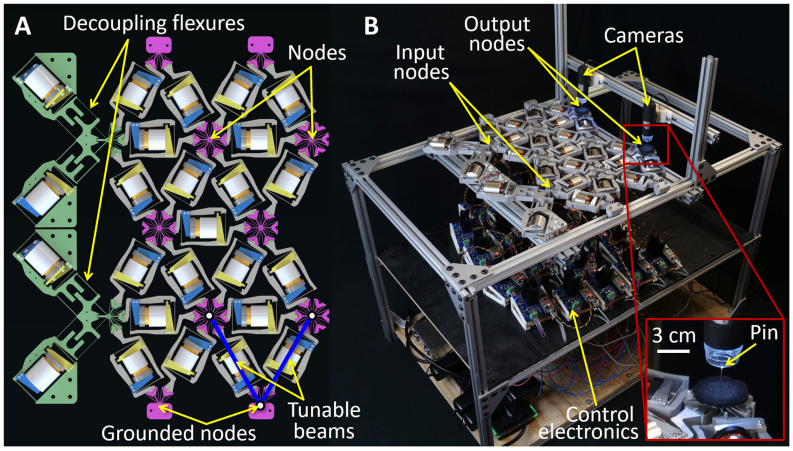


Abbildung 1: Mechanische Umsetzung eines MNN (Abb. von [3])

Durch ihre Eigenschaften könnten MNNs viele Verwendungszwecke haben, von der Optimierung von Flugzeugflügeln abhängig von aktuellen Gegebenheiten wie Windstärke und -winkel [3, S. 2] über Optimierung der Schockabsorption von Schutzwesten bis zur dynamischen Änderung der Resonanzfrequenz eines Gebäudes zum Schutz gegen Erdbeben [2, S. 9]. Wenn es möglich ist, die Resonanz auf Schwingungen eines MNNs zu trainieren, könnte man dies auch für einbruchsichere Brücken oder vielleicht sogar bessere Musikinstrumente nutzen.

In diesem Projekt wollen wir verschiedene Optimierungsalgorithmen für MNNs vergleichen. Da wir nichts passendes finden konnten, wollen wir zuerst eine eigene Programmbibliothek zur Simulation, Optimierung, Bewertung und Visualisierung von MNNs erstellen. Diese soll nutzerfreundlich und öffentlich sein, damit wir und andere sie als Basis für weiterführende Projekte nutzen können.

Zur Bewertung der Optimierungsalgorithmen und -parameter sollen MNNs simuliert werden. Ziel dieses Projektes ist es aufgrund des hohen Aufwands nicht, ein MNN selbst physisch umzusetzen; es wird jedoch davon ausgegangen, dass ein Vergleich der Verfahren in einer Simulation auch korrekte Aussagen über das physische Trainieren liefern wird und ein „Vortrainieren“ eines physischen MNN mit einer Simulation die Gesamtzeit des Trainierens verkürzen kann, somit also auch die in einer Simulation verwendeten Optimierungsverfahren relevant sind.

2 Hintergrund und theoretische Grundlagen

Mechanische Neuronale Netzwerke (MNNs) scheinen künstlichen neuronalen Netzwerken (*artificial neural networks*, kurz ANNs) in ihrem Aufbau zwar ähnlich (vgl. Abb. 2 und 1), unterscheiden sich in ihrer Funktionsweise und Umsetzung jedoch stark.

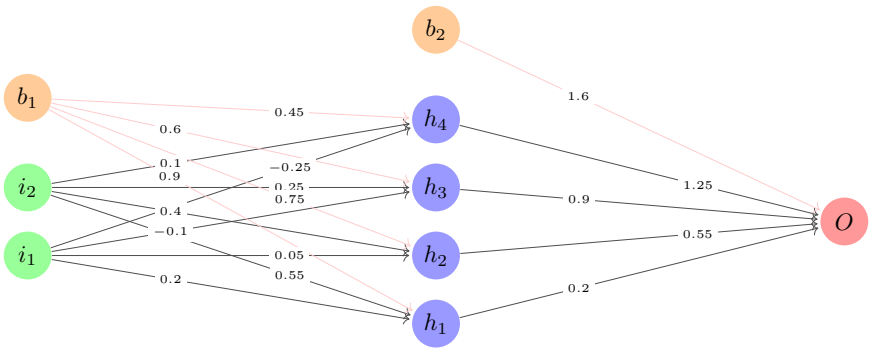


Abbildung 2: Ein Beispiel für ein neuronales Netzwerk bestehend aus einer Eingabeschicht(i_1 und i_2) und zwei vollständig verbundene Schichten (Neuronen h_1 bis h_4 mit Bias b_1 und Neuron O mit Bias b_2). Die Kreise stellen Neuronen dar; die Zahlen auf den Pfeilen die Gewichte zwischen den jeweiligen Neuronen.

Die von Lee *et al.* (2022) vorgeschlagene Architektur besteht aus drei Komponenten: zwei fixierten und parallelen Wänden sowie miteinander verbundenen Federn, welche in gleichseitigen Dreiecken angeordnet sind (s. Abb. 1, links). Die gleichseitigen Dreiecke wurden aufgrund ihres höheren Trainingserfolgs gegenüber Quadraten gewählt (vgl. [3, Abb. 5]). Wir nennen die Knotenpunkte zwischen Federn sowie Befestigungspunkte an den „Wänden“ äquivalent zu ANNs Neuronen. Dabei sollte noch einmal klar gestellt werden, dass die Knotenpunkte keine echten Neuronen sind, sondern Massepunkten eines mechanischen physikalischen Systems entsprechen. Durch die Fixierung der äußeren Neuronen auf zwei gegenüberliegenden Seiten durch die Wände wird die Bewegung der Federn so eingeschränkt, dass es eine klare Eingabe auf der einen Seite und Ausgabe auf der anderen gibt.

Während ANNs nur rein mathematische Konstrukte sind, sind MNNs physisch, was Vor- und Nachteile mit sich bringt. So eignen sich MNNs nicht für Datenverarbeitung, u.a. aufgrund der Begrenzung auf hier zwei, maximal drei Dimensionen und, wie später noch beschrieben wird, stellt die technische Umsetzung ebenfalls Schwierigkeiten dar. Doch sie bieten als analoges System einen großen Vorteil: Während ANNs für die Evaluierung von Eingaben Zeit und Ressourcen benötigt, funktionieren trainierte MNNs ohne signifikante Verzögerung.

Gegenüber „konventionellen“ Materialien bieten sie zwei hauptsächliche Vorteile:

1. Es können mehrere Verhaltensweisen auf externe Kräfte gleichzeitig antrainiert werden.
2. Bei der Implementation von Lee *et al.* (2022) kann das MNN durch Sensoren auch beim Einsatz weiterlernen, sodass es sich von selbst an Beschädigung oder Abnutzungen, Größenänderung usw. anpassen kann.

2.1 Simulation von MNNs

Da ein MNN nur aus miteinander verbundenen Federn besteht, lässt sich die Kraft, die auf jedes Neuron wirkt, durch die Summe aller einzelnen Kräfte, die jede Feder auf das Neuron ausübt, beschreiben. Um diese einzelnen Kräfte zu bestimmen, benötigt man eine Funktion, die mithilfe der Auslenkung (Entfernung von der Ruhelage) und der Steifheit der Feder (Federkonstante) die Kraft bestimmen kann. Hierfür wird oft das Hooke'sche Gesetz genutzt, welches die Kraft F als Produkt von Federkonstante k und Auslenkung x angibt [7]. Es gilt also:

$$F = -k * x$$

Um nun mit der Formel für die Kraft, die auf die Neuronen wirkt, die Position dieses Neuronen als Funktion der Zeit angeben zu können, um das Verhalten des MNNs unter verschiedenen Krafteinflüssen zu

analysieren, muss man die obige Formel mithilfe des Newton'schen Gesetzes zu einer Differenzialgleichung für die Beschleunigung des Neurons ($m\ddot{x}$) umwandeln. Im einfachsten Fall, für ein einzelnes bewegliches Neuron und eine Feder, ergibt sich die Gleichung

$$m\ddot{x} = -kx$$

Diese Gleichung kann nun von einem Computerprogramm numerisch gelöst werden. Dafür haben wir das Package DifferentialEquations.jl verwendet. Um die Ergebnisse zu validieren, wurden die Gleichungen zusätzlich noch nach einem selbst programmierten Eulerverfahren gelöst. Dabei addieren wir jeweils in sehr kurzen Zeitabständen das Produkt der berechneten Beschleunigung und des Zeitabstandes zu der Geschwindigkeit jedes Neurons hinzu und aktualisieren die Positionen aller Neuronen mithilfe der berechneten Geschwindigkeiten.

