UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

DLR Deutsches Zentrum für Luft- und Raumfahrt

Nachvollziehbarkeit und Begründbarkeit von Maschinellen Lernverfahren mithilfe von 3D-Visualisierung

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Marcel Bock

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Andreas Schreiber (Deutsches Zentrum für Luft- und Raumfahrt)

Koblenz, im November 2018

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Das Ziel der vorliegenden Arbeit ist die Entwicklung einer Anwendung zur Visualisierung von neuronalen Netzen im dreidimensionalen Raum. Mithilfe der *Unreal Engine* und einem *TensorFlow Plug-in* wird ein *Convolutional Neural Network*, das den *MNIST* Datensatz verwendet, visualisiert. Eine interaktive Anwendung wird entwickelt, in welcher das neuronalen Netz und seine Ergebnisse in Echtzeit betrachtet werden können. Layer und Neuronen werden geometrisch in einer 3D-Szene dargestellt. Neben grundlegenden Informationen für jedes Neuron werden auch die Funktionsweisen der Neuronen durch Einbindung von Testdaten und Darstellung von Ergebnisbildern visualisiert. Aktivierungswerte werden optisch durch farblich-leuchtende Neuronen repräsentiert. Neuronen können gezielt deaktiviert werden, um die Auswirkungen auf die Klassifizierung zu beobachten. Mithilfe einer Technik, die sich am *Level of Detail* Verfahren orientiert, können Modelle in Teilen visualisiert werden. Eine Schnittstelle über ein *Python-Skript* erlaubt es, eigene Modelle zu definieren und auszuwählen, welche Layer und Neuronen des Modells visualisiert werden sollen. Mithilfe einer Evaluation werden Anforderungen festgehalten, welche als Grundlage für Visualisierungen von neuronalen Netzen genutzt werden können.

Abstract

The topic of the present thesis is the developement of an application which visualizes a neural network in threedimensional space. A *Convolutional Neural Network* which uses the *MNIST* dataset is visualized with the *Unreal Engine* and a *TensorFlow plug-in*. An interactive application will be developed with which the neural network and its results can be inspected in real-time. Layer and neurons are represented geometrically in a 3D scene. Besides basic informations about every neuron, the functionality of each neuron is visualized by inclusion of testdata and the depiction of result images. Activation values of neurons are represented visually by glowing colors. Neurons can be deactivated to observe the consequences for the classification. Models can be visualized partially with a technique which is derived from the *Level of Detail* method. A *python script* interface allows to define new models and to decide which parts of a model should be visualized. Requirements will be registered during an evaluation and can be used as a foundation for further visualizations of neural networks.

Inhaltsverzeichnis

1 Einleitung

Künstliche Intelligenz (*KI*) ist ein fester Bestandteil in der Industrie und hält mit jedem Jahr mehr Einzug in den Alltag. Einparkhilfen und automatisierte Lichtanlagen gehören zum Standard neuerer Automodelle und in Zukunft sollen selbstfahrende Autos das Fahren übernehmen. In Videospielen dient künstliche Intelligenz als Gegenspieler und in Bildverarbeitungssoftware werden Objekte und Personen automatisch erkannt. Selbst in Kühlschränken findet sich KI, die den Nutzer zum Beispiel über die leere Milchpackung informiert[1].
Bei all diesen Prozessen ist es wichtig, dass die künstliche Intelligenz keine Fehler macht. Im schlimmsten Fall könnte ein Fehler einen Unfall verursachen, zum Beispiel bei selbstfahrenden Autos. Der Benutzer muss sich auf das System verlassen können. Die Prozesse, die die künstliche Intelligenz durchläuft um zu einem Ergebnis zu kommen, sind komplex. Bei großen Systemen, wie zum Beispiel *AlphaGo* von *Google DeepMind*, werden verschiedene Modelle kombiniert, welche viele unterschiedliche Ebenen (*Layer*) verwenden[2]. Wegen der fehlenden Übersichtlichkeit und der schweren Nachvollziehbarkeit werden solche Systeme als *Black Box* bezeichnet[3].

Es existieren bereits verschiedene Ansätze, um einen Einblick in diese *Black Box* zu ermöglichen. Mithilfe von Graphen werden Datenflüsse visualisiert, die Aktivierung von Neuronen wird dargestellt und Zwischenergebnisse ausgelesen[4].
3D-Engines, wie zum Beispiel die Unreal Engine, bieten die Möglichkeit komplexe Szenarien zu visualisieren und zu gestalten. Sie werden unter anderem für CGI-Filme, Computerspiele, Architekturvisualisierungen oder Simulationen genutzt. Auch für *Virtual Reality* Anwendungen werden sie eingesetzt[5].
Ziel dieser Arbeit ist die Konzeption und Implementierung einer interaktiven Anwendung, welche ein neuronales Netz mithilfe einer 3D-Engine visualisiert. Auf Grundlage von bestehenden Anwendungen soll das Modell in die Engine übertragen werden. Der Schwerpunkt der Visualisierung liegt auf dem Aufbau des Modells, dem Datenfluss und den Zwischenergebnissen der Layer, welche übersichtlich präsentiert werden sollen. Die Nachvollziehbarkeit und Begründbarkeit soll durch diese Form der Präsentation verbessert werden.

[1] Faggella 2018: Artificial Intelligence Industry – An Overview by Segment
[2] Silver et al. 2017: Mastering the Game of GO without human knowledge
[3] Sentinent 2018: Understanding the 'black box' of artificial intelligence
[4] Wolchover 2017: New Theory Cracks Open the Black Box of Deep Learning
[5] Unreal Engine Homepage

In Kapitel 2 wird der aktuelle Stand der Forschung vorgestellt. Eine kurze Einführung in neuronale Netze als auch in bereits vorhandene Software zur Visualisierung wird gegeben.
Die Unreal Engine ist eine der meist verwendeten Engines und bietet über ein Plug-in die Möglichkeit TensorFlow-Modelle zu verwenden. Die Besonderheiten der Unreal Engine und die Verwendung des Plug-in werden in Kapitel 3 vorgestellt.
Ein Konzept zur Visualisierung von neuronalen Netzen wurde herausgearbeitet und die Ergebnisse zusammengefasst (Kapitel 4). Die Umsetzung des Konzepts wird im Kapitel 5 beschrieben.
Eine Evaluation (Kapitel 6) wurde durchgeführt, um Anforderungen an eine Visualisierung festzuhalten. Auf Grundlage eines Fragebogens wurde Feedback über die Verbesserung der Nachvollziehbarkeit und Begründbarkeit gesammelt. Abschließend werden die Ergebnisse zusammengefasst und als Ausblick mögliche Erweiterungen der Visualisierung beschrieben (Kapitel 7).

Diese Masterarbeit wurde in Kooperation mit dem Deutschen Zentrum für Luft- und Raumfahrt, Einrichtung “Simulations- und Softwaretechnik” in Köln durchgeführt.

2 Stand der Forschung

Maschinelle Lernverfahren werden im Bereich der künstlichen Intelligenz verwendet, um Daten zu beschreiben[6]. Sie dienen der Klassifizierung der Daten und dem Auffinden von Mustern. Auf dieser Grundlage können Vorhersagen getroffen und die Daten entsprechend repräsentiert werden. Beim *Supervised*-Lernen werden Testdaten verwendet, deren Klassen bereits bekannt sind, um neue Daten klassifizieren zu können. Werden diese Testdaten nicht verwendet handelt es sich um ein *Unsupervised*-Lernverfahren, welches selbstständig nach Mustern in den Daten sucht.
k-Nearest-Neighbor [7] ist ein Verfahren zur Klassifizierung von Daten und arbeitet auf Nachbarschaften von Datenpunkten. *Decision Trees* [8] nutzen Attribute von Daten, um zu einer Entscheidung zu kommen und repräsentieren den Entscheidungsweg als Baumstruktur. Sogenannte *Deep Learning*-Verfahren nutzen neuronale Netze zur Klassifizierung und Mustererkennung und orientieren sich an der Funktionsweise des menschlichen Gehirns[9].

Verfahren, wie zum Beispiel *k-Nearest Neighbor* oder *Decision Trees*, weisen drei Probleme auf:
Der *Curse of Dimensionality*[10] entsteht durch zu wenige oder zu viele Dimensionen der Daten. Beispielsweise kann es beim *k-Nearest Neighbor* Verfahren zu wenige Nachbarn für einen Datenpunkt geben, um diesen eindeutig klassifizieren zu können oder eine Aussage über die Nachbarschaft ist durch zu viele Dimensionen erschwert.
Local Constancy beschreibt ein Problem, bei dem sich Daten in einer bestimmten Region nur sehr gering voneinander unterscheiden. Dies kann Allerdings zu Fehlern bei der Generalisierung und unerwünschten Effekten führen (siehe Abbildung 1).
Zuletzt gibt es Datensätze, welche nur sinnvoll reduziert werden können, wenn der projizierte Raum nicht linear ist, sondern durch eine Mannigfaltigkeit beschreiben werden kann[11].
Die genannten Probleme werden mithilfe von neuronalen Netzen gelöst.

[6] Tagliaferri 2017: An Introduction to machine learning
[7] Peterson 2009: K-Nearest Neighbor
[8] Gupta 2017: Decision Trees in machine learning
[9] Jones 2014: Wie Maschinen lernen lernen
[10] Rojas 2015: The Curse of Dimensionality
[11] Joshi 2014: What is Manifold Learning?

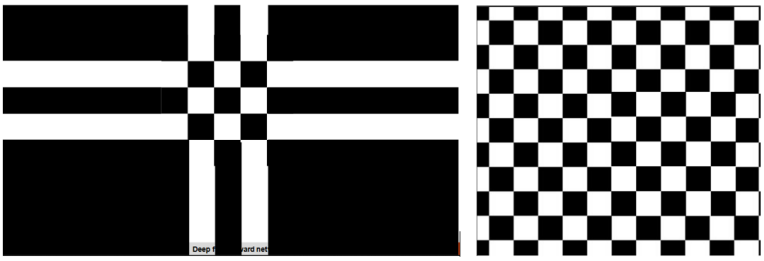


Abbildung 1: Unerwünschter Effekt durch *Local Constancy* (Links), Gewünschte Generalisierung (Rechts) (Staab 2017: Machine Learning and Data Mining)

2.1 Neuronale Netze

Ein neuronales Netz besteht aus Schichten, die sogenannten *Layer*. Jedem Layer werden *Neuronen* zugeordnet, welche Funktionen repräsentieren, die innerhalb des Layers ausgeführt werden. Diese sind zwischen den Layern miteinander verbunden, weshalb von einem Netz gesprochen wird.

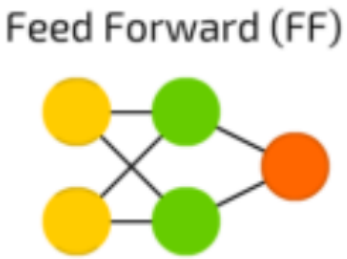


Abbildung 2: *Feed Forward* Netz (van Veenn 2016: The mostly complete chart of Neural Networks, explained)

Es existiert immer ein *Input*- und ein *Output*-Layer. Layer die zwischen diesen liegen werden *Hidden Layer* genannt. *Feed Forward Netze* sind eine einfache Form von neuronalen Netzen, die für *Deep Learning* genutzt werden und nur einen Hidden Layer nutzen (siehe Abbildung 2).

Eine Anwendung für ein solches Modell ist beispielsweise die Erkennung von handgeschriebenen Zahlen. Als Eingabe dient das Bild einer Zahl und jeder Pixel wird zu einem Neuron des Input-Layers, welches ausschließlich den Farbwert des zugehörigen Pixels speichert. Ziel des Modells ist es, die Zahl auf dem Bild zu bestimmen.
Jedem Neuron der anderen Layer wird ein Gewicht (*Weight*) und ein Beeinflussungswert (*Bias*) zugewiesen. Das Gewicht priorisiert jeden Eingabewert und der Bias bestimmt, wie leicht das Neuron zu aktivieren ist. Ein Neuron zu aktivieren bedeutet, dass dieses einen positiven Wert an nach-

folgende Neuronen weitergibt.
Im Falle von *Perzeptronen*, einer Art der Neuronen, wird für einen gegebenen Input x_i folgende Gleichung berechnet [Nie15]:

$$\Sigma_i(w_i * x_i) + b \tag{1}$$

w_i bezeichnet die Gewichte des Neurons und b den Bias. Neuronen können mehrere Eingaben erhalten und für jede Eingabe existiert ein Gewicht, weshalb eine Summe berechnet wird. Sollte der Wert der Gleichung größer als Null sein, wird das Neuron aktiviert und gibt sein Ergebnis an alle Neuronen des nachfolgenden Layers weiter.
Da dieser Wert sehr groß werden kann, werden auch *Sigmoid-Neuronen* verwendet, welche den Wertebereich mithilfe einer *Sigmoid-Funktion* auf $[0, 1]$ beschränken [Nie15]:

$$\frac{1}{1 + exp(-\Sigma_i w_i x_i - b)} \tag{2}$$

Die Sigmoid-Funktion besitzt den Vorteil, dass Veränderungen an den Gewichten oder Bias-Werten nur kleine Veränderungen im Ergebnis zur Folge haben. Es gibt auch andere Funktionen, die für neuronale Netze genutzt werden können, zum Beispiel *ReLu*[12] oder *Softsign*[13]

Auf diese Weise können Neuronen unterschiedliche Merkmale erkennen und jeder Layer erhöht das Abstraktionslevel. Bei dem genannten Beispiel der Erkennung von handgeschriebenen Zahlen können beispielsweise Neuronen des ersten Hidden Layers das Eingabebild filtern. Jedes Neuron besitzt dafür einen anderen Filter, zum Beispiel einen Filter für die Erkennung von horizontalen oder vertikalen Linien. Weitere Hidden Layer erhalten die gefilterten Bilder und können abstraktere und kompliziertere Formen, wie Kreise, Halbkreise oder Ähnliche erkennen.
Der Output-Layer erhält für jede mögliche Klasse einen Zahlenwert, der die Wahrscheinlichkeit angibt, ob es sich auf dem Eingabebild um die entsprechende Zahl handelt.

Damit ein Modell in der Lage dazu ist Zahlen mit hoher Genauigkeit zu bestimmen, muss das Modell trainiert werden. Hierfür werden Testdaten genutzt, bei denen bereits bekannt ist, welchen Klassen sie angehören. Die Differenz zwischen dem gewollten und vom Modell bestimmten Ergebniswert wird als *Kosten* bezeichnet. Ein Modell mit hoher Genauigkeit hat geringe Kosten.
Um die Kosten zu minimieren, müssen die Gewichte und Bias-Werte der Neuronen angepasst werden, da optimale Werte die Genauigkeit der Klassifizierung erhöhen und damit die Kosten senken. Dafür kann das *Gradient*

[12] Becker 2018: Rectified Linear Units (ReLU) in Deep Learning
[13] Serengil 2017: Softsign as a Neural Networks Activation Function

Descent-Verfahren[14] genutzt werden. Gradient Descent findet lokale Minima in Funktionen und kann damit die Kostenfunktion des Modells minimieren.
Die Berechnung kann bei einer großen Menge an Testdaten einige Zeit in Anspruch nehmen. Aus diesem Grund werden die Testdaten in kleinere Gruppen zusammengefasst (*Mini-Batches*) und das Gradient Descent Verfahren nacheinander auf die Gruppen angewandt.
Mithilfe von *Backpropagation*[15] kann das Gradient Descent Verfahren zusätzlich verbessert werden. Es ermöglicht einen Einblick, wie die Veränderung der Gewichte und Bias-Werte die Kostenfunktion beeinflussen.

Es existieren viele verschiedene Arten von neuronalen Netzen[16]. Die bereits genannten *Feed Forward Networks (FFN)* besitzen nur einen Hidden Layer und alle Neuronen sind miteinander verbunden. Viele neuronale Netze verwenden jedoch mehr als nur einen Hidden Layer, zum Beispiel die *Deep Feed Forward Networks (DFF)*. Diese funktionieren wie die FFNs, doch erzielen sie meist bessere Ergebnisse, weil mehr Hidden Layer verwendet werden.
Reccurent Neural Networks (RNN) arbeiten sequentiell. Das heißt, dass die Eingaben der Neuronen voneinander abhängig sind und die Neuronen auf vorherige Ergebnisse Zugriff haben, die bestimmen, wie die aktuelle Eingabe verarbeitet werden soll. Zum Beispiel bei Sprach- oder Texterkennung ist diese Funktion von Vorteil, da Wörter in einem Satz aufeinander aufbauen und ihre Bedeutung sich entsprechend verändern kann.
Die *Convolutional Neural Networks (CNN)* werden vor allem für Bilderkennung genutzt, so wie in dem genannten Beispiel. *Convolutional Layer* dienen dazu die Eingabe zu filtern und bestimmte Merkmale hervorzuheben oder zu unterdrücken. Auf diese Weise können neuronale Netze beispielsweise Objekte, Tiere oder Menschen auf Bildern erkennen (siehe Abbildung 3).

2.2 Techniken zur Visualisierung

Es ist nicht leicht neuronale Netze zu verstehen, da sie oft nicht-lineare Datensätze verwenden und viele Parameter zur Klassifizierung nutzen. Ohne Visualisierung können Fehler nur schwer erkannt und behoben werden, weshalb an Möglichkeiten zur Darstellung von und Interaktion mit neuronalen Netzen gearbeitet wird. *Kahng et al.* haben in einem Paper Design-Ansprüche an eine Visualisierung zusammengestellt [KAKC17].
Wichtig sei es unterschiedliche Formate von Daten zu unterstützen, wie

[14] Donges 2018: Gradient Descent in a Nutshell
[15] Gupta 2017: Deep Learning: Back Propagation
[16] Tchircoff 2017: The mostly complete chart of Neural Networks, explained

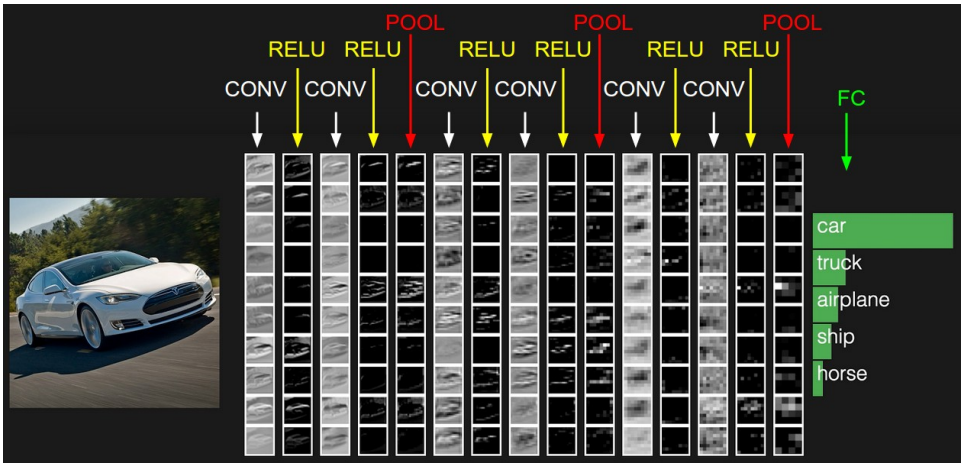


Abbildung 3: Beispiel für ein *Convolutional Neural Network*, welches Objekte in Bildern erkennt (Karpathy 2018: Convolutional Neural Networks (CNNs / ConvNets))

zum Beispiel Text-, Bild- und Audiodaten. Die Menge an Daten müsste reduziert werden, um eine Visualisierung zu ermöglichen. Eine Anwendung sollte mit unterschiedlich komplexen Architekturen zurechtkommen können, weil nicht alle neuronalen Netze linear aufgebaut seien. Damit viele Arten von Modellen unterstützt werden, sollte auf die Generalisierung geachtet werden, so dass mit wenig Aufwand neue Arten in die Visualisierung eingebunden werden können. Um verschiedenen Ansprüchen gerecht zu werden, sollte es dem Benutzer möglich sein Mengen von Daten selbst zu definieren. Zuletzt sollte es möglich sein sowohl einzelne Eingaben, als auch eine Menge von Eingaben im neuronalen Netz nachverfolgen zu können.
Olah et al. von Google beschreiben in ihrem Paper [OSJ+18] die Notwendigkeit von Interfaces, die die Modelle abstrahiert darstellen und aktuelle Methoden, wie Visualisierung von Merkmalen, einbinden.

Verschiedene Techniken zur Visualisierung existieren bereits.
Saliency Maps[17] gehören zu den Verfahren der Segmentierung und werden zum Beispiel bei Bildern genutzt, um Pixel und Bildbereiche hervorzuheben. Für neuronale Netze kann beispielsweise kontrolliert werden, wie Veränderungen der Pixel das Ergebnis einer Klassifizierung beeinflussen. *Simonyan et al.* nutzen diese Technik, um zu kontrollieren, ob die richtigen Regionen im Bild, beispielsweise das zu klassifizierende Objekt, für die Klassifizierung verwendet werden [SVZ13]. Hierfür werden die *Saliency Maps* als Maske über das Eingabebild gelegt (siehe Abbildung 4). Sollte eine nicht-relevante Region, wie zum Beispiel der Hintergrund, markiert werden, deutet dies auf Fehler im neuronalen Netz hin.

Abbildung 4: Beispiel einer *Saliency Map*, die anzeigt, wo im Bild ein Hund erkannt wird ([SVZ13])

Erweiterungen dieser Technik existieren ebenfalls. *Zeiler et al.* nutzen beispielsweise nur positive Gradienten zur Berechnung der *Saliency Maps* [ZF13]. Bestimmte Bereiche des Bildes können auch verdeckt oder gelöscht werden, um Auswirkungen auf die Klassifizierung zu beobachten [FV17].
Auch das *Guided BackProp* Verfahren [SDBR14] und das *DeConvNet* [ZF13] projizieren Ergebnisse der Klassifizierung auf das Eingabebild. Dafür wird die Aktivierung der Neuronen verwendet, um Aktivierungsmuster zu visualisieren (siehe Abbildung 5).

[17] Sharma 2018: What Are Saliency Maps In Deep Learning?

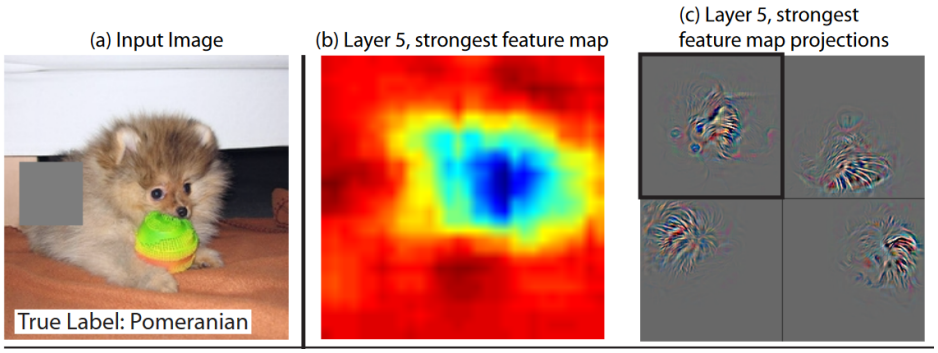


Abbildung 5: Die Aktivierungen auf dem Eingabebild (a) werden als *Heatmap* visualisiert (b) und anschließend auf das Eingabebild projiziert (c) ([ZF13]

Laut *Kindermans et al.* seien diese Ergebnisse jedoch nicht zur Erklärung der internen Vorgänge eines neuronalen Netzes geeignet, da sie nur das Verhältnis vom Signal zu seiner Rauschverteilung reflektieren würden [KSA+18]. Aus diesem Grund haben Kindermans et al. eine eigene Methode vorgestellt. *PatternNet* nutzt eine Rückprojektion, bei der pro Layer die Gewichte des Modells ersetzt werden und die bessere Ergebnisse liefert. *PatternAttribution* erzeugt *Heatmaps*, die den Einfluss eines Bildes auf die Klassifizierung für jedes Neuron anzeigen können(siehe Abbildung 6).

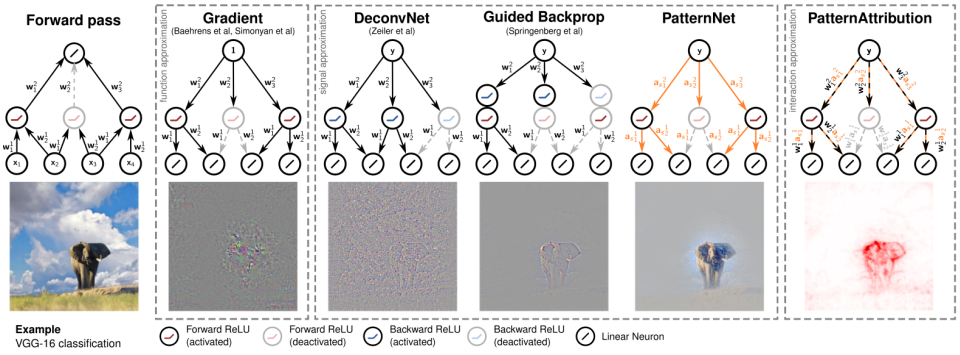


Abbildung 6: Ein Vergleich mehrerer Verfahren, unter anderem mit *PatternNet* und *PatternAttribution* ([KSA+18])

Das *Class Activation Mapping (CAM)* Verfahren von *Zhou et al.* visualisiert Regionen in einem Bild, welche bei der Klassifizierung für eine bestimmte Klasse entscheidend waren [ZKL+15]. Dies funktioniert sowohl für einzelne Layer als auch für mehrere gleichzeitig (siehe Abbildung 7).

