Neuronale Netzwerke und Machine Learning: Eine technische Einführung in Künstliche Intelligenz

Abschließende Arbeit
verfasst von
Magomed Alimkhanov
Klasse: 8b

Betreuer:in:
Prof. Mag. Marius Reiter

Februar 2025

BG/BRG Rohrbach
Hopfengasse 20
4150 Rohrbach-Berg

Abstract

Die vorliegende Arbeit befasst sich mit künstlicher Intelligenz; im Speziellen ihrer Funktionsweise und diversen Trainingsprozessen, welche zur Leistungsoptimierung dieser Systeme erforderlich sind. Dabei wurden unter anderem folgende Fragen ausführlich beantwortet: Wie ist ein neuronales Netzwerk aufgebaut? Wie lernt es, Informationen zu erkennen? Die Recherche fand überwiegend im Bereich der Onlineliteratur statt. Des Weiteren wird dieses Wissen mit einigen programmatischen Beispielen in der Praxis geprüft, etwa durch die Entwicklung einer künstliche Intelligenz zur Zahlenerkennung sowie das Trainieren eines Systems zur Navigation durch einen Irrgarten. Dabei zeigt sich, dass die Funktionsweise künstlicher Intelligenz auf mathematischen Prinzipien wie der Ableitung beruht und auf Konzepten wie der Evolutionstheorie und dem Aufbau des menschlichen Gehirns basiert. Eine Haupterkenntnis ist beispielsweise, dass der Trainingsprozess einer KI sehr ressourcenintensiv ist. Die Architektur eines neuronalen Netzwerks sowie dessen Trainingsalgorithmus variieren je nach Problemstellung und erfordern einen erheblichen zeitlichen Aufwand.

Inhaltsverzeichnis

1 Einleitung

Künstliche Intelligenz ist ein ebenso faszinierendes wie komplexes Feld, das in den letzten Jahrzehnten eine beispiellose Entwicklung durchlaufen hat. Sie findet in nahezu allen Bereichen unseres Lebens Anwendung und verändert die Art und Weise, wie wir arbeiten, lernen und interagieren. Ihre Fähigkeit, Probleme eigenständig zu lösen und aus Erfahrungen zu lernen, hat die Technologiebranche revolutioniert und eröffnet neue Horizonte in Bereichen wie Medizin, Automatisierung und Spieleentwicklung. Zweifelsohne ist KI eine der größten wissenschaftlichen Herausforderungen unserer Zeit, die gleichzeitig enormes Potenzial für die Zukunft birgt. Die vorliegende Arbeit befasst sich mit der Funktionsweise von Künstlicher Intelligenz, insbesondere mit neuronalen Netzwerken und genetischen Algorithmen. Eine zentrale Frage ist, wie KI auf einer mathematischen und intuitiven Ebene funktioniert und wie diese Prozesse in der Programmierung umgesetzt werden können. Ein Schwerpunkt liegt aus diesem Grund naturgemäß auch in der dem zugrundeliegenden mathematischen Theorie dahinter und insbesondere auf der Charakterisierung der typischen Trainingsprozesse von neuronalen Netzwerken und der evolutionären Strategien von genetischen Algorithmen. Die Analyse von praktischen Beispielen, wie der Ziffernerkennung mit dem MNIST-Datensatz, gibt wichtige Einblicke in die Leistungsfähigkeit und die Grenzen dieser Technologien.

2 Neuronale Netzwerke

2.1 Grundlagen

Ein neuronales Netzwerk kann in vielerlei Hinsicht mit einem Gehirn verglichen werden. Ähnlich wie das Gehirn besteht ein neuronales Netzwerk aus vielen Neuronen. Es gibt verschiedene Arten von Neuronen, die sich unter anderem durch ihre Aktivierungsfunktionen unterscheiden. Auf Aktivierungsfunktionen wird in einem späteren Teil der Arbeit genauer eingegangen. Grundsätzlich besitzt jedes Neuron Eingabewerte x_j, die mit Gewichten (Weights) w_j verbunden sind. Zusätzlich besitzt jedes Neuron einen Bias-Wert (Bias) b. All diese Faktoren haben Einfluss auf den Ausgabewert z eines Neurons. Um diesen zu berechnen, wird die gewichtete Summe der Eingabewerte mit dem Bias-Wert addiert (vgl. Nielsen, 2015, #Perceptrons). Mathematisch kann diese Berechnung folgendermaßen dargestellt werden:

$$z = \sum_{j=1}^{n_{\text{inputs}}} w_j x_j + b$$

Dabei steht n_{inputs} für die Anzahl an Eingabewerten und Gewichten.

Mit einem einzelnen Neuron kann nicht viel angefangen werden, deswegen verbindet man die Neuronen miteinander, wodurch ein neuronales Netzwerk entsteht. Es gibt viele verschiedene Arten von neuronalen Netzwerken, das einfachste davon ist das sogenannte „Feedforward Neural Network“. Bei dieser Variante werden Informationen kontinuierlich, d.h. ausschließlich von einer Schicht zur nächsthöheren, weitergeleitet. Es kann in drei Teile unterteilt werden: die Eingabeschicht, die verborgenen Schichten und die Ausgabeschicht (ebd.).

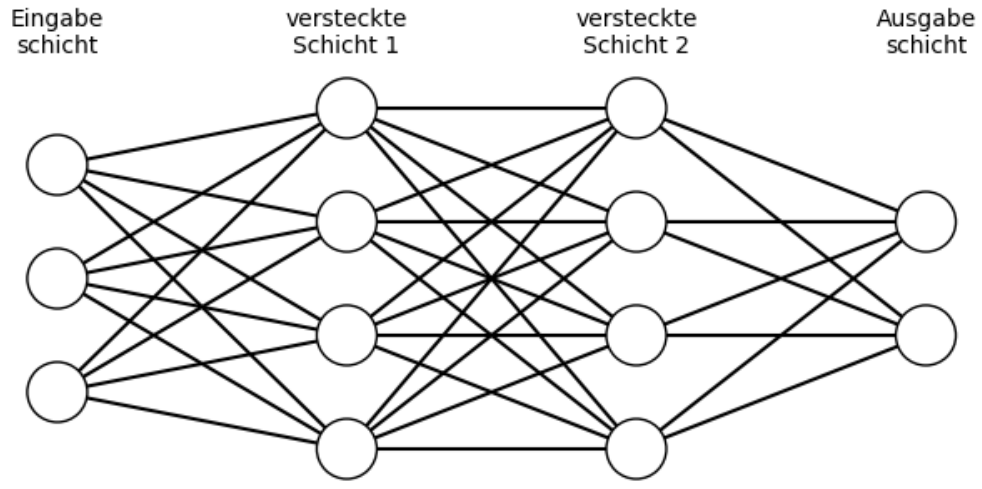

Abb. 1: Neuronales Netzwerk (Verf.)

2.1.1 Die Eingabeschicht (Input Layer)

Die Eingabeschicht empfängt die Daten von externen Quellen. Bei einem Zahlenerkennungsmodell beispielsweise würde die Eingabeschicht die Pixeldaten des Bildes repräsentieren. Dementsprechend benötigt ein 28x28-Bild 784 Eingabeneuronen (vgl. Lheureux, o.J.).

2.1.2 Die verborgenen Schichten (Hidden Layers)

Die verborgenen Schichten sind das, was neuronale Netzwerke so besonders macht. Sie verbinden die Eingabeschicht und die Ausgabeschicht miteinander. Je nach Schwierigkeitsgrad der Anwendung werden mehr und längere verborgene Schichten benötigt. Je mehr verborgene Neuronen es gibt, desto kompliziertere Berechnungen kann das neuronale Netzwerk durchführen (ebd.).

2.1.3 Ausgabeschicht (Output Layer)

Die Ausgabeschicht gibt die endgültigen Vorhersagen des neuronalen Netzwerks zurück. In dem Zahlenerkennungsmodell wären das die Ziffern null bis neun, wobei jede Ausgabe die Wahrscheinlichkeit für eine der Ziffern darstellt (ebd.).

2.2 Forward Propagation

2.2.1 Matrixmultiplikation

Bei der Forward Propagation werden die Eingabewerte des neuronalen Netzwerkes zur nächsten Schicht weitergegeben, bis sie zur Ausgabeschicht kommen und Vorhersagen (Predictions) erzeugen (vgl. Anshumanm2fja, 2024).

Um das neuronale Netzwerk in der Programmierung umzusetzen, verwende ich die Programmiersprache Python in Kombination mit dem Paket „NumPy“. Ein neuronales Netzwerk mit zwei Eingabeneuronen und einem Ausgabeneuron kann wie folgt dargestellt werden:

```python
import numpy as np  # Import des Pakets NumPy

# Die Eingabewerte der Eingabeneuronen
inputs = np.array([[0.3], [0.6]])

# Die Gewichte zwischen den zwei Eingabeneuronen und dem Ausgabeneuron
weights = np.array([0.8, 0.2])
bias = 4  # Der Bias-Wert des Ausgabeneurons

# Berechnung des Ausgabewerts des Ausgabeneurons
output = inputs[0] * weights[0] + inputs[1] * weights[1] + bias
print(output)
```

In diesem Beispiel sind „inputs“ die Eingabewerte der Eingabeneuronen und „weights“ die Gewichtswerte zwischen den Eingabeneuronen und dem Ausgabeneuron. Der Bias-Wert des Ausgabeneurons wird als „bias“ definiert.

Mathematisch kann die Liste an Eingabewerten als Spaltenvektor dargestellt werden:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n_{\text{inputs}}} \end{bmatrix}$$

Hierbei stellt n_{inputs} die Anzahl der Eingabewerte dar und somit hat der Vektor die Dimension $(n_{\text{inputs}} \times 1)$. Der Bias-Wert ist ist hierbei ein Skalar und der Gewichtsvektor ein Zeilenvektor. Diese werden allerdings umgeschrieben.

Im Folgenden werden auch noch die Ausgaben eines neuronalen Netzwerkes mit zwei Eingabeneuronen und zwei Ausgabeneuronen programmatisch berechnet:

```
inputs = np.array([1.2, 3.2])

# Gewichte zwischen Eingabeneuronen und Ausgabeneuronen
weights1 = np.array([0.8, 1.3])  # Gewichte für das erste Ausgabeneuron
weights2 = np.array([3.1, 1.6])  # Gewichte für das zweite Ausgabeneuron

bias1 = 4  # Bias-Wert für das erste Ausgabeneuron
bias2 = 3  # Bias-Wert für das zweite Ausgabeneuron

# Der Ausgabewert des ersten Ausgabeneurons
output1 = inputs[0] * weights1[0] + inputs[1] * weights1[1] + bias1

# Der Ausgabewert des zweiten Ausgabeneurons
output2 = inputs[0] * weights2[0] + inputs[1] * weights2[1] + bias2

print(output1, output2)
```

Jedoch ist diese Schreibweise sehr mühsam und ineffizient, weshalb ich zur Berechnung der Ausgaben Vektoren und Matrizen zusammen in Kombination mit der Matrixmultiplikation verwende. Dazu stelle ich die Bias-Werte ebenso als Spaltenvektor und die Gewichte als Matrix dar:

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n_{\text{neurons}}} \end{bmatrix}$$

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1n_{\text{inputs}}} \\ w_{2,1} & w_{2,2} & \cdots & w_{2n_{\text{inputs}}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_{\text{neurons}}1} & w_{n_{\text{neurons}}2} & \cdots & w_{n_{\text{neurons}}n_{\text{inputs}}} \end{bmatrix}$$

n_{neurons} beschreibt dabei die Anzahl der Neuronen. Der Bias-Vektor $\mathbf{b}$ und die Gewichtsmatrix $\mathbf{W}$ haben beide n_{neurons} Zeilen. Da der Bias-Vektor nur eine Spalte besitzt, hat er die Dimension $(n_{\text{neurons}} \times 1)$. Die Gewichtsmatrix hingegen hat n_{inputs} Spalten und damit die Dimension $(n_{\text{neurons}} \times n_{\text{inputs}})$. Das Gewicht $w_{1,2}$ beschreibt die Verbindung zwischen dem ersten Neuron der aktuellen Schicht und dem zweiten Eingabewert.

Zur Berechnung der Ausgabewerte des neuronalen Netzwerks verwende ich die Matrixmultiplikation. Zur Veranschaulichung für diese Rechenoperation definiere ich die Matrizen $\mathbf{A}$ und $\mathbf{B}$:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{bmatrix}$$

Hierbei hat die Matrix $\mathbf{A}$ die Dimension (2×3) und die Matrix $\mathbf{B}$ die Dimension (3×2).

Bei der Matrixmultiplikation müssen die Spalten der ersten Matrix mit den Zeilen der zweiten Matrix übereinstimmen. Das Ergebnis ist eine neue Matrix, deren Zeilenanzahl der ersten Matrix und deren Spaltenanzahl der zweiten Matrix entspricht. Die Berechnung erfolgt, indem die Zeilen der ersten Matrix mit den entsprechenden Spalten der zweiten Matrix multipliziert und die Produkte anschließend summiert werden (vgl. Jung, 2014). Dies lässt sich wie folgt darstellen:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{bmatrix} =$$

$$\begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} + a_{1,3}b_{3,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + a_{2,3}b_{3,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + a_{2,3}b_{3,2} \end{bmatrix}$$

Die resultierende Matrix hat die Dimension (2×2), da $\mathbf{A}$ zwei Zeilen und $\mathbf{B}$ zwei Spalten besitzt.

Der Vektor $\mathbf{z}$ kann mithilfe der Matrixmultiplikation und einer Matrixaddition folgendermaßen beschrieben werden:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Die Zeilenanzahl von $\mathbf{z}$ entspricht der Zeilenanzahl von $\mathbf{W}$, während die Spaltenanzahl von $\mathbf{z}$ der Spaltenanzahl von $\mathbf{x}$ entspricht. Somit ist $\mathbf{z}$ ein Spaltenvektor mit der Dimension $(n_{\text{neurons}} \times 1)$.

```
inputs = np.array([[1.2], [3.2]])

# Gewichtsmatrix zwischen Eingabeneuronen und Ausgabeneuronen (2 x 2)
weights = np.array(
    [
        [0.8, 1.3],  # Gewichte des ersten Ausgabenneuron
        [3.1, 1.6],  # Gewichte des zweiten Ausgabenneuron
    ]
)

# Bias-Vektor für die Ausgabeneuronen (2 x 1)
biases = np.array([[4], [3]])

# Berechnung der Ausgabewerte (z) (2 x 2) durch Matrixmultiplikation
outputs = np.dot(weights, inputs) + biases
print(outputs)
```

Hierbei stellt „inputs“ den Eingabevektor $\mathbf{x}$ dar, und „weights“ bezeichnet die Gewichtsmatrix $\mathbf{W}$ zwischen Eingabe- und Ausgabeneuronen dar. „biases“ ist der Bias-Vektor $\mathbf{b}$ der Ausgabeneuronen und „outputs“ repräsentiert den Vektor der berechneten Ausgabewerte $\mathbf{z}$.

2.2.2 Batches

Bisher verarbeitet der Code jeweils nur ein Trainingsbeispiel pro Zyklus. Um die Effizienz zu steigern, werden jedoch mehrere Trainingsbeispiele gleichzeitig bearbeitet. Diese Menge an Beispielen wird als Batch bezeichnet. Durch die parallele Verarbeitung lassen sich Berechnungen effizienter durchführen. Daher wird das Training neuronaler Netzwerke in der Praxis meist auf GPUs durchgeführt, da diese über eine große Anzahl an Recheneinheiten verfügen (vgl. Kinsley, 2020, TC: 3:12).

Eine weitere essenzielle Eigenschaft von Batches ist die Normalisierung: Wenn mehrere Beispiele gleichzeitig verarbeitet werden, können Schwankungen in den Ausgabewerten ausgeglichen werden. Dadurch wird das Training stabiler und konsistenter (vgl. Kinsley, 2020, TC: 8:00).

Mathematisch lässt sich ein Batch von Trainingsbeispielen als eine Matrix $\mathbf{X}$ darstellen:

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1 n_{\text{samples}}} \\ x_{2,1} & x_{2,2} & \cdots & x_{2 n_{\text{samples}}} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n_{\text{inputs}} 1} & x_{n_{\text{inputs}} 2} & \cdots & x_{n_{\text{inputs}} n_{\text{samples}}} \end{bmatrix}$$

Dabei bezeichnet n_{samples} die Anzahl der Trainingsbeispiele im Batch. Die Matrix $\mathbf{X}$ hat somit die Dimension $(n_{\text{inputs}} \times n_{\text{samples}})$.

Die Matrixmultiplikation funktioniert weiterhin, solange die Spaltenanzahl der Gewichtsmatrix $\mathbf{W}$ mit der Zeilenanzahl der Eingabematrix $\mathbf{X}$ übereinstimmt. Das Ergebnis ist eine Ausgabematrix $\mathbf{Z}$ mit der Dimension $(n_{\text{neurons}} \times n_{\text{samples}})$.

Der zugehörige Code sieht nun folgendermaßen aus:

```
# Eingabematrix einer Batch mit 4 Trainingsbeispielen,
# wobei jedes Beispiel 2 Eingabewerte enthält (2 x 4)
inputs = np.array([
    [1.2, 3.2, 4.2, 3.1],
    [3.2, 1.2, 0.2, 2.2],
])

# Berechnung der Ausgabewerte durch Matrixmultiplikation
outputs = np.dot(weights, inputs) + biases  # Matrix von Ausgabewerten (2 x 4)
print(outputs)
```

2.2.3 Die Layer-Klasse

Um weitere Schichten hinzuzufügen, kann der bereits vorhandene Code wiederverwendet werden. Um dies effizient umzusetzen, bietet es sich an, Klassen zu schreiben. Daher erstelle ich die Klasse „Layer“. Diese dient als Bauplan für alle Schichten, die instanziiert werden.

```python
class Layer:
    def __init__(self, n_inputs, n_neurons):
        """
        n_inputs: Anzahl an Eingabewerten (bzw. Neuronen der vorherigen
        Schicht).
        n_neurons: Anzahl an Neuronen für diese Schicht.
        """
        # Gewichtsmatrix
        self.weights = 0.1 * np.random.randn(n_neurons, n_inputs)
        # Bias-Vektor
        self.biases = 0.1 * np.random.randn(n_neurons, 1)

    def forward(self, inputs):
        """
        Berechnung des Ausgabewerts für die Neuronen in dieser Schicht
        basierend auf den Eingabewerte "inputs".
        """
        # Eingabewerte für spätere Verwendung speichern
        self.saved_inputs = inputs
        # Ausgabewerte als Matrix
        outputs = np.dot(self.weights, inputs) + self.biases
        return outputs  # Rückgabe der Ausgabewerte
```

Die Parameter „n_inputs“ und „n_neurons“ bestimmen die Struktur der Schicht: n_inputs gibt die Anzahl der Eingabewerte an, während n_neurons die Anzahl der Neuronen innerhalb der Schicht festlegt. Die Gewichtsmatrix „self.weights“ hat die Dimension $(n_{\text{neurons}} \times n_{\text{inputs}})$, während der Bias-Vektor „self.biases“ die Dimension $(n_{\text{neurons}} \times 1)$ besitzt und die Eingabematrix „inputs“ die Form $(n_{\text{inputs}} \times n_{\text{samples}})$ hat. Die „forward“-Methode berechnet die Matrix an Ausgabewerten „outputs“ mit einer Dimension von $(n_{\text{neurons}} \times n_{\text{samples}})$ und gibt diese anschließend aus.

Zur Veranschaulichung erstelle ich ein neuronales Netzwerk mit insgesamt drei Schichten: Eine Eingabeschicht mit zwei Neuronen, einer versteckten Schicht mit vier Neuronen und einer Ausgabeschicht mit fünf Neuronen:

```python
# Eingabeschicht mit 2 Neuronen → verborgenen Schicht mit 4 Neuronen
hidden_layer = Layer(2, 4)

# Verborgenen Schicht mit 4 Neuronen → Ausgabeschicht mit 5 Neuronen
output_layer = Layer(4, 5)

# Ausgabewerte für die verborgene Schicht
hidden_layer_outputs = hidden_layer.forward(inputs)

# Ausgabewerte für die Ausgabeschicht
output_layer_outputs = output_layer.forward(hidden_layer_outputs)
print(output_layer_outputs)

print(output_layer_outputs.shape)
```

Die Eingabeschicht wird durch die Eingabematrix „inputs“ mit der Dimension (2×4) dargestellt. Diese Werte dienen als Eingabe für die versteckte Schicht „hidden_layer“, die vier Neuronen besitzt und daher eine Ausgabematrix „hidden_layer_outputs“ mit der Dimension (4×4) bildet. Anschließend werden die Werte von der Ausgabeschicht „output_layer“ weiterverarbeitet, die fünf Neuronen besitzt und somit eine finale Ausgabenmatrix „output_layer_outputs“ mit der Dimension (5×4) erzeugt. Die Spalten dieser Matrix repräsentieren die Vorhersagen für die einzelnen Trainingsbeispiele, und die Zeilen die Vorhersage eines bestimmten Ausgabeneurons.

