

---

# **Endless Runner with Deep Reinforcement Learning**

ausgearbeitet von: Oliver Mertens, Tobias Mink

Bildbasierte Computergrafik  
Medieninformatik Master  
Fakultät für Informatik und Ingenieurwissenschaften  
TH Köln

Matthias Groß  
Fabian Friedrichs

Gummersbach, 22. Februar 2023

## **Abstract**

Die hier vorliegende Ausarbeitung ist eine Dokumentation der praktischen Arbeit innerhalb des Moduls „Bildbasierte Computergrafik“ dar. Sie basiert auf einem zuvor analysierten vorgestellten wissenschaftlichen Paper mit dem Titel „Playing FPS Games with Deep Reinforcement Learning“ der beiden Autoren Chaplot und Lample. Dort wurde das Spiel Doom mittels einer Kombination aus Objekterkennung, Spieleengine und Deep Reinforcement Learning von einem *agent* eigenständig gespielt. Innerhalb des darauf aufbauenden Projekts wird der Versuch unternommen eine sogenannten „Endless Runner“ mittels einer Kombination aus lediglich Objekterkennung und Deep Reinforcement Learning von einem *agent* spielen zu lassen. Hiermit soll die aufgestellte These, dass Deep Reinforcement Learning in Kombination mit einer Objekterkennung, unabhängig vom Spielegenre erfolgreich zum Spielen eines Spiels eingesetzt werden kann, ohne von einer Spieleengine abhängig zu sein, auf ihre Gültigkeit hin überprüft werden.

---

### **Anmerkung zur Schreibweise**

Zur besseren Lesbarkeit wird auf die gleichzeitige Verwendung weiblicher und männlicher Sprachformen verzichtet. Sämtliche personenbezogenen Bezeichnungen gelten gleichwohl für alle Geschlechter. Es wird darauf aufmerksam gemacht, dass alle beschriebenen Themen dieser Arbeit stets im Kontext der Mathematik stehen und nicht im Allgemeinen formuliert sind.

### **Anmerkung zur Arbeitsaufteilung**

Die gesamten Arbeitsergebnisse dieses Papers wurden im Rahmen des „Bildbasierte Computergrafik“ - Moduls zu gleichen Teilen von den zuvor genannten Autoren erstellt.

---

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>4</b>
1.1 Allgemeine Informationen zum Spielegenre „Endless Runner“ . . . . .	5
1.2 Ein Szenario eines „Endless Runners“ innerhalb der „Unreal Engine“ . . . . .	6
<b>2 Herangehensweise</b>	<b>8</b>
2.1 Betrachtete Netzwerke . . . . .	8
2.1.1 YOLO . . . . .	9
2.1.2 DQN . . . . .	9
2.2 Exkursion: Snake Game . . . . .	10
2.3 Umsetzung des neuen Netzwerks . . . . .	11
2.3.1 YOLOv5 . . . . .	11
2.3.2 DQN . . . . .	15
2.4 Training des Netzwerks . . . . .	16
<b>3 Fazit</b>	<b>18</b>
<b>Literaturverzeichnis</b>	<b>19</b>

---

# 1 Einleitung

Diese Dokumentation basiert auf dem wissenschaftlichen Paper „Playing FPS Games with Deep Reinforcement Learning“ (Chaplot & Lample, 2017) der beiden Autoren Guillaume Lample und Devendra Singh Chaplot, welches im Rahmen des Moduls „Bildbasierter Computergrafik“ im Wintersemester 22/23 von den zuvor genannten Autoren dieser Dokumentation im Vorlauf dieser Ausarbeitung bereits bearbeitet und auch im Plenum vorgestellt wurde. Basierend auf diesem wissenschaftlichen Paper soll ein eigenes Projekt thematisch passend gefunden und ausgearbeitet werden. Diese Ausarbeitung dokumentiert dieses ausgearbeitete Projekt.

Im vorliegenden wissenschaftlichen Paper wurde das Spiel „Doom“ mittels einer Kombination aus Bilderkennungssoftware und Spieleanwendung von einem computergenerierten *agent* eigenständig gespielt. Hierbei wurden Herausforderungen und entsprechende Lösungsansätze präsentiert. Trainiert wurde dieser *agent* durch Ansätze aus dem Bereich „Deep Reinforcement Learning“, die innerhalb dieser Dokumentation nicht noch einmal detailliert erläutert und als bekannt vorausgesetzt werden. (vgl. Chaplot & Lample, 2017)

Das eigenständige Projekt der Autoren dieser Ausarbeitung erweitert die Ansätze des vorliegenden wissenschaftlichen Papers um einige Aspekte, die im Folgenden kurz dargelegt werden. Zunächst einmal wird das Genre des Spiels gewechselt. Anstatt eines First Person Shooters wie „Doom“ wird stattdessen versucht einen sogenannten „Endless Runner“ mittels Deep Reinforcement Learning durch einen *agent* spielen zu lassen. Was genau sich hinter diesem Genre verbirgt wird in Abschnitt 1.1 näher erläutert. Beim eigenständigen Projekt wird allerdings gänzlich auf den Einsatz einer Spieleanwendung verzichtet und auf einen Erfolg mit lediglich einer Bilderkennungssoftware gesetzt. Hiermit soll festgestellt werden wie stark der Erfolg des Deep Reinforcement Learnings, ähnlich wie im vorliegenden wissenschaftlichen Paper, von der Spieleanwendung abhängt.

Im Rahmen dieser Ausarbeitung stellen die beiden Autoren also die These auf, dass Deep Reinforcement Learning in Kombination mit einer Objekterkennung, unabhängig vom Spielegenre erfolgreich zum Spielen eines Spiels eingesetzt werden kann, ohne von einer Spieleanwendung abhängig zu sein.

## 1.1 Allgemeine Informationen zum Spielegenre „Endless Runner“

Ein Endless Runner gehört in die Kategorie der Actionspiele, ist linear aufgebaut, ohne Ende, ohne Pausen und ohne Stages oder Levelwechsel. Es ist also ein fortlaufender Spiel durchlauf. Die Spielschwierigkeit beginnt dabei langsam und einfach, steigert sich jedoch über den Spielverlauf immer mehr. (vgl. Momoda)

Veranstaltungsorte des Spiels sind hierbei, je nach Spiel, häufig Wege, Eisenbahnschienen und Straßen, so wie in Abbildung 1 beispielhaft für drei Spiele aus dieser Kategorie dargestellt.

