Master Thesis

Neural Radiance Fields

Autor:
Magnus Kaffenberger
(Mat. Nr.: 35124)

Prüfer:
Prof. Dr. Bernhard Eberhardt
Prof. Dr. Thomas Keppler

Eine Arbeit eingereicht
zum Erlangen des Grades
Master of Engineering

10. Mai 2021

Eidesstattliche Versicherung

Hiermit versichere ich, Magnus Kaffenberger, an Eides statt, dass ich die vorliegende Masterarbeit mit dem Titel

Neural Radiance Fields

selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der eidesstattlichen Versicherung und die prüfungsrechtlichen Folgen (§23 Abs. 2 Master-SPO der Hochschule der Medien Stuttgart) sowie die strafrechtlichen Folgen (gem. § 156StGB)) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

Unterschrift: _____

Datum: _____

HOCHSCHULE DER MEDIEN

Zusammenfassung

Audiovisuelle Medien
Fakultät Electronic Media

Master of Engineering

Neural Radiance Fields

von Magnus Kaffenberger

Ziel dieser Arbeit ist es eine Einführung in die zu Beginn des Jahres 2020 neu entstandene Technik der neuronalen Radianz Felder zu geben. Die Idee der neuronalen Radiance Fields (NeRF) folgt dem Trend der Verbidung von bildbasierten und physik-basierten Methoden zur computergestützte Generierung von Bildern. NeRF ist dabei hauptsächlich eine bildbasierte Technik die mit Hilfe einer physikalisch basierten Lossfunction ein neuronales Netz optimiert. Die Grundlage ist eine Sammlung von Fotos einer Szene oder eines Objekts und der darin vorliegenden Informationen. Die aufgenommenen Bilder dienen als ”Ground Truth”für die Optimierung eines Multi Layer Perceptron (Mehrlagiges Perzeptron), welches Dichte und Farbwert abhängig einer Position 3D und Blickrichtung 2D modelliert. Das neuronale Netz lernt eine Beschreibung des von der Szene eingenommenen Volumenens. Mehrere Schätzungen des mehrlagigen Perzeptron werden entlang eines Sichtstrahls mithilfe von “Volume Rendering“ zu einem Bild zusammen gefügt, welches mit der ”Ground Truth”verglichen werden kann. Es werden zwei Erweiterungen betrachtet, eine um sich bewegende oder deformierende Teile der Szene zu beschreiben, und ein Verfahren zum Schätzen der Kameraparameter während des Optimierungsvorgangs.

Experimente mit den auf GitHub veröffentlichten Implementierungen haben den experimentellen Stand des Verfahren deutlich gemacht, der Vergleiche durch fehlende Vereinheitlichung erschwert.

HOCHSCHULE DER MEDIEN

Abstract

Audiovisuelle Medien
Fakultät Electronic Media

Master of Engineering

Neural Radiance Fields

by Magnus Kaffenberger

The aim of this paper is to give an introduction to the new technique of neural radiance fields that emerged at the beginning of 2020. The idea of neural radiance fields (NeRF) follows the trend of combining image-based and physics-based methods for computer-aided generation of images. NeRF is mainly an image-based technique that optimizes a neural network using a physics-based loss function. The basis is a collection of photographs of a scene or object and the information present in them. The captured images serve as ”ground truth” for the optimization of a multi-layer perceptron, which models density and color value depending on a position in 3D and viewing direction in 2D. The neural network learns a description of the volume occupied by the scene. Multiple estimates of the multilayer perceptron are merged along a line of sight using volume rendering to form an image that can be compared to the ground truth. Two extensions are considered, one to describe moving or deforming parts of the scene, and secondly a method for estimating camera parameters during the optimization process.

Experiments with implementations published on GitHub have shown their experimental state that makes comparisons difficult due to lack of standardization.