2.3 Regression vs. Klassifizierung

Für die späteren Kapitel ist es wichtig zwischen Regressions- und Klassifizierungsproblemen zu unterscheiden. Regression ist eine Supervised Learning-Methode. Sie wird verwendet, um kontinuierliche numerische Werte vorherzusagen. Dabei wird eine Beziehung zwischen Eingangsvariablen und Ausgabewerten hergestellt. Typische Anwendungen umfassen zum Beispiel die Vorhersage von Verkaufszahlen, Temperaturen oder Immobilienpreisen (vgl. Saxena, 2024).

Klassifizierung ist ebenfalls eine Supervised Learning-Methode, die darauf abzielt, Eingabedaten in diskrete Kategorien einzuteilen. Typische Anwendungen sind die Bilderkennung oder die Spam-Erkennung (ebd.).

2.4 Aktivierungsfunktionen

Ein neuronales Netzwerk ist im Wesentlichen eine Funktionsannäherung. Aktivierungsfunktionen ermöglichen es neuronalen Netzwerken, nicht-lineare Beziehungen zwischen Daten zu modellieren. Ein Neuron ohne Aktivierungsfunktion ist eine lineare Funktion. Besteht ein neuronales Netzwerk nur aus solchen Neuronen, dann kann dieses Netzwerk sich nur an lineare Funktionen annähern und besitzt somit nicht die Fähigkeit, komplexere Funktionen wie eine Sinusfunktion zu approximieren (vgl. Kinsley, 2020, 7:47).

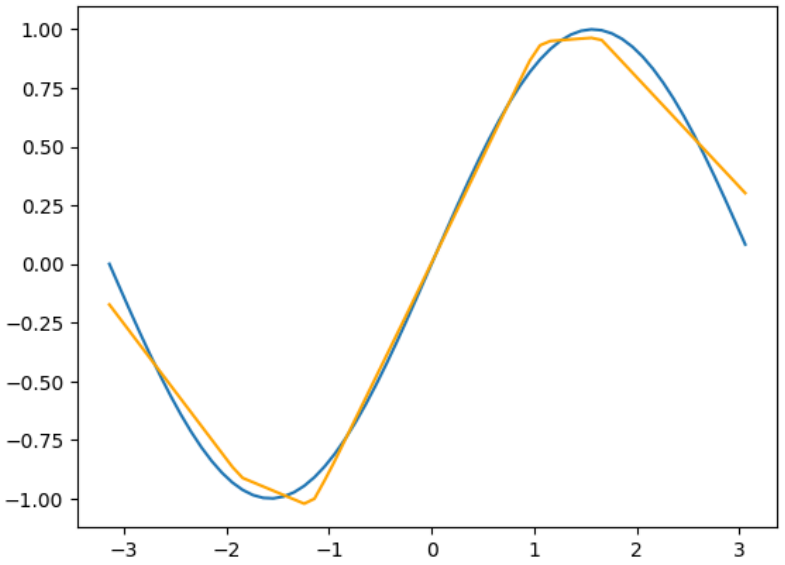

Abb. 2: Funktionsannäherung einer Sinuskurve (Verf.)

Um dieses Problem zu lösen, werden auf das Ergebnis der Neuronen Aktivierungsfunktionen angewendet. Es gibt verschiedene Arten von Aktivierungsfunktionen: Zwei weit verbreitete und beliebte sind die Sigmoid-Funktion und die ReLU-Funktion (Rectified Linear Unit Function) (vgl. Kinsley, 2020, TC: 7:52).

2.4.1 Die Sigmoid-Funktion

Die Sigmoid-Funktion ist eine mathematische Funktion, die den Wertebereich auf ein bestimmtes Intervall beschränkt und eine S-förmige Kurve bildet. Es gibt verschiedene Varianten der Sigmoid-Funktion. Eine davon ist die logistische Sigmoid-Funktion. Diese Funktion beschränkt den Wertebereich auf das Intervall zwischen null und eins. Im Kontext des maschinellen Lernens wird die logistische Sigmoid-Funktion oft einfach als „Sigmoid-Funktion" bezeichnet (vgl. Topper, 2023).

Die Sigmoid-Funktion wird berechnet, indem man eins durch eins plus die Eulersche Zahl e hoch dem negativen Wert dividiert (Kerremans, 2022). Mathematisch lässt sich die Funktion wie folgt darstellen:

$$a_{ji} = \sigma(z_{ji}) = \frac{1}{1 + e^{-z_{ji}}}$$

Dabei steht a_{ji} für den aktivierten Ausgabewert des i-ten Trainingsbeispiel und des j-ten Neurons und z_{ji} für den jeweiligen rohen Ausgabewert. Diese Operation wird hier elementweise für jedes Element in der Matrix $\mathbf{Z}$ angewendet. Um die Sigmoid-Funktion an Schichten von Neuronen anzuwenden, erstelle ich die Klasse „Sigmoid".

```
class Sigmoid:
    def forward(self, raw_outputs):
        """
        Berechnet die aktivierten Ausgabewerte basierend auf den rohen
        Ausgabewerten "raw_outputs".
        """
        activated_outputs = 1 / (1 + np.exp(-raw_outputs))
        # Aktivierte Ausgaben für spätere Verwendung speichern
        self.saved_activated_outputs = activated_outputs
        return activated_outputs
```

Ähnlich wie bei der Layer-Klasse enthält die Sigmoid-Klasse eine Methode namens „forward". Diese berechnet die Matrix aus aktivierten Ausgabewerten „activated_outputs" anhand der rohen Ausgabewerten „raw_outputs". Diese Matrix wird für spätere Verwendung gespeichert und anschließend ausgegeben.

Als Beispiel berechne ich die aktivierten Ausgaben eines neuronalen Netzwerks mit einer Eingabeschicht von zwei Neuronen und einer Ausgabeschicht von vier Neuronen:

```
output_layer = Layer(2, 4)
activation_function = Sigmoid()

raw_outputs = output_layer.forward(inputs)
activated_outputs = activation_function.forward(raw_outputs)
print(activated_outputs)
```

In diesem Beispiel wird die Methode „forward" der Klasse „Sigmoid" verwendet, um die aktivierten Ausgabewerte „activated_outputs" aus den rohen Ausgabewerten „raw_outputs" der Ausgabeschicht zu berechnen.

2.4.2 Die ReLU-Funktion

Eine weitere Aktivierungsfunktion ist die ReLU-Funktion. Der Vorteil der ReLU-Funktion gegenüber anderen Aktivierungsfunktion ist ihre Effizienz. Ihre Funktionsweise ist einfach: Ist ein Wert positiv, wird der Wert beibehalten, ansonsten wird der Wert gleich 0 gesetzt (vgl. Kinsley, 2020, TC: 9:00).

Die Formel dafür kann folgendermaßen dargestellt werden:

$$a_{ji} = \text{ReLU}(z_{ji}) = max(0, z_{ji})$$

Auch dafür erstelle ich eine Klasse:

```
class ReLU:
    def forward(self, raw_outputs):
        """
        Berechnet die aktivierten Ausgabewerte basierend auf den rohen
        Ausgabewerten "raw_outputs".
        """
        self.saved_raw_outputs = raw_outputs
        activated_outputs = np.maximum(0, raw_outputs)
        return activated_outputs
```

In diesem Codeblock werden die rohen Ausgaben „raw_outputs“ mit der Funktion „np.maximum“ in aktivierte Ausgaben „activated_outputs“ umgewandelt.

2.4.3 Die Softmax-Funktion

Die Softmax-Funktion ist eine weitere Aktivierungsfunktion, die aber in der Ausgabeschicht bei Klassifizierungsproblemen durchgeführt wird. Sie transformiert die Rohwerte in Wahrscheinlichkeiten, die zusammen 1 ergeben. Dies ermöglicht es, die Ausgaben des neuronalen Netzwerkes als Wahrscheinlichkeiten für die möglichen Klassen zu interpretieren (vgl. Belagatti, 2024).

Die Softmax-Funktion exponiert die Ausgaben mit Hilfe der exponentiellen Funktion e^y. Anschließend werden diese Werte normalisiert indem sie durch die Summe aller exponentielle Werte dividiert werden (ebd.). Die mathematische Formel sieht dann so aus:

$$a_{ji} = \text{Softmax}(z_{ji}) = \frac{e^{z_{ji}}}{\sum_{k=1}^{n_{\text{outputs}}} e^{z_{ki}}}$$

Wobei n_{outputs} für die Anzahl an Ausgabeneuronen steht.

So wie bei der Sigmoid-Funktion und der ReLU-Funktion erstelle ich auch für die Softmax-Funktion eine Klasse:

```
class Softmax:
    def forward(self, raw_outputs):
        """
        Berechnet die aktivierten Ausgabewerte basierend auf den rohen
        Ausgabewerten "raw_outputs".
        """
        # Exponierte Werte
        exponentiated_values = np.exp(raw_outputs - np.max(raw_outputs, \
                                                           axis=0))
        # Summe der exponierten Werte
        sum_values = np.sum(exponentiated_values, axis=0, keepdims=True)
        # Normalisierte / aktivierte Ausgaben
        normalized_outputs = exponentiated_values / sum_values
        return normalized_outputs
```

In diesem Abschnitt werden zuerst die exponentiellen Werte „exponentiated_values“ mit „np.exp“ berechnet. Anschließend werden diese Werte normalisiert, indem sie durch die Summe der exponierten Werte „sum_values“ dividiert werden und somit die aktivierten Ausgaben „normalized_outputs“ bilden.

2.5 Die Netzwerk-Klasse

Um die verschiedenen Komponenten des neuronalen Netzwerkes, wie die Schichten und Aktivierungsfunktionen, effizient zu verwalten, erstelle ich die Klasse „Network“. Diese Klasse besteht aus einer Liste von Schichten „self.layers“ und einer Liste von Aktivierungsfunktionen „self.activation_functions“. Zusätzlich enthält sie die Methode „forward_propagation“. Diese Methode basiert auf den Eingabewerten „inputs“ der Eingabeschicht und führt eine Forward Propagation durch. Dabei werden schichtweise die rohen Ausgaben jeder Schicht berechnet und dann mit der entsprechenden Aktivierungsfunktion aktiviert. Die endgültigen Ergebnisse in der letzten Schicht sind die Vorhersagen „predictions“ des neuronalen Netzwerkes und werden zurückgegeben. Mit der Methode „add_layer“ können Schichten und ihre jeweiligen Aktivierungsfunktionen zum Netzwerk hinzugefügt werden.

```
class Network:
    def __init__(
        self,
    ):
        self.layers = []
        self.activation_functions = []

    def add_layer(self, layer, activation_function):
        """
        Fügt eine instanzierte Schicht "layer" mit ihrer entsprechenden
        Aktivierungsfunktion "activation_function" zum Netzwerk hinzu.
        """
        self.layers.append(layer)
        self.activation_functions.append(activation_function)

    def forward_propagation(self, inputs):
        """
        Berechnet die Vorhersagen "predictions" des Netzwerkes anhand der
        Eingabewerte "inputs" der Eingabeschicht.
        """
        current_inputs = inputs
        for layer, activation_function in zip(self.layers, \
                                              self.activation_functions):
            raw_outputs = layer.forward(current_inputs)
            activated_outputs = activation_function.forward(raw_outputs)
            # Aktivierte Ausgaben der Schicht werden als Eingabewerte
            # für die nächste Schicht verwendet
            current_inputs = activated_outputs
        predictions = current_inputs
        return predictions
```

Als Beispiel erstelle ich ein Netzwerk, das aus zwei Eingabeneuronen, vier versteckten Neuronen und fünf Ausgabeneuronen besteht:

```
network = Network()
network.add_layer(
    Layer(2, 4),  # Eingabeschicht → versteckte Schicht
    ReLU(),  # Aktivierungsfunktion für die versteckte Schicht
)
network.add_layer(
    Layer(4, 5),  # Versteckte Schicht → Ausgabeschicht
    Softmax(),  # Aktivierungsfunktion für die Ausgabeschicht
)

network.forward_propagation(inputs)
```

2.6 Das Zahlenerkennungsmodell

Um ein neuronales Netzwerk zu trainieren, werden Daten benötigt. Für ein Modell, das zu der Erkennung von Zahlen dient, eignet sich die MNIST-Datenbank. MNIST enthält 60.000 Trainingsbilder und 10.000 Testbilder von handgeschriebenen Ziffern und kann somit zum Trainieren als auch für die Evaluierung verwendet werden (vgl. Khan, 2024).

Für mein neuronales Netzwerk verwende ich für die Eingabeschicht 784 Neuronen, da die MNIST Bilder aus 28 mal 28 Pixels bestehen. Ich habe eine verborgene Schicht mit 20 Neuronen mit der ReLU-Aktivierungsfunktion. Die Ausgabeschicht besteht aus zehn Neuronen, die jeweils die Ziffern null bis neun repräsentieren. Da es sich hier um ein Klassifizierungsproblem handelt, verwende ich für die Ausgabeschicht die Softmax-Funktion.

```
network = Network()
network.add_layer(
    Layer(784, 20),  # Eingabeschicht → versteckte Schicht
    ReLU(),  # Aktivierungsfunktion für die versteckte Schicht
)
network.add_layer(
    Layer(20, 10),  # Versteckte Schicht → Ausgabeschicht
    Softmax(),  # Aktivierungsfunktion für die Ausgabeschicht
)


def test_neural_network(network):
    # Bilder (Eingabewerte) und labels (tatsächliche Zielwerte als
    # Wahrscheinlichkeiten)
    images, labels = load_test_data()

    # Vorhersagen als Wahrscheinlichkeitsverteilung
    predictions = network.forward_propagation(images)

    N = predictions.shape[1]  # Anzahl an Trainingsbeispielen

    # Vorhersagen als Ziffern
    predicted_numbers = np.argmax(predictions, axis=0)

    # tatsächliche Zielwerte als Ziffern
    actual_values = np.argmax(labels, axis=0)

    # Vektor aus "Richtig Falsch" Werten
    comparisons = predicted_numbers == actual_values

    # Summe / Anzahl an richtigen Aussagen
    n_correct_predictions = sum(comparisons)

    # Genauigkeit des neuronalen Netzwerkes
    accuracy = n_correct_predictions / N

    print(accuracy)


test_neural_network(network)
```

Zuerst werden die Bilder „images“ und deren Beschriftungen „labels“ geladen. Anschließend berechnet das neuronale Netzwerk Vorhersagen „predictions“ basierend auf den Bilddaten und gibt diese als Wahrscheinlichkeitsverteilung zurück. Die Ziffer mit der höchsten Wahrscheinlichkeit wird dann mit der tatsächlichen Ziffer je Trainingsbeispiel verglichen. Um die Genauigkeit zu bestimmen, werden die richtigen Aussagen mit der gesamten Anzahl an Testbildern dividiert. Da das Netzwerk noch nicht trainiert wurde, ist die Genauigkeit sehr niedrig.

3 Trainieren eines neuronalen Netzwerkes

3.1 Deep und Shallow Learning

Deep Learning und Shallow Learning sind Teilbereiche des Machine Learning und befassen sich mit dem Trainieren neuronaler Netzwerke. Shallow Learning wird verwendet, um flache (shallow) neuronale Netzwerke zu trainieren, die in der Regel aus zwei oder drei Schichten bestehen. Deep Learning hingegen wird bei tiefen (deep) neuronalen Netzwerken angewendet, um Netzwerke mit mehr als zwei versteckten Schichten zu trainieren (vgl. Lodhi, o.J.).

Flache neuronale Netzwerke sind aufgrund ihrer vereinfachten Architektur schneller und einfacher zu trainieren. Allerdings eignen sie sich daher weniger gut für komplexe Probleme. Tiefe Netzwerke hingegen können durch ihre komplexe Struktur anspruchsvolle Probleme lösen, erfordern jedoch zusätzliche Methoden um Problemen wie Überanpassung zu vermeiden (ebd.).

Bei dem im vorherigen Kapitel angesprochenen Zahlenerkennungsmodell handelt es sich um ein flaches neuronales Netzwerk, da es nur eine versteckte Schicht besitzt.

3.2 Die Loss Function und die Cost Function

Die Begriffe Loss Function (Verlustfunktion) und Cost Function (Kostenfunktion) werden häufig synonym verwendet, haben jedoch grundlegend unterschiedliche Bedeutungen. Die Loss Function bewertet die Leistung einer einzelnen Vorhersage. Sie berechnet den Fehler des Netzwerks für ein einzelnes Trainingsbeispiel, indem sie die Vorhersage mit dem tatsächlichen Zielwert vergleicht (vgl. Alake, o.J.).

Im Gegensatz dazu ist die Cost Function der Mittelwert der Loss Function über das gesamte Trainingsset. Sie bewertet die Gesamtleistung des neuronalen Netzwerks und spielt eine zentrale Rolle im Trainingsprozess. Das Ziel des Netzwerks ist es, die Kosten zu minimieren, um die Genauigkeit der Vorhersagen zu maximieren (ebd.).

Es gibt verschiedene Arten von Cost Functions, die je nach Aufgabestellung in zwei Kategorien eingeteilt werden können: Cost Functions für Regressionsprobleme und Cost Functions für Klassifikationsprobleme (ebd.).

3.2.1 Cost Functions für Regressionsprobleme

Typische Cost Functions für Regressionsprobleme sind der Mean Absolute Error (MAE) und der Mean Squared Error (MSE). Der Mean Absolute Error berechnet den Mittelwert der absoluten Differenzen zwischen den Vorhersagen und den tatsächlichen Zielwerten (vgl. Alake, o.J.). Mathematisch wird sie so dargestellt:

$$MAE = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{n_{\text{outputs}}} |\hat{y}_{ji} - y_{ji}|$$

Hierbei steht N für die Anzahl an Trainingsbeispiel, $\hat{y}_{ji}$ steht für die Vorhersage, also die aktivierte Ausgabe a_{ji} des Neurons j in der Ausgabe Schicht für das i-te Trainingsbeispiel, und y_ji für den tatsächliche Zielwert des Neurons j beim i-ten Trainingsbeispiel.

Der Mean Squared Error hingegen berechnet die quadratischen Differenzen zwischen den Vorhersagen und den tatsächlichen Zielwerten. Durch das Quadrieren werden größere Differenzen stärker bestraft, was den MSE empfindlicher gegenüber Ausreißern macht (ebd.).:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{n_{\text{outputs}}} (\hat{y}_{ji} - y_{ji})^2$$

```
class MeanSquaredError:
    def calculate_cost(predictions, targets):
        losses = np.sum(np.square(predictions - targets), axis=0)
        cost = np.mean(losses)
        return cost
```

Für mein Programm werde ich nur den Mean Squared Error verwenden. Zuerst werden die Verluste „losses“ berechnet, indem ich die Vorhersagen „predictions“ von den Zielen „targets“ subtrahiere und dann die Ergebnisse mit „np.square“ quadriere. Da es mehrere Ausgabewerte pro Beispiel gibt, wird für jedes Beispiel der Verlust über alle Ausgabewerte summiert. Danach berechne ich die Kosten „cost“ indem ich den Mittelwert aller Verluste bilde und diesen ausgebe.

3.2.2 Cost Functions für Klassifizierungsprobleme

Eine typische Cost Function für Klassifizierungsprobleme ist der Categorical Cross Entropy (CCE). Um den Verlust L eines einzelnen Trainingsbeispiel i zu erhalten, wird die negative Summe aller tatsächlichen Zielwerte y_{ji} multipliziert mit den jeweiligen logarithmierten Vorhersagen $\hat{y}_{ji}$ gebildet (vgl. Gómez Bruballa, 2018):

$$L_i = -\sum_{j=1}^{n_{\text{outputs}}} y_{ji} \log(\hat{y}_{ji})$$

Um die Kosten für alle Trainingsbeispiele zu berechnen wird der Mittelwert aller Verluste wie bei den anderen Cost-Funktionen berechnet.

$$CCE \frac{1}{N} \sum_{i=1}^{N} L_i$$

```
class CategoricalCrossEntropy:
    def calculate_cost(predictions, targets):
        predictions = np.clip(predictions, 1e-7, 1 - 1e-7)
        losses = -np.sum(targets * np.log(predictions), axis=0)
        cost = np.mean(losses)
        return cost
```

Für mein Zahlenerkennungsmodell werde ich des Weiteren den Categorical Cross Entropy verwenden. Zuerst begrenze ich die Vorhersagen, damit die Werte nicht zu nah an null oder eins sind, um beim Logarithmieren Verzerrungen der Ergebnisse zu vermeiden. Danach berechne ich die Verluste „losses“ indem ich mit „np.log“ die Vorhersagen „predictions“ logarithmiere und dann mit den Zielen „targets“ multipliziere. Die Ergebnisse werden über alle Klassen mit „np.sum“ summiert. Um die Kosten „cost“ zu berechnen verwende ich wieder „np.mean“ um, den Mittellwert aller Verluste zu berechnen.

3.3 Die Objective Function

Die Objective Function (Zielfunktion) ist eine Funktion, die im Optimierungsprozess entweder minimiert oder maximiert wird. Die Rolle der Objective Function variiert je nach Bereich des Machine Learning. Im Reinforcement Learning zielt die Objective Function darauf ab, die kumulative Belohnung eines Agenten über eine Reihe von Aktionen zu maximieren (vgl. Muns, o.J.).

