Universität Trier

Semester: Sommersemester 2021

Modul: BA2STT2216 - PROJEKTSEMINAR

Dozierende: Herr Kai Kugler, Herr Dr. Sven Naumann

Computerlinguistik-Projekt

Simple audio recognition of speech commands for moonlanding game

Dennis Binz, Nathalie Elsässer, Julia Karst, Sarah Ondraszek, Till Preidt

---

Inhaltsverzeichnis

1 Einleitung – Planung und Ziele

1.1 Termin-Planung

Neben vielen verschiedenen Paar- und Kleingruppentreffen, waren folgende Termine die stattgefundenen Hauptbesprechungen:

1. Einführung, Allgemeines
   06.04.2021
2. Festlegung des Themas, Einrichtung Discord
   13.04.2021
3. Meeting ohne Dozenten – Planbesprechung, Spiel festlegen, Aufgabenverteilung
   23.04.2021
4. Plankonkretisierung, Klärung von Fragen
   27.04.2021
5. Meeting ohne Dozenten – Festlegung des Befehlssatzes, andere Kleinigkeiten
   04.05.2021
6. Zwischensitzung
   11.05.2021
7. Meeting ohne Dozenten – Problembehandlung von Worterkennung, Testen des Spiels und Verbesserungsvorschläge
   19.05.2021
8. Meeting ohne Dozenten – Abschließende Änderungen an Modell und Prediction, Besprechung von Details am Spiel und Code-Dokumentation
   24.05.2021
9. Meeting ohne Dozenten – Status-Überprüfung, Planung Zusammensetzen Audio-Recognition und Prediction, Detailänderungen am Spiel
   01.06.2021
10. Meeting ohne Dozenten – Planung Zusammensetzen Prediction und Spiel
    02.06.2021
11. Zwischenpräsentation – Vorstellen der Zwischenergebnisse
    08.06.2021
12. Meeting ohne Dozenten – Lösen von Problemen innerhalb der Packagestruktur
    11.06.2021
13. Meeting ohne Dozenten – Problemlösung innerhalb des Spieles
    15.06.2021
14. Vorstellen eines Prototypen, Besprechung des weiteren Vorgehens
    22.06.2021

15. Meeting ohne Dozenten – Problemlösung der Keyboard-Simulation
    29.06.2021
16. Meeting mit Dozenten – Klärung letzter Fragen
    06.07.2021
17. Präsentation
    13.07.2021

1.2 Ziele und Methoden

Das Ziel unseres Projektes ist es, ein lauffähiges Spiel zu programmieren, das mit Hilfe von Spracheingaben gesteuert werden kann. Hierzu nehmen wir als Inspiration das Spiel Moonlander[1], bei dem es darum geht, eine Rakete sicher auf dem Mond landen zu lassen. Die Rakete kann auf unterschiedliche Arten gesteuert werden. In unserer Version soll der Fokus darauf liegen, die Rakete per Sprachsteuerung vom Abstürzen abzuhalten.

Unser Spiel wird in Python 3.8[2] programmiert. Es werden dazu unterschiedliche Tools genutzt, die in den einzelnen Abschnitten näher erläutert werden sollen.

Zunächst soll – in separaten Gruppen für Speech Recognition und Game Programming – sowohl ein Sprachmodell für die Spracherkennung unserer Befehle trainiert und getestet werden, als auch das Spiel mit Hilfe von Gaming-Tools selbst erstellt werden. Nachdem sich für einen passenden Datensatz entschieden werden konnte, soll dieser genutzt werden, um das Sprachmodell an die gestellten Anforderungen anzupassen und daraus die passenden Informationen zu ziehen. Zum Testen des Modells sollen sowohl eigene Aufnahmen, als auch der Trainingsinput selbst verwendet werden, um die Akkuratesse der Spracherkennung zu überprüfen.

Die konkreten Anforderungen, die das fertige Programm am Ende des Projektes im besten Fall erfüllen sollte, sind hierbei folgende:

- Das Spiel soll lauffähig sein.
- Zur Spielsteuerung soll es möglich sein, als Input Sprachbefehle zu verwenden.
- Sprachbefehle sollen hierbei während des Spiels gesprochen werden können.
- Wird ein Sprachbefehl gesprochen, so soll dies automatisch erfasst werden.
- Der Sprachbefehl soll durch ein neuronales Netz analysiert werden.
- Der Sprachbefehl soll richtig erkannt werden.

[1] `http://moonlander.eu/`
[2] `https://www.python.org/downloads/release/python-380/`

- Basierend auf der Erkennung soll das Spiel die zum gesprochenen Befehl zugehörige Aktion ausführen.
- Der Vorgang der Erkennung und Ausführung der Sprachbefehle soll in einer Geschwindigkeit geschehen, die es erlaubt, das Spiel ohne größere Verzögerungen zu spielen.

Inwieweit diese Aufgaben erfüllt werden konnten, wird in Abschnitt 9 diskutiert.

Die Skripte werden in einem GitHub-Repository gesammelt und können dort eingesehen oder heruntergeladen werden:
`https://github.com/sarahondraszek/audio_recognition_moonlanding`

2 Daten sammeln, vorverarbeiten und Modell trainieren

Das Training eines entsprechenden Modells ist der Kern einer funktionierenden Spracherkennung. Hierzu war es zunächst nötig, passende Sprachdaten zu akquirieren und ein geeignetes Neuronales Netz auszuwählen, um es anschließend auf unsere Zwecke anzupassen und zu trainieren. Ebenfalls waren hierbei das Anpassen der Trainingsdaten und schlussendlich das Feintuning des Modells Themen, mit denen sich intensiv auseinander gesetzt wurde.

2.1 Geeignete Daten wählen

Zuständigkeit: S. O.

Für das Sprachmodell wurden zunächst verschiedene Datensätze konsultiert, unter anderem der „Mini Speech Command“ Datensatz von Google und die erweiterte Version von selbigen Forschern (Warden 2018). Da nur auf wenige Befehle trainiert wird, wurde zunächst der Mini Speech Command Datensatz gewählt, der unter anderem jeweils 1000 Daten zu den acht Befehlen *up, down, left, right, stop, go, no, yes* enthält, die für das Spiel benötigt werden. Nach ersten Trainingsdurchläufen stellte sich heraus, dass der größere Datensatz, mit ca. 2300 Daten pro Speech Command, bessere Ergebnisse bei der Erkennungsrate liefert. Deshalb wurde zum Trainieren des finalen Modells der große Datensatz genutzt, jedoch nur mit den oben genannten acht Befehlen - alle anderen — z.B. Befehle zum Erkennen von Background Noise — wurden ignoriert. Die Daten liegen als *WAVE*-Dateien vor, haben alle eine Länge von circa einer Sekunde und eine Frequenz von 16kHz. Die Audiodateien sind nach den Befehlen benannt, diese „Labels“

werden auch als die jeweiligen Labels für die Prediction unserer Speech Commands verwendet.

2.2 Testen und Bewerten verschiedener Skripte

Zuständigkeit: S. O.

Um ein geeignetes Sprachmodell trainieren zu können, wird zuerst ein passendes Skript benötigt. Für das Training eines eigenen neuronalen Netzes kamen zwei Kandidaten in Frage: Das Skript „Simple audio recognition: Recognizing keywords“ (Martín Abadi et al. 2015) und „Building a Dead Simple Speech Recognition Engine using ConvNet in Keras“ (Mandal 2017). Beide Skripte ähneln sich, da sie auf TensorFlow bzw. Keras basieren und die entsprechenden Module nutzen. Der größte Unterschied liegt jedoch beim Vorprozessieren der Audiodateien. Im Skript von TensorFlow werden die WAVE-Dateien in Spektogramme, also Bilder, umgewandelt und an ein neuronales Netz gefüttert. Dies kann gegebenenfalls hohe Rechnerleistung erfordern. Im Skript von Mandal werden MFCCs genutzt, die im nächsten Abschnitt erklärt werden. MFCCs sind deutlich schonender im Bezug auf das Computing. Da der Computer bzw. das neuronale Netz Bilder sowieso als „Zahlen“ interpretieren, ist der Schritt über ein Spektogramm überflüssig und kann missachtet werden. Deswegen wurde sich letztlich für das Skript von Mandal entschieden. Wie genau das Skript an dieses Projekt angepasst wurde, wird in späteren Abschnitten erläutert.