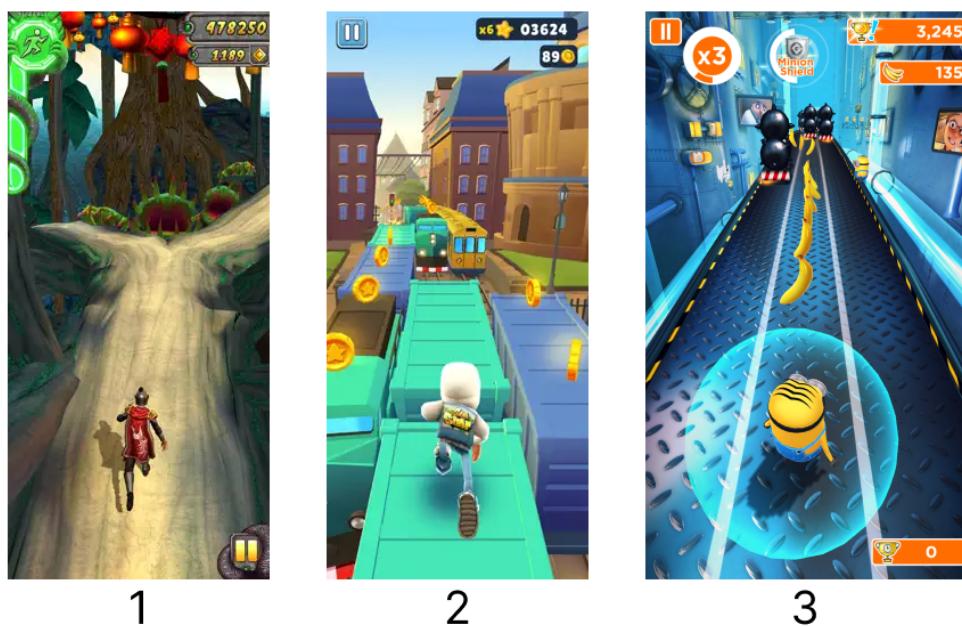


Abbildung 1: Drei Beispiele für das Spielegenre „Endless Runner“. (1: Temple Runner, 2: Subway Surfer, 3: Despicable Me/Minions Rush)

Die Schwierigkeitsstufe des Spiels wird über die Geschwindigkeit der Spielfigur gesteuert. Diese scheint immer schneller zu laufen, als ob diese auf einem riesigen Laufband wäre, dessen Geschwindigkeit zunimmt. Auf der Spielfläche nähern sich während einem Spieldurchlauf immer wieder Hindernisse und/oder andere Dinge die auch einen positiven Effekt hervorrufen können. Die Geschwindigkeit der Hindernisse und anderen Dinge nimmt ebenfalls simultan zur Geschwindigkeit der Spielfigur zu, sodass dem Spieler immer schnellere Reaktionen zum Ausweichen und/oder Einsammeln abverlangt werden. Irgendwann ist der Zeitpunkt erreicht, dass die Reflexe des Spielers nicht mehr ausreichen und der Spieler der Intensität des Spiels erliegt, da er den Hindernissen nicht mehr rechtzeitig ausweichen kann. An dieser Stelle endet ein Spieldurchlauf. (vgl. Momoda)

Das Spielegenre der „Endless Runner“ wird in der Regel auf mobilen Endgeräten gespielt, sodass die Steuerung der Spielfigur über einen Berührungsempfindlichen Bildschirm vollzogen wird. Diese besteht dabei aus verschiedenen Wischbewegungen, die es dem Spieler

---

erlauben die Spielfigur in verschiedene Richtungen zu bewegen. So kann die Spielfigur sowohl nach rechts oder links als auch nach oben oder unten bewegt werden um Hindernissen auszuweichen oder Gegenstände einzusammeln.

## 1.2 Ein Szenario eines „Endless Runners“ innerhalb der „Unreal Engine“

Für die Durchführung des Projektes ist es zunächst nötig ein eigenes Spiel der Kategorie „Endless Runner“ anzufertigen. Hierzu wird die Plattform „Unreal Engine“ verwendet, da die Autoren dieser Ausarbeitung bereits Vorerfahrungen innerhalb dieser Engine vorweisen können und somit ein effizienter Workflow bei der Erstellung des eigenen Spiels garantiert ist. Theoretisch gesehen wäre allerdings auch jede weitere Engine/Plattform denkbar, sofern es innerhalb dieser möglich ist einen „Endless Runner“ zu generieren. Weiteres hierzu in Abschnitt 2.

Damit der zu erstellende „Endless Runner“ tatsächlich auch endlos gespielt werden kann, zumindest bis dem Spieler ein Fehler unterläuft und er falsch abbiegt oder gegen ein Hindernis läuft, muss der Aufbau der Spieleumgebung automatisiert werden. Hierzu wurde sich entschieden den Aufbau nach einem „Tile-System“ zu gestalten. Das bedeutet, dass jedes generierte Element der Spieleumgebung einen definierten Startpunkt besitzt, der optimal an einen zuvor generierten Entpunkt passt. Die Auswahl der einzelnen Teilstücke kann dann durch Zufall getroffen werden und jeder Spieldurchlauf erhält einen völlig neuen Spielekurs. Der Spieler betrachtet die Spielumgebung aus Sicht der dritten Person und schaut seinem Spielcharakter mittels einer an diesem fixierten Kamera über die Schulter. Abbildung 2 verdeutlicht noch einmal visuell die einzelnen Teilstücke zur automatischen generierung der Spielumgebung und die fixierte Kamera, die den Bewegungen des Spielcharakters folgt.