Zusätzlich muss noch eine Änderung durchgeführt werden, die es ermöglicht, den Ruhezustand des Systems bei bestimmten Krafteinflüssen zu ermitteln. Damit das MNN überhaupt einen Ruhezustand erreichen kann, muss es gedämpft werden. Ohne diese Dämpfung würde das Netzwerk einfach immer weiter schwingen. Die Dämpfung lässt sich durch einen zusätzlichen Term $-\gamma\dot{x}$ beschreiben, also als das Produkt aus Dämpfungskonstante γ und der Geschwindigkeit $\dot{x}$ jedes Neurons. Durch diese Kraft werden die Neuronen abgebremst.

Jedoch können MNNs, welche nur positive Federkonstanten besitzen, nicht alle möglichen Verhaltensweisen lernen, da sie nur Kräfte durch das Material „transportieren". Wenn zum Beispiel eine Kraft, welche nach rechts ausgerichtet ist, auf ein MNN trifft, werden sich alle Neuronen des MNNs nach rechts bewegen (s. Abb. 4). Um auch Verhalten, welche Bewegung entgegen einer Kraft benötigen, umzusetzen, muss dem System Energie zugefügt werden. Dies lässt sich mit negativen Federkonstanten umsetzen. Federn mit negativen Federkonstanten stoßen Neuronen, die sich weit von einander entfernt befinden, ab und ziehen nahe Neuronen noch weiter an (s. Abb. 3). Jedoch sind MNNs mit negativen Federkonstanten nicht stabil, da es zum Beispiel passieren kann, dass sich zwei durch eine Feder mit negativer Federkonstante verbundene Neuronen voneinander entfernen, wodurch sie sich weiter abstoßen würden, wodurch sie sich weiter voneinander entfernen. Um so einen Kreislauf zu verhindern und um die Stabilität des MNNs zu gewährleisten, muss von einem linearen Verhältnis von Kraft und Distanz abgewichen werden. Es muss eine Auslenkung geben, bei welcher die Kraft, die auf die Neuronen wirkt, auch bei negativen Federkonstanten entgegen der Richtung des Auslenkungsvektors wirkt (s. Abb. 3). In diesem Projekt wurde dafür eine Funktion dritten Grades gewählt:

$$F(x,k) = -k * x - x^3$$

Da wir die MNNs jedoch im 2-dimensionalen Raum simulieren wollen, müssen wir die Beschleunigung auch als 2-dimensionalen Vektor angeben, indem wir die Kraft noch mit einem normierten Richtungsvektor multiplizieren, der die Richtung der Feder und demnach auch der Kraft beschreibt. Weiterhin müssen alle Kräfte, die auf ein Neuron wirken, aufsummiert werden. Die vollständige Differenzialgleichung lautet also:

$$m\ddot{\vec{x}}_i = \sum_{j \in \mathrm{NN}(i)} \left(F\left(\Delta x_{ij} - l, k_{ij}\right) * \frac{\Delta\vec{x}_{ij}}{\Delta x_{ij}} \right) - \gamma\dot{\vec{x}}_i + \vec{F}_{\mathrm{ext},i}$$

Hier ist $\Delta\vec{x}_{ij} = \vec{x}_i - \vec{x}_j$, also der Abstandsvektor zweier Neuronen, $\Delta x_{ij} = \|\Delta\vec{x}_{ij}\|$, l die Ruhelänge der Feder, $\vec{F}_{\mathrm{ext},i}$ die externe Kraft auf Neuron i und die Summe läuft über die nächsten Nachbarn $j = \mathrm{NN}(i)$ des Neurons.

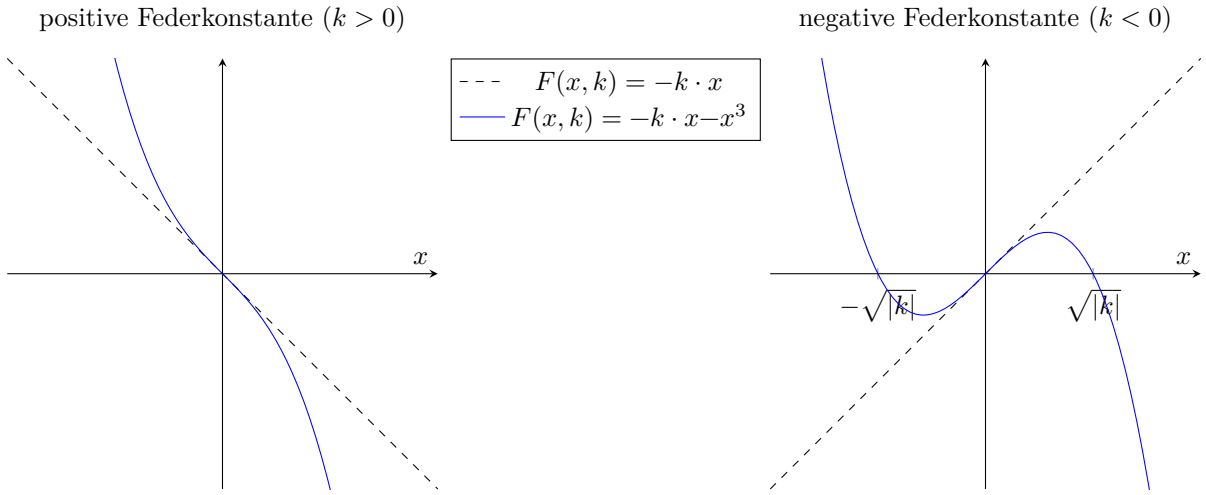


Abbildung 3: Skizze der verwendeten Federkraft $f(x)$ in Abhängigkeit der Auslenkung x für den Fall einer positiven Federkonstante $k > 0$ (links) und einer negativen Federkonstante $k < 0$ (rechts). Die gestrichelte Linie zeigt einen linearen Kraftverlauf $f(x) = -kx$. Im Falle einer positiven Federkonstante (links) entspricht dies dem Hooke'schen Gesetz und führt zu einem stabilen Gleichgewicht, da eine positive Auslenkung der Feder zu einer negativen Kraft führt und umgedreht. Im Falle einer negativen Federkonstante ist das Gleichgewicht $x = 0$ instabil und eine leichte Auslenkung der Feder führt zu einer unendlich großen Auslenkung. Um diese Instabilität zu vermeiden, verwenden wir die Kraft $f(x) = -kx - x^3$ (blaue durchgezogene Linie). Im Falle von $k > 0$ führt dies zu einem leicht nichtlinearen Kraftverlauf, aber immer noch stabilem Gleichgewicht (links). Für $k < 0$ ist das Gleichgewicht immer noch instabil, aber für große Auslenkungen $|x| > \sqrt{|k|}$ wirkt die Kraft stabilisierend

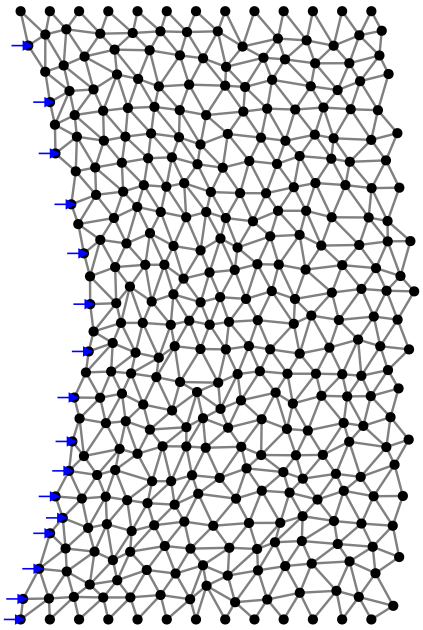


Abbildung 4: Typischer Aufbau eines MMNs. Dieses besteht aus Neuronen (schwarze Kreise), die durch Federn (schwarze Linien) verbunden sind. Die Neuronen am oberen und unteren Ende sind mechanisch fixiert und können sich nicht bewegen. Die blauen Pfeile stellen die Krafteinwirkung auf die Neuronen dar.