2.3 Formatierung der Daten (Wave to MFCC)

Zuständigkeit: N. E.

Für das Entwickeln des Language Models wurde ein Convolutional Neural Network (CNN) genutzt, das als Input jedoch nicht die „rohen“ WAVE-Dateien verarbeiten kann, sondern *Mel Frequency Cepstral Coefficients* (MFCCs).

Für die Darstellung einer Audiodatei in MFCCs wird die Audiodatei zunächst in Abschnitte unterteilt. Auf diesen Abschnitten wird danach eine diskrete Fourier-Transformation (DFT) durchgeführt – hierdurch soll eine Analyse der Frequenzen möglich gemacht werden. (Logan 2000) Um eine Repräsentation der Audiodatei zu gewährleisten, die der der menschlichen Wahrnehmung entspricht, wird zur Darstellung der Frequenzen – und damit auch der Tonhöhen innerhalb der Audiodatei – die sogenannte Mel-Skala verwendet. In der MFCC-Darstellung der Audiodatei sollen schlussendlich Frequenzen und deren zugehöriger Schalldruck gespeichert werden. (Hui 2019) Diese Darstellung hat gegenüber der klassischen „Wellenform“-Darstellung einer Audiodatei den Vorteil,

dass aus MFCCs bestimmte Eigenschaften verschiedener Phoneme – beispielsweise Vokalformanten oder hohen Schalldruck bei hohen Frequenzen von Frikativen – abgelesen werden können, was eine Unterscheidung zwischen Phonemen und somit schlussendlich Worten (oder in unserem Fall „Sprachbefehlen“) vereinfacht.

In unserem Programm werden die MFCCs in einem Array dargestellt. Jeder Audiodatei ist hierbei ein Array zugehörig. In einem solchen Array sind Featurevektoren gespeichert, jeder dieser Vektoren repräsentiert dabei einen Abschnitt der Audiodatei. Diese Abschnitte sollen in unserem Fall circa 32 ms lang sein. Ist unsere Input-Audiodatei also beispielsweise 1 s lang, so erhalten wir daraus für alle 32 ms einen Featurevektor und somit circa 31 Featurevektoren. Geht man von einer durchschnittlichen Phonemlänge von 100 ms aus, werden pro Phonem circa 3 Vektoren generiert. Die Feraturevektoren selbst speichern wiederum die Lautstärke- und Tonhöhe-Daten der MFCCs.

Diese Umwandlung wird in unserem Programm in dem Skript `preprocess_with_yaml` durch die Funktion `wav2mfcc` vorgenommen, die zur Generierung der MFCCs die Python-Bibliothek „Librosa“[3] nutzt. Da die Abtastrate („Samplerate“; in Librosa als Parameter `sr` angegeben) auf 16 kHz gesetzt wird (mehr dazu in Abschnitt 2.5), muss der davon abhängige Parameter `n_fft` ebenfalls angepasst werden. Dieser bestimmt, für welchen Zeitabschnitt jeweils ein Featurevektor gebildet wird und sollte im besten Fall eine Zweierpotenz sein. Dies berechnet sich durch:

$$Zeit = \frac{1}{sr \cdot n\_fft}$$

Da die Abtastrate fest auf 16.000 Hz gelegt wurde, wird für den Parameter n_fft der Wert 512 eingesetzt, um somit alle 32 ms einen Featurevektor zu erhalten. Bei allen weiteren Parametern wurde der default-Wert übernommen.

2.4 Vorprozessierung mittels Skript

Zuständigkeit: S. O.

Die MFCCs können von dem CNN wie ein „Bild“ verarbeitet werden. Hierfür dient die Nutzung eines Skripts, das bereits im Abschnitt „Testen und Bewerten verschiedener Skripte“ kurz vorgestellt wurde. So wird das Skript in verschiedene, separate Module aufgesplittet, die auch autark funktionieren sollen. Die Skripte dienen dem Vorprozessieren der Daten (Trainings- und Sampleset), dem Erstellen entsprechender MFCC-Arrays, dem Trainieren des Modells, dem Vorhersagen eines Audioinputs und der Zusammenführung der Komponenten (Pipeline). In der Pipeline kann man, je nach

[3] `https://librosa.org/`

Schritt, bei dem man sich befindet, Teile auskommentieren, um unnötige Dopplungen zu vermeiden (bspw. bei jedem Durchlauf ein neues Modell zu trainieren).

Der erste Schritt ist das Erstellen von MFCCs bzw. Arrays. In besagtem Skript zum Vorprozessieren werden die WAVE-Dateien geladen, dann zunächst in MFCCs umgewandelt und folgend in Numpy[4]-Arrays formatiert, die in einem Ordner gespeichert werden. Damit liegen sie lokal vor und können beim Training einfach abgerufen werden. Für ein Modell müssen also normalerweise immer nur einmal die Arrays erstellt werden, außer, es ändert sich etwas am Input. Für diesen Vorgang benötigt man das Skript `preprocess_with_yaml`, das wiederum immer an eine `yaml`-Datei geknüpft ist, in der die Labels für die verschiedenen Commands gespeichert sind. Außerdem benötigt man das `create_arrays`-Skript und, natürlich, die `run_pipeline`. Im Preprocessing-Skript werden die Dateipfade gesetzt, so einmal für das Trainingsset, das zu laden ist, für die yaml-Datei und den Ort, an dem am Ende die Numpy-Arrays enden sollen. Es sei darauf zu achten, dass die Trainingsdaten auf dem CL-Server liegen und somit mit einem absoluten Pfad angegeben sind. Im Preprocessing-Skript werden außerdem Funktionen bereitgestellt, die das Trainingsset für das folgende Training aufbereiten (`prepare_dataset`, `get_train_test`).

Bei den Arrays ist es wichtig zu beachten, dass diese oftmals nicht die selbe Länge haben. Deswegen müssen sie *gepadded* werden, also werden die Längen angepasst und fehlende Daten mit Nullen aufgefüllt (siehe die Funktion `wav2mfcc`). Wie groß die Arrays sind, wird manuell mit einer Variable `max_length` bestimmt, die mit der Länge der MFCCs und den `feature dimensions` übereinstimmt, die später eine Rolle im Training spielen.

2.5 Training mittels Skript

Zuständigkeit: N. E., S. O.

Für das Training werden die Daten aus den Numpy-Dateien geladen und dem Trainingsskript übergeben. Dafür werden im Preprocessing-Skript noch die Funktionen `prepare_dataset` und `load_dataset` genutzt, um die Daten zunächst mit Librosa und sklearn[5] „in Form zu bringen“. Hierbei werden sowohl die Ursprungsdaten als auch der Testdatensatz in ein Monosignal umgewandelt, außerdem wird die Abtastrate auf 16 kHz eingestellt. Da es nicht möglich ist, von der ursprünglichen Wellenform der Audiodatei alle Messwerte an unendlich vielen Zeitpunkten auszulesen, bestimmt die Abtastrate, wie viele Messwerte zur Weiterverarbeitung verwendet werden. Eine Abtastrate von 16 kHz bedeutet, dass 16.000 Messwerte pro Sekunde verarbeitet werden.

[4] `https://numpy.org/`
[5] `https://scikit-learn.org/stable/`

Dies ist für unsere Zwecke vollkommen ausreichend. Ist dies geschehen, können die Daten geladen und übergeben werden. Mit `get_train_test` wird außerdem aus allen Numpy-Arrays der MFCCs ein einziger Array erstellt, um dann wieder in zufällige, CNN-lesbare Subteile gesplittet und dem Modell übergeben zu werden. Um das Trainingsskript laufen zu lassen, wird `run_train` verwendet. Hierbei werden zunächst im ursprünglichen `train`-Skript die Parameter festgelegt (feature dimensions, number of channels, batchsize, verbose, number of epochs) und dem run-Skript überliefert. Im Train-Skript werden außerdem die Funktionen `get_model` für das Erstellen des Language Models und `reshape_and_predict` für das Umwandeln der später genutzten Dateien aus dem Sampleset. In `run_train` kann dann `get_model` mit den geeigneten Parametern aufgerufen und das Modell an einem geeigneten Speicherort abgelegt werden. Sowohl die Skripte für das Preprocessing als auch die für das Training sind in einem Modul `preprocessing_and_training` gespeichert.

