

Berliner Hochschule für Technik Berlin
Fachbereich VI – Informatik und Medien

Bachelorarbeit

**Erzeugung von Bildern mittels Neuronalen
Netzen**

Stefan Berger
Medieninformatik
Matrikel-Nr. 854184

Berlin, 6. April 2021

Betreut von: Prof. Dr. F. Gers

Gutachter: Prof. Dr. J. Schimkat

Zusammenfassung

Im Experiment und im Inhalt dieser Bachelorarbeit wird die Frage beantwortet, ob ein künstliches neuronales Netz lernen kann Bilder zu generieren. Objekte auf generierten Bildern besitzen Eigenschaften, die durch einen Shader generiert werden. Ein Shader ist ein Algorithmus, der für ein Objekt eine Textur und Lichtreflexionen berechnet. Kann ein künstliches neuronales Netz darauf trainiert werden, die Form, Textur und die Reflexionseigenschaften eines Objektes zu lernen?

Künstliche Intelligenz hat ein breites Anwendungsspektrum. Computer Vision, also das Erkennen von Objekten in einem Bild, nachträgliche Manipulation von Bildern in Fotobearbeitungssoftware mittels Filtern und Korrekturmöglichkeiten oder Texterkennung sogar von Handschriften sind nur einige Beispiele für Ergebnisse, die künstliche neuronale Netze allein im visuellen Bereich erzielen.

Mathematisch haben künstliche neuronale Netze bereits eine längere Geschichte. Trotz der verschiedenen Einsatzbereiche weisen die unterschiedlichen Algorithmen in ihrem Aufbau einige Ähnlichkeiten auf. In dieser Arbeit werde ich die Grundlagen des maschinellen Lernens erarbeiten. Anhand ausgewählter Implementierungen werde ich die Möglichkeit untersuchen, Bilder durch ein künstliches neuronales Netz generieren zu lassen.

Inhaltsverzeichnis

1. Einleitung	1
1. Ziel der Arbeit	2
2. Bisherige Arbeiten	3
2. Modell	7
1. Tensoren	7
2. Kreuzentropie	8
3. Deep Neural Networks	11
4. Convolutional Neural Networks	12
5. Batch Normalization	13
6. U-Net-Architektur	14
7. Generative Adversarial Networks	14
8. Conditional Generative Adversarial Nets	15
3. Implementierung	16
1. Github	16
2. Ubuntu Linux	17
3. Python	17
4. CUDA	19
5. Image-To-Image-Translation in Python	19
4. Experimente und Resultate	24
1. Vorbereitung der vorhandenen Eingabedaten	24
2. Erstellung eigener Eingabedaten	26
3. Anwendung herkömmlicher Shader	28
4. Hyperparameter	29
5. Performancebeobachtungen	29
5. Diskussion	36
Literatur	41
Bildnachweis	43
Abbildungsverzeichnis	45

Anhang	48
A. Quelltexte	48
1. Image-To-Image-Translation main.py	48
2. Python-Script zur Generierung von JPEG-Bilddateien aus NDJSON-Informationen	55
3. Blender-Python-Script zur Generierung von Tischen	56

1. Einleitung

Obwohl die Idee für eine Maschine, die anhand eingegebener Daten selbstständig Entscheidungen treffen kann und die ersten praktischen Ansätze für künstliche neuronale Netze schon einige Jahrzehnte alt sind, findet der Einsatz derartiger Algorithmen erst seit einigen Jahren statt. Viele Erfindungen, die vor 30 bis 50 Jahren in Filmen und Serien Science Fiction darstellten, sind inzwischen nicht nur Realität, sondern auch alltagstauglich. Zu den wichtigsten Beispielen zählen verbale Schnittstellen an Computersystemen und in Armbanduhren, autonome Fahrzeuge etwa in Gestalt von Parkassistenten und verschiedene Verfahren zur biometrischen Identitätsprüfung.

Andere rasche technologische Fortschritte aus der Vergangenheit haben nicht immer nur die Lebensqualität der beteiligten Personen erhöht, sondern stellten durch Missbrauch gelegentlich sogar Gefahren dar. So wie das Internet auch zur Verbreitung von Falschinformationen und die sichere Verschlüsselung von gespeicherten Daten auch zur Verschleierung von Straftaten genutzt werden kann, ist die Generierung von täuschend echten Bildern unter Umständen geeignet, persönlichen, finanziellen oder anders gearteten Schaden zu verursachen.

Weiterhin existieren bei der Auswahl der Trainingsdaten für künstliche neuronale Netze rechtliche Grenzen. Bilddaten sind in mehr als ausreichenden Mengen vorhanden, berücksichtigen aber zum Beispiel nicht immer das Recht am eigenen Bild. Für diese Bachelorarbeit sind die Anforderungen an die Bildqualität außerdem sehr hoch, da möglichst auch Texturen und Lichtreflexionen erlernt werden sollen. Modellgrafiken mit geeigneten Material- und Beleuchtungseigenschaften gibt es zwar auch, aber nur in weitaus geringeren Mengen. Für solche Fremdarbeiten, die sehr arbeitsaufwendig sind, wäre auch die Klärung der Nutzungsrechte erforderlich geworden. Für Trainingsergebnisse, die das mentale Modell einer breiten Nutzergemeinschaft reflektieren, sind grundsätzlich auch Daten aus möglichst vielen verschiedenen Quellen erforderlich.

Es ist deshalb eine Brücke geschlagen worden zwischen Trainingsdaten, die zum einen aus zufällig ausgewählte Benutzereingaben bestehen, und solchen, die bestimmte Qualitätseigenschaften erfüllen und in beliebiger Menge erstellt werden können. Ein künstliches neuronales Netz soll aus Skizzen des “Quick, Draw!”-Datasets von Google hochwertig gestaltete Figuren generieren.

1. Ziel der Arbeit

Für diese Arbeit habe ich mir zum Ziel gesetzt, anhand wissenschaftlicher Literatur die Grundlagen und Methoden des maschinellen Lernens zu erarbeiten. Im Rahmen des Themas dieser Abschlussarbeit habe ich mit verschiedenen künstlichen neuronalen Netzen experimentiert, um Bilder zu generieren.

Im Zuge der Ausarbeitung des Konzeptes waren zunächst verschiedene Rahmenbedingungen zu bewerten. Trainingsdaten müssen in großen Mengen verfügbar sein. Urheberrechtliche oder gar Lizenzfragen sollten möglichst nicht auftreten. Weiterhin sollten die Eingabebilder eine gewisse Homogenität aufweisen, um ein Modell auf die Aspekte Textur und Shading trainieren zu können. Meine Recherche verschiedener Methoden zur Generierung von Bildern mittels künstlichen neuronalen Netzen ist im nächsten Abschnitt zu den bisherigen Arbeiten näher beschrieben.

Die Entscheidung fiel schließlich auf ein Machine-Learning-Modell namens Image-To-Image-Translation [1] oder kurz Pix2Pix, das unter anderem die Ergebnisse der Canny-Edge-Detection [2] eines Bildes, also nur die Umrisse des abgebildeten Objekts, in das Originalbild zurückübersetzen kann. In meinen Experimenten habe ich das Modell erfolgreich darauf trainiert, handgezeichnete Skizzen in fotorealistische Abbildungen von Alltagsgegenständen zu übersetzen. Die Grundlagen und Bausteine des Algorithmus sind Kapitel 2 und der Algorithums selbst und seine Implementierung in Python im Abschnitt zu Image-To-Image-Translation in Kapitel 3 beschrieben.

Eine umfangreiche Sammlung an handgezeichneten Eingabebildern ist im Dataset des “Quick, Draw!”-Minigames von Google zu finden. Ziel des kostenlosen Browser-Spiels ist es, ein vorgegebenes Motiv innerhalb 20 Sekunden mittels Maus, Touchpad oder einem vergleichbaren Eingabegerät so zu zeichnen, dass ein zeitgleich aktiver Algorithmus die Zeichnung klassifizieren kann. Das Dataset besteht aus den erstellten Zeichnungen, ist nach Motiven sortiert und ebenfalls kostenlos verfügbar. Somit ist es für meine Experimente bestmöglich geeignet. Das Minigame ist auf <https://quickdraw.withgoogle.com/> zu finden. Die Verwendung des “Quick, Draw!”-Datasets ist in Kapitel 4 beschrieben.

Es gibt verschiedene denkbare praktische Anwendungsfälle für diese automatische Übersetzung. Vergleichbar mit Handschrifterkennung oder der Erkennung primitiver geometrischer Formen durch Geräte mit Stift- oder Fingereingabe kann die Erkennung von handgezeichneten Alltagsgegenständen etwa durch Smartboards die Illustration von Alltagssituationen erleichtern. In meinen Experimenten habe ich Skizzen von Autos in fotorealistische Abbildungen übersetzt. Damit ist die Darstellung einer Situation im Straßenverkehr vorstellbar, wie sie in einer Fahrschule vorteilhaft sein könnte. Ein weiterer konkreter Anwendungsfall ist der elektronisch gestützte Entwurf einer Inneneinrichtung mit verschiedenen Möbelstücken.

Es ist in begrenztem Maße möglich, die Zahl der vorhandenen Eingabebilder durch Data Augmentation zu erhöhen [3]. Dabei werden die Bilder gespiegelt, gedreht, skaliert oder auf andere Weise verändert oder verfälscht. Es ist außerdem denkbar, die Eingabedaten selbst zu

erstellen. Weil der Zusammenhang mit prozeduralen Shadern durch die Aufgabenstellung gegeben ist, habe ich mich dazu entschieden einen wesentlichen Teil der Eingabebilder zu entwerfen und automatisiert zu erstellen. Es sind Bilder verschiedener Alltagsgegenstände entstanden, die sich in der Form unterscheiden, deren Texturen und Lichtverhältnisse sich aber gleichen. Dadurch war es mir möglich das Modell auf die Übersetzung der handgezeichneten Skizzen in passende, fotorealistische Bilder zu trainieren. Das Erstellen der 3D-Modelle sowie die Automatisierung ist ebenfalls in Kapitel 4 beschrieben.

Eingabedaten für das Training der Image-To-Image-Translation sind also zum einen handgezeichnete Skizzen und zum anderen fotorealistische Abbildungen alltäglicher Gegenstände. In der Literatur wird häufig das Bild, das übersetzt werden soll, als Input bezeichnet. Das Bild, in das übersetzt werden soll, wird Ground Truth oder Target genannt [3]. Ich verwende in diesem Dokument dieselben Bezeichnungen oder die deutschen Übersetzungen Eingabebild für Input und Zielbild für Target.

Im Verlauf der Experimente habe ich den Trainingsfortschritt und -erfolg verschiedener Eingabebilder und Algorithmen betrachtet. Die interessantesten Größen beim maschinellen Lernen sind die Menge der Eingabedaten, die Anzahl der Trainingsdurchläufe und im Zusammenhang damit die Dauer des Trainings. Das tatsächliche Trainingsergebnis habe ich in Augenschein genommen und subjektiv für zufriedenstellend oder nicht zufriedenstellend befunden.



Abb. 1.1.: Das Modell generiert das Ergebnis in mehreren Tausend Schritten aus zufälligem Rauschen.

2. Bisherige Arbeiten

Künstliche neuronale Netze finden erst seit wenigen Jahren breite Aufmerksamkeit, seit auch Heimcomputer in der Lage sind die hohe Anzahl der erforderlichen Rechenoperationen in annehmbarer Zeit auszuführen. Seitdem sind wenige, englischsprachige Einführungen in die Thematik entstanden. Ein häufig genanntes Buch ist "Deep Learning", das online kostenfrei zugänglich ist [4]. Ebenfalls online und kostenfrei ist das Buch "Dive into Deep Learning" [5]. Ein weiteres, praxisorientierteres Buch ist "Deep Learning with Python" [3], das inzwischen in der zweiten Auflage erschienen ist.

1. Einleitung

Erste Recherchen haben verschiedene Arten und Implementierungen Bilder generierender künstlicher neuronaler Netze herausgestellt. Das Model aus “DRAW: A Recurrent Neural Network For Image Generation” [6] kann darauf trainiert werden, handschriftliche Ziffern wie die des MNIST-Datasets [7] zu generieren.

Die Schlüsselerkenntnis in “A Neural Algorithm of Artistic Style” [8] ist, dass Inhalt und Stil eines Bildes voneinander getrennt werden können. Das Model kann zum Beispiel den Stil eines Künstlers auf das Bild eines anderen übertragen.

In anderen Ansätzen mit rekurrenten neuronalen Netzen wird jedes Eingabebild in eine Sequenz von Pixeln umgeformt. Anschließend wird das künstliche neuronale Netz darauf trainiert, teilweise geschwärzte Bilder wieder zu vervollständigen [9], [10]. Es werden auch Bilder generiert und die Fähigkeit des jeweiligen Modells, die Darstellung von Bildern zu lernen, untersucht und mit der anderer Modelle verglichen. Ein weiterer, ebenfalls rekurrenter Algorithmus generiert Bilder aus textuellen Beschreibungen [11].

Besondere Aufmerksamkeit habe ich den Generative Adversarial Networks [12] gegeben. Die Ergebnisse dieser Familie von künstlichen neuronalen Netzen sind teilweise erstaunlich gut, und es gibt bereits einige Beispieldaten und frei verfügbaren Quelltext. Je nach Anwendung des Models variiert die Qualität der Resultate.

In “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks” [13] werden ebenfalls realitätsnahe Ergebnisse erzielt. Anders als bei den bisher erwähnten Arbeiten wird Unsupervised Learning verwendet, um das künstliche neuronale Netz zu trainieren.

Die vielseitigste Generierung von Bildern bietet Image-To-Image-Translation with Conditional Adversarial Networks [1]. In der Arbeit werden Labels in Fotos zurückübersetzt, Schwarzweißbilder koloriert und weitere Anwendungsbeispiele gezeigt. Vor allem die Übersetzung “Kanten nach Foto” (“Edges to Photo”) ist für mein Vorhaben relevant. Das Dokument enthält dafür mehrere Beispiele. Der Algorithmus setzt auf Generative Adversarial Networks [12] und Conditional Generative Adversarial Networks [14] auf.

Eine für die Vorbereitung meiner Experimente erforderliche Software ist Blender. Die Software ist kostenlos verfügbar und ermöglicht die Modellierung von 3D-Szenen. Objekte können aus vorgegebenen Würfeln, Zylindern und vielen weiteren Körpern und auch Flächen erstellt, skaliert und mit verschiedenen Werkzeugen verformt werden. Es stehen Lichtquellen zur Verfügung, und eine Szene kann als Bilddatei gespeichert werden. Zur Verwendung der Software gibt es eine große Auswahl an Literatur.

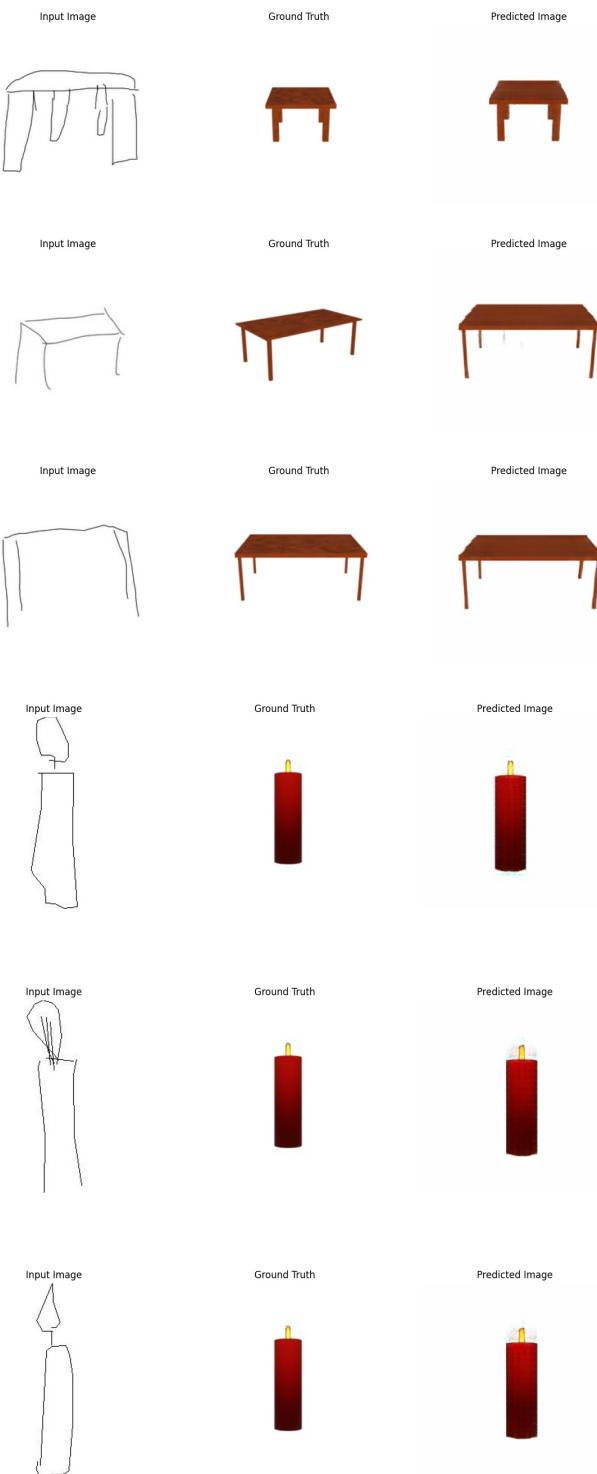


Abb. 1.2.: Meine Ergebnisse zeigen, dass aus einer Handzeichnung ein fotorealistisches Bild generiert werden kann. Die generierten Bilder nehmen ungefähr die Proportionen der Handzeichnungen an, erstellen realistische Darstellungen der Gegengenstände und fügen Texturen und Lichteffekte hinzu.

