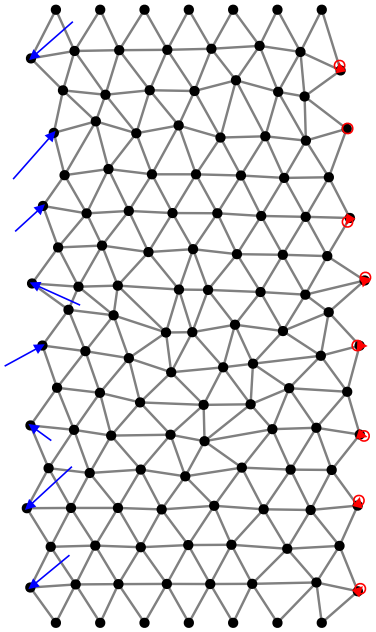
Analyse der Optimierungsverfahren mechanischer neuronaler Netzwerke



Alexander Reimer Matteo Friedrich

Gymnasium Eversten Oldenburg

Betreuer: Herr Dr. Glade

Inhaltsverzeichnis

Abbildungsverzeichnis

Programmausschnitte

Tabellenverzeichnis

Zusammenfassung

Wir haben uns mit dem neuen, vergleichsweise noch wenig erforschten Bereich der *mechanical neural networks*, kurz MNNs, beschäftigt. MNNs sind programmierbare Materialien, welche verschiedene Verhaltensweisen, wie z.B. ein bestimmtes Verformungsverhalten oder ein bestimmtes Resonanzverhalten, lernen können. Somit könnten sie verwendet werden für aerodynamischere Flügel, erdbebensichere Gebäude, ausfallsichere Materialien und vieles mehr.

MNNs bestehen aus Massenpunkten (genannt Neuronen), welche durch Federn miteinander verbunden werden. Ihr Verhalten ergibt sich durch die Federkonstanten (Federhärten) 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 die Simulation von MNNs umgesetzt, bisher verwendete Algorithmen (evolutionäres Lernen und *pattern search*) selbst implementiert, sowie mit neuen Parametern ausprobiert und verglichen. Trotz der Beschränkung auf die Anwendung dieser Algorithmen in Simulationen sollten unsere Analyseergebnisse einen guten Startpunkt für reale MNNs bieten. Der von uns entwickelte Code ist die erste öffentlich verfügbare Implementation eines MNNs und wurde mit dem Ziel entwickelt, möglichst erweiterbar und nutzerfreundlich zu sein.

Zuerst haben wir mit der Analyse des Verformungsverhaltens von MNNs angefangen. Dabei konnten wir mehrere Zusammenhänge zwischen verschiedenen Parametern und dem Erfolg beim Erlernen von Verformungsverhaltensweisen finden, wie z.B. abnehmender Lernerfolg bei zunehmender Anzahl an Reihen des MNN, zunehmender Anzahl an Verhaltensweisen oder größeren Krafteinwirkungen. Diese Ergebnisse decken sich mit Erkenntnissen der bisherigen Literatur, was die Aussagekraft unserer Simulation verdeutlicht. Daraufhin haben wir – soweit wir wissen als die Ersten – das Resonanzverhalten eines MNNs erfolgreich optimiert und eine erste Analyse durchgeführt, mit der Erkenntnis, dass eine höhere Differenz zwischen höchster Zielamplitude und tiefster Zielamplitude vermutlich zu geringerem Lernerfolg führt.

Unsere Ergebnisse zeigen insgesamt: MNNs können mehrere komplexe Verhaltensweisen lernen und diese intelligenten Materialien eröffnen so vielfältige zukünftige technologische Anwendungsmöglichkeiten.

1 Einleitung

Inspiration für dieses Projekt ist ein Artikel von Lee *et al.* (2022) mit dem Titel „Mechanical neural networks: Architected materials that learn behaviors“ [4]. Es ist die erste uns bekannte Veröffentlichung, die 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 Härte 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.

Durch ihre Eigenschaften könnten MNNs viele Verwendungszwecke haben, von der Optimierung von Flugzeugflügeln abhängig von aktuellen Gegebenheiten wie Windstärke und -winkel [4, S. 2] über Optimierung der Schockabsorption von Schutzwesten bis zur dynamischen Änderung der Resonanzfrequenz eines Gebäudes zum Schutz gegen Erdbeben [3, S. 9]. Das Training der Resonanz eines MNNs könnte auch für einbruchsichere Brücken oder vielleicht sogar bessere Musikinstrumente eingesetzt werden.

In diesem Projekt wollen wir verschiedene Optimierungsalgorithmen für MNNs vergleichen. Da es noch

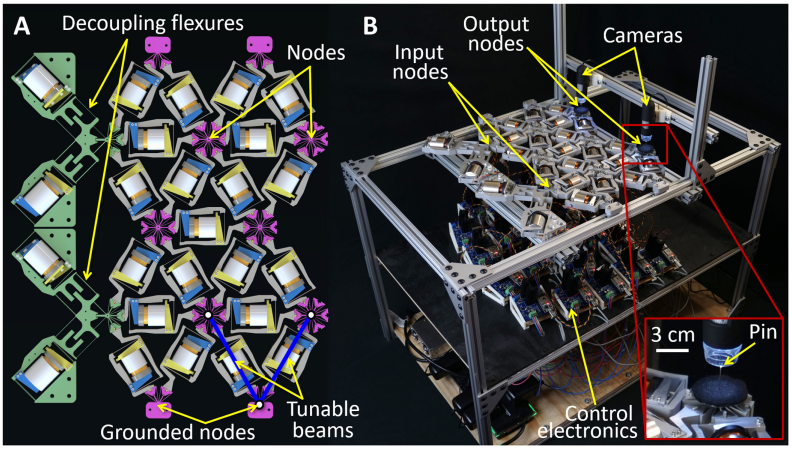


Abbildung 1: Mechanische Umsetzung eines MNNs (Abbildung von [4]).

kein Framework gibt, mit welchem dieser Vergleich einfach durchzuführen ist, 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. Dabei handelt es sich unserem Wissen nach um die erste jemals veröffentlichte Softwarebibliothek zu diesem Thema.

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 MNNs mit einer Simulation die Gesamtzeit des Trainierens verkürzen kann, 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, unterscheiden sich in ihrer Funktionsweise und Umsetzung jedoch stark.

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. [4, Abb. 5]). Wir nennen die Knotenpunkte zwischen Federn sowie Befestigungspunkte an den „Wänden“ äquivalent zu ANNs Neuronen. Dabei sollte noch einmal klargestellt 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 bisher stellt die technische Umsetzung aufgrund der Größe der Federn und der benötigten Steuerelektronik ebenfalls Schwierigkeiten dar. Doch sie bieten als analoges System einen großen Vorteil: Während ANNs für die Evaluierung von Eingaben Zeit und Ressourcen benötigen, funktionieren trainierte MNNs ohne signifikante Verzögerung.

Gegenüber „konventionellen“ Materialien bieten sie drei hauptsächliche Vorteile: Erstens können mehrere Verhaltensweisen auf externe Kräfte gleichzeitig antrainiert werden, zweitens können MNNs durch Sensoren auch beim Einsatz weiterlernen [4], sodass sie sich von selbst an Beschädigung, Abnutzungen, Größenänderung

usw. anpassen könnten, und drittens könnten MNNs teilweise die teure Forschung nach Materialien mit gewünschten Eigenschaften ersetzen.