Im Deep Learning ist das Ziel der Objective Function, die Cost Function zu minimieren, indem die trainierbaren Parameter des neuralen Netzwerkes - also die Gewichte und Bias-Werte - angepasst werden (vgl. Dey, 2019).

Dieses Kapitel konzentriert sich auf die Objective Function im Kontext des Deep Learning, wobei die Objective Function in diesem Fall auch als die Cost Function bezeichnet werden kann.

Die Formel einer allgemeinen Cost Function kann wie folgt dargestellt werden:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i, \theta))$$

Dabei steht x_i für die Eingabewerte des i-ten Trainingsbeispiels und L für die Loss Function, die den Verlust zwischen dem tatsächlichen Zielwerten y_i und den Vorhersagen $f(x_i, \theta)$ eines neuronalen Netzwerks mit den Parametern θ angibt.

3.4 Gradient Descent

Gradient Descent ist ein Optimierungsalgorithmus. Er wird verwendet, um lokale Minima einer Funktion iterativ zu approximieren. Im Bereich des Machine Learning wird der Gradient Descent verwendet, um die Parameter eines neuronalen Netzwerks iterativ so anzupassen, dass die Cost Function minimiert wird. Der Algorithmus lässt sich wie ein Ball auf einer Landschaft mit Hügeln und Tälern darstellen, der schrittweise das Tal (das Minima) hinunterrollt, um den optimalen Punkt zu finden (vgl. Singh, 2025).

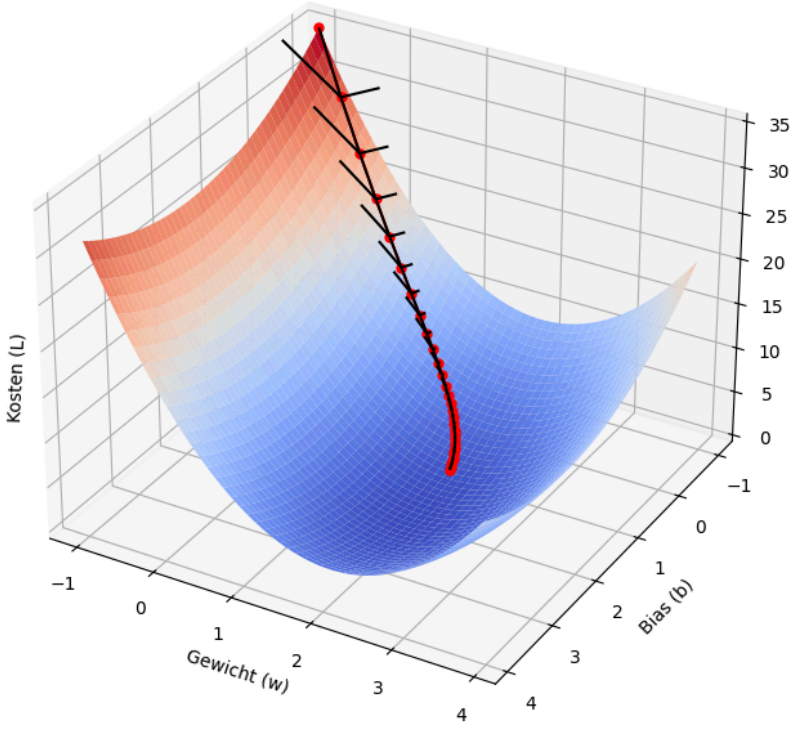

Abb. 3: Cost-Landschaft in Bezug auf Gewichte und Bias (Verf.)

Der Algorithmus berechnet zunächst den Gradienten. Der Gradient gibt an, in welcher Richtung und mit welcher Stärke die Funktion am stärksten steigt. Um also die Cost Function zu minimieren, werden die Parameter in Richtung des negativen Gradienten angepasst. Dazu werden die partiellen Ableitungen der Cost Function nach den Gewichten und Bias-Werten berechnet (ebd.).

Die Cost-Funktion misst den Fehler zwischen den Vorhersagen des Modells und den tatsächlichen Zielwerten. Da die Gewichte und Bias-Werte die Vorhersagen beeinflussen, haben sie demnach einen Einfluss auf die Cost Function. Allerdings tragen einige Parameter stärker zur Veränderung der Cost Function bei als andere. Deshalb werden die Parameter proportional zu ihrer Änderungsrate angepasst. Die partielle Ableitung zeigt an, wie empfindlich die Cost-Funktion auf Änderungen eines bestimmten Parameters reagiert. Gewichte, die eine große Änderung der Cost Function bewirken, werden entsprechend stärker angepasst, um den Fehler zu verringern, während Gewichte mit geringerem Einfluss nur minimal verändert werden (ebd.).

Ein weiterer wichtiger Bestandsteil von Gradient Descent ist die Lernrate (Learning Rate). Die Lernrate ist ein Hyperparameter, der die Schrittgröße in jeder Iteration des Gradient Descent Algorithmus bestimmt. Es ist entscheidend, eine geeignete Lernrate auszuwählen, um den Trainingsprozess effizient zu gestalten. Ist die Lernrate zu niedrig, verläuft der Lernprozess sehr langsam und benötigt viele Iterationen Eine zu hohe Lernrate hingegen kann dazu führen, dass der Algorithmus das Minimum überschreitet und nicht zu einer optimalen Lösung führt (vgl. Pabasara, 2024).

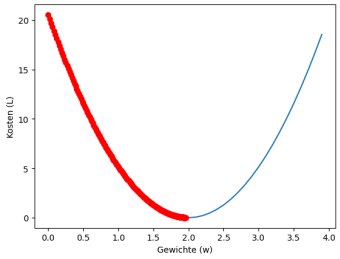

Abb. 4: Gradient Descent mit zu niedriger Lernrate (Verf.)

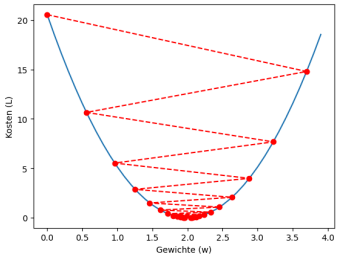

Abb. 5: Gradient Descent mit zu hoher Lernrate (Verf.)

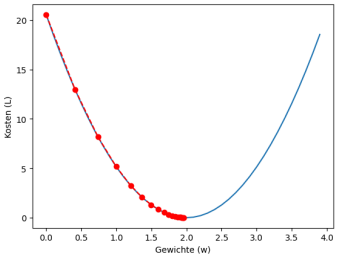

Abb. 6: Gradient Descent mit optimaler Lernrate (Verf.)

Nachdem der Gradient berechnet wurde, erfolgt die iterative Anpassung der trainierbaren Parameter des neuronalen Netzwerks. Dies geschieht, indem die Parameter um den Gradienten, multipliziert mit der Lernrate, verringert werden (vgl. Singh, 2025).

Die Formel lässt sich wie folgt darstellen:

$$\theta := \theta - \eta \nabla J(\theta)$$

Dabei steht θ für die Parameter des neuronalen Netzwerks, η für die Lernrate und $\nabla J(\theta)$ für den Gradienten der Cost-Funktion in Bezug auf alle trainierbaren Parameter θ. Um die Gewichtsmatrix $\mathbf{W}$ und den Bias-Vektor $\mathbf{b}$ einer beliebigen Schicht zu aktualisieren, kann demnach folgende Formel verwendet werden:

$$\mathbf{W}^l := \mathbf{W}^l - \eta \nabla_{\mathbf{W}^l} J$$

$$\mathbf{b}^l := \mathbf{b}^l - \eta \nabla_{\mathbf{b}^l} J(\mathbf{b}^l)$$

Hierbei gibt l an, um welche Schicht es sich handelt. $\mathbf{W}^l$ ist demnach die Gewichtsmatrix der Schicht l Weiterhin gibt $\nabla_{\mathbf{W}^l} J$ den Gradienten in Bezug auf die Gewichtsmatrix $\mathbf{W}^l$. Um Gradient Descent mit Python zu implementieren, erstelle ich die Klasse „GD“:

```
class GD:
    def __init__(self, network, learning_rate):
        """
        network: Das Netzwerk, das optimiert werden soll
        learning_rate: Die Lernrate, die die Schrittgröße bestimmt
        """
        self.network = network
        self.learning_rate = learning_rate

    def update_parameters(self):
        """
        Aktualisiert die Parameter (Gewichte und Bias-Werte) aller
        Schichten im Netzwerk basierend auf den Gradienten
        """
        # Iteriert über alle Schichten des Netzwerks und aktualisiert deren
        # Parameter
        for layer in self.network.layers:
            # Aktualisiert die Gewichte der aktuellen Schicht mit dem
            # negativen Gradienten multipliziert mit der Lernrate, um den
            # Schritt zu skalieren
            layer.weights -= self.learning_rate * layer.gradient_weights
            # Aktualisiert die Bias-Werte der aktuellen Schicht mit dem
            # negativen Gradienten multipliziert mit der Lernrate, um
            # sden Schritt zu skalieren
            layer.biases -= self.learning_rate * layer.gradient_biases
```

Im Konstruktor der Klasse speichere ich das neuronale Netzwerk „network“ und die Lernrate „learning_rate“. Momentan enthält die Klasse nur die Methode „update_parameters“. Diese Methode verwende ich, um die trainierbaren Parameter – also die Gewichte „layer.weights“ und die Bias-Werte „layer.biases“ – jeder Schicht im Netzwerk basierend auf den negativen Gradienten und der Lernrate zu aktualisieren. Der genaue Trainingsprozess und die Berechnung der Gradienten werden in den folgenden Kapiteln erklärt.

3.5 Backpropagation

3.5.1 Grundlagen

Backpropagation ist ein wichtiger Bestandteil im maschinellen Lernen und wird zusammen mit Optimierungsalgorithmen wie Gradient Descent verwendet, um die Gewichte und Bias-Werte eines neuronalen Netzwerks anzupassen und somit die Cost Function zu minimieren. Backpropagation nutzt Ableitungsregeln wie die Kettenregel, um den Gradienten der Cost Function effizient in Bezug auf alle Gewichte und Bias-Werte zu berechnen (vgl. Kostadinov, 2019).

Der Algorithmus kann in drei Schritte unterteilt werden: Der erste Teil wird als „Forward Pass" oder „Forward Propagation" bezeichnet und berechnet eine Vorhersage basierend auf gegebene Eingabedaten. Im zweiten Schritt werden mit Cost Functions die Vorhersagen mit den tatsächlichen Zielwerten verglichen und evaluiert. Der letzte Schritt ist der „Backwards Pass" und hier werden die berechneten Fehler bei der Evaluierung im Netzwerk schichtweise zurück propagiert. Dabei wird berechnet, wie sehr eine Schicht und ein Gewicht oder Bias-Wert zum Fehler beitragen (ebd.).

3.5.2 Grundidee

Angenommen, man betrachtet ein stark vereinfachtes neuronales Netzwerk mit jeweils nur einem Neuron pro Schicht. In diesem Fall besitzt jedes Neuron lediglich ein Gewicht und einen Bias-Wert. Die Verbindungen zwischen den Gewichten und Bias-Werten zur Cost Function lassen sich wie folgt darstellen:

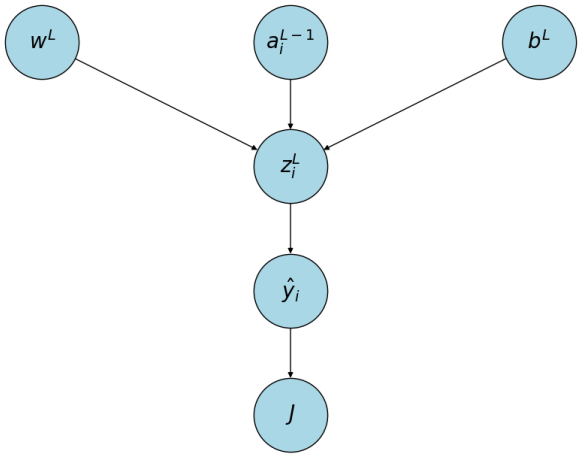

Abb. 7: Neuronales Netzwerk Diagram (Verf.)

Hierbei steht w für das Gewicht, b für den Bias-Wert, z für die rohe Ausgabe und a für die aktivierte Ausgabe. J repräsentiert die Cost Function, während L die Anzahl an Schichten beschreibt. Somit ist z_i^L der rohe Ausgabenwert der Ausgabechicht. $\hat{y}_i$ gibt die Vorhersage des neuronalen Netzwerkes an und i bezeichnet das Trainingsbeispiel.

In der gegebenen Darstellung beeinflusst das Gewicht w^L die rohe Ausgabe z_i^L, die durch die Aktivierungsfunktion transformiert wird, um die aktivierte Ausgabe der letzten Schicht a_i^L zu erzeugen. Diese aktivierte Ausgabe a_i^L stellt die Vorhersage y_i des neuronalen Netzwerks dar und hat somit Einfluss auf die Cost Function J. Um den Gradienten der Cost Function in Bezug auf die Gewichte und Bias-Werte zu berechnen, werde ich die Formeln rückwärts unter der Verwendung der Kettenregel propagieren. Zuerst gebe ich allerdings die partielle Ableitung der Cost Function auf die rohe Ausgabe an:

$$\frac{\partial J}{\partial z_i^L} = \frac{\partial J}{\partial \hat{y}_i^L} \frac{\partial \hat{y}_i}{\partial z_i^L}$$

Ein neuronales Netzwerk besitzt allerdings in der Regel mehrere Neuronen pro Schicht. Daher sieht die partielle Ableitung der Cost Function in Bezug auf die rohe Ausgabe z eines bestimmten Neurons in der Ausgabechicht L stattdessen so aus:

$$\frac{\partial J}{\partial z_{ji}^L} = \frac{\partial J}{\partial \hat{y}_{ji}} \frac{\partial \hat{y}_{ji}}{\partial z_{ji}^L}$$

Hierbei bezeichnet j ein beliebiges Neuron innerhalb der Schicht l, in dem Fall Ausgabeschicht L.

Die rohe Ausgabe wird durch die Gewichte, den aktivierten Ausgaben und den Bias-Wert bestimmt. Mit den folgenden Formeln kann dieser Einfluss bestimmt werden:

$$\frac{\partial J}{\partial w_{jk}^L} = \sum_{i=1}^{N} \frac{\partial z_{ji}^L}{\partial w_{jk}^L} \frac{\partial J}{\partial z_{ji}^L}$$

$$\frac{\partial J}{\partial b_j^L} = \sum_{i=1}^{N} \frac{\partial z_{ji}^L}{\partial b_j^L} \frac{\partial J}{\partial z_{ji}^L}$$

$$\frac{\partial J}{\partial a_{ki}^{L-1}} = \sum_{j=1}^{n_{\text{neurons}}^L} \frac{\partial z_{ji}^L}{\partial a_{ki}^{L-1}} \frac{\partial J}{\partial z_{ji}^L}$$

Dabei steht n_{neurons}^L für die Anzahl an Neuronen in der Ausgabeschicht L und k bezeichnet ein Neuron in der Schicht $l-1$.

Die Gewichte und Bias-Werte beeinflussen die rohen Ausgabe aller Trainingsbeispiele, daher werden diese auf i summiert. Die aktivierte Ausgabe a_{ki}^{L-1} hat nur Einfluss auf ein Trainingsbeispiel, beeinflusst allerdings alle Neuronen j, da es mit allen in der nächsten Schicht verbunden ist. Demnach wird auf j summiert.

Nun fehlt noch die partielle Ableitung der aktivierten Ausgabe zur rohen Ausgabe. Das kann so dargestellt werden:

$$\frac{\partial J}{\partial z_{ki}^{L-1}} = \frac{\partial a_{ki}^{L-1}}{\partial z_{ki}^{L-1}} \frac{\partial J}{\partial a_{ki}^{L-1}}$$

Und somit kann wieder auf die Gewichte und Bias-Werte der Schicht $L-1$ und den aktivierten Ausgaben der Schicht $L-2$ abgeleitet werden. Man propagiert sich rückwärts und berechnet die partiellen Ableitungen der Werte bis man die Gewichte und Bias-Werte der ersten Schicht abgeleitet hat.

3.5.3 Berechnung der Formeln für die Gradienten

Nun ist es wichtig, die exakten Formeln für die partiellen Ableitungen zu bestimmen, um den Gradienten der Gewichte und Bias-Werte zu berechnen. Angenommen, man besitzt die partielle Ableitung der Cost Function in Bezug auf die rohen Ausgabewerte z_{ji}^{l} der Schicht l

$$\frac{\partial J}{\partial z_{ji}^{l}}$$

Die Änderung der Cost Function in Bezug auf die rohen Ausgabewerte z_{ji}^{l} kann auch durch das Symbol δ_{ji}^{l} dargestellt werden. Das wird auch als „Error Signal" bezeichnet und gibt den Fehler eines Neurons j in der Schicht l an (vgl. Stansbury, 2020). Ich werde jedoch weiterhin die Notation $\frac{\partial J}{\partial z_{ji}^{l}}$ verwenden, um eine einheitliche Schreibweise beizubehalten und klarzustellen, dass es sich um die partielle Ableitung der Cost Function in Bezug auf die rohen Ausgabewerte handelt.

3.5.3.1 Gradient der Gewichte, der Bias-Werte und der aktivierten Ausgaben

Die Berechnung der rohen Ausgabewerte kann ähnlich wie im Kapitel „2.1 Grundlagen" beschrieben werden:

$$z_{ji}^{l} = \sum_{k=1}^{n_{\text{inputs}}^{l}} w_{jk}^{l} a_{ki}^{l-1} + b_{j}^{l}$$

Dabei ist der Eingabewert a_{k}^{l-1} eine aktivierte Ausgabe des k-ten Neuron von der vorherigen Schicht $l-1$. n_{inputs}^{l} steht hierbei für die Anzahl an Eingabewerten in der Schicht l und somit auch für die Anzahl an Neuronen in der Schicht $l-1$.

Basierend auf dieser Formel können die partiellen Ableitungen der Cost Function bezüglich der Gewichte, Bias-Werte und aktivierten Ausgaben bestimmt werden:

$$\frac{\partial J}{\partial w_{jk}^l} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_{ji}^l} a_{ki}^{l-1}$$

$$\frac{\partial J}{\partial b_{j}^l} = \sum_{i=1}^{N} \frac{\partial J}{\partial z_{ji}^l} 1$$

$$\frac{\partial J}{\partial a_{ji}^{l-1}} = \sum_{j=1}^{n_{\text{neurons}}^L} \frac{\partial J}{\partial z_{ji}^l} w_{jk}^l$$

Wie oben erklärt haben die Gewichte, die Bias-Werte und die aktivierten Ausgaben Einfluss auf entweder mehrere Trainingsbeispiele oder Neuronen und werden dementsprechend summiert. Die Änderungsrate des Gewichts ist abhängig von der aktivierten Ausgabe, deshalb wird bei der ersten Formel mit a_{ki}^{l-1} multipliziert. Das selbe gilt umgekehrt für die dritte Formel mit w_{jk}^l. Die Änderung des Bias-Wert auf die rohe Ausgabe ist allerdings eins.

Bisher habe ich mich allerdings nur mit den partiellen Ableitung auf bestimmte Parameter befasst. Der Gradient besteht aus mehreren dieser Parameter. Um einen Gradient für die Matrizen zu bilden, können folgende Formeln stattdessen verwendet werden. Die Formel für den Gradienten der Gewichtsmatrix und der aktivierten Ausgabe nutzt die Matrixmultiplikation:

$$\nabla_{\mathbf{W}^l} J = (\nabla_{\mathbf{Z}^l} J)(\mathbf{A}^{l-1})^T$$

$\mathbf{W}$ besitz die Form $(n_{\text{neurons}} \times n_{\text{inputs}})$ während $\nabla_{\mathbf{Z}^l} J$ die Dimension $(n_{\text{neurons}} \times N)$ und $\mathbf{A}^{l-1}$ die Dimension $(n_{\text{inputs}} \times N)$ besitz. Um die Matrixmultiplikation durchzuführen transponiere ich deshalb $\mathbf{A}^{l-1}$ und erhalte eine Matrix $(\mathbf{A}^{l-1})^T$ mit der Dimension $(N \times n_{\text{inputs}})$.

$$\nabla_{\mathbf{A}^{l-1}} J = (\mathbf{W}^l)^T (\nabla_{\mathbf{Z}^l} J)$$

Auch hier muss wieder transponiert werden um die Matrixmultiplikation durchzuführen. Die Matrix $\mathbf{W}^l$ wird dabei transponiert zur Matrix $(\mathbf{W}^l)^T$ mit der Dimension $(n_{\text{inputs}} \times n_{\text{neurons}})$

Letzendlich fehlt noch die Formel für den Gradienten der Cost-Function im Bezug zum Bias-Vektor $\mathbf{b}$:

$$(\nabla_{\mathbf{b}^l} J)_j = \sum_{i=1}^{N} (\nabla_{\mathbf{Z}^l} J)_{ji}$$

Hierbei werden die Spalten der Matrix $\nabla_{\mathbf{Z}^l} J$ mit der Form $(n_{\text{neurons}} \times N)$ summiert, wodurch ein Zeilenvektor mit der Dimension $(n_{\text{neurons}} \times 1)$ entsteht.

