KANTONSSCHULE AM BURGGRABEN

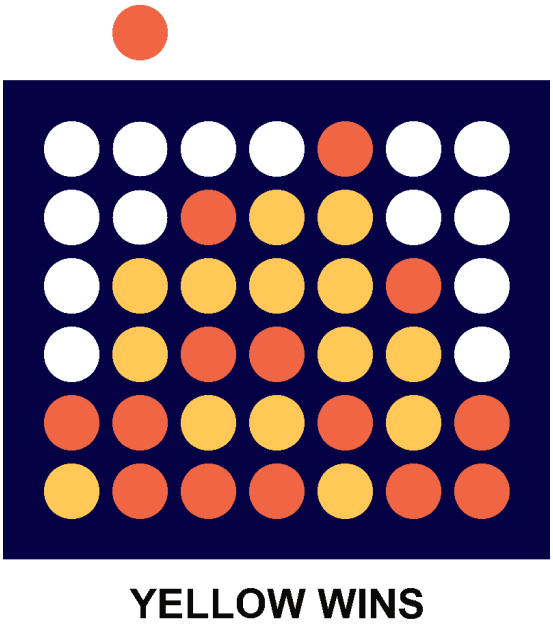
Kantonsschule am Burggraben St. Gallen

Maturaarbeit

Vier Gewinnt - oder doch nicht?

Programmierung zweier künstlichen Intelligenzen für das Spiel *Vier Gewinnt*

Vorgelegt durch:
Hannah Oss



Vorgelegt bei:
Dr. Ivo Blöchliger

25. Januar 2024

Inhaltsverzeichnis

Vorwort

Persönlich interessiere ich mich schon seit längerer Zeit für die Informatik, mit einem speziellen Fokus auf *competitive programming* [1].

Im Semester vor dem Beginn dieser Arbeit bot sich mir die Möglichkeit, im Rahmen der Begabtenförderung der KSBG ein Projekt zu machen. Ich entschied mich, ein neuronales Netzwerk in der Programmiersprache C++ auf Grundlage des Buches *Neural Networks and Deep Learning* von Michael Nielsen [2] zu programmieren, welches mir von einem Freund empfohlen wurde. Dabei wurde mein Interesse für künstliche Intelligenz geweckt, was mich dazu bewegte, auch in der Maturaarbeit ein Thema in diesem Bereich zu wählen. Besonders bot es sich an, weiter auf dem neuronalen Netzwerk, das ich bereits selbst programmiert hatte, aufzubauen. So musste nicht auf Bibliotheken wie *Tensorflow* [3] oder *PyTorch* [4] zurückgegriffen werden.

Weiter wurde ich durch den YouTube-Kanal *AI Warehouse* [5] inspiriert, Reinforcement-Learning zu verfolgen, eine Methode des maschinellen Lernens, in der die künstliche Intelligenz durch Interaktionen mit ihrer Umgebung lernt. Der YouTube-Kanal befasst sich vor allem mit dem Erlernen des Laufens durch Reinforcement-Learning, doch mit den mir zur Verfügung stehenden Ressourcen wäre dies zu komplex gewesen. Ich entschied mich, die Arbeit darauf zu beschränken, dass die künstliche Intelligenz ein einfaches Spiel erlernt.

Kapitel 1

Einleitung

Am 30. November 2022 hat OpenAI den Chat-Bot *ChatGPT* veröffentlicht [6]. Bereits zwei Monate nach der Veröffentlichung erreichte *ChatGPT* mehr als 100 Millionen aktive Nutzer und ist damit die schnellst wachsende App aller Zeiten [7]. Mit *ChatGPT* hat künstliche Intelligenz sehr viel öffentliche Aufmerksamkeit erhalten und war im Jahr 2023 ein grosses Thema in den Medien. *ChatGPT* wurde unter anderem durch eine Methode namens Reinforcement-Learning trainiert [6]. Zwei Algorithmen, die der Reinforcement-Learning Kategorie angehören, sind *Monte Carlo Tree Search* (MCTS) und *Deep Q-Learning* (DQN).

In dieser Maturaarbeit werden genau diese beiden Algorithmen gelernt und anschliessend auf das Spiel *Vier Gewinnt* angewandt. Das Programm wird hauptsächlich in der Programmiersprache C++ geschrieben. Die Sprache C++ wird dabei verwendet, da es eine der schnellsten Programmiersprachen ist [8]. Für das neuronale Netzwerk wird ausserdem CUDA verwendet. CUDA ist ein Programmiermodell, das mithilfe von NVIDIA-Grafikprozessoren (NVIDIA-GPUs) schnellere Berechnungen ermöglicht [9].

Der Aufbau der Arbeit gliedert sich in die Einleitung und sechs Kapitel, wobei jedes Kapitel zuerst die Theorie beschreibt und anschliessend auf die Implementation eingeht. Ausgenommen davon sind Kapitel 6 und Kapitel 7. Alle dargestellten Quellcodes sind vereinfacht und gekürzt. Sie sind eher als Pseudocode zu verstehen. Der vollständige Quellcode sowie ein README.md zur Verwendung des Programmes sind auf GitHub [10] zu finden.

In Kapitel 2 wird das Spiel *Vier Gewinnt* beschrieben. Das Spiel ist gut für diese Arbeit geeignet, da es ein simples Spiel ist, bei welchem trotz der Einfachheit gewisse Strategien entwickelt und angewendet werden können.

Das nächste Kapitel, Kapitel 3, beschreibt den MCTS-Algorithmus. Als bekannter Algorithmus für Strategiespiele gibt es bereits viele Untersuchungen, welche den MCTS-Algorithmus auf *Vier Gewinnt* angewendet haben. Darunter zum Beispiel eine Untersuchung, die MCTS und MCS (*Monte Carlo Search*) vergleicht [11].

Kapitel 4 geht auf neuronale Netzwerke ein. Dies ist nötig, da der DQN-Algorithmus auf einem neuronalen Netzwerk basiert.

Das Kapitel 5 beschreibt den DQN-Algorithmus. Der DQN-Algorithmus ist in der

Lage, viele Atari-Spiele zu meistern [12]. Auch für diesen Algorithmus gibt es bereits Untersuchungen, die ihn auf das Spiel *Vier Gewinnt* angewendet haben. Ein Beispiel dafür ist ein Blogpost von Oleg Sushkov [13].

Anschliessend werden in Kapitel 6 Vergleiche gemacht. Es wird die Laufzeit des neuronalen Netzwerkes mit und ohne Verwendung von CUDA verglichen. Ausserdem werden bestimmte Parameter des MCTS und DQN angepasst und gegenübergestellt. Als letzter Punkt werden die Algorithmen miteinander verglichen.

In Kapitel 7 wird die Arbeit reflektiert und es werden Möglichkeiten zur Erweiterung oder Vertiefung bestimmter Teile der Arbeit aufgezeigt.

Ein Grossteil der in dieser Arbeit verwendeten Quellen ist in englischer Sprache verfasst. Um das Verständnis der englischen Quellen zu erleichtern, werden in dieser Arbeit die englischen Fachausdrücke verwendet.

Alle in dieser Arbeit verwendeten Abbildungen sind selbst erstellt.

Kapitel 2

Das Spiel *Vier Gewinnt*

Das Spiel *Vier Gewinnt* ist ein Strategiespiel für zwei Personen. Jede Person spielt mit Spielsteinen in einer Farbe und versucht, eine Reihe (horizontal, vertikal oder diagonal) von vier Spielsteinen zu bilden.

Das Spiel wurde im Jahr 1988 von James Dow Allen und Victor Allis unabhängig voneinander gelöst [14]. Das bedeutet, dass sie eine Strategie gefunden haben, die immer optimal spielt. In *Vier Gewinnt* gewinnt ein optimaler Spieler immer, wenn er den ersten Zug spielt. Das Ziel dieser Arbeit ist es, die Algorithmen MCTS und DQN zu erlernen und nicht die Perfektion des Spieles *Vier Gewinnt*, weswegen die optimale Strategie bewusst nicht implementiert wird.

2.1 Programmierung

Bevor die computergesteuerten Spieler programmiert werden können, muss als Grundlage das Spiel programmiert werden.

2.1.1 Spiel

Als erster Schritt werden das Spiel und die Benutzeroberfläche programmiert, wofür die Open-Source-Bibliothek SDL2 [15] verwendet wird. Die Benutzeroberfläche besitzt insgesamt drei Screens. Auf dem Anfangsscreen kann ausgewählt werden, ob die Spieler manuell oder vom Computer gesteuert werden sollen. Je nach Auswahl muss der Name des manuellen Spielers oder ein Dateiname angegeben werden. Der Dateiname gibt an, aus welcher Datei der computergesteuerte Spieler geladen werden soll. Der zweite Screen ist das eigentliche Spiel und der dritte Screen dient dazu, den Gewinner anzuzeigen.

Um verschiedene Spieler vergleichen zu können, wird ein ELO-System anhand eines von Raghav Mittal verfassten Artikels [16] implementiert. Der ELO-System ist eine Methode, um die relative Spielstärke von Spielern zu berechnen und ist vor allem im Schach weit verbreitet. Jeder Spieler hat einen ursprünglichen ELO-Wert von 1000. Der K-Faktor, der die maximal mögliche Anpassung pro Spiel angibt, ist auf 32 fixiert. Dieser K-Faktor gilt als Standardwert [17].

2.1.2 Einbauen computergesteuerter Spieler

Für das Programmieren der computergesteuerten Spieler wird zunächst eine Player-Klasse definiert, die es ermöglicht, verschiedene die Spieler mit minimalen Veränderungen in das Spiel einzubauen. Diese Klasse kann in Quellcode 2.1 betrachtet werden.

Quellcode 2.1: Die Player-Klasse

```
1 struct Player {
2     float elo = 1000.0;
3 
4     virtual int get_col(connect_four_board board) = 0;
5     virtual void load(std::string filename) = 0;
6     virtual void save(std::string filename) = 0;
7     virtual void train(int num_games) {}
8 };
```

Jeder Spieler muss einen ELO-Wert und drei Funktionen besitzen. Eine **load**-Funktion, die es ermöglicht, den Spieler aus einer Datei zu laden, eine **save**-Funktion, welche den Spieler in einer Datei speichert und eine **get_col**-Funktion, welche den nächsten Zug des Spielers liefert. Das Speichern des Spielers ermöglicht es, den ELO-Wert und weitere allfällige Informationen zu den Spielern auch nach Programmende noch festzuhalten. Manche Algorithmen, wie zum Beispiel DQN, profitieren von einem Training. Dafür muss die **train**-Funktion implementiert werden. Dieses Training sieht je nach Algorithmus unterschiedlich aus. Grundsätzlich handelt es sich um Spiele, in denen der Spieler gegen sich selbst spielt und Erfahrungen sammelt. Diese Erfahrungen kann der Spieler nutzen, um besser Entscheidungen zu treffen. Damit die Spieler besser verglichen werden können, sammeln sie die Erfahrungen nur während des Trainings.

