Evaluierung und Umsetzung von künstlichen neuronalen Netzen zur Generierung spielbarer Inhalte

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Konrad Mampe

2208749

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüfer: Prof. Dr.-Ing. Sabine Schumann

Zweitprüfer: Prof. Dr. habil. Kai von Luck

Hamburg, 30. 11. 2018

Inhaltsverzeichnis

Abstract

Procedural generation is a technique employed to automatically generate vast amounts of content in a short time, without any human interaction. In video games, this method is typically used to produce playable content i.e. levels. This generation mostly results in repetitive and artificial content, unless enough time is invested. Machine learning can counter this downside by creating content sampled from existing handmade examples, thus making more humane levels with less effort. But this combination of fields is novel and not widespread in the industry. Therefore, this thesis works on proving that procedural generation via machine learning is not only already capable of competing against the previously mentioned technique, but against handmade content as well. The proof is obtained by implementing a machine learning generator and evaluating it versus the established methods. Conclusively these types of generators are already capable of providing an alternative to the traditional methods.

Zusammenfassung

Prozedurale Generierung ist eine Technik, die in kurzer Zeit eine große Menge an Inhalten, ohne jegliche menschliche Interaktion, generieren kann. In Videospielen wird dieses Verfahren verwendet, um spielbare Inhalte, also Level, zu erstellen. Wenn nicht genug Zeit in die Umsetzung investiert wird, resultiert die Generierung in repetitiven und künstlichen Inhalten. Maschinelles Lernen kann diesem Nachteil entgegenwirken, indem es sich an handgefertigten Beispielen orientiert und dadurch einfacher menschlich wirkende Level schaffen kann. Diese Kombination von Bereichen ist relativ neu und in der Industrie nicht weit verbreitet. Daher wird in dieser Arbeit der Nachweis erbracht, dass prozedurale Generierung durch maschinelles Lernen nicht nur bereits mit der zuvor genannten Technik in Konkurrenz stehen kann, sondern auch mit handgemachten Inhalten. Diese Aussage wird bewiesen, indem ein Generator mit maschinellem Lernen implementiert und gegen die etablierten Methoden evaluiert wird. Schlussfolgernd ist diese Art von Generatoren in der Lage eine Alternative zu den traditionellen Methoden zu bieten.

1 Einleitung

Unerforschte Welten und neue Herausforderungen. Das sind einige der Aspekte, welche Menschen dazu bringen sich in Videospiele zu stürzen. Doch irgendwann ist die neue Welt erforscht und alle Herausforderungen bezwungen. Der Spieler ist fertig mit seinem Abenteuer und sucht sich ein neues. Das ist aber nicht im Interesse des Spieleherstellers, dieser möchte seine Kundschaft so lang wie möglich mit seinem Produkt fesseln. Um dieses Ziel zu erreichen, müssen dem Spieler aber mehr Inhalte zur Verfügung stehen und das ist ein zeit- und kostenaufwendiges Vorhaben. Dieses Vorhaben kann meistens nur von großen Studios bewältigt werden, kleinere Studios oder selbstständige Entwickler haben kaum eine Chance.

Um in diesem Markt mitzuhalten, verwenden viele Entwickler eine Technik namens prozedurale Generierung. Bei dieser Methode handelt es sich um die automatisierte Erzeugung von z.B. Umgebungen, Texten, Gegenständen und vielem mehr. Dieses Mittel wurde in den letzten Jahren immer populärer und hat vielen Entwicklern zum Erfolg verholfen. Diese Technik nimmt zwar weniger Zeit in Anspruch, als Inhalte händisch herzustellen, doch muss auch hier viel Zeit investiert werden, damit die generierten Inhalte sich nicht repetitiv und menschlich anfühlen.

Ein in letzter Zeit immer wieder auftauchender Begriff, der hier für Abhilfe sorgen kann, ist maschinelles Lernen. Maschinelles Lernen wird schon in vielen alltäglichen Bereichen verwendet, z.B. Routenplanung, Spamfilter und Autokorrektur. Auch in Spielen werden bereits verwandte Methoden angewandt, doch meistens nur im Zusammenhang mit Nicht-Spieler-Charakteren, um diesen menschliches bzw. intelligentes Verhalten beizubringen. Maschinelles Lernen ist nämlich dazu in der Lage anhand von menschlichen Beispielen zu lernen, diese zu reproduzieren und mit diesen Erkenntnissen eigene Lösungen zu finden. Bezieht man diesen Aspekt auf das vorhin genannte Problem, kann dieses beseitigt werden, da nun automatisch menschlichere Inhalte mit weniger Aufwand erstellt werden können. Dieser Themenbereich ist aber im Gegensatz zu der seit fast 30 Jahren etablierten prozeduralen Generierung noch relativ neu und unerforscht. Selbstverständlich gibt es schon einige Arbeiten, die sich mit diesem Bereich auseinandersetzen, doch ist dieser in der Industrie noch nicht weit verbreitet.

Zielsetzung

Dementsprechend beschäftigt sich diese Arbeit damit zu zeigen, dass maschinelles Lernen bereits mit algorithmischen Generatoren, aber auch mit handgemachten Inhalten, der manuellen Alternative, in Konkurrenz stehen kann. Der Beweis wird ermittelt, indem ein Generator mit maschinellem Lernen umgesetzt wird und gegen die etablierten Methoden evaluiert wird.

Struktur der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2 die Grundlagen der Thematik erläutert. Es wird darauf eingegangen, worum es sich bei prozeduraler Generierung handelt. In diesem Zusammenhang wird auf das Genre der Roguelike-Spiele eingegangen, welche insofern für das Thema relevant sind, da diese prozedurale Generierung als Hauptmerkmal besitzen.

In Kapitel 3 geht es um die Konzeption. Darin wird die Funktionsweise und einige der Besonderheiten von künstlichen neuronalen Netzen, welche hier Anwendung finden, erklärt. Anschließend wird noch auf verwandte Arbeiten eingegangen, um den Stand der Forschung darzulegen und zu zeigen, welche Möglichkeiten es noch gibt, um sich mit diesem Themenbereich zu befassen.

Danach folgt Kapitel 4 und handelt von der Umsetzung des Generators und der Levelgenerierung. Es wird gezeigt, wie aus einem Spiel Level zu Daten umgewandelt bzw. wie diese erzeugt werden können. Außerdem wird gezeigt, wie diese Daten im Generator verwendet und wie die daraus generierten Inhalte in ein Beispiel integriert werden.

Kapitel 5 legt mit der Evaluation den Grundstein für das letzte Kapitel. Hier wird der umgesetzte Generator zusammen mit einem algorithmischen Generator und handgemachten Inhalten nach vorbestimmten Metriken bewertet und untereinander verglichen. Außerdem wird eine Umfrage zu den Generatoren ausgewertet, um festzustellen, wie die verschiedenen Versionen von echten Spielern empfunden werden.

Kapitel 6 bezieht sich auf die Ergebnisse der Arbeit und stellt mit dem Fazit und der Zusammenfassung den Abschluss der Arbeit und der anfänglichen Forschungsfrage dar.

2 Grundlagen

Wie in Kapitel 1 erwähnt, untersucht diese Arbeit nicht die übliche Thematik der künstlichen Intelligenz in Videospielen, weshalb in den folgenden Kapiteln auf Besonderheiten eingegangen wird. Namentlich PCG für Procedural Content Generation und das Videospiel Genre der Roguelikes.

2.1 Procedural Content Generation

PCG stammt aus dem Obergriff Procedural Generation. Dieser steht für die Generierung von Inhalten nach programmierten Regeln. Dabei kann sich Procedural Generation auf alles beziehen, z.B. Musik, Grafiken und Texte. Die Spezifizierung PCG bezieht sich in Videospielen nur auf spielbare Inhalte, bspw. Gegner, Gegenstände und Umgebung. Das Besondere an PCG ist, dass die Ergebnisse dieser Generierung entweder zufällig oder pseudozufällig sind, sodass nicht zu hundert Prozent bestimmt werden kann, was generiert wird. Dieser Aspekt steht außerdem dafür, dass eine PCG-Software imstande ist, Inhalte ohne menschliche Hilfe zu erstellen [Shaker & Togelius & Nelson 2016]. Diese Technik bringt Vor- und Nachteile mit sich:

Vorteile	Nachteile
Früher: Kleinere Dateigrößen	Viele Tests sind erforderlich
Zeitsparend	Kein einzigartiges Gefühl
Hoher Wiederspielwert	Aufwendig
Verabschiedung vom klassischen Level Designer	

Tabelle 2.1: Vor- und Nachteile von PCG

Wie anhand von Tabelle 2.1 zu sehen ist, sind die Vor- und Nachteile von PCG relativ ausgewogen.

Als der Speicher von Datenträgern noch stark limitiert war und schnelle Downloads keine Option waren, wurde PCG verwendet, um viel Inhalt über kleine Datenträger zu vermitteln. Die Dateigrößen konnten sehr klein gehalten werden, da spielbare Inhalte, wie Level, nicht auf den Datenträgern gespeichert wurden, sondern erst während der Laufzeit generiert wurden. Dieser Punkt ist heutzutage aber nicht mehr relevant, da der Ursprung dieses Problems gelöst wurde. Nun bringt PCG andere Vorteile mit sich.

Sobald ein PCG-System erstellt wurde, geht keinerlei Zeit mehr verloren, um Level manuell zu erstellen. Insofern das System ordentlich funktioniert, besteht die Möglichkeit eine unendliche Anzahl an Inhalten in Sekunden zu generieren. Das ist zeitsparend und je nach Situation auch kostengünstiger.

Die Eigenschaft der unendlich generierbaren Inhalte sorgt für einen weiteren Vorteil, einen hohen Wiederspielwert. Dadurch dass nie wirklich vorherbestimmt werden kann, welcher Inhalt als nächstes generiert wird, kann sich der Spieler länger mit dem Spiel auseinandersetzen, da immer ein neues Spielerlebnis erzeugt wird.

Ein vollendetes PCG-System bringt viele Vorteile. So ein System fertigzustellen, ist aber je nach Komplexität sehr aufwendig. Je nachdem wie viele Regeln und Besonderheiten bei der Generierung eines Levels beachtet werden müssen, desto schwieriger wird es dieses System zu verwalten.

Das System muss außerdem ausgiebig getestet werden, da während der Laufzeit dem Spieler keine Level generiert werden sollen, die er nicht durch sein eigenes Können überwinden kann. Es muss dafür gesorgt werden, dass die generierten Inhalte spielbar bleiben.

Nichtsdestotrotz wird nach der Fertigstellung des Systems eine Menge Arbeit gespart. Ein Problem besteht hier jedoch auch, und zwar, dass die Level ggf. nicht sehr menschlich wirken. Damit ist gemeint, dass, durch die streng an Regeln gebundene Generierung, nur Inhalte in einem gewissen Rahmen generiert werden können, wodurch diese nicht einzigartig wirken.

Der letzte Punkt in der Tabelle spiegelt nur den Fakt wider, dass durch ein PCG-System Level Designer abgelöst werden können, wodurch einerseits Geld gespart wird, andererseits professioneller menschlicher Input verloren geht.

2.2 Roguelike

Das Spiel, welches zur Demonstration der Levelgenerierung verwendet wird, gehört zu dem Roguelike Genre. Spiele in diesem Genre zeichnen sich durch einige Merkmale aus, die sich gut für das Thema dieser Thesis eignen.

Abbildung 2.1: Das Testspiel

Roguelikes verwenden per Definition PCG, um Level zu generieren. Die Darstellung der Inhalte erfolgt durch Grafiken in Form von Kacheln und meistens in 2D. Es gibt noch weitere Merkmale, welche jedoch für die Arbeit nicht relevant sind [COLUMN: @Play: The Berlin Interpretation 2009]. Diese Aspekte werden in Kapitel 4.1 der Datenaufbereitung näher erläutert.

Das Spiel funktioniert folgendermaßen:

Der Spieler erscheint an einem Punkt auf einem zehn Mal zehn Schachbrett, dem Level. Das Ziel des Spielers ist es den Ausgang des Levels zu erreichen, gekennzeichnet durch ein grünes Exit-Schild. Sollte das Schild erreicht werden, startet ein neues Level. Von diesen Leveln möchte der Spieler so viele wie möglich beenden. Der Ausgang kann erreicht werden, indem die Spielfigur, der Charakter in der grünen Kapuzenjacke, per Betätigung der üblichen Bewegungstasten in die entsprechende Richtung bewegt wird. Dabei wird der Spieler pro Tastendruck um genau eine Kachel verschoben und verliert für jede Bewegung einen Lebenspunkt. Das Spiel endet, wenn die untenstehende Lebensanzeige des Spielers den Wert null erreicht. Auf dem

Weg zum Ausgang befinden sich ein paar Besonderheiten in Form von Gegenständen und Gegnern. Der Lebensanzeige können durch das Aufsammeln von Essen oder Trinken Punkte hinzugefügt werden. Dementsprechend ist es im Interesse des Spielers diese aufzusammeln, da dadurch die Lebenszeit verlängert wird. In einem Level befinden sich meistens auch Gegner, welche versuchen den Spieler daran zu hindern den Ausgang zu erreichen. Sollte der Spieler mit einem der Gegner, als Zombie dargestellt, in Kontakt treten, verliert dieser Lebenspunkte. Die Gegner bewegen sich abwechselnd mit dem Spieler ebenfalls pro Zug um eine Kachel. Ansonsten sind in einem Level immer Wände vorhanden, die der Spieler verwenden kann, um durch das Level zu manövrieren. Manche dieser Wände lassen sich bei Kontakt durch mehrmaliges Betätigen einer Bewegungstaste zerstören. Andere Wände, wie die am Rande des Spielfelds, können nicht zerstört werden.

Das Spiel wurde im Rahmen eines Tutorials von Unity mit deren gleichnamiger Engine erstellt [Unity 2015]. Dieses Spiel wurde dementsprechend nach Bedarf modifiziert.

3 Konzeption

Dieses Kapitel erläutert das Kernelement der Arbeit, und zwar künstliche neuronale Netze. Außerdem wird auf Arbeiten eingegangen, welche sich mit ähnlichen Themen beschäftigen.