Basierend auf diesen Formeln erweitere ich die Klasse „Layer" um die Methode „backwards":

```
class Layer(Layer):
    def backwards(self, gradient_raw_outputs):
        """
        Berechnet den Gradienten der Cost Function in Bezug zu den
        Gewichten und Bias-Werten der aktuellen Schicht und aktivierten
        Ausgaben der vorherigen Schicht.

        gradient_raw_outputs: Gradient der Cost Function in Bezug zu den
        rohen Ausgaben der aktuellen Schicht (dJ/dZ).
        """

        # Gradient der Cost Function in Bezug zu den Gewichten von der
        # aktuellen Schicht (dJ/dW).
        self.gradient_weights = np.dot(
            gradient_raw_outputs,
            self.saved_inputs.T,
        )

        # Gradient in Bezug zu den Bias-Werten (dJ/db).
        self.gradient_biases = np.sum(gradient_raw_outputs, axis=1, \
                                      keepdims=True)

        # Gradient in Bezug zu den aktivierten Ausgaben der vorherigen
        # Schicht (dJ/dA).
        gradient_activated_outputs = np.dot(self.weights.T, \
                                            gradient_raw_outputs)
        return gradient_activated_outputs
```

Die Methode „backwards" verwendet den Parameter „gradient_raw_outputs", der den Gradienten der Cost Function in Bezug auf die rohen Ausgaben $\mathbf{Z}^l$ darstellt. Dieser Gradient wird genutzt, um die Gradienten der Cost Function in Bezug auf die Gewichte, Bias-Werte und aktivierten Ausgaben zu berechnen, wie in den Formeln beschrieben. Den Gradient in Bezug auf die aktivierten Ausgaben gebe ich für die Berechnung im kommenden Abschnitt aus.

3.5.3.2 Gradient der aktivierten Ausgaben

Nun werde ich die partiellen Ableitungen der Cost Function in Bezug auf die rohen Ausgaben erklären. Als Beispiel verwende ich hierbei die Sigmoid Activation Function und die ReLU Activation Function. Wie im Kapitel „2.4.1 Die Sigmoid-Funktion" und „2.4.2 ReLU Function" bereits erklärt, sehen diese Formeln dafür so aus:

$$a_{ji}^{l} = \sigma(z_{ji}^{l}) = \frac{1}{1 + e^{-z_{ji}^{l}}}$$

$$a_{ji}^{l} = \mathrm{ReLU}(z_{ji}^{l}) = max(0, z_{ji}^{l})$$

Die Sigmoid-Funktion leite ich mithilfe von Geogebra auf folgende Formel ab:

$$\frac{da_{ji}{}^{l}}{dz_{ji}^{l}} = \sigma(z_{ji}^{l})(1 - \sigma(z_{ji}^{l}))$$

Bei der ReLU Function werden rohen Ausgaben zu null wenn sie negativ sind und bleiben unverändert wenn sie positiv sind. Die Änderungsrate ist also null, wenn die Eingabe negativ ist, und eins, wenn sie positiv ist. Das kann mit folgender Formel dargestellt werden:

$$\frac{da_{ji}^{l}}{dz_{ji}^{l}} = \frac{dmax(z_{ji}^{l}, 0)}{dz_{ji}^{l}} = \begin{cases} 1 & \text{if } z_{ji}^{l} > 0 \\ 0 & \text{otherwise} \end{cases} = (z_{ji} > 0)$$

Nachdem die Ableitung der aktivierten Ausgaben auf die rohen Ausgaben bestimmt wurde, lässt sich mithilfe der partiellen Ableitung der Cost Function in Bezug auf die aktivierten Ausgaben der Gradient der Cost Function in Bezug auf die rohen Ausgaben berechnen. Die Berechnung dafür stelle ich mit folgenden Formeln dar:

$$\nabla_{\mathbf{Z}^l} J = (\nabla_{\mathbf{A}^l} J) \odot \mathbf{A}^l \odot (1 - \mathbf{A}^l)$$

$$\nabla_{\mathbf{Z}^l} J = (\nabla_{\mathbf{A}^l} J) \odot (\mathbf{Z}^l > 0)$$

Hierbei steht $\odot$ für die Rechenoperation „Hadamard Product“. Diese ist ähnlich wie die Matrixaddition, mit dem Unterschied, dass Elemente einer bestimmten Spalte und einer bestimmte Reihe elementweise multipliziert und nicht addiert werden (vgl. Pal, 2019). Zur Veranschaulichung wird das mit einem Beispiel illustriert:

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix} \odot \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{bmatrix} = \begin{bmatrix} a_{1,1}b_{1,1} & a_{1,2}b_{1,2} \\ a_{2,1}b_{2,1} & a_{2,2}b_{2,2} \\ a_{3,1}b_{3,1} & a_{3,2}b_{3,2} \end{bmatrix}$$

Basierend auf den Code erweitere ich die Klasse "Sigmoid" und die Klasse "ReLU" beide mit der Methode "backwards":

```
class Sigmoid(Sigmoid):
    def backwards(self, gradient_activated_outputs):
        """
        Berechnet den Gradienten der Cost Function in Bezug zu
        den rohen Ausgaben der aktuellen Schicht (dJ/dZ)

        gradient_activated_outputs: Gradient der Cost Function in Bezug
        zu den aktivierten Ausgaben der aktuellen Schicht (dJ/dA)
        """
        # Gradient in Bezug zu (A*(A-1))
        d_activated_d_raw = self.saved_activated_outputs * (
            1 - self.saved_activated_outputs
        )

        gradient_raw_outputs = gradient_activated_outputs * d_activated_d_raw
        return gradient_raw_outputs


class ReLU(ReLU):
    def backwards(self, gradient_activated_outputs):
        """
        Berechnet den Gradienten der Cost Function in Bezug zu
        den rohen Ausgaben der aktuellen Schicht (dJ/dZ).

        gradient_activated_outputs: Gradient der Cost Function in Bezug
        zu den aktivierten Ausgaben der aktuellen Schicht (dJ/dA).
        """
        # Gradient der Cost Function in Bezug zu den rohen Ausgaben (dJ/dZ).
        gradient_raw_outputs = gradient_activated_outputs * \
            (self.saved_raw_outputs > 0)
        return gradient_raw_outputs
```

3.5.3.3 Gradient der Vorhersagen

Als nächstes bestimme ich den Gradienten der Cost Function in Bezug auf die Vorhersagen $\hat{\mathbf{Y}}$. Dabei steht $\hat{\mathbf{Y}}$ gleichzeitig für die Matrix der aktivierten Ausgaben $\mathbf{A}$ der letzten Schicht L. Vorerst verwende ich den Mean Squared Error als Cost Function. Die Formel dafür ist wie im Kapitel „3.4.1 Cost Functions für Regressionsprobleme“ beschrieben:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{n_{\text{outputs}}}(\hat{y}_{ji} - y_{ji})^2$$

Die partielle Ableitung dieser Cost Function in Bezug zu der Vorhersage $\hat{y}_{ji}$ kann durch folgende Rechenoperationen beschrieben werden:

$$\frac{\partial J}{\partial \hat{y}_{ji}} = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{n_{\text{outputs}}} 2(\hat{y}_{ji} - y_{ji}) = \frac{2}{N}(\hat{y}_{ji} - y_{ji})$$

Da es sich um den vorhergesagten Wert $\hat{y}_{ji}$ eines einzelnen Neurons j für das Trainingsbeispiel i handelt, fallen beiden Summenoperatoren weg. Um nun den Gradienten des Mean Squared Error in Bezug auf alle Vorhersagen $\hat{\mathbf{Y}}$ zu berechnen, kann folgende Formel verwendet werden:

$$\nabla_{\hat{\mathbf{Y}}} J = \frac{2}{N}(\hat{\mathbf{Y}} - \mathbf{Y})$$

Hierbei beschreibt $\hat{\mathbf{Y}}$ die Matrix der Vorhersagen und $\mathbf{Y}$ die Matrix der tatsächlichen Zielwerten. Beide Matrizen besitzen die Dimension $(n_{\text{outputs}} \times N)$. Die resultierende Gradientmatrix $\nabla_{\hat{\mathbf{Y}}} J$ besitz ebenfalls die gleiche Dimension.

Um die Berechnung umzuseten, erweitere ich die Klasse „MeanSquaredError“ ebenso um die Methode „backwards“:

```
class MeanSquaredError(MeanSquaredError):
    def backwards(predictions, targets):
        """
        Berechnet den Gradienten des Mean Squared Error in Bezug zu den
        Vorhersagen.
        """
        N = predictions.shape[1]  # Anzahl an Trainingsbeispielen
        gradient_predictions = (2 / N) * (predictions - targets) / \
            len(predictions)
        return gradient_predictions
```

3.5.3.4 Categorical Cross Entropy + Softmax

Anschließend fehlt noch die Ableitungen für den Categorical Cross Entropy und für die Softmax Activcation Function. Die partielle Ableitung des Categorical Cross Entropy auf die rohe Ausgabe z_{ji} kann mithilfe der Kettenregel folgendermaßen dargestellt werden:

$$\frac{\partial J}{\partial z_{ji}} = \frac{\partial \hat{y}_{ji}}{\partial z_{ji}} \frac{\partial J}{\partial \hat{y}_{ji}}$$

Anstatt die partiellen Ableitungen einzeln auszurechnen, können diese allerdings kombiniert werden um die Berechnungen effizienter zu machen. Die partielle Ableitung der Loss Function L für das Trainingsbeispiel i in Bezug zu dem rohen Ausgabewert z^l_{ji} wird berechnet, in dem der vorhergesagte Wert $\hat{y}_{ij}$ mit dem jeweiligen tatsächlichen Zielwert y_{ij} subtrahiert wird (vgl. Kurbiel, 2021).

Mathematisch kann dies folgendermaßen dargestellt werden:

$$\frac{\partial L_i}{\partial z^l_{ji}} = \hat{y}_{ji} - y_{ji}$$

Hierbei handelt es sich allerdings um die partielle Ableitung der Loss Function L_i für ein beliebiges Trainingsbeispiel i. Die partielle Ableitung der Cost Function J für den gesamten Trainingsset auf die rohen Ausgaben sieht hierbei stattdessen so aus:

$$\frac{\partial J}{\partial z_{ij}} = \frac{1}{N} \sum_{i=1}^{N} \hat{y}_{ij} - y_{ij} = \hat{y}_{ij} - y_{ij}$$

Ähnlich wie bei der Ableitung des Mean Squared Error fällt hier der Summenoperator weg. Die Berechnung des Gradienten $\mathbf{Z}$ in Bezug zu der Cost Function kann man demnach so darstellen:

$$\nabla_{\mathbf{Z}} J = \frac{1}{N}(\hat{\mathbf{Y}} - \mathbf{Y})$$

Anders als bei dem Categorical Cross Entropy handelt es sich hierbei nicht um den Gradienten im Bezug auf $\mathbf{A}$ sondern um den Gradienten im Bezug auf $\mathbf{Z}$

Die Formeln setze ich nun mit Python um in dem ich wie zuvor die Klassen „CategoricalCrossEntropy“ und „Softmax“ mit der Methode „backwards“ erweitere:

```
class CategoricalCrossEntropy(CategoricalCrossEntropy):
    def backwards(predictions, targets):
        """
        Berechnet den Gradienten des Categorical Cross Entropy in
        Bezug zu den rohen Ausgaben der Ausgabenschicht (dJ/dZ).
        """
        N = predictions.shape[1]  # Anzahl an Trainingsbeispielen
        gradient_raw_outputs = (predictions - targets) / N
        return gradient_raw_outputs


class Softmax(Softmax):
    def backwards(self, gradient_raw_outputs):
        # Gibt die Gradienten direkt weiter (Softmax wird in Kombination
        # mit Categorical Cross Entropy verwendet).
        return gradient_raw_outputs
```

„CategoricalCrossEntropy.backwards“ berechnet hierbei den Gradienten in Bezug zu den rohen Ausgaben „gradient_raw_outputs“ und gibt diesen wieder. Da diese Berechnung mit Softmax kombiniert wurde, gibt „Softmax.backwards“ diesen Gradienten ohne Modifizierung direkt aus.

3.5.4 Die Backpropagation-Methode

Um die Backpropagation durchzuführen, erstelle ich für die Klasse „Netzwerk“ eine neue Methode namens „backpropagation“.

```
class Network(Network):
    def __init__(self, cost_function):
        self.layers = []
        self.activation_functions = []
        self.cost_function = cost_function

    def backpropagation(self, predictions, targets):
        # Gradient der Cost Function in Bezug zu den Vorhersagen (dJ/dY).
        # Beim Categorical Cross Entropy + Softmax sind diese allerdings
        # im Bezug zu den rohen Ausgaben der Ausgabenschicht (dJ/dZ).
        gradient_predictions = self.cost_function.backwards(predictions, \
                                                            targets)
        # Der Gradient der Vorhersagen ist identisch mit dem Gradient der
        # aktivierten Ausgaben der Ausgabenschicht
        gradient_activated_outputs = gradient_predictions
        # Rückwärts berechnet, von Ausgabeschicht zu Eingabeschicht.
        for layer, activation_function in zip(
            reversed(self.layers), reversed(self.activation_functions)
        ):
            # Gradient der Cost Function in Bezug zu den aktivierten Ausgaben
            # der aktuellen Schicht(dJ/dA).
            gradient_raw_outputs = activation_function.backwards(
                gradient_activated_outputs
            )

            # Gradienten der Cost Function in Bezug zu den Gewichten (dJ/dW)
            # und den Bias-Werten der aktuellen Schicht (dJ/db).
            # Berechnet zusätzlich den Gradienten der Cost Function in
            # Bezug zu den rohen Ausgaben der vorherigen Schicht(dJ/dZ).
            gradient_activated_outputs = layer.backwards(gradient_raw_outputs)
```

Diese Methode berechnet zuerst den Gradienten der Vorhersagen, oder rohen Ausgabewerte der letzten Schicht im Falle von Categorical Cross Entropy, „gradient_activated_outputs“ und berechnet dann schichtweise rückwärts den Gradient der rohen Ausgaben mit „aktivierung.backwards(gradient)“ und den Gradient der Gewichte, Bias-Werte und aktivierten Ausgaben mit „schicht.rueckwaerts(gradient)“. Nachdem die Gradienten berechnet wurden, kann die Gewicht- und Bias-Werte-Aktualisierung durchgeführt werden, wie beschrieben in Kapitel „3.4 Gradient Descent“

3.6 Stochastic Gradient Descent

Es gibt verschiedene Varianten von Gradient Descent. Diese haben ihre Vor- und Nachteile. Die bisher besprochene Variante ist auch als „Batch Gradient Descent“ oder „Vanilla Gradient Descent“ bekannt und berechnet den Gradient im Bezug auf das gesamte Trainingsset (vgl. Roy, 2024).

Dies ist allerdings sehr ineffizient. Eine Alternative, um dieses Problem zu lösen, ist unter dem Namen „Stochastic Gradient Descent“ (SGD) bekannt. Anstelle den Gradienten für das gesamte Trainingsset zu berechnen, wird hierbei stattdessen der Gradient für ein einzelnes Trainingsbeispiel oder für ein Batch von Trainingsbeispielen berechnet. Dadurch wird zwar Zufälligkeit eingeführt und es werden mehr Iterationen benötigt als bei Gradient Descent, jedoch verlaufen die einzelnen Durchläufe deutlich schneller (ebd.).

Es gibt zwar verschiedene Gradient Descent-Varianten, allerdings entscheide ich mich trotzdem für Stochastic Gradient Descent, da es vielversprechende Ergebnisse für mein Zahlenerkennungsmodell liefert. Um SGD zu implementieren, erstelle ich eine Klasse namens „SGD“, die von der Gradient Descent-Klasse „GD“ erweitert wird:

```
class SGD(GD):
    def create_batches(self, inputs, targets, batch_size):
        N = inputs.shape[1]  # Anzahl an Trainingsdaten
        # Trainingsset in mini Batches eingeteilt
        batches = []
        for i in range(0, N, batch_size):
            batch_inputs = inputs[:, i : i + batch_size]
            batch_targets = targets[:, i : i + batch_size]
            single_batch = (batch_inputs, batch_targets)
            batches.append(single_batch)
        return batches

    def optimise(
        self,
        inputs,
        targets,
    ):
        # Vorwärtsdurchlauf: Berechnung der Vorhersagen
        predictions = self.network.forward_propagation(inputs)

        cost = self.network.cost_function.calculate_cost(predictions, targets)

        # Rückwärtsdurchlauf: Berechnung der Gradienten
        self.network.backpropagation(predictions, targets)

        # Aktualisiert die Gewichte und Bias-Werte basierend auf
        # die Gradienten
        self.update_parameters()
        return cost

    def train(self, inputs, targets, batch_size):
        batches = self.create_batches(inputs, targets, batch_size)
        cost_history = []
        avg_epoch_cost = 1
        while avg_epoch_cost > 0.2:
            epoch_cost_history = []
            for batch_inputs, batch_targets in batches:
                cost = self.optimise(batch_inputs, batch_targets)
                self.optimise(batch_inputs, batch_targets)

                epoch_cost_history.append(cost)
                cost_history.append(cost)
```

```
            avg_epoch_cost = np.mean(epoch_cost_history)

        return cost_history
```

Die Methode „create_batches“ teilt das Trainingsset in Batches ein, die jeweils eine Batchgröße von „batch_size“ besitzen. In der Methode „optimise“ wird zuerst die Forward Propagation durchgeführt und anschließend die Backpropagation, um die Gradienten für die Gewichte und Bias-Werte zu berechnen. Danach werden die Gewichte und Bias-Werte mit den Gradienten durch die Methode „update_parameters“ aktualisiert. Dies stellt eine einzelne Iteration dar. In der Methode „train“ wird diese Aktualisierung für jede Batch durchgeführt. Eine Epoche ist erfolgreich durchgeführt wenn der Prozess für alle Batches durchgeführt wurde. Der Trainingsprozess erstreckt sich für so viele Epochen, bis die mittleren Kosten eine gewisste Grenze, in dem Fall 0.2, unterschritten haben.

3.7 Trainieren eines Zahlenerkennungsmodells

Mit Hilfe der Techniken die in diesem Kapitel besprochen wurden, kann ich das Zahlenerkennungsmodell nun trainieren. Der Code sieht folgendermaßen aus:

```
# Categorical Cross Entropy als Cost Function
network = Network(CategoricalCrossEntropy)
network.add_layer(
    Layer(784, 20),  # Eingabeschicht → versteckte Schicht
    ReLU(),  # Aktivierungsfunktion für die versteckte Schicht
)
network.add_layer(
    Layer(20, 10),  # Versteckte Schicht → Ausgabeschicht
    Softmax(),  # Aktivierungsfunktion für die Ausgabeschicht
)

training_images, training_labels = load_trainings_data()

sgd = SGD(network, learning_rate=0.1)
cost_history = sgd.train(
    training_images,
    training_labels,
    batch_size=64,
)

test_neural_network(network)
```

Da es sich um ein Klassifizierungsproblem handelt, verwende ich den Categorical Cross Entropy als meine Cost Function. Der Aufbau für das Netzwerk ist identisch mit dem Modell aus Kapitel „3.6 Das Zahlenerkennungsmodell“. Nachdem ich das Netzwerk instanziert habe, lade ich die Trainingsdaten „training_images“ mit den entsprechenden Zahlenbeschriftungen „training_labels“ der MNIST Bibliothek. Anschließend wird das Netzwerk mit einer Lernrate von 0.1 und einer Batchgröße von 64 trainiert. Das Netzwerk kann anschließend durch die Methode „test_neural_network“ mit den Testdaten getestet werden. Die Leistung des Netzwerks pro Iteration sieht vereinfachert so aus:

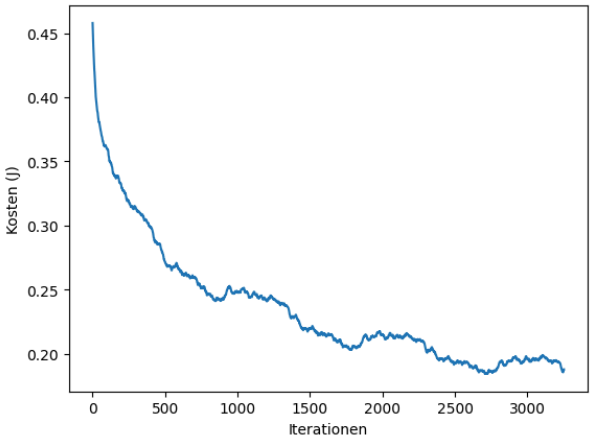

Abb. 8: Kosten-Graph des Zahlenerkennungsmodell mit Stochastic Gradient Descent (Verf.)

Wie man anhand der Abbildung erkennen kann, springt die Leistung des Netzwerks, da sich das Netzwerk jeweils an ein Batch des Trainingssets anpasst. Allerdings erreicht es bereits nach wenigen Sekunden Trainingszeit eine Genauigkeit von über 90 Prozent.

3.8 Das fertige Zahlenerkennungsmodell

Der Quellcode des Zahlenerkennungsmodells befindet sich einerseits im Anhang dieser Arbeit, ist jedoch auch online über das folgende GitHub-Repository verfügbar: https://github.com/scrythe/it_learns_numbers.