2.1.3 Zwei simple Spieler

Um bereits das Spielen gegen den Computer zu ermöglichen, werden zwei Spieler implementiert. Der erste Spieler handelt komplett zufällig. Der zweite Spieler schaut jeweils zwei Züge in die Zukunft und entscheidet aufgrund dieser Informationen seinen Zug. Falls er das Spiel mit einem Zug gewinnen kann, spielt er diesen. Falls der Gegner drei Spielsteine in einer Reihe hat, spielt der Spieler, wenn möglich, einen zufälligen Zug, der verhindert, dass der Gegner seine Reihe vervollständigen kann. Ansonsten handelt er rein zufällig.

Kapitel 3

Der Algorithmus MCTS

MCTS ist einer der bekanntesten Algorithmen zum Finden eines guten Zugs in einem rundenbasierten Spiel [18]. Es ist ein Algorithmus, der oft für Brettspiele verwendet wird, da er sehr gute Resultate erzielt. Zum Beispiel verwendet *AlphaGo Zero*, ein Algorithmus, der gegen die weltbesten Go-Spieler gewinnt, eine Variante des MCTS-Algorithmus [19].

3.1 Grundlagen

Dieses Unterkapitel dient zur Erklärung der Funktionsweise des MCTS-Algorithmus. Dabei stammen die Informationen aus einem von Michael Liu verfassten Artikel [20]. Dieser Artikel ist aus dem Jahr 2017, doch die Inhalte sind immer noch aktuell.

3.1.1 Begriffserklärungen

Die Erklärung des MCTS-Algorithmus setzt Vorwissen im Bereich der Graphentheorie voraus. Die Begriffe der Graphentheorie, die in dieser Arbeit häufig verwendet werden, sind im Folgenden erklärt. Für eine genauere Erklärung wird auf [21] verwiesen.

Node

Eine Node ist ein Knoten in einem Graphen.

Tree

Ein Tree ist ein Graph, der zusammenhängend und azyklisch ist. Das heisst, dass von jeder Node des Graphen jede andere Node über genau einen Weg erreichbar ist.

Root-Node

In einem Tree wird oft eine Node als Root-Node r bezeichnet. Die Root-Node ist die Node, die als Ursprung des Trees angesehen werden kann.

Parent-Node

Jede Node v ausser der Root-Node r hat eine direkte Verbindung zu genau einer Node p, die auf dem Weg zwischen v und r liegt. Diese Node p wird als Parent-Node von v bezeichnet.

Child-Node

Die Child-Nodes einer Node v sind alle Nodes, die v als Parent-Node haben.

Ancestor

Die Ancestors einer Node v sind alle Nodes, die auf dem Weg zwischen r und v liegen. Dabei wird r ebenfalls dazu gezählt.

3.1.2 Algorithmus

Wie der Name *Monte Carlo Tree Search* schon verrät, durchsucht der Algorithmus den Spiel-Tree. Dabei werden die schon erforschten Spielstände in einem Tree abgespeichert, der nach und nach weiter ausgebaut wird.

Bevor der MCTS-Spieler den nächsten Spielzug wählt, geht er durch folgende vier Phasen:

1. Selection

   Ausgehend vom momentanen Spielstand wird der bereits bekannte Tree abgegangen, wobei immer zu der Node mit dem höchsten UCB-1-Wert gegangen wird. Der UCB-1-Wert ist definiert als $\frac{w_v}{n_v} + c\sqrt{\frac{\ln n_p}{n_v}}$, wobei w_v die Anzahl der gewonnenen Simulationen ausgehend von der Node v, n_v die totale Anzahl der Simulationen ausgehend von der Node v und n_p die totale Anzahl der Simulationen ausgehend von der Parent-Node p ist. Der Parameter c ist der Exploration-Parameter, welcher steuert, wie fest die Erkundung des Spiel-Trees gegenüber dem Besuchen von gut eingeschätzten Spielständen gewichtet werden soll. Dieser besitzt in Theorie den Wert $\sqrt{2}$. Sobald eine Node erreicht ist, welche noch nicht besuchte Child-Nodes hat, wird die Expansion gestartet.

2. Expansion

   In der Expansion wird der Tree des MCTS-Spielers durch eine weitere Node erweitert. Dies geschieht dadurch, dass er zufällig eine der noch nicht besuchten Child-Nodes auswählt und dem Tree hinzufügt, indem er die ausgewählte Child-Node besucht.

3. Simulation

   Der neu besuchten Child-Node wird ein Wert w_v zugeordnet, indem das Spiel von diesem Spielstand aus mehrfach simuliert wird. Je nach Ausgang des Spiels wird

der Wert w_v der Child-Node um +1, 0 oder -1 angepasst. Ausserdem wird n_v pro Simulation um +1 erhöht.

4. Backup

   Die berechneten Werte w_v und n_v der Node v werden anschliessend an die Ancestors zurückpropagiert. Das geschieht dadurch, dass w_v und n_v entsprechend zu den Werten w_a und n_a der Ancestorts addiert werden.

Diese vier Phasen werden mehrmals durchgeführt, bevor ein Zug gewählt wird. Eine Durchführung der vier Phasen wird als Iteration bezeichnet. Anschliessend wählt der MCTS-Spieler den nächsten Zug, indem er zu der Child-Node mit dem besten Wert $\frac{w_v}{n_v}$ geht.

3.2 Programmierung

Um den MCTS-Algorithmus zu programmieren, muss die Player-Klasse aus Quellcode 2.1 implementiert werden. Es wird auch die **train**-Funktion implementiert, da der MCTS-Spieler durch Erfahrungen seinen Spiel-Tree erweitern kann und somit dazu lernt. Bevor mit der Implementierung begonnen werden kann, braucht es noch einige Überlegungen zur Darstellung des Spiels und der Spielstände.

3.2.1 Darstellung von Spiel und Spielständen

Das Spiel *Vier Gewinnt* ist eher durch einen DAG (gerichteter, azyklischer Graph) repräsentierbar, denn ein Spielstand kann von verschiedenen Spielständen aus durch einen Zug erreicht werden. In dieser Arbeit wird das Spiel jedoch durch einen Tree dargestellt, damit der MCTS-Algorithmus in seiner originalen Form verwendet werden kann.

Die Nodes des Trees werden als eine Ganzzahl repräsentiert. Dabei wird Gleichung 3.1 verwendet, um die Nummer einer Node v zu berechnen, wobei p die Nummer der Parent-Node von v und c die Nummer der gewählten Spalte von 1 bis 7 ist. Die Root-Node hat die Nummer 0.

$$v = 7 \cdot p + c \tag{3.1}$$

Da die Umrechnung von v zu p und umgekehrt durch diese Methode sehr simpel ist, wird sie verwendet, auch wenn gewisse Zahlen eine nicht erreichbare Node, also einen invaliden Spielstand, repräsentieren.

Das Spiel kann aufgrund der Feldgrösse von 6×7 maximal 42 Züge haben, v kann also einen Wert von 0 bis 7^{42} annehmen. Die Zahl 7^{42} kann nicht mit 64 Bits ausgedrückt werden, weswegen ein Int128 nötig ist.

3.2.2 Algorithmus

Der MCTS-Algorithmus kann anhand des Codes aus Quellcode 3.1 implementiert werden. Die vier Phasen, die in Unterabschnitt 3.1.2 beschrieben sind, können abgesehen von der Simulation relativ direkt implementiert werden.

Quellcode 3.1: MCTS Code für den Algorithmus

```
1  void MCTS::run(connect_four_board board) {
2      connect_four_board old_board = board;
3      for (int i = 0; i < iterations; i++) {
4          board = old_board;
5
6          board = select(board);
7          board = expand(board);
8
9          int result = 0;
10         for (int j = 0; j < num_simulations; j++) result += simulate(board);
11
12         backup(board.game_state, result);
13     }
14 }
```

Um die Simulation durchzuführen, werden verschiedene Methoden ausprobiert. Hierbei wird auf die in Unterabschnitt 2.1.3 beschriebenen Spieler zurückgegriffen, welche zufällige Simulationen und Simulationen mit minimaler Logik ermöglichen.

Mit der **get_col**-Funktion kann nun die in Quellcode 3.1 beschriebene Funktion aufgerufen werden. Anschliessend kann der Zug, der zur Child-Node mit dem besten Wert $\frac{w_v}{n_v}$ führt, zurückgegeben werden.

Für das Training spielt ein MCTS-Spieler mehrere Spiele gegen sich selbst und speichert dabei die Werte w_v und n_v jeder besuchten Node.

Kapitel 4

Neuronale Netzwerke

Viele der besten künstlichen Intelligenzen basieren auf neuronalen Netzwerken, welche das erste Mal 1944 von Warren McCulloch und Walter Pitts beschrieben wurden. Die Idee hinter den neuronalen Netzwerken ist die Nachahmung des menschlichen Gehirns. Es werden sogenannte Neuronen miteinander verbunden und die Informationen über die Verbindungen zwischen den Neuronen weitergegeben. [22]

In dieser Arbeit wird ein Feed-Forward-Netzwerk behandelt. Dabei werden die Neuronen in sogenannten Layern organisiert und die Informationen werden von Layer zu Layer weitergegeben.

4.1 Grundlagen

Dieses Unterkapitel erklärt überblicksmässig die wichtigsten Funktionen eines neuronalen Netzwerkes. Die Informationen stammen aus [2], worauf auch für genauere Erklärungen und Gleichungen verwiesen wird.

4.1.1 Begriffserklärungen und Notation

Um die Erklärungen des neuronalen Netzwerkes zu vereinfachen, werden in diesem Unterkapitel einige Begriffe definiert und Notationen eingeführt.

z-Wert

Der z-Wert ist der Begriff für den Input eines Neurons. Der z-Wert eines Neurons i in Layer l wird als z_i^l notiert. z^l ist die Notation für den Vektor aller z-Werte in Layer l.