3.1 Künstliches neuronales Netz

Ein Artificial Neural Network (ANN) bzw. künstliches neuronales Netz (KNN) ist ein Computersystem, welches im Entferntesten von der menschlichen Gehirnstruktur inspiriert wurde. Das Ziel war es ein System zu schaffen, das Probleme auf eine menschliche Art und Weise lösen kann. Also ein System, welches lernfähig ist, ohne für eine bestimmte Aufgabe programmiert worden zu sein. Ein Anwendungsbeispiel wäre z.B. die Kategorisierung von Bildern. Dabei kann das KNN trainiert werden, indem diesem vorgefertigte Trainingsdaten zur Verfügung gestellt werden. In dem Fall könnten verschieden gekennzeichnete Bilder verwendet werden, welche z.B. mit der Markierung „Hund“ oder „Nicht Hund“ beschriftet wurden. Dadurch kann das KNN für sich eigene Charakteristiken erstellen, mit denen es selbst neue und ungekennzeichnete Bilder einordnen kann [Géron 2018].

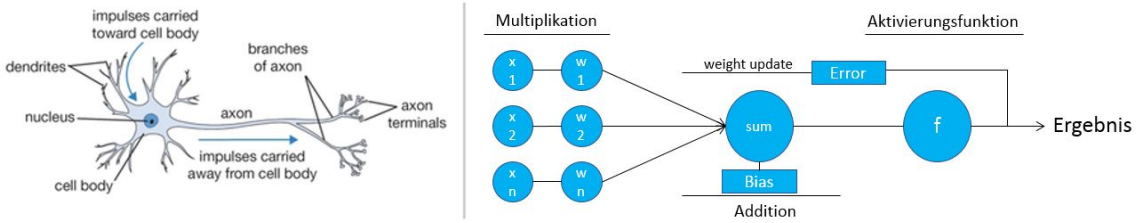


Abbildung 3.1: Biologisches Neuron gegen künstliches Neuron [Byl 2018]

Das menschliche Gehirn besteht aus einer großen Ansammlung einzelner Bausteine, darunter die Neuronen. Diese sind alle untereinander verbunden und bilden eine Signalkette. Signale werden aber nur weitergeleitet, wenn das jeweilige Neuron auch aktiviert wurde, z.B. wenn ein Finger bewegt werden soll, werden nur die dafür notwendigen Neuronen aktiviert. Der technische Aspekt kann folgendermaßen dargestellt werden, auch zu sehen in Abbildung 3.1:

Eingehende Signale werden durch die Eingänge, die Dendriten, aufgenommen. Diese Signale können als Zahlenwerte repräsentiert und als Input x bezeichnet werden.

Diese werden im nächsten Schritt im Kern eines Neurons, dem Nucleus, zu einem Gesamtinput sum verrechnet.

$$sum = \sum(w_1 * x_1 + w_2 * x_2 + w_3 * x_3) + b$$

Wie an der obenstehenden Formel zu sehen, wird diese Berechnung aber noch von zwei anderen Werten beeinflusst, namentlich dem Bias b und den Gewichten w. Die Gewichte bestimmen den Einfluss eines Eingangs auf die Gesamtrechnung. Der Bias ist hingegen unabhängig von den Eingängen und Gewichten. Dieser ist dafür verantwortlich, ob ein Neuron aktiviert wird oder nicht. Zur Aktivierung eines Neurons muss ein gewisser Schwellenwert überwunden werden. Dieser Schwellenwert gehört zu der sogenannten Aktivierungsfunktion, welche in Abbildung 3.1 der nächste Schritt in der Kette ist.

Solche Funktionen haben ihren namentlichen Ursprung in einer Eigenschaft der Neuronen. Die Neuronen leiten nicht jedes Signal weiter, dass sie erhalten. Nur wenn das Ergebnis der Berechnung einen Schwellenwert der Aktivierungsfunktion überschreitet, wird das Signal durchgelassen. Es besteht eine große Auswahl dieser Funktionen, welche zur Problemlösung verwendet werden können. Ein bekanntes Beispiel einer Aktivierungsfunktion wäre die „Binary step“ Funktion, welche folgendermaßen aussieht:

$$f(x) = \begin{cases} 0 & for \quad x < 0 \\ 1 & for \quad x \geq 0 \end{cases}$$

Falls das Ergebnis des Neurons falsch ist, wird die ganze Berechnung mit einer Änderung erneut ausgeführt. Vorerst wird die Differenz zwischen dem Ergebnis und dem Erwarteten berechnet und als Error Value bezeichnet. Nun werden die Gewichte angepasst, indem der Wert der Inputs mit der Error Value multipliziert wird. Der Bias ändert sich durch die simple Addition seines alten Wertes und der Error Value. Dieser Prozess wird so lange wiederholt, bis das erwartete Ergebnis berechnet wurde oder eine gewisse Anzahl an Durchläufen, Epoche, erreicht wurde. Korrekte Ergebnisse werden dementsprechend weiter gelassen und beeinflussen die Gewichte und den Bias nicht.

Alle Werte innerhalb eines Neurons werden normalisiert, um in einer vergleichbaren Reichweite zu liegen. Die Gewichte und der Bias können Anfangs einen zufälligen Wert annehmen, welcher zwischen dem Maximum und Minimum der normalisierten Eingangswerte liegt. Gewichte und Bias können jederzeit abgespeichert werden, um den Lernfortschritt festzuhalten und diese jederzeit abrufen zu können [Byl 2018] [Cleve & Lämmel 2016].

Ein KNN ist dann nur eine Verbindung beliebig vieler künstlicher Neuronen, die schichtweise strukturiert sind. Dementsprechend wandern die Eingangssignale von der ersten bis zur letzten Schicht. Dabei gibt es folgende Schichten:

1. Input Layer: Diese Schicht ist nur dafür da, um den Input an die nächste Schicht weiterzuleiten. Diese Schicht ist in einem System nur einmal vorhanden.
2. Hidden Layer: In dieser Schicht findet die eigentliche Berechnung durch die Neuronen statt. Von dieser Schicht kann es unendlich viele geben.
3. Output Layer: Im letzten Layer wird lediglich das Ergebnis ausgeben. Diese Schicht ist in einem System ebenfalls nur einmal vorhanden.

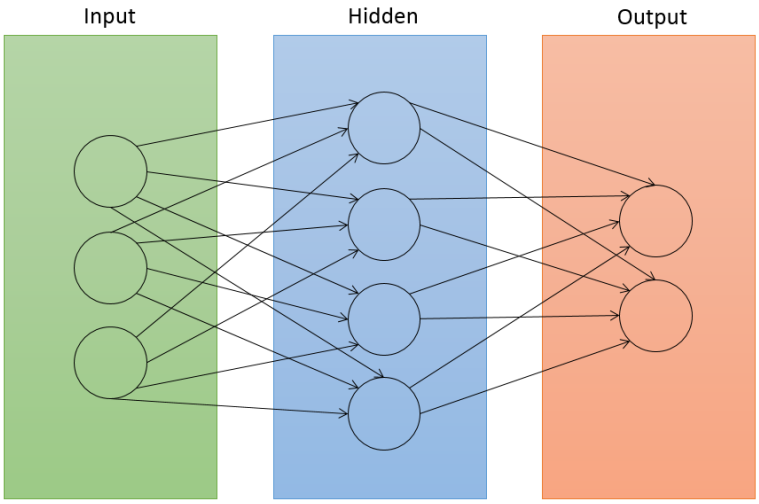


Abbildung 3.2: Die Schichten eines KNN

Ein vollständiges KNN funktioniert wie ein einzelnes Neuron mit der Besonderheit, dass wenn das Ergebnis falsch ist, alle vorhandenen Gewichte und Bias angepasst werden. Dieser Prozess wird als Back Propagation bezeichnet. Wie bei einem Neuron wird ein Durchlauf im KNN als Epoche bezeichnet, von denen beliebig viele ausgeführt werden können [Géron 2018] [Byl 2018].

3.1.1 Rekurrentes neuronales Netz

Ein rekurrentes neuronales Netz (RNN) ist eine Klasse der KNN. Das Besondere an einem RNN ist, dass die Verbindungen der Neuronen einen gerichteten Graphen bilden. Dadurch läuft ein RNN im Gegensatz zu einem KNN nicht nur von links nach rechts durch, sondern es besteht auch eine Verbindung von den Ausgängen zu dem Netzwerk selbst. In dem Fall erhält ein RNN nach jedem Schritt nicht nur die Eingangswerte, sondern auch den vorherigen Output. Also leitet ein Netzwerk eine Nachricht an seinen Nachfolger weiter.
Funktionstechnisch könnte dieser Aspekt so aussehen, wobei die Variablen die gleiche Bedeutung haben wie in den vorherigen Erklärungen mit der Ausnahme von y, welches das vorherige Ergebnis darstellen soll und T, welches für den jetzigen Schritt steht:

$$y_{(t)} = \phi({x_{(t)}}^T \times w_x + {y_{(t-1)}}^T \times w_y + b)$$

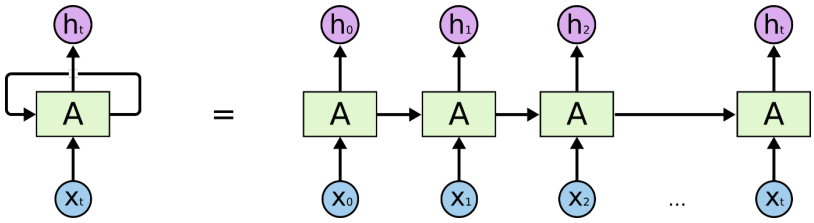


Abbildung 3.3: Darstellung eines RNN [Olah 2015]

Somit erhalten RNNs eine Art Gedächtnis, da nun Informationen von vorherigen Durchläufen in die Nächsten mitgenommen werden [Géron 2018].

Mit dieser Eigenschaft eignen sich RNNs gut, um Sequenzprognosen zu erstellen. Dieser Fakt kann dazu genutzt werden, um Musik zu verfassen oder Bilder zu generieren, da diese nichts anderes als Sequenzen sind. Das kann auch auf spielbare Inhalte übertragen werden, dazu mehr in Kapitel 4.1 der Datenaufbereitung.

Das für diese Arbeit verwendete RNN [Tay 2017] ist eine Umsetzung von Andrej Karpathys character-level multi-layer RNN [Karpathy 2015] in Python mit Googles TensorFlow Library.

3.1.2 Long Short-Term Memory

RNNs alleine sind zwar schon geeignet für Sequenzprognosen, jedoch leiden sie darunter, dass das Merken über größere Zeiträume mit ihnen nicht möglich ist. Falls z.B. ein Text übersetzt werden soll, reicht ein gewöhnliches RNN nicht aus, falls sich bestimmte Zusammenhänge über mehrere Absätze ziehen. Dieser Aspekt wurde aber durch einen zusätzlichen Baustein verbessert, der Long Short-Term Memory (LSTM) Einheit. So eine Einheit wird als Memory Cell bezeichnet, da diese dafür da ist sich Zusammenhänge über mehrere Iterationen zu merken. Ein LSTM besteht aus drei kleineren Bausteinen, einem Input Gate, einem Output Gate und einem Forget Gate. Diese Gates sind nichts anderes als eigene künstliche Neuronen, die dafür zuständig sind den Informationsfluss zu kontrollieren. Daher auch der englische Begriff Gate, für Tor [Olah 2015].

In dieser Arbeit wurde dementsprechend nicht nur ein gewöhnliches RNN verwendet, sondern ein LSTM-Netzwerk.

3.2 Verwandte Arbeiten

Maschinelles Lernen im Zusammenhang mit PCG zu verwenden, ist zwar nicht komplett neu, jedoch ist der Forschungsbeitrag im Vergleich zu üblichen PCG-Arbeiten relativ gering. Auch wenn der Forschungskorpus nicht zu groß ist, gibt es sehr gute

Arbeiten, die unter anderem Ähnlichkeiten mit dieser aufweisen. In diesem Kapitel werden drei Arbeiten vorgestellt, welche verschiedene Methoden oder Varianten verwenden, um maschinelles Lernen für PCG umzusetzen.

A Hierarchical MdMC Approach to 2D Video Game Map Generation

Die folgende Arbeit stammt von Sam Snodgrass und Santiago Ontanón [Snodgrass & Ontanón 2015]. In dieser Arbeit wird ein mathematisches System namens Markow-Kette (MC) verwendet. Bei dieser handelt es sich um ein System, dass die Wahrscheinlichkeiten festhalten kann, mit welchen ein Zustand in einen anderen wechselt.

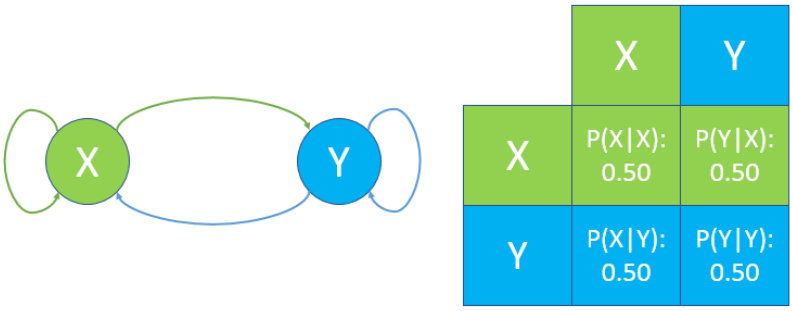


Abbildung 3.4: Darstellungen einer Markow-Kette

Abbildung 3.4 zeigt die übliche Darstellungsweise einer MC, wobei die Matrix eher verwendet wird als die Graphendarstellung. Wenn in diesem Fall ein neuer Zustand hinzugefügt wird, würde eine neue Zeile und eine neue Spalte zur Matrix hinzukommen. Bei der Graphendarstellung wären es fünf Pfeile. Markow-Ketten wachsen nämlich quadratisch, dementsprechend können diese mit nur wenig Zuständen sehr groß werden. Eine Graphendarstellung wäre dadurch schnell unübersichtlich [Powell 2014].

Doch wurden hier nicht nur normale MC verwendet, sondern mehrdimensionale Markow-Ketten (MdMC). Bei MdMC, im Gegensatz zu gewöhnlichen MC, sind mehrere Ketten vorhanden, welche miteinander verbunden sind und dadurch Informationen zu einer größeren Kette beisteuern können.

