**Dokumentation**

**Neuronales Netzwerk für Tic-Tac-Toe**

Inhalt

[Programmieren der Game Engine 1](#_Toc168768623)

[Grundidee des neuronalen Netzwerkes 1](#_Toc168768624)

[Daten Sammeln 2](#_Toc168768625)

[Training und Optimierung des neuronalen Netzwerkes 2](#_Toc168768626)

[KI Game Engine 4](#_Toc168768627)

[Problem finden und lösen 4](#_Toc168768628)

# Programmieren der Game Engine

Zuerst brauchte ich eine Game Engine für Tic-Tac-Toe. Die Engine sollte erkennen, welcher Spieler gerade am Zug ist, ob der eingegebene Zug regelkonform ist und ob ein Spieler gewonnen hat oder es ein Unentschieden gab. Ausserdem habe ich darauf geachtet, dass die Spieldaten gut verständlich für das neuronale Netzwerk sind. Darum habe ich mich dafür entschieden, dass ein leeres Feld durch eine Null repräsentiert wird, der erste Spieler durch eine Eins und der zweite durch eine Minuseins.

# Grundidee des neuronalen Netzwerkes

Als nächstens habe ich mir Gedanken darüber gemacht, wie und mit welchen Daten ich mein neuronales Netzwerk trainiere. Dabei habe ich mich für Supervised Learning entschieden, da wir bereits im Unterricht angeschaut haben, wie man ein neuronales Netzwerk mit Supervised Learning trainiert und ich somit ein besseres Verständnis davon habe. Als Trainingsdaten bekommt das Netzwerk einen Vektor mit neun Einträgen; jeder dieser Einträge repräsentiert ein Feld auf dem Tic-Tac-Toe Brett und der Vektor einen Spielzustand. Als Outputdaten brauchte ich den besten Zug, welchen ich zuerst durch einen Minimax-Algorithmus berechnete. Somit sollte mein Netzwerk am Schluss perfekt spielen.

# Daten Sammeln

Zuerst änderte ich meine Game Engine so, dass zwei Player, die zufälligen Züge machen, gegeneinander spielen. Damit habe ich 10'000 mögliche Spielverläufe generiert. Als nächstens liess ich von ChatGPT einen Minimax-Algorithmus schreiben, welchen ich dann noch abänderte, damit er meine Daten versteht. Nun konnte ich die Spielzustände in den Minimax-Algorithmus einlesen und generierte so meine Trainingsdaten. Die rund 80‘000 Spielzustände und perfekten Züge habe ich dann in zwei unterschiedlichen TXT-Dateien gespeichert.

# Training und Optimierung des neuronalen Netzwerkes

Für das neuronale Netzwerk habe ich das Programm vom Unterricht genommen und leicht abgeändert, damit es meine Trainingsdaten brauchen kann. Als nächstens kam das Grundgerüst des Netzwerkes, hier begann ich mit einem Hidden-Layer, welcher aus neun Neuronen besteht, und einem Output-Layer bestehend aus neun Neuronen. Für die Aktivierungsfunktion des Hidden-Layers habe ich über Tanh und ReLu nachgedacht. Schlussendlich habe ich mich für Tanh entschieden, da ich mit 1, 0 und -1 arbeite und ReLu für alle Zahlen, die null oder kleiner sind, null ausgibt und es somit zu Komplikationen kommen kann. Für den Output-Layer nahm ich die Softmax Aktivierungsfunktion, welche einfach zu verstehende Outputs generiert. Als Fehlerfunktion benutzte ich „Mean squered error“ und als Optimizer SGD.

Ich habe dann verschiedene Batch Sizes und Anzahl Epochen durchlaufen lassen. Hierbei lernte das Netzwerk eher langsam. Mit einer Batch Size von 50 kam es nach etwa 250 Epochen auf eine Genauigkeit von rund 65 %, lernte danach viel langsamer und hatte nach 1500 Epochen noch keine Genauigkeit von 80 % erreicht.