Activation

Eine Activation ist der Begriff für den Output eines Neurons. Analog zur Notation des z-Wertes wird die Activation als a_i^l oder a^l notiert.

Activation-Function

Die Activation-Function ist die Funktion, welche aus dem z-Wert eines Neurons die Activation des Neurons berechnet. Die Activation-Function wird als $\sigma(z)$ notiert.

Weights

Eine Weight ist eine Gewichtung der Verbindung zwischen zwei Neuronen. Die Weight zwischen Neuron i in Layer l und Neuron j in Layer $l-1$ wird als w_{ij}^l notiert. Alle Weights zwischen Layer l und $l-1$ werden als w^l notiert.

Biases

Jedes Neuron besitzt eine Bias. Diese trägt unabhängig von a^{l-1} immer zum z-Wert z_i^l bei. Die Bias eines Neurons i in Layer l wird als b_i^l notiert und alle Biases in Layer l werden als b^l notiert.

Cost-Function

Die Cost-Function ist eine Funktion, welche angibt, wie sehr die Vorhersage a^N eines Netzwerkes von dem gewollten Output g abweicht. Die Cost-Function wird als $C\left(a^N, g\right)$ notiert.

Loss

Der Loss eines Netzwerks ist der Output der Cost-Function und wird als L notiert.

Gradient

Der Gradient eines Neurons ist als $\frac{\partial L}{\partial z_i^l}$ definiert. Er gibt also an, inwiefern eine kleine, positive Änderung von z_i^l den Loss verändert. Die Notation ist analog zu der Notation des z-Wertes δ_i^l oder δ^l.

Fully-Connected-Layer

Ein Fully-Connected-Layer ist ein Layer, bei dem jedes darin enthaltene Neuron mit jedem Neuron des vorherigen Layers verbunden ist.

Convolutional-Layer

Ein Convolutional-Layer berücksichtigt anders als ein Fully-Connected-Layer auch die örtliche Struktur des Inputs. Dafür verwendet es drei Konzepte. Es ist in Unterabschnitt 4.1.5 genauer beschrieben.

Feed-Forward

Bei dem Feed-Forward berechnet das Netzwerk aus einem gegebenen Input den entsprechenden Output.

Backpropagation

Die Backpropagation ist der Prozess, bei dem alle $\frac{\partial L}{\partial w_{ij}^l}$ und $\frac{\partial L}{\partial b_i^l}$ berechnet werden. Das bedeutet, dass für jede Weight und Bias berechnet wird, inwiefern eine kleine, positive Änderung der Weight oder Bias den Loss verändert.

Update

Das Update passt die Weights und Biases eines Netzwerkes so an, dass der Loss minimiert wird.

4.1.2 Feed-Forward im Fully-Connected-Layer

Im Feed-Forward wird ein Input durch das Netzwerk weitergegeben und daraus ein Output berechnet.

Zuerst werden die Activations a^1 des Input-Layers auf die Werte des Inputs gesetzt. Anschliessend können aus a^1 die z-Werte für das nächste Layer, z^2, über eine Matrix-Vektor-Multiplikation mit w^2 berechnet werden. Daraus kann a^2 durch $a_i^l = \sigma\left(z_i^l\right)$ ermittelt werden. Dieser Prozess wird iterativ von vorne nach hinten durchgeführt, bis die Activation des Output-Layers, also der Output des Netzwerkes, berechnet ist.

4.1.3 Backpropagation im Fully-Connected-Layer

In der Backpropagation wird aus dem Output a^N des Netzwerkes und aus dem gewolltem Output g berechnet, inwiefern kleine, positive Änderungen der Weights und Biases den Loss verändern.

Aus dem Loss L, der durch die Cost-Function C gegeben ist, kann der Gradient des letzten Layers ermittelt werden. Anschliessend können alle Gradients iterativ von hinten nach vorne berechnet werden, da δ^{l-1} durch δ^l ausgerechnet werden kann. $\frac{\partial L}{\partial w_{ij}^l}$ und $\frac{\partial L}{\partial b_i^l}$ können beide über δ_i^l bestimmt werden, wobei $\frac{\partial L}{\partial w_{ij}^l}$ auch von a^{l-1} abhängt.

4.1.4 Updaten der Weights und Biases im Fully-Connected-Layer

Während des Updates werden die Weights und Biases so angepasst, dass der Loss minimiert wird. Im Folgenden wird erklärt, wie diese Anpassung aus $\frac{\partial L}{\partial w_{ij}^l}$ und $\frac{\partial L}{\partial b_i^l}$ berechnet werden kann.

Angenommen $\frac{\partial L}{\partial w_{ij}^l}$ ist negativ. In diesem Fall sollte die Weight erhöht werden, da so der Loss kleiner wird. Im gegenteiligen Fall sollte die Weight verringert werden, wodurch auch der Loss kleiner wird. Die Weight w_{ij}^l muss also um $-\eta\frac{\partial L}{\partial w_{ij}^l}$ angepasst werden. Dabei

ist η eine kleine, positive Zahl und wird Learning-Rate genannt. Analog werden die Biases angepasst.

4.1.5 Convolutional-Layer

Während das Fully-Connected-Layer die relativen Positionen der Input-Neuronen nicht berücksichtigt, ist dies beim Convolutional-Layer der Fall. Diese Eigenschaft ist besonders bei Bildern wichtig, da die Positionen der Pixel eine wichtige Rolle spielen. Dabei gibt es drei Konzepte, die das Convolutional-Layer von einem Fully-Connected-Layer unterscheiden.

1. Local-Receptive-Fields

   Ein Neuron im Convolutional-Layer ist nur mit einem kleinen Feld des vorherigen Layers verbunden. Dieses Feld wird als Local-Receptive-Field bezeichnet. Dadurch, dass ein Neuron im Convolutional-Layer nur auf sein Local-Receptive-Field reagiert, wird die örtliche Struktur des Inputs berücksichtigt.

2. Geteilte Weights und Biases

   Für ein Convolutional-Layer gibt es mehrere Feature-Maps. Die Feature-Maps können als verschiedene Filter für den Input verstanden werden, denn alle Neuronen einer Feature-Map teilen sich die Weights und Biases, die sie mit ihrem Local-Receptive-Field verbinden.

3. Pooling

   Um die Dimensionen des Netzwerkes zu reduzieren, gibt es nach einem Convolutional-Layer häufig ein Pooling-Layer. Dabei werden mehrere benachbarte Neuronen zu einem Neuron zusammengefasst.

Das Feed-Forward, die Backpropagation und das Updaten der Weights und Biases funktionieren ähnlich wie im Fully-Connected-Layer. Deshalb wird hier nicht näher darauf eingegangen.

4.2 Programmierung

Das neuronale Netzwerk ist zu Beginn dieser Arbeit bereits funktionsfähig vorhanden. Es führt seine Berechnungen auf dem Hauptprozessor (CPU) durch. Im Zuge dieser Arbeit wird das Programm mithilfe des Programmiermodells CUDA so umgeschrieben, dass die Berechnungen auf dem Grafikprozessor (GPU) durchgeführt werden. Dafür wird CUDA C++ [23] verwendet. Das Programmiermodell stammt von NVIDIA, darum ist eine NVIDIA-GPU notwendig. Dafür wird vom TechLab an der KSBG ein Computer zur Verfügung gestellt. Auf diesen kann über eine SSH-Verbindung auch von extern zugegriffen

werden. Um CUDA zu erlernen, werden relevante Ausschnitte aus dem Buch *CUDA by Example* [24] von Jason Sanders und Edward Kandrot gelesen. Diese Arbeit setzt an späteren Stellen voraus, dass der Leser mit dem Programmiermodell CUDA vertraut ist.

4.2.1 Feed-Forward im Fully-Connected-Layer

Als erster Schritt wird das Feed-Forward im Fully-Connected-Layer umgeschrieben.

Die Gleichung für das Berechnen von a^l kann in C++ anhand Quellcode 4.1 umgesetzt werden.

Quellcode 4.1: C++ Code für das Feed-Forward in einem Fully-Connected-Layer

```
1 void fully_connected_layer::feedforward() {
2     for (int neuron = 0; neuron < data.n_out.x; neuron++) {
3         for (int prev_neuron = 0; prev_neuron < data.n_in.x; prev_neuron++) {
4             z[l][neuron] += a[l-1][prev_neuron] * w[l][neuron][prev_neuron];
5         }
6         z[l][neuron] += b[l][neuron];
7         a[l][neuron] = activation_function(z[l][neuron]);
8     }
9 }
```

Der Quellcode kann mit CUDA umgeschrieben werden, indem der äussere For-Loop die Dimensionen für das Grid und der innere For-Loop die Dimensionen für einen Block liefern. In Quellcode 4.2 ist dies dargestellt.

Quellcode 4.2: CUDA Code für das Feed-Forward in einem Fully-Connected-Layer

```
1 void fully_connected_layer::feedforward() {
2     dev_feedforward<<<data.n_out.x, data.n_in.x>>>();
3     cudaDeviceSynchronize();
4 }
```

Innerhalb eines Threads muss der Wert $w_{ij}^l a_j^{l-1}$ berechnet werden. Die berechneten Werte der Threads eines Blocks müssen anschliessend aufsummiert werden. Das stellt ein Problem für die Parallelisierung dar, da es nicht möglich ist, dass mehrere Threads gleichzeitig auf die gleiche Speicheradresse schreiben.

Dies wird zuerst mit der **AtomicAdd**-Funktion [23] gelöst, welche erlaubt, dass mehrere Threads auf die gleiche Speicheradresse schreiben. Jedoch stellt sich diese Variante als langsam heraus. Vermutlich könnten auch mit **AtomicAdd** gute Resultate erzielt werden, wenn die in dem Blog [25] beschriebenen Ideen umgesetzt werden, doch eine Parallel-Reduction [26] scheint vielversprechender. Eine Parallel-Reduction ermöglicht es, dass ein Vektor in $\mathcal{O}(\log n)$ aufsummiert wird und umgeht das Problem, dass mehrere Threads versuchen, gleichzeitig auf die gleiche Speicheradresse zu schreiben. Dieser Ansatz ist mit einigen Optimierungen sehr viel schneller.

