Bearbeitungsbeginn: [01.03.2023]

Vorgelegt am: [tt.mm.jjjj]

**Thesis**

zur Erlangung des Grades

**Bachelor of Science**

im Studiengang Medieninformatik

an der Fakultät Digitale Medien

***Alexander Reiprich***

***Matrikelnummer: 263006***

**Computer Vision als Werkzeug zur Steuerung von Videospielen – Analyse und Entwicklung eines Programmes basierend auf Reinforcement Learning**

*Erstbetreuer: Prof. Dr. Uwe Hahne*

*Zweitbetreuer: Prof. Dr. Ruxandra Lasowski*

Inhaltsverzeichnis

[Abstract 3](#_Toc137036902)

[1. Einführung 4](#_Toc137036903)

[1.1. Motivation und Ziel der Arbeit 4](#_Toc137036904)

[1.2. Aufbau der Thesis 5](#_Toc137036905)

[2. Grundlagen 6](#_Toc137036906)

[2.1. Stand der Forschung 6](#_Toc137036907)

[2.2. Computer Vision 6](#_Toc137036908)

[2.3. Machine Learning 6](#_Toc137036909)

[3. KIs im Bereich Unterhaltung 7](#_Toc137036910)

[3.1. Videospiele 7](#_Toc137036911)

[3.2. Livestreaming 7](#_Toc137036912)

[4. Entwicklung 7](#_Toc137036913)

[4.1. Spielprinzip, Projektablauf und Anforderungen 7](#_Toc137036914)

[4.2. Umsetzung der verschiedenen Bestandteile 9](#_Toc137036915)

[4.2.1. Screenshots 11](#_Toc137036916)

[4.2.2. Input Sending 14](#_Toc137036917)

[4.2.3. Pattern Matching 16](#_Toc137036918)

[4.2.4. Kombination der Module 18](#_Toc137036919)

[4.2.5. Aufsetzen eines Environments 20](#_Toc137036920)

[4.2.6. Jupyter Notebook 28](#_Toc137036921)

[4.2.7. Custom Theming 30](#_Toc137036922)

[4.2.8. Tesseract 34](#_Toc137036923)

[4.2.9. Erster Lernversuch 40](#_Toc137036924)

[4.2.10. Neues Rewardsystem 43](#_Toc137036925)

[4.3. Bewertung und Einordnung 49](#_Toc137036926)

[5. Fazit und Ausblick 49](#_Toc137036927)

[5.1. Fazit 49](#_Toc137036928)

[5.2. Ausblick 49](#_Toc137036929)

[Literaturverzeichnis 49](#_Toc137036930)

# Abstract

Videospiele bilden einen wichtigen Aspekt der Unterhaltung in der heutigen Gesellschaft. Spieleentwickler:innen verbringen viel Zeit damit, Spiele zu entwickeln und zu testen, um dem Konsument:innen schlussendlich das ansprechendste Produkt zu liefern. Besonders bei Spielen mit einer Online-Multiplayer-Funktionalität ist ein nicht unwichtiger Teil hiervon die Entwicklung von Maßnahmen gegen die Manipulation durch andere Programme. Diese können in großem Maße das Spielerlebnis anderer beeinträchtigen, in dem sie den Nutzenden unfaire Vorteile geben.

Während ein Großteil der sogenannten Cheating-Software direkt auf die Spieldateien zugreift, gibt es auch andere Möglichkeiten, durch ein externes Programm auf das Spiel Einfluss zu nehmen. In dieser Arbeit soll die Technologie der Computer Vision zusammen mit Machine Learning im Hinblick auf das Steuern von Videospielen untersucht werden. Als Untersuchungsgegenstand liegt hierbei das 2001 erschienene Rhythmusspiel „StepMania“ zugrunde, welches extern durch eine künstliche Intelligenz gesteuert werden soll.

Die Arbeit soll in einen theoretischen sowie praktischen Teil untergliedert werden, wobei der Fokus der Arbeit auf den praktischen Teil gerichtet werden soll. Im Theorieteil wird sich mit dem generellen Konzept von Machine Learning und KI im Bezug auf Videospiele auseinandergesetzt. Hierbei soll die Funktionsweise von Lernalgorithmen behandelt werden, und wie diese Art der KI bereits in Spielen angewandt wird. Ebenso wird auf die Anwendung von KI in der Unterhaltungsbranche, spezifisch im Livestreaming-Bereich eingegangen.

Im praktischen Teil der Arbeit soll selbst mithilfe von Computer Vision und einem Lernalgorithmus ein Programm entwickelt werden, welches das bereits angesprochene Spiel „StepMania“ lernen und spielen soll. Die technische Umsetzung soll in Python mit Unterstützung durch verschiedene Libraries wie OpenCV und Ähnliche erfolgen. Das Ergebnis dieses Teils soll ein Programm sein, mit welchem man ein beliebiges Level in „StepMania“ mit 100% Genauigkeit abschließen kann.

# Einführung

## Motivation und Ziel der Arbeit

Künstliche Intelligenz ist derzeit so im Trend wie noch nie. Die Anzahl der Publikationen im Bereich KI hat sich im Vergleich von 2010 zu 2021 mehr als verdoppelt, insbesondere in den Themenbereichen Pattern Recognition und Machine Learning (Zhang et al. 2022, S. 17-19). Durch den Hype des Chat-Bots „Chat-GPT“ Anfang 2023, als auch durch das wachsende Interesse an KI-Kunst durch Dienste wie „midjourney“ sind die Fähigkeiten von KI ein wachsendes Thema, nicht nur im Bereich der Informatik, sondern der ganzen Gesellschaft - an manchen Hochschulen und Universitäten wird beispielsweise das Nutzen von künstlicher Intelligenz, z.B. in Form von ChatGPT, bereits verboten (Barthel und Ciesielski 2023).

Dabei beschränkt sich das Angebot von KI nicht bloß auf Wissen. Auch „Kunst“ von künstlicher Intelligenz ist mittlerweile möglich – beispielsweise durch die KI midjourney eines amerikanischen Forschungsinstituts. Nach der Eingabe, was für ein Bild erzeugt werden soll, produziert die KI vier unterschiedliche Varianten des gewünschten Motivs, wobei die Qualität stark variiert (Spehr 2023). Mit dieser neuen Art der Bilderzeugung kommen natürlich auch Probleme – sowohl moralische, als auch juristische. Nachdem 2022 ein Werk einer KI als Sieger eines Kunstwettbewerbes gekürt wurde, gab es Diskussionen, inwiefern Bilder, welche von KI erzeugt wurden, tatsächlich „Kunst“, und die Ersteller:innen tatsächlich „Künstler“ sind (Roose 2022). Ebenfalls war es unklar, ob mit KI erstellte Bilder rechtlich unter Copyright geschützt sind. Nutzer:innen behaupten, die eingegebenen Befehle, die zur Kreation des Bildes geführt haben, wären ihr eigenes Werk, was jedoch nach einem Urteil des U.S. Copyright Office nicht der Fall ist (Novak 2023).

Führt man diesen Gedanken, dass Künstliche Intelligenz kreativ wird, weiter, stößt man schnell darauf, dass es mittlerweile auch Kanäle auf der Livestream-Plattform Twitch gibt, welche komplett KI-gesteuert sind. Eine der bekanntesten Kanäle in diesem Bereich ist „Neuro-Sama“. In den Livestreams dieses Kanals sitzt keine Person vor der Kamera, sondern eine KI, welche Videospiele spielt und auf ihre Zuschauer eingeht (Xiang 2023).

All diese Dinge haben mich dazu inspiriert, das Thema KI im Bereich Medien, und insbesondere im Bereich Videospiele, genauer zu untersuchen. Diese Arbeit ist dabei in zwei Teile unterteilt – im theoretischen Teil soll zuerst die Funktionsweise von KI erklärt werden, wobei dort genauer auf die Funktionsweise von Reinforcement Learning, einem spezifischen Lernalgorithmus, eingegangen werden soll. Ebenso soll das Thema KIs zur Medienproduktion und -verarbeitung anhand von zwei Beispielen behandelt werden. Im praktischen Teil dieser Arbeit soll selbst eine KI zur Steuerung eines Videospiels entwickelt und trainiert werden. Dabei wird das Vorgehen dokumentiert, der Vorgang und die Herangehensweise beschrieben und erklärt, und schlussendlich retrospektiv analysiert. Während das Ziel dieses Teils zwar ein fertiges Produkt ist, soll der Fokus dabei mehr auf dem Prozess der Entwicklung liegen – welche Probleme aufkommen, wie diese gelöst werden, was es für Alternativen gibt, und wie das Projekt schlussendlich kritisch zu betrachten ist.

## Aufbau der Thesis

Im ersten Kapitel dieser Arbeit geht es um die Thesis an sich. Dabei soll im ersten Unterkapitel zuerst die Motivation dargestellt werden um anschließend den theoretischen und praktischen Teil dieser Ausarbeitung zu erläutern. Im zweiten Unterkapitel wird der Aufbau der Arbeit genauer beschrieben.

Das zweite Kapitel beschäftigt sich mit den Grundlagen, welche dargestellt werden, um ein genaueres Verständnis des Themas und der nachfolgenden Kapitel zu gewährleisten.

Nachdem die Grundlagen des Themas im zweiten Kapitel definiert wurden, werden diese genutzt um im dritten Kapitel genauer auf den Bereich KI in Medienproduktion und -verarbeitung einzugehen. Hier werden zwei Beispiele analysiert, welche in diesen Bereich eingeordnet werden können.

Das vierte Kapitel beinhaltet den praktischen Teil der Arbeit. Hier wird zuerst der Aufbau des Projektes sowie die Anforderungen beschrieben, um dann in den darauffolgenden Unterkapiteln chronologisch auf den Entwicklungsprozess einzugehen. Eines dieser Unterkapitel beinhaltet dabei entweder ein Modul bzw. Funktionsweise des Programmes oder einen wichtigen Schritt in der Entwicklung. Abschließend wird das Vorgehen analysiert, bewertet und kritisiert.

Im fünften Kapitel werden die Ergebnisse der vorangegangenen Kapitel in einem Fazit zusammengefasst, und auf die Zukunft von künstlicher Intelligenz in Medien eingegangen.

# Grundlagen

## Stand der Forschung

*Der Stand der Forschung soll kurz den generellen Stand in Bezug auf KIs in allen Bereichen behandeln, und insbesondere auf die aktuelle Entwicklung im Bereich KIs als Spieler von Videospielen eingehen.*

## Computer Vision

*In diesem Kapitel soll das Konzept der Computer Vision erklärt werden, mit dem in der Umsetzung gearbeitet wird. Dabei werden Themen wie z.B. Pattern Recognition angesprochen.*

## Machine Learning

*Ähnlich zu dem Kapitel 2.2. wird hier Machine Learning erklärt. Es soll eine Übersicht über die Funktionsweise von Künstlicher Intelligenz geben – da es viele unterschiedliche Methoden und Herangehensweisen gibt, sollen sich zwei oder drei rausgesucht werden, welche für unterschiedliche Zwecke genutzt werden (z.B. evolutionary algorithms, reinforcement learning, …). Dabei soll stets auf den praktischen Part eingegangen werden, um so Theorie und Praxis zu verbinden und ein Beispiel zu haben, an dem sich Leser:innen orientieren können.*

# KIs im Bereich Unterhaltung

## Videospiele

*Hier soll hauptsächlich auf die Neuro-Sama KI eingegangen werden, sowie weitere Projekte, welche mithilfe von Lernalgorithmen Videospiele steuern. Dabei soll gezeigt werden, welche Technik dahinter steckt – das Wissen aus Kapitel 2.3. soll hier wieder aufgegriffen werden.*

## Livestreaming

*Während in Kapitel 3.1. nur auf den „Spiel-Teil“ von Neuro-Sama eingegangen werden soll, wird hier nochmal explizit darauf eingegangen, dass Neuro-Sama auf Nachrichten der Zuschauer eingeht, und ebenfalls auf die Geschehnisse auf dem Bildschirm reagiert. Ebenso soll auf das Projekt „Nothing, Forever“ eingegangen werden – ein Twitch-Kanal, welcher kontinuierlich eine fiktive Episode der amerikanischen Sitcom „Seinfeld“ streamt, die komplett KI generiert ist.*

# Entwicklung

*In der Umsetzung soll ein Computerprogramm konzipiert und umgesetzt werden, welches mit Hilfe eines Lernalgorithmus ein simples Rhythmusspiel erlernt und spielt.*

## Spielprinzip, Projektablauf und Anforderungen

*Hier sollen die Anforderungen an das fertige Produkt dargestellt werden. Ebenso soll das Spielprinzip erklärt werden, und welche Herausforderungen sich dabei präsentieren.*

StepMania 5 ist sowohl ein Open-Source Rythmusspiel, als auch eine Spielengine, welche es dem Nutzer ermöglicht, beliebte Rhythmusspiele auf dem PC zu simulieren und zu spielen. Der Gameplay-Aspekt ist vergleichbar mit beliebten Arcade-Spielen wie Dance Dance Revolution (DDR), jedoch kann StepMania auch mit der Tastatur oder einem Controller gespielt werden, und basiert, anders als DDR nicht nur auf dem Konzept einer Tanzmatte als Eingabegerät. [Quelle hinzufügen]

Der Spieler kann aus verschiedenen sogenannten Charts wählen. Diese sind entweder von anderen Nutzern erstellt, oder beim Download des Spiels bereits enthalten. Jeder Chart besteht aus Pfeilen, welche von unten nach oben scrollen und dabei dem Rhythmus des im Chart spielenden Liedes folgen. Der Spieler hat hier die Aufgabe, zum richtigen Zeitpunkt, wenn die sich bewegenden Pfeile auf die stationären Pfeile am oberen Bildschirmrand treffen, die passenden Tasten zu drücken. Dabei bekommt der Spieler, je nach Genauigkeit des Treffzeitpunktes, Punkte. Hat er den Chart zu Ende gespielt, folgt der Resultscreen, wo der Spieler eine Note bekommt, welche auf der Anzahl und der Genauigkeit der getroffenen Pfeile basiert.