2.6 Feintuning

Zuständigkeit: S. O.

Im Trainingsteil wird mit Hilfe von Keras[6] die Datei dann noch einmal umgeformt, um der *Shape* eines Convolutional Neural Networks gerecht zu werden. Der Input richtet sich nach den Feature Dimensions, die sich in den, bzw. dem, Numpy-Array(s) befinden.

Die Parameter, die für das Modell gebraucht werden, werden größtenteils manuell eingestellt, so zum Beispiel die Feature Dimensions, die auf 40 und 20 gesetzt werden (vgl. 11 und 20 für drei Audio-Commands im Datensatz der Autors). Diese Werte ergaben sich aus verschiedenen Testdurchläufen und erweisen bis jetzt ein zuverlässiges Training. Da acht Speech Commands vorliegen, wurden acht Klassen ausgewählt. Zum Training werden die Epochen auf einen Wert von 200 gesetzt, mit einem batch-size von 100. Channel und Verbose bleiben auf einem Wert von Eins (vgl. Mandal 2017).

Keras stellt einige Methoden für das Training zur Verfügung, von denen unter anderem die *.add*-Methode genutzt wird, um dem Neural Network die verschiedenen weiteren Layer hinzuzufügen (Conv2D kernel size des Inputs, Dropout von 0.4, activation-function layers (softmax, relu), input-shape etc.). Für das Kompilieren wurden einzelne Optimierer[7] für die Loss-Function, Learning Rate usw. ausprobiert, unter anderem Adadelta, Adam und Adagrad. Hierbei hat sich Adam als zuverlässigste Wahl herausgestellt, da das Modell auf 97% Akkuratesse kommt.

[6] `https://keras.io/`

[7] `https://keras.io/api/optimizers/`

3 Audio-Input generieren, Modell-Prediction testen

Das zuvor vorgestellte Modell musste während des Trainingsprozesses mehrmals getestet werden. Auch war es nötig, herauszufinden, in welchem Format die späteren Inputdateien, die während des Spieles aufgezeichnet werden sollen, vorliegen müssen, um auf eine möglichst hohe Erkennungsrate zu kommen. Deswegen wird im Folgenden die Generierung des Audio-Inputs und das anschließende Testen der dadurch erzeugten Dateien näher erläutert.

3.1 Audio-Input generieren

Zuständigkeit: N. E.

Für den Test des Modells wurden entsprechende Testbefehle generiert. In einem ersten Durchlauf wurde versucht herauszufinden, mit welcher Art von Audiodateien die Prediction überhaupt funktionieren kann. Um ein bestmögliches Ergebnis zu erzielen, wurde nach dem ersten Testlauf entschieden, dass die Prediction auf WAVE-Dateien, die bereits von vornherein mit einer 16 kHz Abtastrate und im mono-Format aufgezeichnet wurden, am besten funktionieren kann. Da dies dem Format entspricht, in das die Dateien beim Vorverarbeiten der Daten umgewandelt werden würden, führt die Umwandlung in diesem Fall zu keinerlei Informationsverlust. Des Weiteren beträgt die Samplingtiefe in den von uns gewählten Audiodateien bestenfalls 16-bit, dementsprechend besitzen die Dateien eine Bitrate von 256 kBit/s. Somit verwenden wir Audiodateien, die zwar einerseits noch eine ausreichende Qualität für die Prediction haben, andererseits wird jedoch die Dateigröße möglichst klein gehalten (max. 32 KB), um die Prediction nicht unnötig zu verlangsamen. Das Audiodateiformat entspricht im Endeffekt dem der ursprünglichen Trainingsdaten.

Als Aufnahmesoftware wurde das Programm „Audacity“[8] gewählt. Im zweiten Durchlauf wurden dann WAVE-Dateien mit den Spezifikationen, die sich im ersten Durchlauf als optimal herausgestellt hatten, aufgezeichnet. Hierbei wurde jeder Befehl etwa 30 Mal hintereinander in jeweils einem Audiostream eingesprochen, sodass am Ende für jeden Befehl eine circa 30-sekündige Audiodatei vorlag. Nachdem sich das Zuschneiden der langen Audiodateien mit „ffmpeg“[9] als zu ungenau herausgestellt hatte, wurde das Zuschneiden halb-automatisch in Audacity selbst mit Hilfe von Textmarken in 1-sekündigen Intervallen und anschließendem Export mit Trennung an den Textmarkengrenzen durchgeführt. Eine Korrektur der Daten erfolgte ebenfalls in Audacity, indem

[8] `https://www.audacity.de/`
[9] `http://ffmpeg.org/`

an einigen Stellen Stilleperioden manuell herausgeschnitten oder hinzugefügt wurden. Der zweite Testsatz enthielt so im Endeffekt 247 Audiodateien.

Obwohl die Prediction auf dem zweiten Befehlssatz mit einer hohen Akuratesse abschließen konnte, gab es vorübergehend ein Problem mit dem Befehl „left“, der in vielen Fällen falsch erkannt wurde. Deswegen wurde ein neuer Test- und ggf. Trainingssatz an Audiodateien für diesen Befehl erstellt. Dazu konnten 8 Freiwillige gefunden werden, die mehrere Audio-Dateien mit dem „left“-Befehl generierten und mir im Containerformat Ogg zukommen ließen. Nach einer Umwandlung der Dateien in das passende WAVE-Format konnten auf den Audiodateien dieselben Schritte angewendet werden wie auf dem zweiten Testsatz. So entstanden weitere 162 „left“-Dateien. Das Problem der Prediction dieses Befehles konnte später anderweitig gelöst werden.

Nachdem die Prediction auf den bisher generierten Dateien zufriedenstellend war, wurde ein dritter Testsatz erstellt, der statt 1-sekündige WAVE-Dateien 3-sekündige WAVE-Dateien enthielt. Hiermit sollte getestet werden, ob die Prediction auch auf längeren Dateien möglich wäre. Es wurde dazu dasselbe Vorgehen wie im zweiten Durchlauf gewählt, mit dem einzigen Unterschied, dass die Textmarkenintervalle auf 3 Sekunden erhöht wurden. In diesem Vorgang wurden 168 Audiodateien erzeugt. Eine zufriedenstellende Prediction konnte auf diesem Datensatz nicht erfolgen, woraus geschlussfolgert werden konnte, dass es für die Prediction essenziell ist, 1-sekündige Audiodateien zu generieren.

3.2 Modell-Prediction testen

Zuständigkeit: S. O.

Mit Hilfe der vorig generierten Audiodateien soll das Audio-Recognition-Modell getestet werden. Hierfür wird das Modell geladen und mit einem Testscript auf die Menge der übergebenen WAVE-files aus dem Sampleset angewandt. Im Skript `run_pipeline` kann angegeben werden, welcher Datensatz als Input herhalten soll. Dafür muss der Pfad `sample_path` angepasst werden. Das Model kann via `keras.models.load_model` aus dem entsprechenden Pfad geladen werden. Die Labels werden erneut aus der YAML-Datei gezogen und wiederverwendet, um aus dem Dateipfad die entsprechenden Dateien zu lesen. Der Output besteht aus einer .csv-Tabelle, die die Übereinstimmung der Prediction mit dem Label visualisiert und in einem separaten Ordner gespeichert wird. Ein Beispiel einer solchen Datei befindet sich im Appendix als Abbildung 1.

Es kommen zwei Funktionen zum Einsatz: Zunächst kann das Skript `predict.py` genutzt werden, um daraus entweder für einen größeren Datensatz Vorhersagen machen zu

lassen (`make_prediction`), oder um später beim Spielen für den Input eines Commands nur genau eine Vorhersage zu machen (`make_single_prediction`).

3.3 Implementierung einer Simulation von Tastaturereignissen

Zuständigkeit: T. P.