In diesem Experiment haben die Autoren sich eine bereits bestehende Menge an Super Mario Bros. Leveln zunutze gemacht, um neue Level zu generieren. Mit dieser Menge konnten Sie Ihrem System die Wahrscheinlichkeiten beibringen, mit welchem ein Level Baustein auf einen anderen folgt. Dabei wurden die Level Bausteine aufgeteilt. Vorerst wird ein Level mit großen Bausteinen generiert, diese werden als groß bezeichnet, da sie mehrere kleine Bausteine beinhalten. Danach wird die innere Struktur eines großen Bausteins bestimmt, also die Anordnung der kleinen Bausteine.

Herausgestellt hat sich, dass andere Methoden bessere Ergebnisse in der Generierung erzielt haben, da hier die Anzahl an spielbaren Leveln geringer war. Das Besondere ist aber, dass mehrdimensionale Markow-Ketten, ohne viel zu modifizieren, auf andere Spiele angewendet werden können, um Level zu generieren. Getestet wurde dieser Fakt an einem Spiel namens Loadrunner, welches wie Mario ein Plattformspiel ist. Der Unterschied ist aber, dass Loadrunner sich nicht nur von links nach rechts spielt, sondern auch von oben nach unten. Dementsprechend sind die Verhältnisse der Bausteine in Loadrunner komplexer als in Super Mario. Die Menge an spielbaren Leveln blieb gering, jedoch wurden welche erstellt.

Linear levels through n-grams

Diese Arbeit wurde von Steve Dahlskog, Julian Togelius und Mark J. Nelson verfasst und befasst sich mit der Generierung von linearen Leveln mit der Hilfe von N-Grammen [Dahlskog & Togelius & Nelson 2014]. N-Gramme sind eine Technik, welche hauptsächlich in Natural-language processing und Text Mining angewendet wird. Ein N-Gramm ist ein Paar von vorkommenden Worten in einem gegebenen Fenster in einer Wortsequenz. Bildlicher ist dies einfacher zu verstehen:

Beispielsatz: "Was ist ein N-Gramm?"		
Monogramm (N=1)	Bigramm (N=2)	Trigramm (N=3)
Was	Was ist	Was ist ein
ist	ist ein	ist ein N-Gramm
ein	ein N-Gramm	
N-Gramm		

Tabelle 3.1: N-Gramme

Der Wert N bestimmt die Größe des N-Gramms, bei einem Trigramm wären es z.B. drei Worte. Pro N-Gramm wird ein Wort vorwärts gerückt, um das nächste zu erhalten. N kann beliebig groß werden und wird dann als Multigramm bezeichnet. Mit N-Grammen werden aus einer Sammlung von Strings Wahrscheinlichkeitstabellen gebildet, welche Werte für die Wahrscheinlichkeiten von aufeinanderfolgenden Worten beinhalten. Diese werden verwendet, um neue Strings zu generieren [Jurafsky & Martin 2014].

Die Experimente wurden mithilfe von Leveln aus dem Spiel Super Mario Bros. durchgeführt. Die Level wurden aber nicht als eine Zusammensetzung einzelner Kacheln betrachtet, sondern als eine Sequenz von Spalten. Dadurch konnte ein ganzes Level als ein einzelner von links nach rechts lesbarer String dargestellt werden. Das ermöglichte die Verwendung von N-Grammen, wie es auch in der Text- und Musik Generierung auffindbar ist. Diese Methode soll schnell und simpel sein, aber auch ordentlich spielbare Inhalte produzieren. Das Ziel dieser Arbeit war es Level zu generieren, die

stilistisch so nah wie möglich an das Original rankommen. Wenn N-Gramme anhand der vorher genannten Level erstellt werden, wird die Wahrscheinlichkeitstabelle aller vertikalen Bausteine des Originals in Abhängigkeit ihrer Vorgänger erstellt. Diese Wahrscheinlichkeiten können dann als Gewichte im Generierungsprozess verwendet werden. Für dieses Vorhaben wurden Wahrscheinlichkeitstabellen für Monogramme, Bigramme und Trigramme, aus den Original-Leveln erstellt. Bei einem Monogramm entspricht die Wahrscheinlichkeit jedes Bausteins der Häufigkeit gegenüber allen anderen Bausteinen. Bei einem Bigramm sind die Wahrscheinlichkeiten von einem Baustein vom vorherigen abhängig. Und bei einem Trigramm dementsprechend von zwei vorherigen Bausteinen.

Es stellte sich heraus, dass verschiedene Werte für N sehr verschiedene Level generieren. Monogramme geben zufällige Strukturen ohne jegliche Zusammenhänge aus, was auf die Unabhängigkeit von anderen Stücken zurückzuführen ist. Bigramme funktionieren besser, jedoch werden viele Segmente wiederholt. Trigramme waren am besten im Sinne der ursprünglichen Fragestellung. Die Wahrscheinlichkeitswerte aller vertikalen Bausteine in durch Trigramme erstellten Leveln waren am ähnlichsten zu den Werten der originalen Inhalte. Dementsprechend erfüllte die angewandte Methodik die ursprünglichen Kriterien, stilistisch ähnliche Inhalte zu generieren.

Super Mario as a String: Platformer Level Generation Via LSTMs

In der folgenden Arbeit wird ein LSTM-Netzwerk verwendet, um spielbare Inhalte zu generieren. Wie in den vorherigen Arbeiten wird sich ebenfalls auf Super Mario bezogen. Das Besondere an dieser Arbeit ist, dass sich stark auf ein Spiel konzentriert wird, um so gut wie möglich viele spielbare Inhalte erstellen zu können. Auf LSTMs wurde bereits in Kapitel 2 eingegangen, weshalb die Grundlagen an dieser Stelle nicht erneut erklärt werden, diese können bei Bedarf in dem entsprechendem Kapitel nachgeschlagen werden. Dementsprechend wird hier nur auf die Besonderheiten eingegangen. Die Arbeit stammt von Adam J. Summerville und Michael Mateas [Summerville & Mateas 2016]. Die Autoren wendeten folgende Techniken an, um Ihr Ziel zu erreichen:

- Snaking: Mit Snaking wurde eine Technik zum Einlesen und Bauen der Level bezeichnet. Üblicherweise werden Strings von links nach rechts und String für String gelesen und generiert. Beim Snaking werden die Daten Spalte für Spalte von oben nach unten nach oben oder umgekehrt verarbeitet. Gezeichnet würde das wie eine Schlange aussehen, welche sich durch das Level zieht, daher der Begriff Snaking. Da Super Mario Level sich eher in die Breite als in die Höhe ziehen, sorgt der Prozess dafür, dass ein sinniger Zusammenhang zwischen einzelnen Kacheln besteht. Außerdem verdoppelt diese Technik beim Einlesen der Daten die Anzahl an Trainingsmaterial, da diese von der einen und der anderen Seite unterschiedlich dargestellt werden.

- Path Information: Wie der Name schon deutet, handelt es sich um die Weginformationen des Spielers. Speziell ist hier der Pfad gemeint, welchen der Spieler verwenden kann, um ein Level zu beenden. Diese Pfade werden dann als Informationen zusammen mit den Trainingsleveln übergeben. Dieser Aspekt sorgt dafür, dass mit einer höheren Wahrscheinlichkeit Level generiert werden, die auch spielbar sind. Das liegt daran, dass eben nicht nur die Geometrie des Levels beim Generieren beachtet wird, sondern auch ein möglicher Weg. Diese Informationen müssen aber nachgereicht werden, da solche Daten in Leveln normalerweise nicht vorhanden sind. Ein anderer Vorteil ist aber, dass es in Super Mario immer verschiedene Wege gibt, um das Ende eines Levels zu erreichen. Das sorgt dafür, dass auch hier das Trainingsmaterial vergrößert werden kann, indem mehrere Pfade für Level beachtet werden.

- Column Depth: In diesem Punkt ging es darum, dass der Generator einen sich steigernden Schwierigkeitsgrad verwendet, welcher das Level zum Ende hin geometrisch schwieriger aufbaut als am Anfang. Das würde die klassische Schwierigkeitsänderung in Spielen widerspiegeln. Das LSTM war in der Lage sich an 200 vergangene Kacheln zu erinnern. Umgerechnet wären das ungefähr 12 Spalten. Um einen Schwierigkeitsgrad einzuführen, wurde ein Wert verwendet, welcher alle fünf Spalten erhöht wird. Das würde bedeuten, dass bei den Säulen null bis vier der Wert noch null beträgt, bei fünf bis neun wäre der Wert auf eins und bei zehn bis vierzehn dann auf zwei. Durch diesen Wert konnte die Schwierigkeitsstufe und der Levelfortschritt festgehalten werden. Somit konnte dem Generator befohlen werden, abhängig vom Wert komplexere Muster zu generieren.

Anschließend erstellte der erweiterte Generator eine Menge an spielbaren Inhalten. Diese wurden dann mit Spiel spezifischen Metriken analysiert. Dabei stellte sich heraus, dass die Einführung der Path Information sich am positivsten auf die Spielbarkeit der Inhalte auswirkte. Die anderen Techniken sorgten dafür, dass die spielbaren Level auch interessant wirkten.

4 Umsetzung

Im folgenden Kapitel wird maschinelles Lernen mit Procedural Content Generation verknüpft, um spielbare Inhalte zu generieren. Das Ziel ist ein Programm zu erstellen, welches spielbare Level erstellt, indem es an einer vorhandenen Menge an Beispielen lernt.

4.1 Datenaufbereitung

Alle in Kapitel 3.2 erwähnten Arbeiten nutzen für Ihre Forschung Inhalte aus Super Mario. Das in dieser Arbeit verwendete Spiel besitzt aber keine vorgefertigten Inhalte, da diese erst zur Laufzeit generiert werden. Dementsprechend handelt dieses Kapitel davon, wie aus diesem Spiel Inhalte übersetzt und erstellt werden können, damit diese für ein RNN brauchbar werden.

4.1.1 Datenrepräsentation

Wie in Kapitel 2.2 erwähnt, besitzt das hier verwendete Spiel Eigenschaften, welche sich gut dazu eignen mit einem RNN Level zu generieren. Und zwar handelt es sich bei diesem Spiel um ein 2D Spiel, welches sich aus vielen einzelnen Kacheln zusammensetzt und dessen Spielfläche im Gegensatz zu anderen Spielen mit lediglich 100 Kacheln pro Level verhältnismäßig gering ist. Aus Kapitel 3.1 konnte entnommen werden, dass sich RNNs gut für Sequenzprognosen eignen. Deshalb müssen die Inhalte des Spiels so umgewandelt werden, dass diese in den Bereich der Sequenzprognosen fallen. Letztendlich handelt es sich bei diesen Leveln aber auch nur um eine Sequenz von Kacheln. Dies kann folgendermaßen wie ein Text von links nach rechts gelesen werden:

Reihe 1: Kachel 1, Kachel 1, Kachel 1, ..., Kachel 1

Reihe 2: Kachel 1, Kachel 3, Kachel 4, ..., Kachel 1

...

Reihe 10: Kachel 1, Kachel 1, Kachel 1, ..., Kachel 1

Abbildung 4.1: Rudimentäre Textform eines Levels

Durch diese Darstellung kann ein Level als eine Sequenz von Kacheln beschrieben werden. Für ein vollständiges Level müssen zehn Reihen mit jeweils zehn Kacheln vorhanden sein. Um diese Beschreibung zu vereinfachen, wird jeder Kachel bzw. jedem Baustein ein Buchstabe zugeordnet, welcher diesen in Textform repräsentiert.

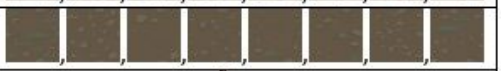
Alphabet	
Charakter	Kachel
X (Unzerstörbare Wand)	
Y (Zerstörbare Wand)	
- (Boden)	
O (Nahrung)	
Z (Gegner)	
E (Ausgang)	
I (Eingang)	

Abbildung 4.2: Charakter-Kachel Alphabet

Im Fall dieses Spiels ist es möglich noch weiter zu vereinfachen, da ein Charakter mehrere Kacheln repräsentieren kann. Das ist möglich, weil viele Kacheln untereinander keine spielerischen Unterschiede besitzen, sondern sich nur ästhetisch voneinander unterscheiden. Dieser Aspekt sorgt aber auch dafür, dass, wenn sich Level strukturtechnisch ähneln, diese durch ihre Ästhetik trotzdem abwechslungsreich wirken. In Abbildung 4.3 ist eines dieser Level mit entsprechender Textrepräsentation zu sehen:


```
XXXXXXXXXX
X----E---X
X-O---YY-X
X-----YYYX
X--------X
X--Z-----X
X--------X
X--YYY---X
X---Y---IX
XXXXXXXXXX
```

Abbildung 4.3: Leveldarstellungen

Mit der in Abbildung 4.3 zu sehenden Textform kann das RNN nun arbeiten. Bevor aber Berechnungen gestartet werden können, wird im RNN noch eine weitere Umwandlung durchgeführt. Und zwar zieht sich das Netz alle Charaktere aus den übergebenen Beispielen und baut sich daraus ein Alphabet, ähnlich wie in Abbildung 4.2 zu sehen. Diese Charaktere werden dann mittels One-Hot-Kodierung als Vektoren dargestellt, zu sehen in Abbildung 4.4. Somit werden die Ergebnisse auch in Vektoren dargestellt, diese beinhalten dann die Gewichtung, bzw. die Wahrscheinlichkeit, eines folgenden Charakters in Abhängigkeit seiner Vorgänger. Die grün markierten Werte in Abbildung 4.4 stellen die Werte dar, welche im nächsten Durchlauf erhöht werden sollen. Die roten Werte zeigen wiederum die Werte, die im nächsten Durchlauf gesenkt werden sollen. Dementsprechend ist es bereits am Anfang möglich einzelne Bausteine zu generieren und somit auch vollständige Level. Diese Level können dann mit den übergebenen Trainingsdaten verglichen werden, um die Anpassung der Gewichte für den nächsten Durchlauf zu bestimmen.