Abb. - Screenshot vom Gameplay von StepMania 5

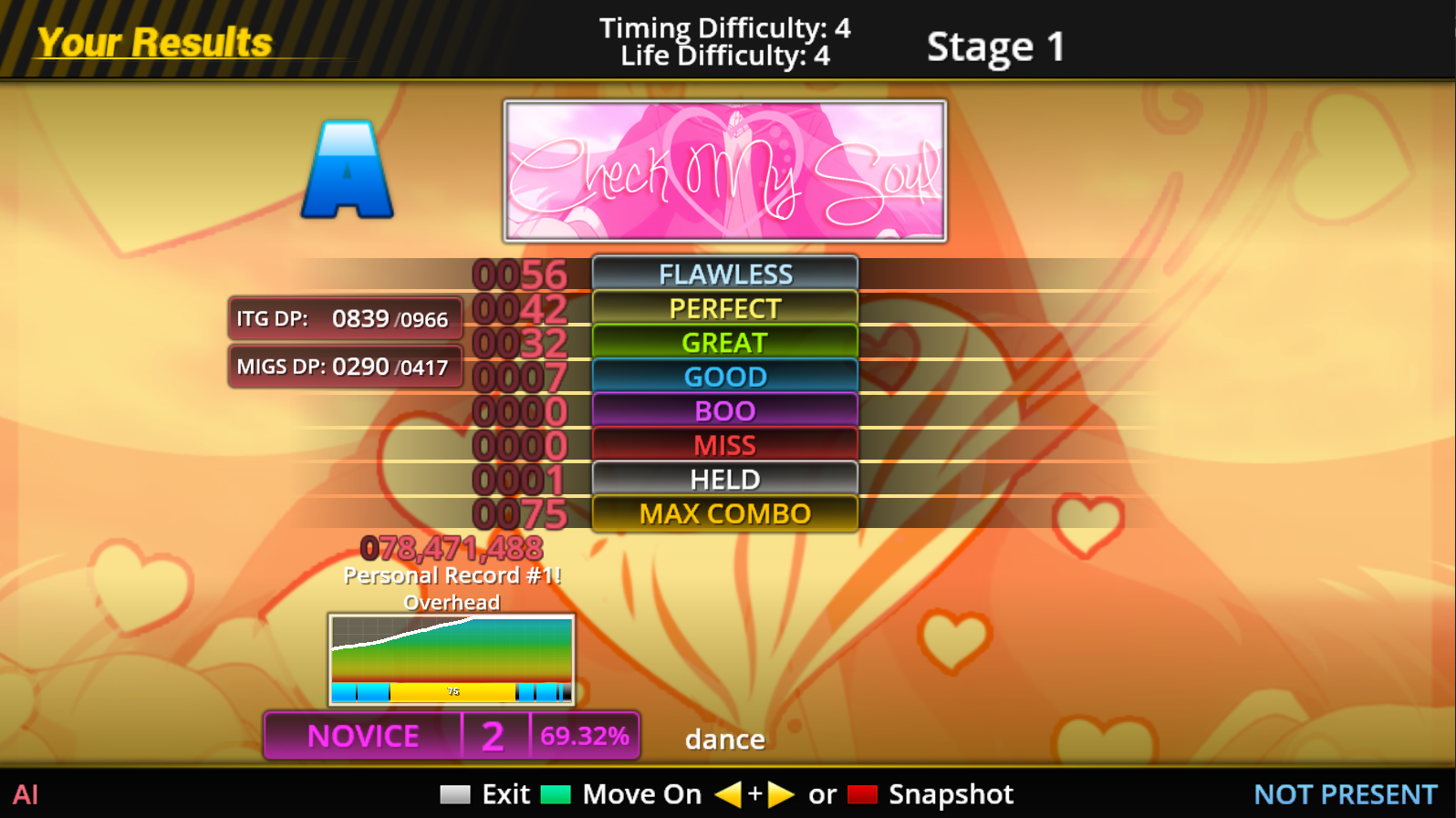


Abb. - Screenshot des Resultscreens nach Abschluss eines Charts

Ziel des Projektes ist es, eine KI zu entwickeln und zu trainieren, die es schafft, StepMania Charts zu spielen. Wichtig zu erwähnen ist dabei, dass es nicht das Ziel ist, eine “perfekte” KI zu trainieren, die jeden beliebigen Chart mit einer einhundertprozentigen Genauigkeit spielen kann, da dafür der Trainingsprozess zu lange dauern würde, und dies den Umfang dieser Thesis übersteigt. Vielmehr geht es um den Prozess des Entwickelns einer solchen KI, welche Probleme dabei auftreten, wie sie gelöst werden können, und wie sich ein Modell nach kurzen Trainingseinheiten verhält. Ein genereller Lernerfolg ist das Ziel, kein perfektes Modell.

Während der Entwicklung der KI und deren Bestandteile wurden alle Schritte und Ansätze dokumentiert. Dies ist in den folgenden Kapiteln dargestellt. Dabei sind auch Ideen und Konzepte aufgeführt, welche es schlussendlich nicht in die finale Version geschafft haben. Warum sich anfangs für diese Ansätze entschieden wurde, und weshalb sie später nicht weiterverfolgt wurden, wird im Laufe der Dokumentation erklärt. Die Kapitel sind in chronologischer Reihenfolge aufgebaut, sodass es sein kann, dass Elemente in einem Kapitel eingeführt, aber in einem späteren Kapitel überarbeitet oder entfernt werden.

## Umsetzung der verschiedenen Bestandteile

*In diesem Kapitel wird der Prozess der Umsetzung beschrieben, unterteilt in die einzelnen Bestandteile/Module. Dabei werden Codebeispiele und Erklärungen herangezogen um das Erklärte besser nachvollziehen zu können.*

Bevor das Projekt komplett aufgesetzt wurde, musste festgelegt werden, welche Informationen aus dem Spiel bzw. für das Spiel benötigt werden, die später im Environment verarbeitet, und zum Lernen genutzt werden sollen.

Hier wurden drei Konzepte etabliert, die als erstes als separate Module umgesetzt wurden, um sie später im Environment zu nutzen.

Screenshots - Der wohl wichtigste Part ist das Aufnehmen von Screenshots, welche dem Modell übergeben werden, und aus denen dann die entsprechenden Aktionen bestimmt werden sollen.

Verschicken von Inputs - Um die Aktionen, die durch das Modell bestimmt wurden, an das Spiel zu übertragen, muss eine Möglichkeit gefunden werden, aus den Aktionen Tastendrücke berechnen zu lassen, welche im Anschluss an das Spiel gesendet werden können.

Pattern Matching - Innerhalb der Screenshots muss das Geschehen interpretiert werden, um akkurates Feedback geben zu können. Dies soll mit der Hilfe von Pattern Matching erfolgen.

Die folgenden Kapitel beschäftigen sich also mit der Umsetzung dieser drei Module, welche dann abschließend kombiniert werden sollen, um untereinander zu interagieren. Bevor tatsächliche künstliche Intelligenz dabei zum Einsatz kommt, sollten, um diese Kombination zu testen, die drei Elemente so untereinander funktionieren, dass beim Erkennen eines Pfeils durch das Pattern Matching ein Signal an das Spiel gesendet wird. Im Optimalfall würde so eine Art “perfektes Modell” entstehen, welches zwar nicht “intelligent” ist, aber die Charts mit ausreichender Genauigkeit spielen kann. Im Anschluss wird dann evaluiert, wie diese Kombination als Einheit funktioniert, welche Schlüsse daraus gezogen werden können, und wie weiter vorgegangen wird.

### Screenshots

Für das Programm müssen schnell viele Screenshots erzeugt werden, um einen kontinuierlichen Input zu gewährleisten. Die Performance ist dabei besonders wichtig, da mehr Bilder pro Sekunde gleichzeitig mehr Informationen für das Modell bedeuten. Um dies zu erreichen wurden zwei Methoden ausprobiert.

Für die erste Methode wurde das Python Package “ImageGrab” genutzt, welches mit folgendem Code einen Screenshot erzeugt hat:

**def** **screenshot**(hwnd, bbox):  
 win32gui.SetForegroundWindow(hwnd)  
 img = ImageGrab.grab(bbox)  
 size = img.size   
 path = r'assets\screenshot.png'  
 img.save(path)

Die zweite Methode, welche innerhalb eines Thesistreffens vorgeschlagen wurde, hat das Package MSS-Tools genutzt, um dasselbe Ergebnis, aber schneller zu erzielen. Hierfür wurde der oben verwendete Code leicht abgeändert.

**def** **screenshot**(hwnd, bbox):  
 win32gui.SetForegroundWindow(hwnd)  
 img = mss.mss().grab(bbox)  
 path = r'assets\screenshot.png'  
 mss.tools.to\_png(img.rgb, img.size, output=path)

Um die beiden Methoden zu vergleichen wurde ein Benchmarktest genutzt, welcher die beiden Methoden in ihrer Geschwindigkeit verglichen hat.

**def** **benchmark**(hwnd, bbox):  
 start = time.time()  
 i = 0  
 **while** i < 100:  
 screenshot(hwnd, bbox, i)  
 i += 1  
 end = time.time()  
 print (**end** - **start**)

Der Index i wurde in den oben genannten Methoden als Übergabeparameter integriert um den File-Output so zu ändern, dass jedes neue Bild unter einem anderen Namen abgespeichert wird, um realistische Ergebnisse zu erhalten.

In der folgenden Tabelle sieht man die durchschnittliche Geschwindigkeit der beiden Methoden nach 20 Ausführungen:

|  |  |
| --- | --- |
| **Methode** | **Zeit** |
| ImageGrab | 7,389 Sekunden |
| MSS-Tools | 14.517 Sekunden |

Auf Grund dieses Benchmarks wurde entschieden, weiterhin mit ImageGrab zu arbeiten.

Nach Hinzufügen eines Aufrufs der Methode ImageGrab.crop() wurde festgestellt, dass die Geschwindigkeit der Funktion sich bemerkbar verschnellert hat, desto kleiner das Bild zugeschnitten wurde. Der Zeitverlust durch das Aufrufen der Funktion wird also durch das Abspeichern der kleineren Datei überschattet. Dementsprechend wurde der Code aus Methode 1 um folgende Zeile ergänzt:

img = img.crop((110, 35, 435, 530))

Um den Prozess des Screenshotting in Kombination mit der Analyse der Bilder zu beschleunigen, wurde ebenfalls eine Konvertierung zum Ende der Funktion hinzugefügt.

opencvImg = img.convert('RGB')  
opencvImg = numpy.array(opencvImg)  
opencvImg = opencvImg[:, :, ::-1].copy()  
return opencvImg

Nachdem das Environment implementiert wurde, war der nächste Schritt die Optimierung. Da nun auch die restliche Umgebung um das Screenshot-Modul herum existiert, wurde ein erneuter Benchmark durchgeführt, welcher nun im vollständigen Environment stattfindet.