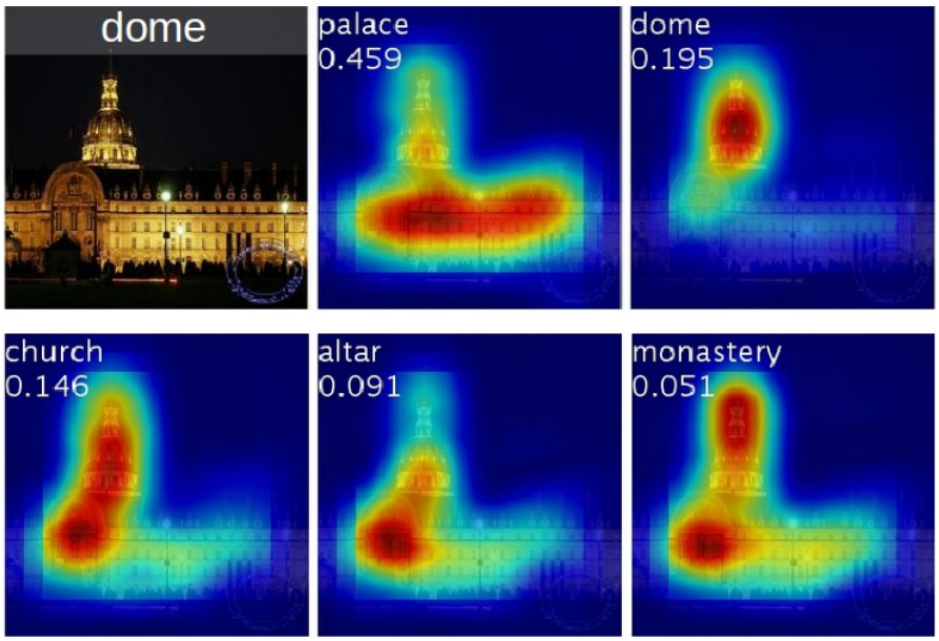


Abbildung 7: Ergebnisse des *CAM*-Verfahrens für unterschiedliche Klassen([ZKL+15])

Um Datenpunkte zu visualisieren wird das *t-SNE*-Verfahren genutzt [vdMH08]. Dieses ist ein Verfahren zur Reduzierung von Dimensionen in Daten. Datenpunkte werden im 2D- oder 3D-Raum, auf Grundlage ihrer Ähnlichkeit zueinander, platziert. Ähnliche Daten liegen nah beieinander, während Daten mit großen Unterschieden weit auseinander liegen(siehe Abbildung 8).

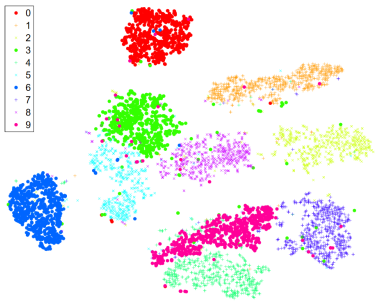


Abbildung 8: Gleiche Daten liegen beim *t-SNE* Verfahren nah beieinander, wodurch Cluster entstehen ([vdMH08])

Für neuronale Netze ist es beispielsweise möglich die abstrakten Vektoren als Datenpunkte zu nutzen und darzustellen, wodurch Fehler des Modells ersichtlich werden können[18].

Im folgenden Abschnitt werden drei wissenschaftliche Arbeiten zur Visualisierung von neuronalen Netzen vorgestellt, welche als Grundlage und Inspirationsquelle für die vorliegende Abschlussarbeit verwendet wurden.

2.2.1 Google

GoogleNet[19] ist ein neuronales Netz, welches im Jahr 2014 den *ImageNet*-Wettbewerb[20] gewann. Auf Grundlage dieses Netzes stellen *Olah et al.* Konzepte für Interfaces vor, welche die Nachvollziehbarkeit von *GoogleNet* verbessern sollen [OSJ+18].

Von besonderem Interesse seien die *Hidden Layer*, denn diese würden die Eingaben auf neue Art und Weise präsentieren. Normalerweise liegen die Ergebnisse der Layer als Vektoren vor. Um diese zu visualisieren benutzen Olah et al. *Feature Visualization*[21] und erstellen sogenannten *semantic dictionaries* (siehe Abbildung 9).

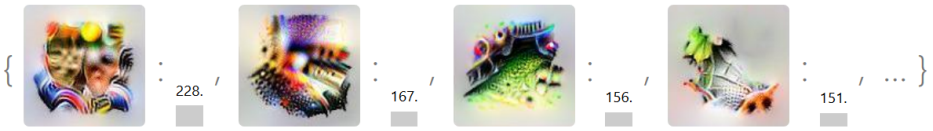


Abbildung 9: Darstellung eines *semantic dictionaries*([OSJ+18])

Semantic dictionaries ordnen die aktivierten Neuronen auf Grundlage ihrer Aktivierungswerte. Jedem Neuron wird mithilfe der *Feature Visualization* ein abstraktes Bild zugeordnet, welches die Funktionsweise des Neurons visualisiert. Auf diese Weise werden Aktivierungen mit „ikonischen Repräsentationen "[OSJ+18] verbunden. Weitere Verfahren, wie zum Beispiel Dimensionsreduktion [22] könnten ebenfalls angewandt werden, um die Ergebnisse zu verbessern.
Für Olah et al. verdeutlichen die *semantic dictionaries*, dass eine formale Beschreibung der Ergebnisse von Neuronen nur schwer möglich sei.

[18]Gupta 2017: Using T-SNE to Visualise how your Model thinks
[19]Szegedy et al. 2014: Going Deeper with convolutions
[20]ImageNet Homepage: ImageNet Homepage
[21]Olah et al. 2017: Feature Visualization
[22]Sorzano et al. 2014: A survey of dimensionality reduction techniques

Als nächste Idee wird die Möglichkeit präsentiert, die Ergebnisse der Neuronen eines Layers zusammenzufassen und auf das Eingabebild zu projizieren. Hierdurch würde der Benutzer einen Einblick erhalten, wie ein bestimmter Layer das Bild interpretiert. Werden mehrere Layer miteinander verglichen entsteht eine Abfolge, die den Prozess des neuronalen Netzes bei der Klassifizierung zeigen würde (siehe Abbildung 10). Um die Aktivierungswerte der Neuronen einzubinden, skalieren Olah et al. die einzelnen Bilder entsprechend.

Abbildung 10: Pro Pixel wird eine Zusammenfassung des *semantic dictionaries* gezeigt und damit jeweils ein Layer visualisiert([OSJ+18])

Auch Olah et al. nutzen *Saliency Maps* (siehe Abschnitt 2.2), jedoch weisen sie auf zwei Probleme hin. Die Pixel der Bilder sind ihrer Meinung nach nicht aussagekräftig genug, um ihren Einfluss auf die Klassifizierung interpretieren zu können, da bereits eine Verschiebung des Bildes Änderungen hervorrufen kann. Außerdem würden Saliency Maps nur eine Klasse gleichzeitig anzeigen und nicht die Ergebnisse der *Hidden Layer* visualisieren. Aus diesem Grund nutzen Olah et al. *Attribution* und verbinden diese mit Saliency Maps.
Der Einfluss auf die Klassifizierung wird mithilfe der abstrakten Bilder ermittelt, welche vorher auf das Eingabebild projiziert wurden (siehe Abbildung 10), anstatt die ursprünglichen Pixel zu betrachten. Für eine bestimmte Klasse kann visualisiert werden, wo im Bild diese erkannt wurde und mit welchem Aktivierungswert. Der Anteil an den einzelnen Klassen kann für jeden Pixel angezeigt werden. So würden laut Olah et al. auch die Zusammenhänge zwischen den Layern dargestellt (siehe Abbildung 11).

Anstatt die Ergebnisse der Neuronen für einen einzelnen Pixel zusammen-

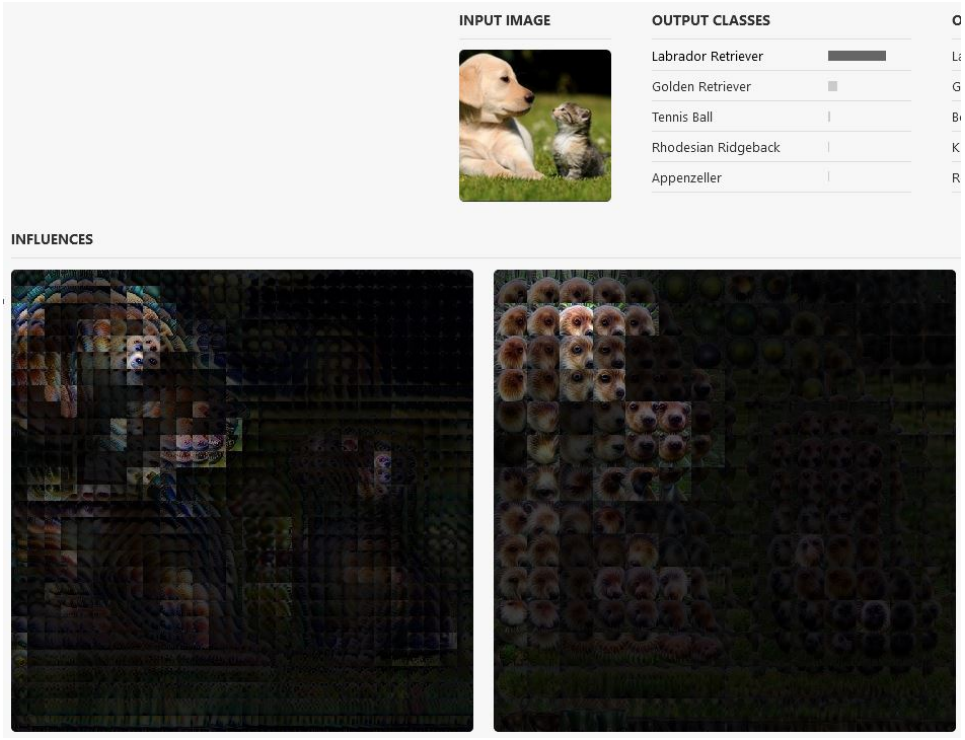


Abbildung 11: Die Klasse *Labrador* wird in den hervorgehobenen Pixeln der Layer erkannt. Je heller, desto größer der Aktivierungswert ([OSJ+18])

zufassen, präsentieren Olah et al. ebenfalls die Möglichkeit die Ergebnisse für alle Pixel zusammenzufassen und auf dem Bild darzustellen (*Channel Attribution*). Hierdurch würde ein weiterer Einblick in die *Hidden Layer* ermöglicht werden (siehe Abbildung 12).

Probleme bei der Visualisierung sehen Olah et al. vor allem in der Menge der zu verarbeitenden Informationen. Außerdem seien ihre genutzten Verfahren verlustbehaftet und würden dadurch potentiell wichtige Aspekte vernachlässigen.
Um die Menge an Informationen zu reduzieren wird *Matrix Faktorisierung*[23] vorgeschlagen. Mithilfe dieser Methode könnten Gruppen von Neuronen zusammengefasst werden, allerdings würden Informationen verloren gehen. Aus diesem Grund müsste der Benutzer entscheiden, welche Informationen priorisiert werden sollen. Das Verständnis der Ergebnisse könnte auf diese Weise erhöht werden.
Auch eine Gegenüberstellung von Modellen wäre ihrer Meinung nach vorstellbar, um die Ergebnisse mehrerer Modelle vergleichen zu können.

[23] Kersting, Weihs 2014: Wissensentdeckung in Datenbanken

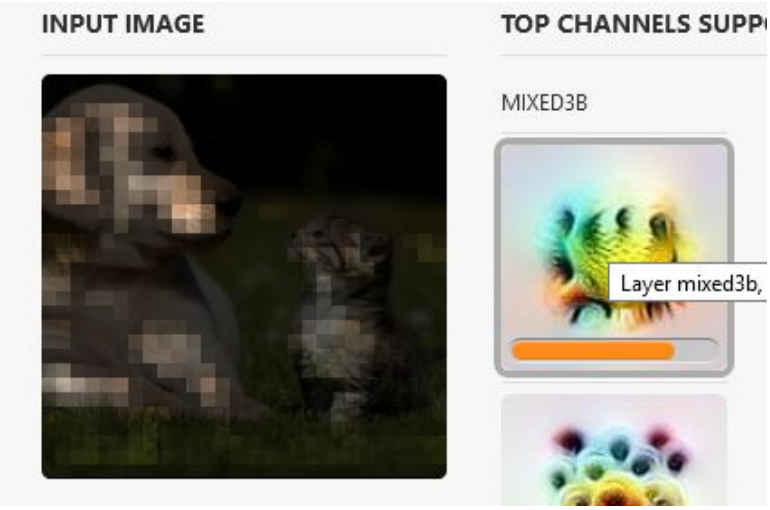


Abbildung 12: Ein Neuron des *MIXED3B*-Layer wird für die Klasse *Labrador* auf den hervorgehobenen Pixeln erkannt ([OSJ+18])

Die Bestandteile von neuronalen Netzen werden von Olah et al. in einer Tabelle einzeln betrachtet und in Zusammenhang gesetzt (siehe Abbildung 13).

Abbildung 13: Mithilfe dieser Tabelle sollen neue Konzepte für Visualisierungen entwickelt werden. ([OSJ+18])

Diese Tabelle soll dazu dienen, mögliche Interfaces zu entwickeln und dabei verschiedene Schwerpunkte zu modellieren. Durch neue Methoden könnten solche Systeme erweitert werden. Hierbei würde auch das Feedback der Benutzer eine wichtige Rolle spielen, um die Interfaces zu verbessern.

2.2.2 ActiVis

ActiVis wurde von *Kahng et al.* im August 2017 vorgestellt [KAKC17]. Drei Aspekte stehen im Fokus der Anwendung: Das gesamte Modell wird als Graph dargestellt und kann vom Benutzer inspiziert werden, während die verwendeten Instanzen und Mengen an Daten eingeschränkt werden kön-

nen. Zudem ist die Anwendung mit verschiedenen Modellen und Daten verwendbar.

Der Benutzer hat die Möglichkeit den Graph eines Modells zu betrachten (siehe Abbildung 14).

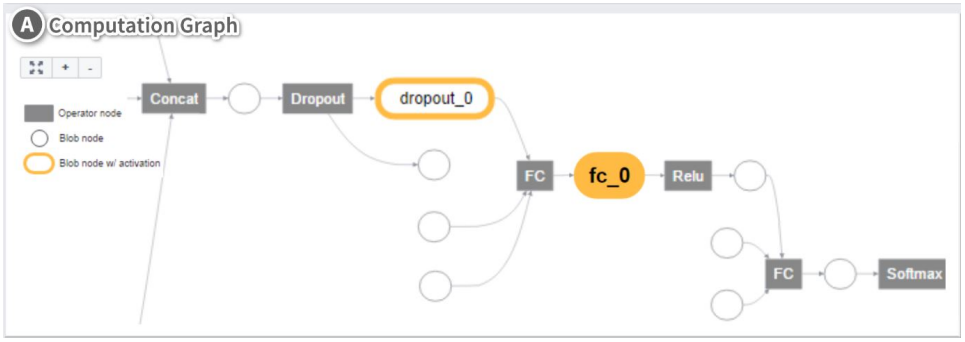


Abbildung 14: Das Modell wird mithilfe von Knoten und Kanten als Graph visualisiert [KAKC17]

Es handelt sich um einen *computational graph*[24], bei dem Variablen als Kanten und Operatoren als Knoten repräsentiert werden. Dies soll dem Benutzer ein besseres Verständnis des Datenflusses ermöglichen. Knoten können ausgewählt werden, worauf sich ein neues Fenster mit Informationen zu den Layern und Neuronen öffnet (siehe Abbildung 15).
Eine *Activation Matrix* stellt den Aktivierungsgrad der Neuronen dar, der *Projected View* zeigt die Klassenverteilung der Datenpunkte und der ausgewählte Knoten im Graph wird eingeblendet, inklusive dem vorangehenden und nachfolgendem Knoten. Es können mehrere Knoten gleichzeitig ausgewählt werden, um Ergebnisse miteinander zu vergleichen.
Wird in dieser Übersicht eine einzelne Instanz selektiert, werden die Ergebnisse der Klassifizierung für die einzelnen Klassen angezeigt (siehe Abbildung 16). Hier werden sowohl korrekt als auch falsch klassifizierte Daten aufgelistet. Die Menge der zu verwendeten Daten (*Subsets*) kann im Vorfeld vom Benutzer eingeschränkt werden. Unter anderem sind die zu verwendenden Klassen, Attribute und Eigenschaften auswählbar.

[24]Sabinasz 2017: Deep Learning From Scratch I: Computational Graphs

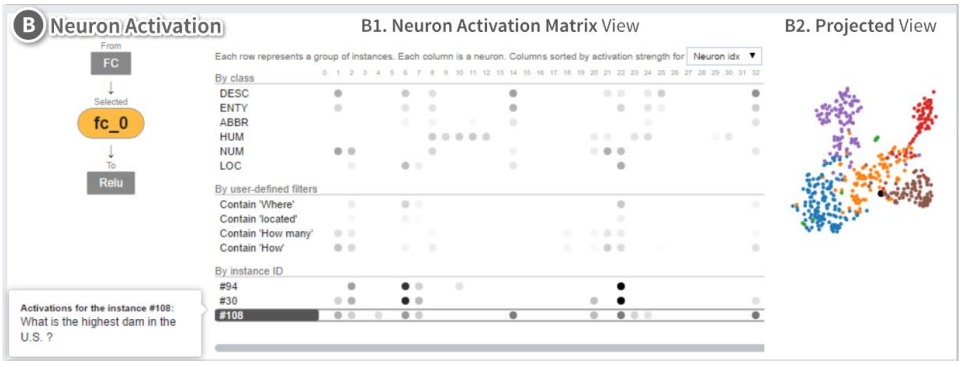


Abbildung 15: Sowohl der ausgewählte Knoten des Graphen also auch Neuronen, Klassen und Instanzen werden visualisiert. Instanz #108 wurde selektiert.[KAKC17]

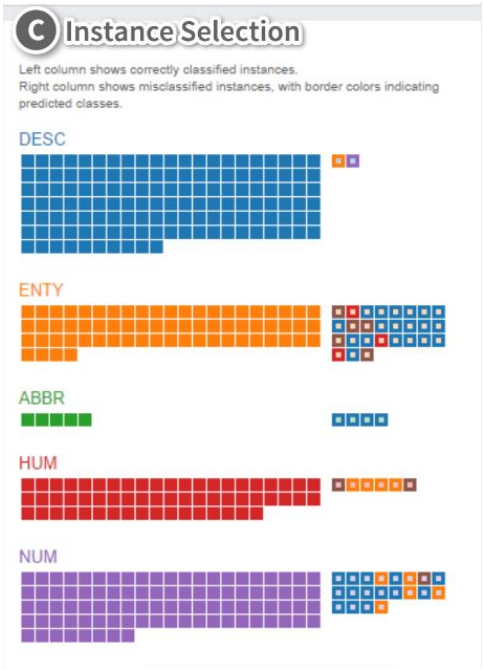


Abbildung 16: Für die Instanz #108 wird die Verteilung der Klassen visualisiert. Links werden korrekt klassifizierte Klassen angezeigt, Rechts die falsch klassifizierten Klassen. [KAKC17]

Auf der *Facebook*-Plattform für maschinelle Lernverfahren, *FBLearner Flow*[25], wurde *AcitVis* zur Nutzung bereitgestellt. Damit Entwickler die Software nutzen können, müssen drei Methoden (*preprocess*, *process*, *postprocess*) dem eigenen Code hinzugefügt werden. Durch Modularisierung werden automatisch die notwendigen Daten generiert, die *FBLearner Flow* benötigt. Dies

[25]Dunn 2016: Introducing FBLearner Flow

ermöglicht es der Software mit unterschiedlichen Datensätzen und Modellen zu arbeiten, wie zum Beispiel Modellen die Texte, Bilder oder Sprachaufnahmen verwenden. Die Interaktionsmöglichkeiten mit dem Modell können vom Entwickler eingeschränkt werden. Sowohl die Knoten, die der Benutzer einsehen kann, als auch die Erstellung von *Subsets* können beschränkt werden.

Mithilfe einer Evaluation haben Kahng et al. festhalten können, dass die Benutzer ihrer Software ihre eigenen Modelle besser verstanden haben, weil sie die Darstellung flexibel ihren Bedürfnissen entsprechend anpassen konnten. Die Darstellung der Trainingsparameter half dem Verständnis der internen Vorgänge des Modells. Durch gezielte Auswahl von *Subsets* konnten die Modelle verbessert werden, indem Daten gefunden wurden, die sich als effektiver für die Klassifizierung herausstellten.

2.2.3 TensorFlow

Im Jahr 2011 startete das *Google Brain*[26] Projekt, in dessen Rahmen das Team von Dean et al. die Software *DiestBelief* entwickelte [27]. Diese Software wurde genutzt, um beispielsweise Modelle für Objekterkennung, Spracherkennung oder Bildklassifizierung zu trainieren und fand unter anderem in der Google Suchmaschine, bei Google-Maps und auch Youtube seine Anwendung.
Darauf aufbauend wurde die Open Source Software *TensorFlow* entwickelt [AAB+16]. *TensorFlow* bietet ein Interface um maschinelle Lernalgorithmen umzusetzen und basiert auf einem Datenflussmodell. Es ist auf vielen Systemen anwendbar und so gestaltet, dass neue Modelle darauf aufbauen können.
Auf Grundlage dieser Software wurde *TensorBoard* entwickelt [AAB+16]. Ziel dieser Anwendung ist es, den Datenfluss-Graphen von TensorFlow zu visualisieren und interne Vorgänge transparenter zu präsentieren. Knoten des Graphen werden zu Blöcken zusammengefasst, um große Modelle darstellen zu können (siehe Abbildung 17). Blöcke mit ähnlichen Strukturen werden optisch hervorgehoben, um Hauptregionen des Graphen zu verdeutlichen.
Der Benutzer kann mit dem System interaktiv arbeiten. Es kann hineingezoomt und der Inhalt von Blöcken betrachtet werden, um genauere Informationen zu erhalten. Auch die Veränderung des Modells über die Zeit kann veranschaulicht werden (siehe Abbildung 18), beispielsweise der Verlauf der Verlustfunktion(*Loss-Function*), die Verteilung von Gewichten in-

[26] Google Brain Homepage
[27] Dean et al. 2012: Large Scale Distributed Deep Networks

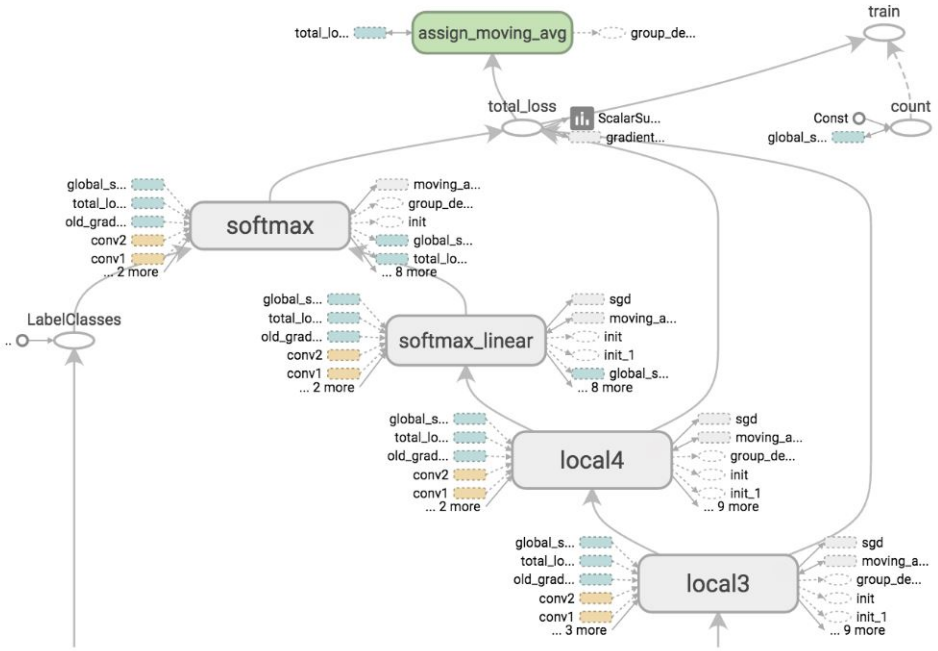


Abbildung 17: Der von *TenorBoard* visualisierte Graph eines Modells. Knoten werden zu Blöcken zusammengefasst. [AAB+16]

nerhalb eines Layers oder die Filter eines *Convolutional Neural Networks*.

Um den Einstieg in neuronale Netze zu erleichtern wurde *TensorFlow Playground* entwickelt [SCS+17]. Diese Anwendung läuft im Browser und visualisiert ein neuronales Netz, welches vom Benutzer manipuliert werden kann.
Neben den Layern werden die Neuronen als Boxen angezeigt (siehe Abbildung 19). Zwischen den Neuronen bestehen Verbindungen, deren Farbe die Gewichte symbolisiert. Jedes Neuron wird durch eine *Heat-Map* visualisiert, welche die Funktionsweise optisch darstellen soll. Die Neuronen des ersten Layers dienen beispielsweise dazu, die Daten in horizontale oder vertikale *Cluster* einzuordnen, während spätere Layer komplexere *Cluster* erstellen.
Heat-Maps können auf das Ergebnis projiziert werden, indem ein Neuron ausgewählt wird. Auch die Testdaten können im Ergebnis eingeblendet werden. Für das Training des Netzes kann der Benutzer Trainingsdaten auswählen oder die Aktivierungsfunktion verändern (ReLu, Sigmoid, Linear, tanH).
Diese Funktionen sollen helfen die Mathematik hinter neuronalen Netzen besser zu verstehen und die Effekte von Trainingsparametern kennen zu lernen. Auch der individuelle Einfluss von Neuronen auf das Ergebnis soll

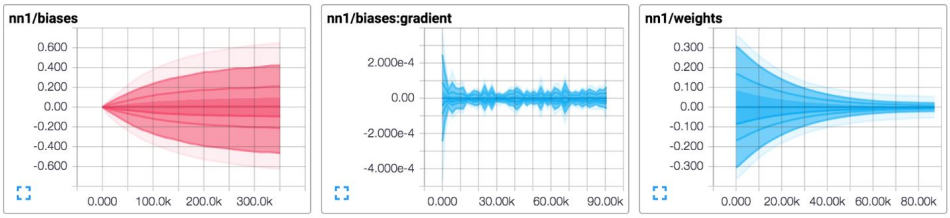


Abbildung 18: Die Veränderung von Werten, wie den Bias- und Gewichtswerten, kann visuell dargestellt werden [AAB+16]

verständlicher werden. In Zukunft sollen weitere Möglichkeiten hinzugefügt werden mit dem neuronalen Netz zu interagieren und die Anwendung soll auch für erfahrene Benutzer weiter ausgebaut werden.