2. Modell

1. Tensoren

In der Literatur zur Funktionsweise künstlicher neuronaler Netze wird ausnahmlos auch der Begriff Tensor verwendet. Ein Tensor ist ein vielgestaltiges mathematisches Konstrukt. Beispielsweise sind Vektoren und Matrizen Tensoren. In “Deep Learning with Python” [3] findet sich die folgende Beschreibung:

Mehrdimensionale Arrays, in NumPy ndarray (N-Dimensional Array) genannt, werden auch Tensoren genannt. Generell verwenden alle derzeitigen Systeme für Maschinelles Lernen Tensoren als zugrundeliegende Datenstruktur. Tensoren sind elementar für dieses Fachgebiet.

In seinem Kern ist ein Tensor ein Container für numerische Daten. Tensoren sind die Verallgemeinerung der Matrizen für eine beliebige Anzahl Dimensionen. Im Zusammenhang mit Tensoren werden die Dimensionen häufig Achsen genannt. Die Anzahl der Achsen wird auch als Stufe bezeichnet. Ein Skalar ist ein Tensor nullter Stufe. Ein Vektor ist ein Tensor erster Stufe und eine Matrix ein Tensor zweiter Stufe.

Ein Tensor ist durch drei Schlüsselattribute definiert:

- Anzahl der Achsen (Stufe): Zum Beispiel hat ein Tensor dritter Stufe drei Achsen, und eine Matrix hat zwei Achsen. In Python-Bibliotheken wie NumPy oder Tensorflow wird sie auch als *ndim* des Tensors bezeichnet.
- Form: Dies ist ein Tupel aus Ganzzahlen, das die Dimension jeder Achse beschreibt. Beispiele sind (3, 5) für die Form einer Matrix und (3, 3, 5) für einen Tensor dritter Stufe. Ein Vektor hat eine Form mit einem einzelnen Element, etwa so: (5,), während ein Skalar eine leere Form hat: () .
- Datentyp (in Python-Bibliotheken gewöhnlich *dtype* genannt): Dies ist der Typ der im Tensor enthaltenen Daten. Zum Beispiel könnte der Datentyp eines Tensors *float16*, *float32*, *float64*, *uint8* und so weiter sein.

[3]

Bilder erscheinen als Tensoren dritter Stufe, deren Achsen der Höhe, Breite und den Farbkanälen (Rot, Grün und Blau) entsprechen. [5]

Viele in dieser Arbeit verwendeten Tensoren beschreiben RGB-Bilddaten. Sie sind deshalb dritter Stufe, und die enthaltenen Daten sind Ganzzahlen zwischen 0 und 255. Die Form der durch das künstliche neuronale Netz generierten Bilder ist (256, 256, 3). Die Dimensionen beschreiben hier also die Breite, Höhe und Farbtiefe der Bilddateien.

2. Kreuzentropie

Kreuzentropie ist ein Begriff aus der Wahrscheinlichkeitsrechnung. In dem Buch Deep Learning von Ian Goodfellow et al. [4] wird auf Wahrscheinlichkeitsrechnung wie folgt eingegangen:

In Anwendungen mit künstlicher Intelligenz wird Wahrscheinlichkeitsrechnung auf zwei wesentliche Arten verwendet. Erstens erklären die Regeln der Wahrscheinlichkeit, wie KI-Systeme urteilen sollten, deshalb werden Algorithmen so gestaltet, dass sie vielfältige Ausdrücke mittels Probabilistik berechnen oder annähern. Zweitens können Wahrscheinlichkeitsrechnung und Statistik das Verhalten zu planender KI-Systeme theoretisch analysieren.

Es gibt drei mögliche Quellen von Unsicherheit:

1. Die dem modellierten System innewohnende Zufälligkeit. Zum Beispiel beschreiben die meisten Interpretationen der Quantenmechanik die Dynamik der subatomaren Partikel als probabilistisch. Es können auch theoretische Szenarios mit zufälligen Dynamiken gebildet werden, wie ein hypothetisches Kartenspiel, in dem die Karten wirklich in einer zufälligen Reihenfolge gemischt werden.
2. Unvollständige Beobachtbarkeit. Auch deterministische Systeme können stochastisch anmuten, wenn nicht alle Variablen, die das Verhalten des Systems bestimmen, beobachtbar sind. Zum Beispiel, im Monty-Hall-Problem (Ziegenproblem), wird ein Kandidat einer Spielshow aufgefordert, zwischen drei Toren zu wählen und gewinnt einen Preis, der sich hinter der gewählten Tür befindet. Zwei Tore führen zu einer Ziege, während ein drittes zu einem Auto führt. Das Ergebnis aufgrund der Wahl des Kandidaten ist deterministisch, aber aus Sicht des Kandidaten ist das Ergebnis unsicher.
3. Unvollständige Modellierung. Wenn ein Modell einen Teil der beobachteten Informationen verwerfen muss, resultieren die verworfenen Informationen in Unsicherheit bei den Vorhersagen des Modells. Nehmen wir zum Beispiel an, ein Roboter kann die genaue Position jedes Objektes um ihn herum feststellen. Wenn der Roboter den Raum zum Vorhersagen der zukünftigen Position dieser Objekte diskretisiert, dann lässt diese Diskretisierung den Roboter über die präzise Position von Objekten unmittelbar unsicher werden: Jedes Objekt könnte irgendwo innerhalb der diskreten Zelle sein, in der es festgellt wurde.

In vielen Fällen ist es praktikabler, eine einfache aber unsichere Regel statt einer komplexen aber sichereren zu verwenden, selbst wenn die wahre Regel deterministisch ist und ein Modellierungssystem die Genauigkeit besitzt, eine komplexe Regel bereitstellen zu können.

Zum Beispiel ist die einfache Regel “Die meisten Vögel fliegen” einfach zu entwickeln und weitgehend nützlich, während eine Regel in der Form “Vögel fliegen, außer sehr junge Vögel, die noch nicht gelernt haben zu fliegen, kranke und verletzte Vögel, die ihre Fähigkeit zu fliegen verloren haben, flugunfähige Vögel, einschließlich Kasuare, Strauße und Kiwis...” ist aufwändig zu entwickeln, zu warten und zu kommunizieren und nach all dem Aufwand immer noch fragil und fehleranfällig.

Obwohl es naheliegend ist, dass Möglichkeiten Unsicherheit darzustellen und zu diskutieren notwendig sind, ist es nicht sofort offensichtlich, dass die Wahrscheinlichkeitsrechnung alle Werkzeuge für Anwendungen künstlicher Intelligenz bereitstellt.

Wahrscheinlichkeitsrechnung wurde ursprünglich entwickelt, um die Häufigkeit von Ereignissen zu analysieren. Es ist leicht erkennbar, wie Wahrscheinlichkeitsrechnung verwendet werden kann, um Ereignisse wie das Ziehen einer bestimmten Hand in einem Pokerspiel zu studieren. Diese Art von Ereignissen sind oft wiederkehrend.

Wenn ein Ergebnis die Wahrscheinlichkeit p hat einzutreten, heißt das, dass wenn das Experiment (z.B. eine Hand Karten ziehen) unendlich oft wiederholt würde, dann hätten die Wiederholungen zu dem Anteil p dieses Ergebnis zur Folge.

Dieser Denkansatz scheint auf Vorhaben, die nicht wiederholbar sind, nicht anwendbar zu sein. Wenn ein Arzt seinen Patienten analysiert und feststellt, dass der Patient eine 40-prozentige Chance besitzt die Grippe zu haben, heißt das etwas sehr unterschiedliches – der Patient kann nicht unendlich oft repliziert werden, und es gibt keinen Grund zur Annahme, dass unterschiedliche Replikas des Patienten die gleichen Symptome zeigen, aber unterschiedliche zugrundeliegende Krankheiten haben.

In dem Fall des Arztes, der den Patient diagnostiziert, steht Wahrscheinlichkeit für den Grad der Vermutung, wobei mit 1 absolute Sicherheit, dass der Patient die Grippe hat, angegeben wird und mit 0 absolute Sicherheit, dass der Patient die Grippe nicht hat.

Die vorherige Art von Wahrscheinlichkeit, direkt verbunden mit der Häufigkeit, in der Ereignisse eintreten, ist als frequentistische Wahrscheinlichkeit bekannt, während die zweite Art, verbunden mit dem qualitativen Grad der Sicherheit, bekannt ist als Bayessche Wahrscheinlichkeit. [4]

Das Buch Dive into Deep Learning von Aston Zhang et al. [5] nähert sich der Kreuzentropie wie folgt:

Um die Ausgaben als Wahrscheinlichkeiten interpretieren zu können, muss (auch für neue Daten) garantiert sein, dass sie nicht-negativ sind und sich zu 1 addieren. Weiterhin brauchen wir ein Trainingsziel, das das Modell darin unterstützt genaue Wahrscheinlichkeiten

zu schätzen. In allen Fällen sollten, wenn ein Classifier 0.5 ausgibt, die Hälfte dieser Beispiele tatsächlich zu der vorherberechneten Klasse gehören.

Es wird eine Verlustfunktion benötigt, um die Qualität der vorherberechneten Wahrscheinlichkeiten zu messen. Die Softmax-Funktion gibt einen Vektor \hat{y} aus, der als die geschätzte bedingte Wahrscheinlichkeit jeder Klasse für die gegebene Eingabe x interpretiert werden kann, zum Beispiel $\hat{y}_1 = P(y = \text{cat} | x)$.

Angenommen, das gesamte Dataset $\{X, Y\}$ enthält n Beispiele, wobei das Beispiel mit dem Index i aus einem Featurevektor $x^{(i)}$ und einem One-Hot-Label-Vektor $y^{(i)}$ besteht. Die Schätzungen können mit der Wirklichkeit verglichen werden, indem überprüft wird, wie wahrscheinlich die tatsächliche Klasse für die gegebenen Features gemäß dem Modell ist:

$$P(Y | X) = \prod_{i=1}^n P(y^{(i)} | x^{(i)}). \quad (2.1)$$

Entsprechend der Maximum-Likelihood-Methode wird $P(Y | X)$ maximiert, was equivalent zur Minimierung des negativen Log-Likelihood ist:

$$-\log P(Y | X) = \sum_{i=1}^n -\log P(y^{(i)} | x^{(i)}) = \sum_{i=1}^n l(y^{(i)}, \hat{y}^{(i)}), \quad (2.2)$$

wobei für jedes Paar von Label y und Vorhersage des Modells \hat{y} über q Klassen die Verlustfunktion l

$$l(y, \hat{y}) = - \sum_{j=1}^q y_j \log \hat{y}_j \quad (2.3)$$

ist. Die Verlustfunktion in (2.3) wird gemeinhin als Cross-Entropy-Verlustfunktion bezeichnet. Da y ein One-Hot-Vektor der Länge q ist, verschwindet die Summe über alle seine Koordinaten j , bis auf einen Term. Da alle \hat{y}_j vorherberechnete Wahrscheinlichkeiten sind, ist deren Logarithmus niemals größer als 0. Folglich kann die Verlustfunktion nicht weiter minimiert werden, wenn das tatsächliche Label mit *Sicherheit*, das heißt wenn die vorhergesagte Wahrscheinlichkeit $P(y | x) = 1$, korrekt vorhergesagt wird.

Nehmen wir nun an, dass wir nicht nur ein einziges Ergebnis, sondern eine ganze Verteilung über Ergebnisse beobachten. Wir können dieselbe Darstellung nutzen wie bisher. Der einzige Unterschied ist, dass statt einem Vektor nur aus binären Einträgen wie zum Beispiel $(0, 0, 1)$ jetzt ein allgemeiner Wahrscheinlichkeitsvektor wie zum Beispiel $(0.1, 0.2, 0.7)$ vorliegt. Die

bisher verwendete Mathematik um den Verlust zu definieren funktioniert noch genauso gut, nur ist die Interpretation etwas allgemeiner. Es ist der Erwartungswert des Verlustes für eine Verteilung über Labels. Dieser Verlust wird Cross-Entropy-Verlust genannt und es eine der meistverbreitet eingesetzten Verlustfunktionen für Klassifizierungsaufgaben.

Hier ist ein kurzes Beispiel für die Anwendung der Kreuzentropie. Es sei P ein wahre Verteilung mit der Wahrscheinlichkeitsverteilung $p(x)$ und Q die geschätzte Verteilung mit der Wahrscheinlichkeitsverteilung $q(x)$. Angenommen eine Klassifizierungsaufgabe basierend auf n gegebenen Datenbeispielen $\{x_1, \dots, x_n\}$. Angenommen die positiven und negativen Klassenlabel y_i sind als 1 beziehungsweise 0 kodiert und ein künstliches neuronales Netz ist parametrisiert durch θ . Wenn es das Ziel ist, das beste θ zu finden, damit $\hat{y}_i = p_\theta(y_i | x_i)$, kann naturgemäß die Maximum-Likelihood-Methode angewendet werden. Konkret ist für die wahren Label y_i und die Vorhersagen $\hat{y}_i = p_\theta(y_i | x_i)$ die Wahrscheinlichkeit als positiv klassifiziert zu werden $\pi_i = p_\theta(y_i = 1 | x_i)$.

Daher wäre die Log-Likelihood-Funktion

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^n \pi_i^{y_i} (1 - \pi_i)^{1-y_i} \\ &= \sum_{i=1}^n y_i \log(\pi_i) + (1 - y_i) \log(1 - \pi_i). \end{aligned} \tag{2.4}$$

Die Log-Likelihood-Funktion zu maximieren ist identisch mit dem Minimieren von $-l(\theta)$, und daher kann hiermit der beste Wert für θ gefunden werden. [5]

3. Deep Neural Networks

Die nachfolgende Beschreibung zu Deep Neural Networks ist in dem Buch “Deep Learning with Python” [3] zu finden:

Deep Learning ist ein spezifisches Gebiet des maschinellen Lernens: Ein neuer Ansatz zum Lernen von Repräsentationen von Daten, der den Schwerpunkt auf das Lernen in aufeinanderfolgenden Schichten mit zunehmend aussagekräftigen Repräsentationen legt. Das “Deep” in “Deep Learning” ist kein Hinweis auf ein durch diesen Ansatz erlangtes tieferes Verständnis, sondern steht für diese Idee von aufeinanderfolgenden Schichten von Repräsentationen.

Die Anzahl der Schichten, aus denen ein Datenmodell zusammengefasst ist, wird Tiefe (engl.: depth) des Modells genannt. Andere passende Namen für dieses Feld hätten “layered representations learning” oder “hierarchical representations learning” sein können. Modernes

2. Modell

Deep Learning umfasst oft mehrere zehn- oder hunderttausend aufeinanderfolgendende Repräsentationsschichten, die alle automatisch durch Verrechnung von Trainingsdaten erlernt werden.

Währenddessen gehen andere Ansätze des maschinellen Lernens eher in die Richtung des “shallow learning”, beschränken sich also auf eine oder zwei Repräsentationsschichten. Diese Schichten von Repräsentationen werden fast immer durch Modelle erlernt, die neuronale Netze genannt werden. Diese Modelle sind buchstäblich wie aufeinander gestapelte Schichten strukturiert.

Der Begriff neuronales Netz ist ein Referenz auf Neurobiologie, doch obwohl die Entwicklung einiger zentraler Deep-Learning-Konzepte teilweise durch unser Verständnis des Gehirns inspiriert war, sind Deep-Learning-Modelle keine Modelle des Gehirns. Es gibt keine Anhaltspunkte darauf, dass das Gehirn etwas vergleichbares wie die Lernmechanismen implementiert, die in modernen Deep-Learning-Modellen verwendet werden. Gelegentlich begegnen einem populärwissenschaftliche Artikel in denen proklamiert wird, dass Deep Learning wie das Gehirn funktioniert oder nach dem Vorbild eines Gehirns modelliert wurde. Das ist aber nicht der Fall. Für Neueinsteiger in das Fachgebiet wäre es verwirrend und kontraproduktiv, Deep Learning als etwas zu betrachten, das mit Neurobiologie in irgend einem Zusammenhang steht. Deep Learning ist ein mathematisches Gerüst zum Lernen von Repräsentationen von Daten.

Ein Deep-Learning-Netzwerk ist wie ein mehrstufiger Informations-Destilliations-Vorgang, bei dem Information durch aufeinanderfolgende Filter fließt und zunehmend bereinigt und für einen bestimmten Zweck nutzbar wieder herauskommt. Die Idee hinter Deep Learning ist einfach, kann aber wie Magie anmuten, wenn der simple Mechanismus ausreichend skaliert wird. [3]

4. Convolutional Neural Networks

Für Bilddaten verarbeitet ein Convolutional Neural Network, kurz CNN, typischerweise Paare von Tensoren dritter Stufe, nämlich einerseits auf die Eingabedaten und andererseits auf einen sogenannten Filter oder auch Kernel. Die Eingabedaten sind in der ersten Schicht des Netzes die RGB-Pixelinformationen und in allen weiteren konvolutionalen Schichten die Ausgabe der vorherigen Schicht. Ein Filter ist eine Anzahl von trainierbaren Parametern, enthalten in einem Tensor.