Für diesen Test wurden drei Songs, in den Anhängen als Episoden bezeichnet, mit der ImageGrab-Methode und mit der MSS-Tools-Methode durch das Environment absolviert und die Anzahl der Bilder pro Sekunde verglichen.

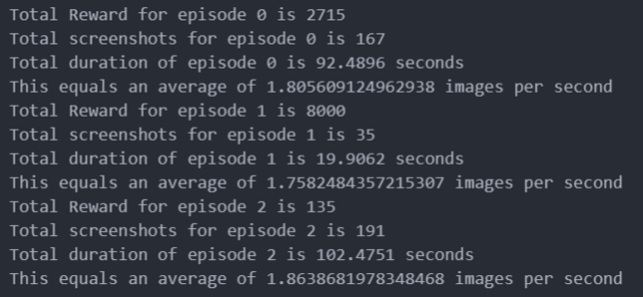


Abb. - Ergebnis des Benchmarks mit MSS-Tools

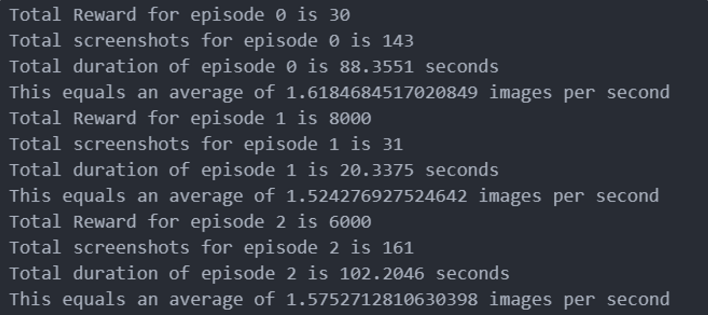


Abb. - Ergebnis des Benchmarks mit ImageGrab

Wie man sehen kann, ist, entgegen des ersten Benchmarks außerhalb des Environments, die MSS-Tools-Methode etwas schneller. Dies scheint zwar relativ wenig, jedoch sind das, wie man beispielsweise an der letzten Episode sehen kann, 30 Bilder mehr, welche das Programm verarbeiten kann.

Zusammen mit dem Umstellen der Screenshot-Methode, wurde das Programm weiter optimiert, in dem pro Step nur ein Screenshot gemacht wird, statt drei. Anfangs wurden drei Screenshots pro Step gemacht - jeweils einer für die Observation des Gameplays, einer für die Berechnung des Rewards, und einer für die Prüfung ob der momentane Song vorbei ist. Das Programm wurde so umgeschrieben, dass nun ein Screenshot gemacht wird, welcher in den drei Methoden auf den relevanten Teil zugeschnitten wird.

Ebenso wurde die Auflösung des Spiels weiter reduziert, um weniger Pixel abfotografieren zu müssen, was die Dateigröße reduziert und das Programm performanter macht.

Dies beiden Änderungen steigerten die Performance um etwa 250%.

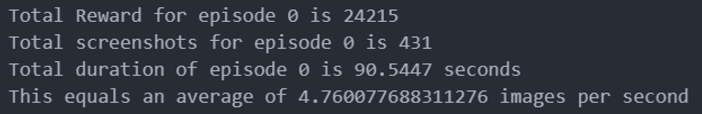


Abb. - Ergebnis des Benchmarks nach den Optimierungen

### Input Sending

Das Programm soll Inputs an das Spiel, welches sich im Vordergrund in einem anderen Fenster befindet, schicken.

Die erste Herangehensweise an dieses Problem basierte auf dem Package pywinauto, bzw. das Modul pywinauto.keyboard mit der Funktion send\_keys(). Folgender Code funktionierte zwar, jedoch nur in Texteditoren oder Inputfelder, nicht im Spiel.

**import** pywinauto  
pywinauto.keyboard.send\_keys({f down})  
pywinauto.keyboard.send\_keys({f up})

Mit diesem Code wurde der Buchstabe “F” in Textfelder eingegeben, dieser Input ist jedoch im Spiel nicht als Eingabe angekommen.

Nach weiterer Recherche kam das Package pydirectinput auf, welches, anders als pywinauto, die Tastenbefehle über DirectX verschickt. Pywinauto nutzt Virtual Key Packets zum Versand des Inputs. Der Unterschied ist hierbei, dass das Verschicken über DirectX explizit für Videospiele vorgesehen ist, und so die Kommunikation von Programm zu Spiel unterstützt und gewährleistet wird.

Der folgende Code sendet mit der Hilfe von pydirectinput den Buchstaben “F” an das Spiel.

**import** pydirectinput  
pydirectinput.keyDown('f')  
pydirectinput.keyUp('f')

Da ein objektorientierter Ansatz verfolgt wird, wurde diese Funktion in eine Klasse umgelagert, wobei die gewünschte Taste per Übergabeparameter angesteuert werden kann.

**class** **SendInput**:  
 **def** **sendKey**(self, key):  
 pydirectinput.keyUp(key)  
 pydirectinput.keyDown(key)

StepMania 5 beinhaltet neben den normalen Pfeilen auch noch andere Varianten - hier soll nur der sogenannte “Hold-Arrow” behandelt werden. Dieser erwartet vom Spieler, dass er die Taste gedrückt hält, bis der Pfeil aufhört. Die momentane Umsetzung des Input-Sendings ermöglicht dies nicht. Daher wurde die sendKey()-Methode in zwei Methoden aufgeteilt, pressKey() und releaseKey(). Ebenso wurde ein globales Array festgelegt, welches festhält, welche Taste zurzeit gedrückt wird.

**class** **SendInput**:  
  
 **def** **holdKey**(self, key):  
 pydirectinput.keyDown(key)  
  
 **def** **releaseKey**(self, key):  
 pydirectinput.keyUp(key)

self.cur\_held\_buttons = {'a': False, 's': False, 'w': False, 'd': False}

### Pattern Matching

Um Informationen aus den Screenshots des Gameplays nehmen zu können, muss eine Möglichkeit gefunden werden, die Pfeile aus den Bildern erkennen zu können.

Mit Hilfe von den Libraries cv2 und numpy konnte eine Funktion zur Erkennung der einzelnen Pfeile und deren Richtungen erstellt werden.

**import** cv2 **as** cv  
**import** numpy **as** np  
  
**def** **analyzeImage**():  
 img\_rgb = cv.imread(r'assets\screenshot.png')  
 **assert** img\_rgb **is** **not** None, "image could not be read"  
 img\_gray = cv.cvtColor(img\_rgb, cv.COLOR\_BGR2GRAY)  
 template = cv.imread(r'assets\small\_arrow\_left.png', cv.IMREAD\_GRAYSCALE)  
 **assert** template **is** **not** None, "template could not be read"  
   
 w, h = template.shape[::-1]  
 res = cv.matchTemplate(img\_gray,template,cv.TM\_CCOEFF\_NORMED)  
 threshold = 0.8  
 loc = np.where(res >= threshold)  
 **for** pt **in** zip(\*loc[::-1]):  
 cv.rectangle(img\_rgb, pt, (pt[0] + w, pt[1] + h), (0,0,255), 2)  
   
 cv.imwrite(r'assets\res.png',img\_rgb)

Der Code nimmt eine der vier Richtungspfeile, welche separat als Datei vorliegen, und vergleicht sie mithilfe des Template Matchings mit dem Screenshot.



Abb. - Vorlage für einen Pfeil

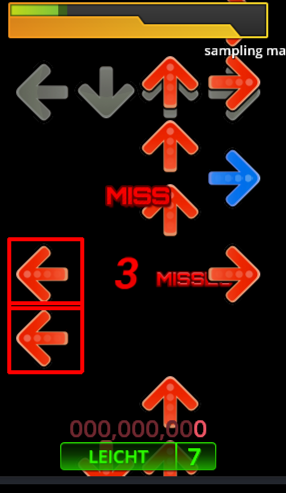


Abb. - Erkannte Pfeile im Spiel

Abschließend hat man so also ein Bild, in welchem die Pfeile auf dem Bildschirm erkannt, und visuell markiert wurden.

Dies ermöglicht nun das Abfragen der Koordinaten der erkannten Bereiche, wodurch man auf die Position der Pfeile schließen kann.

Folgende Fragestellungen ergeben sich nun daraus:

- Wie soll mit andersfarbigen Pfeilen umgegangen werden? Soll separates Pattern Matching ausgeführt werden oder soll das bestehende so umgebaut werden, dass die Farben keinen Unterschied machen?  
- Was passiert bei Pfeilen, die sich überlagern? Pattern Matching kann diese nicht als Pfeile erkennen.  
- Wie soll mit den grauen, statischen Pfeilen am oberen Rand umgegangen werden, da diese bei geringerem Threshold ebenfalls erkannt werden?

Abschließend wurde die Funktion um einen Übergabeparameter erweitert, welche das Bild, welches in der Variable img\_rgb eingelesen wurde, in die Funktion gibt. Dies hat zur Folge, dass die ersten beiden Zeilen der Funktion obsolet wurden, und weitere Nennungen von img\_rgb zu img geändert wurden. Ebenso wurde ein Array angelegt, welches die Positionen der erkannten Patterns speichert und ausgibt. Eine genauere Erläuterung dieser Änderungen befindet sich im folgenden Kapitel.

### Kombination der Module

Nachdem die drei grundlegenden Elemente Screenshots, Input Sending und Pattern Matching als eigenständige Klassen fertiggestellt wurden, ist die nächste Aufgabe, eine Funktion zu erstellen, welche die drei Klassen kombiniert und hintereinander aufruft, um aus einem Screenshot einen Input zu erstellen, welcher dann im richtigen Moment an das Spiel geschickt wird.

Hierbei traten mehrere Probleme auf, welche vorher behoben werden mussten:

Da für das Aufnehmen des Screenshots und das Analysieren/Verarbeiten des Screenshots zwei unterschiedliche Funktionen genutzt werden, musste eine Möglichkeit gefunden werden, wie das Bild, welches durch die ImageGrab.grab() Funktion aufgenommen wurde, mit OpenCV analysiert werden konnte.

Die erste Idee war es, das Bild in der ersten Funktion als .png-Datei abzuspeichern, und in der nächsten Funktion erneut auszulesen.

path = r'assets\current\_screenshot.png'  
img.save(path)

img\_rgb = cv.imread(r'assets\current\_screenshot.png')  
**assert** img\_rgb **is** **not** None, "file could not be read"

Das Ganze dauerte laut Benchmarks im Durchschnitt 0.10 Sekunden, was isoliert betrachtet nicht unfassbar viel, aber dennoch optimierbar ist.

Um den Prozess zu beschleunigen, wurde das Bild nach dem Aufnehmen so konvertiert, dass es von OpenCV direkt verarbeitet werden konnte. So konnte das Bild als Ergebnis der Screenshot-Funktion ausgegeben werden, und so direkt an die Analysefunktion als Übergabeparameter übergeben werden.

win32gui.SetForegroundWindow(hwnd)  
img = ImageGrab.grab(bbox)  
size = img.size   
img = img.crop((110, 35, 400, 530))  
  
# convert ImageGrab image to OpenCV image  
opencvImg = img.convert('RGB')  
opencvImg = numpy.array(opencvImg)  
opencvImg = opencvImg[:, :, ::-1].copy()  
**return** opencvImg

image = Screenshot.screenshot(hwnd, bbox)  
RecognizePattern.analyzeImage(image)

Diese Methode dauerte im Durchschnitt 0.05 Sekunden, was die Geschwindigkeit verdoppelte.

Das nächste Problem entstand aus der Kombination der Analyse und dem Senden der Tastenbefehle. Bei dem momentanen Stand hat man nach der Analyse nur ein Bild mit Rechtecken um die erkannten Stellen im Bild, womit man an sich nichts anfangen kann. Man benötigt die Position der Rechtecke um das richtige Timing für den Tastenanschlag zu bestimmen. Die Position kann relativ einfach bestimmt werden, da die erkannten Punkte Pixelwerte haben, welche wie im vorherigen Kapitel beschrieben, nach der Erkennung in ein Array gespeichert werden.

loc = np.where(res >= threshold)  
arrowArray = []  
**for** pt **in** zip(\*loc[::-1]):  
 # draw rectangle  
 cv.rectangle(img, pt, (pt[0] + w, pt[1] + h), (0,0,255), 2)  
 arrowArray.append(pt)

Dieses Array kann nun in einer weiteren Methode der SendInput-Klasse ausgelesen und interpretiert werden, indem die Y-Werte der einzelnen Locations betrachtet werden. Sobald der Y-Wert einen gewissen Wert unterschreitet, da die Pfeile sich von unten nach oben bewegen, und sich der Y-Wert dementsprechend verringert, wird der Keypress übermittelt.