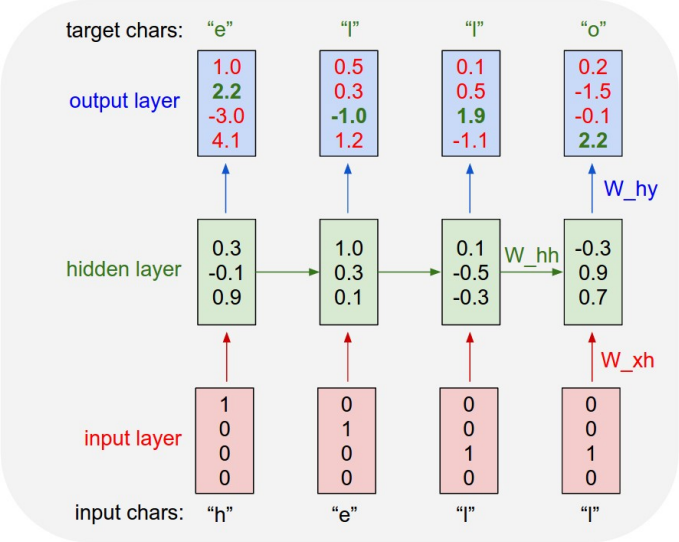


Abbildung 4.4: Das RNN von Innen (Vokabular: "h, e, l, o") [Karpathy 2015]

In dieser Arbeit wird ein gebrauchsfertiges LSTM verwendet, nachzulesen in Kapitel 3.1. Das hat einerseits den Grund, dass in dieser Arbeit verschiedene Generierungen verglichen werden, dementsprechend müssen diese einen vergleichbaren Zeitaufwand besitzen. Andererseits besitzt das Spiel einen geringen Grad an Komplexität, weshalb eine an das Problem unangepasste Lösung ausreichend ist.

4.1.2 Datenerzeugung

Bevor neue spielbare Inhalte generiert werden konnten, musste ein LSTM-Netzwerk gefunden werden, welches diesen Anspruch erfüllen kann. Um ein solches Netzwerk zu finden, sind aber bereits Daten notwendig, mit welchen es getestet werden kann. Das hier verwendete Spiel stammt aus dem Roguelike Genre, weshalb eine Menge an Leveln üblicherweise nicht vorhanden ist. Weil die manuelle Anfertigung von Leveln aufwendig ist, wurde das Spiel modifiziert, um Level in entsprechender Textform auszugeben. Dabei handelt es sich um Daten, die dem Basisgenerator des Spiels entsprechen, diese sind nicht komplex, da sie nach festen und simplen Mustern generiert werden, zu Testzwecken sind diese Daten ausreichend. Anhand dieser Daten konnten brauchbare LSTMs ausfindig gemacht werden. Diese Daten werden aber nicht beim späteren Training des Netzwerks verwendet, weil die generierten Inhalte ansonsten mit den Inhalten des Basisgenerators gleichzusetzen sind. Zum Training des LSTMs mussten manuell Level erstellt werden. Der Vorteil von LSTMs an dieser Stelle ist, dass die Regeln der Generierung nicht definiert werden müssen, sondern einfach eine gewisse Anzahl an Beispielen geliefert werden muss, damit sich das Netzwerk die Regeln der Daten selbst beibringen kann. An dieser Stelle sollte gleichviel Arbeit in die Aufbereitung der Level fließen wie in die Erstellung des Grundspiels, da die Generatoren sonst nicht vergleichbar sind.

Bei der manuellen Erstellung der Level können bereits diverse Faktoren bestimmt werden, welche beeinflussen, wie die generierten Level im Hinblick auf Schwierigkeitsgrad, Spielbarkeit und Ästhetik aussehen werden.

Die Spielbarkeit erschließt sich daraus, dass ein Level beendet werden kann. Also, dass das Ende ohne Fehler erreicht werden kann. Der Spieler soll nicht von Wänden blockiert werden oder von zu vielen Gegnern umzingelt sein. Alle zum Training übergebenen Daten müssen spielbar sein, da diese sonst dafür sorgen können, dass auch nicht spielbare Level generiert werden. Die Mindestanforderung für ein spielbares Level ist, dass ein Eingang, ein Ausgang und die Außenwände vorhanden sind.

Der Schwierigkeitsgrad wird durch die Anzahl an Gegnern und den verschiedenen Wänden bestimmt. Eine hohe Anzahl an zerstörbaren Wänden macht es leichter sich durch ein Level zu bewegen. Eine hohe Anzahl an Gegnern macht dies wiederum schwerer. Dementsprechend sind Level kompliziert, falls diese viele Gegner und wenige Wände aufweisen und leichter, wenn diese weniger Gegner und mehr Wände besitzen. Das Vorhandensein von Nahrung macht ein Level immer leichter, da ein Gegner oder eine Wand nicht auf der gleichen Kachel wie Nahrung erscheinen kann und diese nur positive Effekte mit sich bringt. In Betracht dieser Faktoren können Inhalte mit verschiedenen Schwierigkeitsgraden erstellt werden.

Die Ästhetik betreffend gilt, je komplexer und regelloser die Level aufgebaut sind, desto schwerer lernt das RNN. Wenn dementsprechend in den Daten Regeln klar strukturiert sind, lernt das RNN schneller und mit weniger Daten. Aber um für eine hohe Varietät zwischen Inhalten zu sorgen, sollten viele sehr verschiedene Level erstellt werden, auch wenn das Training dadurch länger dauert oder mehr Daten erfordert. Dieser Aspekt sollte auch genutzt werden, da die Regeln zum Aufbauen der Level nicht selbst programmiert werden müssen, wodurch mehr Möglichkeiten entstehen Level zu gestalten. Ein Beispiel dafür ist in Abbildung 4.5 zu sehen.

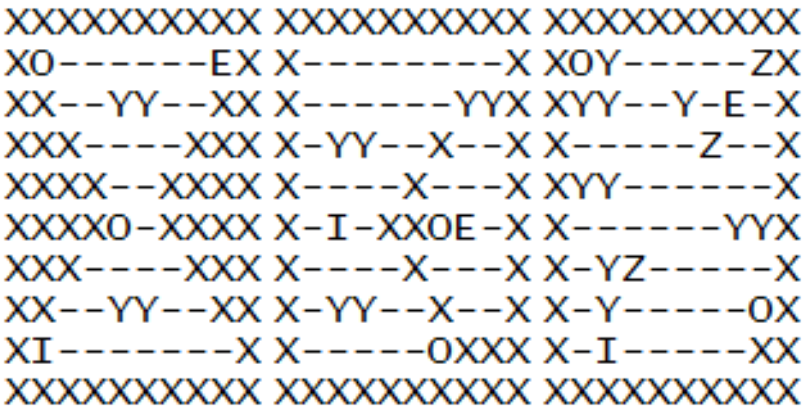


Abbildung 4.5: Handgemachte Inhalte

Diese manuell erstellten Level sollten vor dem Training auf die vorher genannten Faktoren überprüft werden, da fehlerhafte Daten die generierten Level sonst negativ beeinflussen können.

4.2 Training

Mit der Erzeugung der Daten ist der Großteil der aktiven Arbeit erledigt, da es nun an der Maschine liegt sich zu trainieren und das nimmt nur Zeit in Anspruch. In diesem Aspekt werden bestimmte Parameter verwendet, die sich unter anderem darauf auswirken, wie erfolgreich ein kompletter Trainingsdurchlauf ist. In diesem Kapitel wird auf diese Parameter, verwendete Trainingsdaten und die Integration der generierten Inhalte ins Spiel eingegangen.

4.2.1 Parameter

Die folgenden Parameter, mit ihren respektiven Werten, welche durch das Experimentieren herausgefunden wurden, erwiesen sich als erfolgreich:

- Sequenzlänge: Die Sequenzlänge bestimmt den Bereich, den das LSTM in diesem Fall betrachten soll. Wie in 4.1.1 erwähnt, besteht ein Level aus 100 Kacheln, das entspricht einer Sequenz von 100 Zeichen. In der Textrepräsentation kommt noch ein Zeichen an das Ende jeder Reihe, welches das Ende dieser darstellt und noch ein Zeichen in eine leere Reihe nach dem Level, welches das Ende des Levels darstellt. Dementsprechend besteht ein Level aus 111 Zeichen. Dieser Wert wird somit als Sequenzlänge übergeben. Dadurch kann das LSTM ein Level von Anfang bis zum Ende in Betracht ziehen.

- RNN Größe und Schichtenanzahl: Zum Generieren der Level wurde ein RNN verwendet, welches neben der Eingabe- und Ausgabeschicht aus zwei Zwischenschichten besteht, wobei jede Zwischenschicht 512 Neuronen beinhaltet. Diese Art von Netzwerk wird als weites Netzwerk bezeichnet, ein Netzwerk mit mehreren Schichten wiederum als tiefes Netzwerk. In der Regel reichen zwei Schichten bereits aus um, Lösungen für diverse Probleme zu finden. Die Generierung von Leveln als Sequenz ist davon nicht ausgeschlossen.

- Dropout Rate: Bei Dropout handelt es sich um eine Regulierungsmethode, bei welcher Eingänge und Verbindungen in einem LSTM-Block durch eine bestimmte Wahrscheinlichkeit ausgeschlossen werden, in die Berechnungen und die Gewichtsanpassung, während des Trainings einzufließen. Dieser Parameter wird verwendet, um ein Problem zu beseitigen, bei welchem sich das LSTM über längere Zeit verbessert, sich ab einem gewissen Punkt aber wieder verschlechtert. Die hier verwendete Dropout Rate liegt bei dem Wert 0.1.

- Learning Rate: Bei der Learning Rate handelt es sich um eine Funktion, welche mit der Zeit abnimmt. Sie ist dafür verantwortlich, wie schnell ein LSTM etwas Gelerntes wieder vergisst und durch etwas Neues ersetzt. Die Learning Rate beträgt in dieser Arbeit den Wert 0.0001.
- Anzahl der Epochs: Die hier gewählte Anzahl an Epochs beträgt 2500, nach dieser Einheit konnten keine Verbesserungen in der Generierung festgestellt werden. Um was es sich bei einem Epoch handelt, kann in Kapitel 2.3 nachgelesen werden.

4.2.2 Trainingsdaten

Die in Abbildung 4.6 zu sehenden Level sind Teil der zum Training verwendeten Daten. Diese wurden entsprechend der erwähnten Faktoren in Kapitel 4.1.2 manuell erstellt. Die Level sind alle sehr verschieden voneinander. Manche Level beinhalten keine bis drei Gegner, keine bis vier Nahrungsgegenstände, wenige oder viele Wände und die Ein- und Ausgänge, welche sich immer an unterschiedlichen Positionen befinden. Durch diese Unterschiede wurden Level generiert, die alle unterschiedliche Schwierigkeitsstufen aufweisen. Diese mussten nach der Generierung im Spiel mit einem sich steigernden Schwierigkeitsgrad in Reihenfolge gebracht werden. Insgesamt wurden 250 Level zum Trainieren des Models verwendet. Im Folgenden werden Level aus den Trainingsdaten analysiert, um deren Besonderheiten hervorzuheben:

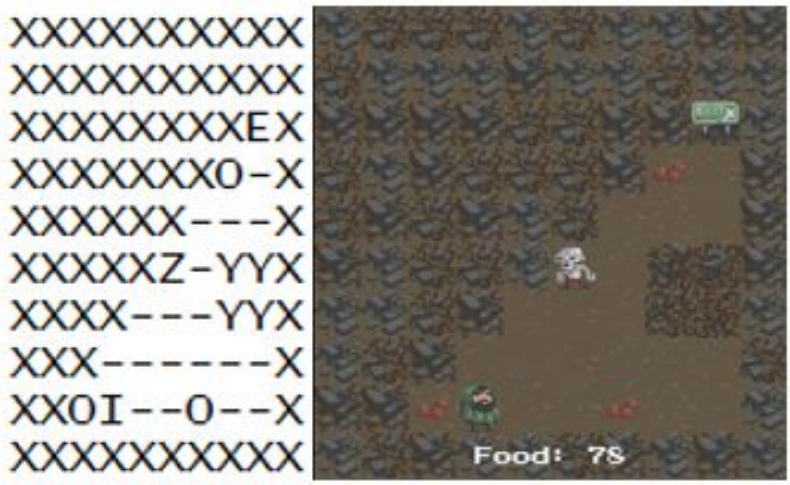


Abbildung 4.6: Level mit hoher Anzahl an unzerstörbaren Wänden

Das Spielfeld im Spiel besteht aus einer Fläche von zehn Mal zehn Kacheln. Dabei sind die äußeren Wände immer unzerstörbar, damit der Charakter das Level nicht verlassen kann. Um eine noch größere Variation an Leveln zu generieren, können die unzerstörbaren Wände (X) benutzt werden, z.B. indem das Level verkleinert oder eine Art Labyrinth generiert wird. Die Wände X an dem äußeren Rand der Fläche zu generieren, ist keine große Herausforderung. Falls diese Wände auch auf dem Spielfeld verwendet werden sollen, muss dafür gesorgt sein, dass diese den Spieler am Beenden des Levels nicht hindern, da das Spielfeld sonst als nicht spielbar gilt. Also wäre eine Art von Überprüfung notwendig. Dies wäre schon eine größere Herausforderung. Aber durch ein LSTM, welches sich an vorgefertigten Leveln orientiert, fällt dieser Aspekt weg. Es muss lediglich dafür gesorgt werden, dass in jedem Trainingslevel die Wände X den Spieler niemals davon abhalten ein Level zu beenden, wodurch das Netzwerk auch nahezu nie Level generiert, welche so aufgebaut sind.

Abbildung 4.7: Eingang und Ausgang an verschiedensten Positionen

Die Ein- und Ausgänge sind mit den unzerstörbaren Wänden die grundlegendsten Bestandteile, damit ein Level funktioniert. Falls diese zu nah aneinander platziert sind, ist das Level zu einfach und schnell zu bewältigen und wenn diese weiter voneinander entfernt sind, muss gewährleistet sein, dass diese auch erreichbar sind. Umsetzungstechnisch müsste eine entsprechende Überprüfung eingebaut werden. Im Fall von maschinellem Lernen kann bereits bei der Erstellung und Kontrollierung der Trainingsdaten festgestellt werden, ob die Verhältnisse zwischen Ein- und Ausgang korrekt sind. Dadurch kann sich das Netzwerk selbst Regeln zum korrekten Setzen dieser Objekte erstellen.

Abbildung 4.8: Strukturen in einem Level

Durch das manuelle Erstellen der Trainingsdaten konnten zahlreiche Strukturen, ohne jegliches Programmieren entwickelt und eingebaut werden. Das wäre natürlich auch in gewöhnlicher PCG möglich, bei maschinellem Lernen besteht jedoch noch die Möglichkeit, dass diese Strukturen verändert oder mit anderen kombiniert werden, wodurch mehrere verschiedene Strukturen generiert werden können, als ursprünglich übergeben wurden.