Aktuell sind MNNs bereits technisch umsetzbar, wie Lee *et al.* (2022) gezeigt haben. Aufgrund ihrer Größe sind sie zwar noch nicht für Anwendungsfälle wie Schutzwesten nutzbar, jedoch ist ein Einsatz für z.B. Brücken bereits denkbar. Neben der Größe ist momentan auch die Komplexität und der Energiebedarf der MNNs ein Problem: Für negative Federkonstanten ist eine aktive Energiezufuhr zu den von Lee und Mitarbeitern aus Elektromagneten konstruierten Federn sowie eine entsprechende Kontrollelektronik notwendig [4]. Eine mögliche Lösung für beide Probleme könnten *binary stiffness beams* bieten – kompliente Systeme, welche mit einmaliger Kraftzufuhr jeweils zwischen einer möglichst hohen und einer möglichst geringen Federkonstante geschaltet werden können [3]. *Binary stiffness beams* können jedoch keine negativen Federkonstanten abdecken, welche für viele komplexe Verformungsverhalten (z.B. Zusammenziehen des MNNs von beiden Seiten bei Krafteinwirkung auf linker Seite) notwendig sind.

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 Hookesche Gesetz $F = -k \cdot x$ genutzt, welches die Kraft F als Produkt von Federkonstante k und Auslenkung x angibt [12].

MNNs, welche nur positive Federkonstanten besitzen, können 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. Um auch eine Verhaltensweise, welche Bewegung entgegen einer Kraft benötigt, umzusetzen, muss dem System Energie zugefügt werden. Dies lässt sich mit negativen Federkonstanten umsetzen. Federn mit negativen Federkonstanten stoßen weit voneinander entfernte Neuronen ab und ziehen nahe Neuronen noch weiter an. Jedoch sind MNNs mit negativen Federkonstanten bei Verwendung des Hookeschen Gesetzes nicht stabil, da es zum Beispiel passieren kann, dass sich zwei durch eine Feder mit negativer Federkonstante verbundene Neuronen voneinander entfernen, wodurch sie sich abstoßen und somit noch weiter voneinander entfernen würden. 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. Wir haben deshalb für die Kraft anstelle des Hookeschen Gesetzes eine Funktion dritten Grades gewählt: $F(x,k) = -k \cdot x - x^3$ (s. Abb. 2).

Zur Analyse des Verhaltens des MNNs unter verschiedenen Krafteinflüssen muss die Position des Neurons als Funktion der Zeit angegeben und dafür die obige Formel mithilfe des zweiten newtonschen Gesetzes zu einer Differenzialgleichung für die Beschleunigung des Neurons ($\ddot{x}$) umgewandelt werden. Im einfachsten Fall, für ein einzelnes bewegliches Neuron und eine Feder, ergibt sich die Gleichung $m \cdot \ddot{x} = -k \cdot x - x^3$, wobei die Masse m bei uns 1 beträgt und so weggelassen werden kann.

Um den Ruhezustand des Systems bei bestimmten Krafteinflüssen analysieren zu können, muss außerdem eine Dämpfung ergänzt werden. Denn ohne eine Dämpfung würde das Netzwerk einfach immer weiter schwingen und nie einen Ruhezustand erreichen (s. Abb. 3, blauer Graph). 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. Je größer der Wert für γ gewählt wird, desto stärker ist das MNN gedämpft und desto schneller erreicht es demnach auch ein Kräftegleichgewicht (s. Abb. 3).

Da wir die MNNs jedoch im 2-dimensionalen Raum simulieren wollen, müssen wir die Beschleunigung auch

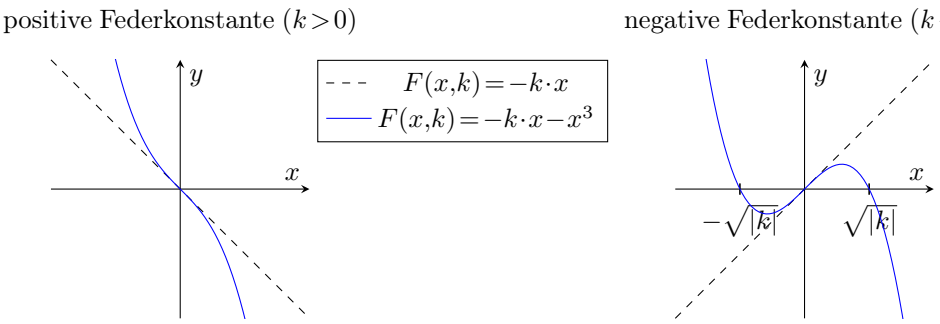


Abbildung 2: Skizze der verwendeten Federkraft $F(x,k)$ in Abhängigkeit der Auslenkung x für den Fall einer positiven Federkonstante $k>0$ (links) und einer negativen Federkonstante $k<0$ (rechts). Die gestrichelte Linie zeigt einen linearen Kraftverlauf $F(x,k)=-kx$. Im Falle einer positiven Federkonstante (links) entspricht dies dem Hookeschen 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 $F(x,k)=-kx-x^3$ (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.

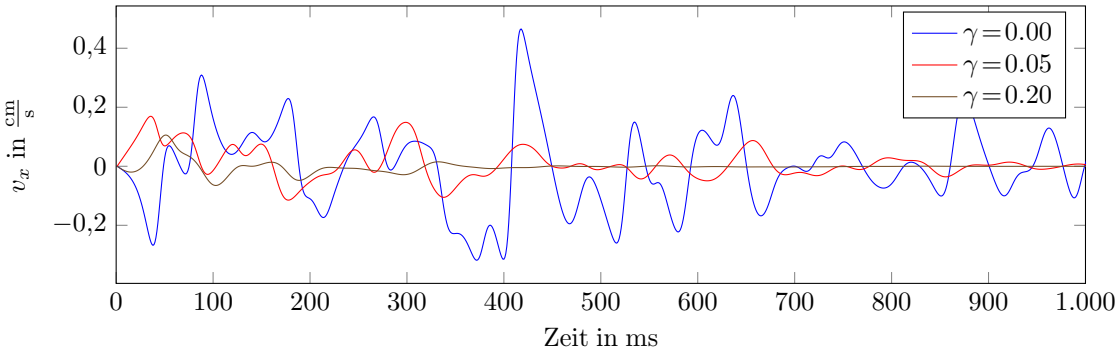


Abbildung 3: Geschwindigkeit eines zufällig gewählten Neurons entlang der x-Richtung (v_x) bei unterschiedlichen Werten für die Dämpfung γ. Man sieht, dass sich die Einschwingzeit mit größeren Werten für γ schnell verkleinert. Die Daten wurden in einem System mit 22 Neuronen und zufälligen Federkonstanten aufgenommen.

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:

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

Hier ist $\Delta\vec{x}_{ij}=\vec{x}_i-\vec{x}_j$, also der Abstandsvektor zwischen Neuron i und Neuron j, $\Delta x_{ij}=\|\Delta\vec{x}_{ij}\|$, 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. Durch Testläufe in einfachen geometrischen Konfigurationen mit wenigen (1-3) Neuronen haben wir uns davon überzeugt, dass die erzeugten Simulationen physikalisch plausibel sind, was z.B. die Möglichkeit von Vorzeichenfehlern ausschließt. Für eine größere Zahl an Neuronen entstehen dabei komplizierte Dynamiken.

2.2 Resonanzanalyse von MNNs

Die Resonanz eines Objektes gibt an, wie es auf externe Schwingungen reagiert. Sie lässt sich mit einer Resonanzkurve beschreiben, welche die Amplitude für verschiedene Erregerfrequenzen angibt. Damit eine Resonanzkurve ermittelt werden kann, muss von einer konstanten externen Kraft zu einer von der

Zeit abhängigen externen Kraft abgewichen werden. Dabei schwingt die Kraft wie eine Sinuskurve mit unterschiedlichen Frequenzen, wodurch eine Schwingung in der Position der Neuronen erzwungen wird. Nun muss die Amplitude der Schwingungen der Positionen der Neuronen nur noch in Abhängigkeit der Frequenz der Schwingungen der Kraft betrachtet werden (s. Abb. 10).