Inhaltsverzeichnis

Kapitel 1

Mathematische Grundlagen

Zu Beginn wird erklärt, was ein Koordinatensystem ist und wie es hilft Objekte im Raum relativ zueinander zu positionieren. Gefolgt von Transformationen welche beschreiben wie die Anordnung von Objekten in einem Koordinatensystem beschrieben und geändert werden können. Abschließend werden homogene Koordinaten eingeführt, welche die Berechnung von Transformationen mit einem Computer erleichtern.

1.1 Koordinatensysteme

Koordinatensysteme werden verwendet, um die Position eines Punktes relativ zu einer Referenz anzugeben. Im 2D kartesischen Koordinatensystem wird jeder Punkt durch eine geordnete Menge in Form von zwei Koordinaten dargestellt. (x, y) Dabei beschreibt X und Y einen mit Vorzeichen dotierten Abstand eines Punktes zu zwei senkrecht aufeinander stehenden Achsen, genannt "X-Achse" und "Y-Achse". Da die Reihenfolge der Koordinaten nicht getauscht werden kann, ist die Menge geordnet. Das heißt ein Punkt an der Stelle (y, x) entspricht nicht einem Punkt an der Stelle (x, y)!

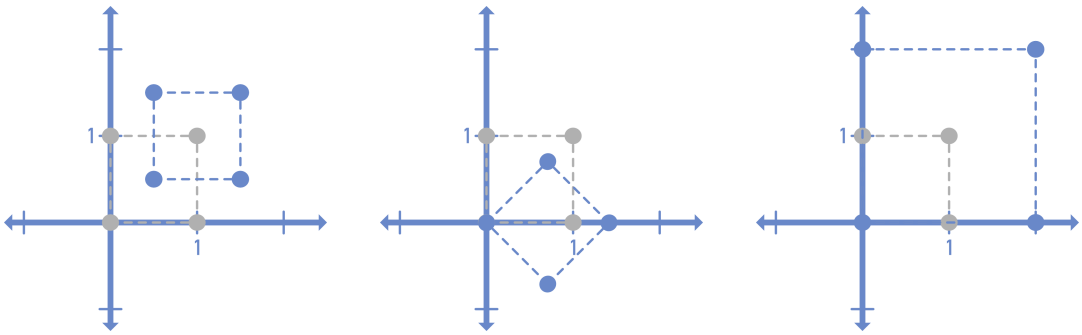
An der Position $(0, 0)$ schneiden sich beide Achsen, sie wird als Ursprung bezeichnet.

In drei Dimensionen hat ein kartesisches 3D-Koordinatensystem drei zueinander senkrecht stehende Achsen (X-Achse, Y-Achse und Z-Achse). Hier wird jeder Punkt durch ein Triple (x, y, z) dargestellt.

In welche Richtung (oben, rechts, vorne) die jeweilige Achse zeigt, ist dem Anwender überlassen und muss anfänglich definiert werden. Abhängig von der Verwendung werden die Achsen von 3d kartesischen Koordinatensystemen in unterschiedlichen Richtungen definiert. Eine Anwendung, die für technisches Zeichnen entwickelt wurde, definiert hergeleitet von einem Zeichentisch:

- Die X-Achse entlang der vor einem liegenden Seite des Zeichenblatts
- Die Y-Achse entlang der vom Betrachter weg zeigenden Blattkante.
- Die Z-Achse als vom Blatt weg zeigend, in den Raum hinein.

Auf einem Computermonitor würde die X-Achse entlang der Bildschirmbreite und die Y-Achse entlang der Bildschirmhöhe gezeigt werden. Damit steht die resultierende Z-Achse im Vergleich zum Betrachter nicht mehr in die Höhe, sondern horizontal vom Bildschirm weg in Richtung des Betrachters.



(A) Translation (B) Rotation (C) Skalierung

ABBILDUNG 1.1: Affine Transformationen erhalten parallele Linien.

Viele 3D-Programme mit einem Viewport wählen die X-Achse entlang dem Horizont der Szene, die Y-Achse vom Boden weg in die Raumhöhe zeigend, und die Z-Achse in den Raum entlang der Blickrichtung des Betrachters.