Sowohl für die Aufnahme des Befehls, als auch für die Simulation eines Tastendrucks, wurden im Skript „input_key_ctrl_micr.py“ - nachfolgend „Input-Skript“ genannt - entsprechende Funktionen implementiert, auf die das Spiel zugreifen kann. Grundsätzlich ist die Aufnahme also von der Tastatur-Simulation unabhängig und das Spiel verwaltet im Skript „game.py“ die bereitgestellten Funktionen. Zusätzlich war es notwendig, ebenfalls die Simulationen für das Drücken und Loslassen einer Taste unabhängig voneinander und in separaten Funktionen zu implementieren. Denn bei einer sequentiellen Ausführung ohne Zwischenschritte wäre die zeitliche Differenz so gering, dass das Spiel den Tastendruck nicht erkennen und nicht darauf reagieren könnte, bevor dieser wieder gelöst wird. Darüber hinaus sei noch erwähnt, dass die Simulation von Tastatur-Ereignissen ursprünglich gar nicht geplant war, da ich sie zunächst nicht für notwendig hielt. Das Ziel war, den Sprach-Befehl im Spiel direkt zu verarbeiten, indem die entsprechenden Spiel-internen Funktionen aufgerufen und die gewünschten Aktionen (z.B. das Beschleunigen der Rakete beim Herunter-Fallen-Lassen) ausgeführt werden. Im „game.py“-Skript erfolgt die Abfrage einer Taste allerdings mittels sogenannter „Key_Events“, welche Bestandteil des Pygame-Packages sind, welches wiederum für die Entwicklung der Spielabläufe hauptsächlich verwendet wurde (siehe Abschnitt 4). Die oben genannten Spiel-internen Funktionen in den im jeweiligen Kontext relevanten Klassen bestimmter Spiel-Objekte (wie z.B. der Rakete) erwarten jedoch die Übergabe des Rückgabewertes eines dieser Key_Events zur genaueren Weiterverarbeitung, sodass es nicht möglich ist, diese ohne ein solches Event aufzurufen. Statt das Spiel umzustrukturieren, erschien es weniger aufwändig, den Zwischenschritt der Simulation von Tastatur-Ereignissen zu implementieren. Dazu wird im Input-Skript das Package pyautogui[10] importiert. Somit kann dieses Skript insgesamt folgende drei Funktionen für das Spiel bereitstellen:

1. ,,`record_order`‘‘ zum Aufnehmen einer Wave-Datei (siehe Abschnitte 4.3 und 5.1)
2. ,,`simulate_key_press`‘‘ nutzt Funktion „keyDown“ aus pyautogui zum Simulieren eines Tastendrucks

[10] `https://pyautogui.readthedocs.io/en/latest/`

3. ,,release_key‘‘ nutzt Funktion „keyUp“ aus pyautogui zum Simulieren des Loslassens einer Taste

Sowohl „simulate_key_press“, als auch „release_key“ erwarten einen String, der angibt, welche Taste gedrückt bzw. gelöst werden soll. Die Umwandlung der Prediction in eine Tastaturbelegung ist Teil des „game.py“-Skripts, weshalb der genaue sequentielle Ablauf diesbezüglich in Abschnitt 5.2 beschreiben wird.

4 Spiel „Moonlanding“

Das „Moonlanding“-Spiel ist der Hauptteil unseres Projektes und umfasst den für die Anwender*innen sichtbaren Teil des Programms. Neben der GUI für das Spiel werden in diesem Abschnitt ebenfalls die Aufnahme der Sprachbefehle und alle weiteren Schritte, die vor der Prediction des Sprachbefehls durchgeführt werden müssen, beschrieben.

Genauere Informationen zu den Spielregeln und benötigten Bibliotheken finden sich zudem im `Benutzerhandbuch` (Zuständigkeit: J. K., unterstützt von N. E.).

4.1 Programmierung

Zuständigkeit: D. B., J. K.

Unsere Version des „Moonlanding“-Spiels wurde mithilfe von Pygame[11] programmiert. Ziel des Spiels ist es, eine Rakete sicher auf dem Mond landen zu lassen. Um dies zu erreichen, muss die Rakete zum Zeitpunkt der Landung (Height = 0) die entsprechende Geschwindigkeit, sowie den richtigen Neigungswinkel aufweisen. Die Geschwindigkeit sollte hierbei kleiner als 10 sein und der Winkel sollte im Bereich von -6 bis 6 liegen. Zudem gilt es während des Falls einem kommenden Meteor auszuweichen und dem Wind, der die Rakete zufällig nach rechts oder links neigt, entgegen zu wirken. Ebenso muss darauf geachtet werden, dass der Rakete nicht der Treibstoff ausgeht, man kann daher nicht unbegrenzt Schub geben.

Da unser Spiel ein studentisches Projekt darstellt und somit nicht veröffentlicht werden soll, wurden zunächst ausschließlich Grafiken aus dem Internet zur Gestaltung verwendet. Nach und nach wurden diese durch eigene Designs ersetzt, dies betrifft jedoch nicht die Bilder des Mondes, Meteors oder Mikrofons[12] (vgl. Abschnitt 4.2).

Das Spiel besteht aus ingesamt vier Bildschirmen: Der Spielerauswahl `choose_player`, dem Starbildschirm `menu`, dem eigentlichen Spiel, welches in der Funktion `main` gere-

[11] `https://www.pygame.org/`

[12] `https://images.app.goo.gl/RB3dVF8AnEVa7HpA6` (Mond), `https://images.app.goo.gl/smK2zpax9r9riSqz8` (Meteor) und `https://images.app.goo.gl/MZy5rfyPgGdHPt2y8` (Mikrofon)

gelt wird, sowie einem Endbildschirm `end_screen`. Die Abfolge wird schließlich in der Funktion `game` festgelegt.

Beim Starten des Spiels erhält man zunächst die Möglichkeit, per Tastendruck ('1', '2' oder '3') zwischen drei Raketen auszuwählen. Die gewählte Rakete beeinflusst dabei den Schwierigkeitsgrad des Spiels. Der Schwierigkeitsgrad, die zwei Bilder für die gewählte Rakete, sowie zwei Bilder für die jeweilige Flagge werden an die Funktion `main` weitergegeben. Nach der Spielerauswahl gelangt man auf den Startbildschirm und kann durch Drücken der Taste 'Enter' (Sprachbefehl: 'Go') mit dem Spiel beginnen.

Der Status der Rakete lässt sich der Anzeige am oberen Rande des Bildschirms entnehmen. Dort finden sich Angaben zu der Geschwindigkeit (Velocity), der Höhe (Height) und dem Neigungswinkel (Angle) der Rakete. Ebenfalls finden sich dort Informationen dazu, in wie vielen Schritten und auf welcher Höhe mit einem Meteor zu rechnen ist. Letztere Angaben müssen genutzt werden, um eine Kollision mit dem Meteor zu vermeiden. Eine Treibstoffanzeige (Fuel) gibt an, wie viel Treibstoff noch verfügbar ist. Die Rakete stürzt ab, wenn ihr der Treibstoff ausgeht, daher ist es nicht möglich unbegrenzt Schub zu geben. Unterhalb der Statusanzeige befindet sich die Angabe der 'Last Order', welche den letzten Befehl (sowohl Sprachsteuerung als auch Tastendruck) anzeigt.

Eine Steuerung der Rakete ist mittels der Pfeiltasten auf der Tastatur, als auch per Sprachsteuerung möglich. Die Rakete fällt nicht automatisch, d. h. das Spiel wartet erst auf einen Befehl, bevor es eine Aktion ausführt, sodass Zeit genug bleibt, um den Sprachbefehl zu erkennen. Die Sprachsteuerung des Spiels wird durch das Drücken der Leertaste aktiviert. Anschließend wird drei Sekunden lang das Audio aufgenommen. Während der Audioaufnahme wird auf dem Bildschirm ein Mikrofon-Symbol eingeblendet. Um zu erkennen, welche Taste von dem Nutzer gedrückt wurde, wird `event.key` benutzt, welches von Pygame bereitgestellt wird. Nutzt man die Sprachsteuerung, so wird ein entsprechender Tastendruck simuliert (vgl. Abschnitt 3.3).

Zur Programmierung des Spiels wurden im Wesentlichen zwei Sprite-Klassen verwendet: die Klasse *Player* für die Rakete und die Klasse *Meteor* für das Hindernis, dem es auszuweichen gilt. Zusätzlich gibt es die Sprite-Klassen *Moon*, *Microphone*, *Explosion* und *Flag*, welche jeweils die Anzeige der entsprechenden Bilder regeln.