Während der Erarbeitung unseres Projekts ist uns aufgefallen, dass unser MNN – zumindest im einfachsten Fall mit nur einer einzigen Feder – einem Duffing-Oszillator entspricht, da dieser auch ein Federpendel mit einem kubischen Verhältnis von Kraft und Auslenkung, welches einer schwingenden Kraft ausgesetzt ist, simuliert [13]. Bei diesem Duffing-Oszillator handelt es sich um ein chaotisches System, welches komplexe Verhaltensweisen aufzeigt. Eine charakteristische Eigenschaft des Duffing-Oszillators ist zum Beispiel seine nicht eindeutige Zuordnung von Erregerfrequenz und Schwingungsamplitude in der Resonanzkurve. Während bei linearen Oszillatoren die Resonanzfrequenz klar definiert ist, kann beim Duffing-Oszillator die Resonanzfrequenz von verschiedenen Parametern und Anregungen abhängen. Dies führt dazu, dass sich verschiedene Phänomene wie Hystereseverhalten oder Sprünge in der Amplitude zeigen können, was die Analyse und das Verständnis des Systems herausfordert. Ein komplexeres Resonanzverhalten einzelner Federn könnte jedoch gerade für unser Projekt von Vorteil sein, da es wahrscheinlich auch komplexere Resonanzkurven, welche erlernt werden können, für das ganze System ermöglicht.

2.3 Optimierungsverfahren von MNNs

Um ein MNN zu optimieren, benötigt man zuerst eine oder mehrere Verhaltensweisen, welche vom MNN erlernt werden sollen. Wir unterscheiden dabei zwischen zwei Arten an Verhaltensweisen. Ein Verformungsverhalten besteht aus den Kräften, die auf die Neuronen in der ersten Spalte wirken, und den Änderungen der Positionen der Neuronen in der letzten Spalte, die durch die gegebenen Kräfte bewirkt werden sollen. Ein Resonanzverhalten besteht aus den Frequenzen und Amplituden der auf die Neuronen der ersten Spalte wirkenden Kräfte sowie den zu erreichenden Amplituden entlang der x-Achse der Neuronen der letzten Spalte. Um zu bewerten, wie gut ein MNN die vorgegebenen Verhaltensweisen abdeckt, werden zuerst nach / während der Simulation die Fehler berechnet. Diese sind beim Verformungsverhalten die Abstände zwischen Ruheposition und Zielposition und beim Resonanzverhalten die Differenzen zwischen tatsächlicher Amplitude und Zielamplitude. Dieser Fehler wird quadriert und zum Schluss wird der Durchschnitt aller quadrierten Fehler berechnet. Diese Verlustfunktion nennt sich *mean squared error* (MSE).

2.3.1 Genetische Algorithmen

Genetische Algorithmen können MNNs optimieren, indem sie sich an Ideen der Evolution orientieren [10]. Lee *et al.* (2022) haben das genaue Verfahren, welches sie für die evolutionäre Optimierung verwendet haben, weder genau beschrieben noch veröffentlicht. Wir haben unser Verfahren deshalb, basierend auf eigener Erfahrung und [10], folgendermaßen aufgebaut: Mehrere MNNs mit zufälligen Federkonstanten bilden die erste Population. Der genetische Algorithmus beginnt damit, jedem MNN der Population einen zu minimierenden *score* (bei uns der MSE) zu geben. Die MNNs mit dem geringsten MSE werden ohne Mutation direkt an die nächste Generation übergeben. Anschließend wird der Rest der nächsten Population durch *crossover* bestimmt. Dies funktioniert, indem zwei MNNs aus der Population ausgewählt werden, wobei MNNs mit geringerem MSE 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 mit einer vorgegebenen Mutationsstärke vorgenommen, um die genetische Vielfalt zu erhöhen. Dieses Verfahren wird wiederholt, bis ein zufriedenstellendes MNN gefunden wurde.

2.3.2 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 MSEs gegeben haben, wird das Vorzeichen der Änderungsrate umgekehrt und, falls diese bereits negativ war, wird sie um 10 % gesenkt. Mit den genannten Werten wäre die Entwicklung der Änderungsrate also $1 \to -1 \to 0.9 \to -0.9 \to 0.81...$ Dies wird für eine vorgegebene Anzahl an Wiederholungen (Epochen) durchgeführt, wobei eine Epoche nicht einen Durchlauf aller Federkonstanten, sondern die Anwendung der aktuellen Änderungsrate auf eine einzelne Federkonstante bezeichnet.

3 Vorgehensweise

3.1 Materialien

- Laptop (i7-11800H, 64 GB RAM, RTX 3060)
- Julia als Programmiersprache, mit folgenden wichtigen Softwarebibliotheken:
  - DifferentialEquations.jl zum Lösen der Differenzialgleichungen für die Simulation
  - Graphs.jl und MetaGraphsNext.jl für Modellierung des MNNs als Graph
  - GLMakie.jl und Observables.jl für Visualisierung
  - CSV.jl, Arrow.jl und DataFrames.jl für Speicherung und Verarbeitung der Versuchsergebnisse

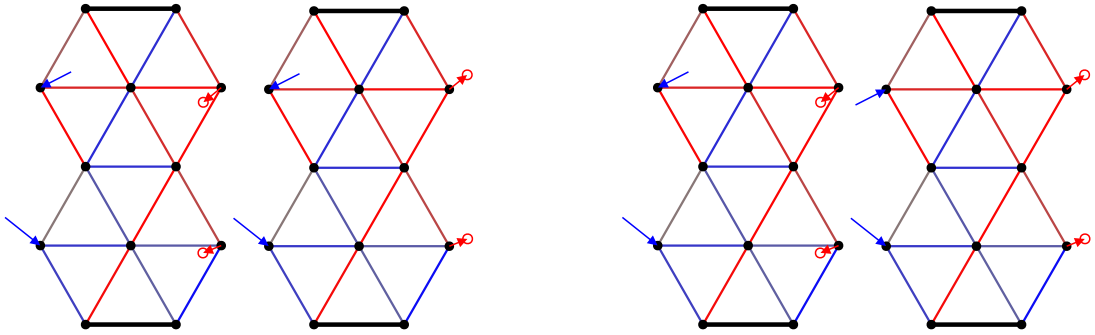
3.2 Methoden

Als erster Schritt war die Umsetzung der Simulation notwendig. Nach Herleitung der Differenzialgleichung (s. Abschnitt „Simulation von MNNs“) haben wir diese zunächst mit dem 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. Jedoch ist das Eulerverfahren zum Lösen der Gleichungen nicht sehr praktikabel, da man zwischen großen Zeitschritten mit kumulativen Ungenauigkeiten und kleinen Zeitschritten mit langer Laufzeit wählen muss. Deshalb haben wir die Julia-Bibliothek DifferentialEquations.jl verwendet, welche automatisches Lösen von Differenzialgleichungen mit verschiedenen Algorithmen ermöglicht. Wir haben die in der Dokumentation [2] empfohlene Kombination von Tsit5 (basierend of Ch. Tsitouras' Runge-Kutta-Verfahren aus 2011 [11]) mit Rosenbrock23 verwendet, welche identische Ergebnisse zu einem Euler-Verfahren mit sehr geringen Zeitschritten liefert, jedoch gleichzeitig deutlich schneller ist (ca. 60 ms im Vergleich zu ca. 500 ms Rechenzeit, um ein 17-Neuronen-MNN für einen Zeitraum von 100 Sekunden zu simulieren). Diese Kombination haben wir für alle Ergebnisse und Analysen verwendet und werden wir als Tsit5 abkürzen.

Um zu testen, wie gut sich ein MNN mit den verschiedenen Algorithmen optimieren lässt, wird zuerst eine Architektur für die MNNs benötigt. Wir haben dafür die im Abschnitt „Hintergrund und theoretische Grundlagen“ beschriebene Architektur von Lee *et al.* (2022) verwendet, damit eine bessere Vergleichbarkeit zu den bereits gefundenen Ergebnissen besteht. Die auf das System einwirkenden Kräfte wirken auf die linke Seite des MNNs ein 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 und Parameter mit einer vereinfachten Problemstellung zu testen und zu verstehen, bevor man sie in tatsächlichen Anwendungsfällen einsetzt.