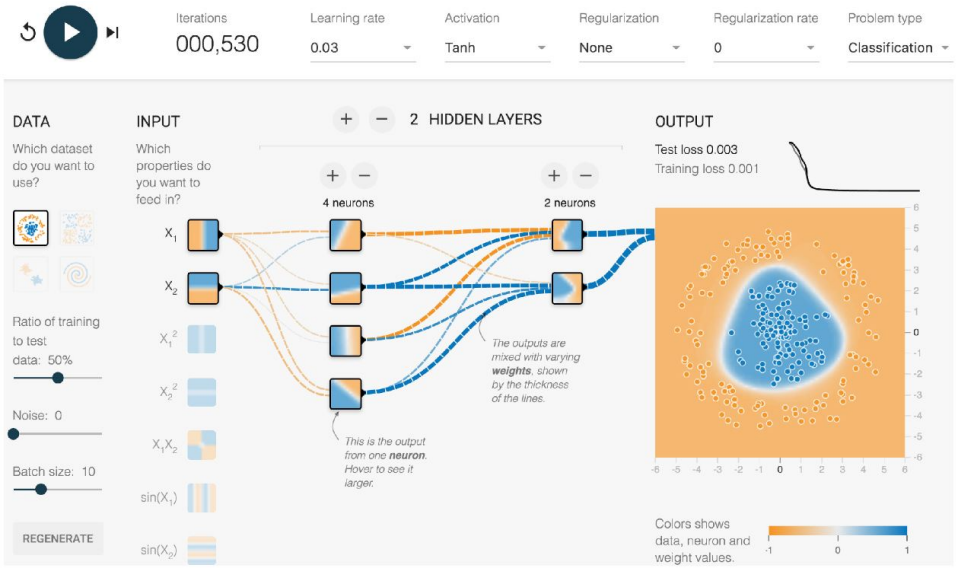


Abbildung 19: Die Benutzeroberfläche von *TensorFlow Playground* visualisiert Neuronen, Layer und die Ergebnisse des neuronalen Netzes [SCS+17]

2.3 Fazit

In diesem Kapitel wurden verschiedene wissenschaftliche Arbeiten vorgestellt, welche sich mit der Visualisierung von neuronalen Netzen und deren Anforderungen auseinandersetzen. Anwendungen wie *ActiVis* oder *TensorFlow Playground* legen Wert darauf, dass der Benutzer mit dem Modell interagieren kann. Informationen können gezielt ein- und ausgeblendet werden und dies soll dabei helfen die Modelle besser zu verstehen. Durch Anpassung der Parameter und der Visualisierung der herbeigeführten Veränderungen sollen die Modelle verbessert werden können.
Der Einfluss der Layer und Neuronen auf das Ergebnis der Klassifizierung steht hierbei im Mittelpunkt. Für die genaue Visualisierung werden verschiedene Ansätze genannt und umgesetzt. Vor allem die Projektion der Zwischenergebnisse auf die Eingabe und ihr Einfluss auf die Klassifizierung werden genutzt, um die Funktionsweise von Neuronen und Layer besser zu verstehen. Dabei wird auch auf abstrakte Bilder zurückgegriffen, die die Funktionsweise symbolisieren sollen.

Alle gefundenen und hier vorgestellten Anwendungen visualisieren die Informationen mithilfe von 2D-Interfaces. Aus diesem Grund entstand die Idee ein Konzept zu entwickeln, welches ein neuronales Netz mithilfe eines 3D-Interfaces visualisiert. In das Konzept sollten Erkenntnisse aus den, in diesem Kapitel vorgestellten, Arbeiten einfließen, um die Informationen in angemessener Form zu präsentieren. Als Leitfrage für den zu entwickelnden Prototypen diente die Hypothese, dass eine Visualisierung im dreidimensionalen Raum die Nachvollziehbarkeit und Begründbarkeit ebenfalls verbessern würde.

3 Verwendete Software

Zur Umsetzung des Konzepts für die vorliegende Abschlussarbeit und Visualisierung eines neuronalen Netzes wurden die *Unreal Engine* (Abschnitt 3.1) und *TensorFlow* (Abschnitt 3.2) genutzt. In diesem Kapitel werden die beiden Software-Programme vorgestellt und ihr Funktionsumfang erläutert.

3.1 Unreal Engine

Die *Unreal Engine* wurde von Epic Games im Jahr 1998 entwickelt und ist seit dem 14. März 2014 in der Version 4 verfügbar[28]. Bis zu einem festgelegten Umsatz ist die Engine kostenfrei nutzbar[29].
Neben einer Grafik-Engine, die *DirectX 11* und Effekte wie *Global Illumination* oder *GPU Partikel Simulation* unterstützt[30], bietet die Unreal Engine einen Level-Editor, eine Physik- und Sound-Engine und unterstützt Netzwerkfunktionen. Auch mit *Virtual Reality* Hardware wie der *HTC Vive* oder der *Oculus Rift* ist die Engine kompatibel.
Als Programmierumgebung bietet die Unreal Engine eine *Visual Scripting* Umgebung an[31]. Diese nutzt *Blueprints* als Klassen und ein Graph, bestehend aus Knoten (Funktionen) und Kanten (Daten), visualisiert den programmierten Code. Aus diesem Graphen lässt sich *C++-Code* automatisch generieren. Auch andere Programmiersprachen wie Python werden mithilfe von Erweiterungen unterstützt [32].

Damit Entwickler ihre eigene künstliche Intelligenz (*KI*) entwickeln und in Projekten nutzen können, stellt die Unreal Engine *Behaviour Trees* und *Environment Query Systems* bereit.
Behaviour Trees ermöglichen es, das Verhalten der *KI* festzulegen und *Environment Query Systeme* erfassen Daten aus der Spielumgebung, welche von der *KI* ausgewertet werden können[33]. Mithilfe dieser Systeme lassen sich Gegenspieler in Videospielen programmieren oder Simulationen durchführen.
Auch Algorithmen aus dem Bereich der maschinellen Lernverfahren, wie zum Beispiel der *Q-Learning* Algorithmus[34], können umgesetzt werden. Dieser Algorithmus ermöglicht es der künstlichen Intelligenz beispielswei-

[28] Unreal Engine - Homepage
[29] Unreal Engine - EULA FAQ
[30] Unreal Engine - Dokumentation: *Rendering and Graphics*
[31] Unreal Engine - Dokumentation: *Blueprints Visual Scripting*
[32] Unreal Engine - Dokumentation: *Plugins*
[33] Tran 2018: Unreal Engine 4 Tutorial: Artificial Intelligence
[34] Waktins et al. 1992: Q-Learning

se Rätsel selbstständig lösen zu können[35]. Durch eine Verbindung mit *TensorFlow* kann auch ein neuronales Netz dafür genutzt werden (siehe auch Abschnitt 3.2.1). Auch das Plug-In *UnrealCV* in Verbindung mit *OpenAI Gym* nutzt *Reinforcement Learning*[36] in der Unreal Engine, um beispielsweise in einer Szene Gegenstände wie Pflanzen oder Türen zu finden und mit diesen zu interagieren[37].

Alternativen Es existieren auch Alternativen zur Ureal Engine, wie zum Beispiel die *Unity Engine*[38]. Diese Software bietet ein eigenes System für künstliche Intelligenz, das *ML-Agents Toolkit*[39]. Mit diesem lässt sich ebenfalls *Reinforcement Learning* in Projekte integrieren. Es existiert bisher kein TensorFlow Plug-In (siehe Abschnitt 3.2.1), doch mithilfe des *Amazon Web Services*[40] steht *TensorFlow* zur Verfügung und kann für neuronale Netze genutzt werden[41].

3.2 TensorFlow

Für die vorliegende Abschlussarbeit wurde sich für die Unreal Engine als Software entschieden, da bereits Erfahrung im Umgang mit der Software vorhanden war und das TensorFlow Plug-in eine vielversprechende Lösung zur Nutzung von TensorFlow versprach. Im folgenden Abschnitt wird das Plug-in vorgestellt und seine bereitgestellten Funktionen erläutert.

3.2.1 TensorFlow Plug-in

Von Jan Kaniewski wurde ein *TensorFlow Plug-in* entwickelt, welches die Verwendung von maschinellen Lernverfahren in der Unreal Engine ermöglicht[42]. Für die maschinellen Lernverfahren wurde die Open Source Software *TensorFlow* eingebunden und über C++, Python und Blueprints können die Funktionen der Software genutzt werden.
Eine Installation ist nicht notwendig, es wird aber eine aktuelle *CUDA*-Installation vorausgesetzt, falls die GPU genutzt werden soll[43]. Das Plug-in wird nach dem Herunterladen in den Hauptordner des eigenen Projekts

[35] Github: Unreal AI
[36] Simonini 2018: An introduction to Reinforcement Learning
[37] UnrealCV - Homepage
[38] Unity - Homepage
[39] Unity - Machine Learning
[40] Amazon Web Services - Machine Learning
[41] Lange 2017: Bringing gaming to life with AI and deep learning
[42] Github: TensorFlow Plug-in
[43] NVIDIA CUDA Homepage

kopiert und die notwendigen Dateien werden beim Starten des Projekts automatisch installiert. Es steht ebenfalls ein Projekt als Beispiel zum Download zur Verfügung, welches im Abschnitt 3.2.2 erläutert wird.

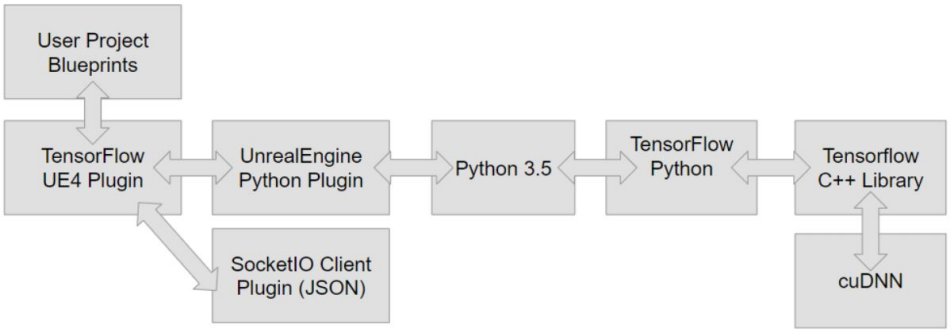


Abbildung 20: Die Architektur hinter dem *TensorFlow Plug-in* (Github)

In Abbildung 20 ist die Architektur dargestellt, in die das Plug-in eingebettet wurde. Es dient als Schnittstelle zwischen den Klassen des Unreal Projekts (*Blueprints*), Python und einem JSON-Modul.
Die *UnrealEngine Python* Komponente dient vor allem der Nutzung von Python, aber auch der Unterstützung von *Multithreading* und dem Packetverwaltungsprogramm *pip*[44]. Für den Datenaustausch zwischen Python und der Unreal Engine wird eine *JSON*[45] Komponente verwendet.

Python API Das zu verwendende Python Skript, in dem beispielsweise ein neuronales Netz vom Entwickler implementiert wurde, muss in den Unterordner „*{Project Root Folder}/Content/Scripts*“ kopiert werden. Zusätzlich müssen die Dateien *tensorflow*, *unreal_engine* und *TFPluginAPI* in das Skript importiert werden. Jede Klasse, die das Plug-in verwenden soll, muss von der Klasse *TFPluginAPI* abgeleitet sein und die Methoden *onSetup*, *onJsonInput*, *onBeginTraining* und *getApi* implementieren (siehe Abbildung 21). *onBeginTraining* soll die Logik für das Training des maschinellen Lernverfahrens beinhalten und Eingaben für das Modell werden über *onJsonInput* übergeben. *getApi* muss eine Instanz der selbstdefinierten Klasse zurückgeben.

[44] pip - Dokumentation
[45] Introduction JSON

```python
import tensorflow as tf
import unreal_engine as ue
from TFPluginAPI import TFPluginAPI

class ExampleAPI(TFPluginAPI):

        #expected optional api: setup your model for training
        def onSetup(self):
                pass

        #expected optional api: parse input object and return a result object, which will be converted to json for UE4
        def onJsonInput(self, jsonInput):
                result = {}
                return result

        #expected optional api: start training your network
        def onBeginTraining(self):
                pass

#NOTE: this is a module function, not a class function. Change your CLASSNAME to reflect your class
#required function to get our api
def getApi():
        #return CLASSNAME.getInstance()
        return ExampleAPI.getInstance()
```

Abbildung 21: Der grundlegende Aufbau jedes Python Skripts, welches mit dem Plug-in verwendet werden soll.

Blueprint API Jedem *Blueprint* in der Unreal Engine kann die *Tensorflow-Component* hinzugefügt werden (siehe Abbildung 22). Diese Komponente benötigt den Namen des Python-Skripts, welches genutzt werden soll. Einstellungen, wie zum Beispiel die Auswahl von *Multithreading* können ebenfalls in der Engine vorgenommen werden.
Alle Daten, die zum Python-Skript geschickt werden sollen, müssen in das *JSON-Format* formatiert werden. Dies ist mit der Methode *Encode Json*, welche von der Unreal Engine bereitgestellt wird, möglich und mithilfe von selbsterstellten *Struct*-Variablen können auch komplexe Datenstrukturen versendet werden (siehe Abbildung 23). Auf diese Weise können auch Bilder übertragen werden. Funktionen des Python-Skripts werden mithilfe von Strings aufgerufen, die den Namen der Funktion definieren.

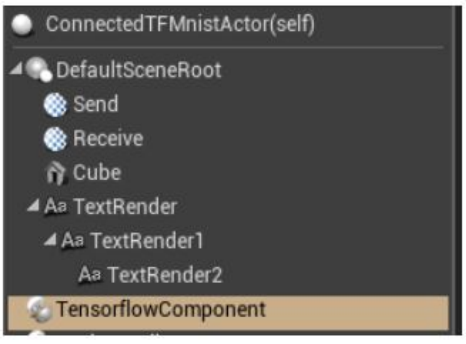


Abbildung 22: Jedem Blueprint kann eine *TensorflowComponent* hinzugefügt werden, um Zugriff auf die Funktionen zu ermöglichen.

Die Unreal Engine nutzt ein sogenanntes *Event-System*[46], um auf Ereignisse (*Events*) zu reagieren und Teile des programmierten Graphen auszuführen. Es sind bereits drei Arten von Events vorgegeben:
OnInputResult wird aufgerufen, sobald die Funktion *onJsonInput* (siehe Absatz **Python API**) beendet wurde und übermittelt die Daten, die von dieser Funktion zurückgegeben werden. Das Event *OnTrainingComplete* wird aktiviert, wenn die *onBeginTraining* Methode beendet wurde. Standardmäßig wird die Dauer des Trainings übermittelt.
Zuletzt ist es möglich mit *OnEvent* neue Events zu definieren. Hierfür muss neben den Daten auch ein Name für das Event übermittelt werden, welcher in der Unreal Engine kontrolliert werden kann, um anschließend spezielle Funktionen aufrufen zu können.

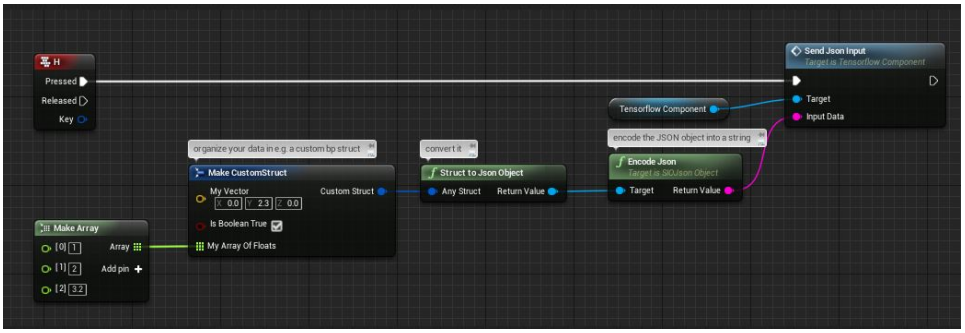


Abbildung 23: Alle Dateien müssen mit *Encode JSON* ins JSON-Format umgewandelt werden. Sogenannte *Struct*-Variablen ermöglichen komplexe Datenstrukturen zu definieren.

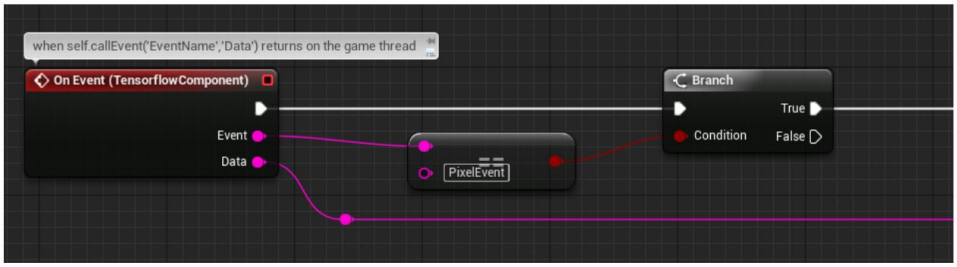


Abbildung 24: Eigene Events können mit *OnEvent* definiert werden. Der Name des Events muss zusätzlich zu den Daten übergeben werden.

Weitere Funktionen Ebenfalls in das Plug-in integriert ist eine Sound-Komponente, die es ermöglicht Sound aufzunehmen und wiederzugeben.

[46] Unreal Engine - Dokumentation: Events

Diese Funktion ist für maschinelle Lernverfahren gedacht, die zum Beispiel für Spracherkennung verwendet werden.

3.2.2 MNIST Demo

Neben dem *TensorFlow Plug-in* aus Abschnitt 3.2.1 stellt Jan Kaniewski eine Demo bereit, um die implementierten Funktionen zu präsentieren[47]. Im folgenden Abschnitt wird diese Demo und ihr Umfang erläutert, weil auf dieser die Abschlussarbeit und die programmierte Visualisierung (siehe Abschnitt 5) aufbauen.

MNIST Netzwerk Als Grundlage für die Klassifizierung liegt der *MNIST Datensatz*[48] in dieser Demo vor. Dieser Datensatz beinhaltet 60.000 Beispielbilder und 10.000 Testdaten von handgeschriebenen Zahlen. Da die Bilder in ihrer Größe bereits normalisiert und zentriert sind ist dieses ein beliebtes Beispiel, um maschinelle Lernverfahren zu nutzen, ohne die Daten noch vorher bearbeiten zu müssen.

Unreal Engine Blueprints Beim Starten der Unreal Engine Demo erscheint eine Umgebung zur Visualisierung der Klassifizierung (siehe Abbildung 25) und das Training des Modells beginnt. Auf der linken Seite des Bildschirms wird die vom Benutzer aktuell ausgewählte Zahl angezeigt, welche über die Tastatur geändert werden kann. In der Mitte der Szene befindet sich zunächst eine schwarze Textur, über der Informationen des Modells angezeigt werden, wie zum Beispiel die Dauer des Trainings oder das Ergebnis der Klassifizierung. Eine Auswahl der Trainingsdaten wird auf der rechten Seite, während des Trainings, eingeblendet.
Neben der Auswahl der Eingabezahl über die Tastatur ist es auch möglich die Zahl per Maus oder Touchpad zu zeichnen. Hierfür muss eine Webseite im Browser geöffnet werden, die die Eingabe ermöglicht. Sobald eine Zahl gezeichnet wurde, wird diese auf der Textur in der Mitte der Szene angezeigt und anschließend klassifiziert.

Zur Umsetzung der Klassifizierung wurden hauptsächlich zwei Blueprints von Kaniewski programmiert. Das *ConnectedTFMnistActor* Blueprint dient zur Verarbeitung der Eingaben des Benutzers. Bei Eingabe einer Zahl über die Tastatur überprüft die Klasse, welche Zahl eingegeben wurde und ändert die angezeigte Textur in der Szene entsprechend. Sobald der Nutzer

[47] Github - TensorFlow Plug-in Demo
[48] MNIST Datensatz - Download

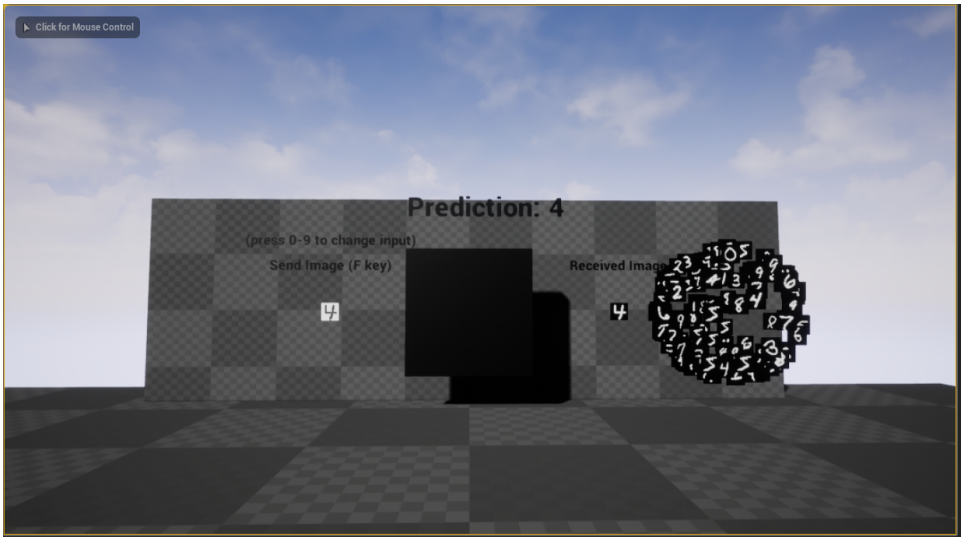


Abbildung 25: In der Demo war bereits eine Visualisierung der Klassifizierung enthalten.

die *F-Taste* drückt, wird die ausgewählte Zahl an die TensorFlow Komponente geschickt. Mit der *G-Taste* hat der Benutzer die Möglichkeit das Training anzuhalten oder auszuführen.
Das Event *OnTrainingComplete* ist zuständig dafür, dass die übergebene Trainingsdauer als Text angezeigt wird und bei dem Event *OnInputResults* liest die Klasse das Ergebnis der Klassifizierung aus. Die Bilder der Testdaten werden angezeigt, sobald *ConnectedTFMnistActor* das Event *PixelEvent* von TensorFlow empfängt. Um während des Trainings die Verlustfunktion und Genauigkeit des Netzwerks zu visualisieren, werden die entsprechenden Daten beim Event *TrainingsUpdateEvent* ausgelesen.

Das zweite bereitgestellte Blueprint ist das *TensorFlowComponent*-Blueprint. Dieses beinhaltet alle notwendigen Funktionen um Daten zwischen der Unreal Engine und dem Python-Skript auszutauschen. Beispielsweise wird die Methode *SendJsonInput* aufgerufen, sobald der Benutzer die *F-Taste* gedrückt hat. Sollte der Benutzer die Zahl im Browser per Hand eingeben wird die Methode *SendJsonInputTexture* genutzt, welche zunächst das übertragene Bild in ein *ImageStruct*[49] konvertiert und dann an das Python-Skript sendet. Auch die Möglichkeit neudefinierte Funktionen aufzurufen ist in dieser Klasse implementiert.
Mithilfe von *Event Dispatchern*[50] kann die TensorFlow Komponente auch mit der *ConnectedTFMnistActor*-Klasse kommunizieren. Diese Art von Events wird weitergeleitet an andere Klassen, um dort Methoden aufrufen zu kön-

[49] Unreal Engine - Dokumentation: Struct Variables
[50] Unreal Engine - Dokumentation: Event Dispatchers

nen.

Python Skript Das verwendete Python Skript für die Demo heißt „*mnist-KerasCNNopt.py*“ und besteht aus zwei Klassen. Die Klasse *MnistKeras* erbt von *TFPluginAPI*, wie in Abschnitt 3.2.1 beschrieben. In dieser Klasse werden die benötigten Funktionen implementiert. Innerhalb dieser Klasse befindet sich die Klasse *StopCallback*, welche abgeleitet ist von der *Keras Callback Klasse*[51]. Diese Klasse dient dazu die Verlustfunktion und Genauigkeit des neuronalen Netzes in festen Intervallen auszulesen und an die Unreal Engine zu schicken, wo die Werte dem Benutzer im Level und dem Konsolenfenster angezeigt werden. Hierfür wird die Methode *callEvent* genutzt (siehe Abbildung 26).

```
self.outer.callEvent('TrainingUpdateEvent', logs, True)
```

Abbildung 26: Mit dem *TrainingUpdateEvent* werden Daten (*logs*) während des Trainings an die Unreal Engine geschickt

outer bezeichnet eine Instanz der *TFPluginAPI*-Klasse, welche die Funktion *callEvent* besitzt. Der Funktion wird ein String als Name für das Event übergeben, eine Variable *log*, in der sich die Daten befinden und eine Boolean-Variable. Wird Letztere als *True* übergeben, wird das *JSON-Format* genutzt. Auch Testdatenbilder werden in regelmäßigen Abständen auf diese Weise übertragen, um sie zu visualisieren (siehe Abbildung 25).
Eine Instanz der *StopCallback* Klasse wird erzeugt, sobald der Benutzer das Level startet und die Methode *onSetup* aufgerufen wird. Diese Instanz wird als Klassen-Variable gespeichert, um sie später nutzen zu können.

Sobald das neuronalen Netz eine Eingabe von der Unreal Engine erhält, wird die Methode *onJsonInput* aufgerufen. Bei dieser MNIST-Demo handelt es sich um Bilddaten, welche in einem JSON-Objekt *jsonPixels* übergeben werden (siehe Abbildung 27).

```
{"pixels":[...], "size":{"x":<...>,:"y":<...>}}
```

Abbildung 27: Der Inhalt des JSON-Objekts für Bilddaten. *pixels* enthält die Pixelwerte, *size* die Größe des Bildes.

Unter dem Eintrag *pixels* sind die Bilddaten enthalten und als 1-dimensionales

[51] Keras Dokumentation - Callbacks

Array gespeichert. Die Höhe und Breite des Bildes wird im Eintrag *size* hinterlegt. Um mit den Daten arbeiten zu können müssen die Daten zunächst in ein *numpy-Array* transformiert werden, welches das richtige Format erhält, mit der *TensorFlow* Methoden arbeiten können (siehe Abbildung 28).

```
#prepare the input
x_raw = [jsonInput['pixels']]
x_raw = np.reshape(x_raw, (1, 28, 28))

ue.log('image shape: ' + str(x_raw.shape))
#ue.log(stored)

#convert pixels to N_samples, height, width, N_channels input
    tensor
x = np.reshape(x_raw, (len(x_raw), 28, 28, 1))
```

Abbildung 28: Das Eingabebild muss für die Klassifizierung formatiert werden.

Um den TensorFlow Graphen ausführen zu können, muss eine zuvor erstellte *Session*[52] geladen werden. Diese dient als Schnittstelle zwischen dem Programm und der ausführenden Maschine und speichert Informationen über den Graphen (siehe Abbildung 29).

```
K.set_session(self.session)

with self.session.as_default():
 output = self.model.predict(x)

 ue.log(output)

 #convert output array to prediction
 index, value = max(enumerate(output[0]), key=operator.
     itemgetter(1))

 result['prediction'] = index
 result['pixels'] = jsonInput['pixels'] #unnecessary but useful
     for round trip testing
```

Abbildung 29: Der Ablauf der Klassifizierung. Es wird eine *Session* genutzt, um den Graphen nutzen zu können.