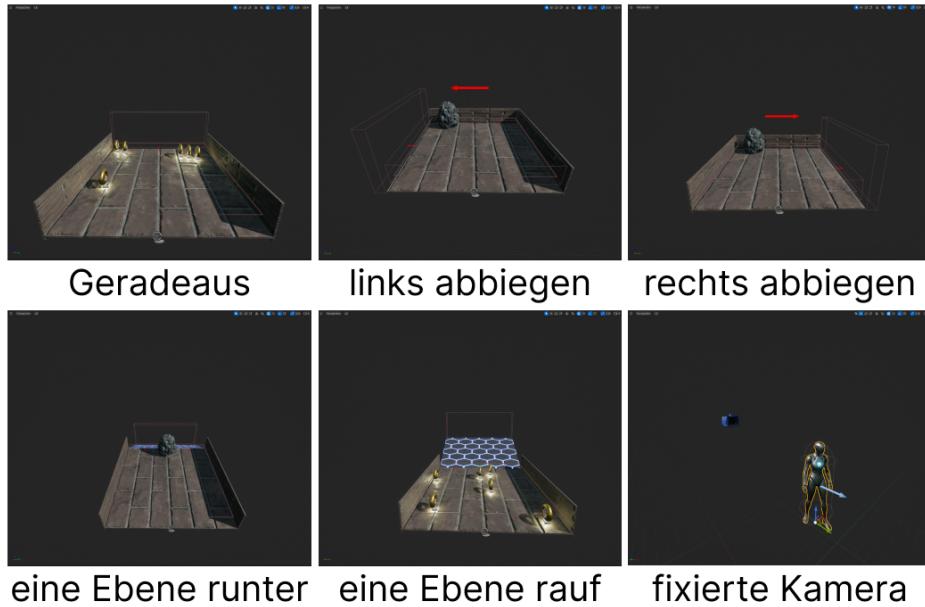


Abbildung 2: Die einzelnen Teilstücke, die zur automatischen Generierung der Spieleumgebung ausgewählt werden und die, den Bewegungen der Spielfigur folgende Kamera

Zusätzlich zur automatischen Generierung der Spieleumgebung besitzen die jeweiligen Teilstücke diverse Spawnpunkte, an denen unterschiedliche Dinge ebenfalls nach dem Zufallsprinzip innerhalb der Spielumgebung gerendert werden können. So kann es sein, dass der Spieler an der jeweiligen Stelle entweder ein Hindernis (Obstacle) vorfindet, dem es aus dem Weg zu gehen gilt, oder eine Münze (Coin), die es einzusammeln gilt. Hiermit wird also deutlich, dass der Spieler während eines Spieldurchlaufs verschiedene Dinge hat, auf die er achten muss. Obstacle gilt es zu umlaufen, Coins können/müssen gesammelt werden und an der richtigen Stelle muss eine weitere Entscheidung getroffen werden, ob nach rechts oder links abgebogen wird.

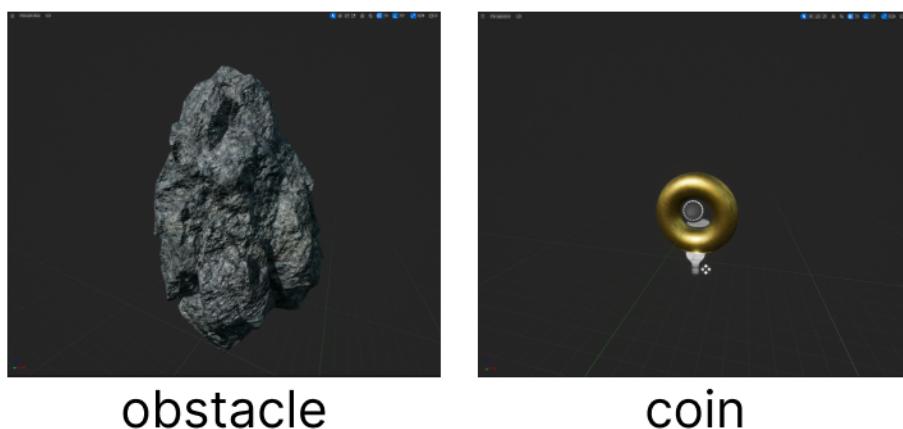


Abbildung 3: Obstacles müssen umlaufen werden, Coins gilt es einzusammeln

---

Als Steuerungseingaben hat der Spieler dazu die folgenden Möglichkeiten:

- input „A“ - nach links strafen / nach links abbiegen
- input „D“ - nach rechts strafen / nach rechts abbiegen
- (input „Leertaste“ - springen)

Die vorherige Liste impliziert schon, dass mit einer jeweiligen Taste an manchen Stellen im Spiel unterschiedliche Funktionen durchgeführt werden können. Abhängig von der Position der Spielfigur kann beispielsweise beim betätigen der „A“-Taste eine seitliche Bewegung nach links durchgeführt oder sich für das Abbiegen nach links entschieden werden. Diese Unterscheidung führt das Spiel eigenständig aus und der Spieler muss nur die entsprechende Taste verwenden. Die Aktion des Springens ist in der vorherigen Liste bewusst ausgeklammert, da im Laufe des iterativen Prozesses entschieden wurde darauf zu verzichten. Ein noch komplexeres Spiel, welches auch noch das Springen ermöglicht und auch erfordert ist nicht Zielführend und erfordert mehr Arbeitsaufwand an der falschen Stelle. Daher kann das Spiel mit den zuvor genannten Möglichkeiten als ausreichend betrachtet werden, um sich daraufhin wieder der eigentlichen Herausforderung dieser Ausarbeitung zu widmen.

## 2 Herangehensweise

Das Ziel des Projekts soll es sein, ein ähnliches Model wie im zuvor betrachtetem Paper zu entwickeln, welches allerdings alle Informationen über das direkte einlesen von Pixeln erhält. Diese Entscheidung beruht zum einen auf der Tatsache, dass keine Umgebung wie „ViZDoom“ zur Verfügung steht. Zum anderen soll damit herausgefunden werden, inwieweit diese begrenzte Informationsquelle dazu genutzt werden kann, um einen *agent* zu trainieren. Auf einen zusätzlichen Input aus der zuvor in Abschnitt 1.2 genannten „Unreal Engine“ wird daher, im Gegensatz zum betrachteten Paper, bewusst verzichtet. Diese Engine dient also nur zur Entwicklung und Bereitstellung des Spiels. Das zu entwickelnde Model muss also eigenständig die Fähigkeit besitzen, Bildinformationen vom laufenden Spiel einzulesen, diese zu interpretieren und darauf basierend eine Entscheidung zu treffen um mit dieser direkt in das Spielgeschehen einzutreten.

Zu Beginn wurden daher unterschiedliche bestehende Architekturen und Projekte recherchiert, um einen Überblick über bereits bestehende Ansätze und Umsetzungsmöglichkeiten zu erlangen. Diese werden in Abschnitt 2.1 vorgestellt und näher erläutert.