Für eine systematische Analyse müssen zusätzlich zu der Architektur der MNNs auch die verschiedenen zu erlernenden Verhaltensweisen festgelegt werden. Bei der Optimierung des Verformungsverhaltens nutzen wir dafür zufällig generierte Vektoren für die einwirkenden Kräfte (Kraftvektoren) sowie die gewünschten Positionsänderungen in der letzten Schicht (Zielvektoren). Bei diesem Vorgehen können jedoch auch unmögliche Kombinationen von Verhaltensweisen generiert werden, z.B. entgegengesetzte Positionsänderungen bei gleichen einwirkenden Kräften (s. Abb. 4 (a)). Um dies zu verhindern, überprüfen wir beim Generieren

mehrerer Verformungsverhaltensweisen, dass für ein Neuron der gerade zufällig generierte Kraft- und Zielvektor mit allen anderen Vektoren für dieses Neuron mindestens einen vorgegebenen 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 und die Skalierung der Länge der Zielvektoren.



(a) **(b)**

Abbildung 4: Die schwarzen Punkte sind die Neuronen, die Verbindungslinien die Federn. Negative Federkonstanten sind blau, positive sind rot, Federkonstanten nahe 0 sind grau. Die zwei dicken schwarzen Linien repräsentieren die Wände, an denen Neuronen fixiert sind.
(a) Zwei Verhaltensweisen, die gemeinsam unmöglich erfolgreich anzutrainieren sind. Bei beiden sind die Kraftvektoren (blaue Pfeile) gleich, die Zielpositionen (rote Kreise) unterscheiden sich jedoch.
(b) Zwei Verhaltensweisen, deren gemeinsames Antrainieren möglich sein könnte, die bei uns jedoch nicht vorkommen könnten, da mindestens ein Neuron der ersten Schicht (hier das untere) 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 (a) wurde jedoch der Kraftvektor des oberen Neurons verändert.

Dieses Vorgehen verhindert zwar unmögliche Szenarien, schließt jedoch auch womöglich schwere, aber dennoch mögliche Kombinationen von Verhaltensweisen aus (s. Abb. 4 (b)). Trotzdem sollte es für eine Analyse reichen, da es um einen Vergleich verschiedener Parameter geht und dieses Verhalten immer gleich ist.

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, das Skript Compare.jl für das programmatische Testen und Vergleichen verschiedener Hyperparameter für Verformungsverhalten und das Skript CompareResonance.jl für Resonanzverhalten. 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, Anzahl an Verhaltensweisen, 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. Für CompareResonance.jl haben wir eine Speicherung von Netzwerk und Verhaltensweisen in Binärdaten mithilfe des Arrow-Dateiformats implementiert, um leichter Resonanzkurven und Analysen der Federkonstanten erstellen zu können. Das geschah nach der Umsetzung von Compare.jl, weshalb dort bisher nur CSV-Dateien verwendet wurden.

Mithilfe von Multithreading werden mehrere Netzwerke gleichzeitig trainiert, oft 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 den Optimierungsalgorithmen bedenkt. Da so eine Unterscheidung verschiedener Läufe auf Basis der 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 Läufe mithilfe der CSV-Dateien und der Quellcode-Versionierung mit Git reproduzierbar sein sollten. Insgesamt werden in Compare.jl zum Vergleichen für jedes Netzwerk alle fünf Epochen also folgende Daten gespeichert:

- aktuelle Zeit
- UUID
- Gesamtzahl an Trainingsepochen seit Erstellung
- Anzahl an Reihen und Spalten des Netzwerks
- Anzahl an Verhaltensweisen
- Art der Simulation (Eulerverfahren vs. Tsit5)
- die Laufzeit der Simulation (in Sekunden)
- oben genannte Parameter für Erstellung der Verhaltensweisen (Mindestwinkel, Skalierungen)
- aktuelle Mutationsstärke (falls genetischer Algorithmus)
- MSE des Netzwerks

Bei den verschiedenen Testdurchläufen werden immer unterschiedliche Kombinationen von Anzahl der Reihen, Anzahl der Spalten, Anzahl der Verhaltensweisen und den unterschiedlichen Optimierungsalgorithmen festgelegt und der MSE 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 Optimierung der Resonanzkurven

Zur Berechnung des MSEs bei der Resonanzanalyse nehmen wir die letzten 20 % der Simulation, bei denen das MNN einen stabilen Zustand erreicht haben sollte. Wir teilen die Differenz des größten ($x_{\max,i}$) und kleinsten ($x_{\min,i}$) erreichten Wertes der x-Koordinate eines ausgewählten Neurons i in diesem Zeitraum durch zwei, um die maximale Amplitude zu erhalten, $A_i = (x_{\max,i} - x_{\min,i})/2$. Die Gesamtamplitude ergibt sich über die Mittelung über alle Neuronen der letzten Spalte.

3.2.2 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 [6, 9, 8], ist uns die Idee gekommen, diesen Algorithmus auch für MNNs umzusetzen.

Der Prozess beginnt mit der Vorwärtspropagation, bei der Eingabedaten durch das ANN fließen und eine Ausgabe erzeugt wird. Der erzeugte Ausgabefehler wird dann durch den Vergleich mit den gewünschten Ausgabewerten berechnet. Im nächsten Schritt wird der Fehler rückwärts durch das ANN propagiert, um die Beiträge jedes Neurons zur Fehlerentstehung zu quantifizieren. Die Gewichtungen werden entsprechend dem Fehler 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 der Verhaltensweise 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 erst bestimmt werden, was der Winkel zwischen dem Fehlervektor und dem Vektor zwischen den beiden Neuronen der Federn ist. Wenn dieser z.B. 90° beträgt, ist eine Änderung der Federkonstante nicht notwendig, da die Kraft den Fehler aufgrund ihrer Richtung nicht beeinflussen kann. Wenn er 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. Beträgt der Winkel 180°, dann wollen wir die Federkonstante senken, da die Kraft genau in die falsche Richtung ausgeübt wird.

Allgemein soll die Federkonstante also um den folgenden Term erhöht werden:

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

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

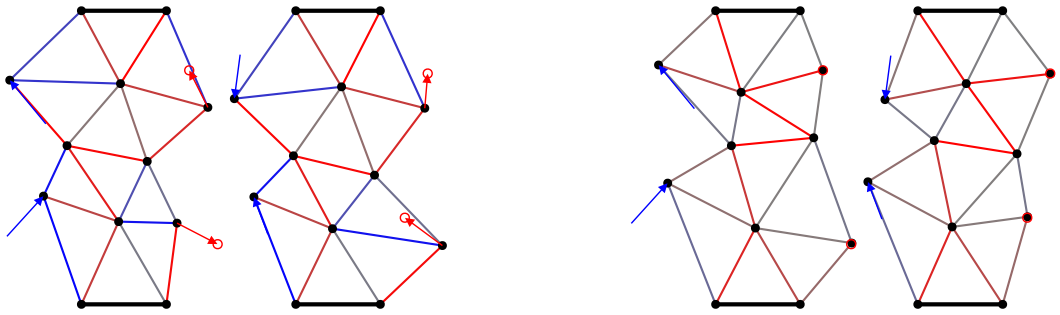
Leider ist es uns nicht gelungen, ein MNN mithilfe von Backpropagation zu trainieren, da der MSE nicht signifikant gesunken ist. Dies könnte unter anderem daran liegen, dass Backpropagation voraussetzt, dass die Daten / Kräfte nur in eine Richtung weitergegeben werden können, was bei MNNs nicht der Fall ist. Wenn man eine Federkonstante ändert, beeinflusst dies die Positionen aller Neuronen. Diese komplexen Wechselwirkungen werden bei der Backpropagation nicht berücksichtigt, da sie nur für künstliche neuronale Netzwerke entwickelt wurde.