2.2 Optimierungsverfahren von MNNs

Um ein MNN zu optimieren, benötigt man zuerst einmal einen Trainingsdatensatz an verschieden Verhaltensweisen, welche erlernt werden sollen. In unserem Fall besteht ein Verhalten aus den Kräften, die auf die Neuronen in der ersten Schicht wirken, und die Änderungen der Positionen der Neuronen in der letzten Schicht, die durch die gegeben Kräfte bewirkt werden sollen. Mithilfe eines Trainingsdatensatzes lässt sich nun auch bestimmen, wie gut ein Modell die Verhaltensweisen umsetzt, indem man die Entfernung jedes Neurons zu den in den Trainingsdaten gegebenen Positionen berechnet und für alle Neuronen aufsummiert. Man berechnet also die Abweichungen der Neuronen zu den Trainingsdaten. Zuletzt wird diese Abweichung noch quadriert. Diese Verfahren nennt sich *mean squared error* (MSE).

2.2.1 Genetische Algorithmen

Genetische Algorithmen können MNNs optimieren, indem sie sich an Ideen der Evolution orientieren [6]. Mehrere MNNs bilden eine Population ab. Der genetische Algorithmus beginnt damit, jedem MNN der Population einen *score* zu geben, welcher beschreibt, wie gut dieses MNN die Trainingsdaten abbilden kann. Das Ziel ist es, diese Zahl zu minimieren. Die besten MNNs werden ohne Mutation direkt an die nächste Generation übergeben. Anschließend wird der Rest der folgenden Population durch *crossover* bestimmt. Dies funktioniert, indem zwei MNNs aus der Population ausgewählt werden, wobei bessere MNNs eine höhere Wahrscheinlichkeit haben, ausgewählt zu werden. Die Federkonstanten der beiden ausgewählten MNNs werden beim *crossover* genutzt, um ein neues MNN zu erstellen, das Federkonstanten beider „Elternteile“ besitzt. Am Ende einer Iteration werden noch zufällige Mutationen vorgenommen. Die Stärke dieser Mutation nennt sich Lernrate. Dieses Verfahren wird solange wiederholt, bis ein zufriedenstellendes MNN gefunden wurde.

2.2.2 Partial Pattern Search

Bei dem von Lee *et al.* (2022) verwendeten *partial pattern search* (PPS) Verfahren werden zu Beginn alle Federkonstanten auf den gleichen voreingestellten Wert gesetzt, hierfür haben wir den von Lee *et al.* (2022) verwendeten Wert von 1,15 übernommen. Weiter gibt es zu Beginn eine festgesetzte Änderungsrate von 1. Nun wird in zufälliger Reihenfolge zu jeder Federkonstante die aktuelle Änderungsrate addiert. Falls der MSE mit der veränderten Federkonstante nicht niedriger ist als vorher, wird die Änderung rückgängig gemacht. Sollte es in einem Durchlauf aller Federkonstanten keine Verbesserung des MSE gegeben haben, wird das Vorzeichen der Änderungsrate umgekehrt und, falls diese bereits negativ war, um 10 % gesenkt. Mit den genannten Werten wäre die Entwicklung der Änderungsrate also $1 \to -1 \to 0.9 \to -0.9 \to 0.81\ldots$ Dies wird für eine vorgegebene Anzahl an Wiederholungen durchgeführt, wobei eine Wiederholung nicht einen Durchlauf aller Federkonstanten, sondern die Anwendung der aktuellen Änderungsrate auf eine einzelne Federkonstante bezeichnet.

3 Vorgehensweise

3.1 Materialien

- Laptop (i7-11800H, 16 GB RAM, RTX 3060)
- Julia
  - DifferentialEquations.jl und LinearAlgebra.jl für Autodifferenzierung der Simulation des Federsystems
  - Graphs.jl und MetaGraphsNext.jl für Modellierung des Netzwerks als Graph
  - Plots.jl, GLMakie.jl und Observables.jl für Visualisierung
  - CSV.jl und Statistics.jl für Speicherung und Verarbeitung von Versuchsergebnissen

3.2 Methoden

Um zu testen, wie gut sich ein MNN mit den verschiedenen Algorithmen optimieren lässt, wird zuerst eine Architektur für die MNNs benötigt. In diesem Projekt wurde eine bereits vorgeschlagene Struktur verwendet [3], damit eine bessere Vergleichbarkeit zu den bereits gefundenen Ergebnissen besteht. Dabei werden die Neuronen in gleichseitigen Dreiecken angeordnet und verbunden. Diese Dreiecke werden in einem Gitter zusammengebracht, welches oben und unten fixiert ist. Die auf das System einwirkenden Kräfte greifen an der linken Seite des MNNs an und die gewünschten Veränderungen finden auf der rechten Seite statt. Diese Architektur wurde nicht dafür entwickelt, ein reales Szenario, in dem MNNs eingesetzt werden könnten, zu simulieren, sondern dient dazu verschiedene Optimierungsalgorithmen mit einer vereinfachten Problemstellung zu testen und zu verstehen, bevor man sie einsetzt.

Für eine systematische Analyse müssen zusätzlich zu der Architektur der MNNs auch die verschiedenen Verhaltensweisen festgelegt werden. Dafür nutzen wir zufällig generierte Vektoren für die einwirkenden Kräfte (Kraftvektoren) sowie die gewünschten Positionsänderungen in der letzten Schicht (Zielvektoren). Jedes Neuron der letzten Schicht hat dabei ein 50 % Chance, dass der Zielvektor $\begin{pmatrix}0\\0\end{pmatrix}$ ist, es also trotz der einwirkenden Kräfte auf der Stelle bleiben „soll“. Bei den Kraftvektoren hat jedes Neuron der ersten Schicht eine 50 % Chance, keine einwirkende Kraft zu haben, jedoch gibt es immer mindestens einen Kraftvektor.

Die Kraftvektoren werden während jedem Simulationsschritt auf die entsprechenden Neuronen angewendet, die Zielvektoren legen die Zielpositionen ausgehend von der ursprünglichen Startposition fest.

Bei diesem Vorgehen sind unmögliche Kombinationen von Verhaltensweisen möglich, z.B. entgegengesetzte Positionsänderungen bei gleichen einwirkenden Kräften (s. Abb. 5).