In der Computervision wird ein Kamerabild aufgenommen und Zeilenweise von oben nach unten durchlaufen. Daraus folgend zeigt die Y-Achse entlang der Bildzeilen nach unten. Die X-Achse entlang der Bildspalten nach rechts und die Z-Achse entlang der optischen Achse der Kamera in den Raum vom Betrachter weg.

Im Fall von OpenGL hingegen, zeigt die Y-Achse nach oben, X-Achse nach rechts und die positive Z-Achse zum Betrachter hin.

Wenn Angaben von Koordinaten zwischen Anwendungen ausgetauscht werden, muss dabei die Orientierung der Basisvektoren beachtet und eventuell angepasst werden.

1.2 Transformationen

Im kartesischen Koordinatensystem werden Translation, Rotation und Skalierung wie folgend berechnet:

$$\text{Translation:} \quad v' = v + t \qquad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} \tag{1.1}$$

Kartesische Koordinaten geben ein eins-zu-eins Verhältnis zwischen Zahl und Form. Werden die Koordinaten einer Form verschoben, ändert sich die Geometrie der Form. Wenn $P(x,y)$ ein Punkt einer Form ist, kann dieser durch die Operation $x' = x + 3$ an die Stelle $P'(x',y)$ um drei Einheiten entlang der Richtung der X-Achse verschoben werden. Das gleiche gilt für die Y-Achse, $y' = y + 1$ verschiebt dabei um eine Einheit zum Punkt $P'(x,y')$, der sich eine Einheit weiter oben befindet. Wenn die gleiche Transformation auf jeden Punkt eines Objekts angewandt wird, verschiebt sich das Objekt als Ganzes.

$$\text{Skalierung:} \quad v' = St \quad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{1.2}$$

Eine Skalierung verringert oder vergrößert die Distanz zwischen den Punkten des Objekts relativ zu einem Referenzpunkt, meist dem Ursprung. Dabei wird der Koordinatenvektor jedes Punktes komponentenweise mit einem Skalar multipliziert.

$$\text{Rotation in 2D:} \quad v' = R_\Theta v \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos\Theta & -sin\Theta \\ sin\Theta & cos\Theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1.3}$$

Als intrinsische Rotation ausgedrückt:

$$R = R_z(\alpha)R_y(\beta)R_x(\gamma) = \begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\beta & 0 & sin\beta \\ sin\beta & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\gamma & -sin\gamma \\ 0 & sin\gamma & cos\gamma \end{bmatrix} \tag{1.4}$$

Eine Rotation bewegt alle Punkte eines Objekts in einer Kreisbahn um den Ursprung des Koordinatensystems. Der Rotationsmittelpunkt bleibt dabei Unverändert. Eine beliebige Rotation um den Ursprung kann durch eine Kombination von Rotationen um die Hauptachsen abgebildet werden.

Da eine Skalierung oder Rotation eine Matrixmultiplikation ist, kann eine Rotation und Skalierung in einem Schritt berechnet werden, eine Translation ist hingegen eine Addition und benötigt daher extra Operationen.

Homogene Koordinaten ermöglichen es mehrere Transformationen mit einer Matrixmultiplikation auszuführen.

1.3 Affine Transformationen

Eine affine Transformation beschreibt eine Verschiebung, Rotation, Skalierung oder Scherung eines Objekts. Dabei bleiben ursprünglich parallele Linien, in einer Gerade liegende Punkte und Längenverhältnisse unverändert. Affine Transformationen bewahren sowohl die Kollinearität als Abstandsverhältnisse und sind damit eine spezielle Form der perspektivischen Projektion, bei der kein Teil von R^3 in das Unendliche abgebildet wird. Ein Dreieck kann durch affine Transformationen in jedes beliebige Dreieck transformiert werden.