Zum Testen des Modells steht außerdem eine interaktive Webanwendung unter diesem Link zur Verfügung: https://it-learns-numbers.streamlit.app. Zunächst lädt man ein Bild mit handschriftlichen Zahlen hochladen und grenzt anschließend die gewünschte Zielziffer ein. Das Modell gibt daraufhin eine Wahrscheinlichkeitstabelle für alle möglichen Ziffern zurück. Dabei ist zu beachten, dass der Text (also die Ziffern) dunkler als der Hintergrund sein muss, da das Modell auf diese Darstellung hin trainiert wurde.

Die Vorhersagen des Modells sind nicht immer korrekt. Das liegt daran, dass das Trainingsdatenset, auf das es trainiert wurde, nicht alle möglichen Schreibweisen oder reale Anwendungsfälle vollständig abdeckt und daher eine gewisse Varianz in den Ergebnissen auftreten kann.

4 Genetische Algorithmen

4.1 Grundlagen

Genetische Algorithmen sind eine Art von Optimierungsalgorithmen, die mit dem Prozess der Evolution vergleichbar sind. Genetische Algorithmen werden verwendet, um mithilfe von biologischen Prozessen wie Reproduktion und natürlicher Selektion Lösungen für Probleme zu finden (vgl. Kanade, o.J.).

Genetische Algorithmen eignen sich hervorragend für Probleme, bei denen aus einer großen Anzahl von Möglichkeiten Lösungen gefunden werden müssen. Außerdem können sie für die Lösung von kombinatorischen Problemen, bei denen eine optimale Anordnung von Elementen in einer begrenzten Liste gesucht wird, verwendet werden (ebd.).

Eine einfache Anwendung für genetische Algorithmen ist das „Knapsack"-Problem. Bei diesem Problem ist ein Rucksack gegeben, in den man Gegenstände hineinlegen kann, die jeweils ein Gewicht und einen Geldwert besitzen. Ziel ist, dass der Rucksack eine möglichst hohe Summe an Geldwerten enthält, die nicht die Gewichtgrenze überschreitet (vgl. Bhayani, o.J.).

Um die Funktionsweise von genetischen Algorithmen zu illustrieren, verwende ich die Programmiersprache Python, um das „Knapsack"-Problem zu lösen.

4.2 Population

Ein wichtiger Bestandteil von genetischen Algorithmen ist das Konzept einer Population, die eine Kollektion von Individuen darstellt. Ein Individuum repräsentiert dabei eine mögliche Lösung zu einem Problem (vgl. Kanade, o.J.).

```python
class Item:
    def __init__(self, name, mass, value):
        self.name = name
        self.mass = mass
        self.value = value


item_list = [
    # Handy mit 3kg Masse und einem Geldwert von 5€
    Item("Handy", mass=3, value=5),
    Item("Laptop", 6, 10),
    Item("Diamant", 1, 30),
    Item("Brot", 1, 1),
]


class Individual:
    def __init__(self, item_bits):
        self.item_bits = item_bits

    def print_items_in_backpack(self):
        for item, is_in_backpack in zip(item_list, self.item_bits):
            if is_in_backpack:
                print(
                    f"Gegenstand: {item.name}, Masse: {item.mass}kg, \
                    Geldwert: {item.value}€"
                )


individual = Individual([0, 1, 0, 1])

individual.print_items_in_backpack()
```

Die Repräsentation eines Individuums stelle ich mit einer Liste von binären Zahlen dar. Ist die Zahl 0, dann ist der Gegenstand nicht im Rucksack. Ist die Zahl 1, dann ist er schon im Rucksack. Die Position des Gegenstandes sagt aus, um welchen Gegenstand es sich handelt.

Zur Modellierung eines eines Individuums erstelle ich die Klasse „Individual“, deren Attribut „item_bits“ diese Liste speichert. Die verfügbaren Gegenstände werden durch die Klasse „Item“ erstellt. Jedes Objekt dieser Klasse beinhaltet einen Namen, eine Masse und einen Geldwert. Die Liste „item_list“ enthält alle verfügbaren Gegenstände. Der erste Gegenstand in dieser Liste ist zum Beispiel ein Handy mit einer Masse von 3kg und einem Wert von €5.

Als Beispiel verwende ich das Individuum [0, 1, 0, 1]. Das bedeutet, dasss der Rucksack den Laptop und das Brot enthält. Das wird durch die Methode „print_items_in_backpack“ veranschaulicht.

Der Genetische Algorithmus startet mit einer initialen Population. Diese Population wird zufällig generiert und bildet durch Operatoren wie Selektion, Crossover und Mutation die Population in der nächsten Generation. Dieser Vorgang wird iterativ durchgeführt, um dann zu einer optimalen sowie effektiven Lösung zu kommen. Diese Operatoren spiegeln Prozesse wie natürliche Selektion, Reproduktion und genetische Variation in der Natur wider (vgl. Kanade, o.J.).

```
import random


class Individual(Individual):
    def create_random_individual():
        random_item_bits = []
        for _ in item_list:  # für jedes Element in der Gegenstände Liste
            bit = random.choice([0, 1])  # zufällig 1 oder 0 wählen
            random_item_bits.append(bit)

        return Individual(random_item_bits)


class Population:
    def __init__(self, population_size):
        self.population_size = population_size
        self.create_initial_population()

    def create_initial_population(self):
        self.population = []
        while population_size > len(self.population):
            individual = Individual.create_random_individual()
            self.population.append(individual)

    def print_population(self):
        for individual in self.population:
            print(individual.item_bits)


population_size = 8
population = Population(population_size)
population.print_population()
```

In diesem Code-Ausschnitt erzeuge ich eine Population von Individuen. Eine Population wird mit der Methode „create_initial_population“ in der Klasse „Population“ erstellt. Dabei werden Individuen mit einer zufälligen Sequenz von 0- und 1-Bits erzeugt, die die Gegenstände im Rucksack darstellen. Dies geschieht durch die statische Methode „create_random_individual“ in der Klasse „Individual“. Die Anzahl der Individuen in der Population wird durch das Attribut „population_size“ festgelegt.

4.3 Fitness-Funktion

Die „Fitness"-Funktion evaluiert, wie „fit" ein Individuum oder wie gut eine mögliche Lösung in der Population ist. Um eine effektive Lösung zu einem Problem zu finden, ist es sehr wichtig, eine gute Fitness-Funktion zu kreieren. Eine schlechte Fitness-Funktion kann potenziell gute Lösungen als schlecht bewerten und schlechte Lösungen als gut und führt somit zu einer nicht effektiven Lösung für ein bestimmtes Problem (vgl. Bhayani, o.J.).

```
class Individual(Individual):
    def calculate_fitness(self, mass_limit):
        total_mass = 0
        total_value = 0
        # Gehe jeden Gegenstand des Individuums durch
        for item, is_in_backpack in zip(item_list, self.item_bits):
            if is_in_backpack:
                total_mass += item.mass
                total_value += item.value

        if total_mass > mass_limit:
            self.fitness_value = 0
            return

        self.fitness_value = total_value


class Population(Population):
    def calculate_fitness(self, mass_limit):
        for individual in self.population:
            individual.calculate_fitness(mass_limit)


mass_limit = 5
population = Population(population_size)
population.calculate_fitness(mass_limit)
```

Um den Fitness-Wert in meinem Programm zu berechnen, verwende ich die Methode „calculate_fitness". Der Fitness-Wert eines Individuum entspricht der Summe aller Geldwerte der Gegenstände, die sich im Rucksack befinden. Überschreitet allerdings das Gesamtgewicht der ausgewählten Gegenstände das Massenlimit („mass_limit"), so hat ein Individuum einen Fitness-Wert von 0.

4.4 Genetische Operatoren

4.4.1 Selektion

Um die Population der nächsten Generation zu bilden, werden Individuen aus der aktuellen Population genommen. Diese Individuen werden reproduziert, um dann Nachkommen zu generieren. Grundsätzlich sollen die besseren Individuen in die nächste Generation übergehen, in der Hoffnung, dass ihre Nachkommen noch besser werden. Es gibt mehrere Methoden, um diese Selektion durchzuführen. Eine Methode ist die „Tournament-Selektion". Bei dieser Methode werden zwei Individuen zufällig ausgewählt und miteinander verglichen. Das Individuum mit dem höheren Fitness-Wert wird dann als Elternteil für die nachkommende Generation bestimmt (vgl. Bhayani, o.J.).

```
def tournament(enemy1, enemy2):
    if enemy1.fitness_value > enemy2.fitness_value:
        return enemy1
    else:
        return enemy2


def selection(population):
    enemies = random.sample(population, 4)  # 4 zufällige Individuuen
    winner1 = tournament(enemies[0], enemies[1])
    winner2 = tournament(enemies[2], enemies[3])
    return [winner1, winner2]


selection(population.population)
```

Bei meinem Programm werden vier Individuen aus der aktuellen Population zufällig ausgewählt. Die zwei Gewinner gehen zur nächsten Operation über. Der jeweilige Gewinner ist das Individuum, das den höheren Fitness-Wert besitzt.

4.4.2 Kreuzung

Kreuzung („Crossover") ist ein Genetischer Operator, der zur Erzeugung neue Individuen basierend auf deren Eltern verwendet wird. Dabei werden die Gene der Eltern generiert, um Nachkommen zu bilden. Es gibt verschiedene Arten Methoden der Kreuzung, eine davon ist „single-point crossover". Bei dieser Methode wird ein Punkt innerhalb der Chromosomen ausgewählt, der die genetischen Informationen der Eltern in zwei Abschnitte aufteilt. Die Nachkommen erhalten den ersten Teil von einem Elternteil und den restliche Abschnitt vom anderen Elternteil (vgl. Dutta, o.J).

```
def crossover_parents(parent1, parent2):
    bits_amount = len(parent1.item_bits)
    half_amount = int(bits_amount / 2)

    # Erste Hälfte von Elternteil 1 plus zweite Hälfte von Elternteil 2
    child1_bits = parent1.item_bits[:half_amount] + parent2.item_bits[half_amo

    # Erste Hälfte von Elternteil 2 plus zweite Hälfte von Elternteil 1
    child2_bits = parent2.item_bits[:half_amount] + parent1.item_bits[half_amo

    child1 = Individual(child1_bits)
    child2 = Individual(child2_bits)
    return (child1, child2)
```

Die Funktion „crossover_parents“ in meinem Code generiert neue Individuen indem sie die Bit-Sequenz der Eltern kombiniert. Dabei wird die erste Hälfte der Bits von einem Elternteil und die zweite Hälfte vom anderen übernommen.

4.4.3 Mutation

Mutation ist ebenfalls ein Genetischer Operator und wird eingesetz, um die genetische Vielfalt innerhalb einer Population zu erhöhen. Es gibt verschiedene Mutationsmethoden, eine davon ist „bit flip mutation“. Bei diesem Algorithmus werden zufällige Bits eines Individuums invertiert, das heißt, ein Bit, das den Wert eins hat, wird zu null, und ein Bit, das den Wert null hat, wird zu eins (vgl. Sil, o.J.).

```
def mutate_child(individual):
    bits_amount = len(individual.item_bits)
    random_bit = random.randrange(bits_amount)
    individual.item_bits[random_bit] = (
        1 - individual.item_bits[random_bit]
    )  # 1 wird zu null und umgekehrt
```

Mein Code mutiert Nachkommen mit der Funktion „mutatate_child“. Dabei wird ein zufälliges Bit ausgewählt und invertiert.

4.4.4 Elitismus

Durch die zufällige Auswahl der Individuen kann es allerdings passieren, dass das beste Individuum nicht für die nächste Generation verwendet wird. Um das zu vermeiden, wird Elitismus eingesetzt. Dabei wandert das Individuum mit dem höchsten Fitness-Wert direkt in nachkommende Generation über (vgl. Mitchell, 1996, S. 126).

```
class Population(Population):
    def create_new_population(self):
        new_population = []
        best_individuals = self.population[0:2]
        new_population.extend(best_individuals)
```

Die Individuen in meiner Population werde ich später basierend auf ihren Fitness-Wert sortieren. Die ersten zwei Individuen in der Population sind demnach die Besten. Diese werden dann in die neue Generation ohne genetischer Veränderung übernommen.

4.5 Neue Population

```
class Population(Population):
    def create_new_population(self):
        new_population = []
        best_individuals = self.population[0:2]
        new_population.extend(best_individuals)
        while population_size > len(new_population):
            parent1, parent2 = selection(self.population)

            child1, child2 = crossover_parents(parent1, parent2)

            mutate_child(child1)
            mutate_child(child2)

            new_population.append(child1)
            new_population.append(child2)

        return new_population
```

Eine neue Generation bilde ich mit der Methode „create_new_population“. Zuerst werden die besten zwei Individuen unverändert in die „new_population“ kopiert. Danach wird eine Schleife ausgeführt. Innerhalb der Schleife werden zwei Partner durch Tournament-Selektion ausgesucht. Diese bilden mit Kreuzung Nachkommen. Diese Nachkommen werden mit „mutatate_child“ mutiert und dann zur neuen Population hinzugefügt. Dieser Prozess wiederhollt sich, bis die gewünschte Populationsgröße erreicht ist.

4.6 Finalisierung

```
class Population(Population):
    def start(self, mass_limit):
        self.calculate_fitness(mass_limit)
        for _ in range(500):
            self.population = self.create_new_population()
            self.calculate_fitness(mass_limit)

            self.population.sort(
                reverse=True, key=lambda individual: individual.fitness_value
            )

            best_individual = self.population[0]

        print(best_individual.fitness_value)
        print(best_individual.item_bits)


population_size = 20
```

```
mass_limit = 3000
Population(population_size).start(mass_limit)
```

Um eine angenäherte Lösung für das Knapsack-Problem zu finden verwende ich die Methode „start“. Diese berechnet zuerst den Fitness-Wert aller Individuen. In der Schleife wird Code für jede Generation ausgefürt. Zuerst wird die neue Population erstellt, dann werden die Fitness-Werte der Individuen berechnet, und dann werden die Individuen basierend auf ihrem Fitness-Wert sortiert, um den Elitismus anzuwenden. Diese Schleife läuft für 500 Generation und gibt anschließend ein Ergebnis zurück.

5 Das Training mit NeuroEvolution of Augmenting Topologies

5.1 Grundlage

NeuroEvolution of Augmenting Topologies (NEAT) ist ein genetischer Algorithmus, der verwendet wird, um sowohl die Struktur als auch die Architektur von neuronalen Netzwerken zu optimieren. Im Gegensatz zu traditionellen Algorithmen wie dem Gradient Descent, die mit einer festgelegten Topologie arbeiten, beginnt NEAT mit einem sehr einfachen Netzwerk. Dieses kann auch aus direkten Verbindungen zwischen Eingabe- und Ausgabeneuronen bestehen. Diese Struktur des Netzwerks wird dann im Laufe der Evolution schrittweise komplexer, indem beispielsweise neue Neuronen oder Verbindungen hinzugefügt werden. Der Vorteil von NEAT ist, dass es nicht nur die Parameter des Netzwerks, sondern auch die Topologie optimiert, wodurch es in der Lage ist, neuartige und effektive Netzwerkarchitekturen zu entdecken. (Burton-McCreadie, 2024)

Die Methoden, die zur Evolution eines Netzwerks verwendet werden, überschneiden sich mit denen, die im Kapitel „5. Genetische Algorithmen“ besprochen wurden. NEAT beginnt mit einer Population von Genomen, die als Baupläne für neuronale Netzwerke dienen. Jedes Genom beschreibt dabei eine mögliche Struktur eines neuronalen Netzwerks. Diese Genome werden mit einer Fitnessfunktion evaluiert, damit die Netzwerke miteinander verglichen werden können. Anschließend werden Konzepte wie Reproduktion, Mutation und Rekombination angewendet, um neue Netzwerke zu entwickeln und die Population von Genomen im Laufe der Generationen zu verbessern (ebd.).

Ein weiterer wichtiger Bestandteil von NEAT ist das Konzept der Speciation. Strukturen, die langfristig vorteilhaft für ein neuronales Netzwerk sind, können dessen Leistung anfänglich sogar verschlechtern. Es benötigt Zeit um, diese neuen Strukturen effizient zu nutzen. Demnach ist es notwendig, Mechanismen anzuwenden, um diese Strukturen vorrübergehen zu beschützen, damit diese nicht sofort von temporär leistungsfähigeren Netzwerken verdrängt werden (ebd.).

Dieser Mechanismus wird durch Speciation bereitgestellt. Diese stellt sicher, dass innovative, aber noch nicht optimal performende Netzwerke nicht von lang etablierten, aber weniger vielversprechenden Lösungen verdrängt werden. Dies wird erreicht, in dem Spezien geformt werden, die die Paarung von Netzwerke einschränken. Herschen zu große Unterschiede zwischen zwei Genomen, so können diese nicht gepaart werden. Auf diese Weise bleiben Differenzen zwischen Netzwerken erhalten um unterschiedliche Bereiche des Lösungsraums zu erkunden (ebd.)

5.2 Der Irrgarten

Um die Anwendungspotenziale von NEAT zu veranschaulichen, bespreche ich in diesem Kapitel das Training meiner Künstlichen Intelligenz an einem selbst programmierten Spiel. Hierbei handelt es sich um einen einfachen Irrgarten. Das Ziel der KI ist es, vom Startpunkt zum Endpunkt zu gelangen. Eine Illustration des Spiels sieht folgendermaßen aus:

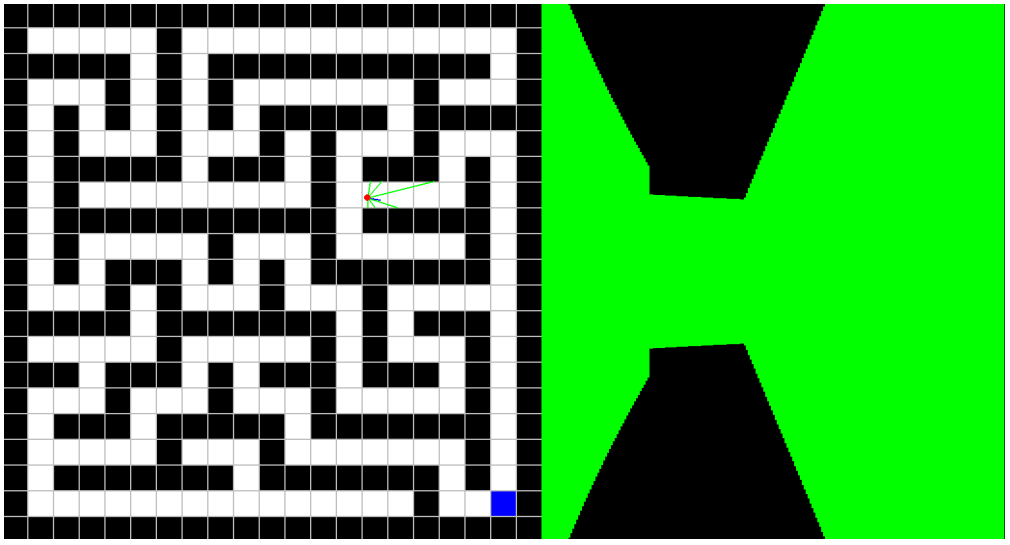

Abb. 9: Das Irrgarten Spiel (Verf.)

Die KI startet in oberen linken Ecke und muss möglichst schnell zur blauen Zelle gelangen. Wichtig ist hierbei zu beachten, dass die KI kein vollständiges Bild des Irrgartens besitzt und nur die Information in dessen unmittelbare Umgebung erhält. Die rechte Seite der Abbildung zeigt eine visuelle Darstellung dieser eingeschränkten Wahrnehmung.

Dieses Kapitel befasst sich allein mit dem Training der KI. Hierfür verwende ich die Library „neat-python“, die die notwendigen Funktionen für die Evolution des neuronalen Netzwerks bereitstellt. Anders als in den vorherigen Kapiteln werde ich den verwendeten Quellcode nicht detailliert beschreiben, da viele der verwendeten Konzepte außerhalb des Bereichs der Künstlichen Intelligenz liegen. Der gesamte Code ist allerdings hinten im Anhang zu finden.

5.3 Die Netzwerk Konfiguration

Ein wichtiger Bestandteil beim Trainieren einer künstlichen Intelligenz mit der neat-python Bibliothek ist die Konfig-Datei. Diese beinhaltet sämtliche Einstellungen und Parameter für den Evolutionsprozess sowie die anfängliche Struktur der KI. Die Datei enthält viele Einstellungen, deshalb werde ich nur die wesentlichsten Informationen besprechen. Der vollständige Code für die Datei ist jedoch ebenso im Anhang zu finden.