4.2.2 Backpropagation im Fully-Connected-Layer

Das Berechnen des Gradients im letzten Layer kann ohne eine Parallel-Reduction gelöst werden, da die Threads nicht auf die gleiche Speicheradresse schreiben. Bei dem Zurückpropagieren des Gradients ist dies jedoch der Fall, darum wird hier eine Parallel-Reduction verwendet. Für die Berechnung von $\frac{\partial L}{\partial w_{ij}^l}$ und $\frac{\partial L}{\partial b_i^l}$ können erneut Threads laufen gelassen werden, die unabhängig voneinander die entsprechenden Werte berechnen.

4.2.3 Update im Fully-Connected-Layer

Das Update der Weights und Biases ist eine Operation, bei der keine Threads auf die gleiche Speicheradresse schreiben und kann deswegen unkompliziert parallelisiert werden.

4.2.4 Convolutional-Layer

Das Umschreiben des Convolutional-Layers ist etwas aufwändiger. Es kann nicht direkt aus dem C++ Quellcode oder der Formel abgelesen werden, welche Vektoren aufsummiert werden müssen. Sobald die Vektoren, die aufsummiert werden müssen, identifiziert sind, können die Ideen aus dem Fully-Connected-Layer verwendet werden, um den Code umzuschreiben.

4.2.5 Max-Pooling-Layer

Ein Max-Pooling-Layer dient dazu, die Dimensionen des Netzwerkes zu reduzieren. Dies ist zum Beispiel beim MNIST-Datensatz [27] nötig, da die Input-Grösse 28×28 beträgt und die Dimensionen sehr schnell sehr gross werden. Bei dem MNIST-Datensatz handelt es sich um einen Datensatz, welcher verwendet werden kann, um ein Netzwerk zu trainieren, welches handgeschriebene Ziffern erkennt. Im benötigten Netzwerk hat der Input eine Grösse von 6×7, also die Spielfeldgrösse von *Vier Gewinnt*. Die Dimensionen sind wesentlich kleiner, weshalb kein Max-Pooling-Layer benötigt wird.

Kapitel 5

Der DQN-Algorithmus

Der DQN-Algorithmus verwendet ein neuronales Netzwerk, um den nächsten Spielzug zu wählen. Es ist eine Variante des *Q-Learnings*. Im folgenden Kapitel sind die wichtigsten Punkte zum DQN-Algorithmus beschrieben. Die Informationen dazu stammen aus einem Online-Kurs [28].

5.1 Grundlagen

Die Idee hinter dem Algorithmus ist es, dass ein neuronales Netzwerk bewertet, wie gut ein Zug in einem bestimmten Spielstand ist. Aus den Bewertungen der möglichen Züge wird der nächste Zug gewählt. Der Spielstand ist der Input für das Netzwerk und der Output ist die Bewertung der Züge.

5.1.1 Begriffserklärungen

Um die Erklärungen für den DQN-Algorithmus zu vereinfachen, werden im folgenden Kapitel einige Begriffe erklärt.

Agent

Als Agent wird im Reinforcement-Learning das bezeichnet, was entscheidet, welcher Zug ausgeführt werden soll.

Reward

Der Agent erhält für bestimmte Züge durch den Reward r ein positives oder negatives Feedback.

Experience

Die Erfahrungen, welche der Agent macht, werden als Experiences bezeichnet. Eine Experience e besteht aus vier Werten:

- Spielstand s
- gewählter Zug a
- Reward r, der durch Zug a in Spielstand s erhalten wird
- nächster Spielstand s', der durch Zug a in Spielstand s erreicht wird

Q-Value

Der Bewertung eines Zuges in einem gegebenen Spielstand wird Q-Value genannt. Das Q-Value entspricht der erwarteten Summe der Rewards, die bis Ende des Spiels erreicht werden, wenn im Spielstand s der Zug a ausgeführt wird.

Q-Function

Die Funktion, die aus Spielstand s das Q-Value für Zug a berechnet, wird als Q-Function bezeichnet. Die optimale Q-Function $Q_*(s, a)$ steht für diejenige Q-Function, welche für jedes Paar bestehend aus Spielstand s und Zug a das korrekte Q-Value liefert.

Target

Das Target ist ein Begriff für den erwarteten Output des Netzwerkes für das jeweilige Paar von Spielstand und Zug, welches in der Experience e gewählt wurde.

5.1.2 Bellman-Equation

Die Bellman-Equation beschreibt eine fundamentale Eigenschaft der optimalen Q-Function. In Gleichung 5.1 ist die Bellman-Equation für *Vier Gewinnt* dargestellt.

$$Q_*(s, a) = E\left[r - \gamma \max_{a'} Q_*(s', a')\right] \tag{5.1}$$

Ein Zug ist gut, wenn er einen hohen Reward erhält oder der nächste Spielstand schlecht für den Gegner ist. Dabei wird der Parameter γ (Discount-Factor) verwendet, um Rewards in näherer Zukunft eine höhere Gewichtung zu geben. Üblicherweise besitzt er einen Wert, der etwas tiefer als 1 ist.

5.1.3 Berechnung des Loss

Um den Agenten zu trainieren, muss der Loss für das neuronale Netzwerk berechnet werden. Im DQN-Algorithmus geschieht dies über die Bellman-Equation (Gleichung 5.1). Der Loss wird jeweils für eine Experience berechnet und bezieht sich nur auf die Vorhersage, die das neuronale Netzwerk für den gewählten Zug a im Spielstand s macht. Für alle anderen Vorhersagen des Spielstands s ist der Loss 0.

Das Target T wird anhand Gleichung 5.2 definiert. Es entspricht der rechten Seite der Bellman-Equation (Gleichung 5.1).

$$T := r - \gamma \max_{a'} Q\left(s', a'\right) \tag{5.2}$$

Der Loss L wird über $C\big(Q\left(s, a\right), T\big)$ berechnet.

Durch diesen Loss konvergiert das neuronale Netzwerk nach und nach zur optimalen Q-Function, bei der die Bellman-Equation für alle Paare von Spielstand und Zug gilt.

5.1.4 Epsilon-Greedy-Strategie

Für den Agenten ist es während des Trainings wichtig, ein Gleichgewicht zwischen dem Ausprobieren neuer Züge und dem Verwenden von Zügen, die er als gut einschätzt, zu finden. Eine Methode, um ein gutes Gleichgewicht zu finden, ist die Epsilon-Greedy-Strategie. Dabei wird mit einer Wahrscheinlichkeit von ϵ ein zufälliger Zug und mit einer Wahrscheinlichkeit von $1-\epsilon$ der Zug mit dem höchsten Q-Value gewählt. Das bedeutet, dass der Agent mit der Wahrscheinlichkeit ϵ neue Züge ausprobiert und mit der Wahrscheinlichkeit $1 - \epsilon$ Züge wählt, welche er als gut bewertet.

Zu Beginn des Trainings wird der Epsilon-Wert ϵ auf einen hohen Wert gesetzt. Damit wird sichergestellt, dass der Agent am Anfang viele Möglichkeiten ausprobiert und so Erfahrungen sammelt. Mit der Zeit wird ϵ immer kleiner, was dazu führt, dass der Agent immer mehr die Züge wählt, die er als gut bewertet.

5.1.5 Experience-Replay

In einem Spiel sind die Experiences stark korreliert, da sie nacheinander auftreten. Dem neuronalen Netzwerk würde es aufgrund dieser hohen Korrelation schwerfallen, das Gelernte zu verallgemeinern. Um das zu verhindern, wird eine Methode namens Experience-Replay verwendet. Jede Experience wird im sogenannten Replay-Memory gespeichert.

Durch die vier Werte der Experience kann der Loss des neuronalen Netzwerkes für diese Experience berechnet werden. Um das Netzwerk zu trainieren, werden zufällige Experiences aus dem Replay-Memory ausgewählt. Die ausgewählten Experiences werden als Batch bezeichnet. So wird verhindert, dass das neuronale Netzwerk auf viele korrelierte Experiences direkt nacheinander trainiert wird.

5.1.6 Main-Netzwerk und Target-Netzwerk

Durch das ständige Lernen des neuronalen Netzwerkes ändert sich auch immer das Target T, welches vom Output des Netzwerkes abhängt (siehe Gleichung 5.2). Dadurch wird das Lernen instabil. Um das Lernen etwas zu stabilisieren, wird ein zweites Netzwerk eingeführt, welches als Target-Netzwerk bezeichnet wird. Es wird nur für das Berechnen des Targets T verwendet. Während das Main-Netzwerk ständig trainiert wird und seine Weights und Biases ändert, bleibt das Target-Netzwerk unverändert. Erst nach einer

Anzahl an Zügen c werden die Weights und Biases aus dem Main-Netzwerk ins Target-Netzwerk kopiert. Dabei ist c ein Parameter, der zu Beginn des Trainings festgelegt werden muss.

5.2 Programmierung

Für den DQN-Algorithmus wird das in Kapitel 4 beschriebene neuronale Netzwerk verwendet. Bei der Implementation wird versucht, den DQN-Algorithmus möglichst genau umzusetzen.

Es muss die Klasse aus Quellcode 2.1 implementiert werden, um den DQN-Algorithmus in das Spiel einzubauen. Da ein Training für das neuronale Netzwerk benötigt wird, muss nebst den drei Funktionen **load**, **save** und **get_col** auch die **train**-Funktion implementiert werden.

5.2.1 Überlegungen zum Trainieren des neuronalen Netzwerkes

Bevor das Training implementiert wird, muss entschieden werden, wie viel der Agent erlernen muss und welche Hilfestellungen ihm gegeben werden. Dabei wird die Entscheidung getroffen, dass der Agent auch die Regeln des Spiels erlernen muss. Beim Versuch, einen illegalen Zug zu spielen, erhält der Agent einen Reward von -1. Falls das Spiel durch einen Zug des Agenten gewonnen wird, erhält er einen Reward von 1. Ansonsten ist der Reward 0. Nach dem Training werden vom Agenten nur noch erlaubte Züge in Betracht gezogen.

Als Input wird dem neuronalen Netzwerk ein Vektor der Länge $2 \times 6 \times 7$ gegeben. Die ersten 7×6 Werte geben mit 1 oder 0 an, ob auf dem entsprechenden Feld ein Stein des Agenten ist. Die nächsten 6×7 Werte geben an, ob das Feld von dem Gegner besetzt ist.

5.2.2 Berechnung des Loss

Das neuronale Netzwerk aus Kapitel 4 ist so implementiert, dass der Funktion **SGD**, in der das Training des neuronalen Netzwerkes stattfindet, ein Vektor mit dem Input und ein Vektor mit dem gewollten Output übergeben wird. Als gewollter Output muss ein Vektor berechnet werden, welcher zu dem Loss führt, der in Unterabschnitt 5.1.3 beschrieben ist. Quellcode 5.1 zeigt, wie dieser Vektor berechnet wird.