Die Rakete rotiert abhängig von dem zufällig berechneten Winkel, sowie des jeweiligen Schwierigkeitsgrades der Rakete. Die Winkeländerung wird hierbei in der Funktion `angle_change` vorgenommen und graphisch mittels der Funktionen `rot_center` und `rotate` dargestellt. Durch das Drücken der Pfeiltasten nach links bzw. rechts (Sprachbefehle: 'Left' bzw. 'Right') kann der Winkel korrigiert werden kann. Durch Drücken der Pfeiltaste nach oben (Sprachbefehl 'Up') wird ein Standardschub erzeugt (auch die-

ser ist abhängig von dem gewählten Schwierigkeitslevel), der die Geschwindigkeit der Rakete verringert. Möchte man der Rakete keinen Schub geben, so wird dies durch das Drücken der Pfeiltaste nach unten (Sprachbefehl: 'Down' oder 'Go') erfüllt. Der gegebene Befehl wird jeweils in der Funktion `update` verarbeitet.

In der Klasse Meteor werden zufällig die Anzahl der Schritte bis zur Kollision, sowie die Kollisionshöhe erzeugt. Die verbleibende Schrittzahl liegt dabei zwischen 5 und 15, die Kollisionshöhe zwischen 100 und 400. Ist die Kollisionshöhe erreicht, so bewegt sich der Meteor über den Bildschirm. Kommt es dabei zu einer Kollision mit der Rakete, so hat man das Spiel verloren.

Bei einem Absturz der Rakete erscheint eine Explosion auf dem Bildschirm, bei erfolgreicher Landung wird hingegen eine Flagge angezeigt. Zuletzt erscheint ein Endbildschirm mit einer entsprechenden Statusanzeige, welche von der Funktion `main` übergeben wurde. Man hat die Möglichkeit, durch Drücken der Taste 'r' (Sprachbefehl 'Yes') zurück zur Spielerauswahl zu gelangen oder das Spiel zu beenden (Sprachbefehl 'No').

4.2 Ingame Graphics

Zuständigkeit: S. O.

Die im Spiel verwendeten Raketen inkl. Besatzung und die Explosion stammen von mir. Für die Mondlandschaft oder den Meteoriten übernehme ich keine Haftung.

4.3 Audio-Recording

Zuständigkeit: T. P.

Um vom Spiel aus eine Aufnahme zu starten, bzw. eine Wave-Datei zu erzeugen, aus welcher dann ein Befehl predicted wird, muss das „game.py“-Skript auf die im Input-Skript vordefinierte Funktion „record_order“ zugreifen. Gleichzeitig sollte der eigentliche Spielablauf möglichst unabhängig von dieser Funktion bleiben. Um dies zu erreichen, habe ich im „game.py“-Skript die für die Tastensteuerung ohnehin schon implementierten Key_Events um eine Abfrage der Leertaste ergänzt (K_Space). Der Nutzer kann das Spiel somit ganz normal mit den Tasten steuern oder die Leertaste drücken, um die Sprachsteuerung zu aktivieren. Das hier beschriebene Aufnahme-Verfahren einer Wave-Datei bildet den ersten Schritt in diesem Prozess. Eine Einsicht in die Zusammenhänge zwischen Aufnahme, Prediction und Spiel folgt in Abschnitt 5.2.
Zunächst soll hier das Aufnahme-Verfahren an sich erläutert werden:
Bei einem Aufruf von „record_order“ wird das Programmgeschehen in das Input-Skript

verlagert. Dieses Skript importiert für die Aufnahme die Packages sounddevice[13] und scipy.io.wavfile[14]. Das sounddevice-Package wird für den Zugriff auf an das System angeschlossene (oder integrierte) Audio-Geräte benötigt. In unserem Falle erfolgt der Zugriff auf das Mikrofon, welches in den Systemeinstellungen als Standard deklariert ist.

Hinweis: In der momentanen Version von Moonlander steht eine Auswahl diesbezüglich noch nicht zur Option. Stattdessen können Änderungen durch den Nutzer in der Systemsteuerung des Betriebssystems vorgenommen werden.

Mithilfe der Funktion „rec“ aus sounddevice wird die Aufnahme mit entsprechenden Parametern gestartet. Diese Parameter müssen, wie schon in vorigen Abschnitten erwähnt, mit den Eigenschaften der Wave-Dateien, mit denen das neuronale Netz trainiert wurde, übereinstimmen. Aktuell beträgt die Abtatsrate 16 kHz und die Dauer der Aufnahme drei Sekunden, wobei nachträglich auf eine Sekunde geschnitten wird (siehe Abschnitt 4.4). Zusätzlich ist definiert, dass die Wave-Datei nur eine Ton-Spur enthält (mono). Während die Aufnahme läuft, wird mit der Funktion „wait“ aus sounddevice das Programm solange angehalten, bis die Aufnahme beendet wurde. Aus dem Package scipy.io.wavfile wird die Funktion „write“ verwendet, um die Aufnahme in die Wave-Datei zu schreiben und als solche abzuspeichern. Das Speichern erfolgt im Projekt-Ordner „audio_recognition_moonlanding“ unter der Bezeichnung „recorded_order.wav“. Da dieser Pfad konstant ist, wird diese Datei mit jeder neuen Aufnahme überschrieben. Anschließend folgt die Übergabe an das Skript „cut_1sec.py“ zum Herausfiltern von Intensitäts-Höhepunkten und Kürzen, sowie die Übergabe an das neuronale Netz für die Prediction. Diese beiden Schritte werden in den folgenden Abschnitten erläutert. Abschließend wird der vorhergesagte Befehl von der hier beschriebenen Funktion als String zurückgegeben.

4.4 Trimmen des Audio-Inputs

Zuständigkeit: N. E.

Unser Modell kann eine zufriedenstellende Prediction nur durchführen, wenn der Audio-Input als maximal 1-sekündige Datei vorliegt. Da es schwierig ist, einen Sprachbefehl nach einem Tastendruck in unter einer Sekunde zu artikulieren, haben wir uns dazu entschieden, zunächst – wie in Abschnitt 4.3 beschrieben – eine 3-sekündige Mikrofonabfrage vorzunehmen und diese dann abzuspeichern, um die 3-sekündige Datei dann erst im Anschluss auf eine Sekunde zuzuschneiden. Das Zuschneiden der Datei beruht

[13] `https://python-sounddevice.readthedocs.io/en/0.4.1/`
[14] `https://www.kite.com/python/docs/scipy.io.wavfile`

auf der Annahme, dass der lauteste Part der aufgezeichneten Audiodatei der Part sein sollte, in dem der Sprachbefehl artikuliert wurde.

Um den lautesten, 1-sekündigen Part der Audiodatei zu finden, wird die Python-Bibliothek „pydub“[15] verwendet. Mit Hilfe der Klasse „`AudioSegment`“ kann ein beliebiger Teil der Audiodatei herausgeschnitten und gespeichert werden.

Erster Ansatz

In den ersten Versuchen des Zuschneidens wurde ein 1-sekündiges Fenster über die Audiodatei gelegt. Dieses Fenster wurde – beginnend am Anfang der Datei, d. h. bei 0 ms – fortlaufend um 10 ms verschoben. Vor jedem weiteren Verschieben wurde dann die Lautstärke des gerade betrachteten Fensters berechnet. Dies geschah mit der von `AudioSegment` bereitgestellten Funktion `.dBFS`, die die Lautstärke einer WAVE-Datei in der Einheit dBFS (*Decibels relative to full scale*; maximal erreichbarer Wert: 0 dBFS) zurückgibt. Die Lautstärken der einzelnen Fenster wurden miteinander verglichen und am Ende wurde das Fenster mit der größten Lautstärke so abgespeichert, dass es die ursprüngliche, 3-sekündige Datei überschrieb.