Die Form der Tensoren variiert üblicherweise zwischen den verschiedenen Schichten des Netzes. In fast allen CNNs ([1], [4], [15], [16]) nimmt die Dimensionalität zunächst ab. Die Reduktion kann durch die Konvolution selbst entstehen oder durch Pooling-Schichten.

Beim Pooling werden aus benachbarten Matrixkoeffizienten meist das Maximum, seltener der Durchschnitt oder andere Aggregierungen gebildet. Auf diese Weise wird das neuronale Netz darauf trainiert die relevanten Informationen zu extrahieren. [4]

Den konvolutionalen Schichten folgt in einigen Anwendungsfällen eine voll vernetzte Schicht (engl. Fully Connected Layer, FC), in der für jede Einheit mit jeder Einheit der vorherigen Schicht eine Verbindung besteht. Besonders für die Bilderkennung wird diese Architektur gelegentlich erfolgreich eingesetzt. Die Ausgabe des neuronalen Netzes ist dann ein Vektor, beispielsweise von Wahrscheinlichkeitswerten für das Vorhandensein bestimmter Objekte und gegebenenfalls Bildkoordinaten der erkannten Objekte. Im Fall der Bildgenerierung kann die Ausgabe aber auch wieder ein Tensor mit RGB-Pixelinformationen in der Form ($H \times B \times 3$) sein.

5. Batch Normalization

In “Deep Learning with Python” [3] ist Batch Normalization wie folgt beschrieben:

Normalisierung ist eine breit gefächerte Kategorie von Methoden, die versuchen verschiedene Eingaben für Modelle des maschinellen Lernens einander ähnlicher aussehen zu lassen. Das soll dem Modell helfen zu lernen und für neue Daten zu verallgemeinern. Die verbreitetste Form von Normalisierung besteht darin die Daten an 0 zu zentrieren und mit einer Standardabweichung von < 1 zu versehen, indem der Mittelwert aller Werte von jedem Einzelwert subtrahiert wird und jeder Einzelwert durch die Standardabweichung geteilt wird. Dies geschieht aufgrund der Annahme, dass die Daten normalverteilt sind und stellt sicher, dass diese Verteilung zu einer einheitlichen Varianz zentriert und skaliert ist.

Datennormalisierung sollte nach jedem Transformationsschritt durch das künstliche neuronale Netz geschehen. Auch wenn die Daten, die in ein Dense Net oder Convolutional Neural Net eingegeben werden, den Mittelwert 0 und eine Standardabweichung < 1 besitzen, ist ohne weiteres nicht zu erwarten, dass es sich bei den Ausgabedaten genauso verhält.

Batch Normalization ist eine Art von Schicht, die 2015 von Ioffe und Szegedy [17] eingeführt wurde. Sie kann adaptiv die Daten normalisieren, wenn sich Mittelwert und Varianz im Laufe des Trainings verändern. Das erreicht sie, indem sie einen exponentiellen gleitenden Durchschnitt der batchweisen Mittelwerte und Varianzen der Daten während des Trainings vorhält.

Hauptsächlich hilft die Batch Normalization bei der Backpropagation und erlaubt so tiefere Netzwerke. Einige sehr tiefe Netwerke können überhaupt nur trainiert werden, wenn sie mehrere Batch-Normalization-Schichten enthalten. Batch Normalization wird zum Beispiel in vielen fortgeschrittenen ConvNet-Architekturen verwendet, die Keras zur Verfügung stellt, wie ResNet50, Inception v3 und Xception. Eine Batch-Normalisierungs-Schicht wird typischerweise nach einem convolutional oder einer Densely-Connected-Schicht verwendet. [3]

6. U-Net-Architektur

Die bis hierhin beschriebenen neuronalen Netze besitzen eine gradlinige Struktur, in der die Ausgabe einer Schicht nur an die nächste Schicht übergeben wird. Bei zunehmender Anzahl der Schichten verbessert sich die Performance neuronaler Netze mit diesem Aufbau zunächst, aber verschlechtert sich bei zu vielen Schichten wieder. In einem Residual Neural Network (ResNet) verhindern zusätzliche Verbindungen zwischen nicht direkt aufeinanderfolgenden Schichten diesen Performanceverlust.

Ein U-Net ist eine spezielle Form eines ResNets. Es hat eine annähernd symmetrische Struktur, in der sich die Tensoren zunächst zunehmend verformen und anschließend wieder annähernd ihre ursprüngliche Form annehmen.

U-Nets erzielen selbst mit wenigen Trainingsdaten gute Ergebnisse und benötigen dafür vergleichsweise wenig Rechenleistung [18].

7. Generative Adversarial Networks

Ein Generative Adversarial Network (GAN) besteht zunächst aus einem Generator und einem Discriminator. Der Generator lernt während des Trainings täuschend echt aussehende Bilddaten zu generieren. Der Discriminator wird dagegen darauf trainiert, echte von generierten Bildern zu unterscheiden. Anschließend können beide Modelle “gegeneinander antreten”. Deswegen wird es Generative *Adversarial* Network genannt.

Generative Adversarial Networks basieren auf einem spieltheoretischen Szenario, in dem das Generator-Netzwerk gegen einen Kontrahenten antreten muss. Das Generator-Netzwerk produziert Proben $\mathbf{x} = g(\mathbf{z}; \theta^{(g)})$. Sein Kontrahent, das Discriminator-Netzwerk, versucht zwischen den Proben aus den Trainingsdaten und den vom Generator erstellten Proben zu unterscheiden. Der Discriminator gibt einen Wahrscheinlichkeitswert aus, der durch $d(\mathbf{x}; \theta^{(d)})$ gegeben ist und der die Wahrscheinlichkeit angibt, dass \mathbf{x} ein echtes Trainingsexemplar ist und keine Täuschung, die durch das Modell erstellt wurde. [4]

GANs unterscheiden sich von anderen Modellen durch ihren Aufbau und in Bezug auf das Trainingsziel. Künstliche neuronale Netze ermitteln häufig einen skalaren Wert wie beispielsweise einen Wahrscheinlichkeitswert und minimieren zu diesem Zweck eine Verlustfunktion. In einem GAN sind zwei CNNs im Einsatz. Das erste, der Generator, erstellt Tensoren n-ter Klasse. In diesem Beispiel sind das RGB-Bildinformationen. Das zweite CNN wird Discriminator genannt und bekommt als Eingabe die Ausgabe des ersten CNNs. Der Discriminator wird darauf trainiert, generierte Bilder von Bildern aus dem Trainingsset zu unterscheiden. Er minimiert also eine Verlustfunktion. Der Generator wird darauf trainiert, diese Verlustfunktion zu maximieren. Dieser Vorgang heißt auch Min-Max-Spiel. [12]

8. Conditional Generative Adversarial Nets

Mehdi Mirza beschreibt in seinem Artikel “Conditional Generative Adversarial Nets” [14] eine Klasse von künstlichen neuronalen Netzen, die gegenüber Generative Adversarial Networks noch weitere Vorteile bieten:

Eine Erweiterung des GANs ist das Conditional Generative Adversarial Net (cGAN). In cGANs erhält der Generator zusätzliche Eingabedaten (y , “Ground Truth”), die Hinweise für die Generierung enthalten. Der Discriminator erhält diese zusätzlichen Daten ebenfalls, um die Erkennung während des Trainings zu optimieren.

Dieses Modell kann Ziffern des MNIST-Datasets anhand von Klassenlabels generieren. Es kann auch verwendet werden, um multimodales Bild-Tagging durchzuführen. Dabei kann ein Bild beispielsweise nicht nur durch ein darauf erkennbares Objekt, sondern auch durch Aufzählung mehrerer unterschiedlicher Objekte und noch durch weitere Beschreibungen, wie erkennbare Jahreszeiten, Vorgänge oder Emotionen beschrieben werden.

In einem generativen Modell ohne Konditionen ist keine Steuerung der Art und Weise (Modalitäten) möglich, in der die Ergebnisse generiert werden. Dagegen ist es in einem cGAN möglich, die Generierung zu lenken. Diese Lenkung könnte aufgrund von Klassenlabels oder auch verschiedensten anderen Modalitäten erfolgen.

Um die Verteilung p_g eines Generators über die Daten x zu lernen, implementiert der Generator $G(z; \theta_g)$ eine Mapping-Funktion von anfänglichem Rauschen $p_z(z)$ zum Ergebnisraum. Der Discriminator $D(x; \theta_d)$ gibt einen einzelnen skalaren Wahrscheinlichkeitswert dafür, dass x aus den Trainingsdaten statt aus p_g stammt, aus.

G und D werden simultan trainiert. Die Parameter für G werden angepasst, um $\log(-D(G(z)))$ zu minimieren und die Parameter für D werden angepasst, um $\log D(x)$ zu minimieren, so als würden beide ein Zweispieler-Min-Max-Spiel mit der Wertfunktion durchführen:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))] \quad (2.5)$$

Im Generator werden das anfängliche Rauschen $p_z(z)$ und y kombiniert in verbundenen Hidden Layers repräsentiert. Das gegnerische Trainingsframework erlaubt eine flexible Zusammensetzung dieser Repräsentation.

Im Discriminator werden paarweise x und y sowie x und $G(z|y)$ als Eingaben verwendet. Die Funktion, die das Ziel dieses Zweispieler-Min-Max-Spiels beschreibt ist dann:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x|y)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z|y)))] \quad (2.6)$$

[14]

3. Implementierung

1. Github

Die in dieser Arbeit erstellten und verwendeten Implementierungen und Dateien sind im Github-Repository `stefan-bmio/ba`¹ abgelegt.

Das wichtigste Verzeichnis heißt `PIX2PIX_keras`. Es enthält die Implementierung der Image-To-Image-Translation, die ich für meine Experimente genutzt habe. Sie wird im Abschnitt 3.5 beschrieben. Das Verzeichnis `PIX2PIX_keras/results` enthält weitere Unterverzeichnisse mit den Zwischen- und Endergebnissen der Trainings.

In den Unterverzeichnissen von `training-images` befinden sich die Trainingsdaten aller durchgeföhrten Experimente. Meistens handelt es sich dabei um Paare einer Skizze und eines gerenderten Blender-Modells. Es befinden sich darunter aber auch Ergebnisse einer Kantenerkennung, mit denen anfangs ebenfalls experimentiert wurde. Die Handskizzen waren am Ende jedoch die praxisnäheren und deshalb sinnvolleren Trainingseingaben.

Im Verzeichnis `PIX2PIX` befindet sich eine weitere Implementierung der Image-To-Image-Translation. Sie ist dem Github-Repository `affinelayer/pix2pix-tensorflow`² entnommen. Die Ergebnisse dieser Implementierung, zu finden unter `PIX2PIX/result/images`, waren zu Beginn der Arbeit überzeugend. Deshalb wurde der Ansatz auch weiter verfolgt. Aufgrund schwächerer Performance in Bezug auf den Arbeitsspeicher war es aber vorteilhafter, die andere Implementierung zu verwenden.

Das Repository enthält weiterhin drei NDJSON-Dateien des Google Quick-Draw-Datasets. In Abschnitt 4.1 wird auf dieses Dataset genauer eingegangen. Aus diesen Dateien wie auch aus denen des Verzeichnisses `blender` wurden die Trainingsdaten für das künstliche neuronale Netz erstellt. Blender wird in Abschnitt 4.2 vorgestellt.

Die zwei Verzeichnisse `HED` und `NST` enthalten Implementierungen der Holistically-Nested Edge Detection[19] beziehungsweise Neural Style Transfer [8]. Beide generieren auf ihre eigene Weise Bilder mittels künstlicher neuronaler Netze, konnten in Bezug auf die Zielsetzung dieser Arbeit aber schließlich nicht überzeugen. Das Verzeichnis `canny` beinhaltet die Implementierung der Canny Edge Detection [2].

1 <https://github.com/stefan-bmio/ba>

2 <https://github.com/affinelayer/pix2pix-tensorflow/blob/master/pix2pix.py>

In dem Verzeichnis `video` befindet sich ein Arbeitsbereich zur Erstellung zweier Videos, in denen ein Teil meiner Arbeit veranschaulichend dokumentiert werden sollte. Nur das eine der beiden Videos mit dem Dateinamen `video2_720p.mp4` ist fertiggestellt. Der Fokus der Arbeit lag natürlich auf den praktischen Ergebnissen und dem schriftlichen Teil.

Im Verzeichnis `bib` befinden sich einige Downloads von wissenschaftlichen Artikeln und Büchern im PDF-Format. Das Verzeichnis `pdf` beinhaltet die Dateien, mit denen der schriftliche Teil, also dieses Dokument, erstellt wurde.

2. Ubuntu Linux

Ubuntu ist eine der bekanntesten Linux-Distributionen. Es ist kostenlos und in verschiedenen Editionen für unterschiedliche Zwecke erhältlich. Es kann beispielsweise für Server oder für Desktop-Computer verwendet werden.

Die Wahl des Betriebssystems fiel für dieses Projekt auf Ubuntu, weil die Integration mit der NVIDIA-Grafikkartenerweiterung CUDA mit Abstand am besten dokumentiert ist. Es gibt sowohl für Ubuntu als auch für CUDA eine lebendige Community. Dadurch besteht bei eventuell auftretenden Schwierigkeiten stets die Möglichkeit, auf die Erfahrungen anderer Nutzer zurückzugreifen.

Die Installation des Betriebssystems ist vergleichsweise einfach. Ein Volume-Image im ISO-Dateiformat (ISO 9660) kann vom Ubuntu-eigenen Server heruntergeladen und anschließend entweder auf eine CD oder DVD gebrannt oder auf einen bootfähigen USB-Stick kopiert werden.

Der Software-Assistent des Installationsprogramms hat eine leicht zu bedienende grafische Benutzeroberfläche. Es können alle wichtigen Einstellungen wie Eingabe- und Ausgabesprache, Uhrzeit und Sicherheitseinstellungen vorgenommen werden.

Es ist auch möglich, Ubuntu unter Windows innerhalb der WSL (Windows Subsystem for Linux) zu installieren. Auch dort besteht die Möglichkeit, die Hardwareunterstützung zu nutzen. Ubuntu unter WSL ist dann in der Lage, mittels CUDA die GPU für das Training neuronaler Netze zu verwenden.

Für diese Arbeit wurde Ubuntu als Hauptbetriebssystem genutzt. Die Schritte zur Einrichtung einer Entwicklungsumgebung für maschinelles Lernen sind auf [ubuntu.com³](https://ubuntu.com/tutorials/enabling-gpu-acceleration-on-ubuntu-on-wsl2-with-the-nvidia-cuda-platform/) beschrieben.

³ <https://ubuntu.com/tutorials/enabling-gpu-acceleration-on-ubuntu-on-wsl2-with-the-nvidia-cuda-platform/>

3. Python

Die Programmiersprache Python ist eine beliebte Wahl für Software-Projekte im Bereich des maschinellen Lernens. Ein wichtiger Grund dafür sind die verschiedenen verfügbaren Bibliotheken, die zum Beispiel Bausteine für künstliche neuronale Netze und sogar häufig Algorithmen bereitstellen. Zwei der berühmtesten Bibliotheken dieser Art sind TensorFlow und Keras. In dem O'Reilly-Buch "Python" von Mark Lutz [20] wird Python wie folgt beschrieben:

Python ist eine beliebte, quelloffene Programmiersprache sowohl für eigenständige Programme als auch für Anwendungs-Skripte in vielen verschiedenen Bereichen. Es ist kostenlos, portabel sowie relativ einfach und wird mit bemerkenswertem Enthusiasmus verwendet.

Version 3.0 war die erste in einer Reihe der aufstrebenden und nicht abwärtskompatiblen Abwandlung der Sprache die allgemein als 3.X bezeichnet wird. Version 2.6 behielt die Abwärtskompatibilität zu einer zahllosen Menge bestehenden Python-Quelltextes bei und war die letzte der gemeinhin als 2.X bekannten Reihe.

Für viele hebt sich Python durch seinen Fokus auf Lesbarkeit, Stimmigkeit und Softwarequalität insgesamt aus anderen Tools der Skripting-Welt hervor. Python-Quelltext ist gemacht, um lesbar und dadurch wiederverwendbar und wartbar zu sein – noch viel mehr als traditionelle Skriptsprachen. Die Gleichförmigkeit von Python-Quelltext macht ihn leicht verständlich, selbst wenn man ihn nicht selbst gegschrieben hat. Zusätzlich unterstützt Python fortgeschrittene Mechanismen für wiederverwendbare Software, wie objektorientierte und funktionale Programmierung.

Python erhöht die Entwicklerproduktivität um einiges mehr als kompilierte oder statisch typisierte Sprachen wie C, C++ und Java. Die Größe von Pythoncode ist typischerweise ein Drittel bis ein Fünftel von der entsprechenden C++- oder Java-Quelltext. Das heißt es gibt weniger zu tippen, weniger zu debuggen und schließlich weniger zu warten. Python-Programme sind auch unmittelbar ausführbar, ohne die langwierigen Kompilier- und Linksschritte, die für manche anderen Tools erforderlich sind, und erhöht die Programmierergeschwindigkeit somit noch weiter.