**def** **interpretLocations**(self, locations):  
 # get the next arrow  
 topArrow = locations[0]  
 **if** (topArrow[1] <= 130):  
 # add delay to account for the distance the arrow has yet to travel  
 time.sleep(0.3)  
 self.sendKey('d')

Problematisch ist diese Lösung auf mehrere Arten und Weisen. Zuerst einmal das offensichtliche Problem, dass diese Lösung mit einem festen Delay nicht flexibel ist. Sollte sich die Geschwindigkeit der Pfeile ändern, ist diese Option nicht mehr möglich.

Ebenso gibt es Probleme bei der Erkennung. Da die Pattern Recognition nicht gut genug ist, die statische Reihe der Pfeile, welche als Richtlinie zum Timing dienen, von den sich bewegenden Pfeilen zu differenzieren, kann es vorkommen, dass der statische Pfeil fälschlich interpretiert wird und so durchgängig Inputbefehle gesendet werden. Ebenso werden Pfeile, welche mit diesen statischen Pfeilen überlappen, auch nicht erkannt. Eine Lösungsidee wäre das Cropping des Screenshots so zu ändern, dass die statische Pfeilreihe nicht auf dem Bild zu sehen ist. Dies verkompliziert das Timing der Pfeile jedoch auf lange Sicht gesehen, weshalb diese Lösung suboptimal ist.

### Aufsetzen eines Environments

Nachdem es beim Kombinieren der einzelnen Module mehrere Probleme gab, wurde die Orientierung des Projektes neu definiert. Hierzu wurde der Fokus nicht auf die einzelnen Bestandteile, sondern auf das fertige Endresultat gerichtet - das Environment. Wie in Kapitel 2.3. bereits behandelt, bietet das Environment die Umgebung für den Agent - wie er in jedem Step agieren soll, was er an Feedback bekommt, etc. Das nächste Ziel war also, die in den vorherigen Kapiteln besprochenen Klassen und ihre Funktionen so einzusetzen, dass ein Agent in diesem Environment genutzt werden kann. Hier wurde auch ersichtlich, dass einige Ansätze in den Modulen nicht sinnvoll oder nützlich waren, und ersetzt oder umgeschrieben werden mussten. In diesem Kapitel sollen also die Entwicklung des Environments und die Änderungen, die es mit sich gebracht hat, beschrieben werden.

Ein Environment besteht typischerweise aus mehreren Methoden.

* init() - Die Initialisierung des Environments.
* step() - Die Methode, welche wiederholt aufgerufen wird. Hier wird die Aktion des vorherigen Steps ausgeführt, eine neue Observation aufgenommen, der Reward des Steps bestimmt, und geprüft, ob die derzeitige Episode vorbei ist.
* reset() - Die Methode, welche genutzt wird, um das Environment auf den Anfangszustand zurückzusetzen.
* close() - Die Methode, die das Environment schließt, und beanspruchte Ressourcen wieder freigibt.
* render() - Die Methode, welche das Geschehen visualisiert.

Bei diesem Anwendungsfall ist sowohl render(), als auch close() nicht von Bedeutung, da wir weder die Aktionen visualisieren müssen, noch Ressourcen verwenden, welche freigegeben werden müssen.

Um die step()-Methode übersichtlicher zu gestalten, wurden noch drei weitere Methoden in die Environment-Klasse hinzugefügt:

* get\_observation() - In dieser Methode wird der Screenshot, welcher in jedem Step gemacht wird, so bearbeitet, dass er nur wichtige Informationen für die Erkennung des Gameplays beinhaltet.
* get\_reward() - Wie der Name bereits sagt, wird in dieser Methode der aktuelle Reward berechnet.
* get\_over() - Diese Methode wird ausgeführt, um festzustellen, ob die Episode vorbei ist.

Der Output der step()-Methode besteht aus vier Dingen - der Observation, dem Reward, einem Boolean, welcher aussagt, ob die Episode vorüber ist, und einem Info-Objekt, welches für die Nutzung des Environments selbst nicht relevant ist. Die drei essentiellen Rückgabewerte eines Steps werden also den eigenen, extra hinzugefügten Methoden berechnet und zurückgegeben. Dies ermöglicht nun das einfache Anpassen der einzelnen Bestandteile, sowie das explizite Aufrufen dieser Methoden außerhalb des Environments für Testzwecke.

Wenn das Environment erstellt wird, wird die init()-Methode ausgeführt. Hier soll sich nur um die Initialisierung des Environments und deren Bestandteile gekümmert werden. Die Ausnahmen diesbezüglich sind das Skalieren und Verschieben des Spielfensters, was nur indirekt mit dem Environment selbst zu tun hat. Grund dafür ist, dass die Position des Fensters ausschlaggebend für die Screenshots bzw. das Zuschneiden dieser ist. Wichtiger sind hier die Attribute, in welchen sich Instanzen der vorher erstellten Klassen für Screenshots, Pattern Recognition und Input Sending befinden. Sie werden als Klassenattribute gespeichert um später auf die Methoden dieser Klassen einfacher zugreifen zu können, da man für das Aufrufen der Methoden sonst immer wieder eine Instanz der Klasse hätte erzeugen müssen.

Als Klassenattribute werden auch die Positionen der benötigten Bereiche, also dem Gameplaybereich, dem Scorebereich, etc., abgespeichert.

Ebenfalls essentiell sind die Attribute, welche den Action Space und den Observation Space beinhalten. Der Action Space sagt aus, was der Agent für Inputmöglichkeiten hat. In diesem Fall enthält der Action Space eine von fünf diskreten Zahlen - 0, 1, 2, 3 oder 4. Was diese Zahlen selbst ausdrücken, wird in der Action Map definiert. 1 bis 4 steht für die verschiedenen Richtungen (oben, unten, links und rechts), während 0 für “no\_op”, also “no operation” steht. Hier wird kein Tastendruck ausgeführt.

Im Observation Space wird definiert, was der Agent als Input, also Observation, erhält. Für dieses Environment werden, wie bereits angesprochen, Screenshots von der Gameplay-Region des Spiels gemacht, welche der Agent als Observation nutzen soll. Dafür wird als Observation Space eine Box definiert - ein Raum, in welchem sich mehrere Zahlen zwischen bestimmen Wertintervallen befinden. Hier wird festgelegt, dass sich alle Werte zwischen 0 und 255 befinden, da wir in unserem Fall von der Helligkeit von Pixeln reden, sowie einer Shape von (1, 135, 100). Die Shape steht für die Dimensionen des Bildes, welches wir als Observation nutzen - eine Breite und Höhe von 100 mal 135 Pixeln in nur einem Farbkanal, da die Screenshots in Schwarz-Weiß konvertiert werden, um Daten zu sparen.

**def** **\_\_init\_\_**(self):   
 super().\_\_init\_\_()  
 # Defined action space  
 self.action\_space = Discrete(5)  
  
 # Observation Array  
 self.observation\_space = Box(  
 low=0,   
 high=255,   
 shape=(1, 135, 100),  
 dtype=np.uint8  
 )  
   
 # Define extraction parameters for the game  
 self.screenshot\_helper = Screenshot()  
 self.input\_sending\_helper = SendInput()  
 self.pattern\_recog\_helper = RecognizePattern()  
 self.capture = mss()  
 self.steps = 0  
 self.previous\_reward = 0  
 self.window\_location = {'top': 35, 'left': 10, 'width': 410, 'height': 230}  
 self.game\_location = {'top': 15, 'left': 20, 'width': 100, 'height': 185}  
 self.score\_location = {'top': 215, 'left': 280, 'width': 100, 'height': 25}  
 self.done\_location = {'top': 0, 'left': 0, 'width': 90, 'height':25}  
 self.past\_arrows\_location = {'top': 45, 'left': 50, 'width': 140, 'height': 2}  
 self.cur\_held\_buttons = {'a': False, 's': False, 'w': False, 'd': False}  
 self.action\_map = {  
 0:'no\_op',  
 1:'a',  
 2:'s',  
 3:'w',  
 4:'d',  
 }  
   
 # Adjust window position and size  
 win = pygetwindow.getWindowsWithTitle('StepMania')[1]  
 win.size = (450, 290)  
 win.moveTo(0, 0)

Nachdem das Environment initialisiert wurde, kann es genutzt werden. Dabei ist der Ablauf einfach erklärt - Gestartet wird mit einem Aufruf der Reset-Methode, welche das Environment auf seinen Startzustand setzt. Die Reset-Methode gibt eine Observation des Startzustands zurück, welche dann vom Model genutzt wird, um eine Aktion zu bestimmen, die es basierend auf der Obseration für die Bestmögliche hält. Mit dieser Aktion wird die Step-Methode aufgerufen, welche die Aktion ausführt, und den Reward für diese Aktion, sowie eine neue Observation, an das Model zurückgibt. Ebenfalls wird mitgegeben, ob die Abbruchbedingung des Environments erfüllt ist. Der Reward wird genutzt um die Qualität der vergangenen Aktion zu evaluieren und sich selbst demensprechend anzupassen, während mit der Observation eine neue Aktion bestimmt wird, mit welcher die Step-Methode erneut aufgerufen wird, sollte die Abbruchbedingung nicht erfüllt sein. Dieser Kreislauf wird so lange ausgeführt, bis eine vorher definierte Anzahl an Steps erreicht wurde, oder bis die Abbruchbedingung erfüllt ist.

[BILD DES PROZESSES EINFÜGEN]

Die Step-Methode führt zuerst die Aktion aus, welche sie als Parameter bekommen hat. Dafür wird das Input-Sending-Modul genutzt. Daraufhin wird mit dem Screenshot-Modul ein Screenshot des gesamten Fensters gemacht, und, ebenfalls mit dem Modul, runterskaliert, um die Größe des Bildes für die Observation zu reduzieren. Da für die Erkennung des Ergebnisscreens und die Score-Erkennung Template-Matching genutzt wird, kann hier nicht die runterskalierte Version der Observation genutzt werden. Also werden diese beide Methoden mit dem originalen Screenshot aufgerufen, während für die Methode, die für das Gameplay verantwortlich ist, die runterskalierte Version genutzt wird. Wurden alle drei Methoden aufgerufen, werden ihre Ergebnisse zusammen mit einem Info-Objekt als Output der Step-Methode zurückgegeben.

**def** **step**(self, action):  
 # Manage and send input based on action parameter  
 **if** action != 0:  
 **if** (list(self.cur\_held\_buttons.values())[action - 1]):  
 self.cur\_held\_buttons[list(self.cur\_held\_buttons)[action - 1]] = False  
 self.input\_sending\_helper.releaseKey(self.action\_map[action])

**else**:  
 self.cur\_held\_buttons[list(self.cur\_held\_buttons)[action - 1]] = True  
 self.input\_sending\_helper.holdKey(self.action\_map[action])   
   
 # Take screenshot for done, observation and reward functions  
 screenshot = np.array(self.capture.grab(self.window\_location))[:,:,:-1].astype(np.uint8)  
 downscaled\_screenshot = self.screenshot\_helper.downscaleImage(screenshot, (225, 150), (1, 150, 225))  
 self.steps += 1  
   
 # Checking if the game is over  
 done = self.get\_over(screenshot)  
   
 # Get the next observation  
 new\_observation = self.get\_observation(downscaled\_screenshot)  
   
 # Use score as reward  
 reward = self.get\_reward(screenshot, action)  
 info = {}  
  
 **return** new\_observation, reward, done, info

Nach jedem abgeschlossenen Song wird die Reset-Methode ausgeführt. Diese enthält eine Reihe an Inputs, welche das Spiel vom Ergebnisscreen zuerst zurück in das Hauptmenü, und danach in den nachfolgenden Song navigieren. Abschließend wird ein Screenshot gemacht, welcher an das Model weitergegeben wird.