### 2.1 Betrachtete Netzwerke

Der zukünftige *agent* soll die Fähigkeit besitzen, alle essentiellen Objekte, wie z.B. ein Hindernis oder eine Münze, während des Spielverlaufs zu erkennen und auf Basis dieser Erkenntnis eine *action* zu interpretieren. Es muss also eine oder mehrere Umsetzungen gefunden werden, die diesen Anforderungen gerecht werden.

## 2.1.1 YOLO

Bei der Rechere nach einer solchen Umsetzung werden Beispiele für CNN- und RCNN-Netzwerke betrachtet. Um die gewünschte Performance zu gewährleisten, wurde die Option ein neues CNN-Modell mit PyTorch zu erstellen verworfen, da die Optimierung für einen bestimmten Nutzungskontext zu viel Zeit beansprüchen würde. Aus diesem Grund wurde das bereits optimierte YOLOv5-Netzwerk von Ultralytics ausgewählt. YOLO ist ein beliebtes Objekterkennungs- und Bildsegmentierungsmodell, dass für seine hohe Geschwindigkeit und Genauigkeit populär geworden ist. Dieses ermöglicht, neben einer guten Dokumentation und einer einfachen Implementierung, die genaue Verfolgung jedes einzelnen Trainingsverlaufs, sobald das Netzwerk mit einem eigens erstellten Datensatz von Bildern bestückt wird.

Nachdem das Netzwerk mit einem Datensatz von Bildern trainiert wurde, erhält es als Input einen einzelnen *frame*, beispielsweise ein vorgefertigtes Bild oder einen Screenshot. Falls das Netzwerk dann darin bekannte Objekte erkennt, werden diese durch eine Box eingerahmt. Der Output definiert sich dabei durch einen Sammlung von Angaben der identifizierten Objekte. Diese beschreiben sich dabei durch die Angaben der Koordinaten der Eckpunkte der Box, der Sicherheit des Netzwerk, ob die Angabe richtig ist und dem Namen des Objekts.



Abbildung 4: Links: das vorgefertigte Bild, auf diverse Fahrzeuge zu sehen sind und auch vom Netzwerk erkannt und unterschieden werden.  
Rechts: die entsprechende Tensor-Ausgabe

Wie genau die Implementierung des YOLOv5-Netzwerks innerhalb dieser Ausarbeitung und dem dementsprechenden Projekt aussieht wird in Abschnitt 2.3.1 näher beschrieben.

## 2.1.2 DQN

Im Vergleich zum betrachteten Paper wurde sich bei diesem Projekt für die Umsetzung eines DQNs entschieden. Die Autoren begründen ihre Wahl ein DRQN zu nutzen damit, dass es dem *agent*, durch den auf 90° beschränkten Blickwinkel, nicht möglich sei, alle notwendigen Informationen zu erhalten, um den aktuellen *state* zu beschreiben (vgl. Chaplot & Lample, 2017, S. 2141, A. 2.2).

---

Im Endless Runner-*enviroment* besteht dieses Problem hingegen nicht. Der Blickwinkel ist hier fest definiert und ermöglicht in jedem *frame* die Erfassung aller Informationen, um den *state* des betrachteten *enviroments* zu erfassen.

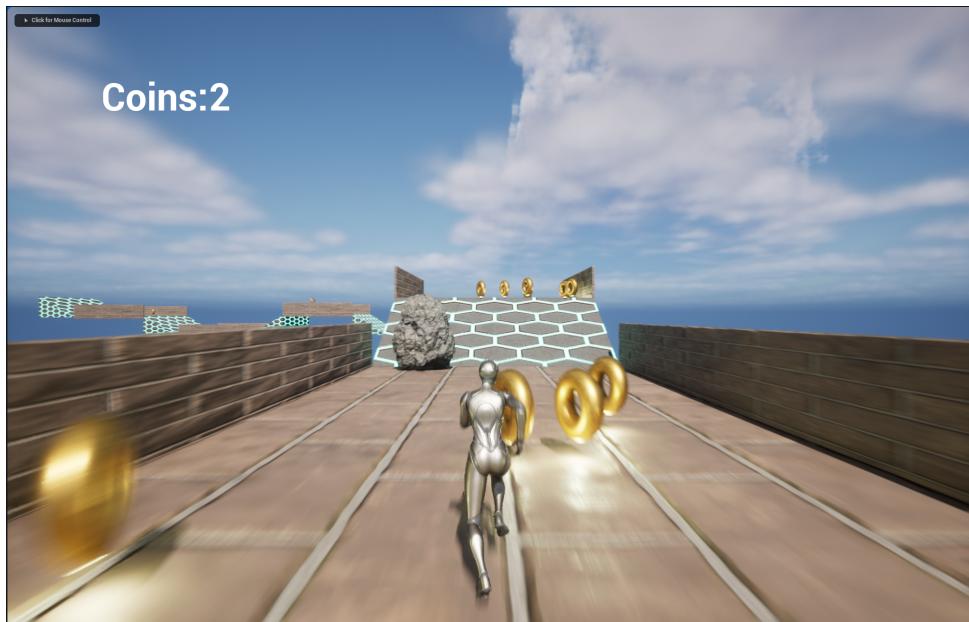


Abbildung 5: Der fertige Endless Runner innerhalb der Unreal Engine

## 2.2 Exkursion: Snake Game

Um die generelle Funktionsweise eines DQNs besser zu verstehen, wurde anhand einer Anleitung ein Netzwerk für das Spiel *Snake* entwickelt. Alle Informationen, die zur Beschreibung des *states* von nötigen sind, werden dabei vom Spiel selber geliefert. Die visuelle Repräsentation des Spiels spielt also keine Rolle. Deshalb besteht das hier entwickelte DQN lediglich durch einen Input-, einem Hidden- und einem Output-Layer. Während sich der Input durch den *state* beschreibt, umfasst der Output die drei Bewegungsmöglichkeiten der Schlange: Vorwärts, Links und Rechts.

Zu jeder neu gestarteten Trainingsphase beginnt der *agent* damit das *enviroment* zu erkunden und anhand der erhaltenen *rewards* seinen Spielstil anzupassen. Dieser tradeoff zwischen exploration und exploitation führt dazu, dass die *actions* des *agents* mit der Zeit weniger willkürlich werden.