Graphen mit (Tanh/Softmax/mean_squered_error/SGD)


Mit diesem Aufbau kam ich also nicht weiter, ich versuchte dann zuerst mal über den Optimizer und die Fehlerfunktion ein besseres Resultat hinzukriegen. Den Optimizer wechselte ich zu „adam“ und die Fehlerfunktion zu „categorical\_crossentropy“. Diese Kombination hatte anfangs eine grosse Lernrate und war nach weniger als 50 Episoden schon auf einer Genauigkeit von 75 %, jedoch war der Loss im Vergleich zu vorher deutlich grösser. Die anfänglich gute Lernrate nahm aber auch rasch ab und nach 200 Episoden hatte das neurale Netzwerk eine Genauigkeit von 76 %.

Ein Bild, das Text, Diagramm, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

Der neue Optimizer und die neue Fehlerfunktion sorgten für eine schnellere Lernrate zu Beginn, jedoch gab es auch hier eine unsichtbare Grenze bei der Genauigkeit, welche die KI nicht überschreiten konnte. Meine Schlussfolgerung daraus war, dass ein Hidden-Layer für Tic-Tac-Toe nicht ausreichen wird. Ich erweiterte mein neurales Netzwerk um einen zweiten Hidden-Layer und änderte die Anzahl Neuronen in den beiden Hidden-Layer zu 729 und 81. Die Fehlerfunktion und den Optimizer habe ich gleich gelassen. Die neue Kombination war ein voller Erfolg, das Netzwerk hatte mit einer Batch Size von 200 bereits nach 20 Epochen eine Genauigkeit von 99 % und nach 100 Epochen war die Genauigkeit auf 100 % und der Loss war praktisch null.

Ein Bild, das Text, Diagramm, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

# KI Game Engine

Da ich nun ein funktionierendes neurales Netzwerk hatte, musste ich nur noch meine Game Engine so anpassen, dass man gegen die KI spielen kann. Da ich mein Programm von Anfang an so programmiert habe, dass es verständlich für die KI ist, war dies keine grosse Aufgabe. Als ich dann einige Testspiele gegen die KI gemacht habe, ist mir aufgefallen, dass sie nicht perfekt spielt und ausserdem illegale Spielzüge macht.

# Problem finden und lösen

Da die KI eine Genauigkeit von 100 % hatte, konnten nur die Trainingsdaten für das Problem verantwortlich sein und dabei kam ich auf zwei Fehlerquellen. Einerseits kann es sein, dass ich zu wenig verschiedene Spielzustände habe und er darum gewisse Zustände gar nicht kennt oder der Minimax-Algorithmus ist falsch. Ich machte mich zuerst an die erste Fehlerquelle und erstellte mir ein Programm, welches alle möglichen Spielzustände, bei denen man noch mindestens einen Zug machen kann, generiert. Das Programm generiert hier zuerst alle möglichen neunstelligen Kombinationen aus den drei Zahlen und dann filtert es die raus, welche bei einem Tic-Tac-Toe Game entstehen können und bei denen noch mindestens ein Zug möglich ist. Als ich dann mit diesen Daten mein neuronales Netzwerk trainierte, machte es zwar keine illegalen Züge mehr, jedoch konnte ich immer noch problemlos gegen die KI gewinnen. Somit muss der Minimax-Algorithmus auch falsch sein. Ich gab also auf, ChatGPT für den Minimax-Algorithmus zu brauchen und schrieb meinen eigenen Minimax-Algorithmus. Dank des neuen Minimax-Algorithmus war ich nicht mehr in der Lage, gegen die KI zu gewinnen. Für weitere Tests liess ich mein Netzwerk gegen den Algorithmus von Google auf Schwierigkeit unmöglich spielen, jeweils zehn Spieler als Spieler eins und zehn als Spieler zwei und von den 20 Spielen gab es 20 Unentschieden. Damit denke ich, dass meine KI in Tic-Tac-Toe nicht verlieren kann. Sie ist aber nicht perfekt, da die KI gegen den Google-Algorithmus auf mittlerer Stufe auch hauptsächlich Unentschieden gespielt hat, obwohl man gegen den Algorithmus mit Leichtigkeit gewinnen kann.

Quellenverzeichnis

Never stop building: „ *Tic Tac Toe: Understanding the Minimax Algorithm* “. *https://www.neverstopbuilding.com/blog/minimax (8. Juni 2024).*