Quellcode 5.1: DQN Code für die Berechnung des Outputs

```
1 float* DQL::get_output(Experience exp) {
2     float* main_out = feedforward(exp.state, main);
3     float* tar_out = feedforward(exp.new_state, target);
4 
5     float max_out = *std::max_element(tar_out, tar_out+OUTPUT_NEURONS);
6 
7     if (exp.reward != 0) main_out[exp.action] = exp.reward;
8     else main_out[exp.action] = -discount_factor*max_out;
```

```
9
10      return main_out;
11  }
```

Um einen Loss von 0 für alle Züge, die nicht dem gewählten Zug entsprechen, zu erhalten, muss der tatsächliche Output berechnet werden. Der gewollte Output wird anschliessend auf diesen Wert gesetzt. Dies funktioniert allerdings nur dann, wenn sich das Netzwerk während des Berechnens des gewollten Outputs und des Berechnens des Loss nicht verändert. Deshalb kann immer nur ein Trainingsbeispiel auf einmal trainiert werden.

Der gewollte Output des Paares bei dem gewählten Zug a entspricht dem Target T aus Gleichung 5.1 und wird in Quellcode 5.1 in Linie 7 und 8 gesetzt. Dabei wird bei $r \neq 0$ der Output des Target-Netzwerkes nicht berücksichtigt. Dies liegt daran, dass der Reward nur dann nicht null ist, wenn das Spiel durch den Zug a gewonnen wird oder wenn der Zug a kein legaler Zug ist. In beiden Fällen ist es nicht sinnvoll, den Reward mit dem Output des Target-Netzwerkes zu verrechnen.

5.2.3 Training

In der Trainingsphase sammelt der Agent Experiences, mit welchen er das neuronale Netzwerk trainiert. Um diese Experiences zu sammeln, spielt der Agent Spiele gegen sich selbst. Dabei wählt er die Züge mit der Epsilon-Greedy-Strategie aus. Die gemachte Experience wird anschliessend im Replay-Memory gespeichert. Das neuronale Netzwerk wird nach jedem Zug durch einen zufällig gewählten Batch von Experiences aus dem Replay-Memory trainiert, indem der Loss wie in Unterabschnitt 5.2.2 beschrieben berechnet wird.

Nach jeweils c Zügen werden die Weights und Biases vom Main-Netzwerk zum Target-Netzwerk kopiert.

Quellcode 5.2: DQN Code für das Training

```
1  void DQL::train(int num_games) {
2      int turn = 0;
3      for (int game = 0; game < num_games; game++) {
4          connect_four_board board;
5
6          while (true) {
7              int action = get_col(board);
8              Experience exp = get_experience(board, action);
9              store_in_replay_memory(exp);
10
11             std::vector<Experience> batch = get_random_batch();
12
13             for (auto experience: batch) {
14                 float *in = get_input(experience.state);
15                 float *out = get_output(experience);
16                 main.SGD(in, out);
```

```
17              }
18
19              board = exp.new_state;
20              turn++;
21
22              if (turn % c == 0) copy_main_to_target();
23              if (board.win() || board.turns == 42) break;
24          }
25          epsilon = reduce_epsilon(epsilon);
26      }
27  }
```

Kapitel 6

Resultate

6.1 Vergleich der Laufzeit des neuronalen Netzwerkes in C++ und in CUDA

Nachdem das neuronale Netzwerk, wie in Kapitel 4 beschrieben, zu CUDA umgeschrieben wurde, werden die Laufzeiten der beiden Versionen verglichen.

6.1.1 Feed-Forward in einem Fully-Connected-Netzwerk

Um das Feed-Forward in einem Fully-Connected-Netzwerk zu vergleichen, wird ein Netzwerk mit n Neuronen im Input-Layer und n Neuronen im Output-Layer erstellt. Um den Zufallsfaktor zu verringern, werden zehnmal 10^4 Inputs durch das Netzwerk geschickt, bevor der Durchschnitt des Laufzeiten berechnet und gespeichert wird. Die Daten sind in Abbildung 6.1 ersichtlich. Es werden die Laufzeiten für n von 5 bis und mit 500 gemessen. Dabei wird n immer um 5 erhöht.

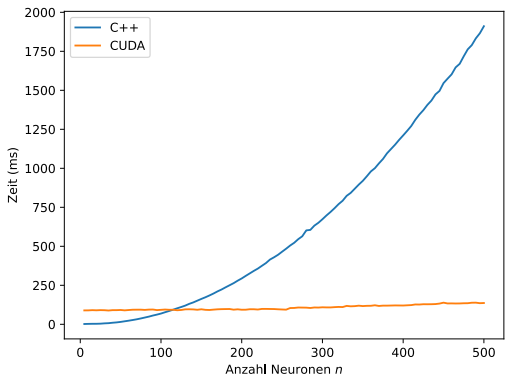
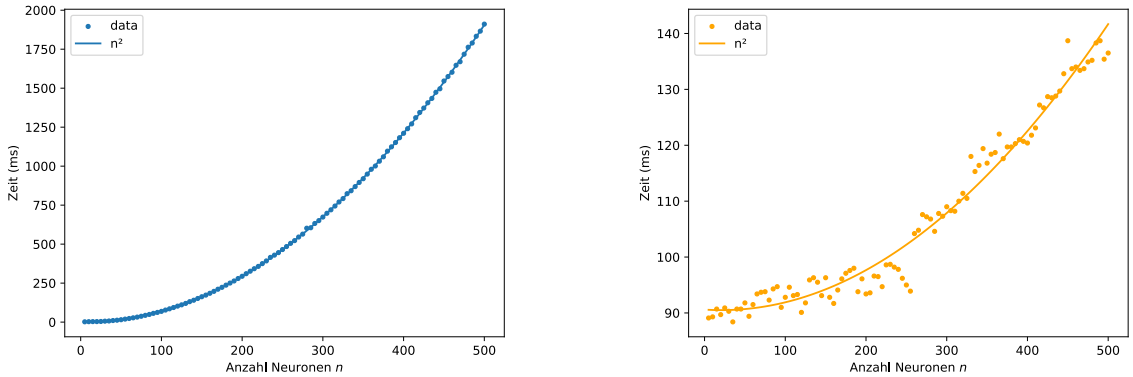


Abbildung 6.1: Vergleich Fully-Connected Feed-Forward

Die Abbildung 6.1 zeigt, dass die Laufzeit des Feed-Forward in C++ deutlich schneller wächst als die des Feed-Forward in CUDA. Beide Graphen haben eine quadratische Laufzeit. Dies ist durch eine Regression (Abbildung 6.2) ersichtlich.



C++ Regression CUDA Regression

Abbildung 6.2: Regressionen Fully-Connected Feed-Forward

In Theorie hat das Feed-Forward durch den Parallel-Reduction-Algorithmus eine logarithmische Laufzeit. Praktisch können aber die Threads und Blocks nicht in konstanter Zeit gestartet werden. Es werden jeweils n Blocks mit je n Threads, also insgesamt n^2 Threads, gestartet. Das ist vermutlich der Grund, weshalb die Laufzeit des Feed-Forwards in CUDA quadratisch wächst. Für eine Bestätigung dieser Vermutung wäre eine vertiefte Untersuchung notwendig.

6.1.2 Training in einem Fully-Connected-Netzwerk

Das Training, bestehend aus dem Feed-Forward, der Backpropagation und dem Update, wird analog zum Feed-Forward verglichen. Das Netzwerk wird zehnmal mit 10^4 Inputs trainiert und die durchschnittliche Laufzeit des Trainierens wird berechnet. Die Daten von C++ und CUDA im Vergleich sind in Abbildung 6.3 ersichtlich.

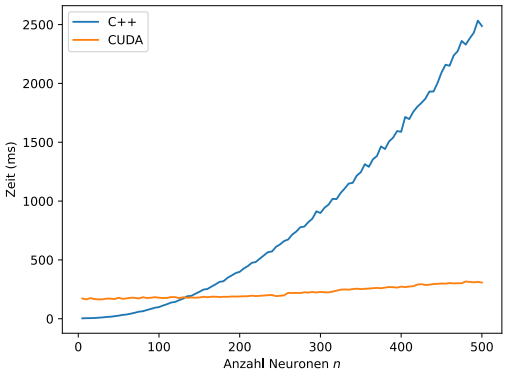
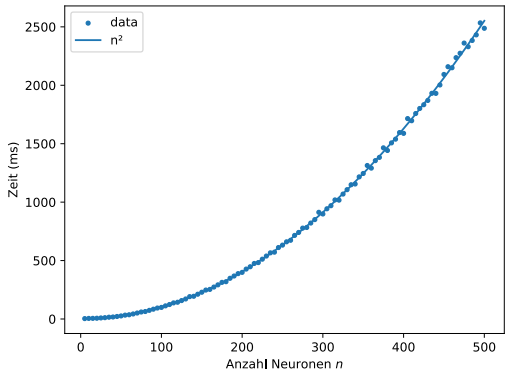


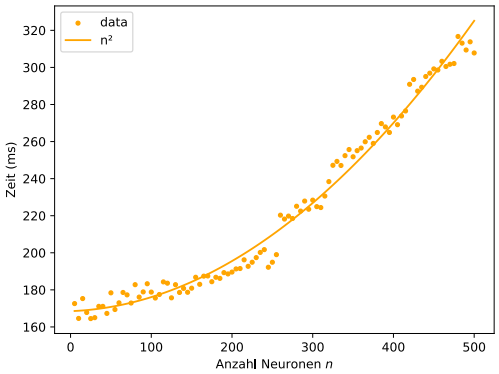
Abbildung 6.3: Vergleich Fully-Connected Training

Die Laufzeit des Trainings wächst in C++ deutlich schneller als die des Trainings in CUDA und beide Funktionen wachsen quadratisch (siehe Abbildung 6.4). In C++ ist die

Laufzeit bei $n = 500$ etwa 1.28-mal so gross, wie die des Feed-Forwards. In CUDA ist die Laufzeit etwa 2.25-mal grösser. Für eine Erklärung wäre eine vertiefte Auseinandersetzung mit den Faktoren, die die Laufzeit eines Programmes beeinflussen, notwendig. Darauf wird aus Zeitgründen verzichtet.