4 Ergebnisse

Bei den Ergebnisgraphen haben wir für die Reproduzierbarkeit die zugrundeliegenden CSV-Dateien mit den Hyperparametern und UUIDs aufgelistet. Diese Dateien sind im GitHub-Repository [7] unter `src/data/` zu finden. Ein Lauf bezieht sich auf jeweils einen Trainingslauf mit einem MNN mit einzigartiger UUID.

4.1 Verformungsverhalten

Wie man in Abb. 6 (blau) und Abb. 5 sehen kann, kann man das Verformungsverhalten von MNNs mithilfe von PPS erfolgreich und mit einem kontinuierlich sinkenden MSE trainieren. Die 500 Epochen, welche mit 14 verschiedenen Konstellationen von jeweils 3 verschiedenen Verhaltensweisen trainiert wurden, benötigten auf unserer Hardware ungefähr eine Rechenzeit von 14 Minuten. Das heißt, dass wir pro Verhaltensweise und Epoche auf eine Laufzeit von ungefähr 0,04 Sekunden kommen. Das Training des Verformungsverhaltens mit dem genetischen Algorithmus ist im Gegensatz zu PPS nicht erfolgreich gewesen. Nach 200 Epochen lag der minimale MSE bei ungefähr 0,55, also ca. zehnfach größer als beim Training mit PPS (s. Abb. 6, roter Graph). Auf diesem Niveau lag der MSE des Netzwerks jedoch schon nach 80 Epochen, er befindet sich also in einem lokalen Minimum.



(a) Vor dem Training **(b)** Nach dem Training (PPS, 1300 Epochen)

Abbildung 5: Das Ergebnis des Antrainierens von zwei automatisch generierten Verformungsverhaltensweisen gleichzeitig, hier für Übersichtlichkeit mit einem kleinen MNN (5 Spalten, 3 Reihen).

Wir haben versucht, dieses Problem zu lösen, indem wir die Mutationsstärke automatisch senken, wenn es keine Verbesserung des MSEs gab, um eine feinere Anpassung der Federkonstanten zu ermöglichen. Dies sollte das schnelle Training mit hoher Mutationsstärke und den niedrigen MSE bei kleiner Mutationsstärke kombinieren, wir konnten jedoch keinen signifikant besseren durchschnittlichen MSE erreichen. 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 andere Vergleich von genetischer Optimierung und PPS wurde von Lee *et al.* (2022) durchgeführt, die zu unterschiedlichen Ergebnissen kamen: 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 [4, Abb. 4]. PPS scheint also in der Tat eine Optimierung der MNNs mit einer kürzeren Laufzeit zu ermöglichen. Für eine Aussage über den niedrigsten erreichbaren MSE müssten

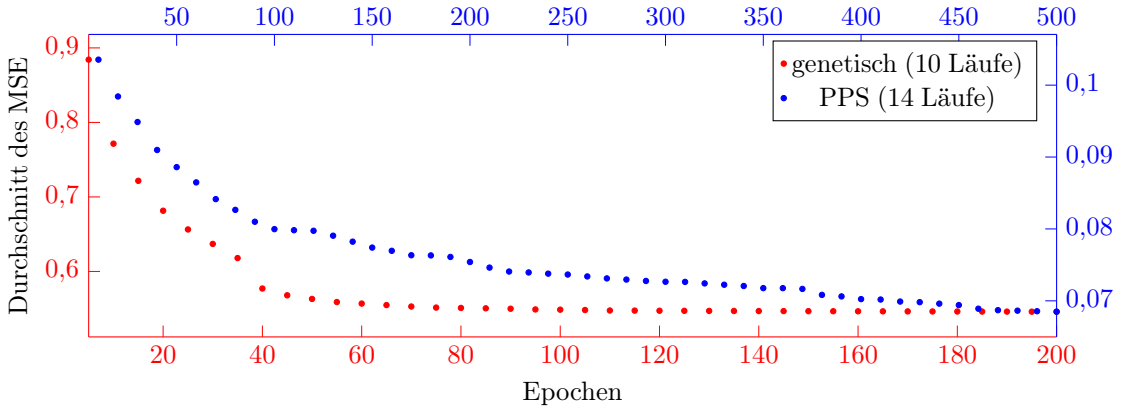
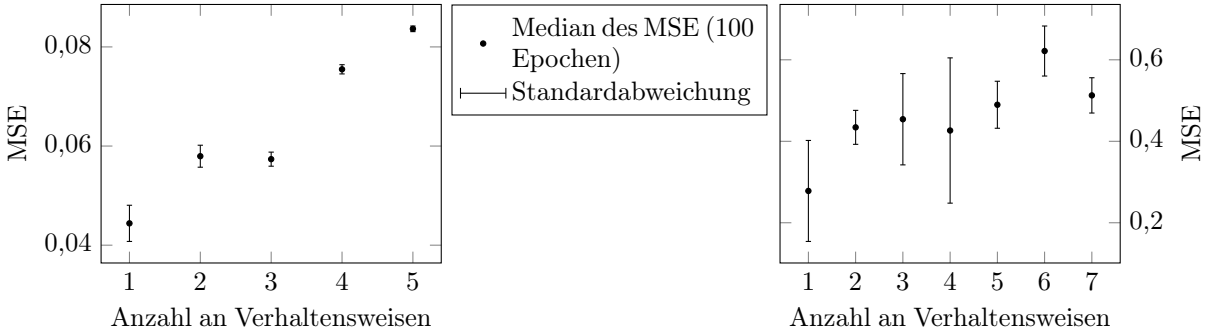


Abbildung 6: Die Mittelwerte des MSEs vor dem Training (0 Epochen) – bei PPS ≈ 0.97 und beim genetischen Algorithmus ≈ 1.18 – werden für eine bessere Achsenskalierung nicht mit angezeigt. Die Epochen der beiden Graphen sind nicht miteinander vergleichbar, da der genetische Algorithmus mehr Zeit pro Epoche und demnach auch insgesamt mehr Zeit für alle Epochen benötigt hat. Es ist dennoch klar erkennbar, dass bei der Optimierung mit PPS der MSE auf unter 0,07 gesunken und dabei kontinuierlich gefallen ist, während er beim genetischen Algorithmus nur ca. 0,55 erreichen konnte und danach nicht weiter signifikant abfiel. Daten PPS: `PPSNumBehaviours_2024-01-14T13-56-06.441.csv`, Daten evolutionär: `EvolutionEpochs_2024-01-14T20-57-11.627.csv`

wir jedoch das Problem des lokalen Minimums für den genetischen Algorithmus lösen und beide Algorithmen über einen längeren Zeitraum laufen lassen.

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 mehr Verhaltensweisen gleichzeitig trainiert werden [4, Abb. 5A u. 5C]. Sie haben diesen Zusammenhang jedoch nur mit dem gradientenbasierten Optimierungsverfahren *fmincon* von MatLab bestätigt, dessen Implementation, soweit wir finden konnten, nicht öffentlich ist, und laut Lee *et al.* (2022) für nicht-simulierte MNNs schwer anwendbar ist [5, S. 6]. Wir konnten dieses Verhalten hingegen sowohl bei PPS als auch zumindest grob beim genetischen Algorithmus bestätigen (s. Abb. 7) – die beiden Algorithmen, die bereits für das erfolgreiche Training von einem echten MNN benutzt wurden. Somit können wir sagen, dass dieser Zusammenhang vermutlich auch bei echten MNNs besteht und – auch wenn weitere Datenpunkte und eine Regression für eine sichere Aussage fehlen – bei PPS vermutlich annähernd linear ist.



(a) Die MNNs wurden mit PPS trainiert, der Median wurde von je 30–34 Läufen gebildet. Daten: `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`.