**def** **reset**(self):  
 # Exit to menu, select new song and start  
 time.sleep(5)  
 pydirectinput.press('enter')  
 time.sleep(6)  
 pydirectinput.press('d')  
 time.sleep(2)  
 pydirectinput.press('enter')  
  
 # Edge Case - 'Roulette' is selected  
 time.sleep(1.5)  
 pydirectinput.press('enter')  
 time.sleep(3)  
 pydirectinput.press('enter')  
  
 # Reset variables  
 self.previous\_reward = 0  
 self.steps = 0  
 self.cur\_held\_buttons = {'a': False, 's': False, 'w': False, 'd': False}  
  
 # Take screenshot to pass to observation  
 screenshot = np.array(self.capture.grab(self.window\_location))[:,:,:-1].astype(np.uint8)  
 downscaled\_screenshot = self.screenshot\_helper.downscaleImage(screenshot, (225, 150), (1, 150, 225))  
   
 **return** self.get\_observation(downscaled\_screenshot)

Die oben angesprochenen Methoden get\_observation(), get\_done() und get\_reward() sind prinzipiell sehr ähnlich. Sie nehmen den Screenshot, welchen sie beim Aufruf der jeweiligen Methode bekommen, und schneiden ihn auf den jeweiligen Teil zu (in der Reward-Methode beispielsweise also auf den Teil welcher die Scoreanzeige enthält). Die get\_observation-Methode gibt nur ihren Teil, welcher das Spielgeschehen zeigt, zurück, während die get\_done()- und die get\_reward()-Methoden unter anderem das Template Matching Modul nutzen, um ihre jeweiligen Informationen verarbeiten zu können. Während bei der get\_done-Methode nur ein Wahrheitswert zurückgegeben wird, muss bei der Reward-Methode der Score weiter interpretiert werden um nützliches Feedback geben zu können. Wenn in den nachfolgenden Kapiteln von der Methode gesprochen wird, welche für die Rewardverteilung zuständig ist, ist damit diese Methode gemeint. Ihre Funktionsweise hat sich im Laufe des Projektes mit am meisten verändert, worauf später noch eingegangen wird.

**def** **get\_observation**(self, img):  
 # Crop gameplay part of the window screenshot  
 obs = img[:, self.game\_location['top']:(self.game\_location['top'] + self.game\_location['height']), self.game\_location['left']:(self.game\_location['left'] + self.game\_location['width'])]  
  
 **return** obs

**def** **get\_over**(self, img):  
 # Crop done part of the window screenshot  
 obs = img[self.done\_location['top']:(self.done\_location['top'] + self.done\_location['height']), self.done\_location['left']:(self.done\_location['left'] + self.done\_location['width'])]  
   
 **return** self.pattern\_recog\_helper.analyze\_results(obs)

**def** **get\_reward**(self, img):  
 # Crop score part of the window screenshot  
 obs = img[self.score\_location['top']:(self.score\_location['top'] + self.score\_location['height']), self.score\_location['left']:(self.score\_location['left'] + self.score\_location['width'])]  
   
 # Calculate the reward by subtracting the previous score from the current score  
 # If the song is over, or the score isn't recognized properly, set reward to 0 to avoid accidently returning a negative reward  
 new\_reward = self.pattern\_recog\_helper.analyze\_score(obs)  
 **if** (new\_reward > self.previous\_reward):  
 reward = new\_reward - self.previous\_reward  
 # Set the current reward as the previous reward for the next iteration  
 self.previous\_reward = new\_reward  
 **else**:  
 reward = 0  
 **return** reward

### Jupyter Notebook

Da das Programmieren von Environments häufiges Ausführen von bestimmten Codeteilen erfordert, ist eine beliebte Herangehensweise das Nutzen eines sogenannten Jupyter Notebooks. Diese basieren auf Project Jupyter und bieten eine webbasierte Grundlage zum Programmieren von Projekten z.B. in den Bereichen Data Science und Machine Learning. Sie ermöglichen es, Code, Dokumentation und Output in einer Datei zu vereinen und sparen so Zeit. [Quelle hinzufügen]

In diesem Fall wurde sich für ein Jupyter Notebook entschieden, da das selektive Ausführen von Code, als auch das direkte Darstellen der aufgenommenen Bereiche und das Erstellen von Graphen mit Hilfe von matplotlib den Entwicklungsprozess erleichtert hat.

Jupyter Notebooks sind in sogenannte Code- oder Markdown-Cells aufgeteilt, in welchen man entweder Code oder Text im Markdown-Format unterbringen kann. Im folgenden Beispiel kann man zwei Code-Cells sehen - die Erste enthält mehrere Import-Aufrufe, welche nur einmal ausgeführt werden müssen um die Module zu importieren. Die Zweite enthält einen Funktionsaufruf, welcher während des Testens des Environments mehrfach aufgerufen werden musste. Da man jede der Code-Cells einzeln ausführen kann und Imports und Variablen über die Laufzeit gespeichert werden, mussten bei jedem Funktionsaufruf nicht alle Import-Aufrufe ausgeführt werden, was Zeit und Rechenkraft spart. Dieses Beispiel zeigt nur einen Anwendungsfall, welcher des Beispiels wegen stark runtergebrochen wurde.

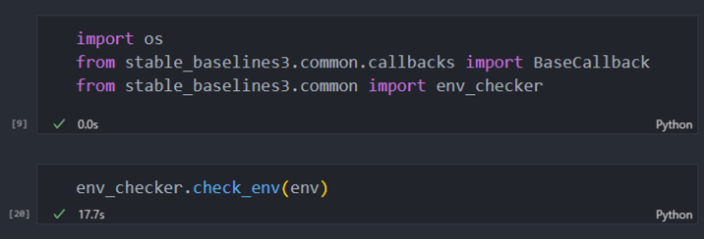


Abb. - Zwei Code-Cells eines Jupyter Notebooks innerhalb der Entwicklungsumgebung Visual Studio Code

Während die Entwicklung des Environments durch die Jupyter Notebooks stark vereinfacht wird, gibt es dennoch Grenzen und Nachteile.

Die Entwicklung wurde insofern eingeschränkt, dass bei jeder Änderung von externen Dateien oder Modulen, der Kernel des Notebooks neu gestartet werden muss. Jedes Notebook läuft auf einem Python Kernel, welcher einem die Möglichkeit gibt, Variablen auch über längere Zeit und außerhalb der Laufzeit aufzubewahren. Dies hat allerdings auch zur Folge, dass alte Versionen von externen Modulen noch verwendet werden, wenn der Kernel nicht neu initiiert wird. Dies ist etwas umständlich, aber nicht vermeidbar.

Ebenfalls musste das Environment zum Dokumentieren der Testläufe in einen sogenannten Monitor-Wrapper untergebracht werden. Um dies zu erreichen, musste das Environment als Klasse in einer eigenen Python-Datei in einen Unterordner des Gym-Packages verschoben werden. Dies bedeutet, dass Änderungen am Environment innerhalb der Notebook-Datei extra in diese Python-Datei kopiert werden mussten, damit das Environment im Wrapper aktuell bleibt. Hätte man eine eigene Python-Datei genutzt, wäre dieser Schritt einfacher.

### Custom Theming

Das größte Problem mit der Pattern Recognition war die Flut an Informationen auf dem Bildschirm. Diese können für den Spieler zwar wichtig sein, für die KI sind sie jedoch unwichtig und verhindern ein klares Erkennen der Pfeile. Auf dem folgenden Bild sieht man, welche Elemente auf dem Spielfeld zu finden sind, und wie die Pattern Recognition damit umgeht.

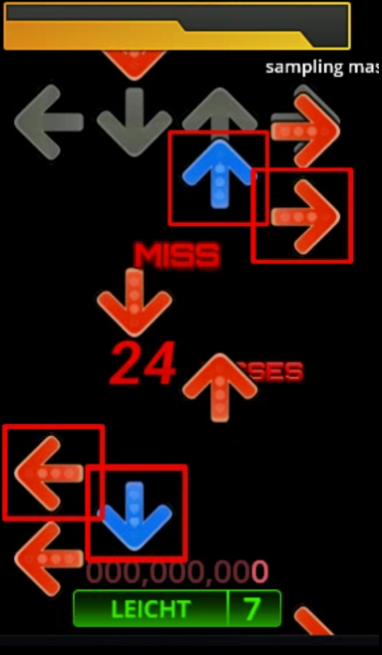


Abb. - Screenshots des Gameplays, bei welchem die erkannten Elemente markiert wurden

Wie man sieht, kann das Programm die beiden Pfeile in der Mitte des Bildes nicht als solche erkennen, da sich dort das sogenannte “Judgement” befindet - ein Feedback für den Spieler, ob und wie gut er die Note getroffen hat. Durch den Text und die Zahl können die Pfeile nicht gut erkannt werden.

Ähnlich sieht es auch mit dem nichterkannten Pfeil in der unteren linken Ecke aus. Er wird nicht erkannt, da er sich neben der Score-Anzeige befindet, welche die Punkte des Spielers zählt. Diese Anzeige ist zwar relevant, da sie für das Feedback des Reinforcement Learning Algorithmus gebraucht wird, jedoch ist sie an dieser Position hinderlich.

Dazu kommen noch weitere Elemente, welche Teile des Spielfeldes verdecken - die Anzeige am oberen Rand, welche zeigt, wie viele Lebenspunkte der Spieler noch hat, oder die Schwierigkeitsanzeige des Levels am unteren Rand.

Um diese Probleme zu umgehen wurden mehrere Möglichkeiten ausprobiert, wobei eine schnell in eine Sackgasse führte. Die erste Idee war es, in den Einstellungen alles auf dem Bildschirm zu deaktivieren, was deaktivierbar ist. Zusätzlich konnte man die Modifikation “Blind” im Spiel auswählen, was in dem momentan gespielten Song die Score-Anzeige und das Judgement versteckt. Das Problem hierbei ist, dass die Score-Anzeige später noch gebraucht wird, weshalb dieser Ansatz schnell ausgeschlossen wurde.

Die zweite und finale Lösung war ein sogenanntes Custom Theme. StepMania 5 unterstützt spielererstellte Veränderungen der Benutzeroberfläche. Verändern kann man diese durch das Hinzufügen und Abändern von .ini und .lua-Dateien, wobei in diesem Anwendungsfall ein paar Zeilen in der metrics.ini-Datei, die Datei, die für den Großteil von Themeänderungen zuständig ist, sowie eine Änderung von zwei Variablen in einer .lua-Datei den gewünschten Effekt erzeugte.

In der metrics.ini-Datei wurde die Score-Anzeige um einen gewissen Pixelwert nach rechts verschoben, sodass sie nicht mehr im Bereich des Gameplays liegt. Ebenfalls wurden dort alle möglichen Overlay-Elemente deaktiviert, welche durch Änderungen dieser .ini-Datei aus- oder angeschaltet werden konnten.

Eine Veränderung der .lua-Dateien wird vom Entwickler nicht empfohlen, jedoch konnte das bereits angesprochene Judgement anders nicht verändert werden. [Quelle hinzufügen] Da ein generelles Ausblenden nicht möglich ist, wurde die Anzeige um einen hohen Wert außerhalb des dargestellten Bereichs geschoben.

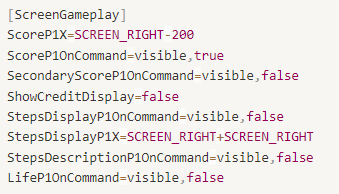


Abb. - metrics.ini - Hier wurde alles auf unsichtbar gestellt, was nicht benötigt wird. Ebenfalls wurde die Score-Anzeige an den rechten Rand des Fensters gesetzt und mit einem Offset in die Mitte des leeren Bereiches rechts neben dem Spielfeld geschoben

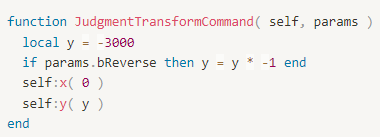


Abb. - Other.lua - Der Y-Wert wurde so verändert, dass sich das Judgement außerhalb des dargestellten Bereichs befindet

Hier sieht man das finale Overlay nach Anwendung der Modifikationen durch das eigenerstellte Theme. Die einzigen Elemente auf dem Bildschirm sind der Gameplay-Bereich, sowie die Score-Anzeige, welche sich isoliert unten auf der rechten Seite befindet.

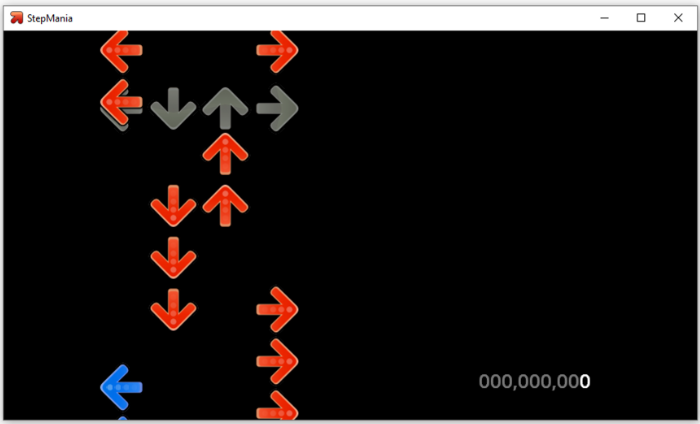


Abb. - User Interface nach den Änderungen am Theme

### Tesseract

Wie in Kapitel 4.2.5. bereits angesprochen, werden für ein zuverlässiges StepMania-Environment Lösungen für zwei Probleme benötigt - Man braucht ein verlässliches Reward-System, welches den vorangegangenen Input bewertet, und man benötigt eine Funktion, welche erkennt, ob ein Song vorüber ist, welche dann den nächsten Song auswählt um das Spielen fortzuführen.