Meine Künstliche Intelligenz besitzt sechs Eingabeneuronen. Diese stellen verschiedene Sensoren dar. In der Abbildung sind das die grünen Striche, die nach vorne gestrahlt werden, bis sie eine Wand berühren. Sie dienen dazu, der KI Informationen darüber zu geben, wie weit entfernt ein Objekt ist. Dabei besitzt sie allerdings nur sechs dieser Sensoren. Eine bessere Illustration für das Blickfeld dieser Künstlichen Intelligenz sieht demnach so aus:

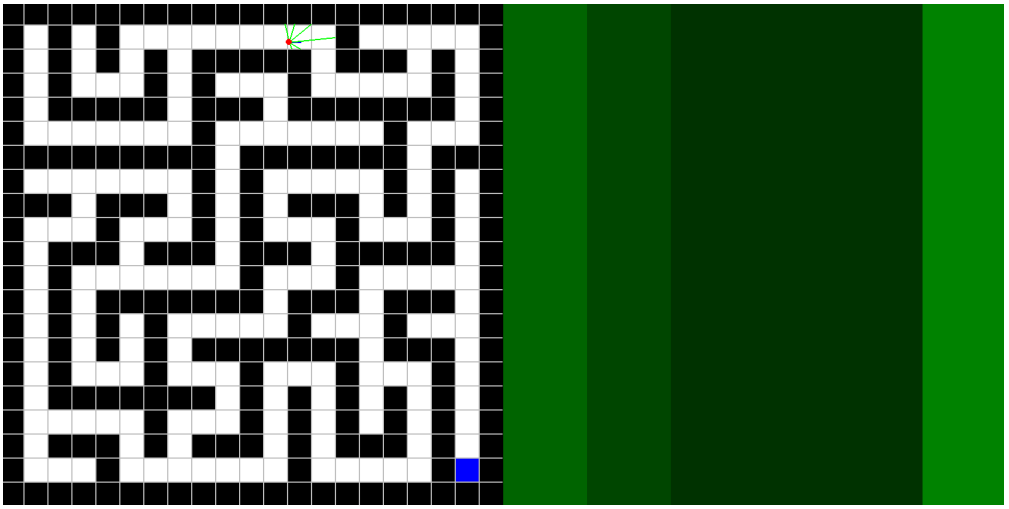

Abb. 10: Blickfeld der KI (Verf.)

Die Spalten stehen hierbei für die Eingabeinformationen, die das Netzwerk erhält, und geben die Entfernungen an, die ein Sensor misst.

Die Eingabeinformationen erhalte ich mit einer Methode namens „get_ai_input_data". Diese verwendet die Raycaster-Klasse, um die Längen der Strahlen zu berechnen und gibt diese aus. Nachdem die Eingabedaten erhalten wurden, generiere ich eine Vorhersage mit der Funktion „self.net.activate(inputs)". Der Code dafür sieht wie folgt aus:

```
class Player:
    ...

    def ai_input(self):
        inputs = self.get_ai_input_data()
        output = self.net.activate(inputs)
        ...
```

„outputs" sind die Vorhersagen des Netzwerks und geben an, welche Aktion die KI ausführt. Ich habe mich für zwei Ausgabeneuronen entschieden, sodass zwei verschiedene Aktionen ausgeführt werden können: eines für das Rotieren und eines für die Bewegung. Für die Aktivierungsfunktion habe ich mich für tanh entschieden, da sie mir die besten Ergebnisse geliefert hat. Der Vorteil dieser Aktivierungsfunktion ist, dass sie die Werte zwischen minus eins und eins begrenzt. Diese Information verwende ich dann, um die Richtung und Bewegung anzupassen. Ist der Wert des ersten Ausgabeneurons über null, so rotiert die KI nach rechts, andernfalls nach links:

```
class Player:
    ...

    def ai_input(self):
        ...
        self.angle -= 0.1  # Links drehen
        # Wenn der Wert des ersten Ausgabeneurons größer als 0 ist, dann
        # dreht sich die KI nach rechts
        if max(output[0], 0):
            self.angle += 0.2  # Zwei mal nach Rechts drehen
        ...
```

Der zweite Ausgabewert kontrolliert, ob sich die KI nach vorne bewegt oder nicht. Dies geschieht, wenn der Wert positiv ist:

```
class Player:
    ...

    def ai_input(self):
        ...
        direction = 0  # Stillstand
        # Wenn der Wert des zweiten Ausgabeneurons größer als 0 ist,
        # dann bewegt sich die KI nach vorne
        if max(output[1], 0):
            direction = 1  # Vorwärtsbewegung
        ...
```

5.4 Die Fitness-Evaluierung

Das Ziel des Netzwerks ist es, die blaue Zelle in möglichst kurzer Zeit zu erreichen. Um dieses Ziel zu erreichen, wird die KI basierend darauf bewertet, wie nah sie der blauen Zelle kommt. Zu diesem Zweck belohne ich ein Genom, sobald es eine neue Zelle betritt.

Jedes Genom besitzt eine Lebensdauer, die in jeder Iteration verringert wird. Diese Lebensdauer kann jedoch nur dann wieder erhöht werden, wenn die KI die Umgebung erkundet. Dies fördert das Lernen und die kontinuierliche Bewegung der KI. Der Code sieht wie folgt aus:

```
class Player:
    ...

    def path_collision(self):
        collision_index = self.rect.collidelist(self.path_cells)
        if collision_index != -1:
            # Neue erkundete Zelle
            if self.path_cells_score[collision_index] == 0:
                self.life_time += 20
                self.genome.fitness += 8
            self.path_cells_score[collision_index] += 1
            if self.path_cells_score[collision_index] > 50:
                self.life_time = 0
                self.genome.fitness -= 80
```

Zuerst wird überprüft, ob die KI mit einer Zelle kollidiert. Anschließend wird festgestellt, ob es sich um eine bereits erkundete oder eine neue Zelle handelt. Sobald sich die KI zu lange bei einer Zelle aufhält, wird sie entfernt und bestraft, indem ihre Lebensdauer „self.life_time“ sowie ihr Fitness-Wert „self.genome.fitness“ entsprechend angepasst werden.

Damit die KI außerdem lernt, die Wände nicht zu berühren, lösche ich alle KIs, die mit den ihnen kollidieren, und bestrafe sie für diese Aktion. Dadurch wird die KI gezwungen, ihre Bewegungen sorgfältig zu planen und die Umgebung aktiv zu erkunden, ohne unnötige Fehler zu machen.

Gelangt die KI jedoch zum Ziel, erhält sie eine sehr große Belohnung. Um jedoch schnellere KIs zu bevorzugen, subtrahiere ich von dieser Belohnung die benötigte Zeit „self.total_time“:

```
class Player:
    ...

    def collision(self):
        collision_index = self.rect.collidelist(self.boxes)
        if collision_index != -1:
            is_goal = self.boxes_type[collision_index]
            # Tod nach Berührung mit Zelle. Belohnung falls Ziel und
            # Bestrafung falls Wand.
            if is_goal:
                self.genome.fitness += 10000
                self.genome.fitness -= self.total_time
            else:
                self.genome.fitness -= 65
            self.life_time = 0
            return
```

Diese Methode „collision“ überprüft, ob die KI eine Wand oder das Ziel berührt und belohnt oder bestraft das Genom basierend auf dem Ergebnis.

5.5 Die Resultate

Nach einer Anzahl von Generationen hat die Künstliche Intelligenz gelernt, den Irrgarten zu navigieren, indem sie sich entweder an der linken oder rechten Wand festhält. Es ist hierbei wichtig zu beachten, dass die KI keine Erinnerungen an vorherige Erfahrungen besitzt und ausschließlich auf die aktuellen Eingabedaten reagiert. Zudem hat die KI gelernt, dass sie, sobald sie Sackgassen entdeckt, stehen bleibt und umkehrt. Diese Verhaltensweisen tragen dazu bei, dass ein neuronales Netzwerk entstanden ist, das in kurzer Zeit einen Irrgarten durchqueren kann.

5.6 Probleme und Schwierigkeiten

Ursprünglich war geplant, eine KI basierend auf einem existierenden Spiel zu trainieren. Da sich dieser Ansatz jedoch als zu mühsam herausstellte, entschied ich mich, ein eigenes Spiel zu rekreieren. Dies erforderte jedoch zusätzliches Wissen in Bereichen wie der Generierung von Irrgärten und der Berechnung der Strahlendistanz, Konzepte die mit der Künstlichen Intelligenz zu tun haben. Zudem gestaltete sich der Prozess der Findung einer passenden Struktur und Fitness-Funktion als schwieriger als erwartet und erforderte viel Experimentieren.

5.7 Die fertige Irrgarten-Navigation mittels NEAT-KI

Der Quellcode der entwickelten KI befindet sich im Anhang dieser Arbeit. Eine weiterentwickelte Version ist zusätzlich im folgenden GitHub-Repository öffentlich zugänglich: https://github.com/scrythe/it_learns_maze.

Zum Ausprobieren des Modells sowie des zugehörigen Spiels kann die folgende Website aufgerufen werden: https://scrythepvp.itch.io/it-learns-maze. Dort besteht die Möglichkeit, das Spiel über den Button "Play" eigenständig zu spielen. Mit dem Button "Train" kann das NEAT-Modell selbstständig trainiert werden. Nach einem Abschluss eines Trainingsprozesses kann die beste entwickelte KI dann über den Button "Test" isoliert beobachtet werden.

6 Fazit

Die vorliegende Arbeit bietet eine umfassende technische Einführung in neuronale Netzwerke und maschinellem Lernen. Beginnend mit den Grundlagen, wie den Aufgaben und Eigenschaften der drei verschiedenen Eingabeschichten und den Aktivierungsfunktionen und dessen Funktionsweisen gibt die Arbeit einen Überblick über den theoretischen Aufbau eines künstlichen neuronalen Netzwerkes. Anhand der beiden selbst programmierten Anwendungsbeispiele (Zahlenerkennung und Irrgarten) werden viele theoretische Konzepte anschaulich und praxisnah dargestellt. Ein Fokus liegt auch auf Konzepten wie Back Propagation, Gradient Descent, Loss, Cost und Objective Functions. Die Arbeit ging auch auf die Unterschiede zwischen Deep und Shallow Learning ein und zeigte, wie diese Konzepte in der Praxis angewendet werden können. Die Implementierung des Stochastic Gradient Descent und die Anwendung auf das Zahlenerkennungsmodell demonstrierten die praktische Relevanz der theoretischen Ausführungen. Eine Fragestellung, die auch zu einem zufriedenstellenden Ergebnis geführt hat, war das Erforschen und Beschreiben von Anwendungen von genetischen Algorithmen und das NEAT-Verfahren zur Lösung von Optimierungsproblemen. Ich war sehr beeindruckt von der Erkenntnis, dass sich viele in der Arbeit behandelten Konzepte mit mathematischem Wissen beschreiben lassen, das ich bereits in der Schule erlernt habe. Schlussendlich kann man konkludieren, dass die Arbeit einen umfassenden Einblick in die Funktionsweise und Anwendung neuronaler Netzwerke und genetischer Algorithmen bietet. Es ist faszinierend, wie viel es über neuronale Netzwerke zu lernen gibt, und wie dieses Wissen im Programmier-Teil der Arbeit Anwendung fand.

7 Literaturverzeichnis

Alake, Richmond (o.J.): Loss Functions in Machine Learning Explained. https://www.datacamp.com/tutorial/loss-function-in-machine-learning [Zugriff: 31.01.2025]

Anshumanm2fja (2024): What is Forward Propagation in Neural Networks? https://www.geeksforgeeks.org/what-is-forward-propagation-in-neural-networks/ [Zugriff: 16.10.2024]

Belagatti, Pavan (2024): Understanding the Softmax Activation Function: A Comprehensive Guide. https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/ [Zugriff: 01.02.2025]

Bhayani, Arpit (o.J.): Genetic algorithm to solve the Knapsack Problem. https://arpitbhayani.me/blogs/genetic-knapsack/ [Zugriff: 16.12.2024]

Burton-McCreadie, Trevor (2024): The NEAT Algorithm: Evolving Neural Networks. https://blog.lunatech.com/posts/2024-02-29-the-neat-algorithm-evolving-neural-network-topologies [Zugriff: 26.02.2025]

Dey, Suman (2019): Understanding Objective Functions in Deep Learning. https://dimensionless.in/understanding-objective-functions-in-deep-learning/ [Zugriff: 20.02.2025]

Dutta, Avik (o.J.): Crossover in Genetic Algorithm. https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/ [Zugriff: 10.02.2025]

Gómez Bruballa, Raúl (2018): Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names. https://gombru.github.io/2018/05/23/cross_entropy_loss/ [Zugriff: 02.02.2025]

Jung, Daniel (2014): Matrizen multiplizieren, Matrixmultiplikation, Übersicht | Mathe by Daniel Jung. https://www.youtube.com/watch?v=OphPlrzeqng [Zugriff: 22.02.2025]

Kanade, Vijay (o.J.): What Are Genetic Algorithms? Working, Applications, and Examples. https://www.spiceworks.com/tech/artificial-intelligence/articles/what-are-genetic-algorithms/ [Zugriff: 16.12.2024]

Khan, Azim (2024) A Beginner's Guide to Deep Learning with MNIST Dataset. https://medium.com/@azimkhan8018/a-beginners-guide-to-deep-learning-with-mnist-dataset-0894f7183344 [Zugriff: 14.01.2025]

Kinsley, Harrison (2020) Neural Networks from Scratch - P.4 Batches, Layers, and Objects. https://www.youtube.com/watch?v=TEWy9vZcxW4 [Zugriff: 16.10.2024]

Kinsley, Harrison (2020): Neural Networks from Scratch - P.5 Hidden Layer Activation Functions. https://www.youtube.com/watch?v=gmjzbpSVY1A [Zugriff: 16.10.2024]

Kostadinov, Simeon (2019): Understanding Backpropagation Algorithm. https://medium.com/towards-data-science/understanding-backpropagation-algorithm-7bb3aa2f95fd [Zugriff: 11.02.2025]

Kurbiel, Thomas (2021): Derivative of the Softmax Function and the Categorical Cross-Entropy Loss. https://medium.com/towards-data-science/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1 [Zugriff: 26.02.2025]

Lheureux, Adil (o.J.): Feed-forward vs feedback neural networks. https://www.digitalocean.com/community/tutorials/feed-forward-vs-feedback-neural-networks [Zugriff: 16.10.2024]

Lodhi Ramlakhan (o.J.): Difference between Shallow and Deep Neural Networks. https://www.geeksforgeeks.org/difference-between-shallow-and-deep-neural-networks/ [Zugriff: 31.01.2025]

Muns, Andy (o.J.): Objective function types: A machine learning guide. https://telnyx.com/learn-ai/objective-function-machine-learning [Zugriff: 20.02.2025]

Mitchell, Melanie (1996): An Introduction to Genetic Algorithms. Fifth printing. Cambridge, Massachusetts: The MIT Press.

Nielsen, Michael (2015): Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/chap1.html [Zugriff: 16.10.2024]

Roy, Rahul (2024): ML | Stochastic Gradient Descent (SGD). https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/ [Zugriff: 26.02.2025]

Saxena, Abhimanyu (2024): Classification vs Regression in Machine Learning. https://www.appliedaicourse.com/blog/classification-vs-regression-in-machine-learning/ [Zugriff: 01.02.2025]

Sil, Pritam (o.J.): Mutation Algorithms for String Manipulation (GA). https://www.geeksforgeeks.org/mutation-algorithms-for-string-manipulation-ga/ [Zugriff: 10.02.2025]

Singh, Abhay (2025): Gradient Descent Explained: The Engine Behind AI Training. https://medium.com/@abhaysingh71711/gradient-descent-explained-the-engine-behind-ai-training-2d8ef6ecad6f [Zugriff: 03.02.2025]

Stansbury, Dustin (2020): Derivation: Error Backpropagation & Gradient Descent for Neural Networks. https://dustinstansbury.github.io/theclevermachine/derivation-backpropagation [Zugriff: 24.02.2025]

Topper, Noah (2023): Sigmoid Activation Function: An Introduction. https://builtin.com/machine-learning/sigmoid-activation-function [Zugriff: 14.01.2025]