Sonstige Bausteine, wie zerstörbare Wände, Gegner und Nahrung, weisen keine Besonderheiten auf. Diese sind lediglich in einer gewissen Anzahl in jedem Level vorhanden, damit diese auch in die generierten Level miteinbezogen werden.

Ein Durchlauf im Training verteilt mittels Eingabeschicht alle vorhandenen Charaktere aus den Trainingsdaten an die Neuronen der ersten Zwischenschicht. Diese leitet ihre Ergebnisse an die zweite Zwischenschicht, welche abschließend dessen Ergebnisse an die Ausgabeschicht weiterleitet. Somit wird zum Ende jedes Durchlaufs eine Menge von 512 Zeichen ausgegeben, welche nach jedem Durchlauf immer mehr die Form funktionierender Level annehmen.

4.3 Integration

Nach einem erfolgreichen Training und der anschließenden Generierung sind viele spielbare Level vorhanden, welche nur noch in das Spiel eingebaut werden müssen. Normalerweise werden Level im Spiel generiert, sobald das vorherige abgeschlossen wurde, quasi sequenziell. In diesem Fall sind die Level vorher generiert worden. Das hat einerseits den Grund, dass der hier angewandte Machine Learning Generator keine 100-prozentige Wahrscheinlichkeit besitzt, spielbare Level zu generieren. Diese Wahrscheinlichkeit kann erreicht werden, jedoch müsste dafür der Generator spezi-

ell auf ein bestimmtes Spiel angepasst und ausgiebig getestet werden. Das ist hier nicht der Fall, da ein bereits fertiges neuronales Netz zur Levelgenerierung verwendet wurde. Andererseits, selbst wenn die Annahme besteht, dass der Generator eine 100-prozentige Wahrscheinlichkeit besitzt, korrekte Level zu generieren, ist das vermutlich nicht der Fall. Das liegt eben daran, dass nicht genau festgestellt werden kann, was für einen Lösungsalgorithmus sich die Maschine beigebracht hat. Und insofern die inneren Mechanismen der Maschine nicht begutachten werden können, kann die Behauptung nicht bestehen, dass dieser auch immer funktioniert. Dementsprechend findet die Generierung der Level nicht während der Laufzeit des Spiels statt.

Die Level müssen deshalb nach dem Schwierigkeitsgrad sortiert werden und als Dateien im Spielverzeichnis vorhanden sein. Das Spiel liest anschließend nach jedem erfolgreichen Level die nächste Leveldatei ein, um mithilfe der Levelübersetzung aus Kapitel 4.1.1 diese in ein spielbares Level umzusetzen. Das passiert solange, bis keine Level mehr vorhanden sind.

Um die in Kapitel 5 stattgefundene Evaluation zu vereinfachen, wurden drei verschiedene Versionen des Spiels angefertigt. Bei Version Eins handelt es sich um das Spiel mit einer gewöhnlich algorithmischen prozeduralen Generierung, Version Zwei beinhaltet die vom maschinellem Lernen generierten Level und Version Drei benutzt die Trainingsdaten. Diese Versionen wurden zum schnellen Zugriff auf einer Website gehostet. Diese ist im Anhang unter jonskony.github.io erreichbar.

Abbildung 4.9: Startseite der Website mit den drei Versionen

5 Evaluation

Die generierten Inhalte aus Kapitel 4 werden in diesem Kapitel anhand von vorbestimmten Metriken bewertet und mit der ursprünglichen Trainingsmenge und den Inhalten aus dem Ursprungsspiel verglichen. Dadurch konnte festgestellt werden, ob die Generation der Inhalte erfolgreich war. Anhand einer Umfrage wird im Anschluss ausgewertet, wie diese Inhalte von echten Spielern empfunden wurden.

5.1 Metriken

Zuvor muss überprüft werden, ob der Generator auch nur spielbare Level ausgegeben hat, da beim maschinellen Lernen keine 100%ige Sicherheit besteht, dass die gegebenen Resultate auch komplett korrekt sind. Die Spielbarkeit der Inhalte kann durch simples Begutachten gewährleistet werden. Es sollte lediglich darauf geachtet werden, ob eine Verbindung zwischen Ein- und Ausgängen besteht und ob das Level ordentlich durch unzerstörbare Wände begrenzt ist. Durch die Erfüllung dieser Aspekte gilt ein Level bereits als spielbar. Dieser Vorgang kann auch von einem Filterprogramm übernommen werden, aber durch die Erstellung so eines Programms ist schon viel Vorarbeit für einen herkömmlichen Generator nötig und das ist nicht das Ziel dieser Arbeit. Nach diesem Prozess sind nur noch spielbare Level vorhanden.

Der Erfolg des Generationsvorgangs kann nun anhand von zwei Metriken bewertet werden, der Dichte des Levels und dem berechneten Schwierigkeitsgrad. Diese Metriken erwiesen sich als am genausten, um ein Level zu beschreiben.

Bei der Dichte handelt es sich um das Verhältnis von begehbaren zu nicht begehbaren Elementen innerhalb der Spielfläche. Die Dichte ist dementsprechend der Faktor, welcher bestimmt, wie offen oder unzugänglich ein Level ist. Die Berechnung der Dichte ist die Division aus der Anzahl an unbegehbaren Elementen und der Größe der Spielfläche. Bei der Größe der Spielfläche handelt es sich um eine konstante Variable mit dem Wert 64. Bei den unbegehbaren Elementen handelt es sich um die Bausteine von Gegnern und Wänden. Der Wert 64 erschließt sich aus der Größe der Spielfläche minus der unzerstörbaren Außenwände. Eine Formel zur Berechnung der Dichte würde dementsprechend folgendermaßen aussehen:

$$Dichte = Nicht\,begehbare\,Elemente\,/\,64$$

Das Ergebnis dieser Berechnung muss anschließend gerundet und verdoppelt werden, damit es auf einer Skala von null bis eins in 0.1 Schritten dargestellt werden kann. Bei einem Level mit acht nicht begehbaren Elementen würde die Dichte dementsprechend den Wert 0.2 ergeben. Das wäre ein Level, in welchem sich der Spieler mit großer Freiheit bewegen kann. Ein Level mit einem Dichte-Wert von 0.7 würde die Bewegungen des Spielers stark beeinträchtigen. Beispiele für Level mit einer hohen und niedrigen Dichte sind in Abbildung 5.1 dargestellt.

Die zweite Metrik ist der berechnete Schwierigkeitsgrad eines Levels. Die Metrik wird explizit als berechnet bezeichnet, da der wahrgenommene Schwierigkeitsgrad von Spieler zu Spieler unterschiedlich ist, wodurch dieser schwerer festzulegen ist. Der Schwierigkeitsgrad wird von der ungerundeten Dichte, der Anzahl an Gegnern und der Anzahl an Nahrungsmitteln beeinflusst, da diese Elemente sich auf die Schwierigkeit eines Levels auswirken. Die Metrik wurde folgendermaßen berechnet:

$$Schwierigkeitsgrad = Dichte \times$$

$$Anzahl\,an\,Gegnern\; - (Anzahl\,an\,Nahrungsmitteln \times 0.025)$$

Die Formel ergibt sich aus dem Zusammenhang der Dichte und der Anzahl an Gegnern, welche die ausschlaggebenden Faktoren in der Berechnung des Schwierigkeitsgrades sind, diese Abhängigkeit wird dementsprechend mit einer Multiplikation dargestellt. Die Anzahl an Nahrungsmitteln ist unabhängig von der Multiplikation, da diese sich nur abwertend oder gar nicht auf die Schwierigkeit auswirken kann, deshalb wird dieser Aspekt als Subtraktion in die Formel einbezogen. Der Wert 0.025 ergibt sich aus der Schlussfolgerung zwischen der Anzahl an möglichen Nahrungsmitteln und der Skalengröße.

Wenn für ein Level folgende Werte angenommen werden

- Dichte = 0.2
- Anzahl an Gegnern = 2
- Anzahl an Nahrungsmitteln = 1

ergibt die Berechnung einen Schwierigkeitsgrad von 0.8. Das Ergebnis wurde wie die Dichte gerundet und verdoppelt. Dieser Wert repräsentiert ein Level in welchem Gegner den Spieler stark behindern. Bei einem niedrigen Schwierigkeitsgrad hat es der Spieler dementsprechend leichter. Beispiele für Level mit einem hohen und niedrigen Schwierigkeitsgrad sind in Abbildung 5.1 dargestellt.

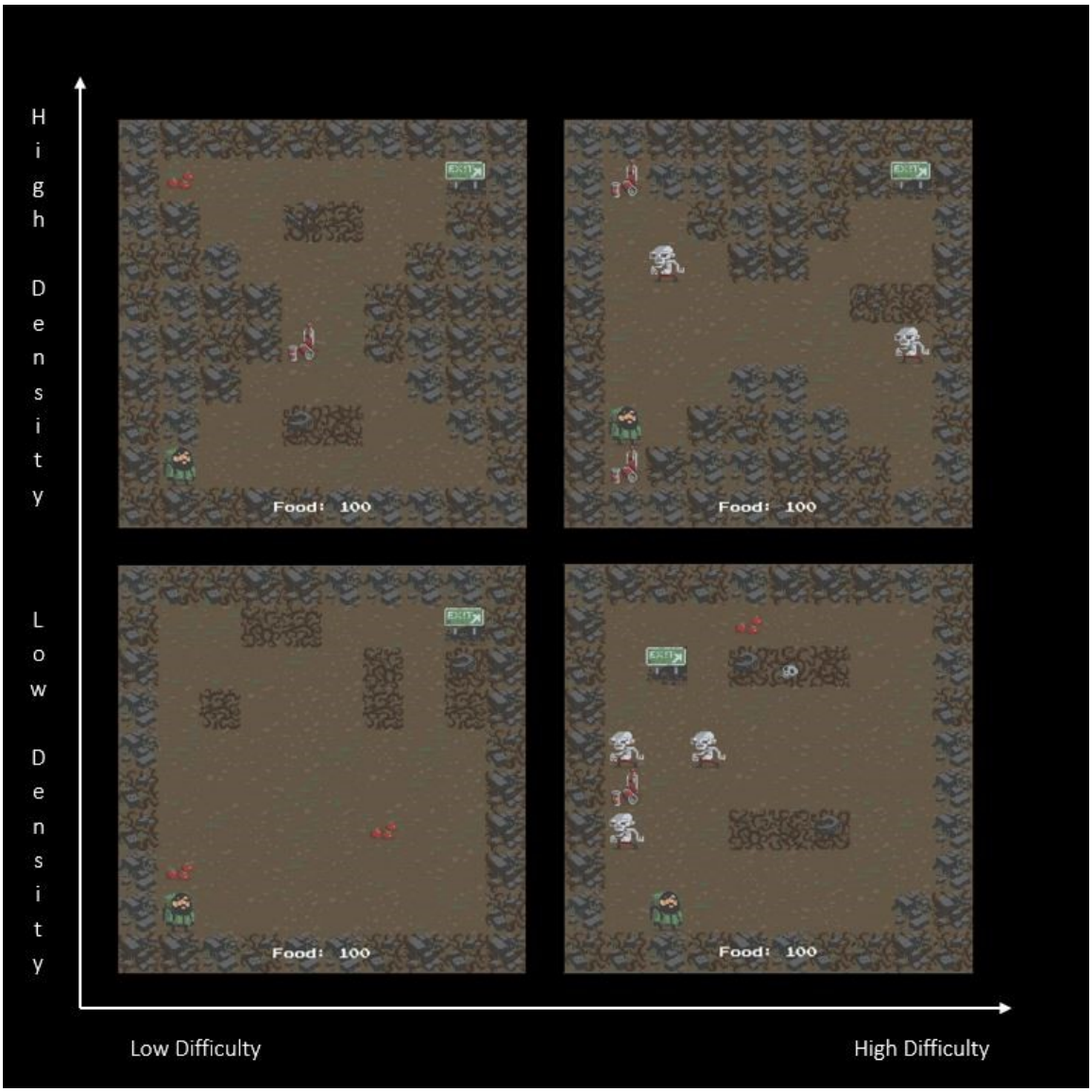


Abbildung 5.1: Level geordnet nach variierender Dichte und variierendem Schwierigkeitsgrad

5.2 Metrische Analyse

Um den Erfolg der generierten Level zu veranschaulichen und zu bewerten, wurden alle Level der Trainings-, der generierten und der Basisdaten nach den in Kapitel 5.1 genannten Metriken bewertet. Dieser Prozess geschah durch eine kleine Anwendung. Daraufhin wurden alle Level mit ihren verschiedenen Werten nach Häufigkeit sortiert und zusammengezählt, um daraus eine zur Darstellung nutzbare Heatmap zu generieren.