Während einem Trainingsdurchlauf spielt der *agent* unbegrenzt viele Spieldurchläufe nach volgendem Muster:

- Aktuellen *state* bestimmen
- *action* bestimmen
- *action* ausführend und dadurch reward erhalten und einen möglichen game over Zustand bestimmen

- Neuen *state* bestimmen
- Q-values für betrachteten state-Wechsel und erhaltenen reward bestimmen
- Alle Informationen zwischenspeichern
- (Optionale wenn game over) Spiel zurücksetzen, Q-values der gesammelten Informationen eines Spieldurchlaufs bestimmen und Zähler für gespielte Spiele um eins hochsetzen

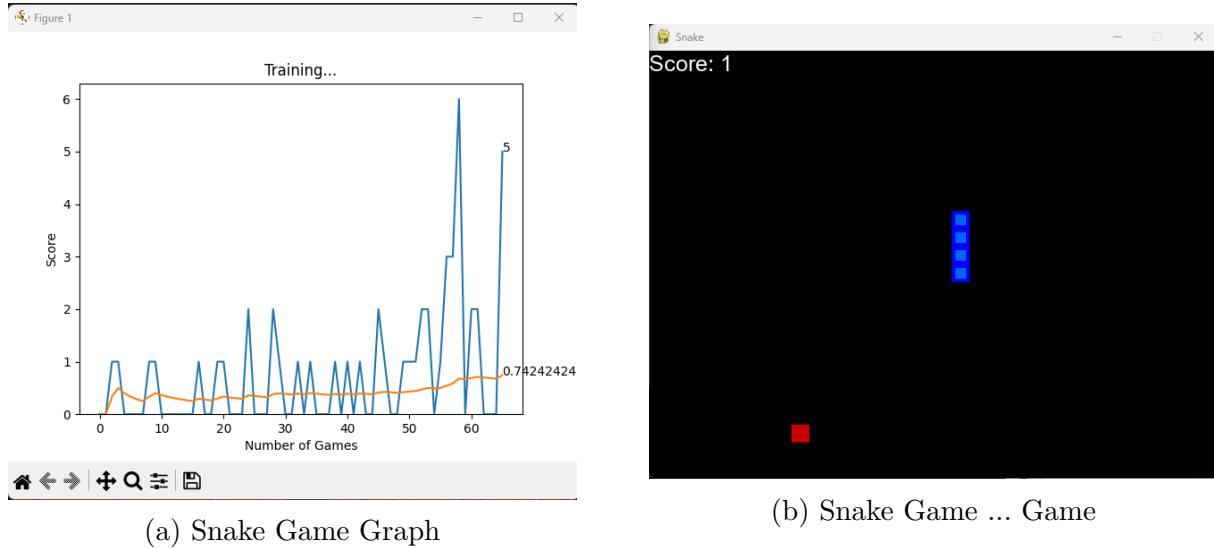


Abbildung 6: Optionaler text

Das Snake-Game kann hier gefunden werden: freeCodeCamp.org (2022).

## 2.3 Umsetzung des neuen Netzwerks

### 2.3.1 YOLOv5

In diesem Abschnitt wird die bereits in Abschnitt 2.1.1 zuvor erwähnte Implementierung von YOLOv5 erläutert. Die Arbeit mit diesem Netzwerk wurde für das vorliegende Projekt in sechs Schritte eingeteilt, die logisch aufeinander aufbauen und im Folgenden detailliert besprochen werden. Diese Schritte lauten:

1. Installieren und importieren der notwendigen Libraries
2. Das erste Standardmodell laden
3. Standbild/Einzelbild Objekterkennung testen
4. Objekterkennung in Echtzeit einrichten
5. Ein eigenes Modell trainieren
6. Eigenes Modell laden und zur weiteren Bearbeitung verwenden

Zunächst müssen einige Libraries importiert und eine Entwicklungsumgebung aufgesetzt werden. Welche genau das sind und welche Schritte hierzu nötig sind ist innerhalb des

---

GitHub-Repositories detailliert beschrieben und wird in dieser Dokumentation nicht noch einmal aufgeführt.

Nachdem die grundlegenden Libraries importiert und eine Entwicklungsumgebung aufgesetzt worden ist muss sich zunächst für ein Modell aus dem YOLOv5-Netzwerk entschieden werden. Hier stehen einige zur Auswahl, die sich, wie Abbildung 7 verdeutlicht sowohl in Genauigkeit als auch Geschwindigkeit unterscheiden. Je nach Anwendungsfall kann ein langsameres aber auch genauereres Modell von Nöten sein oder genau umgekehrt.

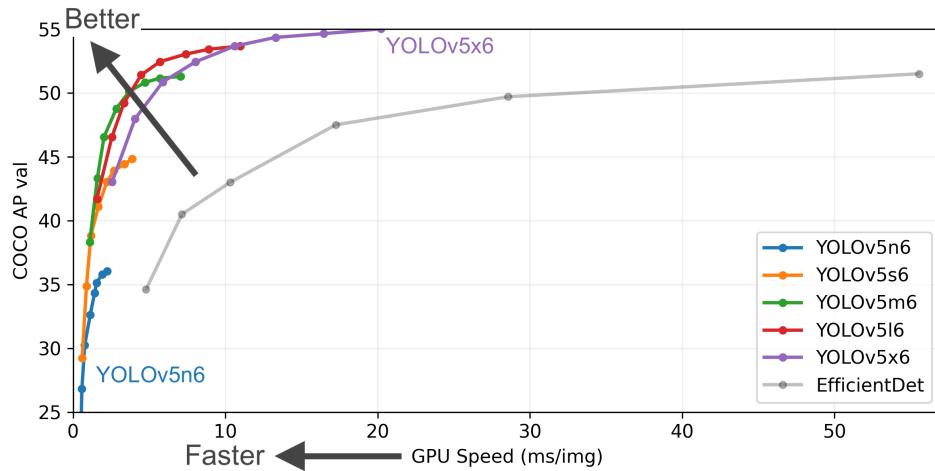


Abbildung 7: YOLO Versionen

Da das vorliegende Spiel aus dem Spielegenre der „Endless Runner“ stammt und somit eine kontinuierlich steigende Spielgeschwindigkeit aufweist, die wiederum auch eine kontinuierlich steigende Geschwindigkeit in der Entscheidungsfindung erfordert wurde sich, nach einigen Testläufen, für den Einsatz des YOLOv5s Modells entschieden. Dieses bietet, wie Abbildung 7 verdeutlicht, parallel zu einer hohen Bearbeitungsgeschwindigkeit eine hohe Genauigkeit bei der Objekterkennung.