Die *with*-Umgebung dient als Kontextmanager und beendet eine *Session* automatisch, nachdem alle Vorgänge innerhalb des Graphen beendet wurden. In dieser Demo wird das Modell genutzt, um die Eingabe des Benut-

[52] TensorFlow Dokumentation - Session

zers mit der *predict*-Methode[53] zu klassifizieren. Das Ergebnis der Klassifizierung wird als Array (*output*) zurückgegeben, in dem für jede der Zehn möglichen Klassen (0-9) ein Wahrscheinlichkeitswert gespeichert wurde. Der höchste Wert wird ausgelesen und der entsprechende Index als Ergebnis der Klassifizierung zurückgegeben.

Zum Trainieren des Modells wird die Methode *onBeginTraining* genutzt. Bei jedem Aufruf der Methode wird die aktuelle *Session* zurückgesetzt, damit die Werte des Graphen neu beschrieben werden können. Verschiedene Trainingsparameter können vor dem Training angepasst werden, unter anderem die *Batch*-Größe, die Anzahl an Durchläufen (*Epochs*) und an Klassen.
Der MNIST-Testdatensatz steht in *TensorFlow* zur Verfügung und kann in Trainings- und Testdatensätze aufgeteilt werden. Die *Keras*-Klasse[54] übernimmt dabei die Konvertierung der Vektoren in Klassenmatrizen. Es steht nur das sequentielle Modell zur Verfügung, dessen Layer linear aufgebaut sind. Weitere Modelle können implementiert werden, indem eine eigene Klasse von *Keras* abgeleitet wird.
Layer werden dem Modell mithilfe der *add*-Methode hinzugefügt[55] (siehe Abbildung 30).

```
# create model
model = Sequential()
model.add(Conv2D(30, (5, 5), input_shape=input_shape, activation
    ='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(15, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

Abbildung 30: Layer werden dem Modell hinzugefügt und Einstellungen übergeben.

In dem gegebenen Beispiel wurden jeweils zwei *Convolutional Layer* und *Pool Layer* verwendet, darüber hinaus ein *Dropout*- und *Flatten Layer* . Drei *Dense Layer* bilden die letzten Layer des neuronalen Netzes. Es stehen verschiedene Einstellungen pro Layer zur Verfügung [56]. Den *Convolutional Layer* wird beispielsweise die Anzahl an Neuronen, die Größe der Filter-

[53] TensorFlow Dokumentation - Keras Model
[54] TensorFlow Dokumentation - Keras
[55] TensorFlow Dokumentation - Sequential
[56] TensorFlow Dokumentation - Layers

maske und die Aktivierungsfunktion vorgegeben. *Pooling Layer* wird vorgegeben um welchen Faktor die Eingabe verkleinert werden soll.
Je nach gewünschtem Modell und Art der Eingabedaten stehen weitere Layer zur Verfügung, welche zum Beispiel auch 3D-Datensätze unterstützen.