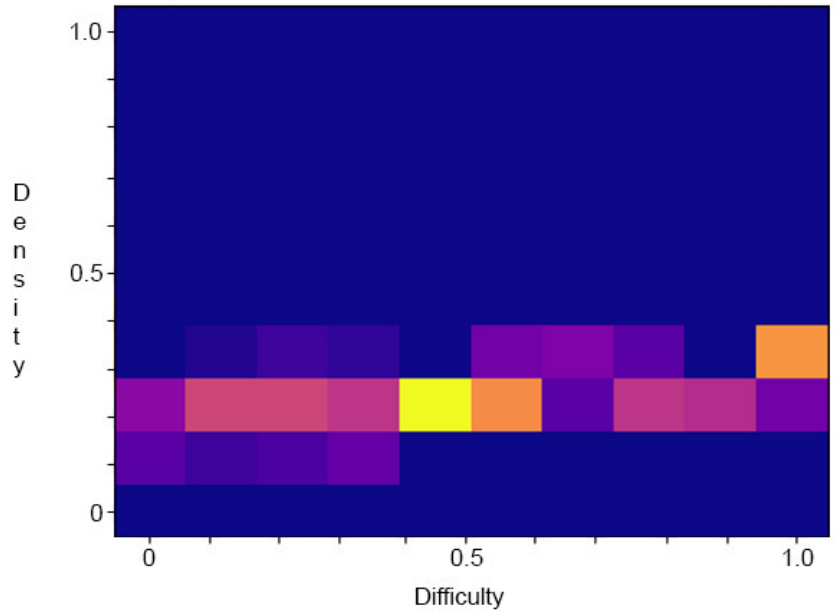


Abbildung 5.2: Heatmap der Inhalte des Basisgenerators

In Abbildung 5.2 sind die Häufigkeiten der verschiedenen Level aus dem Basisgenerator zu sehen. Auffallend an dieser Abbildung ist, dass die hier untersuchten Level alle einen konstant geringen Dichtewert besitzen. Dieser Wert resultiert aus dem Aspekt, dass der Basisgenerator Objekte nach einem festgelegten Minimum und Maximum platziert. Diese Vorgehensweise ist einerseits relativ simpel, andererseits verhindert diese, dass bei einer niedrig festgelegten Anzahl an Objekten eine Überprüfung eines möglichen Weges für den Spieler notwendig ist, da die Anzahl der Objekte so niedrig ist, dass der Weg zum Ende des Levels nicht blockiert werden kann. Diese Vorgehensweise hat aber den negativen Effekt, dass die Level eher willkürlich aussehen, sich nicht handgemacht anfühlen und generell nur offene Leveltypen vorhanden sind. Der durchschnittliche Schwierigkeitsgrad sammelt sich eher in der Mitte. Das liegt daran, dass die Dichte wie erwähnt konstant gering ist und der Schwierigkeitsgrad mit der Dichte zusammenhängt, die Dichte senkt immer den Wert des Schwierigkeitsgrades. Im Basisgenerator wird der Schwierigkeitsgrad durch einen Logarithmus mit der Basis zwei bestimmt, indem dieser als Eingabewert die aktuelle Levelnummer enthält. Das sorgt dafür, dass der Schwierigkeitsgrad konstant steigt, je weiter der Spieler kommt. Dies ist auch eine relativ simple Lösung, doch bei höheren Level würde der Schwierigkeitsgrad nur noch sehr langsam steigen, weshalb alle Versionen des Spiels auf 25 Level begrenzt wurden, um vergleichbar zu bleiben. Also sind die Level des Basisgenerators alle relativ einseitig, jedoch erfüllen sie ihren Zweck.

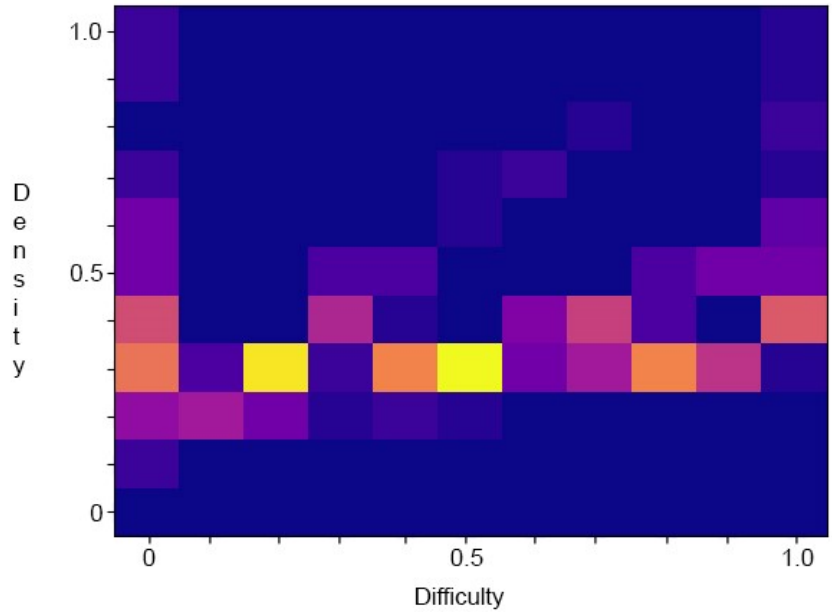


Abbildung 5.3: Heatmap der Trainingsdaten

In Abbildung 5.3 sind die Häufigkeiten der verschiedenen Level aus den Trainingsdaten dargestellt. Anhand dieser Grafik wird dargestellt, dass die meisten Level eher offen sind, da sich die Mehrheit der Level in dem oberen Viertel der Dichte befinden. Bei dem Schwierigkeitsgrad sind die Level regelmäßiger verteilt, wobei die Minima und Maxima weniger vertreten sind. Wie in Kapitel 4.1.2, erwähnt wurden diese Daten manuell erzeugt, wodurch bereits automatisch eine Evaluierung nach der Spielbarkeit stattgefunden hat. Die Abbildung veranschaulicht dementsprechend, dass sich Level im oberen Viertel der Dichte als besonders spielbar erwiesen haben. Informationen zu den Trainingsdaten können in Kapitel 4.2.2 nachgeschlagen werden.

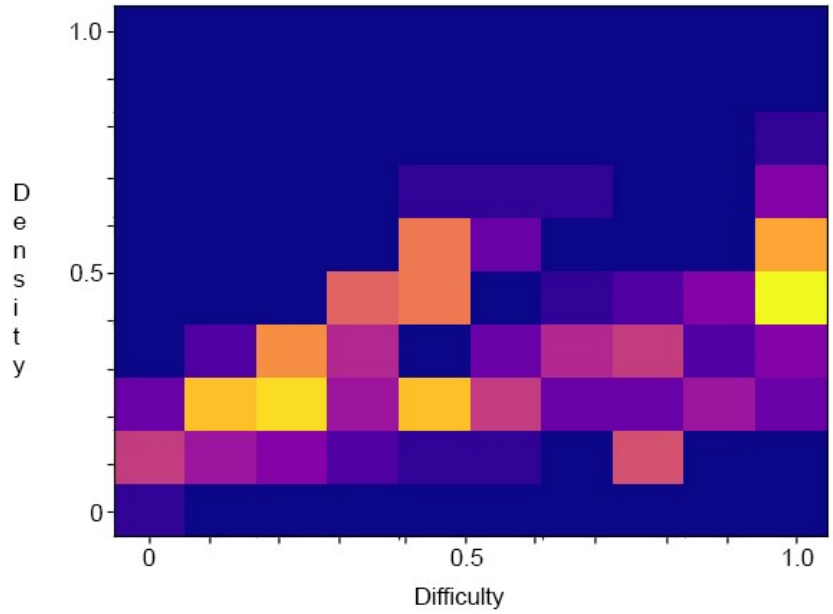


Abbildung 5.4: Heatmap der generierten Inhalte

In Abbildung 5.4 sind die Häufigkeiten der verschiedenen Level aus den generierten Daten dargestellt, welche mithilfe der in Abbildung 5.3 veranschaulichten Trainingsdaten erstellt wurden. Es sticht heraus, dass eine größere Varietät in den Werten der Dichte als in den Trainingsdaten vorhanden ist. Jedoch sammelt sich diese Varietät um den Mittelwert aus den vorherigen Daten, wodurch die Generierung der Level in Betracht der Dichte erfolgreich war. Außerdem wird dargestellt, dass die Level, welche sich in Abbildung 5.3 im oberen Bereich befinden, nicht mehr vorhanden sind. Das hat den simplen Grund, dass in den Trainingsdaten von dieser Levelvariation nicht genügend Beispiele vorhanden waren, wodurch diese kaum Einfluss auf den Lernprozess der Maschine hatten und diese somit einfach vergessen wurden. Wie in den Trainingsdaten ist auch hier eine regelmäßige Verteilung der Schwierigkeitsgrade zu erkennen, mit der Ausnahme, dass nun mehr Level als sehr schwer eingestuft wurden. Rückblickend kann die Generierung der Level im Hinblick auf die Trainingsdaten als erfolgreich eingestuft werden, da sich Abbildung 5.3 und Abbildung 5.4 genügend ähneln.

5.3 Umfrageergebnisse

Kapitel 5.2 hat für die technische Bestätigung der Generation gesorgt. Dieses Kapitel soll menschliche Sachverhalte, wie den empfundenen Schwierigkeitsgrad oder den Unterhaltungswert, darstellen, da diese Aspekte von einer technischen Auswertung nicht ermittelt werden können. Um dieses Vorhaben zu erreichen, wurde eine Umfrage durchgeführt. Diese Umfrage befindet sich im Material A. Das Ziel ist es zu ermitteln, wie die verschiedenen Versionen des Spiels gegeneinander abschneiden, wenn diese von echten Spielern bewertet werden. Außerdem gilt es festzustellen, ob die vom hier verwendeten Generator generierten Inhalte als handgemacht eingestuft werden, oder nicht, da es das Ziel dieses Generators ist seine Inhalte so menschlich wie möglich aussehen zu lassen.

Bei Version Eins handelt es sich um die Inhalte aus dem Basisgenerator. Anhand der Antworten A.2, A.4 und A.5 ist zu sehen, dass die Schwierigkeit der Inhalte überwiegend als leicht eingestuft wurde. Über die Hälfte der Befragten haben es geschafft diese Version komplett durchzuspielen und haben dafür meistens nur einen Versuch gebraucht. Diese Werte sind nachvollziehbar, da die Level dieses Generators eine Schwierigkeitskurve besitzen, welche erst in höheren Inhalten herausfordernd wird, dementsprechend ist der Großteil der Level einfach. In Diagramm A.3 fällt auf, dass die Inhalte sehr schlecht im Punkt Variation abgeschnitten haben. Ähnliche Ergebnisse waren vorhersehbar, da die Häufigkeit einzelner Elemente durch einen zufälligen Wert bestimmt wird, der zwischen einem fest gewähltem Minimum und Maximum liegt. Deshalb gibt es kaum Variation in der Anzahl der Hindernisse, Nahrungsmittel oder Gegner. Auch wenn die bisher erwähnten Aspekte eher negativ klangen, hat die Version im Punkt Unterhaltung positiv abgeschnitten, wie anhand von Abbildung A.1 veranschaulicht wurde. Diese Version wurde zwar nie als äußerst gut empfunden, doch befinden sich viele Antworten im mittleren bis guten Bereich. Das liegt wahrscheinlich daran, dass durch den niedrigen Schwierigkeitsgrad viele Probanden ein Erfolgserlebnis bei dieser Version verspürt haben.

Die interessanteste Antwort in dieser Umfrage handelt davon, ob korrekt vermutet wurde, wer diese Inhalte erstellt hat, ein Algorithmus, eine künstliche Intelligenz oder ein Mensch. In diesem Fall, in Abbildung A.6 zu sehen, haben über 70% der Probanden richtig festgestellt, dass es sich bei dieser Version um einen Algorithmus handelt. Diese Auswertung zeigte im Großen und Ganzen keine spannenden Funde. Die algorithmische Version erfüllt ihren Zweck, sie generiert weder sonderlich gute noch sonderlich schlechte Inhalte.

Als Nächstes folgt die Version Drei, die handgemachten Inhalten werden vorgezogen, Version Zwei folgt danach. Bei Version Drei handelt es sich um die handgemachten Inhalte, welche auch zum Training von Version Zwei verwendet wurden. Anhand von A.14, A.16 und A.17 ist zu erkennen, dass diese Inhalte als relativ schwer eingestuft wurden. Die Mehrheit der Antworten befindet sich im mittleren bis hohen Bereich, manche sogar im sehr hohen Bereich. Das ist darauf zurückzuführen, dass im Gegensatz zur algorithmischen Version keine berechenbare Schwierigkeitskurve verwendet wurde, sondern die Level nach menschlicher Intuition schwieriger wurden. Außerdem ist der Mensch dazu in der Lage eine andere Ebene der Komplexität zu nutzen, indem Wände und Nahrungsmittel ebenfalls zur Veränderung des Schwierigkeitsgrades verwendet werden. Dementsprechend ist es keine Überraschung, dass nur ungefähr 30% der Teilnehmer in der Lage waren diese Version durchzuspielen. Meistens wurde aber auch nur ein Versuch unternommen. Die Antworten zur Frage der Varietät der Inhalte gingen in A.15 in alle Richtungen. Die meisten Antworten befinden sich im mittleren bis hohen Bereich, doch gibt es auch einige Ausreißer nach ganz unten. Diese Antworten lassen sich dadurch nachvollziehen, dass die Level nicht nach bestimmten Regeln erstellt worden sind, sondern willkürlich durch den Designer. Die Ausreißer lassen sich erklären, in dem behauptet wird, dass die Inhalte den Teilnehmern schlichtweg nicht gefallen haben oder diese als langweilig empfunden wurden. Nichtsdestotrotz ist in dieser Version eine eher hohe Varietät an Inhalten vorhanden. Diese Antworten spiegeln sich aber nicht negativ auf den Unterhaltungswert wider, dieser befindet sich laut A.13 im Guten bis sehr guten Bereich. Spielerisch ist das verständlich, da mit einer ordentlichen Schwierigkeitskurve auch eine bessere Herausforderung entsteht, die überwunden werden muss.

Genau wie bei Version Eins fiel es den Teilnehmern leicht zu erkennen, wer diese Inhalte erstellt hat, in diesem Fall war es ein Mensch. Das haben laut A.18 über 80% der Teilnehmer festgestellt. Von Menschen gebaute Level sind einfach zu erkennen, da diese immer gewisse Eigenschaften, wie z.B. Symmetrie und Harmonie besitzen.

Nun folgt die Auswertung des hier umgesetzten Generators, Version Zwei. Ein Fakt, der im Verlauf dieser Auswertung immer wieder vorkommt, ist, dass es sich bei dieser Version um eine Kombination aus handgemachten Inhalten und einer technischen Umsetzung handelt. Dieser Aspekt spiegelt sich in den folgenden Ergebnissen gut wider. Der Schwierigkeitsgrad dieser Version ist ungefähr auf dem gleichen Level wie der von Version Drei, zusehen an A.8, A.10 und A.11. Somit haben es bei einem Versuch auch nur ungefähr 30% der Befragten geschafft, die Version zu beenden. Diese Werte sind nicht sonderlich überraschend, da sich diese Version an Version Drei orientiert hat, um Inhalte zu generieren. Die Antworten zur Frage der Varietät in A.9 sehen aus wie eine Mischung aus den vorher behandelten Versionen. Wo hingegen die Befragten sich bei Version Eins und Drei relativ einig waren, gibt es keinen festen Punkt, bei dem sich die Antworten zu Version Zwei sammeln. Eine Zusammenrechnung dieser resultiert ungefähr in einer mittleren Varietät, was diesen Generator etwas besser macht als den algorithmischen, aber etwas schlechter als die handgemachten Inhalte. Im nächsten Aspekt dem Unterhaltungswert kommt es wieder zu einer Mischung der Antworten aus vorherigen Versionen. Anhand von A.7 ist zu sehen, dass die meisten Antworten, mit ein paar Ausnahmen, sich im mittleren bis guten Bereich befinden. Somit liegt dieser Generator in der Bewertung wieder zwischen Version Eins und Drei.