Ist also nun eine Entwicklungsumgebung aufgesetzt und auch das ausgewählte Modell geladen, so kann dieses mithilfe eines Standbildes getestet auf korrekte Funktion getestet werden. Sollte dies von Erfolg gekrönt sein, so kann ab diesem Zeitpunkt jedes beliebige Standbild mit dem geladenen Standard-Modell zur Objekterkennung verwendet werden. Voraussetzung für den Erfolg ist natürlich, dass dieses Bild Objekte enthält, die im Standard-Datensatz des Standard-Modells enthalten sind. Abbildung 4 aus Abschnitt 2.1.1 hat bereits solch ein Testbild gezeigt. Hier wurde versucht auf diesem Testbild diverse Fahrzeuge nicht nur zu erkennen, sondern auch in die Kategorien „Bus“, „Car“ und „Truck“ einzufügen.

Die Nächste größere Herausforderung ist, die bereits für Standbilder funktionierende Objekterkennung auch für Bewegtbilder anbieten zu können. Das Spiel wird im späteren Verlauf logischerweise nicht mit Standbildern gespielt. Stattdessen benötigt der *agent* fortlaufenden Input von Bewegtbildern, um auf deren Basis Entscheidungen treffen zu

---

können. Aus diesem Grund wird die Klasse „WindowCapture“ ins Leben gerufen, die es ermöglicht sowohl ein gezieltes Fenster auf dem Desktop, als auch einen gesamten Monitor für die Objekterkennung bereitzustellen. Hierzu wird die Auflösung des entsprechenden Fensters oder Monitors zunächst automatisch berechnet, um sie im nächsten Schritt auf ein für die Objekterkennung benötigtes Format zu transformieren. An dieser Stelle muss allerdings erwähnt werden, dass der Begriff „Bewegtbild“ an dieser Stelle eigentlich nicht korrekt ist. Was tatsächlich passiert, ist in einer enormen Geschwindigkeit Screenshots vom ausgewählten Fenster oder Monitor angefertigt werden und an das Modell übergeben werden. So sind es eigentlich weiterhin einzelne Standbilder, die allerdings durch ihre hohe Anzahl und die hohe Geschwindigkeit in der sie angefertigt werden, die Immersion von Bewegtbildern erzeugen. Wie diese einzelnen Frames weiter interpretiert werden können erläutert der Abschnitt 2.4.

Innerhalb dieses Abschnittes soll allerdings noch präsentiert werden, wie mithilfe von YOLOv5 ein eigenes Modell und ein eigener Datensatz angelernt werden kann. Die Klasse „WindowCapture“ ist nämlich ebenfalls in der Lage auf Wunsch Screenshots nicht einfach nur in ein vordefiniertes Modell einzupflegen, sondern ermöglicht auch das Abspeichern dieser Bilder nach einem gewissem Format, dass zum Anlernen eines neuen Modells und Datensatzes benötigt wird. Zur Erstellung eines neuen Datensatzes muss zunächst allerdings festgelegt werden welche neuen Elemente von der Objekterkennung erkannt werden sollen. Für das vorliegende Projekt lauten diese:

- coin
- explosion
- obstacle
- player
- turnLeft
- turnRight

Münzen und Hindernisse wurden zuvor schon besprochen. Was hier neu hinzukommt sind zum einen die Label „turnLeft“, „turnRight“, die den *agent* erkennen lassen sollen ob er links oder rechts abbiegen muss. Zum anderen ist das Label „player“ hinzugekommen, was den *agent* seine eigene Spielposition bzw. die Spielfigur erkennen lässt und zuletzt das Label „explosion“, das eine Explosion und somit auch eine Kollision mit einem Hindernis erkennbar macht. Solch eine Explosion hat das Ende des Spieldurchlaufs zur Folge und ein neuer Durchlauf beginnt automatisch.

Für jedes dieser Labels müssen Screenshots angefertigt werden, auf denen die entsprechenden Elemente zu sehen sind. Je mehr Screenshots pro Label vorhanden sind umso besser kann das Netzwerk und die Objekterkennung angelernt werden. Für dieses Projekt wurde sich für eine Screenshotanzahl von 20 Bildern pro Label entschieden, da diese im Folgeschritt händisch erst einmal identifiziert und markiert werden müssen. Das bedeutet, dass auf jedem Screenshot für das Label „coin“ sämtliche Münzen eigenhändig eingeramt und gelabelt werden müssen, so wie in Abbildung 8 dargestellt.

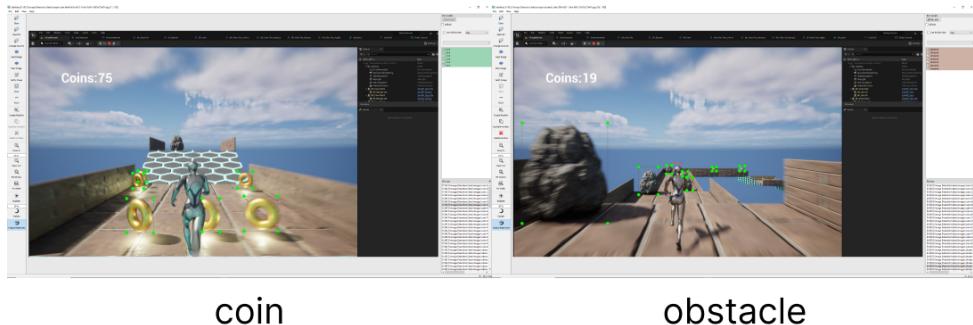


Abbildung 8: Alle Elemente des entsprechenden Labels müssen händisch identifiziert und markiert werden

Bei sechs Labeln, 20 Bildern pro Label und mehreren Iterationsprozessen entsteht hier schnell eine enorme Datenflut, die es zu handeln gilt. Ist dieser Datensatz allerdings fertig erstellt kann ein neues eigenes Modell mit eben diesem trainiert werden. Das bedeutet, dass das zugrundeliegende Netzwerk sich sowohl Bilder als auch Label anschaut und versucht zu lernen wie beispielsweise ein Hinderniss aussieht. Dieser Lernprozess kann über diverse Artefakte beobachtet und nachvollzogen werden. Abbildung 9 zeigt beispielsweise die während des Lernprozesses erstellte „Precision-Confidence“-Kurve über die eingesehen werden kann wie schnell das Netzwerk Präzise und sicher Elemente der jeweiligen Klassen erkennen konnte.