Die meisten Python-Programme laufen unverändert auf allen wichtigen Computerplattformen. Um Pythoncode zwischen Linux und Windows zu portieren, ist es zum Beispiel gewöhnlich nur erforderlich den Skriptcode von einer Maschine zur anderen zu kopieren. Überdies bietet Python multiple Optionen für die Erstellung portabler grafischer Benutzerschnittstellen, Datenbankanwendungen, webbasierter Systeme und anderem. Selbst Betriebssystemschnittstellen, einschließlich Programmausführungen und Verzeichnisverarbeitung sind so portable wie irgend möglich.

Python enthält eine große Sammlung vorgefertigter und portabler Funktionalitäten, bekannt als die Standardbibliothek. Diese Bibliothek unterstützt eine Reihe von Programmieraufgaben auf Anwendungsebene, von Musterkennung in Texten bis Netzwerkprogrammierung. Außerdem kann Python sowohl mit selbstentwickelten Bibliotheken als auch einer

unüberschaubaren Kollektion von Unterstützungssoftware dritter erweitert werden. Pythons Drittanbieter-Bereich bietet Tools zur Webseitenerstellung, numerische Programmierung, Zugriff auf serielle Ports, Spieleentwicklung und vieles mehr. Die Erweiterung NumPy wird zum Beispiel als kostenloses und mächtigeres Equivalent zum numerischen Programmiersystem Matlab bezeichnet.

Pythonskripte können leicht mit anderen Teilen einer Anwendung kommunizieren, indem sie eine Vielzahl integrierter Mechanismen nutzen. Solche Integrationen erlauben es, Python als Produktanpassungs- und Produkterweiterungstool zu verwenden. Pythoncode kann heute C- und C++-Bibliotheken aufrufen, von C- und C++-Programmen aufgerufen werden, in Java- und .NET-Komponenten integriert werden, mittels Frameworks wie COM und Silverlight kommunizieren, sich mit Geräten via serieller Schnittstelle verbinden und über Netzwerke mit Schnittstellen wie SOAP, XML-RPC und CORBA interagieren. Es ist kein alleinstehendes Tool. [20]

4. CUDA

Cuda ist eine NVIDIA-proprietäre Hardware- und Software-Architektur. Auch zu CUDA existiert ein Buch. Es heißt “CUDA by Example: An Introduction to General-Purpose GPU Programming” [21] und enthält die folgende Beschreibung:

Die CUDA-Architektur ist das Schema, nach dem NVIDIA-Grafikkarten gebaut wurden, die sowohl traditionelle Grafik-Rendering-Aufgaben als auch allgemeine Aufgaben durchführen können. Zum Programmieren der CUDA GPUs wird die Sprache CUDA C verwendet. CUDA C ist im Wesentlichen die Programmiersprache C mit einer Handvoll Erweiterungen, welche die Programmierung hoch parallelisierter Maschinen wie NVIDIA GPUs ermöglichen.

Anders als frühere GPU-Generationen, die Rechenressourcen in Vertex- und Pixelshader aufteilten, enthält die CUDA-Architektur eine einheitliche Shader-Pipeline, welche die Zuordnung allgemeiner Berechnungen zu jeder arithmetisch-logischen Einheit (ALU) auf dem Chip durch ein Programm erlaubt. Diese ALUs wurden mit einem Befehlssatz entworfen, der für allgemeine Berechnungen statt für spezielle Grafikberechnungen zugeschnitten ist. Weiterhin wurde den Execution Units auf der GPU freier Lese- und Schreibzugriff auf den Speicher sowie Zugriff auf einen softwaregesteuerten Cache, genannt Shared Memory, gegeben.

Zusätzlich zu der Sprache für das Schreiben von Code für die GPU stellt NVIDIA einen spezialisierten Hardwaretreiber zur Verfügung, der die hohe Rechenleistung der CUDA-Architektur ausschöpft. Kenntnis der OpenGL- oder DirectX-Programmierschnittstellen ist nicht länger erforderlich. [21]

5. Image-To-Image-Translation in Python

Der Image-To-Image-Translation-Algorithmus oder kurz Pix2Pix-Algorithmus verwendet ein GAN, um Bilder in Bilder zu übersetzen. Dafür kommen zwei CNNs zum Einsatz, nämlich eins für den Generator und eins für den Discriminator. [1]

Die Datei main.py befindet sich im Anhang 1. Es ist eine Beispielimplementierung zum Artikel “Image-to-Image Translation with Conditional Adversarial Networks” [1] und ist im Github-Repository tensorflow/docs⁴ zu finden. Für meine Experimente habe ich hauptsächlich diese Implementierung verwendet. Sie wurde für die erste Hauptversion von Tensorflow geschrieben und war zunächst mit meinem Setup nicht ausführbar. Deshalb folgen den Import-Anweisungen am Anfang des Python-Skripts weitere Imports und Konfigurationen, um die Kompatibilität herzustellen. Im Verlauf des Programmes wird dann die erste Tensorflow-Version genutzt.

Eine weitere Änderung ist das selbst erstellte Dataset, das statt der im Beispiel verwendeten Gebäudefassaden geladen wird. Es wurden auch Ausgaben entfernt, wegen derer die Ausführung des Skripes unterbrochen wurde. Dabei wurden Bildinformationen und Beispielbilder nach dem Laden des Datasets, dem Aufteilen in Eingabebild und Ground Truth, der Erzeugung von Bildrauschen, Down- und Upsampling sowie den ersten Trainingsdurchläufen ausgegeben, die während des Trainings der Skizzen- und Renderbilder nicht betrachtet werden mussten. Außerdem wurden in der Original-Implementierung bei jedem Training Diagramme des Generators und des Discriminators erzeugt.

Die erste Funktion namens `load` öffnet eine Datei und liest diese als JPEG-Bilddatei. Sie wird verwendet, um die Trainingsdaten zu laden. Weil die Trainingsdaten aus zwei Bildern pro Datei bestehen, nämlich jeweils einem Eingabebild und dem erwarteten Ergebnis (Ground Truth), extrahiert die Funktion zusätzlich die beiden Bilder und gibt jede in einer eigenen Variablen zurück.

Die Funktion `resize` ist ebenfalls für die Verarbeitung zweier Bilder vorgesehen und skaliert beide auf die übergebene Breite und Höhe.

Die Funktion `random_crop` stellt einen zufälligen Ausschnitt der zwei Eingabebilder frei. Sie wird aufgerufen, nachdem die Eingabebilder hochskaliert wurden. In der Beispielimplementierung wie auch in dieser Arbeit beträgt die Bildgröße vorher 286x286 Pixel und 256x256 Pixel nach dem Freistellen. Die zurückgegebenen Bilder haben dann wieder die gleichen Abmessungen wie die Eingabedaten.

In der Funktion `normalize` werden die RGB-Werte der beiden Eingabebilder normalisiert. Dadurch sollen zu große Schritte auf dem Gradienten verhindert werden, die das Konvergieren verhindern könnten. [3] Die RGB-Werte sind zunächst als Ganzzahlen im Bereich von 0 bis 255 gegeben und werden in Fließkommazahlen im Bereich -1 bis 1 umgerechnet.

⁴ <https://github.com/tensorflow/docs/blob/master/site/en/tutorials/generative/pix2pix.ipynb>

In `random_jitter` wird zunächst `resize` aufgerufen, um die Eingabebilder auf 286x286 Pixel zu skalieren. Anschließend wird `random_crop` auf die skalierten Bilder angewendet. Schließlich werden aufgrund einer Zufallszahl beide Eingabebilder mit einer Wahrscheinlichkeit von 50 Prozent gespiegelt.

Die beiden Funktionen `load_image_train` und `load_image_test` laden die Eingabe- und Zielbilder, skalieren diese auf 256x256 Pixel und normalisieren die Eingabedaten durch Aufrufe der Funktionen `load`, `resize` und `normalize`. Nur die Funktion zum Laden der Trainingsbilder ruft für die geladenen Bilder `random_jitter` auf.

Die Funktion `downsample` erzeugt eine Schicht eines CNNs mit optionaler Batch Normalization und der Leaky-ReLU-Aktivierungsfunktion. Diese Schichten werden im Encoder des Generators sowie im Discriminator verwendet. Batch Normalization kommt nur in der jeweils ersten Schicht des Generators und des Discriminators nicht zum Einsatz.

In der Funktion `upsample` entstehen die übrigen CNN-Schichten des Generators. Der Entfaltung, bei der die Faltung der Funktion `downsample` umgekehrt wird (engl. Transposed Convolution [22]), folgt hier immer die Batch Normalization sowie optional Dropout mit einem Wahrscheinlichkeitswert von 0.5. Das ist in den ersten drei Schichten des Decoders im Generator der Fall und bedeutet, dass statistisch die Hälfte der Aktivierungen „fallengelassen“ werden, also nicht in das Trainingsergebnis eingehen. Als Aktivierungsfunktion kommt ReLU zum Einsatz.

In `build_generator` werden die Schichten des Generators zusammengefügt. Im Encoder wird achtmal `downsample` aufgerufen, bis die Dimensionen des Eingabetensors, die am Anfang der Bildgröße entsprechen (Breite und Höhe, also 256x256 Pixel), durch die Konvolution auf 1 reduziert sind. Die Dimension, welche die Anzahl Farbkanäle der Eingabebilder repräsentiert, erhöht sich im Encoder auf den Wert 512. Die Werte dieser Achse werden als Features [5] [3] [22] bezeichnet.

Der Decoder ruft seinerseits siebenmal `upsample` auf und führt eine weitere Entfaltung durch, wodurch die Dimensionen des bearbeiteten Tensors wieder jeweils dieselbe Form wie in den Eingabebildern annehmen. Schließlich werden die Skip-Connections zwischen den Schichten 1 bis 8 des Encoders und des Decoders hergestellt.

In der Funktion `generator_loss` ist die Backpropagation des Generators implementiert. Zuerst wird dafür die Kreuzentropie des Ergebnisses des Discriminators und eines gleich großen Tensors, der mit Einsen initialisiert wird, gebildet. Das Ergebnis ist das GAN-Loss. Anschließend kommt die L1 Loss Function, auch Mean Absolute Error genannt, zum Einsatz. Dieses L1-Loss wird aus dem Durchschnitt der absoluten Differenz der Ground Truth und des Generatorergebnisses berechnet. Der Gesamtverlust des Generators ist das GAN-Loss plus das Produkt aus dem Regularisierungsparameter Lambda, der konstant 100 beträgt, und dem L1-Loss. Die Funktion `generator_loss` gibt den Gesamtverlust, das GAN-Loss und das L1-Loss zurück.

3. Implementierung

In `build_discriminator` wird der Discriminator aus verschiedenen neuronalen Schichten zusammengesetzt. Zwei Tensoren `input_image` und `target_image` mit denselben Anzahlen Achsen und denselben Dimensionen wie die Eingabebilder werden zu einem einzelnen Tensor konateniert. Anschließend wird für diesen Tensor dreimal `downsample` aufgerufen, bis der Tensor die Dimensionen (32x32x256) hat. Es folgt ein Padding mit Nullen, eine weitere Konvolution sowie Batch Normalization und die Leaky ReLU Aktivierungsfunktion. Nach einem weiteren Null-Padding und einer weiteren Konvolution hat der Tensor die Dimensionen (30x30x1).

In der Funktion `discriminator_loss` ist die Backpropagation des Discriminators implementiert. Aus jeweils einem Tensor mit den gleichen Dimensionen wie die Eingabebilder und mit Einsen initialisiert wird die Kreuzentropie mit einem Trainingsbild und dem Ergebnis des Generators gebildet. Die jeweiligen Werte für den Verlust des Discriminators werden als Gesamtverlust zurückgegeben.

Die Funktion `generate_images` verwendet das Model des Generators, um aus Eingabebildern eigene Bilder zu generieren. Sie erstellt anschließend eine Ausgabedatei mit einem Beispiel bestehend aus einem Eingabebild, der zugehörigen Ground Truth und dem daraus generierten Bild.

In der Funktion `train_step` wird ein Generator mit dem übergebenen Input-Bild und dem ebenfalls übergebenen Target-Bild initialisiert. Außerdem werden zwei Discriminator initialisiert, nämlich einmal mit dem Input- und dem Target-Bild und einmal mit dem Input-Bild und dem durch den Generator erstellten Bild.

Die Initialisierung des Generators und der Discriminator anhand der vorliegenden Bilder werden durch das Framework aufgezeichnet. Dadurch ist es anschließend möglich, die Gradienten mittels der in `generator_loss` und `discriminator_loss` berechneten Fehler des Generators und des Discriminators zu berechnen, um diese Werte an die jeweiligen Optimizer zu übergeben.

Schließlich wird in `train_step` eine maschinenlesbare Zusammenfassung des aktuellen Trainingsschrittes erstellt. In TensorBoard kann diese Zusammenfassung eingelesen werden, um den aktuellen Trainingsdurchlauf und Trainingserfolg zu beobachten und abgeschlossene Trainingsdurchläufe zu vergleichen.

In der Funktion `fit` wird zuerst ein Bild des Testsets geladen. Das Beispielbild wird verwendet, um durch Aufrufen von `generate_images` den aktuellen Trainingserfolg in Form einer Bilddatei im Ausgabebezeichnis zu speichern. Dieser Vorgang erfolgt nach je 1000 Trainingsschritten.

Über die Trainingsbilder wird wiederholt iteriert und für jedes Trainingsbild `train_step` aufgerufen.

Nach je 5000 Trainingsschritten speichert `fit` außerdem einen Checkpoint. Dadurch kann einerseits das Training unterbrochen und zu einem späteren Zeitpunkt fortgesetzt werden. Andererseits wird das Modell auf diese Weise persistiert, und es ist später möglich, das

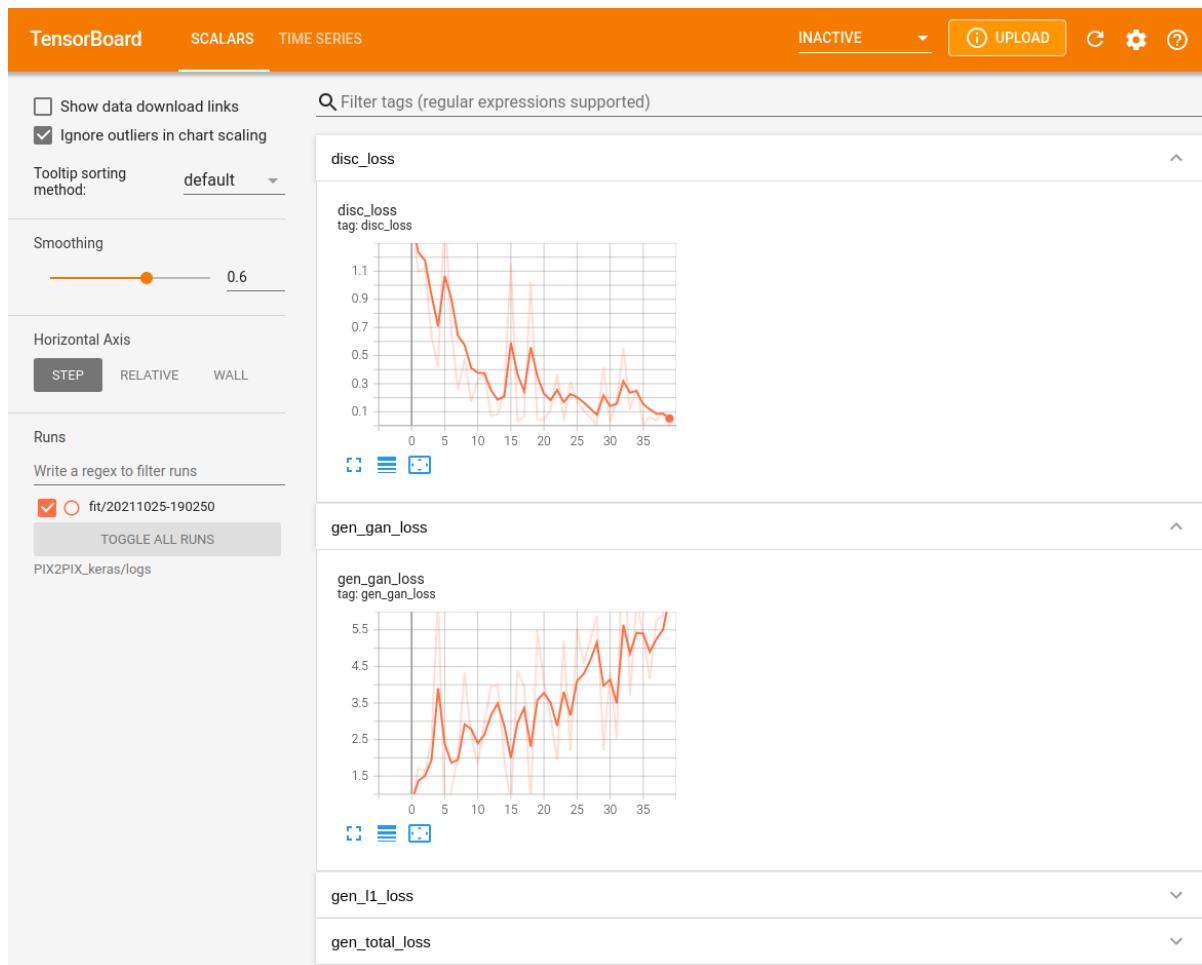


Abb. 3.1.: Im TensorBoard können die Fehlerraten des aktuellen Trainingsdurchlaufs abgelesen oder frühere Trainingsdurchläufe miteinander verglichen werden.

künstliche neuronale Netz für seinen eigentlichen Einsatzzweck zu nutzen, ohne es erneut zu trainieren.