```
model.fit(x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test),
    callbacks=[self.stopcallback])
score = model.evaluate(x_test, y_test, verbose=0)
ue.log("mnist keras cnn training complete.")
ue.log('Test loss:' + str(score[0]))
ue.log('Test accuracy:' + str(score[1]))
```

Abbildung 31: Das Modell wird trainiert und anschließend evaluiert.

Nachdem die Layer dem Modell hinzugefügt wurden, wird das Modell kompiliert, wobei die Verlustfunktion, der Optimierer und die Metrik angegeben werden müssen, mit denen das Modell arbeiten soll. Mit der *fit*-Methode wird das neuronale Netz, entsprechend der gewählten Einstellungen, trainiert. Nach dem Training kann die *evaluate*-Methode genutzt werden, um die Genauigkeit und die Verlustfunktion ausgeben zu lassen (siehe Abbildung 31).

Das Modell und die verwendete *Session* zur Erstellung des Modells müssen am Ende der Methode gespeichert werden, damit die *onJsonInput*-Methode das Modell nutzen kann, um eine Eingabe zu klassifizieren. Es ist ebenfalls möglich diese Informationen in einer Datei speichern zu lassen, um das Modell wiederzuverwenden.

4 Konzept

Die vorliegende Abschlussarbeit beschäftigt sich mit dem Thema der *Nachvollziehbarkeit und Begründbarkeit von maschinellen Lernverfahren*. Aus diesem Grund war das Ziel ein neuronales Netz zu visualisieren und Informationen der unterschiedlichen Layer bereitzustellen.
Wie bereits in Kapitel 2 erwähnt wurde, gibt es bereits Ansätze zur Visualisierung. Diese konzentrieren sich hauptsächlich auf die Darstellung im zweidimensionalen Raum. Deshalb sollte im Rahmen dieser Arbeit eine *3D-Engine* zur Visualisierung genutzt werden. Solche *Engines* bieten unter anderem Werkzeuge zur Erstellung und Gestaltung von interaktiven 3D-Szenen (siehe Abschnitt 3.1).
Aus der Literaturrecherche wurde abgeleitet, dass vor allem die Möglichkeit der Ein- und Ausblendung von Informationen und die angemessene Darstellung der Ergebnisse der einzelnen Layer für den Benutzer wichtig sein werden (siehe Kapitel 2).

Einbindung von TensorFlow Für die Unreal Engine stand ein *TensorFlow Plug-in* zur Verfügung (siehe Kapitel 3.2.1), welches für die Erstellung des neuronalen Netzes genutzt werden sollte. Die Verwendung des Plug-ins war unkompliziert und es gab bereits eine Demo (siehe Kapitel 3.2.2), die die Funktionen präsentierte. Aus dieser Demo wurde ein erstes Konzept für den späteren Datenfluss erstellt (siehe Abbildung 32).

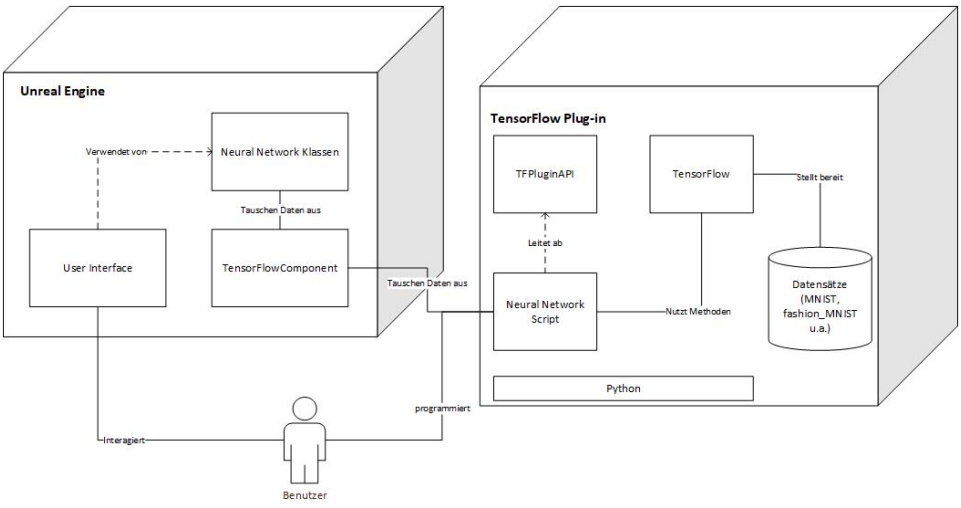


Abbildung 32: Die Komponenten der Visualisierung und ihre Abhängigkeiten.

Ein Benutzer sollte über Menüs (*User Interface*) in der Unreal Engine mit

dem neuronalen Netz interagieren können und sobald Informationen angezeigt werden sollen, würde eine Anfrage an die TensorFlow Komponente (*TensorFlowComponent*) geschickt werden. Für die Anfrage sollten Daten übergeben werden, welche die genauen Informationen spezifizieren. Die TensorFlow Komponente sollte diese Daten als *JSON-Objekt* formatieren und an das zu verwendende Python-Skript (*Neural Network Script*) senden, welches vom Benutzer programmiert wurde. Dieses Skript sollte auf die Informationen die *TensorFlow* bereitstellt zugreifen, formatieren und diese an die *Unreal Engine* zurückschicken. In der Unreal Engine sollten die Daten an die jeweilige Klasse (*Neural Network Klassen*) weitergeleitet werden, wo diese aus dem *JSON-Format* ausgelesen, gespeichert und schließlich im *User Interface* angezeigt werden würden. Die Kommunikation zwischen der Unreal Engine und dem Python-Skript sollte durch das Event-System erfolgen (siehe Kapitel 3.2.1).

Erstellung von neuronalen Netzen Die Erstellung eines neuronalen Netzes sollte im *Python-Skript* vorgenommen werden und die Visualisierung sollte beliebig große Modelle unterstützen. Aus diesem Grund sollte ein Algorithmus entwickelt werden, welcher die Informationen über den Aufbau des Modells dazu verwendet, das neuronale Netz automatisch in der Szene zu erstellen. Für dieses Konzept wurde ein Klassendiagramm erstellt, welches den Aufbau von Modellen veranschaulicht (siehe Abbildung 33).
Jedes neuronale Netz sollte aus einer beliebigen Anzahl an Layern bestehen. Diese würden sich in verschiedene Unterarten, wie zum Beispiel *Convolutional Layer* oder *Dense Layer* unterteilen. Jeder Layer sollte aus Neuronen, welche ebenfalls in entsprechende Unterarten aufgeteilt sind, bestehen und mit dem jeweiligen Layer verwendet werden. Die Unterteilung sollte dazu dienen nur notwendige und vorhandene Informationen für eine Art von Layer anzeigen zu lassen.
Layer und Neuronen sollten mithilfe von geometrischen Formen im dreidimensionalen Raum visualisiert werden. Für Layer sollten flach-skalierte Würfel (*Cubes*) und für Neuronen Kugeln (*Spheres*) dienen, welche intuitiv miteinander assoziiert werden sollten. Das neuronale Netz müsste auch die Eingabe des Benutzers anzeigen, wie auch das Ergebnis der Klassifizierung.

Darstellung großer Modelle Für sehr große neuronale Netze muss die Anzahl an geometrischen Objekten in der Szene reduziert werden, um eine Visualisierung in Echtzeit zu ermöglichen. Das Konzept hierfür orientierte sich an dem *Level-of-Detail (LoD)*-Verfahren, welches unter anderem

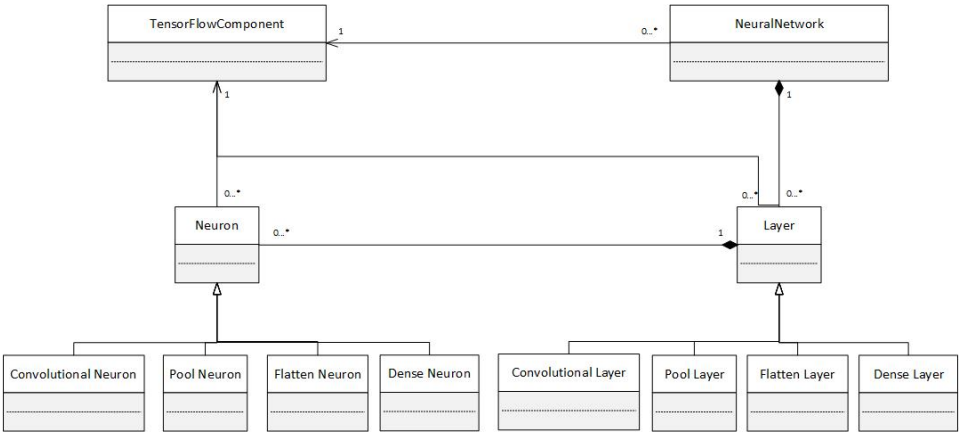


Abbildung 33: Das Klassendiagramm für den Aufbau von neuronalen Netzen. Layer und Neuronen werden in unterschiedliche Arten eingeteilt.

in Videospielen seine Anwendung findet [57]. Es werden unterschiedliche Auflösungen von 3D-Objekten erstellt, wobei die Auflösung der Anzahl an *Eckpunkten* entspricht. Je nach Entfernung zum Spieler wird eine andere Modellauflösung zur Darstellung gewählt. Diesem Prinzip folgend sollte das Modell, je nach Entfernung des Benutzers, unterschiedlich detailliert dargestellt werden. Umso näher sich der Benutzer zum neuronalen Netz befindet, desto mehr Layer und Neuronen sollten angezeigt werden.
Als Alternative wurde sich überlegt, die Neuronen eines Layer gezielt ein- und ausblenden zu lassen. Dies sollte ebenfalls Rechenleistung sparen und darüber hinaus dem Benutzer das neuronale Netz übersichtlicher präsentieren.

Menüs Es können viele Informationen für neuronale Netze zur Visualisierung genutzt werden (siehe auch Kapitel 2). Aus diesem Grund sollten die Menüs übersichtlich gestaltet und die Informationen angemessen dargestellt werden. Der Benutzer sollte jederzeit die Möglichkeit haben die Informationen zu reduzieren und sich die Veränderung der Werte, wie zum Beispiel der *Bias-Werte*, anzeigen zu lassen. Für die Performance sollten die Informationen nur aktualisiert werden, wenn der Benutzer die entsprechenden Menüs öffnet.
Jedes Neuron sollte ein Menü besitzen und die Informationen sollten in Kategorien eingeteilt werden. Für jede Kategorie würde ein eigenes Untermenü erstellt werden, um die Menüs übersichtlich zu gliedern und dem Benutzer die Möglichkeit zu geben, nur gewünschte Informationen einblenden zu lassen.

[57] Unity Dokumentation - Level of Detail

Ein Hauptmenü sollte dazu dienen Einstellungen vor dem Training des neuronalen Netzes vorzunehmen und das Training neu starten zu können. Werte des Modells, wie zum Beispiel die Dauer des Trainings oder die Genauigkeit der Klassifizierung, sollten ebenfalls in ein eigenes Menü integriert werden, damit der Benutzer diese Werte jederzeit kontrollieren kann.

Interaktion Der Benutzer sollte mithilfe der Maus und Tastatur mit der Anwendung interagieren können. Mit den *WASD*-Tasten würde sich ein Benutzer in der Szene bewegen können und mit der Maus wäre das Umsehen in der Szene möglich.
Beschriftungen der Layer sollten es dem Benutzer erleichtern sich in der Szene zu orientieren und spezielle Farben sollten dazu dienen, ausgewählte Objekte optisch hervorzuheben. Neuronen könnten zum Beispiel mithilfe der Maus ausgewählt werden, woraufhin sich die Farbe der Geometrie ändern könnte, um zu signalisieren, dass das entsprechende Neuron ausgewählt wurde. Weitere Tasten der Tastatur sollten verwendet werden, um spezielle Menüs, wie das Hauptmenü, aufrufen und schließen zu können.

5 Umsetzung

Das in Kapitel 4 vorgestellte Konzept zur Visualisierung eines neuronalen Netzes wurde innerhalb von drei Monaten umgesetzt. Im folgenden Abschnitt werden die umgesetzten Inhalte thematisiert und es wird erklärt, wie und mit welchen Hilfsmitteln das Modell visualisiert wurde.

5.1 UserActor_Blueprint

Für den Benutzer wurde die *UserActor_Blueprint* Klasse erstellt, welche die Interaktion in der Szene ermöglicht und die Menüs zugänglich macht, die im Benutzer-Fenster eingeblendet werden können. Wenn die Szene gestartet wird, wird die Eingabe für den Nutzer freigeschaltet und das *Hauptmenü* und das *CameraInterface*-Menü (siehe Abschnitt 5.3) werden der Klasse als Variable (*_setup_Menue*, *_cameraView*) hinzugefügt . Hierzu dienen die Methoden *createSetup* und *createCameraView*.
Sobald das Training gestartet werden soll wird aus dieser Klasse die Methode *TraningStart* der TensorFlow Komponente aufgerufen und die Trainingsparameter aus dem Hauptmenü übergeben. Anschließend wird das Training gestartet.
Aktiviert der Benutzer die rechte Maustaste wird der Mauszeiger, der standardmäßig ausgeblendet ist, eingeblendet. Dies dient dazu mit dem Netzwerk interagieren zu können. Solange sich der Benutzer durch die Szene bewegt könnte ein Mauszeiger störend wirken, weshalb dieser wieder ausgeblendet werden kann durch erneutes betätigen der rechten Maustaste. Der aktuelle Zustand des Mauszeigers wird von der Variable *_mouseActive* gespeichert.
Drückt der Benutzer die M-Taste wird das Hauptmenü mit der Methode *ShowMenue* geöffnet und mit der N-Taste wird die Methode *showCameraView* aufgerufen, welche das *CameraInterface*-Menü anzeigt.

5.2 Das neuronale Netz

In Kapitel 4 wurde in Abbildung 33 die Klassenstruktur für ein neuronales Netz vorgestellt. Diese Struktur wurde in der *Unreal Engine* umgesetzt und die einzelnen Klassen und ihre Funktionen werden im folgenden Abschnitt erläutert.

5.2.1 Netzwerk Klasse

Ein neuronales Netz wird mithilfe der Klasse *NeuralNetwork* in eine Szene eingebunden. Visuell wird das Netzwerk nicht repräsentiert. Es wird le-

diglich durch seine *Layer* und *Neuronen* visualisiert. Jedes Netzwerk besitzt einen eindeutigen Namen (*_id*), damit auch mehrere Netzwerke einer Szene hinzugefügt werden können. Die Hauptaufgabe der Klasse ist es, das Modell automatisch zu visualisieren.

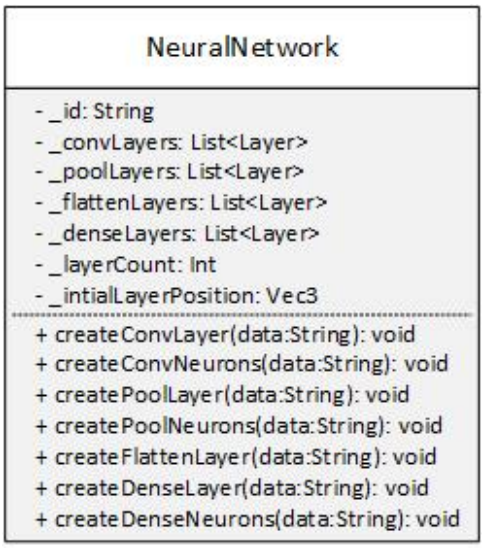


Abbildung 34: Die Klasse *NeuralNetwork*, welche zur Erstellung des Modells in einer Szene genutzt wird.

An der Startposition eines Modells wird die Eingabe des Benutzers als Textur angezeigt. Darauf folgen die *Layer*, welche in der Klasse gespeichert sind. Für jede unterstützte Art von Layer (*Convolutional Layer, Pool Layer, Flatten Layer, Dense Layer*) existiert eine eigene Liste (siehe Abbildung 34). Mithilfe einer initialen Position(*_initialLayerPosition*), welche durch die Position des *Actors*[58] in der Szene bestimmt wird, können die Layer automatisch platziert werden. Bei der Erstellung der Layer wird die *_initialLayerPosition* um 500 Einheiten entlang der *x-Achse* verschoben. Dieser Wert ergab eine ausreichende Distanz zwischen den Layern, damit sich der Benutzer zwischen ihnen bewegen kann (siehe Abbildung 35).

Für jede Art von Layer wurde eine eigene Methode zur Erstellung geschrieben (*createConvLayer, createPoolLayer, createFlattenLayer, createDenseLayer*), wobei jede Methode gleich aufgebaut ist. Sobald die notwendigen Daten zur Erstellung aus dem *Python Skript* empfangen wurden (siehe Abschnitt 5.4), wird die übergebene Namensliste ausgelesen und für jeden Namen ein Layer erstellt. Dafür wird eine *while-Schleife* verwendet, welche die Variable *_layerCount* als Abbruchbedingung nutzt. Jeder Layer erhält einen *Na-*

[58] Unreal Engine Dokumentation - Actors

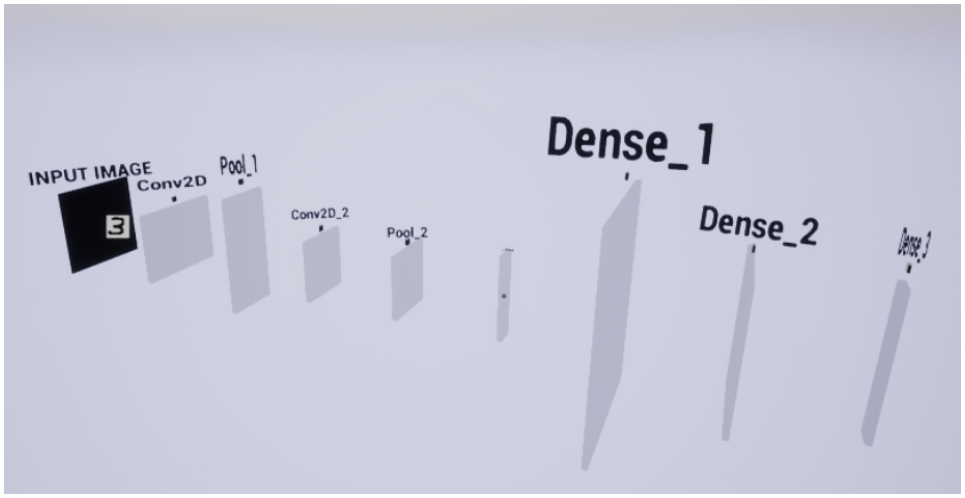


Abbildung 35: Das neuronale Netz wird durch Layer repräsentiert. Am Anfang des Modells wird die Eingabe des Benutzers angezeigt.

men als String-Variable (siehe Abschnitt 5.2.2) und wird der entsprechenden Liste hinzugefügt. Nur die Klasse des Layers, die erstellt wird, ändert sich (siehe Abbildung 36).

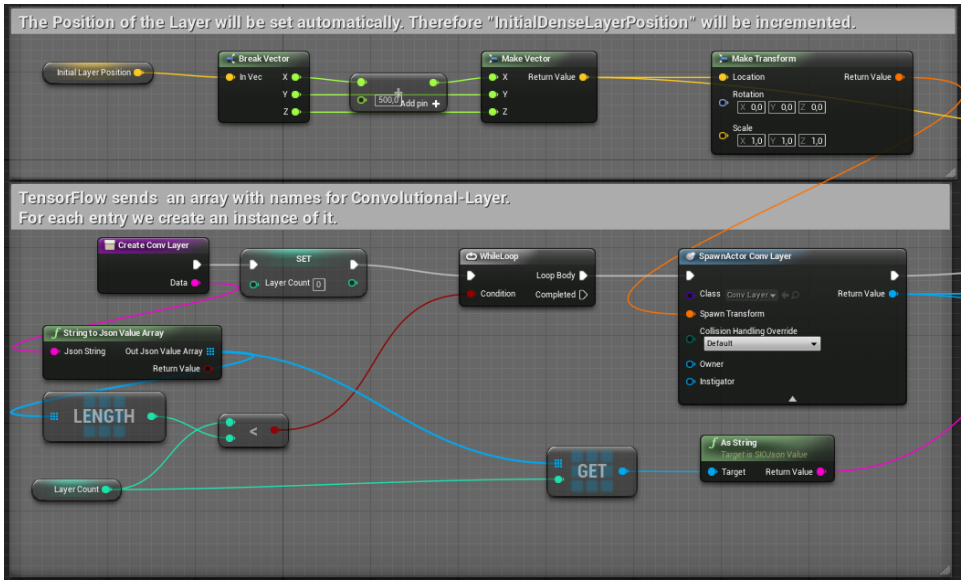


Abbildung 36: *createConvLayer* erstellt die *Convolutional Layer* und platziert diese automatisch in der Szene.

Nachdem die Layer erstellt wurden, können die Neuronen hinzugefügt werden. Auch hier existiert für jede Art von Neuron (siehe Abbildung 33)

eine eigene Methode (*createConvNeurons*, *createPoolNeurons*, *createDenseNeurons*). Wie bei der Erstellung der Layer, funktionieren die Methoden identisch und nur die Klasse des Neurons, die erstellt wird, ist unterschiedlich. Innerhalb der Methoden wird durch die Liste der jeweiligen *Layer* iteriert und die Informationen zur Erstellung der Neuronen (Gesamtanzahl, Anzahl in der Breite, Anzahl in der Höhe) werden an die Methode *createNeurons* des Layers weitergegeben (siehe Abbildung 37).

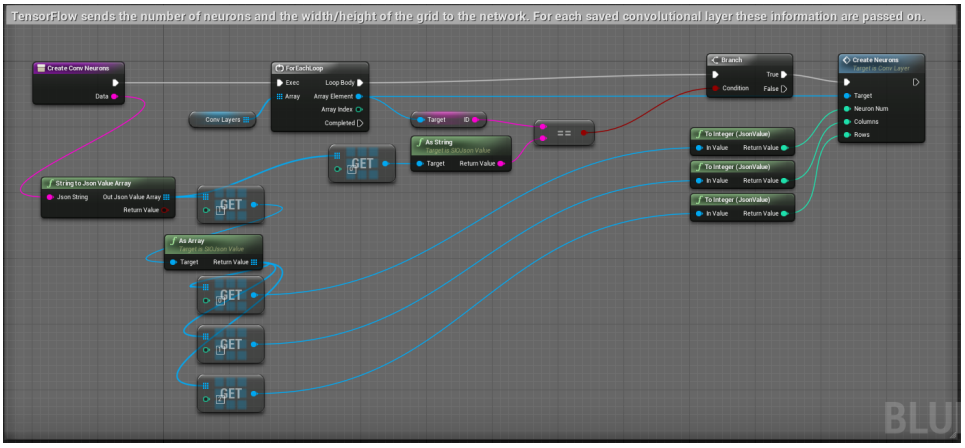


Abbildung 37: *createConvNeuron* gibt die Informationen zur Erstellung der *Convolutional Neuronen* an die *Convolutional Layer* weiter.

Den genannten Methoden entsprechend, empfängt die *NeuralNetwork* Klasse *Events* vom Python Skript (siehe Abschnitt 5.4). Die Namen der *Events* entsprechen den Namen der Methoden, um einen eindeutigen Bezug herstellen zu können.
Somit ermöglicht diese Klasse die automatisierte Erstellung eines neuronalen Netzes und die Logik der Visualisierung der Struktur des Modells wurde zentral in einer Klasse programmiert. Für den Benutzer soll sich auf diese Weise die Erstellung der Visualisierung einfacher gestalten, da nur die Namen der Layer und die Anzahl an Neuronen pro Layer übergeben werden müssen.

5.2.2 Layer Klassen

Jeder *Layer* wird durch einen *Cube*, der entlang einer Achse skaliert wurde, repräsentiert (siehe Abbildung 35). Auf diese Weise soll das Netzwerk als *Schichtenmodell* visualisiert werden, welches Platz in der Szene spart. Um jeden *Layer* in der Szene identifizieren zu können, wird sein Name über der Geometrie angezeigt. Auch um Daten vom Python Skript zu erhalten

ist der Name wichtig, da alle Layer eines Typs die gleichen *Events* empfangen. Unter jedem Namen ist ein kleiner Quader platziert worden, mit dem der Benutzer interagieren kann. Klickt der Benutzer auf diesen, werden die *Neuronen* des Layers ein- oder ausgeblendet.

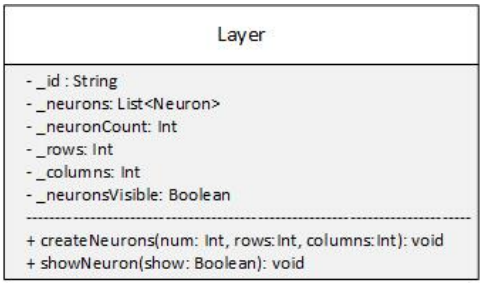


Abbildung 38: Jede Layer-Klasse besitzt die hier gezeigten Variablen und Funktionen.

In der Liste *_neurons* speichern die Layer ihre zugehörigen Neuronen (siehe Abbildung 38). Mithilfe der Methode *createNeurons* werden diese erstellt. Die Methode erwartet die Anzahl an Neuronen (*neuron num*) und zwei Werte, die über die Anordnung in der Szene entscheiden (siehe Abbildung 39).

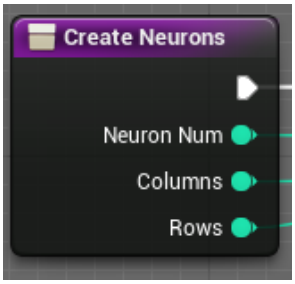


Abbildung 39: Die Neuronen werden in einem zweidimensionalen Raster angeordnet. Reihen und Spalten legt der Benutzer fest.

rows und *columns* entscheiden darüber, in wie vielen Reihen und Spalten die Neuronen platziert werden. Es wird immer ein zweidimensionales Raster (*Grid*) angelegt (siehe Abbildung 40). Sobald die Methode aufgerufen wird, wird der Geometrie eine Größe, auf Grundlage der Anzahl an Neuronen, zugeordnet. Die gewählten Werte der Skalierung ergaben sich durch Ausprobieren und stellen sicher, dass alle Neuronen innerhalb der Geometrie des Layers liegen. Auf diese Weise soll der Benutzer bereits anhand der Größe des Layers sehen können, wie viele Neuronen enthalten sind.

Anschließend werden die Neuronen, entsprechend der übergebenen An-

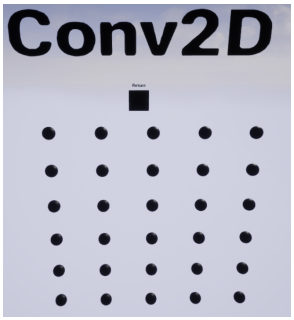


Abbildung 40: Die Neuronen werden in einem zweidimensionalen Raster angeordnet. Reihen und Spalten legt der Benutzer fest.

zahl *neuron_num*, erstellt. Die Klasse des Neurons entspricht der Art des Layers. Jedes Neuron erhält eine eindeutige *ID* und der Name des zugehörigen Layers wird als *String* gespeichert (siehe Abschnitt 5.2.3). Um Grafikspeicher zu sparen werden die Neuronen zunächst nicht gerendert, indem die Option *Hidden in Game* aktiviert wird (siehe Abbildung 41) .

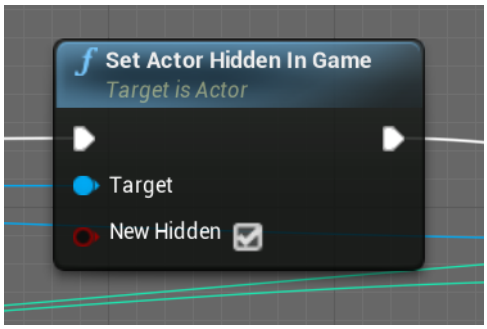


Abbildung 41: Mithilfe der Option *Hidden in Game* lassen sich Objekte einer Szene vom Rendering ausschließen.

Im nächsten Schritt werden die Neuronen in einem gleichmäßigen Raster platziert. Es werden dafür zwei verschachtelte *for-Schleifen* genutzt, welche die Positionen in y- und z-Richtung bestimmen. Dabei wurden die Parameter für den Abstand der Neuronen so gewählt, dass genug Platz zwischen diesen ist, um mit den Menüs interagieren zu können. Auch die Skalierung der Neuronen musste dafür angepasst werden.
showNeurons ist eine Methode, welche aufgerufen wird, sobald der Benutzer den Quader unterhalb des Namens anklickt. Je nachdem ob die Neuronen sichtbar sind, was durch die Variable *_neuronsVisible* gespeichert wird, werden die Neuronen ein- oder ausgeblendet. Dafür iteriert die Methode durch die Liste an Neuronen und aktiviert oder deaktiviert das Attribut *Hidden in Game*. Die Geometrie des Layers wird ausgeblendet, solange die

Neuronen sichtbar sind. Auf diese Weise soll es dem Benutzer ermöglicht werden nur die Neuronen anzeigen zu lassen, die als relevant empfunden werden. Darüber hinaus soll es die Übersichtlichkeit des Modells verbessern.

Das Ziel der Visualisierung der Layer war es, den Aufbau des neuronalen Netzes zu visualisieren. Die Schichten sollen den Ablauf des Netzwerks verdeutlichen und der Benutzer soll sich durch die Anzeige von Namen besser orientieren können. Das Ein- und Ausblenden von Neuronen dient der Individualisierung des Modells für den Benutzer.
In den folgenden Abschnitten werden die einzelnen Klassen der Layer vorgestellt und ihre individuellen Funktionen erläutert.

Convolutional Layer In den *Convolutional Layern* werden *Convolutional Neuronen* gespeichert (siehe Abschnitt 5.2.3). Sobald das Event *ReceiveConvData* des Python Skripts empfangen wird von der Klasse *Conv_Layer*, werden die JSON-Daten für ein Neuron an dieses weitergeleitet. Mithilfe der Variable *_activeNeuron* (siehe Abbildung 42) erhalten nacheinander alle Neuronen, die in der Klasse gespeichert wurden, ihre Ergebnisse (siehe Abschnitt 5.4). Dazu wird die Methode *receiveOutput* der Neuronen aufgerufen.

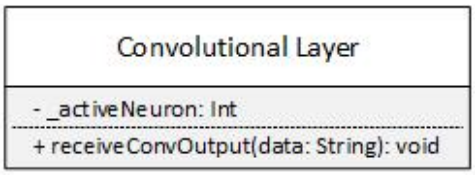


Abbildung 42: Die Klasse der *Covolutional Layer*

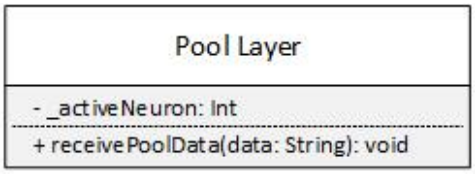


Abbildung 43: Die Klasse der *Pool Layer*

Pool Layer Pool Neuronen werden in der *Pool_Layer* Klasse (siehe Abbildung 43) gespeichert. Wie auch bei den Convolutional Layern werden die

Ergebnisse des neuronalen Netzes durch ein Event, *ReceivePoolData*, empfangen und mithilfe der Variable *_activeNeuron* an die Neuronen weitergeleitet.

Flatten Layer Die Klasse *Flatten_Layer* enthält im Gegensatz zu den anderen Layern keine *TensorFlow Komponente*, weil es bei dieser Art von Layer nur ein Neuron gibt, welches die Informationen darstellt. Aus diesem Grund dient die Klasse nur dazu, das Neuron zu speichern und empfängt keine Daten des Modells.

Dense Layer Als letzte Layer Art wurden die *Dense Layer* mit der Klasse *Dense_Layer* umgesetzt. Sobald das Event *receiveFireRate* empfangen wird, wird die namensgleiche Methode aufgerufen (siehe Abbildung 44) und die JSON-Daten an alle *Dense Neuronen* weitergeleitet. Dafür wird die Methode *receiveFireRate* der Neuronen genutzt (siehe Abschnitt 5.2.3).

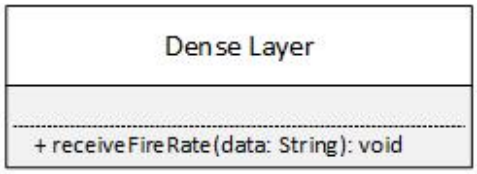


Abbildung 44: Die Klasse der *Dense Layer*

5.2.3 Neuronen Klassen

Die Aufgabe der Klassen für Neuronen ist die Speicherung der von *TensorFlow* zur Verfügung gestellten Informationen. Für die Repräsentation in der Szene wurde sich an dem Paper von *Kahng et al.* orientiert, indem Kreise zur Repräsentation von Neuronen genutzt wurden ([KAKC17]). In der 3D-Szene werden die Neuronen als *Kugel*(*Sphere*) visualisiert. Diese geometrische Form spart sowohl Grafikspeicher als auch Platz in der Szene.

In jeder Klasse der Neuronen (siehe Abbildung 45) wird eine eindeutige Identifikationsnummer (*_id*) gespeichert, zusätzlich zum Namen des zugehörigen Layers (*_layer*). Diese beiden Informationen dienen dazu, das Neuron im Netzwerk einordnen zu können und sollen damit die Übersichtlichkeit verbessern. Angezeigt werden die Informationen in Menüs, welche mithilfe der *Widget Blueprints*[59] erstellt wurden (siehe Abschnitt 5.3). Das verwendete *Widget-Blueprint* wird in der Variable *_interface* gespeichert.
Alle Neuronen einer Art empfangen die gleichen *Events* der TensorFlow

[59] Unreal Engine Dokumentation - Widget Blueprints

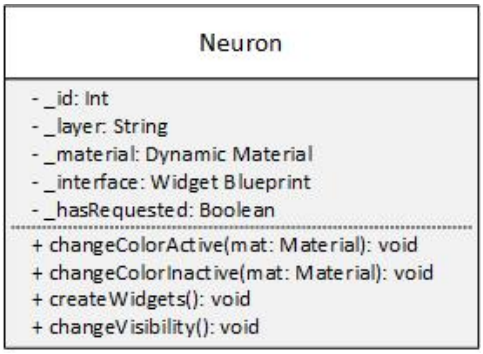


Abbildung 45: Methoden und Attribute, welche jede Neuronen-Klasse besitzt.

Komponente. Um sicherzustellen, dass das richtige Neuron die vom Benutzer angefragten Informationen erhält, wurde die Boolean-Variable *_hasRequested* erstellt. Sobald ein Nutzer ein Neuron aktiviert, wird diese Variable auf *True* gesetzt und nur das ausgewählte Neuron kann Informationen empfangen.
Damit sich ausgewählte- von nicht ausgewählten Neuronen visuell unterscheiden, wurden die Methoden *changeColorActive* und *changeColorInactive* implementiert. Sobald der Mauszeiger sich über einem Neuron befindet, wird die Farbe mit *changeColorActive* zu einem leuchtenden Weißton geändert (siehe Abbildung 46).

Abbildung 46: Ein ausgewähltes Neuron (links) erhält eine weiße, leuchtende Farbe und nicht-ausgewählte Neuronen (rechts) sind standardmäßig schwarz.

Verlässt der Mauszeiger das Neuron wird die ursprüngliche, schwarze Farbe mit der *changeColorInactive* Methode wiederhergestellt. Klickt der Benutzer auf das Neuron, um die Menüs und Informationen aufzurufen, bleibt die weiße Farbe erhalten, bis die Menüs geschlossen werden.
Die Menüs selber werden mit der Methode *changeVisibility* angezeigt, welche überprüft, ob die Menüs bereits sichtbar sind. Sollten sie sichtbar sein, werden sie ausgeblendet, ansonsten eingeblendet (siehe Abbildung 47).

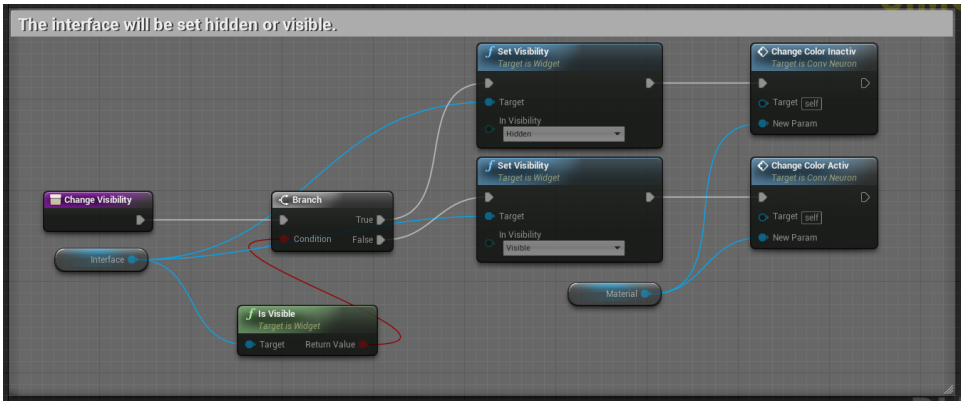


Abbildung 47: Das Menü der Neuronen (*Interface*) wird ein- oder ausgeblendet.

Convolutional Neuron Für die *Convolutional Layer* standen viele Informationen von TensorFlow zur Verfügung, welche visualisiert werden konnten. Jedes *Convolutional Neuron* (Klasse *Conv_Neuron*) besitzt einen Bias-Wert *_bias* (siehe Abbildung 49) und eine Liste *_weights* mit Gewichtswerten (siehe Kapitel 2, Abschnitt 2.1). Diese Werte werden angefordert mit den Methoden *requestBias* und *requestWeight*, sobald der Benutzer ein Neuron angeklickt hat. Die Anfrage erfolgt an die *TensorFlow Komponente*, die diese an das Python-Skript weiterleitet (siehe Kapitel 4).
Um die Informationen auslesen zu können, müssen die *ID* und der Name des zugehörigen Layers übergeben werden. Anschließend empfängt das Neuron mithilfe der Events *ReceiveBiasValues* und *ReceiveWeightValues* die entsprechenden Werte. Die Methoden *ReceiveBias* und *ReceiveWeight* werden durch die Events aufgerufen und die als JSON-String codierten Daten werden ausgelesen und in der Klasse gespeichert.

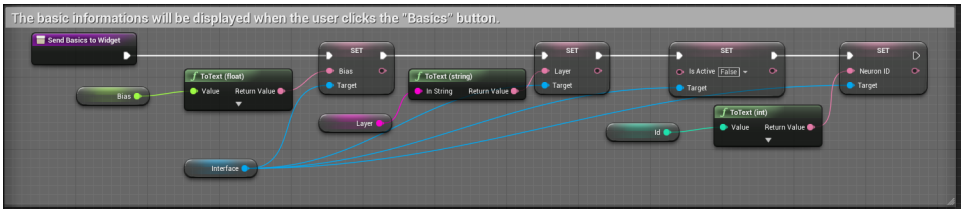


Abbildung 48: Alle gespeicherten Informationen des Neurons (*Bias, ID, Name des Layers*) werden an das Menü geschickt.

Nachdem die Daten empfangen und gespeichert wurden, werden automatisch die Methoden *sendBasicsToWidget* und *sendWeightToWidget* ausgeführt. Diese senden die Werte an das *Widget Blueprint* des Neurons, damit sie in den entsprechenden Menüs angezeigt werden (siehe Abschnitt 5.3.2). *sendBasictsToWidget* sendet neben dem Bias-Wert auch die ID und den Namen

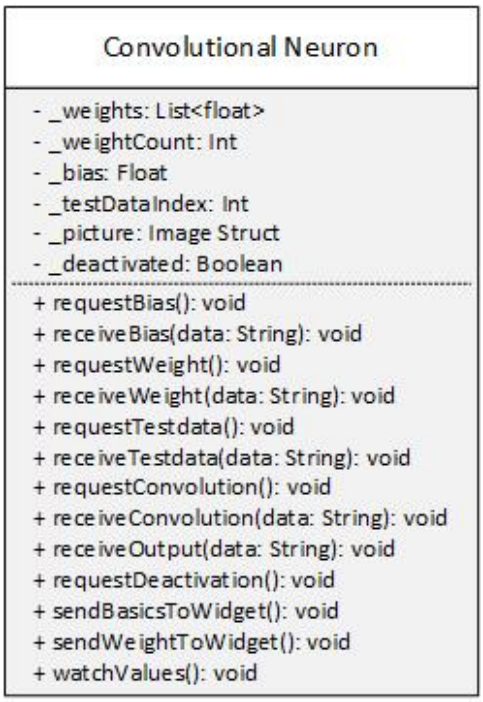


Abbildung 49: Die Klasse der *Convolutional Neuronen* speichert die meisten Informationen im neuronalen Netz.

des Layers an das Menü (siehe Abbildung 48). Jeder Eintrag aus der Liste *_weightList* muss mithilfe der Methode *sendWeightToWidget* in einen String umgewandelt und der korrekten Variable im *Widget-Blueprint* zugeordnet werden.

Auch die Bilder der Testdaten werden angefordert, sobald das Neuron angeklickt wurde. Die Methode *requestTestdata* übergibt der TensorFlow Komponente einen Index (*_testdataIndex*), da die Bilder in einer Liste gespeichert sind. Der Index gibt an, welches Bild ausgelesen und im Menü angezeigt werden soll. Mithilfe des *ReceiveTestdata*-Events wird das Bild empfangen und in der namensgleichen Methode zu einem *ImageStruct* formatiert (siehe Abbildung 50). Dieses kann zu einer 2D-Textur umgewandelt und anschließend im Menü angezeigt werden. Es wird zusätzlich in der Variable *_picture* gespeichert.

Die Testdaten-Bilder können auf Wunsch des Benutzers mit der *Filtermaske* des Neurons gefiltert werden. Damit soll die Aufgabe und Funktionsweise des Neurons nachvollziehbarer werden. Hierfür sendet die Methode *requestConvolution* die ID, den Namen des Layers und den Index *_testdataIndex* an die TensorFlow Komponente. Im Python Skript wird die Filterung

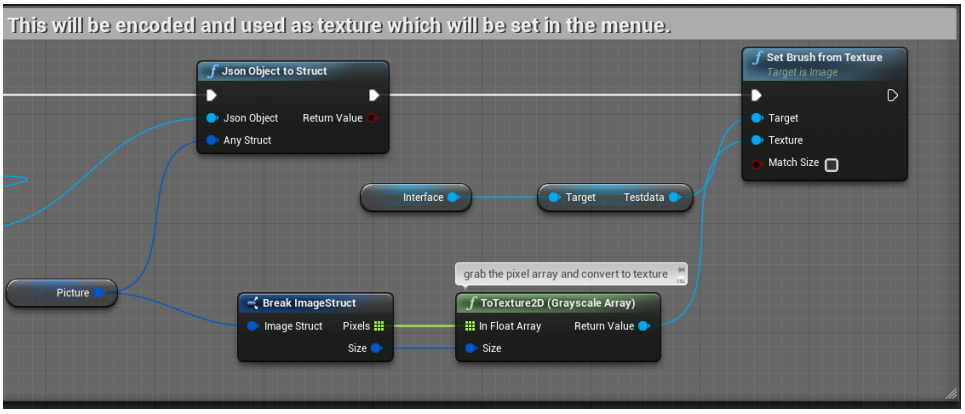


Abbildung 50: Die empfangenen JSON-Daten werden zu einer Textur umgewandelt und im Menü angezeigt.

vorgenommen (siehe Abschnitt 5.4) und das Ergebnis mithilfe des Events *ReceiveConvolution* von der Klasse des Neurons empfangen. Die Methode *receiveConvolution* wandelt den JSON-String in eine 2D-Textur um und schickt die Textur an das Menü.
Um nicht nur auf Grundlage dieser Filterung die Funktionsweise der Neuronen zu visualisieren, wurde die Methode *receiveOutput* implementiert. Diese sorgt dafür, dass das Ergebnis des Neurons, für die vom Benutzer übergebene Eingabe, visualisiert wird. Sobald das Modell eine Klassifizierung durchgeführt hat, erhält das Neuron mit dem *ReceiveOutput* Event das Ergebnis.

Abbildung 51: Ein deaktiviertes Neuron leuchtet rötlich, um es in der Szene zu erkennen.

Für eine bessere Nachvollziehbarkeit, wie wichtig ein Neuron für die Klassifizierung einer Eingabe ist, wurde die Methode *requestDeactivation* entwickelt. Wenn im Menü die Option ausgewählt wird ein Neuron zu deaktivieren (siehe Abschnitt 5.3.2), werden mithilfe der ID des Neurons und dem Namen des Layers der Bias-Wert und die Gewichtswerte im Python Skript auf Null gesetzt. Anschließend kann eine Eingabe erneut klassifiziert werden und der Benutzer kann die Veränderung der Genauigkeit der Klassifizierung betrachten. Damit der Benutzer in der Szene nachvollziehen kann, welche Neuronen deaktiviert wurden, leuchten die Neuronen rötlich (siehe Abbildung 51). Soll die Deaktivierung rückgängig gemacht werden, muss das neuronale Netz neu trainiert werden.

Während des Trainings steht dem Benutzer die Möglichkeit zur Verfügung, in festen Zeitintervallen die Werte des Neurons aktualisieren zu lassen. Hierfür muss im Menü die entsprechende Option gewählt werden (siehe Abschnitt 5.3.2), woraufhin die Methode *watchValues* aktiv wird. Diese führt die bereits beschriebenen Methoden in festen Zeitintervallen aus, wodurch der Bias-Wert, die Gewichtswerte und die Filterung eines festgelegten Testdatenbildes aktualisiert werden (siehe Abbildung 52).

Abbildung 52: In der Methode *watchValues* werden die Werte in festen Zeitintervallen aktualisiert.

Pool Neuron *Pool Layer* dienen dazu die Ergebnisbilder der Neuronen des vorhergegangenen Layers zu verkleinern. Aus diesem Grund besitzt die Klasse *Pool_Neuron* nur die Methode *receivePoolData*, welche das empfangene Bild als 2D-Textur im Menü anzeigt und als Variable *_picture* speichert, sobald eine Klassifizierung durchgeführt wurde (siehe Abbildung 53). Übergeben wird das Bild mit dem Event *ReceivePoolData*.

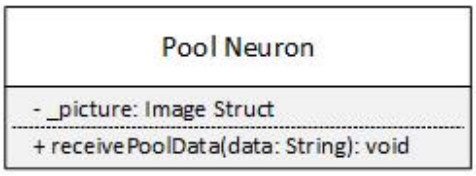


Abbildung 53: Die Klasse der *Pool Neuronen*.

Flatten Neuron Die *Flatten Layer* dienen dazu, die Ergebnisse eines vorhergegangenen Layers in einen eindimensionalen Vektor zu schreiben, mit dem darauffolgende Layer arbeiten. Dieser Vektor wird mit dem Event *ReceiveFlattenData* empfangen und durch die Klasse *Flatten_Neuron* dargestellt (siehe Abbildung 54). Jeder Wert aus dem JSON-Objekt wird in das Array *_flattenData* geschrieben und sobald das Menü des Neurons aufgerufen wird mit *showFlattenData* angezeigt.
Es werden immer fünf Werte aus dem Array gleichzeitig angezeigt und mit der Variable *_page* speichert die Klasse, welche Einträge zur Zeit angezeigt werden. Auch die Indexnummern der Einträge werden dem Benutzer angezeigt. Die Methoden *widgetEntryDecrease* und *widgetEntryIncrease* passen diese Nummern entsprechend an und speichern in der Variable *_variableEntry* den letzten Index, der dem Benutzer aktuell angezeigt wird.

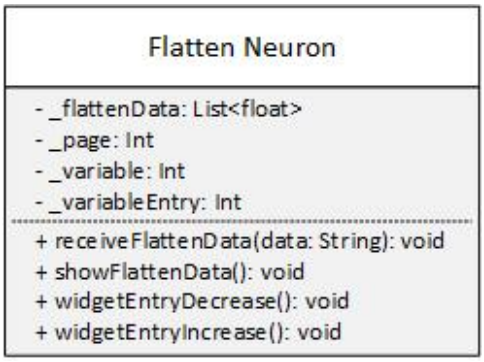


Abbildung 54: Die Klasse der *Flatten Neuronen*.

Dense Neuron Der eindimensionale Vektor des *Flatten Layers* wird von den *Dense Layern* genutzt, um einen Aktivierungswert zu errechnen. Dieser Aktivierungswert wird visualisiert, indem der Wert in eine *RGB-Farbe* übersetzt wird. So soll der Benutzer erkennen können, welche *Dense Neuronen* einen hohen oder niedrigen Aktivierungswert berechnet haben.
Wie bereits bei den *Dense Layern* beschrieben (siehe 5.2.2) wird die Methode *receiveFireRate* für jede *Dense_Neuron* Klasse aufgerufen, sobald der Layer die Daten vom Python Skript erhält. Mithilfe der, in der Klasse gespeicherten, ID liest jedes Neuron den eigenen Aktivierungswert aus und speichert

diesen in der Variable *_fireRate* (siehe Abbildung 55). Anschließend wird die Methode *setFireColor* aufgerufen. In dieser Methode wird der Wert als Grünwert in der Farbe des Materials gesetzt. Zusätzlich leuchtet das Material, wodurch sich die Neuronen besser von der Szene abheben sollen. Je heller der Grünton des Neurons, desto höher ist sein Aktivierungswert.

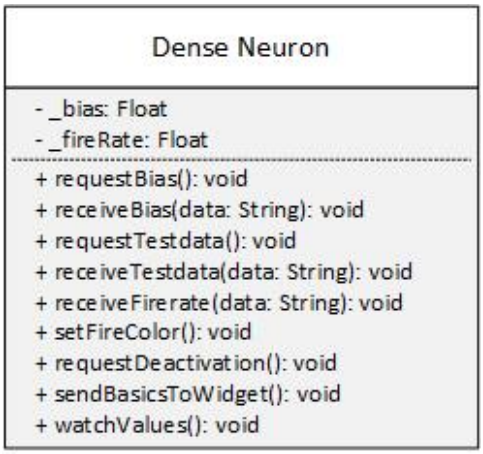


Abbildung 55: Die Klasse der *Dense Neuronen*.

5.2.4 TensorFlow Klasse

Die TensorFlow Komponente *TensorflowComponent* regelt den Austausch der Daten zwischen der Unreal Engine und dem Python Skript (siehe Kapitel 4, Abbildung 32). Bereits vorhandene Methoden (siehe Kapitel 3.2.2) wurden erweitert, sodass die gewünschten Informationen zur Visualisierung von den Neuronen angefordert werden konnten. Vor dem Training des Modells kann der Benutzer im Hauptmenü (siehe Abschnitt 5.3.1) Einstellungen festlegen, welche mit der Methode *sendJsonSetup* an das Python Skript übertragen werden.

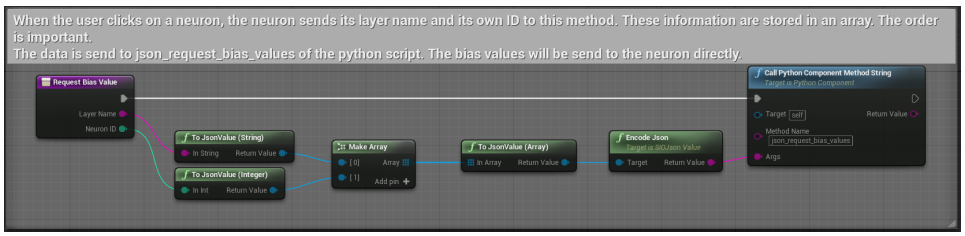


Abbildung 56: Die *TensorFlowComponent*-Klasse schreibt Daten in ein Array und sendet dieses an eine Python Methode des Python-Skripts.

Für die Neuronen stehen Funktionen zur Verfügung, deren Namen denen der Neuronen Methoden (siehe Abschnitt 5.2.3) gleichen. Die von den Neuronen übermittelten Daten werden von der *TensorflowComponent*-Klasse in ein Array übertragen, welches anschließend zu einem JSON-Objekt umgewandelt wird. Dieses Objekt wird an das Python-Skript geschickt, wofür der Name der Python-Funktion manuell angegeben werden muss (siehe Abbildung 56). Zu beachten ist, dass die Reihenfolge im Array der Reihenfolge entsprechen muss, die im Python-Skript erwartet wird, da es sonst zu Fehlern beim Auslesen der Daten kommt.
Diese Vorgehensweise wurde für alle Methoden (*requestBiasValue, requestWeightValue, requestTestData, requestConvolution, requestDeactivation*) identisch umgesetzt.

5.3 Menüs

Jede Klasse von Neuronen hat ein individuelles Menü erhalten, um die von *TensorFlow* bereitgestellten Informationen dem Benutzer übersichtlich darzustellen. Hierbei wurde Wert darauf gelegt dem Benutzer die Entscheidung zu überlassen, welche Informationen eingeblendet werden sollen. Aus dem Grund wurden die Menüs in Kategorien eingeteilt. Zur Erstellung der Menüs wurden *Widget Blueprints* der Unreal Engine verwendet.

5.3.1 Hauptmenü

Die Klasse *Setup* dient als Hauptmenü, welches eingeblendet wird sobald die Szene startet (siehe Abbildung 57). Es dient dazu Einstellungen für das Training des neuronalen Netzes vorzunehmen, wie zum Beispiel die Größe der Testdatenmenge pro Durchlauf (*Batch-Size*) oder die Anzahl an Durchläufen (*Epochs*). Wenn die Schaltfläche *Start Training* gedrückt wird, werden die Daten über die *TensorFlow Komponente* an das Python Skript übertragen, wo das Training mit den gewählten Einstellungen gestartet wird. Daraufhin wird das Menü automatisch ausgeblendet. Mit der *M-Taste* hat der Benutzer die Möglichkeit das Menü jederzeit erneut einzublenden. Dies soll dazu dienen die Werte jederzeit kontrollieren und das Training erneut starten zu können, wenn beispielsweise Änderungen am Modell vorgenommen wurden.

Abbildung 57: Das Hauptmenü erlaubt Trainingsparameter vor dem Training einzustellen.

5.3.2 Menü der Neuronen

Im folgenden Abschnitt werden die Menüs der einzelnen Klassen von Neuronen vorgestellt und ihre Funktionsweise erläutert.

Convolutional_NeuronInterface Jede *Conv_Neuron* Klasse erhält dieses *Widget-Blueprint*, um die Informationen für *Convolutional Layer* darzustellen. Klickt der Benutzer auf ein Neuron mit der linken Maustaste, wird das Neuron aktiviert und ein Informationssymbol eingeblendet (siehe Abbildung 58). Dies soll dem Benutzer symbolisieren, dass nachfolgende Menüs Informationen enthalten. Wird das Symbol angeklickt, wird eine Menüleiste mithilfe der Methode *showMenues* eingeblendet (siehe Abbildung 59), von der aus das gewünschte Untermenü aufgerufen werden kann.

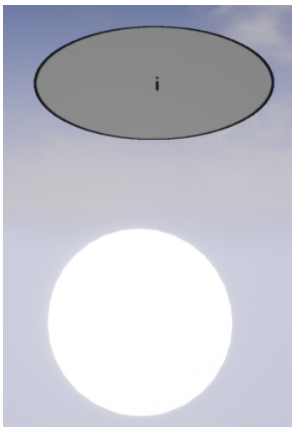


Abbildung 58: Das Informationssymbol dient als Orientierung und Hinweis, dass die Menüs geöffnet werden können.

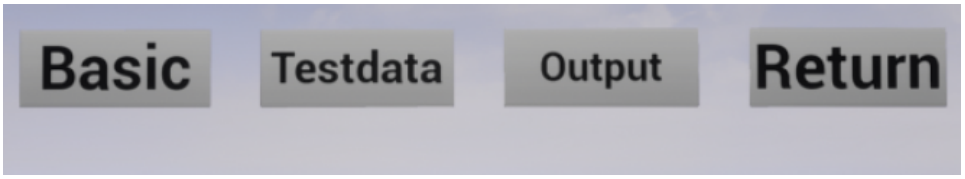


Abbildung 59: Von dieser Menüleiste aus lassen sich die jeweiligen Untermenüs öffnen.

Das Menü *Basic* enthält grundlegende Informationen über ein Neuron, wie seine ID, den Namen des Layers und den Bias-Wert (siehe Abbildung 60). Mit der Checkbox *Is Active* lässt sich das Neuron auf Wunsch deaktivieren und wenn die Option *Watch Values* aktiviert wird, werden der Bias-Wert und die Gewichtswerte in festen Zeitintervallen aktualisiert (siehe Abschnitt 5.2.3). Die Schaltfläche *Show Weights* ermöglicht es dem Benutzer sich die Werte der Filtermaske anzusehen (siehe Abbildung 61).
In dem gezeigten Beispiel besteht die Filtermaske des ersten *Convolutional Layer* aus 25 Zahlenwerten. Die *Unreal Engine* erlaubt es nicht, dass diese Zahlenfelder dynamisch angelegt werden, weshalb alle *Convolutional Layer* 25 Zahlenwerte anzeigen, auch wenn die Filtermaske kleiner sein sollte.

Abbildung 60: Von dieser Menüleiste aus lassen sich die jeweiligen Untermenüs öffnen.

Im Menü *Testdata* wird dem Benutzer ein Bild der Testdaten auf der linken Seite angezeigt (siehe Abbildung 61). Dieses kann mit der *Next*-Schaltfläche geändert werden. In der Mitte dieses Menüs wird erneut die Filtermaske angezeigt. Auf der rechten Seite hat der Benutzer über die Schaltfläche *Convolve* die Möglichkeit das ausgewählte Testdatenbild vom Neuron filtern zu lassen. Das Ergebnisbild wird daraufhin eingeblendet und soll dazu dienen die Funktionsweise des Neurons besser nachvollziehen zu können.

Um das Ergebnisbild des Neurons für eine Klassifizierung anzeigen zu können, wurde das Menü *Output* hinzugefügt. Hier wird das Ergebnisbild angezeigt, sobald eine Eingabe klassifiziert wurde.

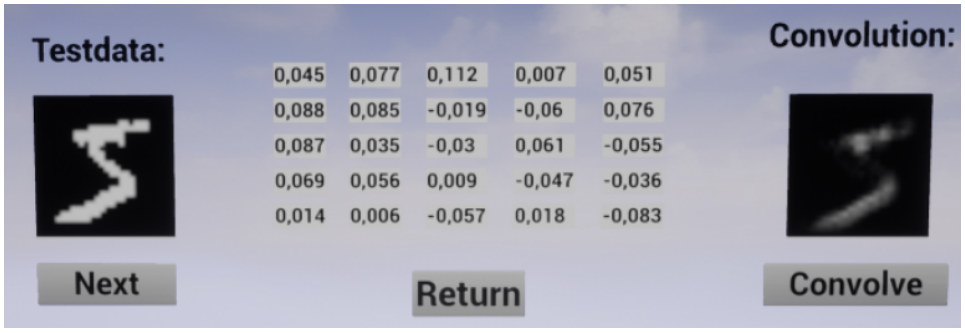


Abbildung 61: Die Testdaten (links), die Gewichtsmaske (mitte) und das Ergebnis der Filterung (rechts) werden in diesem Untermenü angezeigt

Pool_NeuronInterface Für die Klasse *Pool_Neuron* zeigt das Menü *Pool_NeuronInterface*, nach dem Klick auf das Informationssymbol, direkt das Ergebnis des Neurons an. Da *Pool Layer* nur das Ergebnisbild des vorhergegangenen Layers kleiner skalieren (siehe Abschnitt 5.2.2), stehen keine weiteren Informationen zur Verfügung. Das Bild wird für die Visualisierung auf die gleiche Größe gestreckt, wie sie im Menü *Convolutional_NeuronInterface* definiert wird, ansonsten würden die Texturen zu klein werden. Die Streckung verändert das Bild nicht, so dass die Funktionsweise des Layers immer noch nachvollziehbar sein sollte.

Flatten_NeuronInterface *Flatten Layer* stellen nur einen Vektor mit *float-Werten* bereit (siehe Abschnitt 5.2.2). Dieser Vektor wird im *Flatten_NeronInterface* Menü visualisiert, um dem Benutzer die Funktionsweise dieser Layer zu veranschaulichen. Dafür werden fünf Einträge aus dem Vektor angezeigt und die Reihenfolge der Werte durch die Darstellung des Vektor-Index repräsentiert (siehe Abbildung 62). Über die Schaltflächen *Next Page* und *Previous Page* kann der Benutzer sich die nächsten oder vorherigen Werte anschauen (siehe Abschnitt 5.2.3).

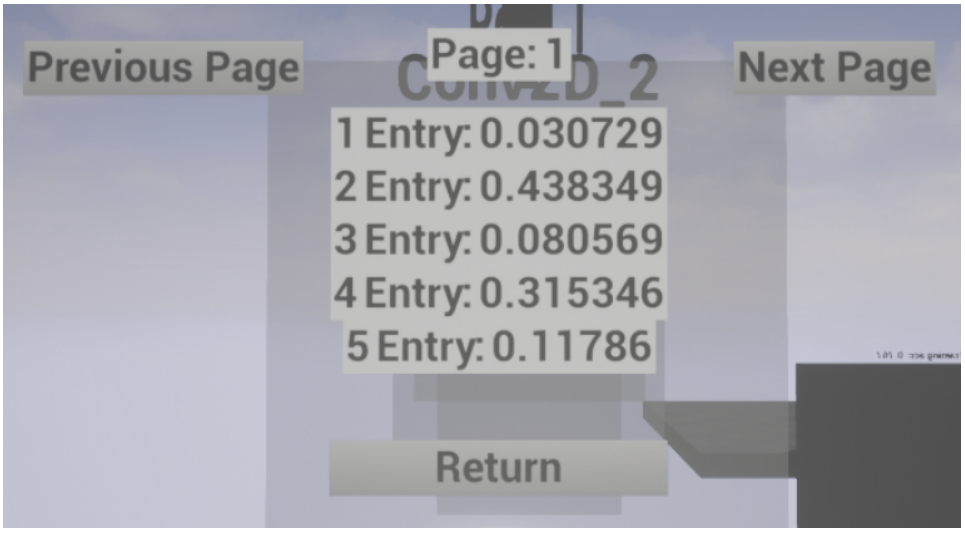


Abbildung 62: Der Vektor des *Flatten Layers* wird visualisiert.

5.3.3 Camera Interface

Die Klasse *CameraInterface* wurde implementiert um die Informationen, die bereits in der Demo des *TensorFlow Plug-in* visualisiert wurden (siehe Abschnitt 3.2.2), in einem separaten Menü zugänglich zu machen. Unter anderem wird die Dauer des Trainings, das Ergebnis und die Genauigkeit der Klassifizierung angezeigt. Mit der *N-Taste* kann der Benutzer ein Fenster öffnen, das am unteren, linken Bildrand eingeblendet wird und diese Informationen anzeigt (siehe Abbildung 63). Das Bild wird mithilfe einer Kamera aufgenommen, die mittig vor dem Objekt platziert wurde.

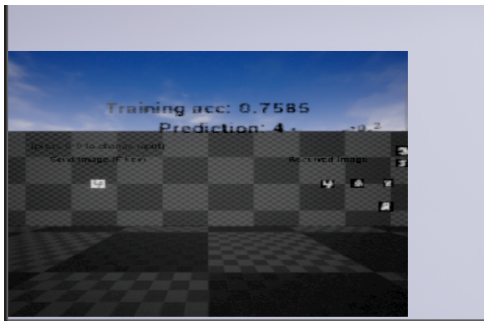


Abbildung 63: Ein zusätzliches Fenster kann eingeblendet werden, um Informationen über das Netzwerk anzuzeigen.

5.4 Python Skript

Vom Benutzer muss ein *Python Skript* bereitgestellt werden, welches den Anforderungen entspricht, die in Kapitel 3.2.1 erläutert wurden. Das Skript, das durch die *MNIST Demo* gegeben war (siehe Abschnitt 3.2.2), wurde erweitert, so dass die Informationen des Modells zur Visualisierung genutzt werden konnten. In diesem Abschnitt werden die programmierten Methoden und Veränderungen an bereits vorhandenen Funktionen vorgestellt.

createNetwork Diese Methode wird genutzt um das Modell von der *Unreal Engine* darstellen zu lassen. Hierfür wird der Name jedes Layers, der dargestellt werden soll, als *String* in der Variable *information_log* gespeichert. Falls mehrere Layer der gleichen Art hintereinander erstellt werden sollen, können auch mehrere *Strings* gespeichert werden.
Mit der *callEvent*-Methode (siehe Kapitel 3.2.2) wird ein Event verschickt, welches für die Art des Layers zuständig ist. Für einen *Convolutional Layer* wird beispielsweise das Event *CreateConvLayer* genutzt. Zusätzlich zum Event wird die Variable *information_log* als JSON-Objekt übergeben, wofür der Boolean-Wert *True* angegeben werden muss.
Anschließend können die *Neuronen* für einen Layer erstellt werden. In der Variable *information_log* muss dafür der Name des Layers eingetragen werden und es wird eine Liste erwartet (siehe Abbildung 64). Der erste Eintrag der Liste bestimmt, wie viele Neuronen der Layer erhalten soll. Die anderen beiden Werte geben an, in wie vielen *Reihen* und *Spalten* die Neuronen angeordnet werden sollen. Auch hier ist es möglich für mehrere Layer der gleichen Art mit einem *information_log* die Neuronen erstellen zu lassen. Die *callEvent*-Methode wird erneut genutzt, um das entsprechenden Event und die Informationen an die *Unreal Engine* zu übergeben. Für *Convolutional Layer* wird zum Beispiel das Event *CreateConvNeurons* verwendet.