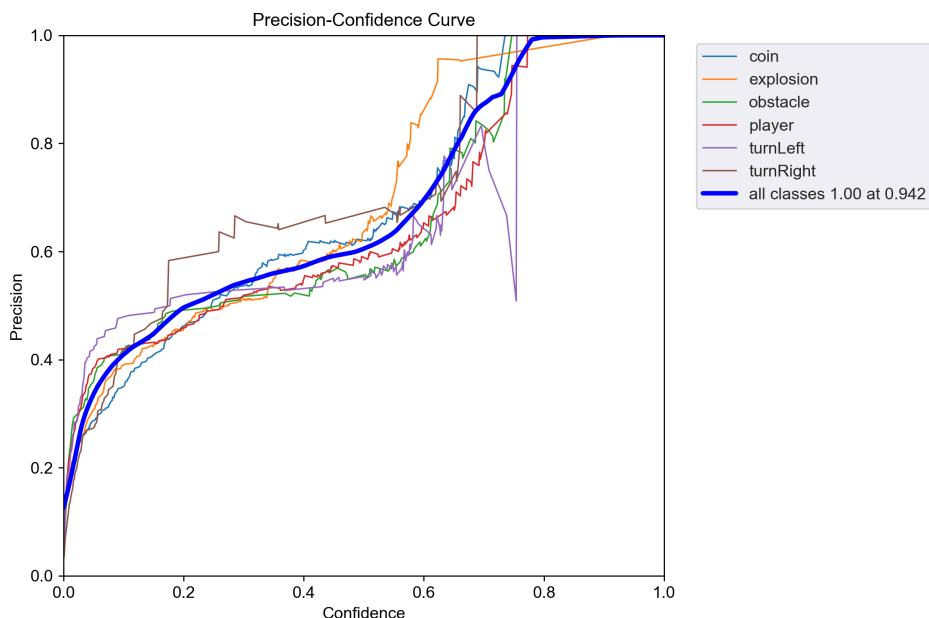


Abbildung 9: Die P-Kurve setzt Präzision und Sicherheit des Netzwerks in Relation und lässt die Veränderung während des Lernprozesses erkennen

Wichtig ist an dieser Stelle zu erwähnen, dass ein positives Ergebnis dieser Artefakte in der Praxis keine eindeutige Treffsicherheit bei der Objekterkennung garantiert. Die Werte stehen nur dafür wie sicher sich das Netzwerk selbst ist Objekte richtig erkannt zu haben.

Trotzdem kann es vorkommen, dass die Objekterkennung nicht korrekt gelernt hat und Dinge falsch oder auch gar nicht erkennt, so wie in Abbildung 10 beispielhaft festgehalten.



Abbildung 10: Trotz erfolgreichem training identifiziert das Netzwerk Bildelemente falsch

An dieser Stelle muss der Prozess des Anlernens erneut durchgeführt werden, im schlimmsten Fall ist auch ein erneuter Datensatz erforderlich. Entscheidend für den Erfolg oder Misserfolg des Anlernens ist letztendlich die Qualität des Datensatzes und die Anzahl der trainierten Epochen. Diese Anzahl sollte möglichst hoch gewählt werden, ist allerdings auch nicht unendlich wählbar. Ab einem gewissen Zeitpunkt stellt das Netzwerk eigenständig fest, dass kein weiterer Fortschritt erreicht wird und bricht den Trainingsprozess ab (siehe Abbildung 11)

```
Stopping training early as no improvement observed in last 100 epochs. Best results observed at epoch 1232, best model saved as best.pt.
To update EarlyStopping(patience=100) pass a new patience value, i.e. `python train.py --patience 300` or use `--patience 0` to disable EarlyStopping.
```

Abbildung 11: Das Netzwerk erreicht keinen Trainingsfortschritt mehr und bricht das Training ab

Wenn der Prozess des Anlernens allerdings erfolgreich durchgeführt wurde und das eigene Modell mithilfe der zuvor erwähnten Klasse „myWindowCapture“ mit Bewegtbildern gefüttert wird, so kann damit begonnen werden den *agent* auf die erkannten Bildelemente auf dem Fenster oder Monitor reagieren zu lassen.

### 2.3.2 DQN

Ein Vorteil der Implementierung des DQN-Netzwerks aus Abschnitt 2.2, ist der gut strukturierten Aufbau des Codes. Allerdings müssen einige Parameter und Funktionen angepasst werden, damit der Output des YOLOv5-Netzwerks genutzt werden kann, um eine *action* im neuen Endless Runner-*enviroment* zu bestimmen.

Damit es dem *agent* möglich ist seinen Spielstil entsprechend der erhaltenen *rewards* und des observierten *enviroments* anzupassen, mussten die Wert des *states* entsprechend des

---

neuen Endless Runner-Kontexts gewählt und bestimmt werden. Neben generellen Angaben zu den erkannten Objekte im Frame, wie z.B. einem *coin*, wird das Vorhandensein eins möglichen Kollisionskurs zwischen dem *player* und einem anderen Objekt übergeben. Diese Information soll den *agent* zusätzlich dabei unterstützen einen Rückschluss auf seine Handlung ziehen und entsprechend die nächste *action* anzupassen, zu können.

State:

- player
- coin
- obstacle
- explosion
- turnLeft
- turnRight
- player\_coin\_collision
- player\_obstacle\_collision

Eine Kollision steht dann bevor, wenn sich die Boxen des *player* und eines anderen Objekts überlappen. Aufgrund der Tatsache, dass die Kamera nicht manuell bewegt werden kann und sich jene immer am *player*-Objekt orientiert, kann davon ausgegangen werden, dass keine Verzerrung der Boxen stattfindet, welche die Berechnung beeinflusst.