Mit diesen Ergebnissen im Sinn kann man behaupten, dass der Generator mit maschinellem Lernen erfolgreich war, da dieser besser abgeschnitten hat als der algorithmische Generator, wessen Nachteile dieser beseitigen sollte. Eine andere wichtige Frage war es zu sehen, ob die Inhalte des hier verwendeten Generators als menschliche Inhalte eingestuft werden oder nicht. Dieser Sachverhalt ist in A.12 veranschaulicht. Fast 50% der Teilnehmer nahmen an, dass die Inhalte aus Version Zwei von einem Menschen stammen. Falls mehr Arbeit in den Generator gesteckt wird, wird dieser in der Lage sein noch menschlichere Inhalte zu erstellen, doch für den hier aufgebrachten Aufwand sind 50% ein deutlicher Erfolg.

Im nächsten Kapitel werden ein paar Beispiele aus den generierten Leveln näher betrachtet, um festzustellen, ob Besonderheiten in diesen aufzuweisen sind.

5.4 Generierte Inhalte

In Kapitel 4.2.2 wurde unter anderem auf die Besonderheiten von Leveln eingegangen, z.B. welche Strukturen möglich sind, an was für verschiedenen Positionen sich Aus- und Eingang befinden können und wie unzerstörbare Wände verwendet werden können. Diese Aspekte werden nun noch einmal anhand der generierten Inhalte betrachtet.

Abbildung 5.5: Level mit vielen unzerstörbaren Wänden

In Abbildung 5.5 sind zwei Level aus der generierten Menge zu sehen. An dem linken Level ist zu erkennen, dass die Maschine gelernt hat, unzerstörbare Wände zu verwenden, um die Spielfläche zu verkleinern. Im rechten Beispiel werden die unzerstörbaren Wände nicht nur verwendet, um die Spielfläche zu verkleinern, sondern auch um die Bewegung des Spielers im Level zu kontrollieren. Das passiert dadurch, dass dem Spieler bestimmte nicht zerstörbare Elemente im Weg stehen, wodurch dessen Bewegungsvermögen eingeschränkt ist.

Abbildung 5.6: Level mit verschieden positionierten Ein- und Ausgängen

Eine andere Besonderheit ist in Abbildung 5.6 zu sehen. Hier sind die Ein- und Ausgänge an verschiedensten Positionen platziert. Dabei werden diese aber nicht direkt nebeneinander platziert, sondern behalten immer einen gewissen Abstand. Selbst wenn diese sich nah aneinander befinden, sind sie durch spielerische Elemente getrennt, wodurch ihre eigentliche Distanz größer erscheint, als sie ist. Außerdem ist die Verbindung dieser nicht durch ein unzerstörbares Element unterbrochen.

Abbildung 5.7: Level mit neuen und bekannten Strukturen

Der letzte Aspekt handelt von neuen und alten Strukturen, welche die Maschine erzeugen kann. Auf der linken Seite von Abbildung 5.7 ist zu sehen, wie Nahrungsmittel von Wänden umgeben sind. Das ist eine Struktur, die so auch in den Trainingsdaten vorhanden ist. Die Maschine ist also in der Lage einzelne Strukturen zu erkennen und zu verwenden. Auf der rechten Seite ist ein Level dargestellt, dessen Ausgang von zerstörbaren Wänden umgeben ist. Diese Struktur ist in der Form in keinem Level

der Trainingsdaten verwendet worden. Dadurch wird ersichtlich, dass die Maschine auch in der Lage ist neue Strukturen zu erstellen, welche in Anbetracht der Spiellogik sinnvoll sind.

6 Fazit

Das folgende Kapitel ist das letzte dieser Arbeit. Es enthält eine Zusammenfassung aller vorherigen Kapitel, die Schlussfolgerung der Forschungsergebnisse und einen Ausblick.

Zusammenfassung

Im Verlauf dieser Arbeit wurden thematische Grundlagen zu PCG, Roguelikes und KNNs erläutert. In diesem Zusammenhang wurde ebenfalls auf verwandte Arbeiten eingegangen, welche in diesem Forschungsbereich Relevanz aufweisen. Nach diesen grundlegenden Bereichen wurde nähergebracht, wie spielbare Inhalte anhand eines Beispiels in ordentliche Daten umgewandelt und erzeugt werden können. Diese Daten wurden für den hier verwendeten Generator (LSTM-Generator) angepasst, damit dieser diese zum Training nutzen kann. Anschließend wurden Level generiert und zusammen mit verschiedenen Generatoren in eine Darstellung integriert. Abschließend wurden die in der Darstellung vorhandenen Generatoren mit spielspezifischen Metriken ausgewertet, um den Erfolg des hier umgesetzten Generators zu bestimmten. Im Anschluss wurde eine Umfrage durchgeführt und ausgewertet, um die Impressionen echter Spieler festzuhalten.

Ergebnis

Anhand dieser Vorarbeit konnte ein Fazit dazu gezogen werden, ob und auf welchem Level ein LSTM-Generator mit einem gewöhnlichen PCG-Generator in Konkurrenz stehen kann. Dazu wird vorab eine Tabelle mit Vor- und Nachteilen von diesem LSTM-Generator erstellt, wie es auch in Kapitel 2.1 für den PCG-Generator passiert ist, um einen visuellen und inhaltlichen Vergleich herzustellen.

In diesem Vergleich ist es bedeutsam zu beachten, dass beide Generatoren in der Lage sein können, einen Vorteil voneinander zu kopieren, dieser Vorteil würde dann bei dem jeweiligen Generator aber aufwendiger sein als bei dem Anderen.

Genauso wie in Tabelle 2.1 ist die Anzahl der Vor- und Nachteile ausgeglichen. Das heißt aber nicht, dass diese Generatoren gleich zu behandeln sind, da manche Nachteile einen höheren Einfluss besitzen. Dementsprechend werden alle Punkte in Tabelle 6.1 anschließend erläutert.

Vorteile	Nachteile
Einzigartige Inhalte	Sehr viele Tests erforderlich
Generierung von komplett neuen Inhalten	Nicht weit verbreitet
Hoher Wiederspielwert	Statische Inhalte
Anpassbar	Datenmangel
Unterstützende Verwendung	Aufwendig
Verabschiedung vom klassischen Level Designer	

Tabelle 6.1: Vor- und Nachteile eines LSTM-Generators

Alle vom Generator erstellten Inhalte wirken handgemacht und einzigartig. Wie in Kapitel 4 erwähnt und demonstriert, lernt der Generator anhand von handgemachten Trainingsdaten, wodurch die generierten Inhalte ebenfalls menschlich wirken. Diese Eigenschaft wurde durch die Auswertungen in Kapitel 5 bestätigt. Dieser Aspekt kann ebenfalls mit einem algorithmischen Generator erreicht werden, jedoch würde in diesem Fall mehr Aufwand entstehen.

In Kapitel 5.4 bei Abbildung 5.7 wird dargestellt, wie der Generator in der Lage ist Strukturen aus den Trainingsdaten zu übernehmen und diese in seinen eigenen Inhalten anzuwenden. Das Besondere ist aber, dass der Generator auch dazu imstande ist, bekannte Strukturen und Bausteine zu kombinieren, um neue Strukturen zu generieren und sinnvoll einzubauen. Dieser Aspekt sorgt dafür, dass mehr und verschiedenere Inhalte erstellt werden können, als in der übergebenen Trainingsmenge vorhanden sind. Das ist eine Eigenschaft, welche mit einem gewöhnlichen Generator eine Menge Arbeit erfordern würde, für den LSTM-Generator aber komplett natürlich ist.

Die ersten beiden Punkte aus Tabelle 6.1 führen zum dritten Punkt, dem hohen Wiederspielwert. Dadurch, dass Level immer wieder neu generiert werden, dabei trotzdem einzigartig wirken und neue Strukturen beinhalten können, kann der Spieler immer wieder durch neue Herausforderungen spielen. Dieser Punkt ist ebenfalls bei PCG-Generatoren vorhanden und ist auch ihre beste Eigenschaft, nur muss mehr Aufwand aufgewiesen werden, um den gleichen Wiederspielwert wie bei einem LSTM-Generator zu erhalten. Das ist auch anhand der Umfrage zu Version Eins und Zwei aus Kapitel 5.3 zu erkennen, da der Generator mit maschinellem Lernen bessere Ergebnisse erzielt hat als die algorithmische Variante.

Ein gewöhnlicher PCG-Generator basiert überwiegend auf Codebasis, d.h. dass dieser nur von den entsprechenden Programmierern gepflegt werden kann. Bei Veränderungen im Spiel müssen diese ebenfalls in den Generator eingebaut werden. Mit einem LSTM-Generator ist das nicht der Fall, da wie in Kapitel 4.1.2 erwähnt, nur die entsprechenden Trainingsdaten angepasst werden müssen, um eine Änderung im Spiel in

den generierten Inhalten zu übernehmen. Außerdem kann diese Tätigkeit auch von jemandem übernommen werden, der keine Programmiererfahrung besitzt.

Falls sich dagegen entschieden wird, einen LSTM-Generator zur Generierung seiner Level zu verwenden, kann dieser in einer anderen Art und Weise in Erwägung gezogen werden, und zwar als Unterstützung. Dadurch, dass der Generator in der Lage ist, komplett neue Inhalte und Strukturen zu generieren, können diese Inhalte als Inspiration verwendet werden, um auf neue Ideen zu kommen oder nur um gewisse Aspekte zu kopieren. Falls eine Software zur Erstellung von Leveln vorhanden ist, kann der Generator in diese eingebaut werden, um nach jedem Schritt Vorschläge zu geben oder um Korrekturen durchzuführen.

Diese positiven Aspekte kommen aber nicht ohne Nachteile. Laut Tabelle 2.1 benötigt ein PCG-Generator viele Tests, um als funktionsfähig wahrgenommen zu werden. Bei einem LSTM-Generator müssen jedoch sehr viele Tests vorhanden sein. Das hat den gleichen Grund, weshalb die Level in dieser Integration nicht in Echtzeit generiert wurden, siehe Kapitel 4.3. Da im Gegensatz zu einem PCG-Generator der Lösungsweg eines LSTM-Generators nicht ausgelesen werden kann, kann dementsprechend nie komplett davon ausgegangen werden, dass der Generator immer einwandfrei funktioniert. Ein LSTM-Generator kann als einwandfrei abgesegnet werden, wenn im Großteil der Fälle spielbare Inhalte generiert werden. Nur kann der eine fehlschlagende Fall nicht ohne zusätzliche Software abgedeckt werden.

Die folgende Schwäche ist nicht sonderlich ausschlaggebend, jedoch könnte es bei der Umsetzung trotzdem Probleme bereiten. Und zwar ist die Generierung von Inhalten mit künstlichen neuronalen Netzen noch kaum in der Spieleindustrie vorhanden. Normalerweise besteht diese Thematik nur im Zusammenhang mit der künstlichen Intelligenz von nicht spielbaren Charakteren. Dementsprechend könnte es schwer werden, Hilfe oder Unterstützung zu finden, wenn diese benötigt wird. Gewöhnliche PCG-Generatoren werden seit Jahren verwendet und haben sich bereits in der Spieleindustrie etabliert.

Dieser Punkt ist spielspezifisch, und zwar sind viele Spiele dynamisch, d.h. die Elemente in Spielen können ihre Positionen ändern oder besondere Fähigkeiten einsetzen, die andere Objekte beeinflussen. Die Darstellung der hier generierten Inhalte ist aber statisch, die Maschine besitzt keinerlei Informationen über die Verhaltensmuster der Gegner. Also kann es vorkommen, dass die statische Darstellung eines Levels spielbar erscheint, während der Laufzeit aber unschaffbar wird. In dem hier verwendeten Spiel stellt das keine große Herausforderung dar, weil dieses relativ simpel ist. In stark dynamischen Spielen muss aber zusätzlich dafür gesorgt werden, dass die Maschine entweder Informationen zu den dynamischen Inhalten erhält, eine Schicht vorhanden ist, welche die Inhalte auf diesen Aspekt überprüft oder viele Tests mit diesen Inhalten durchgeführt werden. Bei PCG-Generatoren fällt dieser Schritt weg, da diese ja

genauen Anweisungen folgen. Es besteht zwar die Möglichkeit, dass die Maschine sich die korrekte Positionierung der dynamischen Elemente beigebracht hat, ohne Informationen über diese zu besitzen, jedoch sollte das nicht dem Zufall überlassen werden.

Eines der größten Probleme im Bereich der künstlichen neuronalen Netze ist der Datenmangel, davon ist der hier behandelte Themenbereich nicht verschont, wobei dieser Aspekt bei Spielen noch komplizierter ist. Es handelt sich bei diesem Problem darum, dass eben nicht genug Daten zum Trainieren eines Netzes bestehen. Wenn das der Fall ist, muss selber dafür gesorgt werden, dass genug Daten für das Training vorhanden sind. Dies ist aber wie in Kapitel 4.1.2 erwähnt zeitaufwendig. Außerhalb von Spielen sind teilweise ganze Datensätze vorhanden, welche frei zur Verfügung stehen, z.B. diverse Texte und Bilder. Dementsprechend kann es passieren, dass Datensätze gefunden werden, welche passend zur eigenen Umsetzung sein können. Spiele sind aber in den meisten Fällen einzigartig, es ist nicht möglich Daten aus einem anderen Spiel zu verwenden, um sein eigenes zu trainieren, wenn diese grundsätzlich unterschiedlich funktionieren. Deshalb muss sich selbst mit der aufwendigen Datenerzeugung und Pflege beschäftigt werden. Mit einem herkömmlichen Generator ist das nicht notwendig, da die Daten während der Laufzeit generiert werden. Dieser Punkt ist der schwerwiegendste und ist in der Relevanz mit der Herstellung eines PCG-Generators gleichzusetzen.

Die genannten Nachteile können zu einem Punkt zusammengefasst werden, und zwar können LSTM-Generatoren äußerst aufwendig sein. Dieser Nachteil hängt aber davon ab, wie viele der vorherigen Nachteile zu treffen, denn falls bereits eine ordentliche Menge an Inhalten besteht und das Spiel nicht sehr dynamisch ist, ist der Aufwand im Verhältnis zu den Ergebnissen, die so ein Generator liefern kann, gering.