8 Abbildungsverzeichnis

9 Anhang

```
import numpy as np  # Import des Pakets NumPy

# Die Eingabewerte der Eingabeneuronen
inputs = np.array([[0.3], [0.6]])

# Die Gewichte zwischen den zwei Eingabeneuronen und dem Ausgabeneuron
weights = np.array([0.8, 0.2])
bias = 4  # Der Bias-Wert des Ausgabeneurons

# Berechnung des Ausgabewerts des Ausgabeneurons
output = inputs[0] * weights[0] + inputs[1] * weights[1] + bias
print(output)

inputs = np.array([1.2, 3.2])

# Gewichte zwischen Eingabeneuronen und Ausgabeneuronen
weights1 = np.array([0.8, 1.3])  # Gewichte für das erste Ausgabeneuron
weights2 = np.array([3.1, 1.6])  # Gewichte für das zweite Ausgabeneuron

bias1 = 4  # Bias-Wert für das erste Ausgabeneuron
bias2 = 3  # Bias-Wert für das zweite Ausgabeneuron

# Der Ausgabewert des ersten Ausgabeneurons
output1 = inputs[0] * weights1[0] + inputs[1] * weights1[1] + bias1

# Der Ausgabewert des zweiten Ausgabeneurons
output2 = inputs[0] * weights2[0] + inputs[1] * weights2[1] + bias2

print(output1, output2)

inputs = np.array([[1.2], [3.2]])

# Gewichtsmatrix zwischen Eingabeneuronen und Ausgabeneuronen (2 x 2)
weights = np.array(
    [
        [0.8, 1.3],  # Gewichte des ersten Ausgabenneuron
        [3.1, 1.6],  # Gewichte des zweiten Ausgabenneuron
    ]
)

# Bias-Vektor für die Ausgabeneuronen (2 x 1)
```

```
biases = np.array([[4], [3]])

# Berechnung der Ausgabewerte (z) (2 x 2) durch Matrixmultiplikation
outputs = np.dot(weights, inputs) + biases
print(outputs)

# Eingabematrix einer Batch mit 4 Trainingsbeispielen,
# wobei jedes Beispiel 2 Eingabewerte enthält (2 x 4)
inputs = np.array([
    [1.2, 3.2, 4.2, 3.1],
    [3.2, 1.2, 0.2, 2.2],
])

# Berechnung der Ausgabewerte durch Matrixmultiplikation
outputs = np.dot(weights, inputs) + biases  # Matrix von Ausgabewerten (2 x 4)
print(outputs)

class Layer:
    def __init__(self, n_inputs, n_neurons):
        """
        n_inputs: Anzahl an Eingabewerten (bzw. Neuronen der vorherigen
        Schicht).
        n_neurons: Anzahl an Neuronen für diese Schicht.
        """
        # Gewichtsmatrix
        self.weights = 0.1 * np.random.randn(n_neurons, n_inputs)
        # Bias-Vektor
        self.biases = 0.1 * np.random.randn(n_neurons, 1)

    def forward(self, inputs):
        """
        Berechnung des Ausgabewerts für die Neuronen in dieser Schicht
        basierend auf den Eingabewerte "inputs".
        """
        # Eingabewerte für spätere Verwendung speichern
        self.saved_inputs = inputs
        # Ausgabewerte als Matrix
        outputs = np.dot(self.weights, inputs) + self.biases
        return outputs  # Rückgabe der Ausgabewerte

# Eingabeschicht mit 2 Neuronen → verborgenen Schicht mit 4 Neuronen
hidden_layer = Layer(2, 4)

# Verborgenen Schicht mit 4 Neuronen → Ausgabeschicht mit 5 Neuronen
output_layer = Layer(4, 5)
```

```
# Ausgabewerte für die verborgene Schicht
hidden_layer_outputs = hidden_layer.forward(inputs)

# Ausgabewerte für die Ausgabeschicht
output_layer_outputs = output_layer.forward(hidden_layer_outputs)
print(output_layer_outputs)

print(output_layer_outputs.shape)

class Sigmoid:
    def forward(self, raw_outputs):
        """
        Berechnet die aktivierten Ausgabewerte basierend auf den rohen
        Ausgabewerten "raw_outputs".
        """
        activated_outputs = 1 / (1 + np.exp(-raw_outputs))
        # Aktivierte Ausgaben für spätere Verwendung speichern
        self.saved_activated_outputs = activated_outputs
        return activated_outputs

output_layer = Layer(2, 4)
activation_function = Sigmoid()

raw_outputs = output_layer.forward(inputs)
activated_outputs = activation_function.forward(raw_outputs)
print(activated_outputs)

class ReLU:
    def forward(self, raw_outputs):
        """
        Berechnet die aktivierten Ausgabewerte basierend auf den rohen
        Ausgabewerten "raw_outputs".
        """
        self.saved_raw_outputs = raw_outputs
        activated_outputs = np.maximum(0, raw_outputs)
        return activated_outputs

class Softmax:
    def forward(self, raw_outputs):
        """
        Berechnet die aktivierten Ausgabewerte basierend auf den rohen
        Ausgabewerten "raw_outputs".
        """
        # Exponierte Werte
```

```
        exponentiated_values = np.exp(raw_outputs - np.max(raw_outputs, \
                                              axis=0))
        # Summe der exponierten Werte
        sum_values = np.sum(exponentiated_values, axis=0, keepdims=True)
        # Normalisierte / aktivierte Ausgaben
        normalized_outputs = exponentiated_values / sum_values
        return normalized_outputs

class Network:
    def __init__(
        self,
    ):
        self.layers = []
        self.activation_functions = []

    def add_layer(self, layer, activation_function):
        """
        Fügt eine instanzierte Schicht "layer" mit ihrer entsprechenden
        Aktivierungsfunktion "activation_function" zum Netzwerk hinzu.
        """
        self.layers.append(layer)
        self.activation_functions.append(activation_function)

    def forward_propagation(self, inputs):
        """
        Berechnet die Vorhersagen "predictions" des Netzwerkes anhand der
        Eingabewerte "inputs" der Eingabeschicht.
        """
        current_inputs = inputs
        for layer, activation_function in zip(self.layers, \
                                    self.activation_functions):
            raw_outputs = layer.forward(current_inputs)
            activated_outputs = activation_function.forward(raw_outputs)
            # Aktivierte Ausgaben der Schicht werden als Eingabewerte
            # für die nächste Schicht verwendet
            current_inputs = activated_outputs
        predictions = current_inputs
        return predictions

network = Network()
network.add_layer(
    Layer(2, 4),  # Eingabeschicht → versteckte Schicht
    ReLU(),  # Aktivierungsfunktion für die versteckte Schicht
)
network.add_layer(
```

```
    Layer(4, 5),  # Versteckte Schicht → Ausgabeschicht
    Softmax(),  # Aktivierungsfunktion für die Ausgabeschicht
)

network.forward_propagation(inputs)

# Lade Tranings und Test Daten

from data.load_data import load_test_data

import gzip
import numpy as np
import pickle


def load_images(file):
    with gzip.open(file, "r") as f:
        f.read(4)  # Überspringen des Headers (Magic Number)
        n_images = int.from_bytes(f.read(4), "big")
        f.read(8)  # Überspringen des Headers (Anzahl Reihen und Zeilen)

        # Lesen der Bilddaten
        # Pixelwerte sind von 0 bis 255 als unsigned Byte gespeichert
        image_data = f.read()
        images = np.frombuffer(image_data, dtype=np.uint8).reshape(n_images, 784).T
        images = (
            images.reshape(images.shape[0], -1).astype(np.float32) - 127.5
        ) / 127.5  # Zwischen -1 und 1
        return images


def load_labels(file):
    with gzip.open(file, "r") as f:
        f.read(8)  # Überspringen des Headers (Magic Number und Anzahl der Labels)
        label_data = f.read()
        labels = np.frombuffer(label_data, dtype=np.uint8)
        labels = np.eye(10)[labels].T  # Von Ziffern zu "Wahrscheinlichkeiten"
        return labels


def load_trainings_data():
    images = load_images("data/train-images-idx3-ubyte.gz")
    labes = load_labels("data/train-labels-idx1-ubyte.gz")
    return images, labes
```

```python
def load_test_data():
    images = load_images("data/t10k-images-idx3-ubyte.gz")
    labes = load_labels("data/t10k-labels-idx1-ubyte.gz")
    return images, labes

network = Network()
network.add_layer(
    Layer(784, 20),  # Eingabeschicht → versteckte Schicht
    ReLU(),  # Aktivierungsfunktion für die versteckte Schicht
)
network.add_layer(
    Layer(20, 10),  # Versteckte Schicht → Ausgabeschicht
    Softmax(),  # Aktivierungsfunktion für die Ausgabeschicht
)


def test_neural_network(network):
    # Bilder (Eingabewerte) und labels (tatsächliche Zielwerte als
    # Wahrscheinlichkeiten)
    images, labels = load_test_data()

    # Vorhersagen als Wahrscheinlichkeitsverteilung
    predictions = network.forward_propagation(images)

    N = predictions.shape[1]  # Anzahl an Trainingsbeispielen

    # Vorhersagen als Ziffern
    predicted_numbers = np.argmax(predictions, axis=0)

    # tatsächliche Zielwerte als Ziffern
    actual_values = np.argmax(labels, axis=0)

    # Vektor aus "Richtig Falsch" Werten
    comparisons = predicted_numbers == actual_values

    # Summe / Anzahl an richtigen Aussagen
    n_correct_predictions = sum(comparisons)

    # Genauigkeit des neuronalen Netzwerkes
    accuracy = n_correct_predictions / N

    print(accuracy)
```

```
test_neural_network(network)

class MeanSquaredError:
    def calculate_cost(predictions, targets):
        losses = np.sum(np.square(predictions - targets), axis=0)
        cost = np.mean(losses)
        return cost

class CategoricalCrossEntropy:
    def calculate_cost(predictions, targets):
        predictions = np.clip(predictions, 1e-7, 1 - 1e-7)
        losses = -np.sum(targets * np.log(predictions), axis=0)
        cost = np.mean(losses)
        return cost

class GD:
    def __init__(self, network, learning_rate):
        """
        network: Das Netzwerk, das optimiert werden soll
        learning_rate: Die Lernrate, die die Schrittgröße bestimmt
        """
        self.network = network
        self.learning_rate = learning_rate

    def update_parameters(self):
        """
        Aktualisiert die Parameter (Gewichte und Bias-Werte) aller
        Schichten im Netzwerk basierend auf den Gradienten
        """
        # Iteriert über alle Schichten des Netzwerks und aktualisiert deren
        # Parameter
        for layer in self.network.layers:
            # Aktualisiert die Gewichte der aktuellen Schicht mit dem
            # negativen Gradienten multipliziert mit der Lernrate, um den
            # Schritt zu skalieren
            layer.weights -= self.learning_rate * layer.gradient_weights
            # Aktualisiert die Bias-Werte der aktuellen Schicht mit dem
            # negativen Gradienten multipliziert mit der Lernrate, um
            # sden Schritt zu skalieren
            layer.biases -= self.learning_rate * layer.gradient_biases

class Layer(Layer):
    def backwards(self, gradient_raw_outputs):
        """
        Berechnet den Gradienten der Cost Function in Bezug zu den
```

```
        Gewichten und Bias-Werten der aktuellen Schicht und aktivierten
        Ausgaben der vorherigen Schicht.

        gradient_raw_outputs: Gradient der Cost Function in Bezug zu den
        rohen Ausgaben der aktuellen Schicht (dJ/dZ).
        """

        # Gradient der Cost Function in Bezug zu den Gewichten von der
        # aktuellen Schicht (dJ/dW).
        self.gradient_weights = np.dot(
            gradient_raw_outputs,
            self.saved_inputs.T,
        )

        # Gradient in Bezug zu den Bias-Werten (dJ/db).
        self.gradient_biases = np.sum(gradient_raw_outputs, axis=1, \
                               keepdims=True)

        # Gradient in Bezug zu den aktivierten Ausgaben der vorherigen
        # Schicht (dJ/dA).
        gradient_activated_outputs = np.dot(self.weights.T, \
                                     gradient_raw_outputs)
        return gradient_activated_outputs



class Sigmoid(Sigmoid):
    def backwards(self, gradient_activated_outputs):
        """
        Berechnet den Gradienten der Cost Function in Bezug zu
        den rohen Ausgaben der aktuellen Schicht (dJ/dZ)

        gradient_activated_outputs: Gradient der Cost Function in Bezug
        zu den aktivierten Ausgaben der aktuellen Schicht (dJ/dA)
        """
        # Gradient in Bezug zu (A*(A-1))
        d_activated_d_raw = self.saved_activated_outputs * (
            1 - self.saved_activated_outputs
        )

        gradient_raw_outputs = gradient_activated_outputs * d_activated_d_raw
        return gradient_raw_outputs


class ReLU(ReLU):
```

```
    def backwards(self, gradient_activated_outputs):
        """
        Berechnet den Gradienten der Cost Function in Bezug zu
        den rohen Ausgaben der aktuellen Schicht (dJ/dZ).

        gradient_activated_outputs: Gradient der Cost Function in Bezug
        zu den aktivierten Ausgaben der aktuellen Schicht (dJ/dA).
        """
        # Gradient der Cost Function in Bezug zu den rohen Ausgaben (dJ/dZ).
        gradient_raw_outputs = gradient_activated_outputs * \
            (self.saved_raw_outputs > 0)
        return gradient_raw_outputs

class MeanSquaredError(MeanSquaredError):
    def backwards(predictions, targets):
        """
        Berechnet den Gradienten des Mean Squared Error in Bezug zu den
        Vorhersagen.
        """
        N = predictions.shape[1]  # Anzahl an Trainingsbeispielen
        gradient_predictions = (2 / N) * (predictions - targets) / \
            len(predictions)
        return gradient_predictions

class CategoricalCrossEntropy(CategoricalCrossEntropy):
    def backwards(predictions, targets):
        """
        Berechnet den Gradienten des Categorical Cross Entropy in
        Bezug zu den rohen Ausgaben der Ausgabenschicht (dJ/dZ).
        """
        N = predictions.shape[1]  # Anzahl an Trainingsbeispielen
        gradient_raw_outputs = (predictions - targets) / N
        return gradient_raw_outputs


class Softmax(Softmax):
    def backwards(self, gradient_raw_outputs):
        # Gibt die Gradienten direkt weiter (Softmax wird in Kombination
        # mit Categorical Cross Entropy verwendet).
        return gradient_raw_outputs

class Network(Network):
    def __init__(self, cost_function):
        self.layers = []
        self.activation_functions = []
```

```
        self.cost_function = cost_function

    def backpropagation(self, predictions, targets):
        # Gradient der Cost Function in Bezug zu den Vorhersagen (dJ/dY).
        # Beim Categorical Cross Entropy + Softmax sind diese allerdings
        # im Bezug zu den rohen Ausgaben der Ausgabenschicht (dJ/dZ).
        gradient_predictions = self.cost_function.backwards(predictions, \
                                                targets)
        # Der Gradient der Vorhersagen ist identisch mit dem Gradient der
        # aktivierten Ausgaben der Ausgabenschicht
        gradient_activated_outputs = gradient_predictions
        # Rückwärts berechnet, von Ausgabeschicht zu Eingabeschicht.
        for layer, activation_function in zip(
            reversed(self.layers), reversed(self.activation_functions)
        ):
            # Gradient der Cost Function in Bezug zu den aktivierten Ausgaben
            # der aktuellen Schicht(dJ/dA).
            gradient_raw_outputs = activation_function.backwards(
                gradient_activated_outputs
            )

            # Gradienten der Cost Function in Bezug zu den Gewichten (dJ/dW)
            # und den Bias-Werten der aktuellen Schicht (dJ/db).
            # Berechnet zusätzlich den Gradienten der Cost Function in
            # Bezug zu den rohen Ausgaben der vorherigen Schicht(dJ/dZ).
            gradient_activated_outputs = layer.backwards(gradient_raw_outputs)

class SGD(GD):
    def create_batches(self, inputs, targets, batch_size):
        N = inputs.shape[1]  # Anzahl an Trainingsdaten
        # Trainingsset in mini Batches eingeteilt
        batches = []
        for i in range(0, N, batch_size):
            batch_inputs = inputs[:, i : i + batch_size]
            batch_targets = targets[:, i : i + batch_size]
            single_batch = (batch_inputs, batch_targets)
            batches.append(single_batch)
        return batches

    def optimise(
        self,
        inputs,
        targets,
    ):
        # Vorwärtsdurchlauf: Berechnung der Vorhersagen
```

```
        predictions = self.network.forward_propagation(inputs)

        cost = self.network.cost_function.calculate_cost(predictions, targets)

        # Rückwärtsdurchlauf: Berechnung der Gradienten
        self.network.backpropagation(predictions, targets)

        # Aktualisiert die Gewichte und Bias-Werte basierend auf
        # die Gradienten
        self.update_parameters()
        return cost

    def train(self, inputs, targets, batch_size):
        batches = self.create_batches(inputs, targets, batch_size)
        cost_history = []
        avg_epoch_cost = 1
        while avg_epoch_cost > 0.2:
            epoch_cost_history = []
            for batch_inputs, batch_targets in batches:
                cost = self.optimise(batch_inputs, batch_targets)
                self.optimise(batch_inputs, batch_targets)

                epoch_cost_history.append(cost)
                cost_history.append(cost)
            avg_epoch_cost = np.mean(epoch_cost_history)

        return cost_history

# Categorical Cross Entropy als Cost Function
network = Network(CategoricalCrossEntropy)
network.add_layer(
    Layer(784, 20),  # Eingabeschicht → versteckte Schicht
    ReLU(),  # Aktivierungsfunktion für die versteckte Schicht
)
network.add_layer(
    Layer(20, 10),  # Versteckte Schicht → Ausgabeschicht
    Softmax(),  # Aktivierungsfunktion für die Ausgabeschicht
)

training_images, training_labels = load_trainings_data()

sgd = SGD(network, learning_rate=0.1)
cost_history = sgd.train(
    training_images,
    training_labels,
```

```
    batch_size=64,
)

test_neural_network(network)

class Item:
    def __init__(self, name, mass, value):
        self.name = name
        self.mass = mass
        self.value = value


item_list = [
    # Handy mit 3kg Masse und einem Geldwert von 5€
    Item("Handy", mass=3, value=5),
    Item("Laptop", 6, 10),
    Item("Diamant", 1, 30),
    Item("Brot", 1, 1),
]


class Individual:
    def __init__(self, item_bits):
        self.item_bits = item_bits

    def print_items_in_backpack(self):
        for item, is_in_backpack in zip(item_list, self.item_bits):
            if is_in_backpack:
                print(
                    f"Gegenstand: {item.name}, Masse: {item.mass}kg, \
                    Geldwert: {item.value}€"
                )


individual = Individual([0, 1, 0, 1])

individual.print_items_in_backpack()

import random


class Individual(Individual):
    def create_random_individual():
        random_item_bits = []
        for _ in item_list:  # für jedes Element in der Gegenstände Liste
```

```
            bit = random.choice([0, 1])  # zufällig 1 oder 0 wählen
            random_item_bits.append(bit)

        return Individual(random_item_bits)


class Population:
    def __init__(self, population_size):
        self.population_size = population_size
        self.create_initial_population()

    def create_initial_population(self):
        self.population = []
        while population_size > len(self.population):
            individual = Individual.create_random_individual()
            self.population.append(individual)

    def print_population(self):
        for individual in self.population:
            print(individual.item_bits)


population_size = 8
population = Population(population_size)
population.print_population()

class Individual(Individual):
    def calculate_fitness(self, mass_limit):
        total_mass = 0
        total_value = 0
        # Gehe jeden Gegenstand des Individuums durch
        for item, is_in_backpack in zip(item_list, self.item_bits):
            if is_in_backpack:
                total_mass += item.mass
                total_value += item.value

        if total_mass > mass_limit:
            self.fitness_value = 0
            return

        self.fitness_value = total_value


class Population(Population):
    def calculate_fitness(self, mass_limit):
```

```
        for individual in self.population:
            individual.calculate_fitness(mass_limit)


mass_limit = 5
population = Population(population_size)
population.calculate_fitness(mass_limit)

def tournament(enemy1, enemy2):
    if enemy1.fitness_value > enemy2.fitness_value:
        return enemy1
    else:
        return enemy2


def selection(population):
    enemies = random.sample(population, 4)  # 4 zufällige Individuuen
    winner1 = tournament(enemies[0], enemies[1])
    winner2 = tournament(enemies[2], enemies[3])
    return [winner1, winner2]


selection(population.population)

def crossover_parents(parent1, parent2):
    bits_amount = len(parent1.item_bits)
    half_amount = int(bits_amount / 2)

    # Erste Hälfte von Elternteil 1 plus zweite Hälfte von Elternteil 2
    child1_bits = parent1.item_bits[:half_amount] + parent2.item_bits[half_amount:]

    # Erste Hälfte von Elternteil 2 plus zweite Hälfte von Elternteil 1
    child2_bits = parent2.item_bits[:half_amount] + parent1.item_bits[half_amount:]

    child1 = Individual(child1_bits)
    child2 = Individual(child2_bits)
    return (child1, child2)

def mutate_child(individual):
    bits_amount = len(individual.item_bits)
    random_bit = random.randrange(bits_amount)
    individual.item_bits[random_bit] = (
        1 - individual.item_bits[random_bit]
    )  # 1 wird zu null und umgekehrt
```

```
class Population(Population):
    def create_new_population(self):
        new_population = []
        best_individuals = self.population[0:2]
        new_population.extend(best_individuals)

class Population(Population):
    def create_new_population(self):
        new_population = []
        best_individuals = self.population[0:2]
        new_population.extend(best_individuals)
        while population_size > len(new_population):
            parent1, parent2 = selection(self.population)

            child1, child2 = crossover_parents(parent1, parent2)

            mutate_child(child1)
            mutate_child(child2)

            new_population.append(child1)
            new_population.append(child2)

        return new_population

class Population(Population):
    def start(self, mass_limit):
        self.calculate_fitness(mass_limit)
        for _ in range(500):
            self.population = self.create_new_population()
            self.calculate_fitness(mass_limit)

            self.population.sort(
                reverse=True, key=lambda individual: individual.fitness_value
            )

            best_individual = self.population[0]

        print(best_individual.fitness_value)
        print(best_individual.item_bits)


population_size = 20
mass_limit = 3000
Population(population_size).start(mass_limit)
```

```
# Code für NEAT KI

import pygame
import neat
import random

pip install pygame-ce

# Konfigurations Datei erstellen

config = """
[NEAT]
fitness_criterion     = max
fitness_threshold     = 100000000
pop_size              = 100
reset_on_extinction   = True

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.01
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.01
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5
```

```
# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 6
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 30
weight_min_value        = -30
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 2.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 20
species_elitism      = 2

[DefaultReproduction]
elitism            = 3
```

```
survival_threshold = 0.2
"""

config_file = open("config.txt", "w")
config_file.write(config)
config_file.close()

# Code zum Erstellen von Irrgarten Sturkturen / Bauplan des Irrgarten als Matrix dargeste
llt


WALL = 1
GOAL = 2

class MazeGenerator:
    @staticmethod
    def generate_maze(size):
        """
        Erstellt ein Irrgarten der angegebenen Größe und gibt es als 2D-Liste zurück.
        """
        grid = MazeGenerator.create_empty_grid(size)
        MazeGenerator.generate_paths(grid, size)
        maze = MazeGenerator.transform_maze(grid, size)
        maze = MazeGenerator.add_borders(maze, size)
        return maze

    @staticmethod
    def create_empty_grid(size):
        """
        Erstellt ein leeres Gitter für das Labyrinth, wobei jede Zelle drei Werte enthält:
        [Ostwand (1 = vorhanden), Südwand (1 = vorhanden), besucht (0 = unbesucht, 1
= besucht)].
        """
        cell_template = [1, 1, 0]  # Standardwerte für jede Zelle
        grid = []
        for _ in range(size):
            row = []
            for _ in range(size):
                row.append(cell_template.copy())
            grid.append(row)
        return grid

    @staticmethod
    def generate_paths(grid, size):
        """
```

```
Erzeugt zufällige Pfade im Labyrinth mit dem Tiefensuchalgorithmus (DFS).
"""
stack = [(0, 0)]  # Startpunkt (oben links)
visited_cells = 1
total_cells = pow(size, 2)