Im Ausführungsstrang wird zunächst eine Option des GPU-Speichermanagements gewählt, um die Größe des reservierten Speichers dynamisch anwachsen zu lassen [22]. Zu Beginn der Experimente hat sich diese Einstellung in Bezug auf die Performance als vorteilhaft herausgestellt.

Es werden auch einige Parameter wie die Bildgröße und die Größe des Trainingssets und Hyperparameter wie die Batchgröße gesetzt. Das Trainings- und das Testset werden eingelesen, der Generator und die Discriminator erstellt und der Adam-Optimizer initialisiert. Nachdem anschließend das Schreiben der Checkpoints und der Zusammenfassung vorbereitet wurde, erfolgt das Training durch Aufruf von `fit`. Im letzten Schritt wird das Modell mit 5 Beispielbildern aus dem Testset aufgerufen.

4. Experimente und Resultate

1. Vorbereitung der vorhandenen Eingabedaten

Die Trainingsdaten liegen für das “Quick, Draw!”-Dataset im NDJSON-Format und als Blender-Dateien vor. Die effizienteste Möglichkeit, die Skizzen und 3D-Modelle für die Verarbeitung in einem CNN vorzubereiten, ist das Rendern und Speichern als Bilddateien. Als Dateiformat kommen JPEG oder PNG infrage. Beide Formate können leicht als Trainingsset mit Tensorflow geladen werden.

Das “Quick, Draw!”-Dataset steht unter der Creative-Commons-Lizenz CC BY 4.0. Eine Zusammenfassung der Lizenz ist auf [creativecommons.org¹](https://creativecommons.org/licenses/by/4.0/) nachzulesen. Der Urheber ist Google, Inc. und das Material kann dem Github-Repository [googlecreativelab/quickdraw-dataset²](https://github.com/googlecreativelab/quickdraw-dataset²) entnommen werden. An dem Material in dem Dataset wurden durch mich keine Änderungen durchgeführt. Die Lizenz erfordert, dass diese Informationen genannt werden, was somit geschehen ist.

NDJSON steht für Newline Delimited JavaScript Object Notation. Die Spezifikation ist im Github-Repository [ndjson/ndjson-spec³](https://github.com/ndjson/ndjson-spec³) zu finden. In einer solchen Datei sind also zeilenweise JSON-Objekte gespeichert. Für jede Zeichnung sind Angaben zu Motiv, Ort und Zeit enthalten. Außerdem ist angegeben, ob die künstliche Intelligenz des Minispiels das Motiv in der Zeichnung korrekt klassifiziert, also erkannt hat. Jede Zeichnung hat weiterhin eine eindeutige ID.

Der relevanteste Teil ist die “Drawings”-Eigenschaft der JSON-Objekte, ein mehrdimensionales Array mit Bildkoordinaten. Es enthält mindestens X-Koordinaten und Y-Koordinaten in jeweils einem Array im “Drawings”-Array. Indem Linien zwischen den Bildkoordinaten in der Reihenfolge der Arrayelemente in ein Bild gezeichnet und in einer Bilddatei gespeichert werden, können die Zeichnungen in beliebigen Bilddateiformaten gespeichert werden. Für diese Arbeit wurde die Konvertierung ebenfalls in Python implementiert.

Die Implementierung im Skript im Anhang 2 beginnt mit den erforderlichen Importen, unter anderem um JSON zu dekodieren und Dateien im JPEG-Format erstellen zu können. Anschließend wird die Anzahl der zu konvertierenden Dateien festgelegt und die NDJSON-Datei als Tensorflow-Dataset vorbereitet.

1 <https://creativecommons.org/licenses/by/4.0/>

2 <https://github.com/googlecreativelab/quickdraw-dataset>

3 <https://github.com/ndjson/ndjson-spec>

In einer Schleife wird nun über die Zeilen der NDJSON-Datei iteriert. Zuerst wird der Fortschritt des Skripts auf der Konsole ausgegeben. Die nächste Zeile des Datasets wird in ein Python-Objekt dekodiert, und die “Drawing”-Eigenschaft, in der die gezogenen Linien enthalten sind, in einer Variablen abgelegt.

Als nächstes wird die Bilddatei vorbereitet. Die Zeichnung soll als schwarze Linie auf weißem Hintergrund in einem 256x256 Pixel großen RGB-Bild gespeichert werden. In der darauffolgenden Zeile wird ein PIL-Objekt erstellt, das Zeichenfunktionen zur Verfügung stellt.

Da eine Zeichnung fast nie die gesamte verfügbare Breite und Höhe der in dem Minigame verfügbaren Zeichenfläche auch nutzt, würde das Ergebnis ohne eine entsprechende Normalisierung der Koordinaten an der linken und der oberen Grenze des Ausgabebildes anliegen und an der rechten und unteren Seite des Bildes ein freier Rand entstehen. Deshalb werden in dem Abschnitt unter dem Kommentar `# normalise coords to center the drawing` die Minimal- und Maximalwerte auf jeder Achse ermittelt und daraus ein Offset für die Koordinaten berechnet, um die Zeichnung im Ausgabebild zu zentrieren.

Über die Koordinaten iterierend wird die Handzeichnung dann in das RGB-Bild gezeichnet. Anschließend wird das Bild als JPEG-Datei unter einem eindeutigen Namen gespeichert.

Das Skript endet, nachdem alle Zeilen der NDJSON-Eingabedatei verarbeitet oder 5000 Bilddateien erstellt wurden.

Bei der Verarbeitung in einem Convolutional Neural Network spielt die Bildgröße in Bezug auf die Verarbeitungszeit eine wichtige Rolle. Bildformate der Größe 256x256 oder kleiner sind üblich und gut geeignet. Für ein GAN ist es zwar nicht erforderlich, aber sinnvoll, für Ein- und Ausgabedaten dieselben Dimensionen festzulegen. Bei der Bildgenerierung aus Skizzen unterscheiden sich Ein- und Ausgabedaten natürlich in der Farbtiefe. Während die Skizzen Graustufenbilder sind, besitzen die generierten Bilder drei Farbkanäle (RGB).

Sowohl während der Entwicklung als auch zur Laufzeit kann es vorteilhaft sein, Farbwerte statt auf der oft verwendeten Skala von 0 bis 255 als Fließkommazahlen im Bereich 0,0 bis 1,0 darzustellen. Auch für die Ausgaben der einzelnen Schichten eines künstlichen neuronalen Netzes kann diese Normalisierung durchgeführt werden.

Ein ebenso wichtiger wie aufwendiger Vorgang ist die Klassifizierung der Trainingsdaten, also die Zuweisung von Eingaben zu den erlernbaren Ergebnissen. Aufgrund der selbst erstellten Ausgabebilder existieren für diesen Zweck keine vorgefertigten Datasets. Die Sortierung und Zuweisung erfolgt deshalb manuell.

2. Erstellung eigener Eingabedaten

Die Vorteile, die selbst erstellte Eingabedaten bieten, rechtfertigen in dieser Arbeit das Erstellen vieler eigener Bilder. Dafür ist es nicht erforderlich, jedes Bild einzeln von Hand zu erzeugen. Blender verfügt über eine Python-Schnittstelle. Mit ihr ist es möglich, etwa die Ausrichtung eines dreidimensionalen Objekts automatisiert mehrmals zu ändern und jeweils ein Bild zu erstellen, sodass aus einem einzigen Modell mehrere Eingabebilder entstehen. Bei einigen Modellen wie zum Beispiel Tischen konnte sogar die Form aus zufälligen Werten für Höhe, Breite und Tiefe generiert werden.

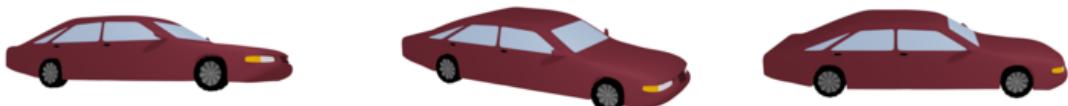


Abb. 4.1.: 3D-Modelle von Fahrzeugen können unterschiedlich in der Szene ausgerichtet werden. Dadurch entstehen verschiedene, aber homogene Eingabebilder.



Abb. 4.2.: Tische können fast vollkommen automatisiert erstellt werden. Für Höhe, Breite und Tiefe der Tischbeine und Tischplatte wurden jeweils zufällige Werte innerhalb bestimmter sinnvoller Bereiche ermittelt. Dadurch konnte mit verhältnismäßig geringem Aufwand eine große Menge an Eingabebildern produziert werden.

3D-Modellierungsprogramme haben prinzipiell ungewohnte Benutzeroberflächen. Das Navigieren in einer dreidimensionalen Szene erfordert Eingaben, die mittels Maus und Tastatur nicht gleichzeitig intuitiv und effizient ausgeführt werden können.

Für das Modellieren ist aber exaktes Arbeiten nötig. Ein 3D-Modellierungsprogramm muss deshalb in der Lage sein, präzise Änderungen im dreidimensionalen Raum nach den Eingaben des Benutzers durchzuführen. Blender besitzt dafür vielfältigste Funktionen. Insbesondere gibt es neben der freien Navigation im Raum verschiedene vordefinierte Perspektiven wie zum Beispiel Draufsicht und Seitenansicht. Es gibt Orientierungslinien in der Szene, an denen ein geometrisches Objekt ausgerichtet werden kann und auch sollte, um korrekt modellieren zu können.

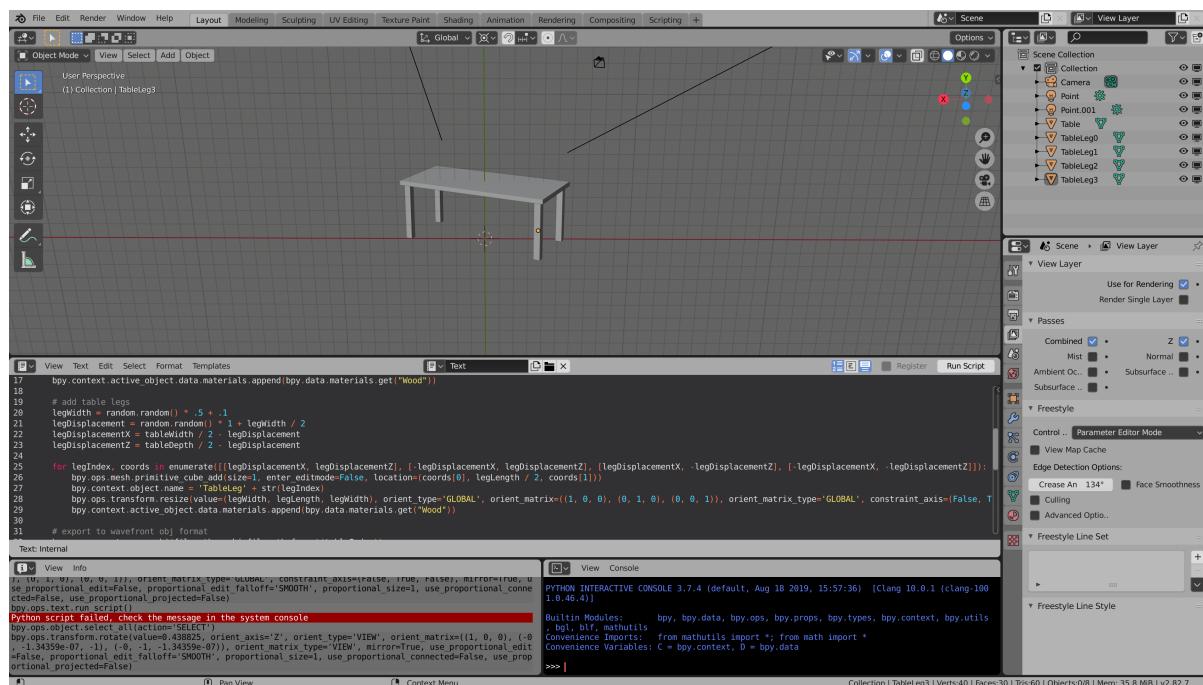


Abb. 4.3.: Blender bietet verschiedene Perspektiven und Werkzeuge, um komplexe Gebilde präzise zu modellieren.

Blender bietet neben 3D-Navigations-Schaltflächen eine große Zahl an Werkzeugen, mit denen Modelle erstellt und bearbeitet werden können, sowie vorgefertigte Formen. Objekte können unter anderem verschoben, verzerrt, geteilt und zusammengefügt werden.

Zudem gibt es Szenen-Objekte, die besondere Funktionen besitzen. Neben der Kamera, deren Position und Ausrichtung für das Ergebnisbild entscheidend ist, können auch verschiebenartige Lichtquellen und Spiegel in der Szene platziert werden, die in die Berechnung des Endergebnisses einbezogen werden. Zudem können Animationseffekte wie Feuer, Wasser und Wind erzeugt werden.

4. Experimente und Resultate

Es wurde ein Blender-Python-Script entwickelt (s. Anhang 3), um Bilder automatisiert erstellen zu lassen. Das Skript exportiert das Ergebnis als Waveform-OBJ und als JPEG-Bilddatei.

Zunächst werden drei Zufallswerte für die Breite, Höhe und Tiefe jeweils der Tischplatte sowie der Tischbeine ermittelt. Für den Abstand der Tischbeine von der Tischkante wird ein weiterer Zufallswert berechnet. Anschließend werden Blender-Standard-Würfel so in der Szene platziert und anhand der Zufallswerte verformt, dass die Form eines Tisches entsteht.

Die Tischform wird mit einer Textur versehen, damit eine Holzoptik entsteht.

Die Objekte werden anschließend ausgewählt, um fünfmal um den entsprechenden Anteil einer Umdrehung ($\pi/5$) gedreht zu werden. Dies geschieht in einer Schleife. Nach jeder Drehung wird eine Bilddatei gespeichert.

Der gesamte Vorgang wird selbst in einer Schleife durchgeführt, sodass nach einer Ausführung 50 verschiedene Tische generiert und zu jedem Tisch fünf Bilddateien, also insgesamt 250 Bilddateien, gespeichert wurden.

3. Anwendung herkömmlicher Shader

In der Computergrafik wird die Darstellung der Oberfläche eines Objekts durch die drei Faktoren Material, Textur und Ausleuchtung bestimmt. Material ist die Grundfarbe der Oberfläche. Es legt fest, wie das sichtbare Spektrum des Lichts von der Oberfläche des Objekts reflektiert wird. Ein Material legt außerdem fest, ob die Oberfläche matt oder metallisch erscheint. Es kann entsprechend der drei Farbschemas RGB, HSV oder Hex dargestellt werden. Wie die Farbe in jedem Schema erscheint ist auch von einem Alpha-Wert, welcher für die Menge der Transparenz steht, abhängig. Textur sind die physischen Charakteristiken der Oberfläche, und Ausleuchtung ist die Hintergrundbeleuchtung oder Licht, welches von Lichtern (Lampen) emittiert wird. [23]

Texturen definieren das physische Erscheinungsbild einer Oberfläche, also etwa wie glatt oder uneben diese erscheint, oder ihre Struktur, welche die visuelle Wahrnehmung der physischen Beschaffenheit des Objekts definiert. Diese Definition bestimmt, woraus die Oberfläche besteht, wie Holz, Ziegelsteine, Wasser und so weiter. Texturen werden durch Algorithmen generiert, wie sie in Blender integriert sind (prozedurale Texturen) oder aus Bilddateien (Bildtexturen). [23]

Shader definieren die Interaktion des Lichts mit der Oberfläche des Objekts. Dabei können Shader aus einem oder mehreren BSDFs (Bidirectional Scattering Distribution Function) bestehen, die wiederum von Mix-und Add-Shadern in der Zusammensetzung gemischt werden. Polygonobjekte sind anfangs immer ungeglättet, das heißt, dass beim Rendern oder in der schattierten Ansicht zunächst immer die einzelnen Flächen zu sehen sind, aus denen sich das

Objekt zusammensetzt. Eine mögliche, wenn auch in Sachen Renderzeit und Speicherverbrauch ungünstige Methode wäre, einfach das Objekt so weit in kleinere Flächen zu unterteilen, dass beim Rendern eine Fläche pro Pixel gerendert wird. In der praktischen Arbeit verwendet man deshalb einen Trick, bei dem die Übergänge zwischen den einzelnen Flächen “glattgerechnet” werden. Übliche Verfahren hier sind das Gouraud oder Phong Shading. [24]

Die Qualität eines Bildes hängt direkt von der Effektivität des Shading-Algorithmus ab, der wiederum von der Modellierungsmethode des Objektes abhängt. Zwei wesentliche Methoden der Objektbeschreibung werden häufig verwendet, nämlich Oberflächendefinition mittels mathematischer Gleichungen und Oberflächenapproximation durch Mosaiken aus polygonalen Flächen. [25]

4. Hyperparameter

Die Pix2Pix-Referenzimplementierung ist bereits für die Übersetzung von Skizzen in Fotos eingestellt. Für das Training waren anfangs 40000 Trainingsschritte vorgesehen. Je nach Größe des Trainingssets werden also mehrere Wiederholungen mit demselben Trainingsset durchgeführt, um zufriedenstellende Ergebnisse zu erzielen. Nach Abschluss der Experimente könnte die Anzahl der Trainingsschritte auf 30000 reduziert werden, um die Dauer des Trainings und die Ergebnisse zu optimieren. Dies ist aber wiederum abhängig von der Größe des Trainingssets. Es war auch im Sinne des Experiments, das Modell einige Tausend Schritte mehr durchführen zu lassen als für das optimale Resultat erforderlich gewesen wären.