Um dies zu verhindern, überprüfen wir beim Generieren mehrerer Verhaltensweisen, dass für ein Neuron der gerade zufällig generierte Kraft- / Zielvektor mit allen anderen Vektoren für dieses Neuron mindestens einen vorgegebene Mindestwinkel bildet. Sollte dies nicht zutreffen, wird der Vektor neu erstellt. Für das Generieren von Verhaltensweisen gibt es so drei Parameter:

- Der Mindestwinkel zwischen Vektoren
- Die Skalierung der Länge der Kraftvektoren
- Die Skalierung der Länge der Zielvektoren

Dieses Vorgehen verhindert zwar unmögliche Szenarien, schließt jedoch auch womöglich schwere, aber nicht unmögliche Kombinationen von Verhaltensweisen aus (s. Abb. 6), zu Beginn sollte es jedoch zum Vergleichen der Optimierungsalgorithmen reichen.

Der von uns geschriebene Quellcode lässt sich auf höchster Ebene in zwei Teile aufteilen: Unser Paket MNN.jl mit der Implementierung des MNN und der Optimierungsverfahren sowie das Skript Compare.jl für das programmatische Testen und Vergleichen verschiedener Hyperparamerter. Unter Hyperparameter verstehen wir einen Parameter, welcher nicht Teil des Netzwerks ist (also z.B. keine Federkonstanten), sondern die Struktur sowie das Training dieses Netzwerks beeinflusst (z.B. Optimierungsalgorithmus, Epochen, Mindestwinkel für Verhaltensweisen oder Dimensionen des Netzwerks).

In Compare.jl verwenden wir CSV.jl zusammen mit DataFrames.jl, um verwendete Hyperparameter sowie den MSE und die Anzahl an bereits trainierten Epochen in CSV-Dateien für spätere Analyse und Vergleiche zu speichern. Das Speichern erfolgt dabei alle fünf Epochen.

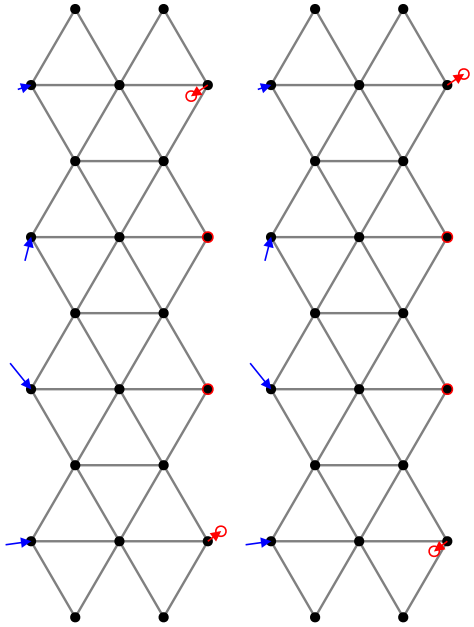


Abbildung 5: Zwei Verhaltensweisen, die gemeinsam unmöglich erfolgreich anzutrainieren sind. Die schwarzen Punkte sind die Neuronen, die schwarzen Verbindungslinien die Federn. Die obersten und untersten zwei Neuronen sind jeweils fixiert. Bei beiden Verhaltensweisen sind die Kraftvektoren (blau) gleich, die Zielpositionen (rote Kreise) unterscheiden sich jedoch. Die Länge der Kraftvektoren wurde für bessere Erkennbarkeit verzehnfacht.

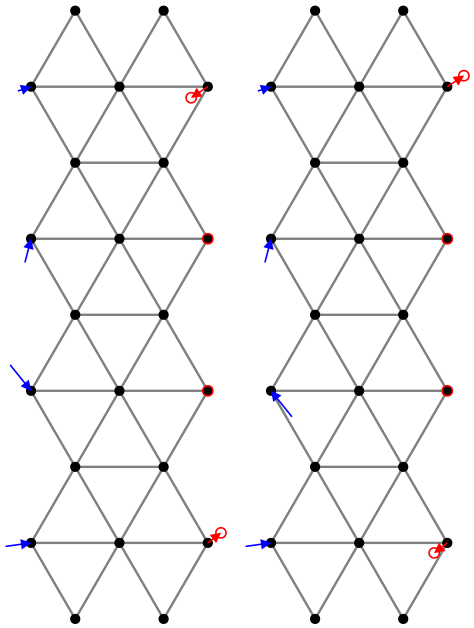


Abbildung 6: Zwei Verhaltensweisen, deren gemeinsames Antrainieren möglich sein könnte, bei uns jedoch nicht vorkommen könnten, da mindestens ein Neuron der ersten Schicht (z.b. erstes von unten) in beiden Verhaltensweisen den gleichen Kraftvektor hat. Somit beträgt der Winkel zwischen diesen 0° und ist entsprechend nie größer als der Mindestwinkel. Im Unterschied zu Abbildung 5 wurde jedoch der Kraftvektor eines Neurons (rechtes Netzwerk, zweites von unten) verändert. Die Länge der Kraftvektoren wurde für bessere Erkennbarkeit verzehnfacht.

Mithilfe von Multithreading werden außerdem mehrere Netzwerke gleichzeitig trainiert, teilweise auch mit gleichen Hyperparametern, da ein einzelner Versuch einer Hyperparameterkombination nicht sehr aussagekräftig wäre, v.a. wenn man die Verwendung von Zufallszahlen bei Erstellung von Verhaltensweisen und bestimmten Optimierungsalgorithmen bedenkt.

Da so eine Unterscheidung verschiedener Versuche auf Basis anderer Hyperparameter nicht möglich ist, wird jedem neuen Netzwerk eine UUID (*universally unique identifier*) zugeordnet und diese ebenfalls gespeichert. Andernfalls wäre zum Beispiel das Erstellen des MSE-Verlaufs eines einzelnen Netzwerks über Epochen hinweg nicht möglich.

Diese UUID wird weiter als Startwert für den Zufallszahlengenerator gesetzt, sodass die Versuche reproduzierbar sind.

Insgesamt werden in Compare.jl zum Vergleichen für jedes Netzwerk alle fünf Epochen folgende Daten gespeichert:

- aktuelle Zeit
- UUID
- Gesamtzahl an Trainingsepochen seit Erstellung
- Anzahl an Reihen und Spalten des Netzwerks
- Anzahl an Verhaltensweisen
- oben genannte Parameter für Erstellung der Verhaltensweisen (Mindestwinkel, Skalierungen)
- Art der Simulation (Eulerverfahren vs. numerisches Lösen)
- wie viele Sekunden die Simulation läuft
- MSE des Netzwerks

Bei den verschieden Testdurchläufen werden immer unterschiedliche Kombinationen von Anzahl der Reihen, Anzahl der Spalten, Anzahl der Verhaltensweisen und den unterschiedlichen Optimierungsalgorithmen festgelegt und der Loss des Netzwerks wird im Verhältnis zur Trainingszeit gespeichert. Mithilfe dieser Daten können nun statistische Analysen durchgeführt werden, um ein besseres Verständnis über die Auswirkungen dieser Eigenschaften zu erlangen.

3.2.1 Backpropagation

Backpropagation ist ein zentrales Konzept in der Funktionsweise von ANNs. Es handelt sich dabei um einen iterativen Optimierungsalgorithmus, der verwendet wird, um die Gewichtungen der Verbindungen zwischen den Neuronen im Netzwerk anzupassen und somit die Leistung des Netzwerks zu verbessern [1]. Da wir uns in unseren vergangenen Jugend forscht Projekten schon mit ANNs auseinandergesetzt haben, ist uns die Idee gekommen, diesen Algorithmus auch für MNNs umzusetzten.

Der Prozess beginnt mit der Vorwärtspropagation, bei der Eingabedaten durch das Netzwerk fließen und eine Ausgabe erzeugt wird. Der erzeugte Ausgabe-Fehler wird dann durch den Vergleich mit den gewünschten Ausgabewerten berechnet. Im nächsten Schritt wird der Fehler rückwärts durch das Netzwerk propagiert, um die Beiträge jedes Neurons zur Fehlerentstehung zu quantifizieren. Die Gewichtungen werden entsprechend des Fehlers angepasst, um diesen zu minimieren.