```
information_log = ['Conv2D',[30, 5, 6]]
self.stopcallback.outer.callEvent('CreateConvNeurons',
    information_log, True)
```

Abbildung 64: Für die Erstellung der Neuronen des *Convolutional Layers* müssen vier Informationen übergeben werden.

onJsonRequestBiasValues Als Eingabeparameter erhält diese Methode den *Namen* eines Layers und die *ID* eines Neurons. Es wird erwartet, dass diese Informationen in einem JSON-Objekt (*jsonInput*) übergeben werden. Hierbei ist zu beachten, dass der *Name* an erster und die *ID* an zweiter Stelle im JSON-Objekt liegen. Bevor diese Werte genutzt werden, wird überprüft,

ob das Modell bereits erstellt wurde. Hierfür wird die Klassen-Variable *model_ready* genutzt, welche einen *Boolean*-Wert speichert. Sobald ein Modell erstellt wurde, wird dieser Wert auf *True* gesetzt.
onJsonRequestBiasValues liest mit der Methode *get_weights()* die Bias-Werte des Layers aus, dessen Namen der Benutzer übergeben hat. *get_weights()* gibt zwei Listen zurück: Eine für die Gewichte (*Weights*) und eine für die Bias-Werte. Aus der Liste der Bias-Werte wird mithilfe des übergebenen Index der Wert für das entsprechende Neuron ausgelesen.
Dieser muss in das 64-bit Format übertragen werden, um als JSON-Objekt übergeben werden zu können. Für die Typumwandlung wird der Befehl *np.float64* des *NumPy*-Paketes genutzt[60]. Anschließend wird der Bias-Wert mit einem *callEvent*-Befehl an die *Unreal Engine* übertragen. (siehe Abbildung 65).

```
if(self.model_ready):
 biases = np.float64(self.model.get_layer(name=jsonInput[0]).
     get_weights()[1][jsonInput[1]])
 self.stopcallback.outer.callEvent('ReceiveBiasValues', biases,
     True)
```

Abbildung 65: Die Bias-Werte stehen in einer Liste, die mit *get_weights()* ausgelesen wird.

onJsonRequestWeightValues Wie auch die Methode *onJsonRequestBiasValues* erhält *onJsonRequestWeightValues* den Namen des Layers und die ID des Neurons als Eingabeparameter. Nach der Überprüfung, ob das Modell erstellt wurde wird erneut die Methode *get_weights()* genutzt, um die Gewichtswerte zu erhalten (siehe Abbildung 67).
Die Gewichtswerte aller Neuronen liegen in einer verschachtelten Liste vor. Jedes Neuron besitzt mehrere Werte, da diese Werte eine Filtermaske repräsentieren. Die Größe der Filtermaske wird vor dem Training des Modells festgelegt. Der erste Maskenwert aller Neuronen wird in der ersten Liste eines *NumPy-Arrays* hinterlegt und somit besteht das Array aus *n* Listen, wobei *n* der Anzahl an Neuronen entspricht.
Um die Werte auszulesen wird eine Liste *weight_list* erstellt und mithilfe von zwei *for-Schleifen* werden die Gewichtswerte für ein Neuron ausgelesen und in *weight_list* übertragen.
In dem genutzten Beispiel musste die Anzahl der Werte in der *weight_list* auf 25 festgelegt werden, weil der erste Layer des Netzwerks eine Maske dieser Größe nutzt und keine dynamische Anpassung in der *Unreal Engine* möglich war (siehe auch 5.3.2).

[60] NumPy Dokumentation

Nachdem alle Werte übertragen wurden, wird die *weight_list* Variable mithilfe des Events *ReceiveWeightValues* an die Unreal Engine geschickt.

```
a = self.model.get_layer(name=jsonInput[0]).get_weights()
weight_list = [[0 for x in range(5)] for y in range(5)]
for i in range(len(a[0])):
 for j in range(len(a[0][0])):
  weight_list[i][j] = np.float64(a[0][i][j][0][jsonInput[1]])
self.stopcallback.outer.callEvent('ReceiveWeightValues',
    weight_list, True)
```

Abbildung 66: Zum Auslesen der Gewichtswerte wird eine doppelte *for-Schleife* genutzt.

onJsonRequestTestdata Als Eingabeparameter für diese Methode dient ein *Index*, welcher genutzt wird, um ein Bild der Testdaten aus dem *Testdatensatz* auszulesen. In der Variable *x_train* liegen die Bilder, mit denen das neuronale Netz trainiert wurde und mithilfe des Index kann das gewünschte Bild ausgelesen werden. Um es im *JSON-Format* an die Unreal Engine zu übermitteln, muss das Bild mit dem Befehl *ravel().tolist()* zu einer Liste umgewandelt werden. Diese Liste wird in der Klassen-Variable *jsonPixels* (siehe Kapitel 3.2.2) unter dem Eintrag *pixels* gespeichert, welche dann mit dem Event *ReceiveTestdata* an die Unreal Engine geschickt wird (siehe Abbildung 67).

```
def onJsonRequestTestdata(self, jsonInput):
 print("-----DEBUGGER: onJsonRequestTestdata-----")
 self.stopcallback.outer.jsonPixels['pixels'] = self.stopcallback
     .outer.x_train[jsonInput[0]].ravel().tolist()
 self.stopcallback.outer.callEvent('ReceiveTestdata', self.
     stopcallback.outer.jsonPixels, True)
```

Abbildung 67: In der Klassen-Variable *jsonPixels* werden die Bilddaten gespeichert und an die Unreal Engine geschickt.

onJsonRequestConvolve Wenn der Benutzer ein Testdatenbild filtern lassen möchte (siehe Abschnitt 5.3.2) wird dieser Methode der *Index* des Testdatenbildes, die *ID* des Neurons und der Name des zugehörigen Layers übergeben. Zuerst wird die Variable *output* formatiert, um die richtige Größe zu haben und das gefilterte Bild speichern zu können. Anschließend wird das Testdatenbild ausgelesen und ebenfalls formatiert, so dass der erste Layer dieses als Eingabe erhalten kann (siehe Abbildung 68).

```
output_width = layer_output_shape[1]
output_height = layer_output_shape[2]
output = np.zeros((output_height, output_width))

x_raw = np.reshape(self.x_train[jsonInput[2]], (1, 28, 28))
x = np.reshape(x_raw, (len(x_raw), 28, 28, 1))
```

Abbildung 68: Die Variable *output* wird angelegt und die Eingabe für den ersten Layer vorbereitet.

In einer *for-Schleife* werden alle Layer des Modells nacheinander aufgerufen, bis der Layer erreicht wird, dessen Name übergeben wurde. Die Ergebnisse der Layer werden in der Liste *layer_output_list* gespeichert und das Ergebnis des vorangegangenen Layers dient dem nächsten Layer als Eingabe. Nachdem die Schleife beendet wurde wird der *run*-Befehl von *TensorFlow* aufgerufen, um die Werte der *Tensoren* auslesen zu können[61] (siehe Abbildung 69).

```
K.set_session(self.session)
with self.session.as_default():
 layer_output_list = []
 for i in range(1,layer_index):
  if(i==1):
   layer_output_list.append(self.model.get_layer(index=i).call(x
       ))
  else:
   layer_output_list.append(self.model.get_layer(index=i).call(
       layer_output_list[i-2]))
 #print("Layer Output: ", layer_output_list[2])

layer_output_list =  self.session.run(layer_output_list)
```

Abbildung 69: Die Variable *output* wird angelegt und die Eingabe für den ersten Layer vorbereitet.

Das Ergebnis für das Neuron wird aus dem Ergebnis des entsprechenden Layers ausgelesen und in der Variable *output* gespeichert. Damit die Klassen-Variable *jsonPixels* nicht überschrieben wird, wir eine neue Variable *jsonPixels_2* verwendet. Dieser Variable wird die Größe des Ergebnisbildes im Eintrag *size* zugewiesen und unter dem Eintrag *pixels* wird das Bild, wie bereits in der Methode *requestTestdata*, hinterlegt. Das Event *ReceiveConvolution* und die Daten werden übertragen.

[61] TensorFlow Dokumentation - Session

onJsonRequestDeactivate Wird ein Neuron deaktiviert (siehe Abschnitt 5.2.3), wird mithilfe dieser Methode der *Bias-Wert* und die *Gewichtswerte* des Neurons auf Null gesetzt.
Für den Bias-Wert genügt es, mit der *tf.assign*-Funktion[62], dem entsprechenden Wert in der Liste des Layers den Wert *0.0* zuzuweisen. Der *run*-Befehl muss ausgeführt werden, um die Änderung im *TensorFlow Graph* zu übernehmen (siehe Abbildung 70).

```
def onJsonRequestDeactivate(self, jsonInput):
 #Set the bias-Value to Zero
 self.session.run(tf.assign(self.model.get_layer(name=jsonInput
     [0]).weights[1][jsonInput[1]], 0.0))
```

Abbildung 70: Dem Bias-Wert wird der Wert *0.0* zugewiesen.

Für die Gewichtswerte wird zunächst die aktuelle Liste kopiert und in der Variablen *a* gespeichert (siehe Abbildung 71). In dieser Liste werden alle Werte des Neurons auf den Wert *0.0* gesetzt. Da in späteren Layern ein Neuron mehrere Eingabebilder erhalten kann, können auch mehrere Listen mit Gewichten existieren. Diese Listen werden durch eine Schleife über die Variable *channel* berücksichtigt. Zum Schluss wird mit der Methode *tf.assign* die komplette Liste der Gewichtswerte des Layers mit der Liste *a* ausgetauscht und erneut mit der *run*-Methode übernommen.

```
 a = self.model.get_layer(name=jsonInput[0]).get_weights()
 channel_num = self.model.get_layer(name=jsonInput[0]).
     input_shape[3]
 for channel in range(channel_num):
  for i in range(len(a[0])):
   for j in range(len(a[0][0])):
    a[0][i][j][channel][jsonInput[1]] = 0.0

 self.session.run(tf.assign(self.model.get_layer(name=jsonInput
     [0]).weights[0], a[0]))
```

Abbildung 71: Dem Bias-Wert wird der Wert *0.0* zugewiesen.

onJsonStartupInput Bevor das Modell trainiert wird, werden der Methode *onJsonStartupInput* die Einstellungen, die der Benutzer im Hauptmenü vorgenommen hat (siehe Abschnitt 5.3.1) übergeben. Die *Batch-Size* und *Epochs* werden als Klassen-Variablen (*batch_size*, *epochs*) gespeichert (siehe

[62] TensorFlow Dokumentation - tf.assign

Abbildung 72). Ob die Testdatenwerte gerundet werden sollen wird in der Variablen *round_values* gespeichert.

```
def onJsonStartupInput(self, jsonInput):
  print("———DEBUGGER: MnistKeras:onJsonStartupInput———")
  self.batch_size = jsonInput[0]
  self.epochs = jsonInput[1]
  self.round_values = jsonInput[2]
```

Abbildung 72: Die Einstellungen aus dem Hauptmenü werden übernommen.

createLayerResults Nachdem das neuronale Netz eine Eingabe klassifiziert hat, wird automatisch die Methode *onJsonLayerResults* aufgerufen. Als Eingabe dient das Bild, dass klassifiziert wurde. Zuerst wird das Bild dem ersten Layer des Modells übergeben und mit der *call*-Methode[63] wird der Layer ausgeführt (siehe Abbildung 73).