Die Verabschiedung vom klassischen Level Designer behandelt den gleichen Punkt wie in Kapitel 2.1. Doch hat die Umfrage aus Kapitel 5.3 ergeben, dass bereits 50% der Befragten die maschinellen Inhalte für handgemachte verwechselt haben. In naher Zukunft könnte also der Großteil der Level Designer wirklich verabschiedet werden. Bis dahin ist es möglich einen LSTM-Generator unterstützend zu verwenden.

Anhand der in Tabelle 6.1 dargelegten und erläuterten Punkte kann nun folgendes Fazit gezogen werden:

LSTM-Generatoren sind in der Lage spielbare und menschlich wirkende Inhalte zu generieren. Diese Inhalte stehen einem klassischen Generator in Nichts nach. Jedoch gehört eine ordentliche Vorarbeit dazu, einen LSTM-Generator umzusetzen. Falls diese Vorarbeit vorhanden ist, kann ein LSTM-Generator in Erwägung gezogen werden, entweder als der einzige Level Generator oder nur unterstützend. Wenn diese Vorarbeit nicht vorhanden ist, sollte ein gewöhnlicher Generator in Betracht gezogen werden, im Anschluss ist es immer noch möglich einen LSTM-Generator unterstützend zu verwenden.

Ausblick

Aus den Erkenntnissen dieser Arbeit wird ersichtlich, dass die prozedurale Generierung durch Techniken des maschinellen Lernens zwar eine Alternative zu den etablierten Generierungsmethoden bilden kann, aber auch seine Nachteile hat, welche die Verbreitung der Technik in der Industrie verlangsamen.

Das wohl größte Hindernis, welches ein generelles Problem in diesem Bereich ist, ist der Datenmangel. In manchen Anwendungsfällen existieren bereits Methoden, die es entweder ermöglichen die notwendige Datenmenge zu reduzieren oder die vorhandene Datenmenge zu vergrößern. Solche Methoden wurden zum Beispiel in manchen der Arbeiten aus Kapitel 3.2 ermittelt und angewandt. Das Problem bei Spielen ist aber, dass diese meistens einzigartig sind, weshalb die Methoden eines Spiels nicht auf ein anderes übertragen werden können. Die offensichtliche Lösung hier wäre es, für jedes Spiel eigene Methoden zu entwickeln, das ist aber sehr aufwendig und negiert den Grund, weshalb maschinelles Lernen an erster Stelle in diesem Bereich verwendet wurde. Aus diesem Problem resultiert eine fortsetzende Forschungsfrage. Und zwar eine Möglichkeit zu ermitteln, Spiele so weit zu generalisieren, dass für so viele Spiele wie möglich die gleichen Methoden angewandt werden können.

A Umfrage

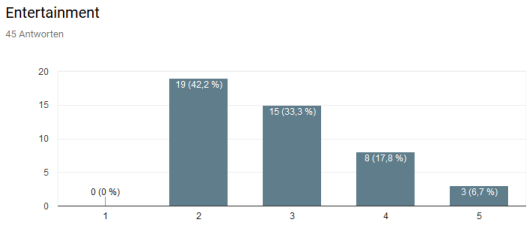


Abbildung A.1: Antworten zur Frage des Unterhaltungswertes für Version Eins

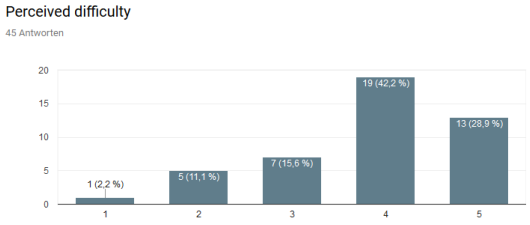


Abbildung A.2: Antworten zur Frage des Schwierigkeitsgrades für Version Eins

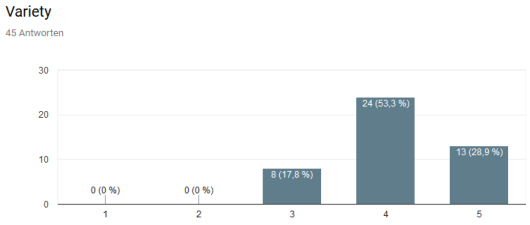


Abbildung A.3: Antworten zur Frage der Varietät für Version Eins

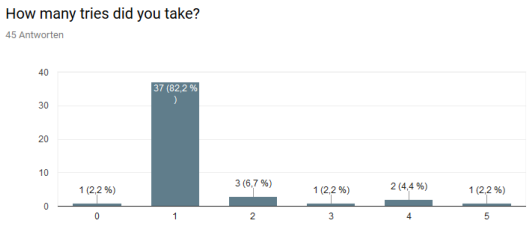


Abbildung A.4: Antworten zur Anzahl der Versuche für Version Eins

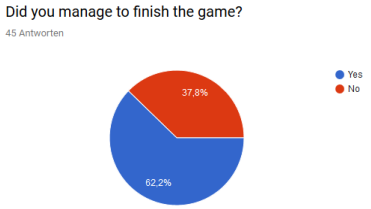


Abbildung A.5: Antworten zur Frage ob Version Eins durchgespielt wurde

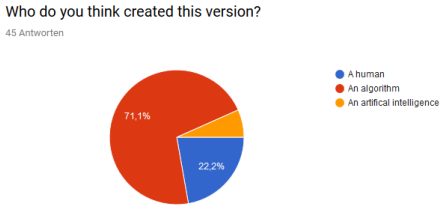


Abbildung A.6: Antworten zur Frage wer die Inhalte für Version Eins erstellt hat

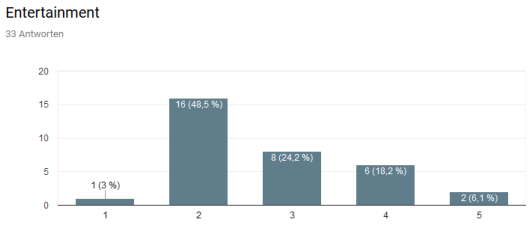


Abbildung A.7: Antworten zur Frage des Unterhaltungswertes für Version Zwei

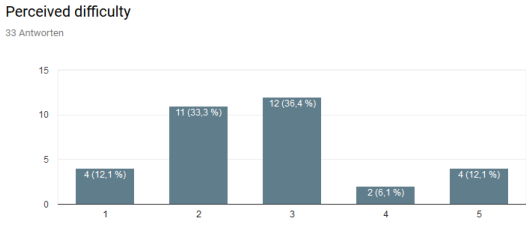


Abbildung A.8: Antworten zur Frage des Schwierigkeitsgrades für Version Zwei

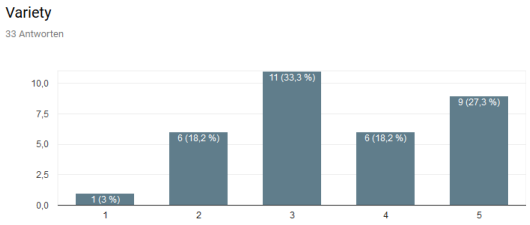


Abbildung A.9: Antworten zur Frage der Varietät für Version Zwei

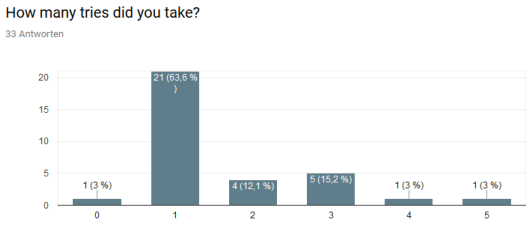


Abbildung A.10: Antworten zur Anzahl der Versuche für Version Zwei

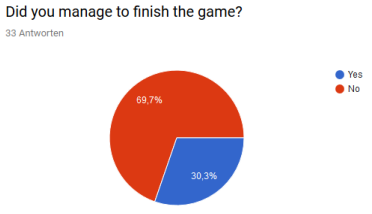


Abbildung A.11: Antworten zur Frage ob Version Zwei durchgespielt wurde

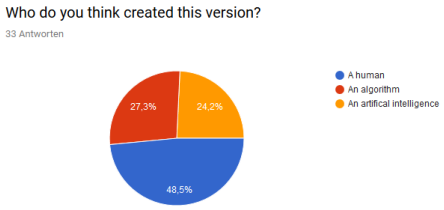


Abbildung A.12: Antworten zur Frage wer die Inhalte für Version Zwei erstellt hat

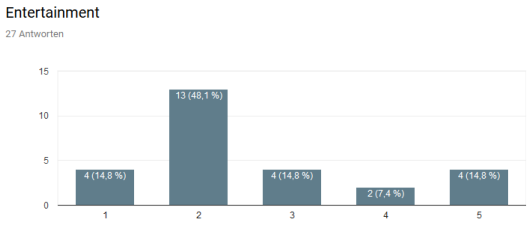


Abbildung A.13: Antworten zur Frage des Unterhaltungswertes für Version Drei

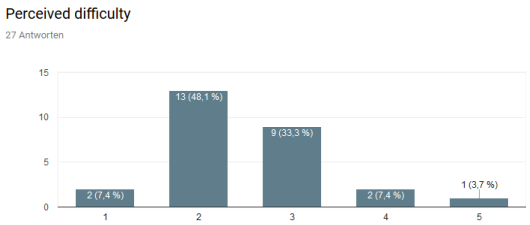


Abbildung A.14: Antworten zur Frage des Schwierigkeitsgrades für Version Drei

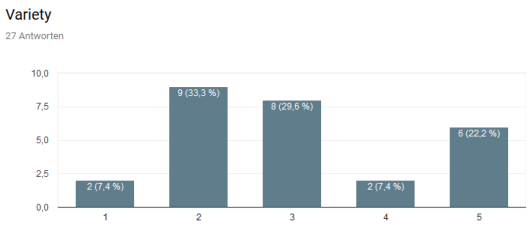


Abbildung A.15: Antworten zur Frage der Varietät für Version Drei

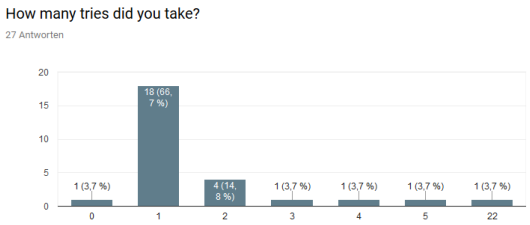


Abbildung A.16: Antworten zur Anzahl der Versuche für Version Drei

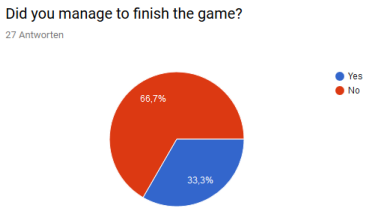


Abbildung A.17: Antworten zur Frage ob Version Drei durchgespielt wurde

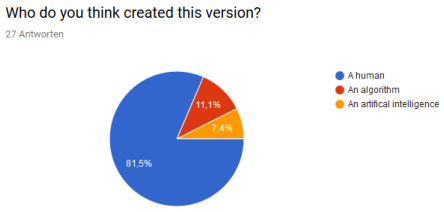


Abbildung A.18: Antworten zur Frage wer die Inhalte für Version Drei erstellt hat

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Byl(2018)
Byl, Penny: *A Beginner's Guide To Machine Learning with Unity. Perceptrons: The making of a Neural Network. The Perceptron*, https://www.udemy.com/machine-learning-with-unity/, 2018, letzter Zugriff: 26. 05. 2018

Cleve & Lämmel(2016)
Cleve, Jürgen; Lämmel, Uwe: *Data Mining (De Gruyter Studium)*, De Gruyter Oldenbourg 2016

Dahlskog & Togelius & Nelson(2014)
Dahlskog, Steve & Togelius, Julian & Nelson, Mark J.: „Linear levels through n-grams“, 2014

Géron (2018)
Géron, Aurélien: *Hands-On Machine Learning with Scikit-Learn and TensorFlow. Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly Media 2017

COLUMN: @Play: The Berlin Interpretation(2009)
Harris, John: *COLUMN: @Play: The Berlin Interpretation*, http://www.gamesetwatch.com/2009/12/column_play_the_berlin_interpr.php, 2009, letzter Zugriff: 21. 05. 2018

Jurafsky & Martin(2014)
Jurafsky, Daniel; Martin, James H.: *Speech And Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition (Prentice Hall Series in Artificial Intelligence)*, Pearson 2014

Karpathy(2015)
Karpathy, Andrej: *Multi-layer Recurrent Neural Networks (LSTM, GRU, RNN for character-level language models in Torch)*, https://github.com/karpathy/char-rnn, 2015, letzter Zugriff: 26. 05. 2018

Karpathy(2015)
Karpathy, Andrej: *The Unreasonable Effectiveness of Recurrent Neural Networks*, http://karpathy.github.io/2015/05/21/rnn-effectiveness/, 2015, letzter Zugriff: 26. 05. 2018

Olah(2015)
Olah, Christopher: *Understanding LSTM Networks*, http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015, letzter Zugriff: 27. 05. 2018

Powell(2014)
Powell, Victor: *Markov Chains Explained Visually*, http://setosa.io/ev/markov-chains/, 2014, letzter Zugriff: 30. 05. 2018

Shaker & Togelius & Nelson(2016)
Shaker, Noor; Togelius, Julian; Nelson, Mark J.: *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Springer 2016

Snodgrass & Ontanón(2015)
Snodgrass, Sam & Ontanón, Santiago: „A Hierarchical MdMC Approach to 2D Video Game Map Generation“, *The Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-15)* 2015

Summerville & Mateas(2016)
Summerville, Adam J. & Mateas, Michael: „Super Mario as a String: Platformer Level Generation Via LSTMs“, *1st International Joint Conference of DiGRA and FDG* 2016

Tay(2017)
Tay, YuXuan: *Character Embeddings Recurrent Neural Network Text Generation Models*, https://github.com/yxtay/char-rnn-text-generation, 2017, letzter Zugriff: 26. 05. 2018

Unity(2015)
Unity: *2D Roguelike tutorial*, https://unity3d.com/learn/tutorials/s/2d-roguelike-tutorial, 2015, letzter Zugriff: 23. 05. 2018

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Konrad Mampe