Um diesen Algorithmus für MNNs zu nutzen, benötigt man zum einen eine Möglichkeit, den Fehler durch das Netzwerk zu propagieren, und zum anderen ein Verfahren, das die Federkonstanten anpasst, um den Fehler jedes einzelnen Neurons zu minimieren. Der Fehler der Neuronen in der letzten Schicht kann einfach mit den Trainingsdaten bestimmt werden. Die Fehler in den Positionen aller anderen Neuronen sind dann der Durchschnitt der Fehler aller Neuronen, mit denen sie verbunden sind. Um herauszufinden, wie wir die Federkonstante einer Feder ändern müssen, damit die Position des Zielneurons einen kleineren Fehler bekommt, muss man erst einmal bestimmen, was der Winkel zwischen dem Fehlervektor und dem Vektor zwischen den beiden Neuronen der Federn ist. Wenn dieser zum Beispiel 90° beträgt, ist eine Änderung der Federkonstante nicht notwendig, da die Kraft den Fehler aufgrund ihrer Richtung nicht beeinflussen kann. Wenn dieser Winkel 0° beträgt, wollen wir die Federkonstante stark erhöhen, da die Kraft der Feder genau mit der Richtung des Fehlers übereinstimmt und wir das Neuron deswegen stärker entlang dieser Richtung bewegen wollen, was durch mehr Kraft erfolgt, wofür wir eine größere Federkonstante benötigen. Wenn der Winkel 180° beträgt, wollen wir die Federkonstante senken, da die Kraft genau in die falsche Richtung ausgeübt wird.

Allgemein wird die Federkonstante also um den folgenden Term erhöht:

$$\frac{\vec{v_1} \cdot \vec{v_2}}{\|\vec{v_1}\| * \|\vec{v_2}\|} * \epsilon$$

wobei v_1 der Fehler eines Neurons, v_2 die Differenz der Positionen der beiden Neuronen einer Feder und ϵ die Lernrate ist.

Leider ist es uns nicht gelungen, ein MNN mithilfe von Backpropagation zu trainieren, da der MSE nicht signifikant gesunken ist. Dies könnte zum einen daran liegen, dass Backpropagation voraussetzt, dass die Daten (oder Kräfte) nur in eine Richtung weitergegeben werden können, was bei MNNs nicht der Fall ist.

4 Ergebnisse

Wie man in Abb. 7 und Abb. 9 sehen kann, kann man ein MNN mithilfe von PPS erfolgreich trainieren. Dabei fällt der MSE immer langsamer und kann nur noch durch viel Rechenzeit signifikant weiter gesenkt werden. Diese 500 Epochen, welche mit 14 verschiedenen Konstellationen von jeweils 3 verschiedenen Verhaltensweisen trainiert wurden, dauerten auf unserer Hardware ungefähr 14 Minuten. Das heißt, dass wir pro Verhaltensweise und Epoche ungefähr 0,04 Sekunden benötigen.

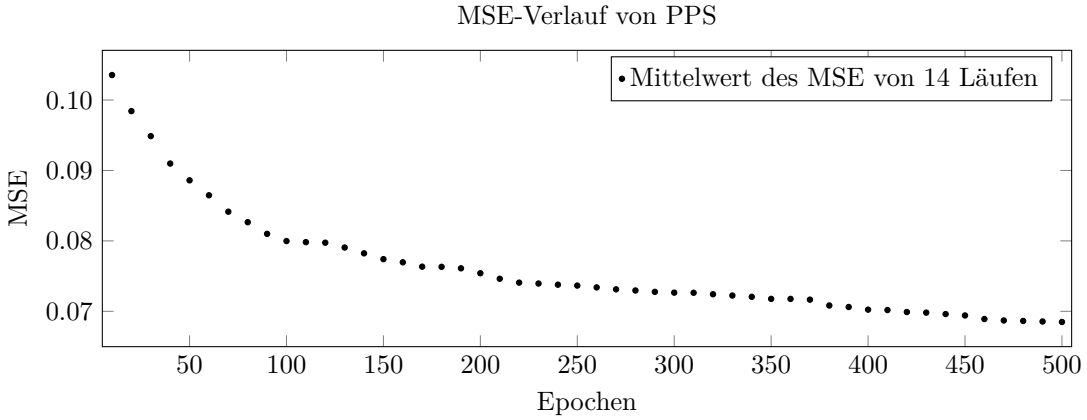


Abbildung 7: Jeder Lauf entspricht einem einzigartigen Netzwerk (eigene UUID). Das Netzwerk wurde mit PPS trainiert, zur Simulation wurde Autodifferenzierung genutzt. Originaldaten sind im Repository [5] in der Datei `src/data/PPSNumBehaviours_2024-01-14T13:56:06.441.csv` zu finden. Der Mittelwert des MSE vor dem Training (0 Epochen) ist $\approx 0{,}968$ und wurde für bessere y-Achsenskalierung entfernt.

Das Training mit dem genetischen Algorithmus ist im Gegensatz zu PPS nicht erfolgreich gewesen. Nach 200 Epochen lag der minimale MSE bei ungefähr 0,55 (s. Abb. 8). Dort befand er sich jedoch auch schon nach 80 Epochen, der MSE des Netzwerk befindet sich also in einem lokalem Minimum. Mit der aktuellen hohen Lernrate von 0,05 werden keine feineren Anpassungen an den Federkonstanten mehr vorgenommen und ein signifikant niedriger MSE kann nicht erreicht werden. Würde jedoch eine kleinere Lernrate gewählt werden, würden es sehr lange dauern, bis überhaupt erst dieser MSE von 0,55 erreicht wird.

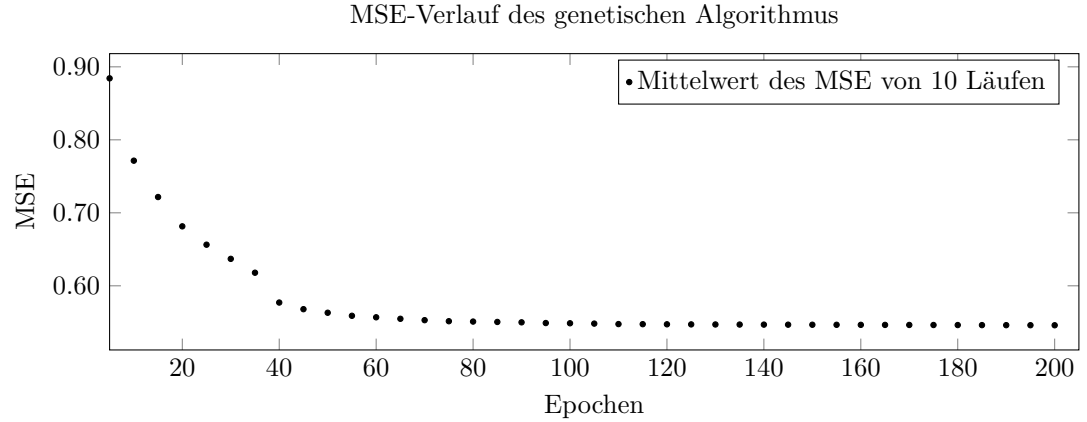


Abbildung 8: Das Netzwerk wurde mit dem evolutionären Algorithmus trainiert, zur Simulation wurde Autodifferenzierung verwendet, die restlichen Hyperparameter aller Netzwerke sind identisch zu den für Abb. 7 verwendeten. Originaldaten sind in der Datei `src/data/EvolutionEpochs_2024-01-14T20:57:11.627.csv` zu finden. Der Mittelwert des MSE vor dem Training (0 Epochen) ist $\approx 0{,}968$ und wurde für bessere y-Achsenskalierung entfernt.

Insgesamt lässt sich also sagen, dass in unserer Implementation PPS deutlich besser abschneidet als der evolutionäre Algorithmus, sowohl was Laufzeit als auch bisher erreichbare MSEs angeht.