Die Eingabebilder sind 256x256 Pixel groß und besitzen einen Farbkanal für Graustufen. Sie werden am Anfang des Trainingsprozesses durch sogenanntes Jittering augmentiert. Dabei werden die Bilder zuerst auf 286x286 Pixel vergrößert und anschließend auf einen zufälligen 256x256 Pixel großer Ausschnitt wieder verkleinert. Änderungen an diesen Pixelgrößen haben im Experiment keine nennenswerte Verbesserung bewirkt.

Der Adam Optimierer [26] erhält für die Learning-Rate den Wert 0,0002. Das Momentum ist auf 0,9 voreingestellt. Diese beiden Werte beeinflussen die Lerngeschwindigkeit und sind in begrenztem Maße anpassbar, sind aber bereits für meinen Zwecke optimal eingestellt.

5. Performancebeobachtungen

Das Training umfasst sinnvollerweise 15000 bis 40000 Iterationen. Die Implementierung sieht vor, dass alle 1000 Iterationen ein Zwischenergebnis in Form dreier Bilder gespeichert wird. Das zu speichernde Beispiel kann dabei zufällig gewählt werden, oder es kann der Fortschritt eines einzelnen Input-Target-Paars über die Iterationen hinweg gezeichnet werden.

4. Experimente und Resultate

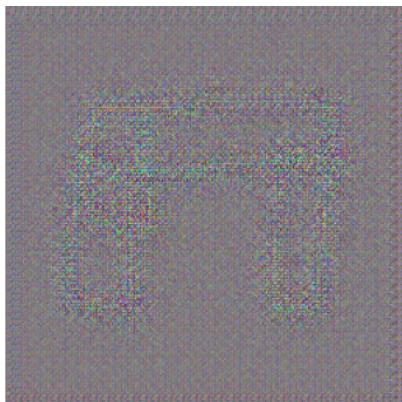


Abb. 4.4.: Generierte Bilder von Tischen vor dem Training, nach 1000 Iterationen und nach 2000 Iterationen

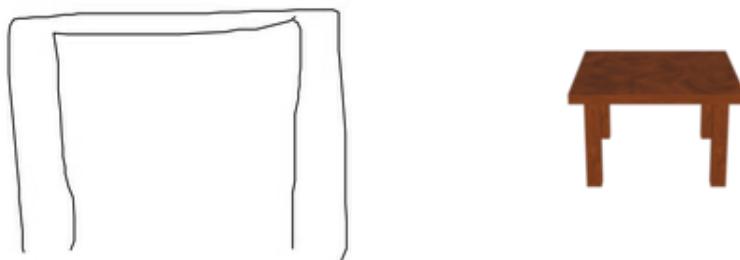


Abb. 4.5.: Ein Beispiel eines Tisches, dessen Textur und Form gelernt werden soll

Abbildung 4.5 zeigt beispielhaft eine handgezeichnete Skizze eines Tisches und das erwartete Ergebnis. In diesem Beispiel wird versucht, aus einem vermutlich quadratischen, gänzlich von der Seite gezeichneten Tisch einen quadratischen, von leicht schräg oben betrachteten texturierten Tisch zu generieren.

In Abbildung 4.4 ist zu erkennen, wie sich innerhalb 2000 Iterationen die Qualität der generierten Bilder von zufälligem Bildrauschen zu einem bereits als Tisch erkennbaren Objekt verbessert. In diesem Trainingsset befanden sich unterschiedlich hohe, unterschiedlich breite und auch im Raum unterschiedlich ausgerichtete Tische. Es ist auch zu erkennen, dass aus der handgezeichneten Skizze bereits ein quadratischer Tisch generiert wird.

Nicht korrekt werden in dieser Phase des Trainings die geraden Linien der Zielbilder erzielt. Auch die Anzahl der Tischbeine stimmt noch nicht.

In Abbildung 4.6 wird der Lernerfolg während der zweiten Hälfte von insgesamt 20000 Iterationen gezeigt. Zunächst ist das Ergebnis noch stellenweise unscharf und die Länge der



Abb. 4.6.: Generierte Bilder von Tischen nach 10000, 15000 und 20000 Iterationen

Tischbeine ist ungleich. Die Kanten sind bereits gerade. Nach 15000 Iterationen ist zu erkennen, dass häufigeres Trainieren nicht unbedingt bessere Ergebnisse bringt. Nach Abschluss des Trainings sieht der generierte Tisch aus wie die Vorgabe im Trainingsset.

Das Modell hat Form und Farbe verschiedener Tische gelernt. Auch die Unregelmäßigkeiten in der Färbung der Tische wurde erlernt. Das ist die zweite Frage, die in dieser Arbeit zu beantworten war. Es ist einem künstlichen neuronalen Netz also möglich Texturen zu erlernen.



Abb. 4.7.: Korrekt ausgerichteter Tisch mit unregelmäßigen Kanten



Abb. 4.8.: Tisch mit geraden Kanten, aber falscher Ausrichtung

4. Experimente und Resultate

In Abbildung 4.7 ist ein Beispielergebnis aus einem Trainingsdurchlauf mit 262 Tischen zu sehen. Der Tisch ist so im Raum ausgerichtet wie im Zielbild vorgesehen. Die Form ist aber ungleichmäßig und das Bild teilweise verschwommen.

Abbildung 4.8 zeigt eines der Ergebnisse aus einem Trainingsdurchlauf mit 960 verschiedenen Tischen. Der Tisch hat gerade und scharfe Kanten, die Tischplatte ist aber parallel zum unteren und oberen Bildrand.

Der Vergleich zeigt, wieviel Bedeutung der Datenqualität des Trainingssets zukommt. Das kleinere Trainingsset war eine Teilmenge des größeren, enthielt aber anteilig mehr schräg im Raum ausgerichtete Tische. Obwohl das mit 960 Bildern trainierte Modell also dieselben schräg ausgerichteten Tische verwendete, generiert es vorzugsweise Bilder von gerade ausgerichteten Tischen.

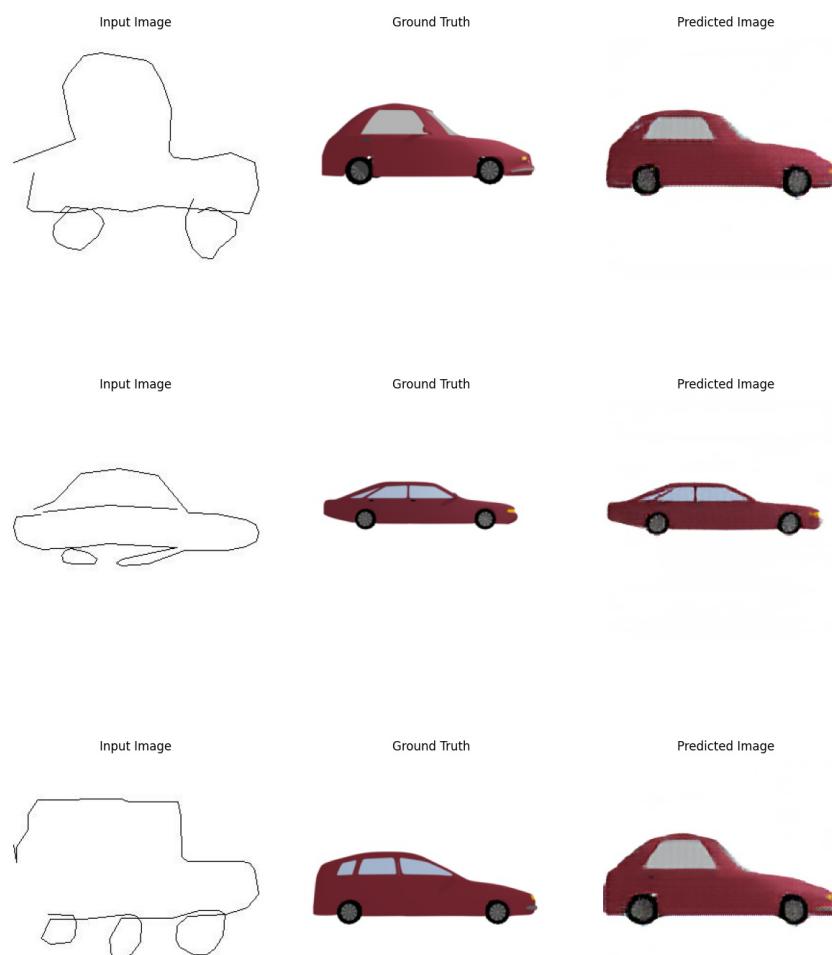


Abb. 4.9.: Beispiele für generierte Autos

Fahrzeuge, wie in Abbildung 4.9, sind eine vergleichbar schwierige Aufgabe für ein künstliches neuronales Netz. Aufgrund der Vielfalt an möglichen Fahrzeugen, für die der Begriff "Auto" (englisch "Car") stehen kann, unterscheiden sich die Zeichnungen im Quick!Draw-Dataset sehr stark und sind nur schwer zu sortieren und Zielbildern zuzuordnen.

Genauso aufwendig ist es, verschiedene Fahrzeuge zu modellieren, um sie als Zielbilder zu verwenden. Anders als bei vielen Eigenschaften von Tischen, wie die Stärke der Tischplatte und der Tischbeine, lassen sich die Unterschiede von PKWs nicht genauso gut durch Zufallszahlen beschreiben. Auch sind die Autos fast ausschließlich in der Seitenansicht gezeichnet, sodass automatisch in verschiedenen Perspektiven generierte Blendermodelle der Fahrzeuge fast nicht nutzbar sind.

Die Handskizzen stellen hauptsächlich PKWs dar und wurden deshalb nur in die drei Kategorien Limousine, Kleinwagen und Kombi unterteilt und entsprechenden Zielbildern zugewiesen.

In den Abbildungen sind unterschiedlich gute Resultate zu sehen. Für einen praktischen Einsatzzweck, wie zum Beispiel ein Lehrmittel in einer Fahrschule, wäre dieses Modell bestenfalls als Grundlage für weiterführende Trainings geeignet.

Die Trainingsergebnisse habe ich hauptsächlich durch Ansicht beurteilt. In Tensorflow wird der Trainingsfortschritt in Zahlen als Diagramm dargestellt. Das verwendete Skript erstellt drei Diagramme für den Generator und ein Diagramm für den Discriminator. Generator und Discriminator werden gleichzeitig trainiert. Der Verlust des Discriminators nimmt im Verlauf des Trainings ab. Nach 20000 bis 30000 Iterationen ist die Abnahme weniger stark.

Der Discriminator wird auf das vom Generator erstellte Bild und auf das Zielbild angewendet. Der Gesamtverlust des Discriminators wird aus diesen beiden Ergebnissen gebildet und bezeichnet den Erfolg beziehungsweise Misserfolg des Discriminators, das durch den Generator erstellte Bild als solches von einem Zielbild zu unterscheiden.

Wie bei cGANs üblich wird der Gesamtverlust des Generators ebenfalls aus zwei verschiedenen Ergebnissen berechnet. Die beiden Faktoren sind in Abbildung 4.11 als gen_l1_loss und gen_gan_loss bezeichnet.

Einerseits wird der pixelweise Unterschied zwischen dem generierten Bild und dem Zielbild als Verlustindikator herangezogen. Dieser Wert wird 100fach gewichtet, hat also einen entsprechend größeren Einfluss auf das Gesamtergebnis. Er nimmt im Laufe des Trainings sichtbar ab.

Das GAN-Loss berücksichtigt das Ergebnis des Discriminators, also wie gut dieser das generierte Bild von einem Zielbild unterscheiden konnte. Im Diagramm ist zu erkennen, dass sich der Verlust im Verlauf des Trainings sogar erhöht. Die Erklärung dafür ist der immer geringere Verlust des Discriminators, der dadurch generierte Bilder und Zielbilder besser voneinander unterscheiden kann.

4. Experimente und Resultate

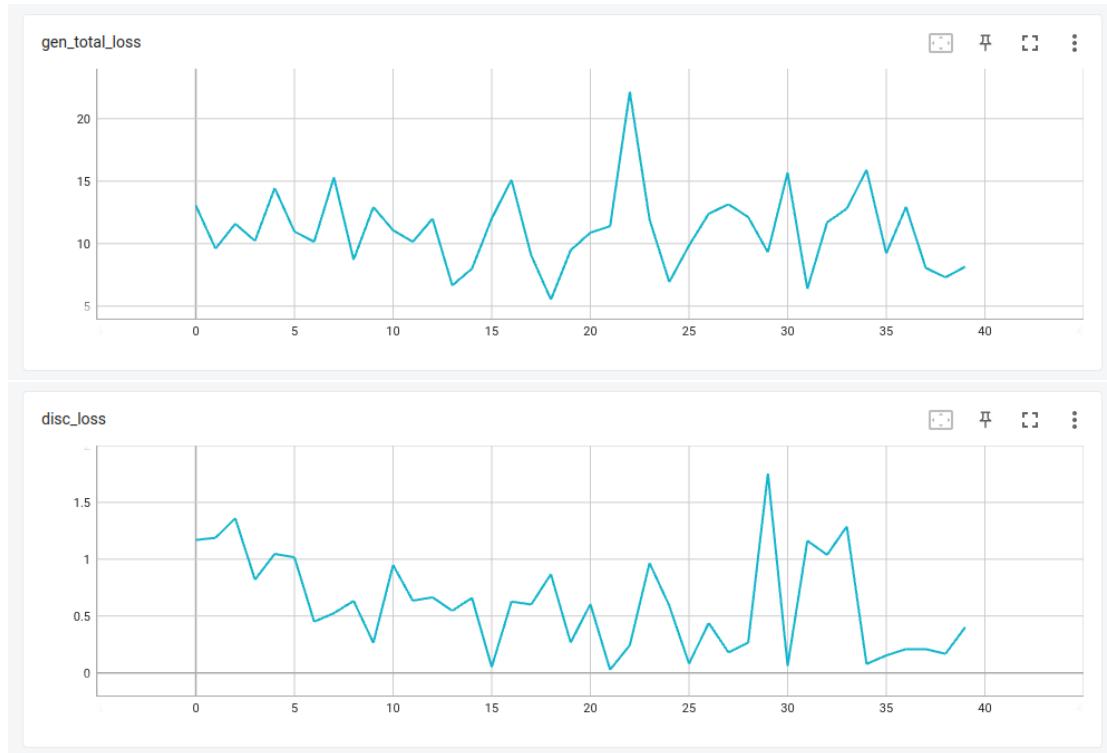


Abb. 4.10.: Die Tensorboard-Diagramme des Gesamtverlusts jeweils für Generator (gen_total_loss) und für Discriminator (disc_loss)

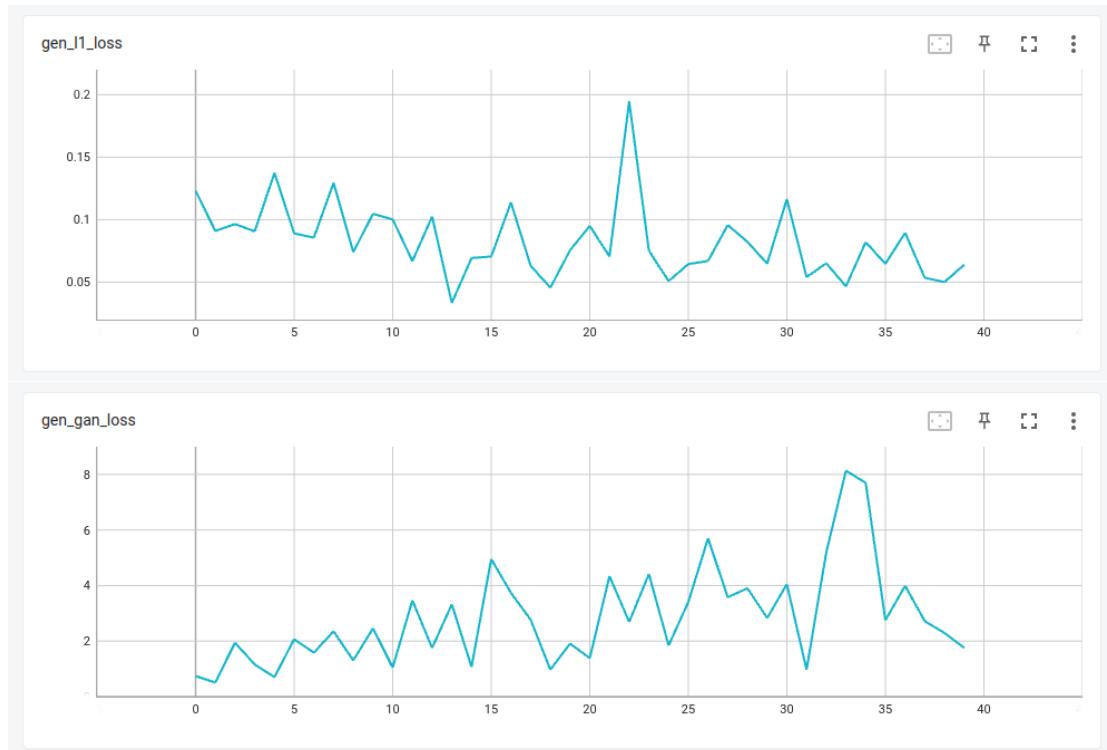


Abb. 4.11.: Die Tensorboard-Diagramme der Generator-Verluste

In den Diagrammen ist der Verlauf eines Trainings über 40000 Iterationen abgebildet. Im Trainingsset befanden sich 262 verschiedene Tische und das Testset enthielt sechs Tische. Trotz des relativ kleinen Trainingssets wurde ein visuell überzeugendes Resultat erzielt. Nach 20000 bis 30000 verbesserte sich der Lernerfolg nicht mehr wesentlich. Ein Versuch mit 80000 Iterationen hat gezeigt, dass eine viel größere Anzahl Iterationen das Resultat eher verschlechtert.