(b) Die MNNs wurden mit dem evolutionären Algorithmus trainiert, der Median wurde von je 15 Läufen gebildet. Daten: `EvolutionNumBehaviours_2024-01-15T16-03-55.251.csv`.

Abbildung 7: Bei (a) und (b) sind abgesehen von Anzahl an Verhaltensweisen und Optimierungsverfahren für jeden Lauf alle Hyperparameter gleich.

Weiter konnten wir einen anderen bisher nur mit *fmincon* analysierten Zusammenhang [4, Abb. 5B] auch mit PPS bestätigen: Je mehr Reihen ein Netzwerk hat, desto höher der MSE, und je mehr Spalten ein Netzwerk hat, desto niedriger der MSE (vgl. Abb. 8). Wir vermuten, der Grund dafür ist, dass sowohl eine erhöhte Anzahl an Reihen als auch an Spalten die Menge an Federn und somit die notwendigen Schritte zur erfolgreichen Optimierung erhöht. Jedoch senkt eine höhere Anzahl an Spalten den MSE wahrscheinlich 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.

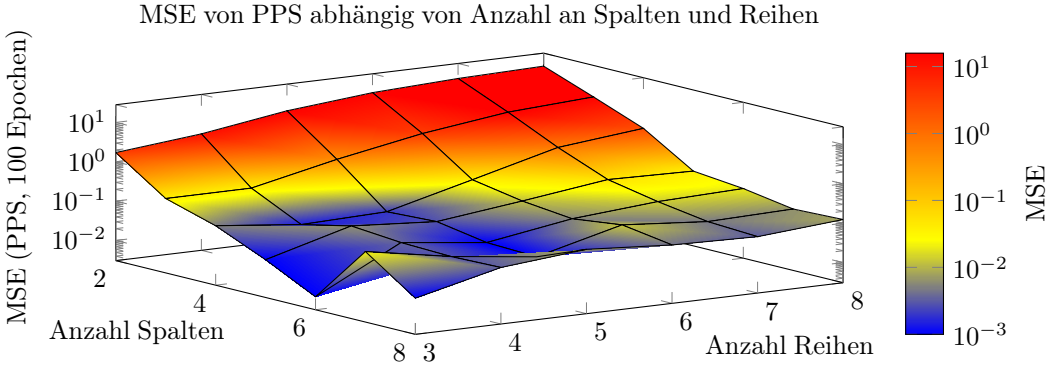


Abbildung 8: MSE für den PPS Algorithmus in Abhängigkeit der Anzahl an Spalten und Reihen. Pro Kombination gab es nur ein Netzwerk. Daten: `PPSNumRowsColumns_2024-01-06T13-23-22.688.csv`

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

Zusätzlich zu der Bestätigung der von Lee *et al.* (2022) bereits gefundenen Ergebnisse haben wir weitere, komplett neue Versuche durchgeführt, wie z.B. die Analyse des Einflusses der Längen der Vektoren (also Größen der Kräfte und Positionsänderung) bei Verformungsverhalten auf den MSE (s. Abb. 9). Die Ergebnisse sind nicht ganz eindeutig, jedoch ist eine positive Korrelation zwischen den Faktoren der Vektoren und dem MSE zu erkennen. Dies könnte dadurch erklärt werden, dass bei langen Kraft- und Zielvektoren deutlich extremere Federkonstanten notwendig sind, die sich von den Startwerten stark unterscheiden. So dauert es länger, diese zu erreichen, und mit der gleichen Trainingszeit können die optimalen Federkonstanten schlechter angenähert werden.

4.2 Resonanzoptimierung

In der bisherigen Literatur wurde – soweit wir finden konnten – eine Anpassung der Resonanzkurve eines MNNs zwar kurz als Einsatzmöglichkeit erwähnt [3], aber nie umgesetzt oder durchgeführt. Also haben wir zu Beginn einzelne Tests durchgeführt (s. z.B. Abb. 10), welche uns zeigten, dass die Optimierung der Resonanzkurve möglich ist. Bei dem Histogramm der Federkonstanten fällt auf, dass es kaum negative Federkonstanten gab. Dies hat sich auch bei 13 weiteren Testläufen, die wir durchgeführt haben, bestätigt: Dort haben die negativen Federkonstanten aller MNNs nur $\approx 1{,}48\,\%$ aller Federkonstanten ausgemacht. Ein weiterer Test, bei dem negative Federkonstanten entfernt wurden, war ebenfalls erfolgreich. Dies bedeutet, dass negative Federkonstanten zwar für Verformungsverhaltensweisen wichtig sind, bei Resonanzoptimierungen aber auch nur positive Federkonstanten ausreichen. Daraus lässt sich schließen, dass zumindest bei der Anpassung von Resonanzkurven ein Problem der Anwendung von *binary stiffness beams* nicht existiert und eine Suche nach kleineren und effizienteren Federdesigns mit ausschließlich positiven, kontinuierlichen Federkonstanten sinnvoll wäre. Insgesamt ist davon auszugehen, dass Federn mit negativen Federkonstanten ein Problem in der Konstruktion darstellen, da sie mit Energie versorgt werden müssen, weshalb es von Vorteil ist, wenn ein MNN keine oder nur kaum negative Federkonstanten besitzt.

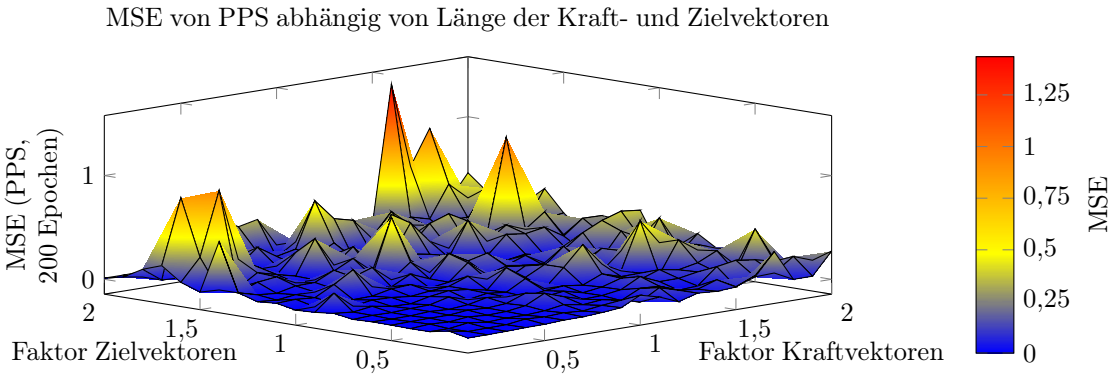


Abbildung 9: MSE für den PPS Algorithmus in Abhängigkeit der Länge der Kraft- und Zielvektoren. Die ursprünglichen Vektoren (jede Komponente zwischen -1 und 1) wurde mit den Faktoren multipliziert. Pro Kombination ist der MSE der Median von drei Läufen, insgesamt gab es 1200 Läufe. Es ist zu erkennen, dass der MSE am geringsten ist, wenn Kraftvektor und Zielvektor möglichst kurz sind. Je höher beide werden, desto höher wird der MSE tendenziell. Daten: `PPSMagModifierGoal_2024-02-18T13-56-29.csv`, `PPSMagModifierGoal_2024-02-18T14-43-15.csv`