C++ Regression



CUDA Regression

Abbildung 6.4: Regressionen Fully-Connected Training

6.1.3 Feed-Forward Convolutional-Netzwerk

Das Feed-Forward in einem Convolutional-Netzwerk wird ebenfalls verglichen. Dafür wird ein Netzwerk mit einem Input-Layer von $n \times 5$ Neuronen, einem Convolutional-Layer mit 3 Feature-Maps und einem Local-Receptive-Field von 5×5 und einem Output-Layer mit n Neuronen erstellt. Es wird erneut zehnmal mit 10^4 Inputs die Laufzeit gemessen und der Durchschnitt berechnet. Der Wert n wird von 5 bis und mit 345 in Schritten von 5 erhöht. Die Daten sind in Abbildung 6.5 ersichtlich.

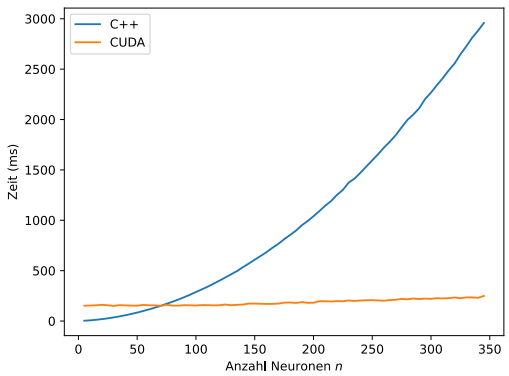
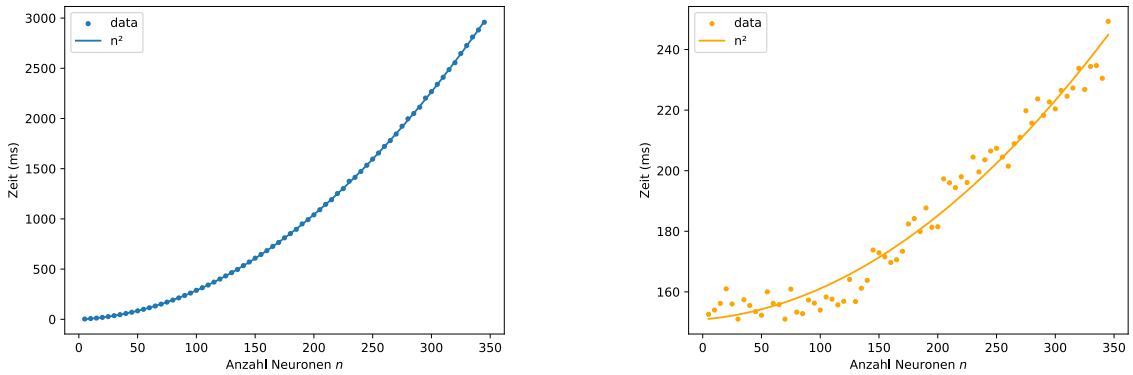


Abbildung 6.5: Vergleich Convolutional Feed-Forward

Die obere Grenze für n wird als 345 gewählt, da bei $n = 350$ mehr als die maximal erlaubten 1024 Threads pro Block gestartet werden [23]. Eine Anpassung des Parallel-

Reduction-Algorithmus könnte dieses Problem beheben, ist jedoch für diese Arbeit nicht notwendig.

Die Laufzeit des Feed-Forward in C++ wächst auch bei einem Convolutional-Netzwerk schneller als die des Feed-Forward in CUDA. Bei $n = 345$ ist die Laufzeit des Feed-Forwards in einem Convolutional-Netzwerk in C++ etwa 3.30-mal so gross, wie die Laufzeit des Feed-Forwards in einem Fully-Connected-Netzwerk. In CUDA ist die Laufzeit etwa 2.09-mal grösser.



C++ Regression CUDA Regression

Abbildung 6.6: Regressionen Convolutional Feed-Forward

6.1.4 Training Convolutional-Netzwerk

Um das Training in einem Convolutional-Netzwerk zu vergleichen, wird dasselbe Netzwerk wie bei Unterabschnitt 6.1.3 verwendet. Analog zu Unterabschnitt 6.1.2 wird das Netzwerk zehnmal mit 10^4 Inputs trainiert, woraus dann der Durchschnitt der Laufzeiten berechnet wird. In Abbildung 6.7 ist der Vergleich der Laufzeiten ersichtlich.

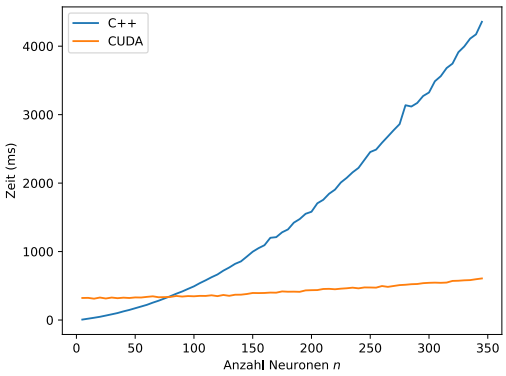
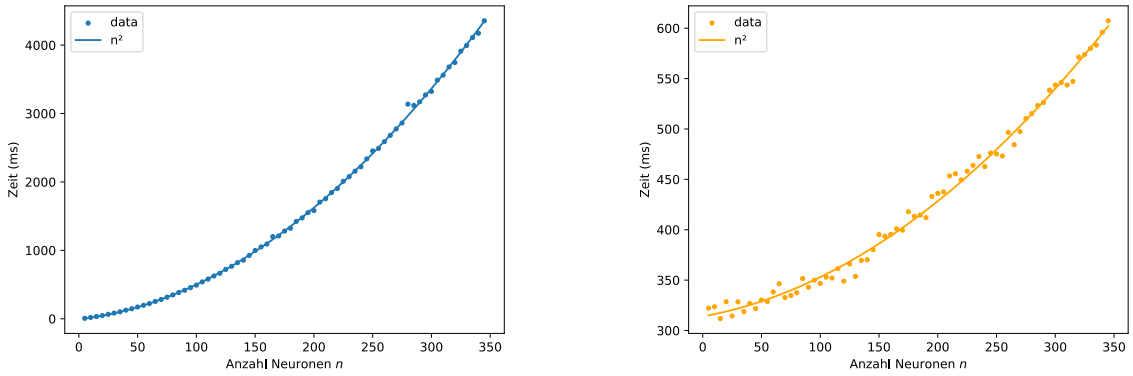


Abbildung 6.7: Vergleich Convolutional Training

Die Laufzeit in C++ wächst noch immer deutlich schneller. Die quadratischen Re-

gressionen sind in Abbildung 6.8 ersichtlich. Auch hier dauert das Training in CUDA im Verhältnis zum Feed-Forward länger als in C++.



C++ Regression CUDA Regression

Abbildung 6.8: Regressionen Convolutional Training

6.2 Vergleich verschiedener MCTS

Es gibt Parameter, die beim MCTS-Algorithmus verändert werden können. In diesem Unterkapitel wird der Einfluss einiger dieser Parameter auf den ELO-Wert des MCTS-Spielers genauer untersucht.

6.2.1 Anzahl Simulationen

Um auszuwerten, wie die Anzahl der Simulationen den ELO-Wert eines MCTS-Spielers beeinflusst, werden mehrere MCTS-Spieler mit unterschiedlich vielen Simulationen trainiert. Für die Anzahl der Simulationen werden die Zweierpotenzen von 2^0 bis und mit 2^{10} verwendet. Alle anderen Parameter bleiben konstant, wobei die Anzahl der Iterationen auf 8 und die Anzahl der Trainingsspiele auf 1 fixiert wird. Um den Zufallsfaktor etwas zu verringern, gibt es jeweils drei MCTS-Spieler, welche gleich viele Simulationen haben. Aus den drei ELO-Werten wird der Durchschnitt berechnet. Die Abbildung 6.9 zeigt die dadurch erhaltenen Daten.

Sehr deutlich zeigt sich, dass die MCTS-Spieler mit zufälliger Simulation schlechtere Resultate erzielen als die MCTS-Spieler mit einer Simulation, die auf minimaler Logik basiert. Dies ist damit zu erklären, dass die zufällige Simulation weniger zuverlässig beurteilt, ob eine Node zu einem Sieg führt oder nicht.

Allerdings haben beide Funktionen eine ähnliche Form. Die Steigung ist zu Beginn sehr gross und nimmt dann immer mehr ab, doch bleibt tendenziell immer nicht-negativ. Mit mehr Simulationen kann genauer beurteilt werden, wie oft aus der neu expandierten Node ein Sieg resultiert. Dementsprechend nähert sich der Wert $\frac{w_v}{n_v}$ immer mehr dem

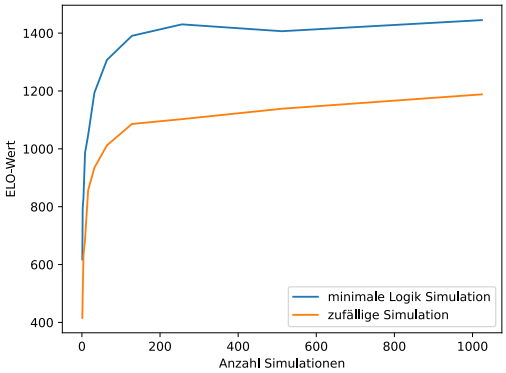


Abbildung 6.9: MCTS Simulationen

tatsächlichen Wert an. Je höher die Anzahl der Simulationen ist, desto weniger ändert eine weitere Simulation den Wert $\frac{w_v}{n_v}$. Die Verbesserung des ELO-Wertes wird also immer kleiner, weswegen auch die Steigung der Funktion immer kleiner wird.

6.2.2 Anzahl Iterationen

Analog zu Unterabschnitt 6.2.1 kann auch der Zusammenhang zwischen ELO-Wert und der Anzahl der Iterationen genauer untersucht werden. Die Anzahl der Simulationen wird hierbei auf 8 fixiert und für die Anzahl der Iterationen werden die Zweierpotenzen von 2^0 bis und mit 2^{10} verwendet. Die erhaltenen Daten sind in Abbildung 6.10 abgebildet.