5. Diskussion

Bildgenerierung mittels künstlichen neuronalen Netzen wurde bereits in verschiedenen Experimenten und mit teilweise sehr guten Ergebnissen durchgeführt und dokumentiert. Es können Bildteile aus den umliegenden Bildinformationen neu generiert werden, wie es mit Fotobearbeitungssoftware möglich ist.

Seit einiger Zeit sind Handschrifterkennungs-Algorithmen verbreitet. Eine Variante der Bildgenerierung ist deren Umkehr, indem zum Beispiel aus einstelligen Zahlen handschriftlich aussehende Ziffern generiert werden. Außerdem kann der Gesamteindruck eines vorhandenen Bildes, auch durch Verwendung eines weiteren Bildes, verändert und auf diese Art ein neues Bild generiert werden.

Die Arbeit mit künstlichen neuronalen Netzen hat gezeigt, dass ein Großteil des Aufwands aus der Erhebung und Vorbereitung der Daten besteht. Geeignete Eingabebilder mussten entweder gesammelt oder selbst erstellt werden. Durch Automatisierung konnte der Erstellungsprozess verkürzt werden, machte aber immer noch einen erheblichen Anteil der Arbeit aus. Sowohl gesammelte als auch selbst erstellte Bilder eignen sich nicht ausnahmslos für den gewählten Zweck. Es war deshalb weiterhin erforderlich, die Bilder händisch zu sortieren. Anschließend war noch jeweils die Zuweisung einer Handskizze zu einem Ergebnisbild erforderlich, die ebenfalls manuell erfolgen musste.

Das Training des Modells ist ein weiterer aufwendiger Prozess. Im Laufe eines Training-durchlaufes eines künstlichen neuronalen Netzes zeichnet sich bereits ein Endergebnis ab. Es ist daher möglich, anhand weniger Zwischenergebnisse schwerwiegende Fehler zu bemerken und den Durchlauf von vorn zu beginnen. Das Endergebnis ist selbstverständlich erst nach Abschluss des Trainings sichtbar. Wenn es darum geht herauszufinden, wieviele Durchläufe für ein perfektes Ergebnis erforderlich sind, kann es erforderlich sein das Modell mehrere Male hintereinander auf dasselbe Ergebnis zu trainieren.

Künstliche neuronale Netze sind mathematische Konstrukte. Zunächst ist es deshalb erforderlich, die Fachbegriffe der entsprechenden Literatur zu kennen. Die wissenschaftlichen Texte liegen hauptsächlich auf Englisch vor, sodass man die Begriffe sinnvollerweise auf Deutsch und auf Englisch lernt. Das künstliche neuronale Netz selbst wird als eine Struktur aus Tensoren dargestellt.

Die Algorithmen des maschinellen Lernens berechnen üblicherweise aus der Eingabe mehrere weitere Schichten aus Tensoren. Die Dimensionalität der Tensoren ändert sich auf jeder weiteren Schicht. Einige Algorithmenklassen, wie zum Beispiel die U-Net-Architektur, sind benannt nach der Art, auf die sich die Dimensionalität der Tensoren ändert.

Für diese Arbeit waren die Generative Adversarial Networks besonders interessant. Mit ihnen werden zwei lernende Komponenten gegeneinander angesetzt. Eine von ihnen lernt die eigentliche Aufgabe zu erfüllen. Die zweite Komponente wird darauf trainiert, die Ergebnisse der ersten zu bewerten und Feedback zu den Ergebnissen zu geben.

Die für maschinelles Lernen erforderlichen Berechnungen können auf handelsüblichen PCs ausgeführt werden. Sie sind aber sehr umfangreich, sodass sich unterstützende Hardware günstig auswirkt. Bei Cloud-Angeboten wie Google Colab und Amazon SageMaker, die auf Implementierungen dieser Art spezialisiert sind, können Algorithmen auf Enterprise-Grade-Hardware ausgeführt werden.

Aber auch Grafikkarten für Spiele-PCs sind durch parallele Tensoroperationen für maschinelles Lernen um ein Vielfaches besser geeignet als die CPU. Für diesen Zweck existiert speziell hergestellte Treibersoftware. Die Parallelisierungssoftware für NVIDIA-Grafikkarten heißt CUDA. Auf die Treiber aufbauend wurden und werden Programmierschnittstellen für verschiedene Programmiersprachen entwickelt.

Eine der weit verbreitetsten Machine-Learning-Programmierschnittstellen ist Tensorflow für Python, die auch für diese Arbeit verwendet wurde. Die Einrichtung auf einem Desktopcomputer ist teilweise umständlich, aber für Tensorflow und CUDA gut dokumentiert. Es gibt Anleitungen für Ubuntu Linux. Die Verwendung der Programmierschnittstelle ist ebenfalls gut dokumentiert. Den in dieser Arbeit verwendeten Machine-Learning-Algorithmus Image-To-Image-Translation ist in der Beispielektion auf tensorflow.org¹ zu finden.

Das Modell muss immer wieder von vorn trainiert werden, bevor erste brauchbare Ergebnisse erzielt werden. Grundsätzlich gilt, dass die Menge der Eingabedaten so groß wie möglich sein sollte, damit ein Lernprozess erfolgreich sein kann. Trotzdem sind weitere Eigenschaften des Datasets wichtig. An dem Beispiel der Tische wurde deutlich, dass die Perspektive, also die Ausrichtung der Tische im Raum, nur bei einem ausreichenden Anteil an schräg ausgerichteten Beispielen erlernt wird. Dafür kann es sogar sinnvoll sein, die Gesamtgröße des Datasets zu reduzieren, damit alle Varianten in genügend großen Anteilen im Dataset vorhanden sind.

Die Anzahl der Trainingsdurchläufe ist eine der wichtigsten Größen beim maschinellen Lernen. In Klassifizierungsalgorithmen gibt es das Phänomen des Overfittings, das die Erfolgsrate bei den Klassifizierungen negativ beeinflusst. Dieses Problem gibt es bei der Bildgenerierung zunächst nicht, solange wie beschrieben die Beispiele im Dataset ausgewogen sind. Trotzdem ist diese Größe selbstverständlich für die Untersuchung und den Einsatz eines ML-Modells aufgrund des Zeitaufwands relevant.

Während der Versuchreihen wurde erkennbar, welche Bildinformationen durch ein künstliches neuronales Netz erlernt werden können, um eigenständig Bilder zu generieren. Die Möglichkeit Bilder überhaupt mittels künstlichen neuronalen Netzen generieren zu können war die erste und wichtigste Frage, die mit dieser Arbeit beantwortet werden sollte. Die Frage wurde recherchiert, und es standen mehrere Methoden zur Auswahl.

¹ <https://tensorflow.org>

5. Diskussion

Die konkreten Einsatzmöglichkeiten für Deep Learning sind, abgesehen von einigen nicht praxisorientierten Anwendungsbeispielen, ausschließlich in wissenschaftlichen Artikeln zu finden. Diese Artikel setzen beim Leser üblicherweise ein Verständnis der Grundlagen voraus und verweisen bestenfalls auf vorherige Arbeiten. Für die grundlegenden Prinzipien des maschinellen Lernens und verschiedene Arten von künstlichen neuronalen Netzen wie CNNs und RNNs gibt es Literatur in Form von Büchern. Die Bücher liegen größtenteils in englischer Sprache vor. Es gibt nur wenig deutsche Literatur. Die wissenschaftlichen Artikel zu dem Thema gibt es ausschließlich auf Englisch.

Die besonders vielseitig einsetzbare Methode namens Image-To-Image-Translation ist in der Lage fehlende Bildinformationen zu generieren, neu zu generieren oder jeweils den umgekehrten Vorgang durchzuführen. Zum Beispiel können Schwarz-Weiß-Bilder koloriert, Luftaufnahmen in Karten übersetzt oder Straßenbilder nur aus Informationen zu Verkehrsteilnehmern und Straßenverläufen generiert werden, beziehungsweise jeweils auch umgekehrt. Mit dieser Bildübersetzung ist es auch möglich, aus handgezeichneten Gesichtern Portraitfotos oder Portraits, die wie von einem Künstler gemalt aussehen, zu generieren. Das kommt dem Generieren von Bildern aus handgezeichneten Skizzen am nächsten, anhand dessen in dieser Arbeit diese Fähigkeit künstlicher neuronaler Netze gezeigt werden sollte.

In dem Artikel zu Image-To-Image-Translation wird die zugrundliegende Situation beschrieben, in der von einem Eingabebild zum Ausgabebild Pixel für Pixel übersetzt wird. Dies trifft etwa auf die Übersetzung eines Umrisses auf ein Foto desselben Objekts zu. Der Algorithmus lernt dann dem Objekt eine Textur zu geben. Die Frage, ob ein künstliches neuronales Netz lernen kann Texturen zu generieren, war die zweite Frage, die in dieser Arbeit zu beantworten war. Die Ergebnisse zu den entsprechenden Versuchen waren größtenteils sehr gut. Die Holztextur für die Tische und auch der Helligkeitsverlauf der brennenden Kerzen wurden sehr gut gelernt.

Eine Schwäche bei der Generierung von Bildern aus Handskizzen scheinen die unregelmäßigen und fast nie geraden Umrisse der skizzierten Objekte zu sein. Auch in den Trainingsergebnissen mit vielen einwandfrei generierten Bildern fanden sich häufig noch Bilder mit unsauberen Stellen. Besonders die scharfen Übergänge wie zum Beispiel vom Rad eines Autos zur Karosserie oder einfach die Kanten des Objektes waren oft verschwommen. Die Flammen auf den Zielbildern der Kerzen waren von einem Lichtschein umgeben. Der wurde zwar gelernt, aber das Ergebnis ist optisch nicht ansprechend. Tische waren im Trainingsset immer mit genau vier Beinen vorgegeben. Auf den Ergebnisbildern waren oft andere Anzahlen der Tischbeine zu sehen.

Einige der Schwierigkeiten könnten gemindert werden, indem die Eingabedaten noch stärker vorbereitet werden. Zum Beispiel könnten die Maße und die Ausrichtung in den Skizzen der Tische ermittelt werden und diese Werte für die Generierung der Ergebnisbilder verwendet werden. Der Lichtschein um die Flamme einer Kerze könnte auch auf eine andere Weise als durch ein künstliches neuronales Netz dem Ergebnisbild nachträglich hinzugefügt werden.

Die praktische Anwendung dieser Methode in einem Anwendungsprogramm zur Innenraumgestaltung ist sehr gut vorstellbar. Dafür müssten nur noch einige weitere Objekte zur Auswahl stehen, wie Möbelstücke, Lampen, Teppiche und so weiter. Ein Classifier wäre dann vorteilhaft, um die Generierung der richtigen Einrichtungsgegenstände sicherzustellen. Auch eine Anwendung in einem Smartboard für die Verkehrserziehung oder in einer Fahrschule ist nach den in dieser Arbeit gezeigten Ergebnissen vorstellbar. Um Fahrzeuge aus verschiedenen Perspektiven generieren zu können sind entsprechende Eingabebilder, sowohl Ergebnisbilder als auch handgezeichnete Skizzen, zu erstellen.

Literatur

1. ISOLA, P. u. a.: *Image-to-Image Translation with Conditional Adversarial Networks*. 2018. Abger. unter arXiv: 1611.07004 [cs.CV].
2. CANNY, J.: A Computational Approach to Edge Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 1986, Jg. 8, Nr. 6, S. 679–698. ISSN 0162-8828. Abger. unter doi: 10.1109/TPAMI.1986.4767851.
3. CHOLLET, F.: *Deep Learning with Python, Second Edition*. Manning, 2021. ISBN 9781638350095. Auch verfügbar unter: <https://books.google.de/books?id=mjVKEAAAQBAJ>.
4. GOODFELLOW, I. u. a.: *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
5. ZHANG, A. u. a.: *Dive into Deep Learning*. 2020. <https://d2l.ai>.
6. GREGOR, K. u. a.: *DRAW: A Recurrent Neural Network For Image Generation*. 2015. Abger. unter arXiv: 1502.04623 [cs.CV].
7. DENG, L.: The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*. 2012, Jg. 29, Nr. 6, S. 141–142.
8. GATYS, L. A. u. a.: *A Neural Algorithm of Artistic Style*. 2015. Abger. unter arXiv: 1508.06576 [cs.CV].
9. CHEN, M. u. a.: Generative Pretraining From Pixels. In: III, H. D.; SINGH, A. (Hrsg.). *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 2020, Bd. 119, S. 1691–1703. Proceedings of Machine Learning Research. Auch verfügbar unter: <https://proceedings.mlr.press/v119/chen20s.html>.
10. OORD, A. van den u. a.: *Pixel Recurrent Neural Networks*. 2016. Abger. unter arXiv: 1601.06759 [cs.CV].
11. RAMESH, A. u. a.: *Zero-Shot Text-to-Image Generation*. 2021. Abger. unter arXiv: 2102.12092 [cs.CV].
12. GOODFELLOW, I. J. u. a.: *Generative Adversarial Nets*. 2014. Abger. unter arXiv: 1406.2661 [stat.ML].
13. RADFORD, A. u. a.: *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2016. Abger. unter arXiv: 1511.06434 [cs.LG].
14. MIRZA, M.; OSINDERO, S.: *Conditional Generative Adversarial Nets*. 2014. Abger. unter arXiv: 1411.1784 [cs.LG].
15. LECUN, Y. u. a.: Object Recognition with Gradient-Based Learning. In: *Contour and Grouping in Computer Vision*. Springer, 1999.

16. RONNEBERGER, O. u. a.: U-Net: Convolutional Networks for Biomedical Image Segmentation. In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer, 2015, Bd. 9351, S. 234–241. LNCS. Auch verfügbar unter: <http://lmb.informatik.uni-freiburg.de/Publications/2015/RFB15a>. (available on arXiv:1505.04597 [cs.CV]).
17. IOFFE, S.; SZEGEDY, C.: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv, 2015. Abger. unter doi: 10.48550/ARXIV.1502.03167.
18. HE, K. u. a.: *Deep Residual Learning for Image Recognition*. 2015. Abger. unter arXiv: 1512.03385 [cs.CV].
19. XIE, S.; TU, Z.: *Holistically-Nested Edge Detection*. arXiv, 2015. Abger. unter doi: 10.48550/ARXIV.1504.06375.
20. LUTZ, M.: *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media, 2013. Animal Guide. ISBN 9781449355715. Auch verfügbar unter: <https://books.google.de/books?id=ePyeNz2Eoy8C>.
21. SANDERS, J.; KANDROT, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN 0131387685.
22. ZACCONE, G.; KARIM, M. R.: *Deep Learning with TensorFlow - Second Edition: Explore Neural Networks and Build Intelligent Systems with Python*. 2nd. Packt Publishing, 2018. ISBN 1788831101.
23. BLAIN, J.: *The Complete Guide to Blender Graphics Computer Modeling & Animation: Computer Modeling & Animation (6th ed.)* A K Peters/CRC Press, 2020. Auch verfügbar unter: <https://doi.org/10.1201/9781003093183>.
24. WARTMANN, C.: *Das Blender-Buch - 3D-Grafik und Animation mit Blender*. 5th. dpunkt.verlag, 2014.
25. PHONG, B. T.: Illumination for Computer Generated Pictures. *Commun. ACM*. 1975, Jg. 18, Nr. 6, S. 311–317. ISSN 0001-0782. Abger. unter doi: 10.1145/360825.360839.
26. KINGMA, D. P.; BA, J.: *Adam: A Method for Stochastic Optimization*. 2017. Abger. unter arXiv: 1412.6980 [cs.LG].