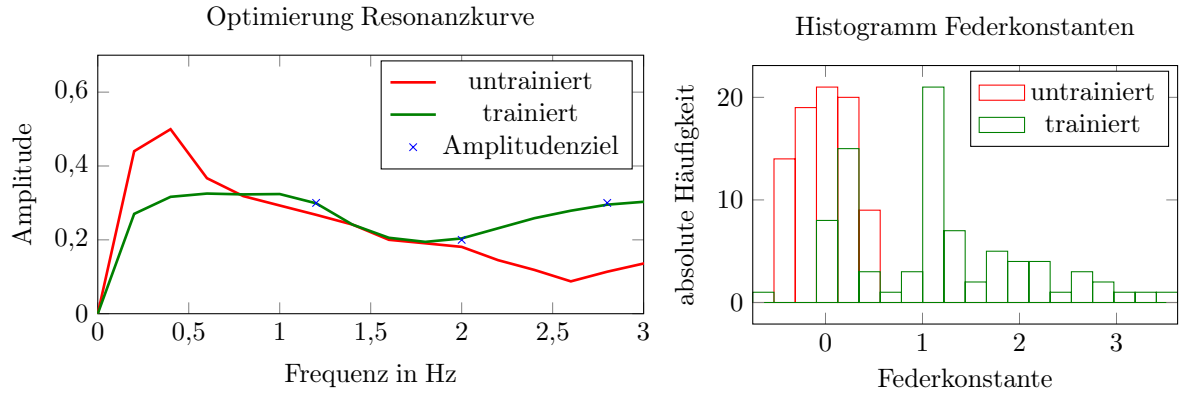


Abbildung 10: Resonanzkurve und Federkonstanten eines MNNs vor und nach dem Training. Es wurde erst 1200 Epochen lang mit PPS trainiert, dann nach Erreichen eines Plateaus des MSEs 150 Epochen lang mit dem evolutionären Algorithmus.

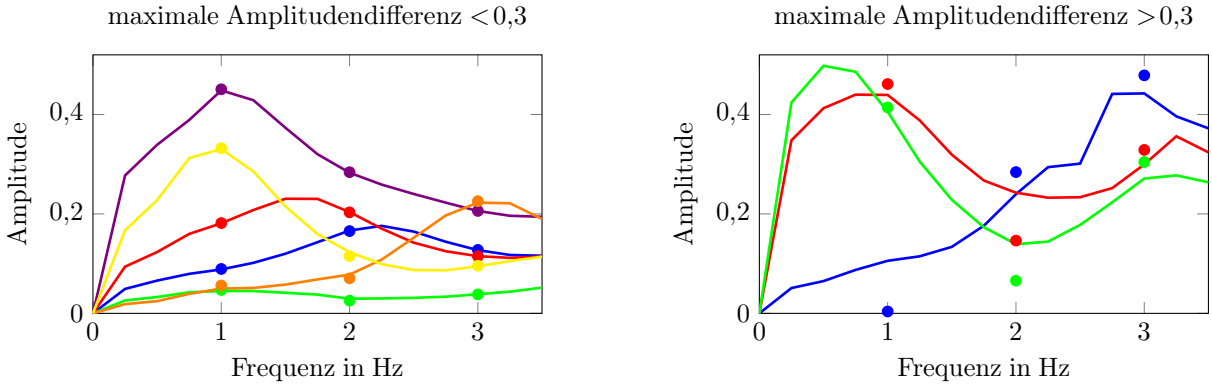


Abbildung 11: In beiden Graphen entspricht jede Farbe einem Lauf. Die Linie ist die tatsächliche Resonanzkurve nach der Optimierung (PPS, 1000 Epochen), die Punkte sind die zufällig generierten Zielamplituden des Laufes. Die maximale Amplitudendifferenz bezieht sich auf die jeweilige Differenz zwischen höchster und tiefster Zielamplitude jedes Laufes. Alle weiteren Hyperparameter waren identisch. Daten: `ResonanceEpochs_2024-03-20T16-05-25.arrow` und `ResonanceEpochs_2024-03-20T17-18-42.arrow`. Für den rechten Graphen wurden für Übersichtlichkeit drei von sieben passenden Läufen zufällig ausgewählt.

Für die erwähnten 13 Testläufe haben wir außerdem jeweils die Differenz zwischen höchster und tiefster Zielamplitude berechnet und damit die Läufe getrennt (s. Abb. 11). Es fällt auf, dass die Resonanzoptimierung bei höheren Zielamplitudendifferenzen deutlich schwieriger ist. Ob nur mehr Training notwendig ist oder wir teilweise schon die Grenzen eines MNNs der verwendeten Größe / Architektur erreicht haben, wird daraus nicht klar. Jedoch ist zumindest eine deutliche Annäherung der Zielamplituden auch bei den Graphen mit großer Zielamplitudendifferenz erkennbar.

4.3 Framework

Unsere Softwarebibliothek [7] soll nicht nur von uns leicht bedient und erweitert werden, sondern auch von anderen. Hierzu fehlen zwar noch eine ausführliche Dokumentation und Anleitungen, doch der Programmcode ist durch die Aufteilung der einzelnen Aufgabenbereiche und Aspekte auf verschiedene Verbünde / Strukturen bereits entsprechend gestaltet (s. Abb. 12 und Programmausschnitt 1). Denn dies bietet gemeinsam mit den Möglichkeiten der Programmiersprache Julia wie der Funktionsüberladung die Option, eigene Aspekte einfach hinzuzufügen (s. Programmausschnitt 2). Der modulare Aufbau der existierenden Funktionalität erlaubt außerdem bereits eine hohe Flexibilität für Nutzer (s. Tabelle 1). Die von uns entwickelte Bibliothek ist unserem Wissen nach die erste öffentlich zugängliche Software-Implementation von MNNs und wir stellen auch die ersten öffentlichen Daten von den Trainingsverläufen für weitere Analysen zur Verfügung [7].

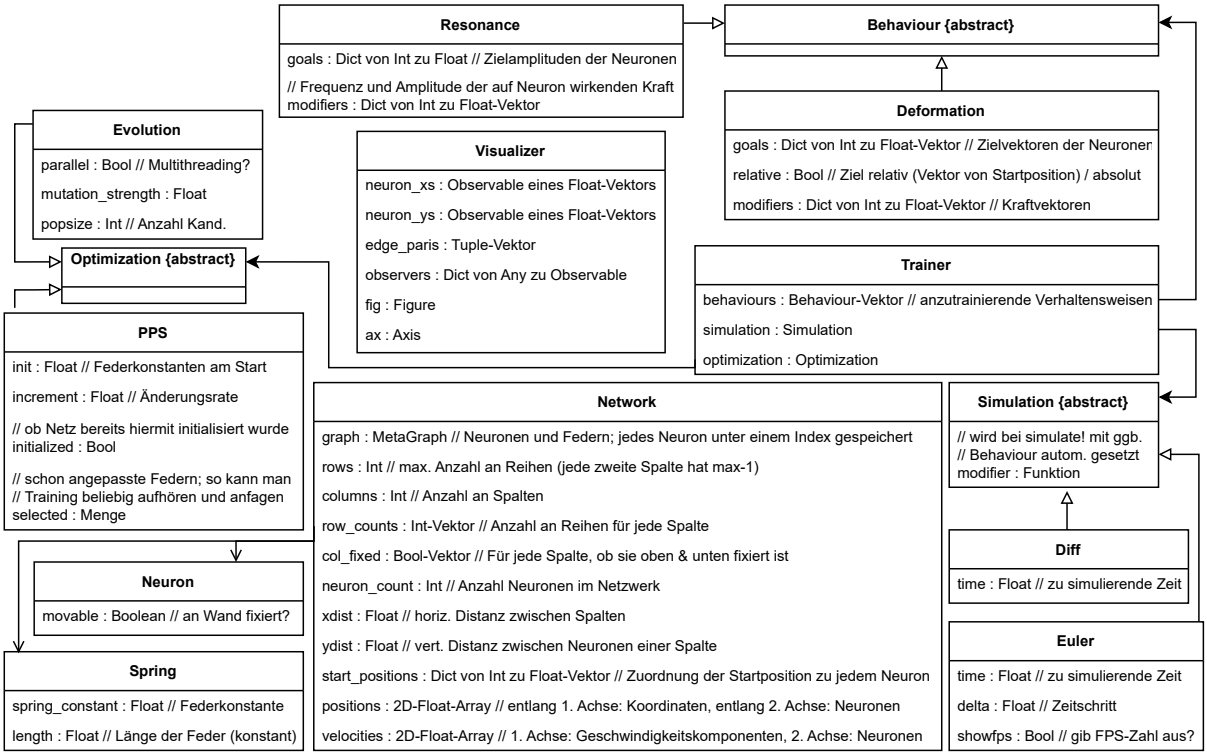


Abbildung 12: UML-Diagramm des Programm-Aufbaus. Da es sich hier um Verbünde und nicht Klassen handelt, sind die Methoden nicht mit aufgelistet.