In der ersten Version des Environments wurde dafür Tesseract genutzt. Tesseract ist ein Open-Source-Programm, welches Optical Character Recognition, auch OCR genannt, nutzt, um Text auf Bildern zu erkennen. Tesseract konnte mit Hilfe dieser OCR-Funktionalität genutzt werden, um verschiedene Teile des Bildschirms für die oben genannten Zwecke zu interpretieren.

Für den ersten Anwendungsfall, also das Reward-System, wurde eine Funktion geschrieben, welche ein Bild als Input nimmt, und ausschließlich auf Zahlen überprüft. Als Input wurde der Teil des Bildschirms genutzt, welcher die derzeitige Punktzahl anzeigt. So wurde die gezeigte Zahl in einen verwertbaren Integer umgewandelt, und konnte so als Reward für das Netz genutzt werden. Wurde nichts erkannt, beispielsweise, weil beim Start eines Songs keine Punktzahl zu sehen ist, wird 0 zurückgegeben.

**def** **get\_number\_from\_image**(self, img):  
 analysed\_string = pytesseract.image\_to\_string(img, config='-c tessedit\_char\_whitelist=0123456789')  
 **if** analysed\_string == '':  
 analysed\_string = 0  
 **return** int(analysed\_string)



Abb. - Erkannte Zahl als Output (oben) mit dem Screenshot der Zahl als Input (unten)

Der zweite Anwendungsfall wird ähnlich gelöst. Hier muss erkannt werden, ob sich das Spiel derzeit auf dem Ergebnisscreen befindet, welcher angezeigt wird, wenn ein Song geendet hat. Um dies zu erkennen, wurde die gleiche image\_to\_string()-Funktion von Tesseract genutzt, jedoch in einem anderen Kontext.

In diesem Bild sieht man den Ergebnisscreen eines Songs.

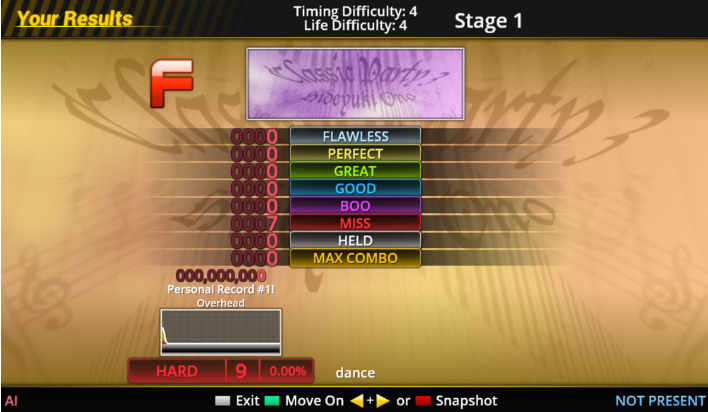


Abb. - Ergebnisscreen eines Charts

Der Teil, auf den sich hier konzentriert wird, ist die obere linke Ecke, in welcher “Your Results” steht. Tesseract hat hier diesen kleinen Ausschnitt des Bildes an Input bekommen, und sollte nun die Wörter “Your Results” erkennen. Wenn dies der Fall ist, wird eine Reihe von Inputs ausgeführt, welche den Screen verlassen und den nächsten Song auswählen.

Für die Erkennung wurde eine Funktion geschrieben, welche als Input sowohl das Bild, als auch ein Array der zu erkennenden Zeichenfolgen und die Maximallänge der zu analysierenden Zeichen enthält.

Ein Array wurde gewählt, da Tesseract vor allem bei niedriger Auflösung manche Buchstaben als andere identifiziert. “Your Results” wurde bei geringerer Auflösung beispielsweise als “Yow Results” oder “Ywr Results” erkannt. Ist dies der Fall, können mithilfe des Arrays häufig auftretende Alternativen einbezogen werden.

Die Maximallänge der Zeichen ist für diesen Anwendungsfall relevant, da Tesseract auch Leerzeichen und Zeilenumbrüche erkennt und zusammen mit dem Text ausgibt. Um ein Umformatieren des Strings zu umgehen, wurde dieser Parameter eingeführt. Diesen Parameter durch die Anzahl der Zeichen im Array selbst zu bestimmen ist in dem Fall nicht möglich gewesen, da die Einträge im Array der Zeichenfolgen unterschiedlich lang sein könnten.

Als Output wird ein Boolean zurückgegeben, welcher aussagt, ob das mitgegebene Bild eines der zu erkennenden Zeichenfolgen enthält.

**def** **is\_text\_in\_image**(self, img, text, chars):  
 done = False  
 res = pytesseract.image\_to\_string(img, lang='eng', config='--psm 7')[:chars]  
 **if** res **in** text:  
 done = True  
 **return** done

Im folgenden Bild sieht man das Ergebnis des Aufrufs:

*text\_recog\_helper.is\_text\_in\_image(obs, ['Your Results'], 12)*.

*obs* bezeichnet hier die Observation, welche ebenfalls im Bild zu sehen ist.

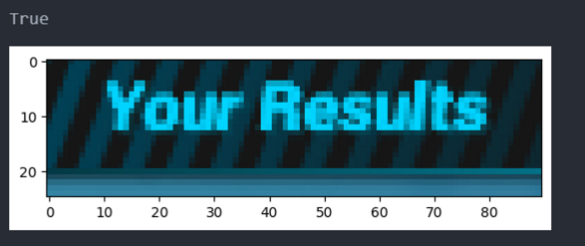


Abb. - Ergebnis des genannten Funktionsaufrufs (oben) und die dazu gehörende Observation (unten)

Während das Nutzen von Tesseract sehr unkompliziert und einfach ist, hat es einen für diesen Nutzungskontext nicht unerheblichen Nachteil - die Performance. Der Aufruf der Ergebnisscreen-Analyse dauerte abgerundet etwa 0.17 Sekunden. Dies klingt nach wenig, jedoch ergibt sich dadurch, dass die Ergebnisscreen-Analyse und die Reward-Analyse beide in jedem Step ausgeführt werden, eine Zeit pro Step von etwa 0.4 Sekunden. Der Großteil der Zeit wird von den beiden Tesseract-Aufrufen eingenommen, da diese auf ein zusätzliches, großes Programm zugreifen müssen.

Um dieses Problem zu lösen, wurde ein Template-Matching-Ansatz eingeführt. Hier soll, ähnlich zu dem Template Matching beim Gameplay, eine Referenz mit dem Screenshot verglichen werden, um zu bestimmen, ob z.B. der Ergebnisscreen erreicht wurde.

Um die Ergebnisscreen-Erkennung zu verbessern, wurde ein Template des “Your Results”-Teils des Ergebnisscreens erstellt. Dieses Template wurde dann in jedem Step mit der derzeitigen Observation verglichen, um festzustellen, ob es sich auf dem Bildschirm befindet.

**def** **analyze\_results**(self, img):  
 img\_gray = cv.cvtColor(img, cv.COLOR\_BGR2GRAY)  
 threshold = 0.9  
 res = cv.matchTemplate(img\_gray, self.template\_results, cv.TM\_CCOEFF\_NORMED)  
 **if** (res >= threshold).any():  
 **return** True  
 **return** False

Diese Änderung führte dazu, dass sich die durchschnittlichen Bilder pro Sekunde von 2,35 auf 3,73 verbessert haben.

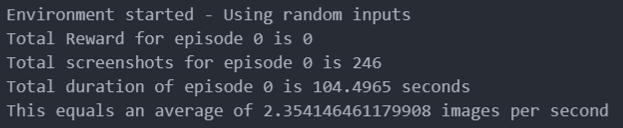


Abb. - Ergebnis des Benchmarktests mit Tesseract

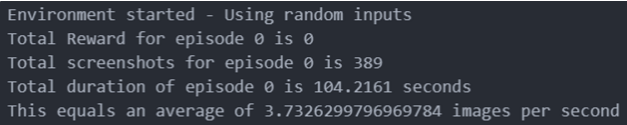


Abb. - Ergebnis des Benchmarktests mit Template Matching

Das gleiche Prinzip wurde nun für das Reward-System angewendet. Hier wurden Screenshots von jeder Zahl als Template geladen, und in einer Funktion mit dem Screenshot der Punktezahl verglichen. Der Teil, welcher diese Funktion von der oberen unterscheidet, ist, dass die erkannten Zahlen in der richtigen Reihenfolge angeordnet werden müssen.

Wird zum Beispiel die Zahl “4732” erkannt, werden alle 10 Templates nacheinander, beginnend mit 0, verglichen. Gibt es eine Übereinstimmung, wird die X-Koordinate des Bildausschnittes, wo die Übereinstimmung stattgefunden hat, zusammen mit der Zahl in ein Array hinzugefügt. Dabei gibt der erste Wert die X-Koordinate an, und der zweite Wert die Zahl, welche erkannt wurde.

Zu diesem Zeitpunkt würde das Array für das Beispiel so aussehen:

[[40, 2], [30, 3], [10, 4], [20, 7]]

Dieses Array würde jetzt der Zahl “2347” entsprechen, weshalb es nun in aufsteigender Reihenfolge nach der X-Koordinate sortiert wird, sodass das Array nicht mehr nach den erkannten Zahlen, sondern der erkannten Position von links nach rechts geordnet ist:

[[10, 4], [20, 7], [30, 3], [40, 2]]

Anschließend werden die Zahlen, also die zweiten Werte in jedem Objekt aus dieser sortierten Reihenfolge ausgelesen und zusammengeführt, sodass das finale Ergebnis “4732” ist.

**def** **analyze\_score**(self, img):  
 threshold = 0.82  
 img\_gray = cv.cvtColor(img, cv.COLOR\_BGR2GRAY)  
 found\_array = []  
 index\_template = 0  
 **for** index\_template **in** range(10):  
 res = cv.matchTemplate(  
 img\_gray,   
 self.template\_numbers[index\_template],   
 cv.TM\_CCOEFF\_NORMED  
 )  
 loc = np.where(res >= threshold)  
 **for** pt **in** zip(\*loc[::-1]):  
 found\_array.append([pt[0], index\_template])  
   
 sorted\_array = sorted(found\_array, key=**lambda** x: x[0])  
 result = ""  
 **for** array **in** sorted\_array:  
 result += str(array[1])  
 **if** result == "":  
 result = "0"  
 **return**(int(result))

Die optimierte Funktionsweise des Reward-Systems ist zwar um einiges schneller, hat allerdings auch seine Nachteile. Durch die niedrige Auflösung der Screenshots kann es vereinzelt vorkommen, dass Zahlen nicht richtig erkannt werden. Dies kann im schlimmsten Fall zu einer falschen Rewardverteilung führen, was den Lernerfolg des Agents einschränken würde.

Durch das Anpassen des Thresholds, als auch der Auflösung des Spiels, konnte man jedoch erreichen, dass nach 10 Tests mit je 5 Episoden alle Zahlen richtig erkannt wurden. Dieses Verfahren ersetzte danach die Tesseract-Erkennung, und sorgte für eine weitere Performancesteigerung von 3.73 auf 4.59 Bilder pro Sekunde.

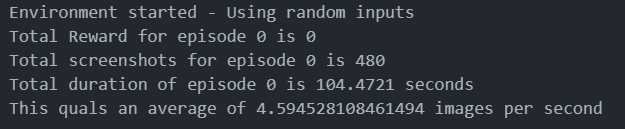


Abb. - Ergebnis des Benchmarktests nach beiden Änderungen

### Erster Lernversuch

Nachdem das Environment und der Agent aufgesetzt und die Performance optimiert wurde, wurde der erste Lernversuch unternommen. Ziel war, diese Fortschritte in mehreren Hinsichten zu analysieren. Zuerst war es wichtig zu wissen, ob es generell Fortschritt gibt. Während des Austestens des Modells war noch kein Wissen vorhanden, weshalb jeder Input zufällig bestimmt, und nur manchmal tatsächlich brauchbare Tastendrücke ausgegeben wurden. Falls nach dieser, für diese Aufgabe doch recht kurze, Zeitspanne bereits kleine Veränderungen zu sehen waren, heißt dies, dass nun nur weiter trainiert werden muss, um das Endziel zu erreichen. Ebenfalls wurde diese Trainingseinheit dafür genutzt, um zu schauen, wie sich die Hyperparameter über einen längeren Zeitraum entwickeln, und wie man die Ergebnisse aus dem Trainingsdurchlauf nutzen kann, um diese Parameter effizienter zu gestalten.