Bildnachweis

Abbildung 3.1: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,
<https://www.tensorflow.org/>, (CC BY 4.0 / Apache License 2.0)

Abbildung 4.3: Blender Foundation,
<https://www.blender.org/>, (GNU GPLv3)

Abbildung 4.10: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,
<https://www.tensorflow.org/>, (CC BY 4.0 / Apache License 2.0)

Abbildung 4.11: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,
<https://www.tensorflow.org/>, (CC BY 4.0 / Apache License 2.0)

TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

Abbildungsverzeichnis

1.1.	Verschiedene Trainingsschritte	3
1.2.	Eigene Beispiele	5
3.1.	TensorBoard	23
4.1.	Verschiedene Perspektiven	26
4.2.	Generierte Tische	26
4.3.	Blender-Arbeitsbereich	27
4.4.	Lernfortschritt am Beispiel von Tischen 1	30
4.5.	Quadratischer Tisch	30
4.6.	Lernfortschritt am Beispiel von Tischen 2	31
4.7.	Tisch korrekt ausgerichtet	31
4.8.	Tisch mit geraden Kanten	31
4.9.	Generierte Autos	32
4.10.	Verlustdiagramm 1	34
4.11.	Verlustdiagramm 2	34

Anhang

A. Quelltexte

1. Image-To-Image-Translation main.py

```
import tensorflow as tf

import os
import time
import datetime

from matplotlib import pyplot as plt
from IPython import display

from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession

def load(image_file):
    # Read and decode an image file to a uint8 tensor
    image = tf.io.read_file(image_file)
    image = tf.image.decode_jpeg(image)

    # Split each image tensor into two tensors:
    w = tf.shape(image)[1]
    w = w // 2
    input_image = image[:, :w, :]
    real_image = image[:, w:, :]

    # Convert both images to float32 tensors
    input_image = tf.cast(input_image, tf.float32)
    real_image = tf.cast(real_image, tf.float32)

    return input_image, real_image

def resize(input_image, real_image, height, width):
    input_image = tf.image.resize(
        input_image, [height, width], method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    real_image = tf.image.resize(
        real_image, [height, width], method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)

    return input_image, real_image

def random_crop(input_image, real_image):
    stacked_image = tf.stack([input_image, real_image], axis=0)
    cropped_image = tf.image.random_crop(
        stacked_image, size=[2, IMG_HEIGHT, IMG_WIDTH, 3])
    )

    return cropped_image[0], cropped_image[1]
```

```

# Normalizing the images to [-1, 1]
def normalize(input_image, real_image):
    input_image = (input_image / 127.5) - 1
    real_image = (real_image / 127.5) - 1

    return input_image, real_image

@tf.function()
def random_jitter(input_image, real_image):
    # Resizing to 286x286
    input_image, real_image = resize(input_image, real_image, 286, 286)

    # Random cropping back to 256x256
    input_image, real_image = random_crop(input_image, real_image)

    if tf.random.uniform() > 0.5:
        # Random mirroring
        input_image = tf.image.flip_left_right(input_image)
        real_image = tf.image.flip_left_right(real_image)

    return input_image, real_image

def load_image_train(image_file):
    input_image, real_image = load(image_file)
    input_image, real_image = random_jitter(input_image, real_image)
    input_image, real_image = normalize(input_image, real_image)

    return input_image, real_image

def load_image_test(image_file):
    input_image, real_image = load(image_file)
    input_image, real_image = resize(
        input_image, real_image, IMG_HEIGHT, IMG_WIDTH
    )
    input_image, real_image = normalize(input_image, real_image)

    return input_image, real_image

def downsample(filters, size, apply_batchnorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)

    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2D(
            filters,
            size,
            strides=2,
            padding='same',
            kernel_initializer=initializer,
            use_bias=False
        )
    )

    if apply_batchnorm:
        result.add(tf.keras.layers.BatchNormalization())

    result.add(tf.keras.layers.LeakyReLU())

    return result

```

A. Quelltexte

```
def upsample(filters, size, apply_dropout=False):
    initializer = tf.random_normal_initializer(0., 0.02)

    result = tf.keras.Sequential()
    result.add(
        tf.keras.layers.Conv2DTranspose(
            filters, size, strides=2,
            padding='same',
            kernel_initializer=initializer,
            use_bias=False
        )
    )

    result.add(tf.keras.layers.BatchNormalization())

    if apply_dropout:
        result.add(tf.keras.layers.Dropout(0.5))

    result.add(tf.keras.layers.ReLU())

    return result

def build_generator():
    inputs = tf.keras.layers.Input(shape=[256, 256, 3])

    down_stack = [
        downsample(64, 4, apply_batchnorm=False), # (batch_size, 128, 128, 64)
        downsample(128, 4), # (batch_size, 64, 64, 128)
        downsample(256, 4), # (batch_size, 32, 32, 256)
        downsample(512, 4), # (batch_size, 16, 16, 512)
        downsample(512, 4), # (batch_size, 8, 8, 512)
        downsample(512, 4), # (batch_size, 4, 4, 512)
        downsample(512, 4), # (batch_size, 2, 2, 512)
        downsample(512, 4), # (batch_size, 1, 1, 512)
    ]

    up_stack = [
        upsample(512, 4, apply_dropout=True), # (batch_size, 2, 2, 1024)
        upsample(512, 4, apply_dropout=True), # (batch_size, 4, 4, 1024)
        upsample(512, 4, apply_dropout=True), # (batch_size, 8, 8, 1024)
        upsample(512, 4), # (batch_size, 16, 16, 1024)
        upsample(256, 4), # (batch_size, 32, 32, 512)
        upsample(128, 4), # (batch_size, 64, 64, 256)
        upsample(64, 4), # (batch_size, 128, 128, 128)
    ]

    initializer = tf.random_normal_initializer(0., 0.02)
    last = tf.keras.layers.Conv2DTranspose(
        OUTPUT_CHANNELS,
        4,
        strides=2,
        padding='same',
        kernel_initializer=initializer,
        activation='tanh'
    ) # (batch_size, 256, 256, 3)

    x = inputs

    # Downsampling through the model
    skips = []
    for down in down_stack:
        x = down(x)
        skips.append(x)

    skips = reversed(skips[:-1])
```

```

# Upsampling and establishing the skip connections
for up, skip in zip(up_stack, skips):
    x = up(x)
    x = tf.keras.layers.concatenate([x, skip])

x = last(x)
return tf.keras.Model(inputs=inputs, outputs=x)

def generator_loss(disc_generated_output, gen_output, target):
    gan_loss = loss_object(tf.ones_like(disc_generated_output), disc_generated_output)

    # Mean absolute error
    l1_loss = tf.reduce_mean(tf.abs(target - gen_output))

    total_gen_loss = gan_loss + (LAMBDA * l1_loss)

    return total_gen_loss, gan_loss, l1_loss

def build_discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)

    inp = tf.keras.layers.Input(shape=[256, 256, 3], name='input_image')
    tar = tf.keras.layers.Input(shape=[256, 256, 3], name='target_image')

    x = tf.keras.layers.concatenate([inp, tar]) # (batch_size, 256, 256, channels*2)

    down1 = downsample(64, 4, False)(x) # (batch_size, 128, 128, 64)
    down2 = downsample(128, 4)(down1) # (batch_size, 64, 64, 128)
    down3 = downsample(256, 4)(down2) # (batch_size, 32, 32, 256)

    zero_pad1 = tf.keras.layers.ZeroPadding2D()(down3) # (batch_size, 34, 34, 256)
    conv = tf.keras.layers.Conv2D(
        512,
        4,
        strides=1,
        kernel_initializer=initializer,
        use_bias=False
    )(zero_pad1) # (batch_size, 31, 31, 512)

    batchnorm1 = tf.keras.layers.BatchNormalization()(conv)

    leaky_relu = tf.keras.layers.LeakyReLU()(batchnorm1)

    zero_pad2 = tf.keras.layers.ZeroPadding2D()(leaky_relu) # (batch_size, 33, 33, 512)

    last = tf.keras.layers.Conv2D(
        1, 4, strides=1, kernel_initializer=initializer
    )(zero_pad2) # (batch_size, 30, 30, 1)

    return tf.keras.Model(inputs=[inp, tar], outputs=last)

def discriminator_loss(disc_real_output, disc_generated_output):
    real_loss = loss_object(tf.ones_like(disc_real_output), disc_real_output)

    generated_loss = loss_object(tf.zeros_like(disc_generated_output), disc_generated_output)

    total_disc_loss = real_loss + generated_loss

    return total_disc_loss

def generate_images(model, test_input, tar, image_index):
    prediction = model(test_input, training=True)
    plt.figure(figsize=(15, 15))

```

A. Quelltexte

```
display_list = [test_input[0], tar[0], prediction[0]]
title = ['Input Image', 'Ground Truth', 'Predicted Image']

for i in range(3):
    plt.subplot(1, 3, i + 1)
    plt.title(title[i])
    # Getting the pixel values in the [0, 1] range to plot.
    plt.imshow(display_list[i] * 0.5 + 0.5)
    plt.axis('off')

plt.savefig('results/' + str(image_index) + '.jpg')

@tf.function
def train_step(input_image, target, step):
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        gen_output = generator(input_image, training=True)

        disc_real_output = discriminator([input_image, target], training=True)
        disc_generated_output = discriminator([input_image, gen_output], training=True)

        gen_total_loss, gen_gan_loss, gen_l1_loss =
            generator_loss(disc_generated_output, gen_output, target)
        disc_loss = discriminator_loss(disc_real_output, disc_generated_output)

        generator_gradients = gen_tape.gradient(
            gen_total_loss, generator.trainable_variables
        )
        discriminator_gradients = disc_tape.gradient(
            disc_loss, discriminator.trainable_variables
        )

        generator_optimizer.apply_gradients(
            zip(generator_gradients, generator.trainable_variables)
        )
        discriminator_optimizer.apply_gradients(
            zip(discriminator_gradients, discriminator.trainable_variables)
        )

    with summary_writer.as_default():
        tf.summary.scalar('gen_total_loss', gen_total_loss, step=step // 1000)
        tf.summary.scalar('gen_gan_loss', gen_gan_loss, step=step // 1000)
        tf.summary.scalar('gen_l1_loss', gen_l1_loss, step=step // 1000)
        tf.summary.scalar('disc_loss', disc_loss, step=step // 1000)

def fit(train_ds, test_ds, steps):
    example_input, example_target = next(iter(test_ds.take(1)))
    start = time.time()

    for step, (input_image, target) in train_ds.repeat().take(steps).enumerate():
        if step % 1000 == 0:
            display.clear_output(wait=True)

            if step != 0:
                print(f'Time taken for 1000 steps: {time.time() - start:.2f} sec\n')

            start = time.time()

            generate_images(generator, example_input, example_target, step.numpy() // 1000)
            print(f"Step: {step // 1000}k")

            train_step(input_image, target, step)

        # Training step
        if (step + 1) % 10 == 0:
            print('.', end='', flush=True)
```

```

# Save (checkpoint) the model every 5k steps
if (step + 1) % 5000 == 0:
    checkpoint.save(file_prefix=checkpoint_prefix)

if __name__ == '__main__':
    config = ConfigProto()
    config.gpu_options.allow_growth = True
    session = InteractiveSession(config=config)

    # Adjust this value to the number of training images
    BUFFER_SIZE = 400

    # The batch size of 1 produced better results
    # for the U-Net in the original pix2pix experiment
    BATCH_SIZE = 1

    # Each image is 256x256 in size
    IMG_WIDTH = 256
    IMG_HEIGHT = 256

    PATH = '../PIX2PIX/images/combined/candles/'
    train_dataset = tf.data.Dataset.list_files(PATH + 'train/*.png')
    train_dataset = train_dataset.map(
        load_image_train, num_parallel_calls=tf.data.AUTOTUNE
    )
    train_dataset = train_dataset.shuffle(BUFFER_SIZE)
    train_dataset = train_dataset.batch(BATCH_SIZE)

    try:
        test_dataset = tf.data.Dataset.list_files(str(PATH + 'test/*.png'))
    except tf.errors.InvalidArgumentError:
        test_dataset = tf.data.Dataset.list_files(str(PATH + 'val/*.png'))
    test_dataset = test_dataset.map(load_image_test)
    test_dataset = test_dataset.batch(BATCH_SIZE)

    OUTPUT_CHANNELS = 3

    generator = build_generator()

    LAMBDA = 100

    loss_object = tf.keras.losses.BinaryCrossentropy(from_logits=True)

    discriminator = build_discriminator()

    generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
    discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)

    checkpoint_dir = './training_checkpoints'
    checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
    checkpoint = tf.train.Checkpoint(
        generator_optimizer=generator_optimizer,
        discriminator_optimizer=discriminator_optimizer,
        generator=generator,
        discriminator=discriminator
    )

    log_dir = "logs/"

    summary_writer = tf.summary.create_file_writer(
        log_dir + "fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    )

    fit(train_dataset, test_dataset, steps=40000)

```

A. Quelltexte

```
# Restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

# Run the trained model on a few examples from the test set
index = 1000
for inp, tar in test_dataset.take(5):
    generate_images(generator, inp, tar, index)
    index = index + 1
```

2. Python-Script zur Generierung von JPEG-Bilddateien aus NDJSON-Informationen

```
import os
from functools import reduce
import tensorflow as tf
import json
from PIL import Image, ImageDraw

MAX_JPGS = 5000

dataset = tf.data.TextLineDataset(['../full_simplified_car.ndjson'])

for index, line in enumerate(dataset):
    if index % 1000 == 0:
        print(f'{index} of {MAX_JPGS}')
    jsonLine = json.loads(line.numpy())
    drawing = jsonLine['drawing']

    im = Image.new('RGB', (256, 256), (255, 255, 255))
    draw = ImageDraw.Draw(im)

    # normalise coords to center the drawing
    x_coords = reduce(lambda a, b: a + b, [stroke[0] for stroke in drawing])
    displacement_x = int((256 - min(x_coords) - max(x_coords)) / 2)
    y_coords = reduce(lambda a, b: a + b, [stroke[1] for stroke in drawing])
    displacement_y = int((256 - min(y_coords) - max(y_coords)) / 2)

    for stroke in drawing:
        draw.line(list(zip(
            [x + displacement_x for x in stroke[0]],
            [y + displacement_y for y in stroke[1]]
        )), fill=0)

    im.save(os.path.join('cars', str(index) + '.jpg'), 'JPEG')

    with open(os.path.join('json', str(index) + '.json'), 'w') as file:
        file.write(json.dumps(drawing))

    if index == MAX_JPGS:
        break
```

3. Blender-Python-Script zur Generierung von Tischen

```
import bpy
import math
import random

obj_filepath = '/home/sberger/blender/tables/obj/random_table{}.obj'
render_filepath = '/home/sberger/blender/tables/rendered/random_table{}_render{}.jpg'

for tableIndex in range(0, 50):
    # add table top
    tableWidth = random.random() * 8 + 6
    tableHeight = random.random() * .5 + .1
    tableDepth = random.random() * 4 + 4
    legLength = random.random() * 4 + 1
    bpy.ops.mesh.primitive_cube_add(
        size=1, enter_editmode=False, location=(0, tableHeight / 2 + legLength, 0)
    )
    bpy.context.object.name = 'Table'
    bpy.ops.transform.resize(
        value=(tableWidth, tableHeight, tableDepth),
        orient_type='GLOBAL',
        orient_matrix=((1, 0, 0), (0, 1, 0), (0, 0, 1)),
        orient_matrix_type='GLOBAL',
        constraint_axis=(False, True, False),
        mirror=True, use_proportional_edit=False,
        proportional_edit_falloff='SMOOTH',
        proportional_size=1,
        use_proportional_connected=False,
        use_proportional_projected=False
    )
    bpy.context.active_object.data.materials.append(bpy.data.materials.get("Wood"))

    # add table legs
    legWidth = random.random() * .5 + .1
    legDisplacement = random.random() * 1 + legWidth / 2
    legDisplacementX = tableWidth / 2 - legDisplacement
    legDisplacementZ = tableDepth / 2 - legDisplacement

    for legIndex, coords in enumerate([
        [legDisplacementX, legDisplacementZ],
        [-legDisplacementX, legDisplacementZ],
        [legDisplacementX, -legDisplacementZ],
        [-legDisplacementX, -legDisplacementZ]
    ]):
        bpy.ops.mesh.primitive_cube_add(
            size=1, enter_editmode=False, location=(coords[0], legLength / 2, coords[1])
        )
        bpy.context.object.name = 'TableLeg' + str(legIndex)
        bpy.ops.transform.resize(
            value=(legWidth, legLength, legWidth),
            orient_type='GLOBAL',
            orient_matrix=((1, 0, 0), (0, 1, 0), (0, 0, 1)),
            orient_matrix_type='GLOBAL',
            constraint_axis=(False, True, False),
            mirror=True,
            use_proportional_edit=False,
            proportional_edit_falloff='SMOOTH',
            proportional_size=1,
            use_proportional_connected=False,
            use_proportional_projected=False
        )
        bpy.context.active_object.data.materials.append(bpy.data.materials.get("Wood"))

# export to wavefront obj format
bpy.ops.export_scene.obj(filepath = obj_filepath.format(tableIndex))
```

```
# rotate and render
bpy.ops.object.select_all(action='DESELECT')
bpy.data.objects['Table'].select_set(True)
for legIndex in range(0, 4):
    bpy.data.objects['TableLeg' + str(legIndex)].select_set(True)
bpy.ops.object.transform_apply(location=False, rotation=True, scale=False)
for renderIndex in range(0, 5):
    # rotate table
    bpy.ops.transform.rotate(
        value=math.pi / 5,
        orient_axis='Y',
        orient_type='GLOBAL',
        orient_matrix=((1, 0, 0), (0, 1, 0), (0, 0, 1)),
        orient_matrix_type='GLOBAL',
        constraint_axis=(False, True, False),
        mirror=True,
        use_proportional_edit=False,
        proportional_edit_falloff='SMOOTH',
        proportional_size=1,
        use_proportional_connected=False,
        use_proportional_projected=False
    )
    # render
    bpy.context.scene.render.filepath = render_filepath.format(tableIndex, renderIndex)
    bpy.ops.render.render(write_still = True)

#delete objects
bpy.ops.object.delete(use_global=False)
```