Tabelle 1: Die verschiedenen bereits implementierten Module für die drei Komponenten der Optimierung eines MNN. Jede beliebige Kombination von jeweils einem Modul der ersten zwei Spalten und mindestens einem Modul der dritten Spalte kann verwendet werden. Momentan ist die Verwendung von *Euler* mit *Resonance* nicht möglich. Beim Trainieren können zu beliebigen Zeitpunkten Komponenten gewechselt werden.

Simulationsverfahren	Optimierungsverfahren	Art der zu untersuchenden Verhaltensweise
Euler	PPS	Deformation (Verformung)
Diff (Tsit5)	Evolution	Resonance (Schwingung / Resonanz)

```julia
using MNN # lade Paket
net = Network(5, 4) # erstelle Netzwerk mit 5 Spalten und 4 Reihen
# 2 Deformation Verhaltensweisen, Tsit5 für Simulation, PPS für Optimierung
t = Trainer(create_deformation_behaviours(net, 2), Diff(500), PPS(net))
train!(net, 100, t) # Training für 100 Epochen
# Netz + Verhaltensweise anzeigen; wird während Simulation autom. aktualisiert
vis = Visualizer(net, behaviour=t.behaviours[1])
simulate!(net, Diff(100), t.behaviours[1], vis = vis) # 1. Verhaltensweise simulieren
```

Programmausschnitt 1: Eine mögliche simple Verwendung von MNN.jl.

```julia
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, t) -> nothing), ...)
function MNN.simulate!(network::Network, sim::MySim;
 ↪  vis::Union{Visualizer,Nothing}=nothing)
   # ...
   # setze Beschleunigungen entspr. Kraftvektoren; aktualisiere Visualisierung
   sim.modifier(network, my_acc_var, t); MNN.update_positions!(vis, network)
   # ...
end
# nutze MNN.jl wie üblich (s. Programmausschnitt 1), nur mit MySim statt Diff, z.B.:
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.

5 Diskussion

Unser neuer Optimierungsansatz Backpropagation war zwar nicht erfolgreich, jedoch konnten wir erfolgreich die Leistung der zwei von Lee *et al.* (2022) verwendeten Optimierungsalgorithmen beurteilen. Dabei stimmen unsere Ergebnisse 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 das Black Box Verfahren *fmincon* verwendet [5, S. 6], da dies schneller ist. Wir haben diese Tests nun direkt mit den zu untersuchenden, für physische MNNs besser nutzbaren, Verfahren durchgeführt, und konnten so verfahrensbedingte Unterschiede ausschließen. Ausnahme bildet die Analyse der Auswirkungen der Spalten- und Reihenzahl, welche aufgrund der hohen benötigten 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 dem Vergleich der Laufzeit beider Algorithmen nun auch bestimmen zu können, welches Verfahren unabhängig von der Laufzeit den niedrigsten MSE erreichen kann. 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. Unser Problem ist, dass beim Stagnieren des MSEs nicht bekannt ist, ob man sich in einem lokalen Minimum befindet und so eine höhere Mutationsstärke notwendig ist, um es zu verlassen, oder über das Minimum „hinwegspringt“ und so eine

niedrigere Mutationsstärke benötigt. Deshalb wollen wir probieren, bei Stagnation des MSEs Mutationsstärke erst für eine bestimmte Anzahl an Wiederholungen zu verkleinern und bei weiterer Stagnation dann zu vergrößern.

Da wir uns bei Verformungsverhalten auf den stabilen Endzustand der MNNs beschränken, könnte man die dafür verwendeten 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 Mutationsstärke für genetische Optimierung).

Weiter wollen wir in der Zukunft Kombinationen an Optimierungsverfahren genauer 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 genetischen Algorithmus, um einen niedrigeren MSE erreichen zu können – vorausgesetzt, wir können bestätigen, dass genetische Algorithmen niedrigere MSEs erreichen können als PPS.

Zudem wollen wir noch mehr Trainingsdurchläufe mit dem Resonanzverhalten durchführen. Unsere bisherigen Ergebnisse sind vielversprechend und öffnen die Tür für viele weitere Einsatzmöglichkeiten von MNNs. Mithilfe der von uns implementierten Resonanzoptimierung wäre es z.B. vorstellbar, 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 passend zu reagieren, oder Brücken, die ihre Resonanzfrequenz dynamisch an die aktuellen Erregerfrequenzen anpassen können, um einen Einsturz zu vermeiden. Es wäre auch sehr interessant, ein MNN zu trainieren, welches sowohl ein optimiertes Resonanzverhalten als auch ein optimiertes Verformungsverhalten aufweist, um so einen noch größeren Teil der möglichen Anwendungsfälle abzudecken. Weiter könnte man versuchen, gezielt MNNs mit reduzierten Amplituden in bestimmten Frequenzbereichen zu trainieren, um spezifisch eine Nutzung für einsturzsichere Brücken zu analysieren.

Unsere Programme sollten ebenso mit einem dreidimensionalen MNN funktionieren. Die Implementation wäre – abgesehen von der Visualisierung – 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, dreidimensionale MNNs zu implementieren und zu analysieren, sobald wir mit allen Optimierungsverfahren erfolgreich zweidimensionale Netzwerke trainieren können und ihre relevanten Hyperparameter analysiert haben. Wenn wir dreidimensionale MNNs realisiert haben, wollen wir andere Geometrien des Systems untersuchen, z.B. ein System, bei dem nur die untersten Neuronen (in der Erde) fixiert sind. Dies könnte die Anregung eines Gebäudes auf ein Erdbeben simulieren. Hier wäre die Idee, ob es möglich ist, den zu erwartenden Schaden zu minimieren.

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 ein 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] *DifferentialEquations.jl Dokumentation*. URL: https://docs.sciml.ai/DiffEqDocs/stable/ (besucht am 20. 02. 2024).

[3] 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.

[4] 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).

[5] 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).

[6] Alexander Reimer und Matteo Friedrich. *KI in der Musik – Computer lernen komponieren*. Feb. 2021. URL: https://github.com/Alexander-Reimer/AI-Composer.jl/blob/paper/JuFo-2021-Landeswettbewerb/Bericht.pdf.

[7] 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).

[8] Alexander Reimer und Matteo Friedrich. „Steuerung per Lidschluss, Erkennung ereigniskorrelierter Potenziale eines Elektroenzephalogramms durch ein neuronales Netz". In: *PTB-OAR* (2023). DOI: 10.7795/320.202308.

[9] Alexander Reimer, Matteo Friedrich und Mattes Brinkmann. *Entwicklung eines Frameworks und eigener Hardware für Brain-Computer-Interfaces*. Jan. 2023. URL: https://github.com/Alexander-Reimer/Interpreting-EEG-with-AI/blob/paper/2023/JugendForscht/main.pdf.

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

[11] Ch. Tsitouras. „Runge–Kutta pairs of order 5(4) satisfying only the first column simplifying assumption". In: *Computers & Mathematics with Applications* 62.2 (2011), S. 770–775. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2011.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0898122111004706.

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

[13] Wikipedia contributors. *Duffing equation — Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=Duffing_equation&oldid=1178997341 (besucht am 14. 03. 2024).