Der einzige Vergleich von genetischer Optimierung und PPS, den wir finden konnten, war von Lee *et al.*

(2022), deren Ergebnisse sich von unseren unterscheiden: Sie konnten mit dem genetischen Algorithmus einen deutlich niedrigeren MSE erreichen als mit PPS. Jedoch haben sie ersteren für mehr als 100 Stunden trainiert, und den zweiten weniger als drei Stunden [3, Abb. 4]; wie bei uns auch braucht der genetische Algorithmus deutlich länger, um die gleiche Leistung wie PPS zu erreichen.

Die Laufzeit scheint in der Tat bei PPS besser zu sein. Für eine Aussage über den niedrigsten erreichbaren MSE müssten jedoch das Problem des lokalen Minimums für den genetischen Algorithmus gelöst und beide sehr lange laufen gelassen werden.

(a) Ein MNN, welches aus 5 Spalten und 3 Reihen besteht.

(b) Ein MNN, welches aus 15 Spalten und 9 Reihen besteht.

Abbildung 9: Die beiden MNNs wurden jeweils mit einer Verhaltensweise mit PPS trainiert. Die Kraftpfeile sind blau eingezeichnet und die roten Kreise geben an, wo sich die Ausgabeneuronen befinden sollen. Man erkennt, dass die Netzwerke erfolgreich trainiert wurden, da sich die Neuronen in / nahe den roten Kreisen befinden.

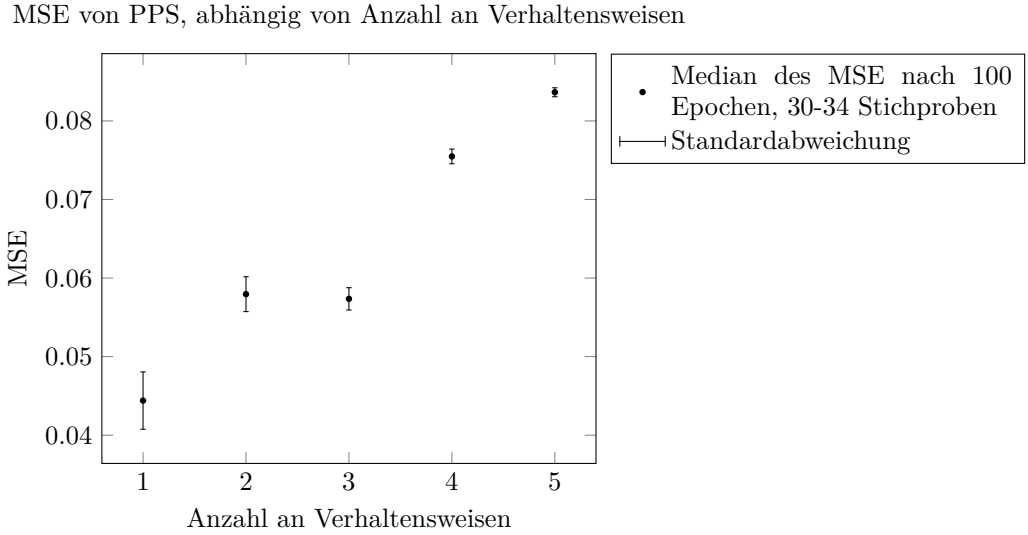


Abbildung 10: Zur Simulation wurde Autodifferenzierung benutzt. Die Hyperparameter sind abgesehen von der Anzahl der Verhaltensweise immer identisch. Originaldaten sind im Repository [5] im Ordner `src/data/`: `PPSNumBehaviours_2024-01-14T15:16:06.598.csv`, `PPSNumBehaviours_2024-01-14T15:08:40.927.csv` und `PPSNumBehaviours_2024-01-14T13:56:06.441.csv`

test

Lee *et al.* (2022) analysierten in ihrer Simulation unter anderem den Zusammenhang von MSE und Anzahl an Verhaltensweisen. Es ist bei ihren Ergebnissen zu erkennen, dass der MSE tendenziell stark steigt, je

MSE des genetischen Algorithmus, abhängig von Anzahl an Verhaltensweisen

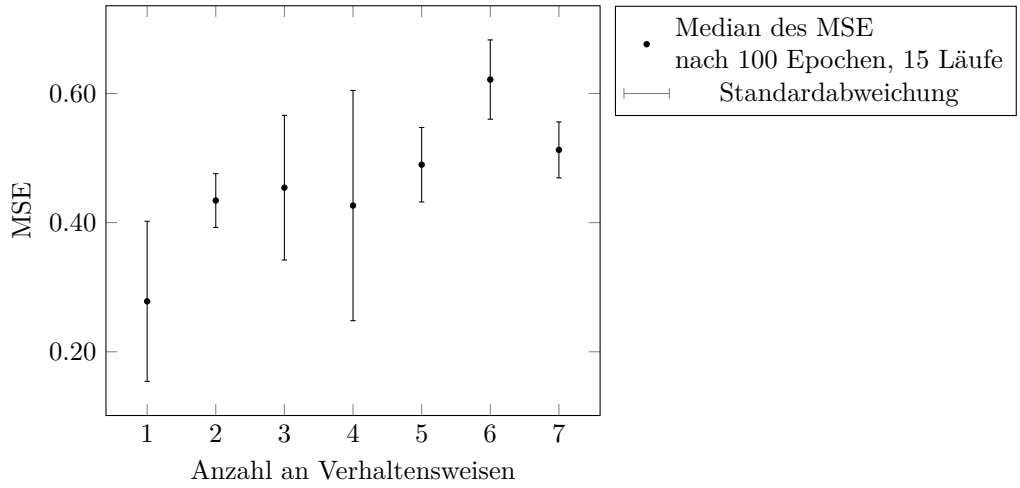


Abbildung 11: Das Netzwerk wurde 100 Epochen mit dem genetischen Algorithmus trainiert, die anderen Hyperparameter sind identisch zu Abb. 10. Originaldaten sind unter `src/data/EvolutionNumBehaviours_2024-01-15T16:03:55.251.csv` zu finden.

mehr Verhaltensweisen gleichzeitig trainiert werden [3, Abb. 5A u. 5C]. Wir konnten dieses Verhalten sowohl bei PPS als auch der evol. Opt. bestätigen (s. Abb. 10 und 11)

Weiter konnten wir den aus [3, Abb. 5B] ersichtlichen Zusammenhang bestätigen, zumindest beim Training mit PPS: Je mehr Reihen das Netzwerk hat, desto höher der MSE, und je mehr Spalten das Netzwerk hat, desto niedriger der MSE (vgl. Abb. 12). Sowohl eine erhöhte Anzahl an Reihen als auch an Spalten erhöht die Menge an Federn und somit die notwendigen Schritte zur erfolgreichen Optimierung. Jedoch senkt eine höhere Anzahl an Spalten den MSE vermutlich dadurch, dass sie entlang der Kraftrichtung von links nach rechts liegen und so auch mehr Möglichkeiten zur Veränderung und Optimierung dieser bieten, während Reihen senkrecht zu dieser Richtung sind und ihre Anzahl zusätzlich proportional zur Anzahl an Kraftvektoren und Zielpositionen ist.