```
  layer_output = np.float32(inputImage)
  layer_output = self.model.get_layer(index=1).call(layer_output)

  layer_list = [layer_output]
```

Abbildung 73: Der erste Layer wird ausgeführt mithilfe der *call*-Methode von TensorFlow.

Das Ergebnis wird als erster Eintrag in der *layer_output* Liste gespeichert. Nacheinander werden alle Layer des Modells aufgerufen und ihre Ergebnisse der Liste angehängt. Wie auch schon in der Methode *onJsonRequest-Convolve* wird das Ergebnis eines vorhergegangenen Layers an den Nächsten weitergegeben. In dem gegebenen Beispiel wurde ein *Dropout-Layer* verwendet, welcher einen bestimmten Prozentsatz der Testdaten entfernt. Dieser Layer wird übersprungen, da dieser nur für das Training gebraucht wird (siehe Abbildung 74). Dafür muss bekannt sein, welchen Index der Layer hat, da es ansonsten zu einem Laufzeitfehler kommt.
Der *run*-Befehl wird am Ende ausgeführt, um die *Tensoren* in der Liste auszuwerten. Danach kann auf die Informationen der Layer zugegriffen werden. Dem Benutzer ist es überlassen, welche Ergebnisse dargestellt werden sollen. Für jede Art von Layer müssen die Informationen auf andere Art und Weise ausgelesen werden.

[63] TensorFlow Dokumentation - Layer

```
for i in range(2,10):
 if(i < 5):
  layer_list.append(self.model.get_layer(index=i).call(
      layer_list[i-2]))
 elif(i > 5):
  layer_list.append(self.model.get_layer(index=i).call(
      layer_list[i-3]))

layer_list = self.session.run(layer_list)
```

Abbildung 74: Die Layer werden nacheinander ausgeführt und ihre Ergebnisse gespeichert.

Für die *Convolutional Layer* stehen die Ausgabebilder zur Verfügung. Diese können, wie bereits bei der *onJsonRequestConvolve*-Funktion beschrieben, ausgelesen und an die *Unreal Engine* übertragen werden. Die Besonderheit bei der *onJsonLayerResults*-Methode ist, dass der Layer zuerst aktiviert werden muss mit dem Event *ActivateConv* (siehe Abbildung 75).

```
self.stopcallback.outer.callEvent('ActivateConv',
    information_log, True)
for neuron in range(layer_output_shape[3]):
 for i in range(output_width):
  for j in range(output_height):
   output_test[i][j] = test[0][i][j][neuron]
 jsonPixels_2['pixels'] = output_test.ravel().tolist()
 self.stopcallback.outer.callEvent('ReceiveConvOutput',
     jsonPixels_2, True)
 output_test = np.zeros((output_height, output_width))
 jsonPixels_2 = {}
 jsonPixels_2['size'] = size

information_log = ['Conv2D']
self.stopcallback.outer.callEvent('DeactivateConv',
    information_log, True)
```

Abbildung 75: Jedes Neuron erhält seine Ergebnisse separat. Es wird erneut *jsonPixels_2* als Variable genutzt.

Dies stellt sicher, dass der richtige Layer die Ergebnisse erhält. Außerdem erhält jedes Neuron einzeln das Ergebnis des Layers mit dem Event *ReceiveConvOutput*. Nach jeder Übertragung der Daten an ein Neuron, werden die verwendeten Variablen, wie zum Beispiel *jsonPixels_2*, zurückgesetzt, damit das nächste Neuron Daten erhalten kann. Am Ende muss der Layer wieder deaktiviert werden, wofür das Event *DeactivateConv* genutzt wird.

Pool Layer erhalten auf dieselbe Weise ihre Ergebnisse, nur die Events wur-

den entsprechend angepasst (siehe Abbildung 76).

```
self.stopcallback.outer.callEvent('ActivatePool',
    information_log, True)
for neuron in range(layer_output_shape[3]):
 for i in range(output_width):
  for j in range(output_height):
   output_test[i][j] = test[0][i][j][neuron]
 jsonPixels_2['pixels'] = output_test.ravel().tolist()
 self.stopcallback.outer.callEvent('ReceivePoolData',
     jsonPixels_2, True)
 output_test = np.zeros((output_height, output_width))
 jsonPixels_2 = {}
 jsonPixels_2['size'] = size

information_log = ['Pool_1']
self.stopcallback.outer.callEvent('DeactivatePool',
    information_log, True)
```

Abbildung 76: Für die *Pool Layer* werden andere Events genutzt, doch der Ablauf zur Übertragung der Informationen gleicht dem Ablauf für *Convolutional Layer*

Für *Flatten Layer* wird diese Routine nicht gebraucht, da der Vektor des Layers direkt als Liste übertragen werden kann (siehe Abbildung 77).

```
test = layer_list[4]
information_log = ['Flatten', test.tolist()]
self.stopcallback.outer.callEvent('ReceiveFlattenData',
    information_log, True)
```

Abbildung 77: Der Vektor der *Flatten Layer* kann als Liste direkt übermittelt werden.

Bei den *Dense Layern* werden die Aktivierungswerte der Neuronen als Liste ausgelesen und an die *Unreal Engine* geschickt. Vorher werden die Werte normalisiert, so dass sie im Wertebereich *[0, 1]* liegen. In der *Dense Layer* Klasse werden diese Werte für *RGB-Farbwerte* genutzt und müssen aus diesem Grund in diesem Wertebereich liegen (siehe Abschnitt 5.2.3). Mit dem Event *ReceiveFireRate* werden die angepassten Werte an die Unreal Engine übertragen (siehe Abbildung 78).