Dieser Ansatz funktionierte zwar im Großen und Ganzen, jedoch war das Programm relativ langsam, da erst über die ganze Audiodatei iteriert werden musste – des Weiteren waren die Zuschnitte nicht zufriedenstellend. Dies wurde getestet, indem ein Testset aus Audiodateien, die durch diese Methode zugeschnitten wurden, erstellt wurde. Hierdurch entstanden 240 WAVE-Dateien. Die Erkennungsrate wurde in einer Tabelle[16], die auch in unserem Github-Repository im Ordner `./_Anderes/Testinput` einsehbar ist, festgehalten und lag lediglich bei 48,75%. Bei der Überprüfung der zugeschnittenen Dateien war auffällig, dass in vielen Fällen der Anfang des gesagten Befehls abgeschnitten wurde. Aus diesem Grund wurde dieser erste Ansatz verworfen.

Zweiter und dritter Ansatz

In einem neuen Versuch wurde die automatische Silence-Detection von pydub verwendet, um zunächst die Parts der WAVE-Datei zu erkennen, die *nicht* still sind. Dazu kann ein sogenannter Silence-Threshold verwendet werden, der definiert, ab welcher Lautstärke ein Audio-Part als „still“ gilt. Dieser Wert wird in unserem Fall zunächst auf -65 dBFS gesetzt, was bedeutet, dass alle Abschnitte der Audiodatei, die leiser als -65 dBFS sind, als „still“ gewertet werden.

[15] `https://pypi.org/project/pydub/`

[16] `https://docs.google.com/spreadsheets/d/12rrdZ2M-mGryZa92hkWps-p2P41sxcDzyp3dKoZ3q54/edit?usp=sharing`

Aus diesen nicht-stillen Parts wird im nächsten Schritt durch die Berechnung der Lautstärke in dBFS der lauteste Part herausgesucht. Ist dieser Part mehr als 1 s lang, ist davon auszugehen, dass die Hintergrundgeräusche lauter sind als der derzeitige Silence-Threshold-Wert. Deswegen wird der Silence-Threshold hochgesetzt (in unserem Fall auf -40 dBFS), damit auch lautere Parts als zuvor noch als Stille erkannt werden. Danach wird ab dem ersten Auftreten von Lautstärke die Audiodatei auf eine Sekunde zugeschnitten. Ist der lauteste Part von vorne herein weniger als eine Sekunde lang, so wurden von der WAVE-Datei zunächst zusätzlich zu dem lautesten Part noch die Millisekunden vor diesem mit exportiert, um die Datei auf genau eine Sekunde zu bringen. Die Theorie, dass dies funktionieren könnte, beruhte auf der Annahme, dass bei den zugeschnittenen Audiodateien des vorherigen Versuches in vielen Fällen der Anfang des Befehls fehlte.

Diese Methode konnte zunächst ebenfalls keine ausreichend guten Ergebnisse erzielen. Die Akkuratesse unseres Modells auf diesen Dateien lag mit 49,58% nur unerheblich höher als zuvor. Die genaue Verteilung zwischen eingesprochenem und vorhergesagtem Befehl kann in der Spreadsheet-Tabelle auf Tabellenblatt 2 eingesehen werden. Das Problem dieser Methode schien darin zu liegen, dass nun in vielen Fällen das Ende der Audiodateien fehlte – somit ergab sich ein gegenteiliges Problem zum vorherigen Versuch. Um dieses zu beheben, wurden im nächsten Schritt den lautesten Parts, die weniger als eine Sekunde lang waren, nicht mehr nur am Anfang mehr Audiomaterial beigefügt, sondern gleichermaßen viel Audiomaterial am Ende und am Anfang. Auch diese Methode wurde getestet und erzielte leider keine bessere Akuratesse. Die genauen Ergebnisse sind in der Spreadsheet-Tabelle auf Tabellenblatt 3 vermerkt.

Endgültiger Ansatz

Da festgestellt werden konnte, dass die Akuratesse der bisherigen Methode(n) stark von dem verwendeten Mikrofon abzuhängen schienen, wurde vor dem Identifizieren von Stille und dem Zuschneiden der Audiodatei ergänzend eine Normalisierung derselben mit Hilfe der `normalize`-Methode von pydub vorgenommen. Normalisieren bedeutet hierbei, dass die Audiodatei verstärkt und somit lauter gemacht wird, sodass alle durch Normalisierung erzeugten WAVE-Dateien etwa gleich laut sein sollten. Das Ziel hiervon war es, das Problem der verschiedenen Mikrofonqualitäten zumindest etwas einzudämmen.

Die Ergebnisse der Methode, die die Normalisierung der Daten vornimmt, ist in Tabellenblatt 4 der Spreadsheet-Tabelle vermerkt. Auch hierbei wurden wie bei jedem vorherigen Durchlauf 240 Testdaten erzeugt. Die Akuratesse konnte auf 68,33% gesteigert werden.

5 Zusammenführung

Nachdem die Aufnahme der Audio-Dateien, das Modell inklusive der Prediction und das Spiel größtenteils unabhängig voneinander programmiert wurden, mussten diese anschließend zusammengeführt werden. Dies wurde in mehreren Schritten durchgeführt. Danach wurde zusätzlich eine endgültige Ordnerstruktur für das gesamte Projekt festgelegt und umgesetzt.

5.1 Audio-Recording und Prediction

Zuständigkeit: N. E.

Zunächst wird im `input_key_ctrl_micr`-Skript mit Hilfe der Methode `record_order` die Aufnahme des Sprachbefehles gestartet. Sobald diese erfolgreich vollzogen werden konnte, wird daraufhin die `segment`-Methode aus dem Skript `cut_1sec` aufgerufen, die die – zuvor als *recorded_ order.wav* abgespeicherte – Audiodatei zuschneidet und überschreibt. Nach dem Zuschneiden wird auf der endgültigen, 1-sekündige WAVE-Datei mit der Funktion `make_single_prediction` aus dem `predict`-Skript eine einzelne Prediction ausgeführt. Hierzu wird die `recorded_order`-Audiodatei selbst an unser Modell übergeben. Die Prediction gibt schlussendlich den erkannten Befehl als String zurück.

5.2 Prediction und Spiel

Zuständigkeit: T. P.

Hinweis: In diesem Abschnitt soll der sequentielle Ablauf bei einer Spiel-internen Verwendung der Sprachsteuerung beschrieben werden. In den Abschnitten 3.3 und 4.3 wurden benötigte Funktionen vorgestellt, auf die hier nicht mehr näher eingegangen wird.

Zunächst wurde zur Vorbereitung die Abfrage von Key_Events im „game.py“-Skript um K_SPACE erweitert, da das Drücken der Leertaste den Zugriff auf die Sprachsteuerung ermöglichen soll. Idealerweise wurde der Hauptteil des Spiels so konstruiert, dass das Programm in einer while-Schleife läuft und darauf wartet, dass eine der abgefragten Tasten gedrückt - also eines der vordefinierten Key_Events ausgelöst - wird, bevor die nächsten Schritte berechnet werden. Die separate Behandlung von Tastatur-Ereignissen ermöglicht sowohl eine reine Tastatursteuerung, also auch eine reine Sprachsteuerung, sowie eine gemischte Steuerung der Rakete im Spiel und der Menü-Optionen im Start- und End-Bildschirm. Wenn also eine andere Taste als die Leertaste gedrückt wird, erfolgt kein Zugriff auf die im Input-Skript vordefinierten Funktionen und die Sprach-

steuerung kommt gar nicht erst zum Einsatz. Dies ist besonders nützlich, wenn der Nutzer eine zu schwache Hardware für die Prediction hat, das Spiel aber trotzdem mit den Tasten und ohne Sprachsteuerung spielen möchte. Zunächst beschränke ich mich beim nachfolgenden Ablauf auf die Nutzung der Sprachsteuerung im eigentlichen Spiel.