grid[0][0][2] = 1  # Markiere Startzelle als besucht
while visited_cells < total_cells:
    x, y = stack[-1]  # Letzte Position im Stack
    neighbors = []

    # Prüfe alle möglichen Nachbarn
    if y > 0 and not grid[y - 1][x][2]:  # Nord
        neighbors.append(0)

    if y < size - 1 and not grid[y + 1][x][2]:  # Süd
        neighbors.append(1)

    if x > 0 and not grid[y][x - 1][2]:  # West
        neighbors.append(2)

    if x < size - 1 and not grid[y][x + 1][2]:  # Ost
        neighbors.append(3)

    if len(neighbors) == 0:
        stack.pop()  # Kein unbesuchter Nachbar -> Zurückgehen
        continue

    rand_i = random.randrange(0, len(neighbors))

    match neighbors[rand_i]:
        case 0:  # Nord
            stack.append((x, y - 1))
            grid[y - 1][x][1] = 0  # Entferne Südwand der neuen Zelle
            grid[y - 1][x][2] = 1  # Markiere als besucht
        case 1:  # Süd
            stack.append((x, y + 1))
            grid[y][x][1] = 0  # Entferne Südwand der aktuellen Zelle
            grid[y + 1][x][2] = 1
        case 2:  # West
            stack.append((x - 1, y))
            grid[y][x - 1][0] = 0  # Entferne Ostwand der neuen Zelle
            grid[y][x - 1][2] = 1
        case 3:  # East
            stack.append((x + 1, y))
            grid[y][x][0] = 0  # Entferne Ostwand der aktuellen Zelle
```

```
                grid[y][x + 1][2] = 1

            visited_cells += 1

    @staticmethod
    def transform_maze(grid, size):
        """
        Wandelt das Irrgarten Gitter in eine Matrix um. Eine Zelle ist entweder ein Ziel, ein
e Wand oder lehr.
        """
        maze_size = size * 2 - 1
        maze = []
        for _ in range(maze_size):
            row = [0] * (maze_size)
            maze.append(row)
        for row_index, row in enumerate(grid):
            for col_index, cell in enumerate(row):
                if (2 * col_index + 1) < maze_size and cell[0]:  # Ostwand
                    maze[2 * row_index][2 * col_index + 1] = WALL
                if (2 * row_index + 1) < maze_size and cell[1]:  # Südwand
                    maze[2 * row_index + 1][2 * col_index] = WALL
                if (2 * row_index + 1) < maze_size and (2 * col_index + 1) < maze_size:
                    maze[2 * row_index + 1][2 * col_index + 1] = WALL  # Eckpunkte

        maze[-1][-1] = GOAL # Setz das Ziel
        return maze

    @staticmethod
    def add_borders(maze, maze_size):
        """
        Fügt eine durchgehende äußere Wand zum Irrgarten hinzu.
        """
        final_size = maze_size * 2 + 1
        for maze_row in maze:
            maze_row.insert(0, 1)  # Links Wand hinzufügen
            maze_row.append(1)  # Rechts Wand hinzufügen
        maze.insert(0, [1] * final_size)  # Oben Wand hinzufügen
        maze.append([1] * final_size)  # Unten Wand hinzufügen
        return maze

# Code zum erstellen von Irrgarten Objeckten (Mit Rendering)


class MazeRendererWithCollision:
    def __init__(self, maze_size: int, cell_width: int):
```

```
        """
        Initialisiert den Renderer und Kollisionslogik für das Labyrinth.
        """
        self.maze = MazeGenerator.generate_maze(maze_size)
        self.cell_width = cell_width
        maze_surface_size = len(self.maze) * cell_width # Gesamtgröße des Labyrinths in Pixeln
        self.image = pygame.Surface((maze_surface_size, maze_surface_size)) # Erstelle eine Oberfläche für das Labyrinth
        self.rect = self.image.get_rect() # Rechteck für das Labyrinth
        self.boxes: list[pygame.Rect] = []  # Liste zur Speicherung der Rechtecke für Wände und Wege und für Kollision
        self.boxes_type: list[bool] = [] # Liste, die angibt, ob eine Kollision Zelle ein Ziel ist
        self.path_cells: list[pygame.Rect] = [] # Liste der Zellen, die als Pfad markiert sind
        self.setup()

    def setup(self):
        """
        Rendert das Labyrinth auf die Oberfläche und speichert die Positionen der Zellen für Kollisionszwecke.
        """
        maze_cell = pygame.Surface((self.cell_width, self.cell_width))
        maze_cell.fill("Black")
        goal_cell = maze_cell.copy()
        goal_cell.fill("Blue")
        self.image.fill("White")

        current_rect = maze_cell.get_rect()

        for maze_row in self.maze:
            current_rect.x = 0
            for maze_block in maze_row:
                if maze_block == WALL:
                    wall = current_rect.copy()
                    self.image.blit(maze_cell, wall)
                    self.boxes.append(wall)
                    self.boxes_type.append(False)
                elif maze_block == GOAL:
                    wall = current_rect.copy()
                    self.image.blit(goal_cell, current_rect)
                    self.boxes.append(wall)
                    self.boxes_type.append(True)
                else:
                    path_cell = current_rect.copy()
                    self.path_cells.append(path_cell)
```

```
                current_rect.x += self.cell_width
            current_rect.y += self.cell_width

        self.draw_grid_lines()

    def draw_grid_lines(self):
        """
        Zeichnet die Gitterlinien für das Labyrinth (die Trennlinien zwischen den Zellen).
        """
        for i in range(len(self.maze) + 1):
            pygame.draw.line(
                self.image,
                "Gray",
                (0, self.cell_width * i),
                (self.image.width, self.cell_width * i),
                2,
            )  # Vertikale Linie
            pygame.draw.line(
                self.image,
                "Gray",
                (self.cell_width * i, 0),
                (self.cell_width * i, self.image.width),
                2,
            )  # Horizontale Linie

    def draw(self, screen: pygame.Surface):
        """
        Zeichnet das Labyrinth auf den angegebenen Bildschirm.
        """
        screen.blit(self.image, self.rect)



# Code für den Raycaster

class Raycaster:
    @staticmethod
    def raycast_horizontal(
        maze, cell_width: int, rect: pygame.FRect, angle: float
    ) -> tuple[float, float, float, bool]:
        """
        Raycasting in horizontaler Richtung (über Zellen hinweg), um die erste Wand oder
das Ziel zu treffen.
        :return: (Ray-Koordinaten, Ray-Länge, ob das Ziel erreicht wurde).
        """
```

```
ray_x = 0
ray_y = 0
off_x = 0
off_y = 0
goal = False

maze_size = len(maze)
maze_width = maze_size * cell_width

if angle == 0:
    return (0, 0, maze_width, goal)
inverse_tan = 1 / math.tan(angle)

if angle > math.pi:  # forward
    ray_y = math.floor(rect.centery / cell_width) * cell_width - 0.0001
    ray_sin = rect.centery - ray_y
    ray_cos = inverse_tan * ray_sin
    ray_x = rect.centerx - ray_cos
    off_y = -cell_width
    off_x = off_y * inverse_tan
elif angle < math.pi:  # backward
    ray_y = math.ceil(rect.centery / cell_width) * cell_width
    ray_sin = rect.centery - ray_y
    ray_cos = inverse_tan * ray_sin
    ray_x = rect.centerx - ray_cos
    off_y = cell_width
    off_x = off_y * inverse_tan
else:  # left or right
    return (0, 0, maze_width, goal)

depth = maze_size
while depth:
    mapx: int = int(ray_x / cell_width)
    mapy: int = int((ray_y / cell_width))
    inside_maze = (mapx < maze_size) and (mapx >= 0)
    if not inside_maze:
        return (0, 0, maze_width, goal)
    wall = maze[mapy][mapx]
    if wall:
        if wall == 2:
            goal = True
        ray_len_x = rect.centerx - ray_x
        ray_len_y = rect.centery - ray_y
        ray_len = math.sqrt(math.pow(ray_len_x, 2) + math.pow(ray_len_y, 2))
        return (ray_x, ray_y, ray_len, goal)
```

```
            ray_x += off_x
            ray_y += off_y
            depth -= 1
        return (0, 0, maze_width, goal)

    @staticmethod
    def raycast_vertical(
        maze, cell_width: int, rect: pygame.FRect, angle: float
    ) -> tuple[float, float, float, bool]:
        """
        Raycasting in vertikaler Richtung (über Zellen hinweg), um die erste Wand oder das Ziel zu treffen.
        :return: (Ray-Koordinaten, Ray-Länge, ob das Ziel erreicht wurde).
        """
        ray_x = 0
        ray_y = 0
        off_x = 0
        off_y = 0
        goal = False

        maze_size = len(maze)
        maze_width = maze_size * cell_width

        if (angle > math.pi / 2) and (angle < 3 * math.pi / 2):  # left
            ray_x = math.floor(rect.centerx / cell_width) * cell_width - 0.0001
            ray_cos = rect.centerx - ray_x
            ray_sin = math.tan(angle) * ray_cos
            ray_y = rect.centery - ray_sin
            off_x = -cell_width
            off_y = off_x * math.tan(angle)
        elif (angle < math.pi / 2) or (angle > 3 * math.pi / 2):
            ray_x = math.ceil(rect.centerx / cell_width) * cell_width
            ray_cos = rect.centerx - ray_x
            ray_sin = math.tan(angle) * ray_cos
            ray_y = rect.centery - ray_sin
            off_x = cell_width
            off_y = off_x * math.tan(angle)
        else:  # forward or backwards
            return (0, 0, maze_width, goal)

        depth = maze_size
        while depth:
            mapx: int = int(ray_x / cell_width)
            mapy: int = int((ray_y / cell_width))
            inside_maze = (mapy < maze_size) and (mapy >= 0)
```

```
            if not inside_maze:
                return (0, 0, maze_width, goal)
            wall = maze[mapy][mapx]
            if wall:
                if wall == 2:
                    goal = True
                ray_len_x = rect.centerx - ray_x
                ray_len_y = rect.centery - ray_y
                ray_len = math.sqrt(math.pow(ray_len_x, 2) + math.pow(ray_len_y, 2))
                return (ray_x, ray_y, ray_len, goal)

            ray_x += off_x
            ray_y += off_y
            depth -= 1
        return (0, 0, maze_width, goal)

    @staticmethod
    def raycast(maze, rect: pygame.FRect, angle: float):
        """
        Führt sowohl horizontales als auch vertikales Raycasting durch und gibt den kürzer
en Ray zurück.
        """
        ray_h = Raycaster.raycast_horizontal(maze.maze, maze.cell_width, rect, angle)
        ray_v = Raycaster.raycast_vertical(maze.maze, maze.cell_width, rect, angle)
        ray = min(ray_h, ray_v, key=lambda ray: ray[2])
        return ray

    @staticmethod
    def raycasting(
        maze,
        rect: pygame.FRect,
        player_angle: float,
        fov: int,
        amount: int,
    ):
        """
        Führt Raycasting für mehrere Strahlen durch (für das Sichtfeld des Spielers und der
KI).
        :return: Eine Liste von Strahlen mit deren Koordinaten und Längen
        """
        fov_rad = math.radians(fov)
        fov_step = fov_rad / amount
        rays: list[tuple[float, float, float, float, bool]] = []
        for current_step in range(amount):
            angle = player_angle - fov_rad / 2 + fov_step * current_step
```

```
            if angle < 0:
                angle += 2 * math.pi
            if angle > 2 * math.pi:
                angle -= 2 * math.pi
            ray = Raycaster.raycast(maze, rect, angle)
            ray_length = ray[2]
            if ray[3] == True:
                ray_length = maze.image.width
            no_fish_angle = player_angle - angle
            if no_fish_angle < 0:
                no_fish_angle += 2 * math.pi
            if no_fish_angle > 2 * math.pi:
                no_fish_angle -= 2 * math.pi
            no_fish_length = math.cos(no_fish_angle) * ray[2]
            rays.append((ray[0], ray[1], ray_length, no_fish_length, ray[3]))
        return rays

# Code für Player Klasse

import neat
from pygame import rect, key
import pygame
import math


class Player:
    speed = 4
    fov = 180
    rays_amount = 6
    LIFE_TIME = 40

    def __init__(
        self,
        pos: tuple[int, int],
        radius: int,
        boxes: list[pygame.Rect],
        boxes_type: list[bool],
        path_cells: list[pygame.Rect],
        genome,
        net: neat.nn.FeedForwardNetwork,
        maze_width: int,
        cell_width: int,
        best_genome: bool,
    ) -> None:
        self.image = pygame.Surface((radius * 2, radius * 2))
```

```
        self.rect: rect.FRect = self.image.get_frect(center=pos)
        self.boxes = boxes
        self.boxes_type = boxes_type
        self.path_cells = path_cells
        self.path_cells_score: list[int] = [0] * len(self.path_cells)
        self.genome = genome
        self.net = net
        self.best_genome = best_genome
        self.life_time = self.LIFE_TIME
        self.total_time = 0

        self.maze_width = maze_width
        self.cell_width = cell_width
        self.maze_inside_width = maze_width - cell_width * 2

        self.image.set_colorkey((0, 0, 0))
        pygame.draw.circle(self.image, "Red", (radius, radius), radius)

        self.direction = pygame.Vector2()
        self.angle = 0
        self.angle_direction = pygame.Vector2()
        self.rays: list[tuple[float, float, float, float, bool]] = []

    def wasd_input(self):
        keys = key.get_pressed()
        self.direction.x = int(keys[pygame.K_d]) - int(keys[pygame.K_a])
        self.direction.y = int(keys[pygame.K_s]) - int(keys[pygame.K_w])
        if self.direction:
            self.direction = self.direction.normalize()

    def angle_input(self):
        keys = key.get_pressed()
        # self.get_ai_input_data()
        self.angle += (keys[pygame.K_d] - keys[pygame.K_a]) / 10
        if self.angle < 0:
            self.angle += 2 * math.pi
        if self.angle > 2 * math.pi:
            self.angle -= 2 * math.pi
        direction = keys[pygame.K_w] - keys[pygame.K_s]
        self.angle_direction.x = math.cos(self.angle)
        self.angle_direction.y = math.sin(self.angle)
        self.direction.x = self.angle_direction.x * direction
        self.direction.y = self.angle_direction.y * direction

    def get_ai_input_data(self):
```

```
        inputs = []
        for ray in self.rays:
            normalised_ray = ray[2] / (
                self.maze_inside_width
            )  # when diagonal, potential to still be above 1
            inputs.append(normalised_ray)
        return inputs

    def ai_input(self):
        inputs = self.get_ai_input_data()
        output = self.net.activate(inputs)
        self.angle -= 0.1  # left
        if max(output[0], 0):  # if below 0 then 0
            self.angle += 0.2  # right
        direction = 0
        if max(output[1], 0):  # if below 0 then 0
            direction = 1  # forward
        if self.angle < 0:
            self.angle += 2 * math.pi
        if self.angle > 2 * math.pi:
            self.angle -= 2 * math.pi
        self.angle_direction.x = math.cos(self.angle)
        self.angle_direction.y = math.sin(self.angle)
        self.direction.x = self.angle_direction.x * direction
        self.direction.y = self.angle_direction.y * direction

    def raycasting(self, maze):
        self.rays = Raycaster.raycasting(
            maze, self.rect, self.angle, self.fov, self.rays_amount
        )

    def move(self):
        self.rect.x += self.direction.x * self.speed
        self.rect.y += self.direction.y * self.speed
        self.collision()

    def collision(self):
        collision_index = self.rect.collidelist(self.boxes)
        if collision_index != -1:
            is_goal = self.boxes_type[collision_index]
            # Tod nach Berührung mit Zelle. Belohnung falls Ziel und Bestrafung falls Wand.
            if is_goal:
                self.genome.fitness += 10000
                self.genome.fitness -= self.total_time
            else:
```

```
            self.genome.fitness -= 65
        self.life_time = 0
        return

    def path_collision(self):
        collision_index = self.rect.collidelist(self.path_cells)
        if collision_index != -1:
            if self.path_cells_score[collision_index] == 0:  # new cell
                self.life_time += 20
                self.genome.fitness += 8
            self.path_cells_score[collision_index] += 1
            if self.path_cells_score[collision_index] > 50:
                self.life_time = 0
                self.genome.fitness -= 80

    def update(self, maze):
        self.raycasting(maze)
        self.ai_input()
        self.move()
        self.path_collision()
        self.life_time -= 1
        self.total_time += 1

    def draw_rays(self, screen: pygame.Surface):
        for ray in self.rays:
            if ray[4]:
                pygame.draw.line(screen, "Blue", self.rect.center, (ray[0], ray[1]), 2)
            else:
                pygame.draw.line(screen, "Green", self.rect.center, (ray[0], ray[1]), 2)

    def draw_look_direction(self, screen: pygame.Surface):
        end_line_x = self.rect.centerx + self.angle_direction.x * 20
        end_line_y = self.rect.centery + self.angle_direction.y * 20
        pygame.draw.line(screen, "Blue", self.rect.center, (end_line_x, end_line_y), 2)

    def draw(self, screen: pygame.Surface, maze):
        self.draw_look_direction(screen)
        if self.best_genome:
            self.draw_rays(screen)
            self.draw_3D(screen, maze)
        screen.blit(self.image, self.rect)

    def draw_3D(self, screen: pygame.Surface, maze):
        line_width = int(self.maze_width / self.rays_amount / 40)
        current_x = self.maze_width + line_width / 2
```

```
        rays = Raycaster.raycasting(
            maze, self.rect, self.angle, 90, self.rays_amount * 40
        )
        for ray in rays:
            length = self.maze_width / ray[3] * self.cell_width
            length = min(length, self.maze_width)
            if ray[4]:
                pygame.draw.line(
                    screen,
                    "Blue",
                    (current_x, self.maze_width / 2 - length / 2),
                    (current_x, self.maze_width / 2 + length / 2),
                    line_width,
                )
            else:
                pygame.draw.line(
                    screen,
                    "Green",
                    (current_x, self.maze_width / 2 - length / 2),
                    (current_x, self.maze_width / 2 + length / 2),
                    line_width,
                )
            current_x += line_width

    def ai_view(self, screen: pygame.Surface, maze):
        line_width = int(self.maze_width / 6)
        current_x = self.maze_width + line_width / 2
        rays = Raycaster.raycasting(
            maze, self.rect, self.angle, 90, 6
        )
        for ray in rays:
            length = self.maze_width / ray[2] * self.cell_width
            length = min(length, self.maze_width)
             # Calculate brightness (closer = brighter, farther = darker)

            brightness = max(0, min(255, int(255 / (1 + ray[3] * 0.05))))

            if ray[4]:  # If hit a wall
                color = (0, 0, brightness)  # Blue with darkness effect
            else:
                color = (0, brightness, 0)  # Green with darkness effect
            pygame.draw.line(
                screen,
                color,
                (current_x, 0),
```

```
                (current_x, self.maze_width),
                line_width,
            )
            current_x += line_width

# Code für Game Klasse

import neat
import pygame


class Game:
    TOTAL_WIDTH = 960
    TOTAL_HEIGHT = 720
    FPS = 60

    def __init__(self, max_rounds):
        pygame.init()
        self.maze = MazeRendererWithCollision(10, 40)
        self.screen = pygame.display.set_mode(
            (self.maze.image.width * 2, self.maze.image.height)
        )
        self.clock = pygame.time.Clock()
        self.running = True

        self.players: list[game.Player] = []
        self.ticks = 0
        self.round = 0
        self.max_rounds = max_rounds

    def setup(self, genomes, config, best_genome):
        if self.round > self.max_rounds:
            self.maze = MazeRendererWithCollision(10, 40)
            self.round = 0
            if self.max_rounds > 0:
                self.max_rounds -= 0.5
        posx = int(self.maze.cell_width * 1.5)
        best_genome_id = best_genome[0]
        for i, genome in genomes:
            best_genome = False
            if i == best_genome_id:
                best_genome = True
            net = neat.nn.FeedForwardNetwork.create(genome, config)
            player = Player(
                (posx, posx),
```

```
                5,
                self.maze.boxes,
                self.maze.boxes_type,
                self.maze.path_cells,
                genome,
                net,
                self.maze.image.width,
                self.maze.cell_width,
                best_genome,
            )
            self.players.append(player)

    def update(self):
        for i, player in enumerate(self.players):
            player.update(self.maze)
            if player.life_time <= 0:
                # print("fitness:", player.genome.fitness)
                del self.players[i]
        self.ticks += 1

    def draw(self):
        self.screen.fill("Black")
        self.maze.draw(self.screen)
        for player in self.players:
            player.draw(self.screen, self.maze)
        pygame.display.update()

# Code für Main Klasse

from typing import Any
import pygame
import os
import neat
import pickle


def eval_genomes(genomes: list[Any], config, game: Game, render: bool):
    def best_genome_max(genome): # Used for rendering while training
        if genome[1].fitness == None:
            return -100
        return genome[1].fitness

    best_genome = max(genomes, key=best_genome_max)
    for _, genome in genomes:
        genome.fitness = 0
```

```
    game.setup(genomes, config, best_genome)
    while len(game.players):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                game.running = False
        game.update()
        if render:
            game.draw()
            game.clock.tick(60)
    game.round += 1



def train_ai(config_file: str, game: Game, n_gen: int, render: bool, checkpoint: str):
    config = neat.Config(
        neat.DefaultGenome,
        neat.DefaultReproduction,
        neat.DefaultSpeciesSet,
        neat.DefaultStagnation,
        config_file,
    )
    # Create the population, which is the top-level object for a NEAT run.
    p = neat.Population(config)
    if checkpoint:
        p = neat.Checkpointer.restore_checkpoint(f"checkpoints/{checkpoint}")

    # Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    checkpointer = neat.Checkpointer(generation_interval=5, filename_prefix="checkpoi
nts/neat-checkpoint-")
    p.add_reporter(checkpointer)

    # Damit zusätzliche Parameter mitgegeben werden können
    def execute_eval_genomes_func(genomes, config):
        eval_genomes(genomes, config, game, render)

    # Run for up to "n_gen" amount of generations.
    winner = p.run(execute_eval_genomes_func, n_gen)
    with open("best.pickle", "wb") as f:
        pickle.dump(winner, f)


def test_ai(config_file: str, game: Game):
```

```
    config = neat.Config(
        neat.DefaultGenome,
        neat.DefaultReproduction,
        neat.DefaultSpeciesSet,
        neat.DefaultStagnation,
        config_file,
    )
    with open("best.pickle", "rb") as f:
        winner = pickle.load(f)
    while game.running:
        game.setup([[0, winner]], config, [0])
        while len(game.players):
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    game.running = False
            game.update()
            game.draw()
            game.clock.tick(60)
        game.round += 26

def execute_train_test(mode, n_gen=0, render=False, checkpoint=None, max_rounds=
2):
    if mode == "test":
        game = Game(max_rounds=2)
        test_ai("config.txt", game)
    else:
        game = Game(max_rounds)
        train_ai("config.txt", game, n_gen, render, checkpoint)
    pygame.quit()
```

Selbstständigkeitserklärung ABA

Name: Magomed Alimkhanov

Selbstständigkeitserklärung

Ich erkläre, dass ich diese abschließende Arbeit eigenständig angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

_____ _____

Ort, Datum Unterschrift

Zustimmung zur Aufstellung in der Schulbibliothek

Ich gebe mein Einverständnis, dass ein Exemplar meiner abschließenden Arbeit in der Schulbibliothek meiner Schule aufgestellt wird.

_____ _____

Ort, Datum Unterschrift

Hinweise:

Diese Erklärung ist mit der ausgedruckten Arbeit zu binden.

(Beim Hochladen der Arbeit auf die ABA-Datenbank bzw. bei der Abgabe per E-Mail oder auf einem digitalen Speichermedium sollte diese Erklärung nicht beigefügt werden.)