```
test = layer_list[5]
max_test = max(test[0])
test[0] /= max_test
information_log = ['Dense_1', test.tolist()]
self.stopcallback.outer.callEvent('ReceiveFirerate',
    information_log, True)
```

Abbildung 78: Die Werte für die *Dense Layer* müssen vorher normalisiert werden.

5.5 Visualisierung großer Netzwerke

In Kapitel 4 wurde bereits eine Konzeptidee vorgestellt, wie sehr große neuronale Netze visualisiert werden könnten. Mehrere tausend Neuronen oder Layer, die gleichzeitig in einer Szene angezeigt werden, würden den Grafikspeicher aktueller Grafikkarten zu stark beanspruchen und eine Echtzeitanwendung wäre nur schwer umsetzbar.
Während der Umsetzung des Konzepts zur Visualisierung von neuronalen Netzen wurde darauf geachtet, dass ein Benutzer die Möglichkeit hat nur Teile des Modells von der *Unreal Engine* visualisieren zu lassen. Die, im letzten Abschnitt 5.4, vorgestellten Methoden *createNetwork* und *createLayerResult* ermöglichen dies und auf diese Weise werden auch große Modelle unterstützt.

Als zweiten Ansatz zur Visualisierung großer Modelle, wurde eine Form des *Level of Detail* Verfahrens umgesetzt. Dieser Ansatz wurde zu Demonstrationszwecken in einer separaten Szene eingebaut, weil die Visualisierung der Informationen des Modells bei dieser Technik nicht von Bedeutung war.
Das Modell der *MNIST Demo* wurde manuell in der Szene nachgebaut. Hierbei wurde ein großer Würfel genutzt, um das gesamte neuronale Netz zu repräsentieren (siehe Abbildung 79). In diesem wurden kleinere Würfel platziert, die jeweils mehrere Layer des Netzes repräsentieren sollten. Schriftzüge über den Würfeln geben dabei an, welche Art von Layer innerhalb der Würfel liegen.
Innerhalb der Layer wurden erneut die Neuronen platziert. Für die *Dense Layer* wurden die Neuronen in Kugeln zusammengefasst, um die Anzahl an gleichzeitig dargestellten Neuronen zu minimieren.
Jeder Objekt-Klasse wurde die Methode *distanceToUser* hinzugefügt. Mithilfe eines *Tick*-Events[64] wird diese Methode in jedem *Frame* ausgeführt. Diese Methode berechnet die Distanz zwischen dem Benutzer und der Geometrie der Objekt-Klasse in der Szene. Wird ein vorher festgelegter Distanzwert unterschritten, wird der entsprechende Layer oder das Neuron eingeblendet. Sobald die Distanz wieder größer ist, wird das Objekt ausge-

[64] Unreal Engine Dokumentation - Event Tick

blendet.
Auf diese Weise kann Grafikspeicher gespart werden, indem nur Teile des Modells gerendert werden. Der Nachteil ist, dass die Berechnung der Distanz bei vielen Objekten Rechenleistung kostet.

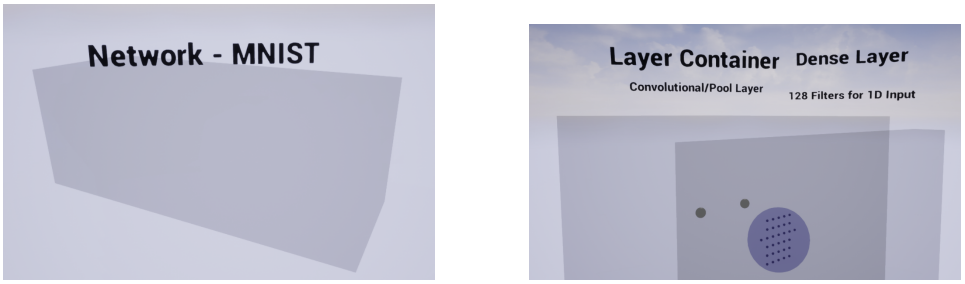


Abbildung 79: Das Modell wird zuerst als Würfel dargestellt (links). Layer und Neuronen sind ausgeblendet. Wenn der Benutzer näher an das Modell herangeht werden Layer und Neuronen sichtbar (rechts).

Um die Berechnungen der Distanz zu vermeiden und eine effizientere Methode für diese Form des *Level of Detail* Verfahrens zu demonstrieren, wurde eine weitere Szene mit einem einzelnen Layer erstellt. In *Layer_Random* wurden die Neuronen zufällig innerhalb des Layers platziert. Zusätzlich erhielt jedes Objekt eine *Bounding Box*, welche Kollisionen in einer Szene automatisch erkennt. Sobald eine Kollision erkannt wurde zwischen dem Benutzer und einem Neuron, wird das entsprechende Neuron eingeblendet.
Damit lassen sich Neuronen eines Layers dynamisch, innerhalb eines festgelegten Radius um den Benutzer, ein- und ausblenden. Dies kann Rechenleistung und Grafikspeicher sparen.
Den genauen Gewinn an Rechenleistung müsste mit einer Evaluation geprüft werden. Im Rahmen dieser Abschlussarbeit war dies aus zeitlichen Gründen nicht möglich und es wurde kein Mehrwert für die Fragestellung dieser Ausarbeitung gesehen.

6 Evaluation

Im folgenden Kapitel wird die Evaluation, die im Rahmen dieser Abschlussarbeit durchgeführt wurde, vorgestellt. Abschnitt 6.1 erläutert das Ziel der Evaluation und nennt die Hypothese. Der Ablauf und der genutzte Fragebogen werden im Abschnitt 6.2 thematisiert. Anschließend werden die Ergebnisse zusammenfassend präsentiert (Abschnitt 6.3). Das Fazit (Abschnitt 6.4) dient dazu, die Evaluation abzuschließen.

6.1 Ziel der Evaluation

Das Thema dieser Abschlussarbeit lautet: *Nachvollziehbarkeit und Begründbarkeit von Maschinellen Lernverfahren mithilfe von 3D-Visualisierung*. Mit diesem Thema einhergehend wurde, nach der Auseinandersetzung mit aktuellen Arbeiten zu diesem Thema (siehe Kapitel 1), eine Hypothese aufgestellt: Die Visualisierung eines neuronalen Netzes in einer 3D-Engine verbessert die *Nachvollziehbarkeit und Begründbarkeit* des Modells.
Die Evaluierung diente dazu, den im Kapitel 5 beschriebenen Prototypen, von Experten bewerten zu lassen und die Frage zu beantworten, welche Ansprüche eine Visualisierung in 3D erfüllen muss, um die Nachvollziehbarkeit und Begründbarkeit von neuronalen Netzen zu verbessern.

6.2 Durchführung der Evaluation

Im folgenden Abschnitt wird der Aufbau des Fragebogens und die gestellten Fragen erläutert (Abschnitt 6.2.1) und anschließend die Durchführung der Evaluation vorgestellt (Abschnitt 6.2.2).

6.2.1 Der Fragebogen

Für die Evaluation wurde ein Fragebogen erstellt, um die Visualisierung bewerten zu lassen. Der Fragebogen wurde hierfür in zwei Teile gegliedert. Im ersten Teilen wurden allgemeine Informationen über die Teilnehmer abgefragt und der zweite Teil diente der Bewertung des Prototypen.
Ziel des ersten Teils des Fragebogens war es, die Erwartungen der Teilnehmer an eine Visualisierung von neuronalen Netzen festzuhalten, um die spätere Bewertung besser nachvollziehen zu können. Zunächst wurde der Kenntnisstand mit maschinellen Lernverfahren im Allgemeinen und mit neuronalen Netzen im Speziellen abgefragt. Es wurde auch gefragt, ob bereits Erfahrungen mit *TensorFlow* vorhanden seien. Für einen besseren Vergleich der Bewertung wurde gefragt, ob die Teilnehmer Software zur Visualisierung neuronaler Netze kennen würden und falls dies der Fall

sein sollte, wie diese Software bewertet worden wäre. Abschließend wurden die Erwartungen an den Prototypen festgehalten.
Der zweite Teil des Fragebogens wurde genutzt, um die Visualisierung bewerten zu lassen. Sowohl einzelne Komponenten als auch die Darstellung des Modells als Ganzes sollten von den Teilnehmern in Hinblick auf die *Nachvollziehbarkeit und Begründbarkeit* bewertet werden. Hierbei wurde gezielt nach der Wichtigkeit der Komponenten gefragt. Über offene Fragen konnte Feedback zur Visualisierung gegeben werden. Es wurde unter anderem gefragt, ob die Erwartungen erfüllt worden wären und welche Aspekte verbessert werden müssten. Die in Kapitel 5.5 vorgestellte Technik zur Visualisierung großer Modelle wurde ebenfalls im Fragebogen berücksichtigt, um zu erfahren, ob diese Form der Darstellung von Vorteil sein könnte.

6.2.2 Ablauf der Evaluation

Für die Erstellung des Fragebogens wurde die Webseite *Umfrageonline* verwendet. Die Nutzung ist für Studenten kostenlos und bei der Gestaltung des Fragebogens stehen verschiedene Vorlagen zur Verfügung. Es können unter anderem *Multiple-Choice* Fragen, *Bewertungsskalen* und *offene Fragen* genutzt und auch eigene Bilder und Texte eingebaut werden. Es können Abhängigkeiten zwischen Fragen definiert werden, so dass Fragen nur angezeigt werden, wenn eine vorherige Frage positiv oder negativ beantwortet wurde.
Die Antworten der Teilnehmer werden vom System automatisch gespeichert und auf Wunsch ausgewertet. Diagramme werden automatisch erstellt. Wegen diesem Funktionsumfang wurde sich für diese Plattform zur Erstellung des Fragebogens entschieden.
Mit zwei Teilnehmern wurde die Evaluation durchgeführt. Die Teilnehmer wurden als *Experten* gezielt ausgewählt. Als *Experte* wurden im Vorfeld Personen definiert, welche beruflich mit neuronalen Netzen arbeiten. Es wurde sich für *Experten* entscheiden, um die Aussagekraft der Evaluation zu maximieren.

Am 2. und 11. November 2018 wurde die Evaluation durchgeführt. Diese fand in einem *Team-Raum* des DLR am Standort Köln statt. Der Raum bot einen großen Fernseher für Präsentationen und eine ruhige Umgebung. Für den Fragebogen wurde ein Laptop zur Verfügung gestellt und der Prototyp wurde auf dem Laptop vorgeführt, auf dem der Prototyp entwickelt wurde. Die Visualisierung wurde in der *Unreal Engine* präsentiert.
Nachdem der erste Teil des Fragebogens ausgefüllt worden war, wurde den Teilnehmern der Prototyp vom Entwickler, über den Fernseher, präsentiert. Es handelte sich dabei um die Version, die in Kapitel 5 beschrieben wurde.

Das neuronale Netz wurde trainiert und anschließend wurden alle visualisierten Informationen zusammenhängend den Teilnehmern präsentiert. Sowohl die Funktionsweise, als auch das Ziel hinter den programmierten Komponenten der Visualisierung wurden erläutert. Es wurde auch auf Funktionen, die nicht wie gewünscht funktionierten, hingewiesen, zum Beispiel die Größe der Filtermaske (siehe Kapitel 5.2.3). Erwähnt wurde auch, dass die Visualisierung des *Flatten Layer* als unpraktisch empfunden wird.
Während der Präsentation hatten die Teilnehmer die Möglichkeit Fragen zu stellen, welche beantwortet wurden. Die Reaktionen der Teilnehmer während der Präsentation wurden beobachtet, um diese in die Auswertung der Evaluation einfließen lassen zu können.

Nach der Präsentation hatten die Teilnehmer die Möglichkeit Feedback zu geben. Dies geschah sowohl über den Fragebogen, als auch über ein persönliches Gespräch. Das Feedback aus den Gesprächen wurde schriftlich festgehalten.
Abschließend wurde den Teilnehmern erklärt, wie eigene Netzwerke mit der Visualisierung genutzt werden können. Hierfür wurden die beiden Methoden *createNetwork* und *createLayerResults* erläutert (siehe Kapitel 5.4). Daraufhin wurde in einer Diskussionsrunde über die Methoden gesprochen, um herauszufinden, wie einfach die Verwendbarkeit der Visualisierung ist.

6.3 Ergebnisse der Evaluation

Im folgenden Abschnitt werden die festgehaltenen Ergebnisse der Evaluation vorgestellt. Zunächst werden die Ergebnisse zusammenfassend wiedergegeben und anschließend dazu genutzt, die Fragestellung aus Abschnitt 6.1 zu beantworten.

6.3.1 Beschreibung

Beide Teilnehmer haben *Informatik* studiert und arbeiteten zum Zeitpunkt der Evaluation als wissenschaftliche Mitarbeiter beim *Deutschen Zentrum für Luft- und Raumfahrt*. Mit maschinellen Lernverfahren hatten beide bereits mehrere Jahre Erfahrung. Ein Teilnehmer arbeitete bereits seit fünf Jahren mit neuronalen Netzen, während der andere Teilnehmer ein Jahr Erfahrung hatte. Beide schätzten sich als *erfahren* ein (siehe Abbildung 80). *Convolutional Neural Networks* waren bekannt, doch wurde nicht mit ihnen direkt gearbeitet. Auch mit *TensorFlow* war nur ein Teilnehmer vertraut. Dieser Teilnehmer kannte durch *TensorFlow Playground* (siehe Kapitel 2.2.3)

	1 keine Erfahrung (1)		2 (2)		3 (3)		4 (4)		5 Sehr erfahren (5)			
	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Ø	±
Erfahrung	-	-	-	-	-	-	2x	100,00	-	-	4,00	0,00

Abbildung 80: Beide Teilnehmer schätzten sich als *erfahren* mit neuronalen Netzen ein.

auch eine Software zur Visualisierung von neuronalen Netzen, die ihm gefallen hat, weil der Einfluss der einzelnen Neuronen auf die Klassifizierung visualisiert wurde. Dafür fehlte ihm die Möglichkeit eigene Modelle verwenden zu können. Der andere Teilnehmer kannte noch keine Software zur Visualisierung.

Die Erwartungen der Teilnehmer an Software zur Visualisierung neuronaler Netze überschnitten sich in einem Punkt: Es sollte angezeigt werden, wann ein Neuron aktiv ist und wie sich die Aktivierung durch Änderungen der Parameter verändert. Darüber hinaus wurde erwartet, dass die Architektur des Modells visualisiert wird und eine Schnittstelle vorhanden ist, die es ermöglicht, verschiedene Arten von Modellen einbinden zu können. Die Visualisierung sollte jedoch immer gleich bleiben. Es sollte zudem möglich sein die Komplexität der Modelle anpassen zu können.

Der zweite Teil des Fragebogens begann mit der Bewertung der Darstellung des Modells. Vor allem die Darstellung der *Layer* wurde als *Sehr Wichtig* empfunden (siehe Abbildung 81).
Auch die Schriftzüge über den *Layern* waren den Teilnehmern *wichtig* und

	Sehr wichtig (1)		Eher wichtig (2)		Unentschieden (3)		Weniger wichtig (4)		Gar nicht wichtig (5)			
	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Ø	±
Input	1x	50,00	1x	50,00	-	-	-	-	-	-	1,50	0,71
Layer	2x	100,00	-	-	-	-	-	-	-	-	1,00	0,00
Namen der Layer	1x	50,00	1x	50,00	-	-	-	-	-	-	1,50	0,71
Neuronen	1x	50,00	-	-	-	-	1x	50,00	-	-	2,50	2,12

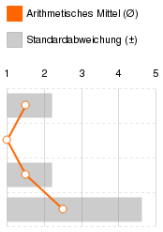


Abbildung 81: Vor allem die *Layer* wurden als *sehr wichtig* empfunden. Bei den Neuronen waren sich die Teilnehmer nicht einig.

die Visualisierung der Eingabe der Klassifizierung. Ein Teilnehmer gab an, dass die Darstellung der *Neuronen* weniger wichtig sei.
Anschließend wurde auf die einzelnen Informationen, die visualisiert werden, eingegangen. Drei Aspekte wurden von den Teilnehmern als *sehr wich-*

tig bewertet: Die Darstellung der Testdaten, die Filterung der Testdaten und die Aktualisierung der Werte während des Trainings (siehe Abbildung 82). Mit *eher wichtig* oder *sehr wichtig* wurden die Basis-Informationen, die Dar-

	Sehr wichtig (1)		Eher wichtig (2)		Unentschieden (3)		Weniger wichtig (4)		Gar nicht wichtig (5)			
	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Ø	±
Basis Informationen	1x	50,00	1x	50,00	-	-	-	-	-	-	1,50	0,71
Gewichtsmaske	1x	50,00	-	-	1x	50,00	-	-	-	-	2,00	1,41
Testdaten	2x	100,00	-	-	-	-	-	-	-	-	1,00	0,00
Convolution	2x	100,00	-	-	-	-	-	-	-	-	1,00	0,00
Output	1x	50,00	1x	50,00	-	-	-	-	-	-	1,50	0,71
Aktivierung (Dense Layer)	1x	50,00	1x	50,00	-	-	-	-	-	-	1,50	0,71
Echtzeit Aktualisierung	2x	100,00	-	-	-	-	-	-	-	-	1,00	0,00

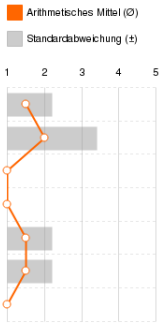


Abbildung 82: Die Bewertung der einzelnen Informationen, die visualisiert werden.

stellung des Ergebnisses eines einzelnen Neurons und die farbliche Repräsentation der Aktivierungswerte der *Dense Neuronen* eingeschätzt. Die dargestellten Werte der Filtermaske der Neuronen wurden von einem Teilnehmer für weder wichtig noch unwichtig gehalten.

Abschließend wurden noch weitere Aspekte, wie zum Beispiel das Hauptmenü, von den Teilnehmern bewertet (siehe Abbildung 83). Die Teilnehmer bewerteten die Echtzeitfähigkeit des Prototypen, die Interaktion mit der Maus und die freie Bewegung in der Szene als *sehr wichtig*. Als wichtig wurde das Hauptmenü empfunden, aber bei dem zusätzlichen Kamera-Fenster (*Informationsfenster*, siehe Kapitel 5.3.3) gab ein Teilnehmer an, dass dieses weder wichtig noch unwichtig sei.

	Sehr wichtig (1)		Eher wichtig (2)		Unentschieden (3)		Weniger wichtig (4)		Gar nicht wichtig (5)			
	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Ø	±
Hauptmenü	1x	50,00	1x	50,00	-	-	-	-	-	-	1,50	0,71
Informationsfenster	1x	50,00	-	-	1x	50,00	-	-	-	-	2,00	1,41
Freies Umsehen	2x	100,00	-	-	-	-	-	-	-	-	1,00	0,00
Maus-Interaktion	2x	100,00	-	-	-	-	-	-	-	-	1,00	0,00
Echtzeit	2x	100,00	-	-	-	-	-	-	-	-	1,00	0,00

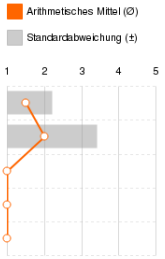


Abbildung 83: Die Bewertung der einzelnen Informationen, die visualisiert werden.

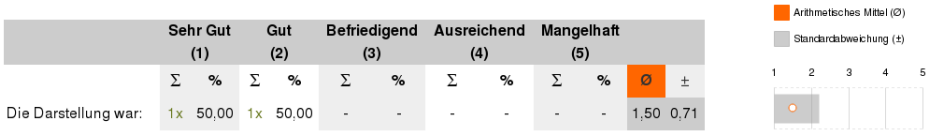
Nach den Bewertungstabellen hatten die Teilnehmer die Möglichkeit über offene Fragen ihren Eindruck von der Visualisierung, in Hinblick auf die

Nachvollziehbarkeit und Begründbarkeit zu schildern. Gut gefallen hat den Teilnehmern die Möglichkeit, die Bilder der Testdaten filtern zu lassen und die Anzeige des Ergebnisses eines Neurons, wenn eine Eingabe klassifiziert wurde. Es wurde herausgestellt, dass die Idee hinter der Visualisierung für sehr nützlich gehalten wurde. Mithilfe einer solchen Visualisierung wäre das *Debuggen* eines neuronalen Netzes möglich. Auch die alternative Darstellung des Modells (siehe Kapitel 5.5) wurde positiv genannt. Diese würde die Übersicht erhöhen und Details reduzieren.
Nicht gut gefallen haben den Teilnehmern die Anzeige der Gewichte der Neuronen, da diese in einer Konsole besser vergleichbar wären und es fehlte ihnen eine Zusammenfassung der Informationen eines *Layers*. Beispielsweise wurde angemerkt, dass die *Aktivierungswahrscheinlichkeit* der Neuronen innerhalb eines Layers gemittelt und für den Layer angezeigt werden sollte.

Auf die Frage, ob die Erwartungen erfüllt wurden, gaben die Teilnehmer an, dass ihre Erwartungen an den Prototypen erfüllt wurden. Die Funktionen der einzelnen Neuronen seien nachvollziehbar visualisiert worden und hätten damit einen Einblick in die *Black Box* (siehe Kapitel 2) gewährt. Durch die alternative Visualisierung haben die Teilnehmer einen Einblick erhalten, wie große Modelle integriert werden können.
Ein Teilnehmer gab an, dass er noch eine Schnittstelle erwartet hätte, die die Integration anderer Modelle, auch unabhängig von *TensorFlow*, ermöglicht.
Auch Verbesserungsvorschläge konnten angegeben werden. Hierbei wurde erneut erwähnt, dass die Informationen der Neuronen für den gesamten Layer zusammengefasst werden sollten und eine Projektion auf die Eingabebilder würde die Nachvollziehbarkeit ebenfalls verbessern. Eine flexible Schnittstelle wurde ebenfalls noch einmal angesprochen.

Abschließend wurde den Teilnehmern die Frage gestellt, wie die alternative Darstellung aus Kapitel 5.5 bewertet werden würde und ob die *Nachvollziehbarkeit* von dieser Form der Darstellung profitieren könnte. Beiden Teilnehmern hat diese Darstellung gut gefallen (siehe Abbildung 84) und es wurde Potential darin gesehen, auf diese Weise sehr große Modelle zu visualisieren.

Über ein anschließendes Gespräch mit den Teilnehmern, in dessen Verlauf auch das *Python Skript* (siehe Kapitel 3.2.2) präsentiert wurde, konnte noch weiteres *Feedback* festgehalten werden. Es wurde gesagt, dass die zwei Methoden *createNetwork* und *createLayerResults* als intuitiv benutzbar empfunden wurden. Dadurch hätten sich die Teilnehmer vorstellen können, ein eigenes Modell auf einfache Weise visualisieren zu lassen. Wünschenswert wäre aber eine Schnittstelle, welche die Modelle automatisch lädt und auf



	Sehr Gut (1)		Gut (2)		Befriedigend (3)		Ausreichend (4)		Mangelhaft (5)			
	Σ	%	Σ	%	Σ	%	Σ	%	Σ	%	Ø	±
Die Darstellung war:	1x	50,00	1x	50,00	-	-	-	-	-	-	1,50	0,71

Abbildung 84: Die Bewertung der *Level of Detail* Technik.

Grundlage von Eingabeparametern der Benutzer das Modell visualisiert. Eine Verwendung von *Virtual Reality* Brillen wurde als nicht vorteilhaft von den Teilnehmern eingeschätzt. Die Interaktion mit den Menüs konnte sich nur schwer vorgestellt werden. Es wurde aber vorgeschlagen die *HoloLens*[65] zu nutzen, da diese in Verbindung mit einer Maus genutzt werden könnte.
Im Rahmen der Gespräche wurde auch über andere Arten von Daten gesprochen, wie zum Beispiel Audiodaten. Die Teilnehmer konnten sich vorstellen, dass eine solche Visualisierung sich auch für diese Daten eignen würde, weil die *Unreal Engine* Audiodaten unterstützt und wiedergeben kann.

6.3.2 Analyse

Die Teilnehmer gaben an, dass vor allem die Darstellung der Layer für die Visualisierung der Struktur des vorgeführten Modells wichtig gewesen seien. Auch die alternative Darstellung des Modells mit dem *Level of Detail* Ansatz gefiel ihnen, weil das neuronale Netz zu Beginn abstrakt dargestellt und erst auf Wunsch des Benutzers detaillierter visualisiert wird. Eine reine Darstellung als *Schichtenmodell* (siehe Kapitel 5.2.2) schien den Teilnehmern nur für kleine Modelle sinnvoll. Die Funktion, die Neuronen ausblenden zu können, wurde aber auch als sinnvoll empfunden.
Auf Grundlage dieser Anmerkungen der Teilnehmer sollte eine Software zur Visualisierung von neuronalen Netzen die Modellstruktur visualisieren, um den Ablauf nachvollziehbarer zu machen. Dabei sollte immer die Möglichkeit bestehen das Modell vereinfacht zu betrachten und nur auf Wunsch des Benutzers Bestandteile einzublenden. Eine dynamische Darstellung des Modells ist unabdingbar für große Modelle, um eine bessere Nachvollziehbarkeit zu erreichen.

Im Abschnitt des Fragebogens, in dem die Teilnehmer Verbesserungen vorschlagen konnten, wurde häufig eine Zusammenfassung der Ergebnisse der Neuronen genannt. Für jeden Layer sollten die Ergebnisse sinnvoll

[65] Microsoft HoloLens

zusammengefasst werden, da dies die Übersichtlichkeit erhöhen würde. Reine Zahlenwerte, wie der Bias-Wert oder die Filtermaske der Neuronen, wurden für die Nachvollziehbarkeit als nicht relevant empfunden. Dafür wurde die Visualisierung der Ergebnisse der Neuronen als sehr wichtig bewertet. Dies würde die Nachvollziehbarkeit erhöhen, da die Funktionsweise der Neuronen mithilfe der Filterung der Testdaten und dem Ergebnisbild einer Klassifizierung verdeutlicht wird. Auch die Aktualisierung der Informationen während des Trainings wurden als hilfreich angesehen. Eine Projektion der Ergebnisse auf die Eingabe wurde jedoch vermisst.
Diese Ergebnisse der Evaluation zeigen, dass Informationen von neuronalen Netzen auf übersichtliche Weise präsentiert werden müssen, um die Nachvollziehbarkeit zu verbessern. Reine Zahlenwerte sollten dabei vermieden werden und die Ergebnisse der Neuronen im Vordergrund stehen. Mithilfe von Testdaten sollten die Funktionsweisen der Neuronen visualisiert werden und Veränderungen der Ergebnisse sichtbar sein. Wichtig ist, dass die Informationen der Neuronen eines Layers zusammengefasst werden und bereits über den Layer abrufbar sind. Dies kann das Auffinden von Fehlern verbessern, da nicht jedes Neuron einzeln betrachtet werden muss. Eine Rückprojektion der Ergebnisse auf die Eingabe könnte die Nachvollziehbarkeit und Begründbarkeit noch weiter verbessern.

Eine Schnittstelle zur Visualisierung eigener Modelle wurde sowohl bei den Erwartungen als auch bei den Verbesserungsvorschlägen genannt. Es wurde von den Teilnehmern Wert darauf gelegt, dass keine eigenen Methoden geschrieben werden müssen, sondern nur die Daten übergeben werden und die Schnittstelle dafür sorgt, dass diese visualisiert werden. Die vorgestellten Methoden, die dem Benutzer für den Prototypen zur Verfügung gestellt werden, wurden dennoch als intuitiv und einfach nutzbar eingeschätzt.
Aus diesen Angaben der Teilnehmer lässt sich der Anspruch einer vollautomatischen Schnittstelle ableiten. Eine Schnittstelle sollte es dem Benutzer überlassen, welche Aspekte eines Modells dargestellt werden sollen und im Idealfall wird ein eigenes Dateiformat dazu verwendet. Wichtig ist, dass ein solches Dateiformat unterschiedliche Arten von Modellen unterstützt und nicht auf *TensorFlow* beschränkt ist.

Die Echtzeitfähigkeit der Visualisierung und die Interaktion mit der Maus war beiden Teilnehmern wichtig für die Anwendung. Auch die Orientierung mithilfe der Schriftzüge wurde als sinnvoll erachtet. Gut gefallen hat den Teilnehmern, dass Veränderungen in Form der Deaktivierung von Neuronen vorgenommen werden konnten. Hier wurden sich auch weitere Möglichkeiten gewünscht, das Modell zu verändern, da dies die Nachvollziehbarkeit des Modells und seiner Ergebnisse verbessern würde. Von der Nutzung einer *Virtual Reality* Umgebung wurde abgeraten, da beide Teilneh-

mer Probleme in der Interaktion mit dem neuronalen Netz sahen und keinen Vorteil für die Nachvollziehbarkeit. Allerdings wurde die *HoloLens* als Alternative vorgeschlagen, da die Interaktion mit der Maus für diese umsetzbar wäre.
Daraus lässt sich ableiten, dass eine Visualisierung in 3D in Echtzeit funktionieren sollte, um mit dem Modell sinnvoll interagieren zu können. Eine Interaktion mithilfe der Maus ist wünschenswert, da Benutzer diese Form der Steuerung gewöhnt sind. Die Übersicht in der Szene sollte zu jederzeit gewährleistet werden, damit der Benutzer die Orientierung nicht verliert und Informationen sollten ein- und ausgeblendet werden können, um die Szene übersichtlicher zu gestalten. Veränderungen sollten am Modell vorgenommen werden können, damit die Nachvollziehbarkeit und Begründbarkeit von Ergebnissen verbessert wird, indem die Auswirkungen der Veränderungen visualisiert werden. Eine Präsentation in einer *Virtual Reality* Umgebung könnte sich als schwer erweisen, weil die Benutzer nicht gewohnt sind über Gesten-Steuerung mit Menüs zu interagieren. Die *HoloLens* von Microsoft könnte sich jedoch als geeignet herausstellen, da diese eine *Mixed-Reality* Umgebung bietet und somit auch eine Maussteuerung eingebaut werden könnte.

6.4 Fazit

Die im Rahmen dieser Abschlussarbeit durchgeführte Evaluation des programmierten Prototypen zur Visualisierung eines neuronalen Netzes, hat gezeigt, welchen Ansprüchen eine Visualisierung in 3D gerecht werden sollte, um die Nachvollziehbarkeit und Begründbarkeit eines Modells zu verbessern.
Eine *Schnittstelle* zwischen dem programmierten Modell eines Nutzers und der Visualisierung sollte flexibel mit unterschiedlichen Programmiersprachen und Modellen umgehen können. Dabei sollte der Benutzer entscheiden, welche Aspekte visualisiert werden und für eine einfache Verwendung ein Modell über eine Datei übergeben können.
Mithilfe der Maus sollte die *Interaktion* in der Szene möglich sein, um intuitiv mit Menüs umgehen zu können. Das Modell sollte dabei in *Echtzeit* betrachtet und verändert werden können, um den Ablauf der Vorgänge innerhalb des Modells besser nachzuvollziehen. Dabei sollten die Informationen übersichtlich eingeblendet werden, um die *Orientierung* in der Szene zu verbessern. Auch die *Veränderung der Ergebnisse* sollte für eine bessere Nachvollziehbarkeit präsentiert werden.
Das *Ein- und Ausblenden* von Informationen ermöglicht es dem Benutzer die Visualisierung den eigenen Ansprüchen anzupassen. Große Modelle sollten auch automatisch von der Anwendung abstrahiert dargestellt werden, so dass eine *stufenweise Darstellung* möglich wird. Für jede Abstrakti-

onsstufe sollten Informationen zur Verfügung gestellt werden. Sowohl für *einzelne Neuronen*, um ihre Funktionsweise nachvollziehen zu können, als auch für den *gesamten Layer*, damit der Benutzer eine Zusammenfassung der Ergebnisse erhält und entscheiden kann, ob eine detaillierte Betrachtung notwendig ist.
Die Einbindung einer *Mixed Reality*-Hardware, wie der *HoloLens*, kann die Visualisierung gegebenenfalls erweitern.

7 Fazit

In diesem Kapitel wird die Entwicklung einer Visualisierung für neuronale Netze abschließend zusammengefasst. Es wird erläutert, welche Ziele erreicht wurden und wie weitere Entwicklungsschritte aussehen könnten, um die Visualisierung zu erweitern.

7.1 Zusammenfassung

Das Ziel der vorliegenden Abschlussarbeit war es, eine Visualisierung von neuronalen Netzen zu programmieren, um die Nachvollziehbarkeit und Begründbarkeit zu verbessern. Als zusätzliche Herausforderung sollte die Visualisierung in eine dreidimensionalen Szene integriert werden. Dafür sollte sich in bestehende Verfahren zur Visualisierung von maschinellen Lernverfahren eingearbeitet werden, um ein eigenes Konzept zu entwickeln.

Implementiert wurde die Visualisierung mithilfe der *Unreal Engine* und des integrierten *Blueprints Visual Scripting*-Systems. Als Grundlage für das neuronale Netz wurde die Software *TensorFlow* genutzt (siehe Kapitel 3.2) und zur Übertragung der Daten diente das *TensorFlow Plug-in* (siehe Kapitel 3.2.1).
Die programmierte Anwendung umfasste zwei Szenen. In der ersten Szene wurde ein *Convolutional Neural Network* visualisiert, welches den *MNIST Datensatz* nutzt. Das Modell und seine Struktur wurden mithilfe von dreidimensionalen Würfeln und Sphären, welche Layer und Neuronen darstellen, in der Szene repräsentiert. Jedem Layer wurden Namensschriftzüge hinzugefügt, um die einzelnen Layer identifizieren zu können. Unterschiedliche Klassen von Layern und Neuronen wurden implementiert, um die Klassen den bereitgestellten Informationen anpassen zu können.
Neuronen werden dem Benutzer erst angezeigt, wenn dieser eine Schaltfläche betätigt, um mehr Übersicht zu schaffen und Grafikspeicher zu sparen. Neuronen besitzen Menüs, in denen Informationen aus *TensorFlow* bereitgestellt werden. Sowohl grundlegende Informationen, wie Bias-Werte, werden bereitgestellt, als auch Funktionen zur Veranschaulichung der Funktionsweise von Neuronen, wie die Filterung von Testdaten. Die Klassifizierung einer Eingabe kann nachverfolgt werden durch die Anzeige der Ergebnisse jedes einzelnen Neurons.
Über ein Hauptmenü wurde die Möglichkeit implementiert Trainingsparameter anzupassen und Neuronen können in der Szene deaktiviert werden, um die Auswirkungen dieser Veränderungen am Modell in der Szene beobachten zu können.
In der zweiten Szene wurde ein Ansatz gezeigt, um große Modelle effizient

visualisieren zu können. Dafür wurde sich an dem *Level of Detail* Verfahren orientiert und das Modell wurde in verschiedene Abstraktionsstufen eingeteilt. Dem Benutzer wird das Modell nur in Teilen angezeigt, auf Grundlage seiner Entfernung zum neuronalen Netz in der Szene.
Im Rahmen dieser Abschlussarbeit war es aus zeitlichen Gründen nicht möglich, die vorgestellten Szenen mit unterschiedlichen und größeren Modellen zu testen. Testweise wurde die Anzahl an Neuronen erhöht, um die Performance der Visualisierung zu testen. Es wurde jedoch kein Nachteil darin gesehen, die Visualisierung anhand des *MNIST*-Modells zu präsentieren, weil die Darstellung der Informationen auch bei großen Modellen übertragbar ist.
Auch an eine Einbindung von *Virtual Reality*-Brillen wie der *Oculus Rift* wurde zwischenzeitlich gedacht. Doch auch hier wurde sich aus zeitlichen Gründen dagegen entschieden.

Im Rahmen einer Evaluation wurde die Visualisierung des neuronalen Netzes von zwei Teilnehmern, welche sich gut mit neuronalen Netzen auskannten, bewertet. Dabei sollte die Leitfrage geklärt werden, welche Anforderungen eine Visualisierung erfüllen muss, um die Nachvollziehbarkeit und Begründbarkeit des Modells zu verbessern.
Mithilfe der Evaluation konnte festgehalten werden, dass einige Anforderungen bereits umgesetzt werden konnten, wie zum Beispiel die Darstellung der Ergebnisse und Funktionsweise von Neuronen. Besonders gut bewertet wurde die Möglichkeit das Modell durch Deaktivierung von Neuronen zu verändern und sich in Echtzeit die Veränderung der Ergebnisse anzusehen. Es konnten auch Anforderungen festgehalten werden, welche noch nicht umgesetzt wurden, wie zum Beispiel eine Schnittstelle, die automatisch die notwendigen Schritte vornimmt, um ein Modell zu visualisieren. Auch die Zusammenfassung von Ergebnissen pro Layer wurde als Anforderung festgehalten.
Somit konnte die zu beantwortende Leitfrage im Rahmen der Evaluation beantwortet werden und die erfassten Anforderungen könnten bei der weiteren Entwicklung von Visualisierungen helfen.

Durch diese Abschlussarbeit konnte ein Einblick gewährt werden in die Erstellung einer Visualisierung von neuronalen Netzen mithilfe von 3D-Engines. Neben einem Einblick in aktuelle Forschungsarbeiten zum Thema der Nachvollziehbarkeit und Begründbarkeit von maschinellen Lernverfahren, konnten auch Anforderungen herausgearbeitet werden, welche von Visualisierungen erfüllt werden sollten. Dadurch konnte der gewünschte Lernerfolg im Rahmen dieser Arbeit erreicht werden.

7.2 Ausblick

Durch die Evaluation (siehe Kapitel 6) konnten Verbesserungsvorschläge festgehalten werden, wie die Visualisierung des neuronalen Netzes erweitert werden könnte. Die Ergebnisse der einzelnen Neuronen könnten für jeden Layer zusammengefasst, in einem neuen Menü für Layer, präsentiert werden. Für jede Klasse von Layer müssten die Informationen auf sinnvolle Art zusammengefasst werden. Bei den *Convolutional Layern* könnten die Ergebnisbilder zusammengerechnet werden und anschließend als Bild für den Layer visualisiert werden. Auch die Ergebnisse der Filterung der Testdaten könnte auf diese Weise zusammengefasst werden. Die Bias-Werte könnten gemittelt werden. Bei den *Dense Layern* würde sich auch eine Bestimmung des Mittelwertes der Aktivierungswerte anbieten. Diese gemittelten Ergebnisse könnten einem Benutzer dazu dienen bestimmte Layer zu selektieren, bevor die einzelnen Neuronen betrachtet werden.
TensorFlow unterstützt Methoden des *Keras*-Pakets und das verwendete Plug-in arbeitete ebenfalls mit *Keras-Modellen*. Es werden noch zusätzliche Methoden von *Keras* angeboten, die für die Visualisierung genutzt werden könnten, um die Nachvollziehbarkeit zu verbessern. Die Methode *visualize_saliency* erstellt automatisch *Saliency Maps* für ein Modell. *visualize_activation* erzeugt Bilder, für die ein Neuron die stärkste Aktivierung zeigt und *visualize_cam* implementiert das *Class Activation Maps*-Verfahren. Diese Methoden könnten für die Visualisierung genutzt werden und müssten in die Menüs der Layer und Neuronen eingefügt werden. Allerdings muss das Plug-in dafür angepasst werden, da eine Einbindung in die jetzige Version zu einem Absturz der Anwendung führte.

Auch andere Arten von Modellen könnten eingefügt werden, wie zum Beispiel *Recurrent Neural Networks*. Zusätzliche Informationen, die die Neuronen dieser Modelle enthalten, können den Menüs hinzugefügt werden und gegebenenfalls werden neue Klassen für Neuronen und Layer benötigt.
Darüber hinaus könnten auch andere Arten von Daten eingebaut werden, wie zum Beispiel Audio- oder Textdaten. Für Audiodaten könnten die Neuronen die Dateien abspielen, was bereits von der Unreal Engine unterstützt wird. Die Audiodateien könnten auch als Frequenzspektren visualisiert werden, um Veränderungen optisch hervorzuheben.
Neuronale Netze für Textklassifizierung könnten wiederum Wörter für jedes Neuron anzeigen oder Wörter in einem Text könnten eingefärbt und für jedes Neuron eingeblendet werden. Texte und Wörter könnten als Texturen in den Menüs angezeigt werden oder aber als Textfelder, welche ebenfalls von Menüs unterstützt werden.

Von den Teilnehmern der Evaluation wurde eine Schnittstelle für die Visualisierung gewünscht, die eine Übergabe einer Datei ermöglicht, die au-

tomatisch in eine Visualisierung übersetzt wird. Dafür müsste die bisherige Schnittstelle des Prototypen ersetzt werden. Eine Dateiübergabe müsste möglich sein und das Modell automatisch ausgelesen werden. Solange *TensorFlow* für das Modell genutzt wird, könnten die implementierten Methoden für diese Arbeit beibehalten werden. Sollte es sich um eine andere Art von Modell handeln, müsste auch das Auslesen der Daten aus dem Modell verändert werden.

Für eine Visualisierung mithilfe der *HoloLens* müsste die Unreal Engine erweitert werden, mithilfe von Plug-ins. In diesem Bereich wurden bereits Plug-ins veröffentlicht, doch unterstützt die Unreal Engine bisher die *HoloLens* ohne diese nicht. *Virtual Reality* Brillen, wie zum Beispiel die *Oculus Rift*, werden hingegen bereits unterstützt und somit wäre es möglich die Visualisierung für eine solche Hardware zu übertragen.

Literatur

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[FV17] Ruth Fong and Andrea Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. *CoRR*, abs/1704.03296, 2017.

[KAKC17] Minsuk Kahng, Pierre Y. Andrews, Aditya Kalro, and Duen Horng Chau. Activis: Visual exploration of industry-scale deep neural network models. *CoRR*, abs/1704.01942, 2017.

[KSA+18] Pieter-Jan Kindermans, Kristof T. Schütt, Maximilian ALber, Klaus-Robert Müller, Dumitru Erhan, Been Kim, and Sven Dähne. Learning how to explain neural networks: Patternnet and patternattribution. In *International Conference on Learning Representations*, 2018.

[Nie15] Michael A. Nielson. *Neural Networks and Deep Learning*. Determination Press, 2015.

[OSJ+18] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schuber, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 2018. https://distill.pub/2018/building-blocks.

[SCS+17] Daniel Smilkov, Shan Carter, D. Sculley, Fernanda B. Viegas, and Martin Wattenberg. Direct-manipulation visualization of deep networks. *CoRR*, abs/1708.03788, 2017.

[SDBR14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

[SVZ13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convoltional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.

[vdMH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research 9*, 2008.

[ZF13] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.

[ZKL+15] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for dsicriminative localization. *CoRR*, abs/1512.04150, 2015.

Anhang

Fragebogen der Evaluation

Nachvollziehbarkeit und Begründbarkeit von machinellen Lernverfahren

Allgemeine Informationen

Vielen Dank für Ihre Teilnahme an der folgenden Evaluation. Im Rahmen meiner Masterarbeit habe ich mich mit der Visualisierung von neuronalen Netzwerken beschäftigt. Mithilfe meiner Visualisierung haben Sie die Möglichkeit ein neuronales Netzwerk anhand des MNIST Datensatzes zu betrachten.
Die Evaluation beginnt mit einem Theorie-Teil, danach wird Ihnen die Visualisierung präsentiert und zum selber ausprobieren zur Verfügung gestellt. Anschließend folgt Ihre Bewertung und eine abschließende Erläuterung + Diskussion über die Portierung von eigenen Netzwerken.

Generelle Informationen *

Studium

Beruf

E-Mail (optional)

Vorkenntnisse

Wie lange arbeiten Sie schon mit maschinellen Lernverfahren? *

Wie lange arbeiten Sie schon mit Neuronalen Netzwerken? *

Vorkenntnisse

Mit welchen maschinellen Lernverfahren haben Sie bereits gearbeitet? *

Mit welchen Neuronalen Netzwerken haben Sie bereits gearbeitet? *

Vorkenntnisse

Wie erfahren schätzen Sie sich mit Neuronalen Netzen ein? *

	1 keine Erfahrung	2	3	4	5 Sehr erfahren
Erfahrung	◯	◯	◯	◯	◯

Vorkenntnisse

Kennen Sie "TensorFlow"? *

◯ ja

◯ nein

Vorkenntnisse

Welche Erfahrungen haben Sie bereits mit "TensorFlow" gemacht? *

Vorkenntnisse

Kennen Sie bereits Software, die Neuronale Netze visualisiert? *

- ◯ ja
- ◯ nein

Vorkenntnisse

Welche Software kennen Sie? *

Wie hat Ihnen die Software gefallen? *

Vergeben Sie eine Schulnote

	1 Sehr Gut	2 Gut	3 Befriedigend	4 Unzureichend	5 Mangelhaft
Note	◯	◯	◯	◯	◯

Was hat Ihnen an der Software besonders gefallen und was hat Ihnen gefehlt? *

Vorkenntnisse

Welche Erwartungen haben Sie an einer Software zur Visualisierung von Neuronalen Netzen? *

In Hinblick auf eine bessere Nachvollziehbarkeit und Begründbarkeit

Nun wird Ihnen die Visualisierung präsentiert

Hiermit ist der erste Teil der Evaluation zu Ende. Nun wird Ihnen von mir die Visualisierung präsentiert und danach können Sie mit dem zweiten Teil fortfahren.

Visualisierung - Bewertung Demo #1

Bitte bewerten Sie in diesem Abschnitt die Visualisierung in Hinblick auf die Nachvollziehbarkeit und Begründbarkeit.

Wie wichtig war Ihnen die Darstellung der Struktur des Netzwerks? *

	Sehr wichtig	Eher wichtig	Unentschieden	Weniger wichtig	Gar nicht wichtig
Input	◯	◯	◯	◯	◯
Layer	◯	◯	◯	◯	◯
Namen der Layer	◯	◯	◯	◯	◯
Neuronen	◯	◯	◯	◯	◯

Wie wichtig war Ihnen die Darstellung der Informationen der Neuronen? *

	Sehr wichtig	Eher wichtig	Unentschieden	Weniger wichtig	Gar nicht wichtig
Basis Informationen	◯	◯	◯	◯	◯
Gewichtsmaske	◯	◯	◯	◯	◯
Testdaten	◯	◯	◯	◯	◯
Convolution	◯	◯	◯	◯	◯
Output	◯	◯	◯	◯	◯
Aktivierung (Dense Layer)	◯	◯	◯	◯	◯
Echtzeit Aktualisierung	◯	◯	◯	◯	◯

Wie wichtig waren Ihnen folgende Aspekte? *

	Sehr wichtig	Eher wichtig	Unentschieden	Weniger wichtig	Gar nicht wichtig
Hauptmenü	◯	◯	◯	◯	◯
Informationsfenster	◯	◯	◯	◯	◯
Freies Umsehen	◯	◯	◯	◯	◯
Maus-Interaktion	◯	◯	◯	◯	◯
Echtzeit	◯	◯	◯	◯	◯

Visualisierung - Positive und negative Erfahrungen

Was hat Ihnen an der Visualisierung gefallen? *

Was hat Ihnen an der Visualisierung nicht gefallen? *

Visualisierung - Erwartungen

Wurden Ihre Erwartungen an die Visualisierung erfüllt? *

- ◯ ja
- ◯ nein

Warum wurden Ihre Erwartungen erfüllt und/oder warum nicht? *

Visualisierung - Verbesserungsvorschläge

Welche Aspekte fehlen Ihrer Meinung nach oder müssten verbessert werden, um die Nachvollziehbarkeit und Begründbarkeit zu verbessern? *

Visualisierung - Bewertung Demo #2

Wie hat Ihnen die Darstellung des Netzwerks gefallen? *

	Sehr Gut	Gut	Befriedigend	Ausreichend	Mangelhaft
Die Darstellung war:	◯	◯	◯	◯	◯

Profitiert die Nachvollziehbarkeit von dieser Form der Darstellung? Warum? *

Abschluss

Abschließend möchte ich mit Ihnen noch über die Möglichkeit sprechen, eigene Netzwerke in die vorhandene Struktur der Visualisierung einzubinden. Die Bewertung dieses Vorgangs wird durch ein persönliches Interview erfolgen.

» Umleitung auf Schlussseite von Umfrage Online