Der Nutzer aktiviert die Sprachsteuerung durch Drücken der Leertaste. Das damit verbundene Key_Event löst erst einmal noch keine Aktion aus, allerdings werden einige boolsche Variablen (z.B. „listen“ auf „True“) aktualisiert, damit nach dem derzeitigen Durchlauf der Schleife einerseits Tasten-unabhängige Spielfunktionen - wie der Schrittzähler des Meteors - pausiert werden können und das Programm andererseits beim erneuten Eintritt in die Schleife weiß, dass eine Mikrofonabfrage erfolgen soll, bevor wieder auf einen Tastendruck gewartet wird. Im nächsten Durchlauf wird also als erstes die „record_order“-Funktion im Input-Skript aufgerufen. Diese startet eine Aufnahme, generiert aus dieser eine Wave-Datei, welche an das neuronale Netz übergeben wird und liefert den vorhergesagten Befehl als String an das Spiel zurück. Dieser wird via simplem if-else-Verfahren auf einen der sechs Begriffe untersucht, die in diesem Spiel-Kontext relevant sind: „right“, „left“, „up“, „down“, „go“ und „stop“. Wurde schließlich einer dieser Begriffe als Befehl zurückgegeben, wird der entsprechende String als Parameter der Funktion „simulate_key_press“ aus dem Input-Skript weiter verwendet (Details dazu im Abschnitt 3.3). Nachdem die Druck-Simulation der entsprechenden Taste ausgelöst wurde, läuft die Schleife im Spiel weiter bis zu der Stelle, an der überprüft werden soll, welche Taste gedrückt wird. Nur dass in diesem speziellen Durchlauf nicht der Nutzer eine Taste betätigt, sondern ein Tastendruck simuliert wird. Die Sprachsteuerung macht sich hier also die ursprüngliche Struktur des Spielablaufs zu eigen. Das Spiel reagiert, wie bei der normalen Tastensteuerung und ruft die relevanten Spiel-internen Funktionen auf. In der „update“-Funktion des Rocket-Objektes wird nochmals überprüft, welche Taste gedrückt wurde, um die Rakete entsprechend reagieren zu lassen. An dieser Stelle wird auch die Funktion „release_key“ aus dem Input-Skript (mit entsprechendem String als Parameter) aufgerufen, um die gedrückte Taste wieder zu lösen. Dies hat aber nur Wirkung, wenn die Simulation eines Drucks der jeweiligen Taste überhaupt stattfand und nicht, wenn der Nutzer selbst eine Taste gedrückt hat. Dass bei der Implementierung der Sprachsteuerung die ursprüngliche Programmstrukur hier von mir genutzt wird, um erst den Tastendruck zu simulieren, und dann das Spiel darauf reagieren zu lassen, bevor dieser wieder gelöst wird, ist auch der Grund dafür, dass - wie in Abschnitt 3.3 beschrieben - die Funktionen von Tastendruck und -lösung überhaupt separat deklariert wurden. Der Rest des Schleifendurchlaufs besteht in der Anpassung der Spiel-Objekte an die neuen Verhältnisse und der Aktualisierung der angezeigten Grafik - schlicht wie bei der normalen Tastensteuerung. Anschließend

beginnt der Kreislauf von vorne, sofern das Spiel nicht terminiert. Die Sprachsteuerung für die Menüs „Start-Bildschirm“ und „End-Bildschirm“ funktioniert ähnlich zu dem hier beschriebenen Beispiel. Anstatt jedoch die Funktionen für das Drücken und Loslassen einer Taste aufzurufen, wird der von „record_order“ zurückgegebene Befehl direkt an die Menüsteuerung gekoppelt - auf dieselbe Weise, wie die Key_Events bei der Tastensteuerung. Wie in Abschnitt 3.3 beschrieben, war dies in der Hauptschleife des Spiels nicht möglich, da bei der Verknüpfung der Spiel-Objekte mit der Tastensteuerung Funktionen aufgerufen werden, welche das jeweilige Key_Event selbst als Parameter fordern. In den kleineren Menü-Schleifen ist dies nicht der Fall und es kann direkt die gewünschte Aktion ausgeführt werden, ohne die Schleife ein zweites mal traversieren zu müssen.

5.3 Modulverwaltung

Zuständigkeit: N. E., S. O.

Da die Skripte teilweise in Einzel- oder Paararbeit entstanden sind, besteht eine weitere Hürde – neben der reinen Zusammenführung – darin, die einzelnen Skripte und Ordner in eine logisch zusammenhängende und nachvollziehbare Ordnerstruktur zu bringen. Unser Ansatz war es hierbei, das vollständige Programm in folgende drei Teile zu gliedern:

1. das Modell („`preprocessing_and_training`“)
2. Aufnahme und Zuschneiden des Audio-Inputs („`record_and_cut`“)
3. das Spiel selbst („`game`“)

Diese drei Programmteile sollen jeweils in eigenen Ordnern liegen und von außen durch eine einzelne `MAIN`-Datei aufrufbar sein. Die Ausführung der `MAIN`-Datei soll hierbei das per Sprachbefehlen steuerbare Spiel starten.

Zunächst wurde festgelegt, welche Skripte und Dateien in welchem der drei möglichen Ordner abgelegt werden:

1. `preprocessing_and_training`
    - `create_arrays`
    - `predict`
    - `preprocess_with_yaml`
    - `run_pipeline`

- `run_train`
- `train`
- Ordner: *command_ sampleset*
- Ordner: *model*
- Ordner: *numpy*
- Ordner: *output_ testing*

2. `record_and_cut`
    - `input_key_ctrl_micr`
    - `cut_1sec`
3. `game`
    - `game`
    - Ordner: *data*

Zusätzlich muss in jedem Ordner und auch in unserem Hauptordner eine `__init__`-Datei abgelegt werden, damit diese als Packages ex- und importiert werden können.

Zu Beginn wurden einige kleinere Verbesserungen vorgenommen. Beispielsweise wurde zuvor die aufgenommene Inputdatei (*recorded_ order.wav*) in einem eigenen Unterordner abgespeichert, was im Zuge der neuen Ordnerstruktur geändert wurde, sodass die Datei nun direkt in dem *record_ and_ cut*-Ordner abgelegt wird. Des Weiteren wurde das Modell aus dem *model*-Ordner ursprünglich anhand der Angabe eines absoluten Pfades geladen. Diese Angabe wurde durch einen relativen Pfad ersetzt.

Das Hauptproblem, das nach den Verbesserungen vorlag war das Folgende:
Die im Ordner `preprocessing_and_training` liegende Datei `yaml-config.yaml` wird sowohl beim Starten des Spieles vom `MAIN`-Skript aus, als auch beim Trainieren des Modells mit dem `run_train`-Skript aufgerufen. Da diese beiden Skripte auf verschiedenen Ebenen liegen, muss die `yaml`-Datei je nach Verwendungszweck anhand unterschiedlicher Pfade aufgerufen werden. Im Falle eines Aufrufes durch das `MAIN`-Skript lautet dieser `.\preprocessing_and_training\yaml-config.yaml`, im Falle eines Aufrufes durch das `run_train`-Skript muss der Ordnername weggelassen werden.

Um dies zu gewähren, wurde eine Variable *is_ game* eingeführt, die beim Kompilieren steuert, welcher Dokumentpfad eingesetzt wird. Der *is_ game*-Variable wird beim Starten des Games automatisch der Wert **True** übergeben (vgl. hierzu das `input_key_ctrl`-Skript).

Falls es Probleme geben sollte, das Training zu starten, muss man das Project-Directory seiner Systemvariable hinzufügen, damit Python die Packages auch wie Module behandeln kann. Dafür verwendet man `import sys sys.path.append(PROJEKTPFAD)` (hier bitte den entsprechenden Pfad des eigenen Rechners + Projekts einfügen!). Dies muss in alle Skripte, die Modul-Teile importieren, eingefügt werden. Natürlich kann man ebenso die Systemvariable manuell in den Einstellungen des PCs verändern.

6 Autodocumentation mit Sphinx

Zuständigkeit: S. O.

Zur Vervollständigung der Programmdokumentation wurde diskutiert, ob nicht eine automatisch erstellte Docstring-Doku sinnvoll sein könnte. Hierfür wurde die Sphinx-Autodocumentation gewählt, die innerhalb von PyCharm installiert und verwendet werden kann[17]. Dafür müssen alle Skripte, die im Programm enthalten sind, vollständig im entsprechenden Format kommentiert sein. Dies wurde testweise für die Preprocessing- und Trainingsskripte gemacht und daraus eine entsprechende Dokumentation gezogen. Das Ergebnis ist eine mäßig funktionierende HTML-Seite, die sowohl ein Readme und eine Projekt-Introduction als auch einen Index und Modulverweise auf die verschiedenen Skripte enthält; Mäßig funktionierend, da die Links zwischen den Abschnitten nicht funktionieren. Schließlich wurde sich dafür entschieden, die Dokumentation der Skripte manuell vorzunehmen und in diesem Dokument einzubinden. Ein Benutzerhandbuch für das Spiel wird extern erstellt und beigefügt.