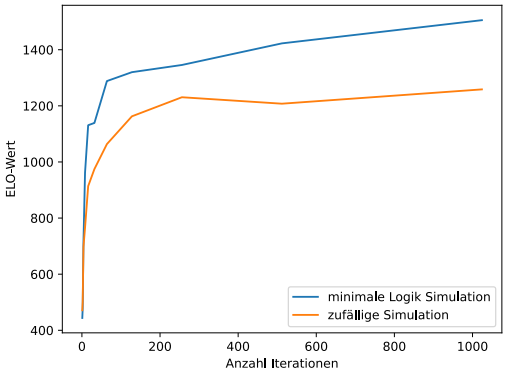


Abbildung 6.10: MCTS Iterationen

Die Formen der Funktionen sind ähnlich wie in Unterabschnitt 6.2.1. Die MCTS-Spieler mit zufälliger Simulation schneiden schlechter ab und die Steigung nimmt mit mehr Iterationen ab, bleibt jedoch tendenziell immer nicht-negativ. Die Erklärung dazu ist analog zu Unterabschnitt 6.2.1.

Die Spannbreite beider Funktionen ist jedoch grösser, als in Unterabschnitt 6.2.1. Daraus lässt sich schliessen, dass die Anzahl der Iterationen einen grösseren Einfluss auf den ELO-Wert hat als die Anzahl der Simulationen.

6.2.3 Verhältnis Iterationen und Simulationen

Für jede Iteration eines MCTS-Spielers werden mehrere Simulationen durchgeführt. Die Anzahl der Simulationen, bevor ein Zug gewählt wird, entspricht also dem Produkt aus Iterationen und Simulationen. Um zu untersuchen, inwiefern das Verhältnis von Iterationen und Simulationen den ELO-Wert beeinflusst, wird das Produkt der beiden Parameter auf den konstanten Wert 2^{10} fixiert. Die Anzahl der Simulationen wird von 2^0 bis und mit 2^{10} in Zweierpotenzen erhöht, während die Anzahl der Iterationen entsprechend verringert wird. Die Anzahl der Trainingsspiele wird auf 1 fixiert. Erneut gibt es, um den Zufallsfaktor zu verringern, drei MCTS-Spieler die gleich viele Simulationen und Iterationen haben. Der Zusammenhang ist in Abbildung 6.11 ersichtlich.

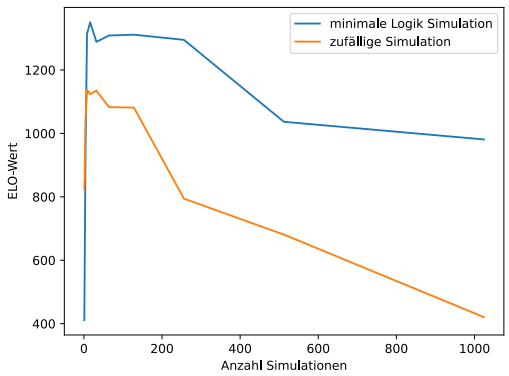


Abbildung 6.11: MCTS Iterationen und Simulationen

Die MCTS-Spieler mit der zufälligen Simulation schneiden auch hier schlechter ab, jedoch sind die Formen der Funktionen neu. Die Steigung ist zuerst positiv, erreicht ein Maximum und verläuft dann in eine negative Richtung. Dies ist damit zu erklären, dass mit mehr Simulationen der ELO-Wert immer weniger stark positiv beeinflusst wird. Gleichzeitig sinkt allerdings die Anzahl der Iterationen, wodurch der ELO-Wert immer stärker negativ beeinflusst wird. Zu Beginn überwiegt noch die positive Steigung, die aufgrund der Simulationen entsteht, doch ab dem Maximum überwiegt der negative Einfluss, der durch die Iterationen auftritt.

Das Maximum liegt zwischen 2^3 und 2^5 Simulationen. Dies deutet darauf hin, dass die Anzahl der Iterationen einen grösseren Einfluss auf den ELO-Wert hat als die Anzahl der Simulationen, was sich mit der Beobachtung aus Unterabschnitt 6.2.2 deckt. Mehr Simulationen führen zu einer genaueren Berechnung des Wertes $\frac{w_v}{n_v}$ einer Node v. Mehr Iterationen führen dazu, dass der Tree weiter expandiert wird. Aus den Ergebnissen lässt sich also schliessen, dass die Erweiterung des Trees einen grösseren Einfluss auf den ELO-Wert hat als die Genauigkeit der Werte $\frac{w_v}{n_v}$.

6.2.4 Anzahl Trainingsspiele

Die Anzahl der Trainingsspiele ist ebenfalls ein Parameter, der verändert werden kann. Sie wird analog zu Unterabschnitt 6.2.1 variiert, wobei die Anzahl der Simulationen auf 8 und die Anzahl der Iterationen auf 8 fixiert wird. In Abbildung 6.12 ist ersichtlich, inwiefern die Anzahl der Trainingsspiele den ELO-Wert eines MCTS-Spielers beeinflusst.

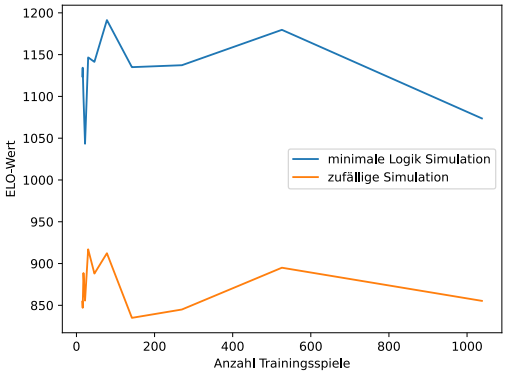


Abbildung 6.12: MCTS Spiele

Die Daten zeigen, dass der ELO-Wert mit mehr Trainingsspielen relativ konstant bleibt. Dies deutet darauf hin, dass das Programm einen Bug hat, denn durch den stärker expandierten Tree sollte der ELO-Wert eigentlich steigen. Der Bug konnte aufgrund der begrenzten Zeit nicht gefunden werden.

6.3 Vergleich verschiedener DQN

Auch ein DQN hat Parameter, die verändert werden können. Es wird nur der Einfluss der Anzahl der Trainingsspiele auf den ELO-Wert untersucht, denn aus Zeitgründen können die restlichen Parameter nicht optimal gwählt werden, wodurch viele Resultate nicht aussagekräftig werden.

6.3.1 Anzahl Trainingsspiele

Um den Zusammenhang zwischen der Anzahl der Trainingsspiele und dem ELO-Wert eines DQN-Spielers zu untersuchen, werden mehrere DQN-Spieler mit einer unterschiedlichen Anzahl von Trainingsspielen trainiert.

Das neuronale Netzwerk des DQN-Spielers besteht aus dem Input- und Output-Layer sowie einem Fully-Connected-Layer mit 30 Neuronen. Das Layer verwendet Leaky-ReLU [29] als Activation-Function. Das Output-Layer verwendet die Activation-Function Tanh [30]. Die Cost-Function wird so gewählt, dass $\frac{\partial L}{\partial z^N}$ der Differenz zwischen dem Output des neuronalen Netzwerkes und dem Target T entspricht. Das neuronale Netzwerk besitzt eine konstante Learning-Rate von 0.01 für die Weights und Biases.

Die Anzahl der Experiences, die nach jedem Zug für das Training verwendet werden, beträgt 32. Das Replay-Memory hat eine Grösse von $9 \cdot 10^6$, was nicht komplett gefüllt wird und deswegen als unlimitiert betrachtet werden kann. Der Parameter c, welcher angibt nach wie vielen Zügen die Weights und Biases des Main-Netzwerkes ins Target-Netzwerk kopiert werden, beträgt 512. Der Discount-Factor γ ist 0.99. Der Epsilon-Wert startet bei 1 und wird nach jedem Spiel um den Faktor 0.99 verringert, bis er den Wert 0.3 erreicht. Anschliessend bleibt er konstant. Für die Anzahl der Trainingsspiele werden die Zweierpotenzen von 2^0 bis und mit 2^{14} verwendet.

Für jede Anzahl der Trainingsspiele gibt es drei DQN-Spieler, was den Zufallsfaktor verringern soll. Der dargestellte ELO-Wert ist der Durchschnitt der ELO-Werte der drei DQN-Spieler.

In Abbildung 6.13 ist ersichtlich, wie sich die Anzahl Trainingsspiele auf den ELO-Wert auswirkt.

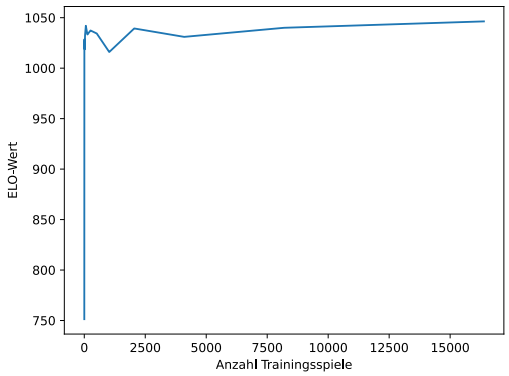


Abbildung 6.13: DQN Spiele

Zu Beginn ist ein starker Anstieg des ELO-Wertes zu erkennen. Ab 4 Trainingsspielen bleibt der ELO-Wert jedoch relativ konstant.

Die hier verwendeten DQN-Spieler lernen schnell, wie sie durch eine vertikale Reihe gewinnen können. Allerdings lernen sie nicht, wie sie horizontal oder diagonal gewinnen können und sie lernen auch nicht, wie sie einen Sieg des Gegners verhindern können. Der Grund dafür sind vermutlich nicht optimal gewählte Parameter.

Der Epsilon-Wert und seine Abnahme dürften der Grund sein, wegen dem die DQN-Spieler nicht lernen, wie sie horizontal oder diagonal gewinnen können. Wenn er zu tief ist, dann hat der DQN-Spieler keine Möglichkeit mehr, genug neue Züge auszuprobieren.

Die Learning-Rate könnte der Grund sein, weshalb der Agent nicht lernt, wie er den Sieg des Gegners verhindern kann. Wenn sie zu klein ist, dann lernt der Agent zu langsam und benötigt viel mehr Training, bis er seine Weights und Biases genug angepasst hat. Ist sie aber zu gross, so ist es möglich, dass sich die Weights und Biases des neuronalen Netzwerkes zu stark verändern und der Loss durch die zu grosse Änderung der Weights und Biases zunimmt.

Um diese Vermutungen zu überprüfen, müsste mehr Zeit in das Finden der optimalen Parameter investiert werden.

6.4 Vergleich computergesteuerte Spieler

Schlussendlich werden die verschiedenen computergesteuerten Spieler verglichen.