MSE von PPS abhängig von Anzahl an Spalten und Reihen

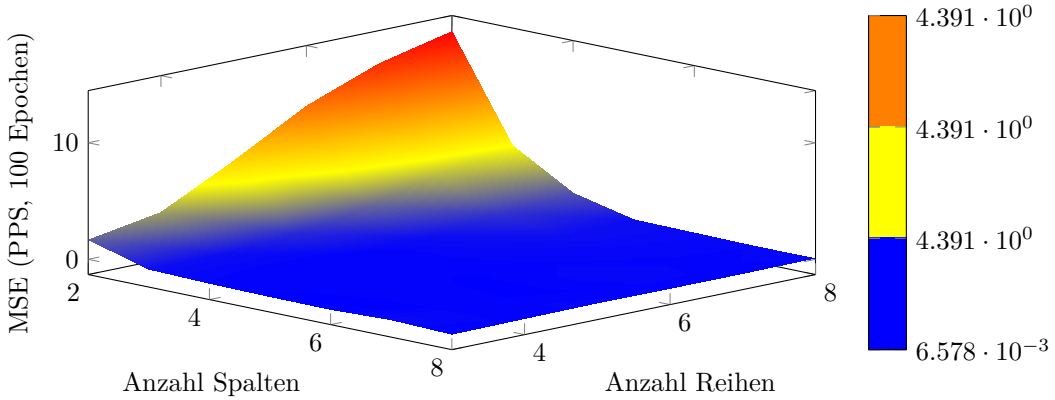


Abbildung 12: Niedrige MSE-Werte sind blau, hohe rot. Pro Kombination gab es nur ein Netzwerk. Datei: `src/data/PPSNumRowsColumns_2024-01-06T13:23:22.688.csv`

Unsere Annahme aus der Einleitung, dass wir auch durch unsere Simulation relevant Ergebnisse erzeugen können, scheint sich also zu bestätigen. Denn wir kommen zu ähnlichen Schlüssen wie Lee *et al.* (2022), die ihren Simulationsalgorithmus und -ergebnisse anhand eines physischen Aufbaus bestätigen konnten.

4.1 Framework

Die von uns geschriebene Bibliothek [5] soll nicht nur von uns leicht bedient und erweitert werden, sondern auch von anderen. Hierzu fehlen zwar noch Dokumentation und Anleitungen, doch der Programmcode ist durch die Aufteilung der einzelnen Aufgabenbereiche und Aspekte der Bibliothek auf verschiedene Verbünde / Strukturen (s. Abb. 13) bereits entsprechend gestaltet (s. Prog. (Programmausschnitt) 1). Denn diese bieten gemeinsam mit den Möglichkeiten der Programmiersprache Julia, z.B. der Funktionsüberladung die Option, einfach eigene Aspekte hinzuzufügen, ohne den Rest selbst schreiben zu müssen (s. Prog. 2).

```
using MNN # lade Paket
net = Network(5,4) # erstelle Netzwerk mit 5 Spalten und 4 Reihen
t = Trainer(net, PPS(), Diff(100), 2) # PPS für Opt., Autodiff. für Sim., 2 Verhaltensw.
train!(net, 100, t) # Training für 100 Epochen
reset!(net) # Positionen der Neuronen zurücksetzen
# Netz + Verhaltensweise anzeigen; wird während Sim. autom. aktualisiert
vis = Visualizer(net, behaviour=t.behaviours[1])
# 100s mit 1. Verhaltensweise simulieren
simulate!(net, Diff(100), t1.behaviours[1], vis = vis)
```

Programmausschnitt 1: Beispielprogramm, welches MNN.jl verwendet

```
using MNN
mutable struct MySim <: MNN.Simulation # MySim als Untertyp von MNN.Simulation
    modifier::Function # wird später autom. von MNN.jl gesetzt
    # ...
end
MySim(...) = MySim((net, acc) -> nothing), ...)

function MNN.simulate!(network::Network, sim::MySim; vis::Union{Visualizer,Nothing}=nothing)
    # ...
    sim.modifier(network, acc) # setze Beschl. entspr. Kraftvektor
    MNN.update_positions!(vis, network) # aktualisiere Visualisierung
    # ...
end
# nutze MNN.jl wie üblich (s. Programmausschnitt 1)
t = Trainer(net, PPS(), MySim(...), 1)
simulate!(net, MySim(...), t.behaviours[1], vis = vis)
```

Programmausschnitt 2: Anpassung von MNN.jl durch Hinzufügen einer eigenen Simulation, für Trainingsverfahren wäre das Vorgehen ähnlich. Das Simulationsverfahren kann ohne Veränderung des Quellcodes des Pakets hinzugefügt und nahtlos mit dem Rest des Pakets verwendet werden. Für Training, Visualisierung, MSE-Berechnung, Verhaltensweisenerstellung etc. können einfach die existierenden Funktionen von MNN.jl weiter verwendet werden.

Zudem ist die von uns entwickelte Bibliothek die erste Open Source-Implementation von MNNs und wir stellen auch die ersten öffentlichen Daten von den Trainingsverläufen für weitere Analysen zur Verfügung [5].

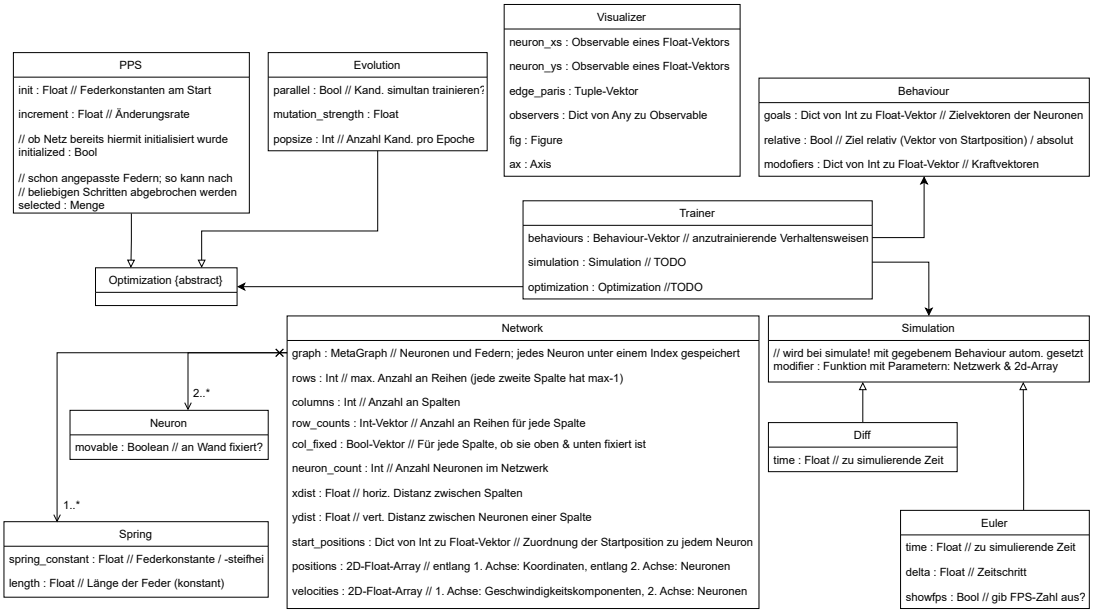


Abbildung 13: UML-Diagramm des Programm-Aufbaus. Da es sich hier um Verbünde und nicht Klassen handelt, sind die Methoden nicht mit aufgelistet.

5 Diskussion

Unser neuer Optimierungsansatz, Backpropagation, war zwar nicht erfolgreich, jedoch konnten wir erfolgreich die Leistung der zwei tatsächlich verwendeten Optimierungsalgorithmen beurteilen, auch in Abhängigkeit einiger Parameter – sie stimmen mit dem aktuellen Forschungsstand überein.

Vor allem unsere Analyse der Abhängigkeit von Spaltenanzahl, Reihenanzahl und Anzahl an Verhaltensweisen ist interessant, da diese für MNNs bisher nur in Simulationen getestet wurden. Jedoch wurde dafür z.B. bei Lee *et al.* (2022) immer Gradient Descent verwendet [4, S. 6], da dies schneller ist. Wir haben diese Tests nun direkt mit den zu untersuchenden Verfahren durchgeführt, und konnten so verfahrensbedingte Unterschiede ausschließen. Ausnahme bildet die Analyse der Auswirkungen der Spalten- und Reihenzahl, welche aufgrund der hohen Rechenzeit bisher nur für PPS durchgeführt wurde. Den Test mit genetischer Optimierung wollen wir als nächstes durchführen, denn er könnte weniger unter einer erhöhten Reihenanzahl leiden, da anders als bei PPS „irrelevante“ Federn die Trainingszeit vermutlich nicht erhöhen würden.