Neben weiteren Funktionen zur Umformung und Anpassung von Datensätzen für die einzelnen Schritte der Testphase, wurde ebenfalls eine Funktion etabliert um dem *agent* das direkte Eingreifen ins Spielgeschehen zu ermöglichen. Dabei wird ein Tastendruck simuliert, was bedeutet, dass das Spiel zur Laufzeit aktiv ausgewählt sein muss. Zudem wurde eine neue *train()*-Funktion erstellt, welche die gesamte Routine beinhaltet, die das Netzwerk bei der Ausführung durchläuft. Jene ist in Codeform im Github vorhanden und wird genauer im folgenden Abschnitt 2.4 erklärt.

## 2.4 Training des Netzwerks

Während des Trainings durchläuft das Netzwerk mehrere Spieldurchläufe, deren Ende durch das erreichen des *game over*-Zustands definiert sind. Dieser Zustand ist dann erreicht, sobald eine Explosion erkannt wurde.

Mit dem Start des Netzwerks, wird der Bildschirm mit einer Wiederholungsrate von zehn *frames* pro Sekunde aufgenommen. Dabei wird jeder einzelne *frame* von der YOLOv5s-Komponente interpretiert und der Output jedes zehnten *frames* zur Bestimmung des *states* genutzt. Die Entscheidung, jeden zehnten *frame* zu betrachten, beruht darauf, dass von Anfang an nicht bekannt war, wie lange es dauert bis die ausgewählte *action* in den aufgenommenen *frames* zu erkennen ist. Nach der Bestimmung des *states* wird eine *action* bestimmt, was zu Beginn noch zufällig erfolgt. Die daraus resultierende *action* wird darauf als direkte Systemeingabe im Spiel ausgeführt.

Nach weiteren zehn *frames* wird ein neuer *state* bestimmt, welcher zur Beurteilung der

---

Einwirkung der getätigten *action* dient. Darauf basierend wird ein reward und ein möglicher *game over*-Zustand bestimmt, welche den Spieldurchlauf direkt beenden würde. Das Netzwerk speichert nun die Informationen des *state*-Wechsels, des erhaltenen *rewards* und des *game over*-Zustands in einem Zwischenspeicher und bestimmt darauf die Q-Values.

Sollte der *game over*-Zustand bei diesem Durchlauf nicht eingetreten sein, geht dieser Kreislauf mit der bestimmung des nächsten *states* weiter. Wenn dieser Zustand allerdings eingetreten ist, dann wird auf der Basis der gesammelten Informationen eines gesamten Spieldurchlaufs, welche im Zwischenspeicher hinterlegt sind, neue Q-values bestimmt. Danach wird der Kreilauf ebenso fortgeführt.

Mit jedem erreichen des *game over*-Zustands wächst die Chance, dass das gelernte Wissen des *agents* in die Auswahl der nächsten *actin* einbezogen wird. Dies beschreibt den, im Abschnitt 2.2 genannten, *tradeoff* zwischen *exploration* und *exploitation*.

---

### 3 Fazit

Zum aktuellen Stand der Bearbeitung liefert das Netzwerk nicht das gewünschte Ergebnis. Das YOLOv5-Netzwerk ist trotz mehrerer Iterationen und umfangreichen Datensatz in der derzeit trainierten Variante nicht in der Lage Bildelemente treffsicher zu erkennen und zu identifizieren. Häufig werden essentielle Elemente nicht erkannt oder falsch identifiziert. Dadurch, dass zur akkurate Objekterkennung wesentlich mehr Daten benötigt werden, welche zum Zeitpunkt der Ausarbeitung nicht zur Verfügung stehen, kann keine akkurate Objekterkennung garantiert werden.

Diese nicht akkurate Objekterkennung behindert den weiteren Verlauf des MDPs. Ohne eine korrekte Bildobjekterkennung kann auch keine korrekte Zuordnung von Rewards und keine Bestimmung der nächsten Aktion erfolgen. Stattdessen läuft der *agent* nahezu kopflos durch die Spielumgebung und erhält nur sehr selten positives oder negatives Feedback. Dementsprechend ist der Lerneffekt des *agents* nahezu nicht existent.

Aufgrund dieser Erkenntnisse muss die ursprüngliche These dieses Projekts, dass Deep Reinforcement Learning in Kombination mit einer Objekterkennung, unabhängig vom Spielegenre erfolgreich zum Spielen eines Spiels eingesetzt werden kann, ohne von einer Spieleanwendung abhängig zu sein, verworfen werden. Wie bereits zuvor erwähnt ist für den Erfolg oder Misserfolg des Projekts eine akurate Bilderkennung erforderlich, da auf einen Input durch eine Spieleanwendung, im vorliegenden Fall der Unreal Engine, verzichtet wurde. Mithilfe dieser Engine wäre ein Anlernen der zu erkennenden Objekte über Positionsdaten von eben diesen, ähnlich wie im Paper von Chaplot und Lample, erfolgsversprechender gewesen.

Vorausgesetzt die Bilderkennung wäre akkurer gewesen, kann davon ausgegangen werden, dass das vorliegende Netzwerk in der Lage gewesen wäre den Input der Bilderkennung zu interpretieren und darauf basierend Aktionen auszuführen. Die dafür erforderliche Anwendungslogik wurde innerhalb dieser Ausarbeitung und innerhalb des Projekts beschrieben und getestet und ist lauffähig. Zu einer erfolgreichen bestätigung der aufgestellten These müsste also lediglich eine akurratere Bilderkennungssoftware verwendet werden.

---

## Literatur

[Chaplot & Lample 2017] CHAPLOT & LAMPLE, Guillaume & Devendra S.: *Playing FPS Games with Deep Reinforcement Learning*. <https://ojs.aaai.org/index.php/AAAI/article/view/10827>. Version: 2017. – Last accessed 22. Februar 2023

[freeCodeCamp.org 2022] FREECODECAMP.ORG, freeCodeCamp.org: *Python + PyTorch + Pygame Reinforcement Learning – Train an AI to Play Snake*. <https://www.youtube.com/watch?v=L8ypSXwyBds>. Version: 2022. – Last accessed 22. Februar 2023

[Momoda ] MOMODA, Jerry: *Endless Runner Games: Evolution and Future*. <http://jerrymomoda.com/analysis-endless-runners/#comment-57>. – Last accessed 22. Februar 2023