Der Lernversuch selbst umfasste rund zweieinhalb Stunden, was nach der optimierten Performance etwas mehr als 50.000 Steps entsprach. Das Ergebnis fiel jedoch schlechter aus als erwartet.

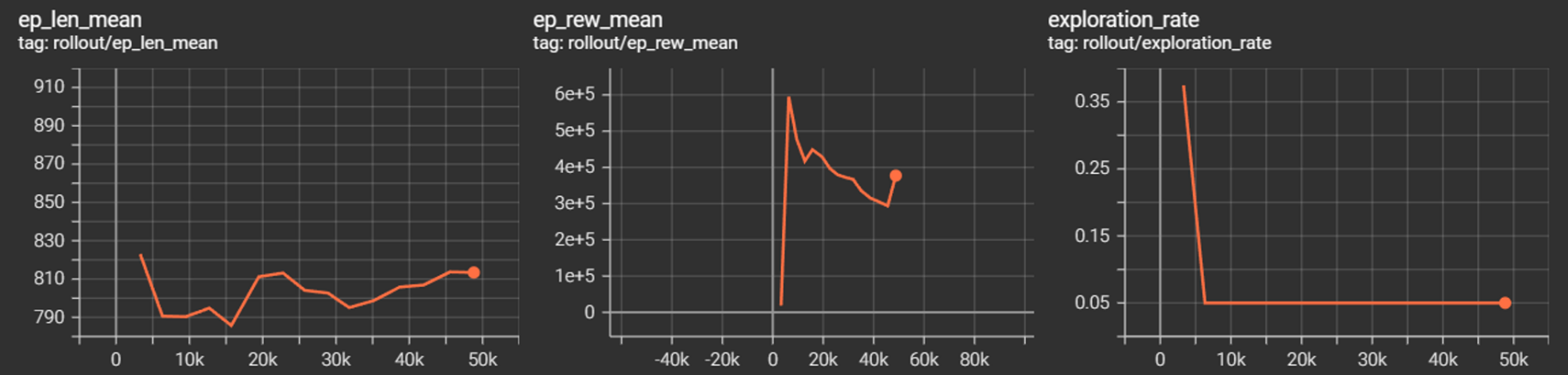


Abb. - Screenshot von Tensorboard - Darstellung der Mittelwerte der Episodenlänge, des Rewards, sowie der Wert der Exploration Rate des ersten Lernversuchs über die Lerndauer

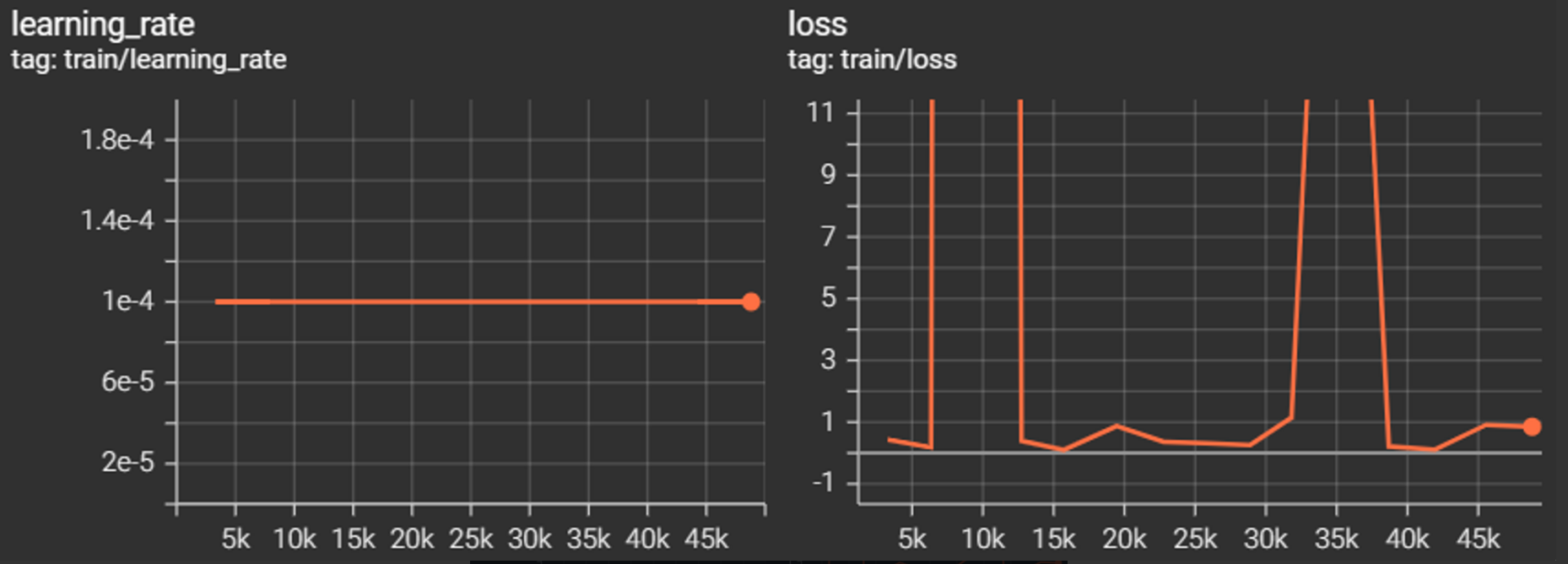


Abb. - Screenshot von Tensorboard - Darstellung der Learning Rate und der Loss Rate des ersten Lernversuchs über die Lerndauer

Auf den Grafiken, welche mit Hilfe von Tensorboard erstellt wurden, konnte man sehen, dass der Mittelwert der Rewards tendenziell über die Zeit sank. Dies bedeutet im Kontext der Anwendung, dass mit längerem Lernen, weniger Punkte erzielt worden sind. Gleichzeitig sieht man, dass die Exploration Rate nach 6.000 Steps bei 0.05 blieb, die Learning Rate sich konstant bei 0.005 befand, und die Loss-Funktion zwar niedrig war, sich vom Wert her jedoch wenig veränderte, bis auf zwei große Ausreißer. Auf die ersten beiden Punkte komme ich gleich zurück, zuerst möchte ich jedoch zeigen, wie das trainierte Model nach 50.000 Steps das Spiel spielte.

An dieser Tabelle kann man sehen, welche Tasten das Model wie oft nach 10 Minuten Spielzeit gedrückt hat.

|  |  |
| --- | --- |
| **Taste** | **Anzahl der Inputs** |
| Oben | 5921 |
| Rechts | 0 |
| Unten | 0 |
| Links | 3 |

Man muss hier nicht weiter erklären, dass es eindeutige Probleme in dem Lernverhalten der KI gibt, da der Fokus auf eine Taste nicht das Ziel dieses Vorhabens ist. Die KI hat hier gelernt, dass es stetig eine gewisse Anzahl an positivem Feedback erhält, wenn eine Taste durchgängig gedrückt, direkt wieder losgelassen, und erneut gedrückt wird. Dies liegt daran, dass der Reward äquivalent zur Punktezahl ist. Das erste Mal, wo eine Punktzahl durch rechtzeitiges Drücken der richtigen Taste erreicht wurde, scheint Grund genug gewesen ist, sich auf diese eine Taste zu versteifen, und alle anderen zu ignorieren.

Dies erklärt auch, weshalb der Mittelwert der Reward sich über die Zeit so verhalten hat. Dass der Wert gesunken ist, ist darauf zurückzuführen, dass die gespielten Songs wenig Tastendrücke dieser einen Taste erforderten. Die gespielten Level hatten also abnehmend viele Noten auf der, in diesem Fall, linken Spalte, was dazu führte, dass der Mittelwert des Rewards über Zeit gesunken ist.

Um zu testen, ob dieses Problem tatsächlich bereits nach kurzer Lerndauer auftrat, oder ob es sich stetig darauf hingearbeitet hat, habe ich die gleiche Grafik für zwei neu-trainierte Models nach 5.000, 10.000 und 25.000 Steps angefertigt.

|  |  |  |  |
| --- | --- | --- | --- |
| **Taste** | **Anzahl der Inputs, 5.000 Steps** | **Anzahl der Inputs, 10.000 Steps** | **Anzahl der Inputs, 25.000 Steps** |
| Oben | 4872 | 9629 | 24.091 |
| Rechts | 3 | 0 | 0 |
| Unten | 0 | 0 | 0 |
| Links | 0 | 0 | 0 |

Man kann erkennen, dass dieses Problem also fundamental an der Umsetzung des Reward-Systems liegt. Da die hier genutzten Werte die Standardwerte für DQN-Models unter Stable Baselines 3 sind, sollte man jedoch auch anmerken, dass sich die genutzten Werte für die Exploration Rate und die Learning Rate eventuell nicht für diesen Anwendungsfall eignen. Genauer gesagt kann es sein, dass die Exploration Rate zu niedrig, und die Learning Rate zu hoch, ist, sodass die KI der Phase der Exploration nicht genug Zeit gibt, um sinnvoll zu lernen. Hier wird sich zu sehr auf das bereits gelernte fokussiert, was in diesem Fall bedeutet, dass sobald ein Ansatz für konstanten Reward gefunden wurde, nur dieser Ansatz verfolgt wird, und zu wenige Alternativen ausprobiert werden.

Für den nächsten Lernversuch galt es also, sowohl das Reward-System, als auch die Hyperparameter entsprechend anzupassen, damit die KI nicht dazu verleitet wird, nur eine Taste zu drücken.

### Neues Rewardsystem

Nachdem im letzten Kapitel festgestellt wurde, dass das Rewardsystem, wie es momentan besteht, nicht funktioniert, galt es nun eine Lösung zu finden, wie man dem Modell Feedback gibt, ohne es auf die falsche Bahn zu leiten.

Zuerst muss das Problem genauer untersucht werden. Da das Modell scheinbar ausgelernt hat, und sich damit zufriedengegeben hat, nur eine Taste zu drücken, gab es zu viel positiven Reward für die einzelnen Aktionen. Bisher wurde der Reward so berechnet, dass der Reward des aktuellen Steps mit dem des vorherigen Steps subtrahiert wurde. Ein richtiger Tastendruck konnte so potentiell mehrere Tausend Punkte bedeuten, und somit auch mehrere Tausend Punkte an Reward an das Modell geben. Dieses Prinzip musste also runterskaliert werden um den Reward zu normieren.

Ebenso gibt es im momentanen Environment keine Möglichkeit, dem Modell zu sagen, dass es etwas schlecht gemacht hat, ihm also negativen Reward zu geben. Der Score ist immer gleichbleibend, und es werden keine Punkte abgezogen, wenn eine Note nicht getroffen wird. Auch existiert keine Erkennung, wenn eine Note verpasst wurde. Dies bedeutet, dass man dem Modell für schlechte Entscheidungen kein negatives Feedback geben kann - es gibt nur neutrales oder positives Feedback. Das Environment musste also dementsprechend angepasst werden.

Das erste Problem, das Runterskalieren des Rewardsystems, ist schnell gelöst. Das Grundprinzip bleibt gleich, man vergleicht also immer noch den Score des vorherigen Steps mit dem des aktuellen Steps, gibt nun aber nicht mehr die Differenz wieder. Hat sich der Score verbessert, wird ein Reward von 1 vergeben. Hat er sich nicht verändert, beträgt der Reward 0. Dies bedeutet, dass das Modell für jede getroffene Note belohnt wird; die Genauigkeit wird in dem Fall außer Acht gelassen, da das jetzige Ziel ein generell funktionierendes Modell ist, und kein besonders genaues. Das Modell kann also verbessert werden, wenn mehr Reward für genauere Tastendrücke vergeben werden, dies würde jedoch den Umfang des Projektes überschreiten.

Für das zweite Problem, das Vergeben von negativem Feedback, gibt es zwei Lösungsansätze. Zuerst kann man prüfen, ob Noten nicht getroffen wurden. Wurden Noten nicht getroffen, bewegen sie sich weiter nach oben aus dem sichtbaren Bereich, während Noten die getroffen wurden, verschwinden. Man könnte nun also prüfen, ob sich in dem Bereich des Fensters über dem Hit-Bereich Noten befinden, und dementsprechend Abzug geben. Problematisch ist hier, dass die Aktion, die dafür verantwortlich ist, dass diese Noten nicht getroffen wurden, schon lange vergangen ist. Man würde hier also Reward entweder zurückhalten bis geklärt wurde, ob Noten verfehlt wurden, oder man vergibt negativen Reward in Steps die eventuell korrekt ausgeführt wurden, und kein schlechtes Feedback verdient haben. Beides sind suboptimale Lösungen.