Weiter wollen wir das Problem des lokalen Minimums des genetischen Algorithmus lösen, um neben der Laufzeit nun auch bestimmen zu können, welche den niedrigsten MSE erreichen kann, unabhängig von Laufzeit. Dies wäre vor allem interessant, da Lee *et al.* (2022) durch die stark unterschiedliche Trainingszeiten der beiden Algorithmen (über 100 Stunden vs. unter 3 Stunden) noch keine ausreichenden Daten für diesen Vergleich liefern. Zum Lösen des Problems wollen wir die Lernrate des genetischen Algorithmus adaptiv machen, sodass sie zu Beginn des Training hoch ist, um den MSE schnell zu senken, und dann immer weiter sinkt, um lokalen Minima zu entkommen.

Da wir uns momentan auf den stabilen Endzustand der MNNs beschränken, könnte man die Differenzialgleichungen stark vereinfachen, indem man nicht die Beschleunigungen, sondern die Geschwindigkeiten der Neuronen mit den auf sie einwirkenden Kräften gleichsetzt. Dadurch würde man Schwingungen in dem Netzwerk verhindern, da sich die Neuronen einfach direkt dem Ruhezustand annähern würden, wodurch die Lösung deutlich einfacher zu beschreiben ist und höchstwahrscheinlich auch schneller berechnet werden kann. Wenn man bedenkt, dass wir beim Trainieren der Netzwerke fast die ganze Zeit mit der Simulation der MNNs verbringen, könnte diese Veränderung eine große Verbesserung der Performance ausmachen und so mehr Vergleiche ermöglichen, vor allem rechenzeitintensive wie der im vorletzten Absatz erwähnte.

Es gibt es noch weitere Hyperparameter, die wir testen möchten sowie teilweise noch den CSV-Dateien als Spalten hinzufügen müssen, wie z.B. die Trainingsparameter für die Optimierungsverfahren (Startwert und Änderungsrate für PPS, *population size* und Lernrate für genetische Optimierung).

Weiter wollen wir in der Zukunft Kombinationen an Optimierungsverfahren analysieren, vor allem die Anwendung von erst PPS und dann genetischen Algorithmen. Denn so könnte man die Geschwindigkeit von PPS nutzen, um den MSE schnell zu senken, und nach Erreichen eines lokalen Minimums einen genetische Algorithmus, um einen niedrigeren MSE erreichen zu können – vorausgesetzt, wir können bestätigen, dass letztere niedrigere MSEs als PPS erreichen können.

Zusätzlich zu der bereits getesteten Architektur gibt es noch viele andere interessante Möglichkeiten, ein MNN aufzubauen. Wenn man zum Beispiel versucht, erdbebensichere Häuser und einsturzsichere Brücken durch MNNs umzusetzen, müsste man das Resonanzverhalten analysieren und untersuchen, inwiefern ein MNN auf Schwingungen reagiert. Vielleicht ist es sogar möglich, dem System eine genaue Resonanzkurve, die erlernt werden soll, als Trainingsdaten zu geben. Dies könnte man umsetzen, indem man den Betrag der auf das MNN einwirkenden Kraft nicht konstant gleich lässt, sondern als Sinusfunktion der Zeit angibt. Dadurch könnte man schwingende Kräfte simulieren, welche auch Schwingungen im Netzwerk auslösen würden. Zudem müsste man auch die Verhaltensweisen ändern, da sie zusätzlich zu der Richtung der einwirkenden Kräfte auch noch die Frequenz der Schwingung einbeziehen müssten und zudem nicht die Positionen der Neuronen als zu lernendes Kriterium für die Verlustfunktion nutzen könnten. Vielmehr müsste sich der Verlust darauf beziehen, wie stark das System, je nach Frequenz der Schwingungen der Kraft, selber anfängt zu schwingen. Zum Beispiel wäre es vorstellbar, so einen Resonanzverstärker für Musikinstrumente zu bauen, welcher nur auf die Frequenzen reagiert, die zu den Tönen der Tonleiter gehören. Aber auch Flugzeugflügel könnten als MNN gebaut werden, um auf die Turbulenzen der sie umströmenden Luft passsend zu reagieren.

Weiter sollten unsere Programme ebenso mit einem dreidimensionalen MNN funktionieren. Die Implementation wäre nicht sehr aufwändig, da keine Änderung der Formeln und nur leichte Programmänderungen notwendig wären. Durch die höhere Komplexität könnten mehr und komplexere Verhaltensweisen gleichzeitig erlernt werden, es würde aber der Trainingsaufwand voraussichtlich sehr stark steigen. Bisher wurden dreidimensionale MNNs trotz der Vorteile (neue Anwendungszwecke, komplexere Verhaltensweisen) weder gebaut noch simuliert. Wir planen, es zu probieren, sobald wir mit allen Optimierungsverfahren erfolgreich zweidimensionale Netzwerke trainieren können und ihre relevanten Hyperparameter analysiert haben.

Zusammenfassend stellen MNNs eine faszinierende Möglichkeit dar, Erkenntnisse der modernen KI mit Materialforschung zu verbinden. Dieses Forschungsfeld ist gerade erst im Entstehungsprozess. Wir sehen hier eine großes Zukunftspotenzial und wollen mit unserem Projekt einen Beitrag dazu leisten.

6 Quellen

[1] Brotcrunsher. *Neuronale Netze - Backpropagation - Backwardpass*. YouTube Video. 2017. URL: https://www.youtube.com/watch?v=EAtQCut6Qno (besucht am 15. 01. 2022).

[2] Jonathan B Hopkins, Ryan H Lee und Pietro Sainaghi. „Using binary-stiffness beams within mechanical neural-network metamaterials to learn“. In: *Smart Materials and Structures* 32.3 (Feb. 2023), S. 035015. DOI: 10.1088/1361-665X/acb519. URL: https://dx.doi.org/10.1088/1361-665X/acb519.

[3] Ryan H. Lee, Erwin A. B. Mulder und Jonathan B. Hopkins. „Mechanical neural networks: Architected materials that learn behaviors“. In: *Science Robotics* 7.71 (2022), eabq7278. DOI: 10.1126/scirobotics.abq7278. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abq7278. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abq7278 (besucht am 28. 11. 2023).

[4] Ryan H. Lee, Erwin A. B. Mulder und Jonathan B. Hopkins. „Mechanical neural networks: Architected materials that learn behaviors“. Supplementary Materials. In: *Science Robotics* 7.71 (2022), eabq7278. DOI: 10.1126/scirobotics.abq7278. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abq7278. URL: https://www.science.org/doi/suppl/10.1126/scirobotics.abq7278/suppl_file/scirobotics.abq7278_sm.pdf (besucht am 28. 11. 2023).

[5] Alexander Reimer und Matteo Friedrich. *Simulation of MNNs*. GitHub Repository. 2023. URL: https://github.com/Alexander-Reimer/Simulation-of-MNNs (besucht am 27. 11. 2023).

[6] Benedikt Schröter. „Genetische Algorithmen - Optimierung nach dem Ansatz der natürlichen Selektion“. In: (März 2020). URL: https://www.cologne-intelligence.de/blog/genetische-algorithmen.

[7] Wikipedia. *Hookesches Gesetz — Wikipedia, die freie Enzyklopädie*. 2023. URL: %5Curl%7Bhttps://de.wikipedia.org/w/index.php?title=Hookesches_Gesetz&oldid=238038175%7D (besucht am 06. 01. 2024).