7 Testphase

Zuständigkeit: D. B., N. E., J. K., S. O., T. P.

Nachdem das Modell und das Spiel zusammengesetzt wurde, und Letzteres mit Hilfe der Sprachsteuerung spielbar sein sollte, traten noch unerwartete Bugs und Komplikationen auf, die im folgenden Abschnitt erläutert werden sollen.

7.1 clock.tick()-Problematik

Wie sich in der Testphase herausstellte, führte der Aufruf von clock.tick(30) bei einigen Nutzer*innen zu Problemen, z.B. wurden die Bilder nicht aktualisiert oder man kam nicht vom Startbildschirm weg. Ursprünglich wurde der Befehl eingebaut, damit

[17] `https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html` und `https://www.jetbrains.com/help/pycharm/generating-reference-documentation.html`

der Meteor mit einer guten Geschwindigkeit durch den Bildschirm fliegt. Zur Fehlerbehebung wurde die entsprechende Zeile im Anschluss wieder auskommentiert und die Geschwindigkeit des Meteors von 25 hinunter auf 1 gesetzt. Je nach verwendetem Rechner bewegt sich der Meteor daher nun unterschiedlich schnell, die Geschwindigkeit bleibt dabei aber noch in einem angemessenen Rahmen.

7.2 Flaggen-Problematik

Nachdem die in Abschnitt 7.1 beschriebene clock.tick()-Problematik erfolgreich gelöst werden konnte, tauchten nach dem Hinzufügen der Flaggen-Simulation bei einigen Nutzer*innen wieder ähnliche Probleme auf. Ebenso konnte das Spiel teilweise nicht vom Terminal aus gestartet werden. Da auch in der Flaggen-Simulation clock.tick() aufgerufen wird, wurde zunächst getestet, ob dies auch in dieser Situation wieder die Fehlerursache ist. Ein Auskommentieren der entsprechenden Zeilen schien das Problem zu lösen. Zudem stellte sich heraus, dass nach dem Schließen einiger gleichzeitig geöffneter Programme das Problem nicht mehr auftrat, auch wenn clock.tick() verwendet wurde. Die Problematik des Startens vom Terminal aus bestand jedoch immer noch, sodass überlegt wurde, einen Container zu nutzen. Dies war aus zeitlichen Gründen jedoch leider nicht mehr umsetzbar.

7.3 Keyboard-Problematik

Nach einigen erneuten Tests stellte sich heraus, dass die Simulation der Tastendrücke die Ursache für einige der oben genannten Probleme darstellte. Dies konnte gelöst werden, indem die Verwendung von `pynput.keyboard` durch die Nutzung von `pyautogui` ersetzt wurde. Daraufhin lief das Spiel bei allen Nutzer*innen problemlos und konnte auch vom Terminal aus gestartet werden.

8 Ergebnisse

In Abschnitt 1.2 wurden konkrete Ziele definiert, die bis zur Beendigung des Projektes im Idealfall erfüllt werden sollten. Von diesen Zielen wurden die meisten zumindest teilweise erreicht.

Es wurde ein lauffähiges Spiel, das mit Hilfe von verschiedenen Sprachbefehlen gesteuert werden kann, programmiert. Des Weiteren können die Sprachbefehle während des laufenden Spieles eingesprochen und von unserem Modell analysiert. Es konnte hierbei eine gute Erkennungsrate erreicht werden, wenngleich diese leider immer noch recht stark mikrofon- und umgebungsabhängig ist. Was nicht erreicht werden konnte, war die

automatische Erkennung, dass ein Sprachbefehl gesprochen wurde. Damit ein Sprachbefehl erkannt wird, muss momentan zuvor immer noch zuerst die Leertaste betätigt werden. Eine Erkennung während einer kontinuierlichen Mikrofonabfrage konnte nicht erreicht werden.

Das Spiel führt die zum erkannten Sprachbefehl zugehörige Aktion zuverlässig aus. Auch läuft die Erkennung in einer Geschwindigkeit, die es ermöglicht, das Spiel ohne größere Verzögerungen zu spielen.

8.1 Fazit

Die Umsetzung eines sprachgesteuerten Spiels stellte sich als durchaus anspruchsvoll heraus. In einigen Fällen mussten durch trial-and-error-Verfahren mehrere Ansätze ausprobiert und immer wieder verworfen werden, bis eine funktionierende Lösung gefunden wurde. Dies zögerte die weitere Entwicklung des Spieles an einigen Stellen hinaus. Bei der in Abschnitt 7.3 angesprochenen Keyboard-Problematik konnte bis zum Ende des Projektes nicht herausgefunden werden, wieso die zuerst verwendete Python-Bibliothek nicht auf jedem Rechner funktioniert hatte, obwohl zunächst viele Ressourcen darauf verwendet wurden, den Fehler zu identifizieren.

Da für diese und ähnliche Problematiken teilweise viel Zeit aufgewendet werden musste, konnten andere geplante Features nicht umgesetzt werden. So wäre es beispielsweise wünschenswert gewesen, eine kontinuierliche Spracherkennung zu implementieren, anstatt das Drücken der Leertaste als Vorraussetzung für die Erkennung des Sprachbefehls zu verwenden. Jedoch konnte dies aufgrund von Zeitmangel nicht mehr umgesetzt werden.

Des Weiteren wäre es von Vorteil gewesen, wenn die einzelnen Teams für die verschiedenen Abschnitte des Programms von Anfang an noch enger zusammen gearbeitet hätten. Hier war es relativ aufwändig, einige Programmteile gegen Ende zusammenzusetzen. Insbesondere die Ordnerstruktur hätte bereits früher festgelegt werden sollen. Damit hätten die Probleme, die insbesondere mit den Pfadangaben auftraten, präventiv verhindert werden können.

Abschließend ist jedoch zu sagen, dass die gesetzten Ziele des Projektes im Großen und Ganzen gut erreicht werden konnten. Mit etwas mehr Zeit wäre es sicherlich möglich gewesen, einige der genannten Punkte noch zu verbessern.

Literatur

Hui, Jonathan. "Speech Recognition — Feature Extraction MFCC & PLP - Jonathan Hui - Medium". In: *Medium* (Aug. 2019). URL: https://jonathan-hui.medium.com/speech-recognition-feature-extraction-mfcc-plp-5455f5a69dd9 (besucht am 17. 05. 2021).

Logan, Beth. "Mel Frequency Cepstral Coefficients for Music Modeling". In: *Proc. 1st Int. Symposium Music Information Retrieval* (Nov. 2000).

Mandal, Manash Kumar. *Building a Dead Simple Speech Recognition Engine using ConvNet in Keras*. Hrsg. von Medium. [Online; posted 21-11-2017]. Nov. 2017. URL: https://blog.manash.io/building-a-dead-simple-word-recognition-engine-using-convnet-in-keras-25e72c19c12b.

Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

Warden, Pete. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition". In: *ArXiv e-prints* (9. Apr. 2018). arXiv: 1804.03209 [cs.CL]. URL: https://arxiv.org/abs/1804.03209.

A Appendix

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
yes	yes	Yes	yes	Yes	Yes	left	yes	yes	yes	yes	yes	no	yes	yes	yes	yes
up	up	up	up	up	up	up	up	go	up	up	up	up	up	up	up	up
stop	stop	stop	stop	stop	stop	stop	stop	up	stop	stop	stop	stop	stop	stop	stop	stop
right	right	right	down	right	right	right	go	go	right	right	right	right	down	right	right	go
no	no	no	go	no	no	no	no	no	go	no	go	no	no	go	down	no
left	left	left	left	left	left	left	left	left	go	left	left	left	left	left	left	left
go	go	go	go	go	go	go	go	go	go	go	go	go	go	go	no	down
down	down	down	down	down	down	down	down	down	down	down	down	up	down	down	down	down

Abbildung 1: Beispielhafter Durchgang + Output des Samplesets