Sinnvoller ist es, zu prüfen, welche Tasten im momentanen Schritt gedrückt werden sollen - man analysiert also, ob sich derzeit eine Note im Hit-Bereich befindet. Man arbeitet hier in der Gegenwart, und nicht in der Vergangenheit, wie es bei den anderen beiden Ansätzen der Fall war. Dies eliminiert das Problem, Reward für bereits vergangene Aktionen geben zu müssen. Der Ablauf wäre nun wie folgt: In jedem Screenshot wird geschaut, ob farbige Pixel in einem bestimmten Pixelbereich vorkommen. Diese farbigen Pixel lassen auf eine Note schließen, welche sich derzeit in dem Hit-Bereich befinden. Dies wird für alle vier Notenspalten einmal ausgeführt. Man hat nun also vier Wahrheitswerte, welche aussagen, ob sich derzeit eine Note in einer der vier Hit-Bereichen befindet, und somit jetzt eine passende Aktion ausgeführt werden soll. Das Modell sollte hier also nun einen Tastendruck zurückgeben. Nun wird geprüft, ob die Ausgabe des Modells mit der Analyse des Hit-Bereichs übereinstimmt - die Note also richtig erkannt und gedrückt wurde. Ist dies der Fall, wird dem Modell positiver Reward, nach dem Runterskalieren also der Wert +1, zurückgegeben. Wurde die Aktion nicht ausgeführt, die Note also verpasst, wird negativer Reward, -1, an das Modell gegeben. Wird nichts erkannt und auch keine Aktion ausgeführt, wird ebenfalls +1 zurückgegeben, da dies in dieser Situation das richtige Verhalten ist. So soll kontinuierliches Tastendrücken abgewöhnt/verhindert werden.

**def** **get\_reward**(self, img, action):  
 # Crop score part of the window screenshot and top of the gameplay section  
 score\_img = img[self.score\_location['top']:(self.score\_location['top'] + self.score\_location['height']), self.score\_location['left']:(self.score\_location['left'] + self.score\_location['width'])]  
 past\_arrows\_img = img[env.past\_arrows\_location['top']:(env.past\_arrows\_location['top'] + env.past\_arrows\_location['height']), env.past\_arrows\_location['left']:(env.past\_arrows\_location['left'] + env.past\_arrows\_location['width'])]  
  
 # If no input should have occured, give negative reward  
 **if** (self.pattern\_recog\_helper.input\_expected(past\_arrows\_img, action) == False):  
 **return** -1  
  
 # If the score increased, give positive reward  
 new\_reward = self.pattern\_recog\_helper.analyze\_score(score\_img)  
 **if** (new\_reward > self.previous\_reward):  
 # Set the current reward as the previous reward for the next iteration  
 self.previous\_reward = new\_reward  
 **return** 1  
   
 # If the score didn't change, and no action has taken place, give a neutral reward  
 **else**:  
 **return** 0

**def** **input\_expected**(self, img, action):  
 hsv = cv.cvtColor(img, cv.COLOR\_BGR2HSV)  
 left\_arrow = hsv[:,5:35,:]  
 down\_arrow = hsv[:,40:70,:]  
 up\_arrow = hsv[:,70:100,:]  
 right\_arrow = hsv[:,105:135,:]  
  
 # Checks for left, down, up and right input  
 **if** (action == 1):  
 **return** self.check\_for\_color(left\_arrow)  
 **elif** (action == 2):  
 **return** self.check\_for\_color(down\_arrow)  
 **elif** (action == 3):  
 **return** self.check\_for\_color(up\_arrow)  
 **elif** (action == 4):  
 **return** self.check\_for\_color(right\_arrow)  
 # If no action took place, return True  
 **else**:  
 **return** True  
   
**def** **check\_for\_color**(self, img):  
 lower\_val = np.array([0, 150, 150], dtype='uint8')  
 upper\_val = np.array([255, 255, 255], dtype='uint8')  
 mask = cv.inRange(img, lower\_val, upper\_val)  
 hasColor = np.sum(mask)  
 **if** (hasColor > 0):  
 **return** True  
 **else**:  
 **return** False

Wichtig ist, diese Art der Erkennung nur zum Validieren des Inputs zu nutzen. Um ein “perfektes” Modell zu erzeugen, könnte man natürlich beim Erkennen eines farbigen Pixels direkt einen Tastendruck triggern - dies hat allerdings nichts mit künstlicher Intelligenz zu tun.

Nachdem das Environment angepasst wurde, wurden mehrere Tests durchgeführt, welche in der Länge variierten. Der längste von ihnen war 150.000 Schritte lang. In der folgenden Grafik sieht man die Vergleiche zwischen den einzelnen Lernprozessen. Man sieht, dass der Mittelwert der Rewards pro Episode kontinuierlich steigt, was Lernfortschritt signalisiert. Der Wert flacht gegen Ende des Lernprozesses ab, was allerdings auch an der kurzen Lerndauer liegt - 150.000 Schritte reichen bei einem komplexen Spiel wie StepMania nicht, um ein gutes Modell zu erzeugen.

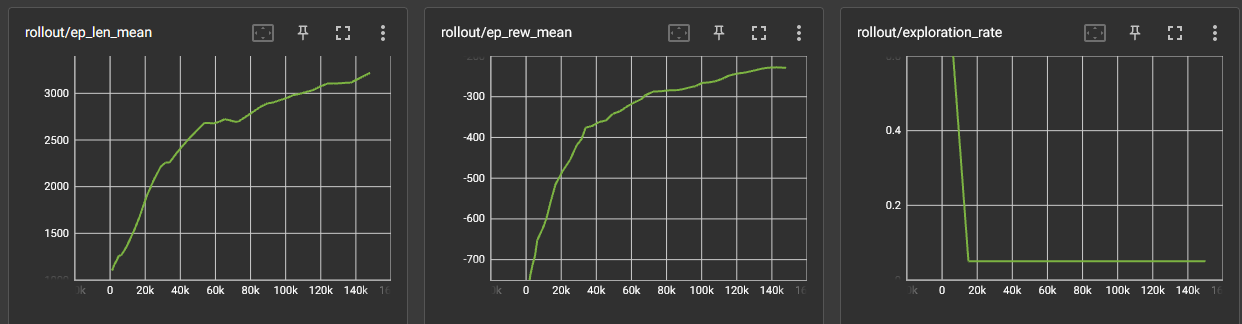


Abb. - Screenshot von Tensorboard - Darstellung der Mittelwerte der Episodenlänge, des Rewards, sowie der Wert der Exploration Rate nachdem das Rewardsystem überarbeitet wurde

Die überarbeiteten Konzepte funktionieren, wie man bei den Resultaten sehen konnte, ziemlich gut. Problematisch wird es allerdings, wenn es in dem Song mehr als nur Pfeile gibt. StepMania hat neben den normalen Pfeilen, welche man nur einmal antippen muss, auch noch andere Pfeilarten. Ebenfalls vorkommen können die bereits angesprochenen “Holds”, also Noten, welche für einen längeren Zeitraum gedrückt gehalten werden müssen, “Mines”, welche gar nicht gedrückt werden dürfen, “Rolls”, welche wiederholend gedrückt werden müssen, und “Lifts”, welche umgekehrt wie normale Pfeile funktionieren - hier muss die Taste vorher gedrückt werden, und im richtigen Moment losgelassen werden.

[Bild der unterschiedlichen Notenarten einfügen]

Während “Rolls” und “Lifts” erst in höheren Schwierigkeitsgraden relevant werden, können “Holds” und “Mines” in jeder Art von Schwierigkeit vertreten sein. Dies ist insofern problematisch, da bei dem Prüfen für negatives Feedback nicht zwischen den einzelnen Pfeilarten unterschieden wird. Eine “Mine” beispielsweise wird im momentanen Zustand der Funktion als Pfeil identifiziert, und das Modell bekommt negatives Feedback, wenn es keine Taste drückt - also genau das Gegenteil von dem, was eigentlich passieren sollte. Ein “Hold” wird von der Funktion für die gesamte Dauer des “Holds” als Pfeil identifiziert. Das Modell soll also permanent diese Taste drücken, was allerdings nicht mit der Funktionsweise des Tastendrücken im Environment funktioniert. Ein Gedrückthalten einer Taste ist auf Grund technischer Limitierungen nur als Toggle umzusetzen - wird auf Grund einer Aktion eine Taste gedrückt, wird sie so lange heruntergedrückt, bis dieselbe Aktion die Taste wieder loslässt. Um einen “Hold” richtig auszuführen, müsste also erst eine Aktion eine Taste runterdrücken, dann warten und keine Aktion mit dieser Taste ausführen bis der “Hold” vorbei ist, um dann mit derselben Aktion die Taste wieder loszulassen. Da aber der “Hold” durchgehend als Pfeil identifiziert wird, erwartet die Funktion, dass in jedem Step diese Taste gedrückt wird. So wird ebenfalls falsches Feedback vermittelt.

## Bewertung und Einordnung

Die Erkenntnisse aus der eigenen Umsetzung sollen in diesem Kapitel retrospektiv betrachtet werden. Dabei soll analysiert werden, inwiefern andere Formen der Umsetzung andere Ergebnisse geliefert hätten, wo Schwierigkeiten mit den genutzten Technologien auftraten, ob und wie diese behoben hätten werden können, und inwiefern man die Ergebnisse auf andere Spiele oder Anwendungsbereiche übertragen kann.

# Fazit und Ausblick

Im letzten Kapitel sollen die Erkenntnisse aus der Arbeit kurz zusammengefasst werden. Zusätzlich soll in die nahe Zukunft geschaut werden, wie sich das Thema KIs in den kommenden Jahren weiter entwickelt, und inwiefern es den Sprung und die Integrierung in die Gesellschaft/den Mainstream schafft.

## Fazit

## Ausblick

Literaturverzeichnis

Barthel, Julia; Ciesielski, Rebecca (2023): Regeln zu ChatGPT an Unis oft unklar. tagesschau. Online verfügbar unter https://www.tagesschau.de/wissen/technologie/ki-chatgpt-uni-wissenschaft-101.html, zuletzt aktualisiert am 15.05.2023, zuletzt geprüft am 31.05.2023.

Novak, Matt (2023): AI-Created Images Aren’t Protected By Copyright Law According To U.S. Copyright Office. Forbes. Online verfügbar unter https://www.forbes.com/sites/mattnovak/2023/02/22/ai-created-images-in-new-comic-book-arent-protected-by-copyright-law-according-to-us-copyright-office/, zuletzt aktualisiert am 22.02.2023, zuletzt geprüft am 31.05.2023.

Roose, Kevin (2022): An A.I.-Generated Picture Won an Art Prize. Artists Aren’t Happy. New York Times. Online verfügbar unter https://www.nytimes.com/2022/09/02/technology/ai-artificial-intelligence-artists.html, zuletzt aktualisiert am 02.09.2022, zuletzt geprüft am 31.05.2023.

Spehr, Michael (2023): Darth Vader küsst Marilyn. Frankfurter Allgemeine Zeitung. Online verfügbar unter https://www.faz.net/aktuell/technik-motor/digital/midjourney-bildgeneratoren-mit-ki-erzeugen-fotos-die-es-nicht-gibt-18753895.html, zuletzt aktualisiert am 20.03.2023, zuletzt geprüft am 31.05.2023.

Xiang, Chloe (2023): This Virtual Twitch Streamer is Controlled Entirely By AI. VICE. Online verfügbar unter https://www.vice.com/en/article/pkg98v/this-virtual-twitch-streamer-is-controlled-entirely-by-ai, zuletzt aktualisiert am 04.01.2023, zuletzt geprüft am 31.05.2023.

Zhang, Daniel; Maslej, Nestor; Brynjolfsson, Erik; Etchemendy, John; Lyons, Terah; Manyika, James et al. (2022): The AI Index 2022 Annual Report. Hg. v. AI Index Steering Committee, Stanford Institute for Human-Centered AI, Stanford University. Online verfügbar unter https://aiindex.stanford.edu/wp-content/uploads/2022/03/2022-AI-Index-Report\_Master.pdf, zuletzt geprüft am 28.03.2023.