Für den MCTS-Algorithmus wird ein Spieler trainiert, dessen Simulation auf minimaler Logik basiert. Die Anzahl der Simulationen wird auf 2^3, die Anzahl der Iterationen auf 2^7 und die Anzahl der Trainingsspiele auf 1 fixiert.

Für den DQN-Algorithmus wird ein Spieler anhand der Parameter aus Unterabschnitt 6.3.1 trainiert. Die Anzahl der Trainingsspiele wird auf 2^3 fixiert.

Ausserdem wird der zufällige Spieler und der auf minimaler Logik basierende Spieler ebenfalls verglichen.

Diese Spieler spielen mehrere Male gegeneinander, woraus die in Tabelle 6.1 ersichtlichen ELO-Werte resultieren.

Spieler	ELO-Wert
MCTS	1490
Spieler mit minimaler Logik	1148
DQN	833
zufälliger Spieler	529

Tabelle 6.1: Vergleich computergesteuerte Spieler

Der MCTS-Spieler schneidet am besten ab. Er entwickelt Strategien und schlägt damit zuverlässig alle anderen computergesteuerten Spieler. Der zweitbeste Spieler ist der Spieler mit minimaler Logik. Dieser Spieler gewinnt, wenn es die Möglichkeit gibt und verliert nur, wenn es nicht anders geht. Der DQN-Spieler schneidet am drittbesten ab. Wie in Unterabschnitt 6.3.1 beschrieben, lernt er, wie er vertikal gewinnen kann, doch er lernt nicht, wie er das Gewinnen des Gegners verhindern kann oder wie er horizontal oder diagonal gewinnen kann. Er hat somit eine Strategie gelernt, durch welche er jedoch nicht zuverlässig gewinnt. Der zufällige Spieler schneidet am schlechtesten ab, denn er verfolgt keine Strategie.

Kapitel 7

Rück- und Ausblick

Im Laufe diese Arbeit wurde viel Wissen über neuronale Netzwerke, künstliche Intelligenz und Reinforcement-Learning erlangt. Doch auch durch die Implementation und Auswertung des Programmes wurden neue Erkenntnisse gewonnen. Das Wissen zur Programmiersprache C++ wurde erweitert und das Programmiermodell CUDA erlernt. Auch Erfahrungen zum Arbeiten an einem grossen Programmierprojekt wurden gemacht: es wurden Bash-Skripte geschrieben, um Prozesse zu automatisieren und Git wurde als Versionsverwaltungssystem verwendet. Technische Probleme, zum Beispiel mit der SSH-Verbindung, traten auf und lehrten, auch in schwierigen Situationen die Motivation und den Fokus auf das Ziel nicht zu verlieren.

Das Programm erfüllt das Ziel dieser Arbeit. MCTS und DQN wurden implementiert und lernten Strategien für das Spiel *Vier Gewinnt*. Fertig ist das Programm jedoch nicht, denn es gibt zahllose Möglichkeiten zur Erweiterung und Verbesserung. Neben den in der Arbeit bereits erwähnten möglichen weiteren Vertiefungen könnte auch die Laufzeit zum Beispiel durch Parallelisierung des MCTS-Algorithmus oder durch weitere Optimierungen des neuronalen Netzwerkes reduziert werden. Ausserdem könnten die Algorithmen an verschiedenen Stellen erweitert werden, zum Beispiel durch Einbauen anderer Activation-Functions für das neuronale Netzwerkes oder durch eine andere Strategie zum Bewerten der Spielstände im MCTS-Algorithmus.

Durch die Arbeit wurden viele neuen Erfahrungen gesammelt und Ideen für weitere Projekte angestossen.

Dank

An dieser Stelle möchte ich mich gerne bei allen Personen bedanken, die mich während der Maturaarbeit unterstützt haben. Angefangen bei meinem Betreuer, Dr. Ivo Blöchliger, der mich während dem Prozess der Arbeit bei allen Fragen unterstützt hat. Weiter möchte ich meiner Familie für die Rücksichtnahme und das Gegenlesen meiner Arbeit danken. Zuletzt geht mein Dank an Linus Lüchinger, der mir bei Problemen immer weiterhelfen konnte und mich in schwierigen Situationen motiviert hat, weiterzumachen.

Abbildungsverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] N. Wang, B. Qi und D. Yao. „Introduction to Competitive Programming.“ (o.J.), Adresse: https://usaco.guide/general/intro-cp?lang=cpp (besucht am 21. 12. 2023).

[2] M. Nielsen. „Neural Networks and Deep Learning.“ (2019), Adresse: http://neuralnetworksanddeeplearning.com (besucht am 13. 11. 2023).

[3] Martín Abadi, Ashish Agarwal, Paul Barham u. a. „TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.“ Software available from tensorflow.org. (2015), Adresse: https://www.tensorflow.org/ (besucht am 22. 12. 2023).

[4] A. Paszke, S. Gross, F. Massa u. a. „PyTorch: An Imperative Style, High-Performance Deep Learning Library.“ H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox und R. Garnett, Hrsg. (2019), Adresse: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf (besucht am 22. 12. 2023).

[5] A. Warehouse. „AI Warehouse.“ (2023), Adresse: https://www.youtube.com/@aiwarehouse (besucht am 22. 12. 2023).

[6] OpenAI. „Introducing ChatGPT.“ (2022), Adresse: https://openai.com/blog/chatgpt (besucht am 14. 01. 2024).

[7] C. Gordon. „ChatGPT Is The Fastest Growing App In The History Of Web Applications.“ (2023), Adresse: https://www.forbes.com/sites/cindygordon/2023/02/02/chatgpt-is-the-fastest-growing-ap-in-the-history-of-web-applications/?sh=4a7766f8678c (besucht am 21. 01. 2024).

[8] I. Gouy. „benchmarksgame.“ (2023), Adresse: https://salsa.debian.org/benchmarksgame-team/benchmarksgame (besucht am 21. 01. 2024).

[9] NVIDIA. „CUDA Toolkit.“ (2024), Adresse: https://developer.nvidia.com/cuda-toolkit (besucht am 23. 01. 2024).

[10] H. Oss. „connect four.“ (2024), Adresse: https://github.com/pizzamoeger/connect_four (besucht am 25. 01. 2024).

[11] M. D. P.-T. Vogt. „Applying Monte Carlo Search and Monte Carlo Tree Search on Embedded Systems to Play Connect Four with a Robotic Arm.“ (2020), Adresse: https://carloconnect.com/pinheiro-torres_vogt_thesis_20.pdf (besucht am 14. 01. 2024).

[12] V. Mnih, K. Kavukcuoglu, D. Silver u. a., „Human-level control through deep reinforcement learning,“ *Nature*, Jg. 518, S. 529–533, 2015. Adresse: https://doi.org/10.1038/nature14236 (besucht am 14. 01. 2024).

[13] O. Sushkov. „Deep Q-Networks.“ (2016), Adresse: https://osushkov.github.io/deepq/ (besucht am 23. 01. 2024).

[14] P. Pons. „Solving Connect 4.“ (2016), Adresse: http://blog.gamesolver.org/solving-connect-four/01-introduction (besucht am 30. 11. 2023).

[15] SDL. „About SDL.“ (o.J.), Adresse: https://www.libsdl.org/ (besucht am 22. 12. 2023).

[16] R. Mittal. „What is An ELO Rating?“ (2020), Adresse: https://medium.com/purple-theory/what-is-elo-rating-c4eb7a9061e0 (besucht am 19. 12. 2023).

[17] M. Mazzola. „Implementing the Elo Rating System.“ (2020), Adresse: https://mattmazzola.medium.com/implementing-the-elo-rating-system-a085f178e065 (besucht am 19. 12. 2023).

[18] S. Sharma. „Monte Carlo Tree Search.“ (2018), Adresse: https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa (besucht am 30. 11. 2023).

[19] J. Hui. „Monte Carlo Tree Search (MCTS) in AlphaGo Zero.“ (2018), Adresse: https://jonathan-hui.medium.com/monte-carlo-tree-search-mcts-in-alphago-zero-8a403588276a (besucht am 30. 11. 2023).

[20] M. Liu. „General Game-Playing With Monte Carlo Tree Search.“ (2017), Adresse: https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238 (besucht am 26. 09. 2023).

[21] P. Sommer, R. Fischer und D. Wolleb-Graf. „Graph Theory.“ (o.J.), Adresse: https://soi.ch/wiki/graphs/ (besucht am 20. 01. 2024).

[22] L. Hardesty. „Explained: Neural networks.“ (2017), Adresse: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414 (besucht am 19. 12. 2023).

[23] NVIDIA. „CUDA C++ Programming Guide.“ (2023), Adresse: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (besucht am 14. 12. 2023).

[24] J. Sanders und E. Kandrot, *CUDA by Example*. Addison-Wesley Professional, 2010, ISBN: 0131387685.

[25] supercomputingblog. „CUDA – Tutorial 5 – Performance of atomics.“ (2009), Adresse: http://supercomputingblog.com/cuda/cuda-tutorial-5-performance-of-atomics (besucht am 14. 12. 2023).

[26] M. Harris. „Optimizing Parallel Reduction in CUDA.“ (2007), Adresse: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf (besucht am 18. 08. 2023).

[27] Y. LeCun, C. Cortes und C. Burges, „MNIST handwritten digit database,“ *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, Jg. 2, 2010. (besucht am 22. 12. 2023).

[28] deeplizard. „Reinforcement Learning - Developing Intelligent Agents.“ (2018), Adresse: https://deeplizard.com/learn/video/nyjbcRQ-uQ8 (besucht am 08. 01. 2024).

[29] PyTorch. „LeakyReLU.“ (2023), Adresse: https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html (besucht am 17. 01. 2024).

[30] MathWorks. „tanh.“ (2024), Adresse: https://www.mathworks.com/help/matlab/ref/tanh.html (besucht am 08. 01. 2024).

Anhang A

Eigenständigkeitserklärung

«Ich bestätige mit meiner Unterschrift, dass ich meine Maturaarbeit selbständig verfasst und in schriftliche Form gebracht habe, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind. Mir ist bekannt, dass eine Maturaarbeit, die nachweislich ein Plagiat gemäss Art. 1quater des Maturitätsprüfungsreglements des Gymnasiums (s. auch Maturaarbeitsbroschüre) darstellt, als schwerer Verstoss gewertet wird.»

Ort & Datum

Unterschrift