Lineare Abbildungen zwischen Vektorräumen: Wir nehmen an, dass V und W Vektorräume sind, x und y Elemente des Vektorraums V und Phi eine Abbildung von V nach W. Die Abbildung Φ ist linear, falls für beliebige Skalare α und β gilt, dass

$$\Phi(\alpha x + \beta y) = \alpha\Phi(x) + \beta\Phi(y) \ .$$

Es gilt folgende Äquivalenz:

1. A ist eine affine Abbildung

2. es gibt eine lineare Abbildung Φ und ein Vektor b in W mit $A(x) = \Phi(x) + b$

ist V und W endlichdimensional, so ist Φ eine mxn-Matrix.

1.4 Homogene Koordinaten

$$\overrightarrow{y} = A\overrightarrow{x} \tag{1.5}$$

In der Computergrafik wird, um einfacher Transformationen in drei Dimensionen ausführen zu können, eine weitere Dimension eingeführt. Im Vergleich zu kartesischen oder affinen Koordinaten werden in 2D aus x, y, die homogenen Koordinaten u, v, w. Dabei werden Vektoren durch den Eintrag einer "1" erweitert und an Matrizen eine Reihe Nullen angehängt.

In zwei Dimensionen:

$$u/w, v/w, 1 = x, y, 1$$

$$\begin{bmatrix} x \\ y\gamma \end{bmatrix} \rightarrow \begin{bmatrix} x/\gamma \\ y/\gamma \end{bmatrix} \quad \text{mit } \gamma \neq 0 \tag{1.6}$$

Homogene Koordinaten ermöglichen mit einer Matrixmultiplikation alle affinen Transformationsarten auf einmal auszuführen.

Kapitel 2

Grundlagen aus der Computer Vision

Das Sehen ermöglicht dem Menschen mit einer komplexen Umgebung zu interagieren. In Lichtgeschwindigkeit kommen Informationen über die Begebenheiten der Umwelt im Auge an und werden hoch parallelisiert[1] im Gehirn verarbeitet. Das ermöglicht dem Menschen eine dreidimensionale Wahrnehmung seiner Umgebung und ist damit Grundlage für das Verständnis und die Beschreibung der Umwelt. In der Computer Vision werden die einfachen Prinzipien von Strahlensatz und dem Zusammenhang von Abbildungen der dreidimensionalen Umwelt auf einen Bildsensor angewandt um Computerprogrammen einen *Sehsinn* zu geben.

2.1 Perspektivische Projektion

Perspektivische Projektion (von lateinisch proicere „hinauswerfen, hinwerfen“) beschreibt wie Strahlen aus der dreidimensionalen Umwelt auf eine zweidimensionale Ebene fallen. Sowohl im Auge als auch in Kameras findet eine perspektivische Projektion des Gesehenen durch eine Optik auf eine Bildebene statt. Obwohl die menschliche Netzhaut und der Kamerasensor Objekte direkt nur in ihren zwei Dimensionen Ausbreitung als Projektion wahrnehmen, kann ein Mensch daraus Schlüsse über die dreidimensionale Ausbreitung ziehen. Das geschieht über eine Veränderung des Betrachtungspunkts (Position der Kamera) und der daraus resultierenden Veränderung der Abbildung. Aus dem Unterschied zwischen zwei Bildern kann ein Mensch Informationen über die Tiefe und Ausbreitung der aufgenommenen Szenen ziehen.

In der Computergrafik gibt es Verfahren wie *Structure-from- Motion* (SfM), die aus Bilddaten mit zugehörigen Kameradaten eine dreidimensionale Szene rekonstruieren können. Die dafür benötigten Kameradaten sind die *extrinsischen* und *intrinsischen* Kameraparameter. Die extrinsischen Parameter beschreiben die Position und Blickrichtung einer Kamera, die intrinsischen Parameter beschreiben das optische System, das die Projektion aus dem dreidimensionalen Raum auf die zweidimensionale Bildebene vornimmt.

Eine Kamera projiziert einen Punkt aus einem 3D Koordinatensystem an der Koordinate (X, Y, Z) auf die 2D Bildebene am Punk (x, y).

Die Projektion eines dreidimensionalen Punktes auf einen Punkt in der Bildebene einer Kamera in homogenen Koordinaten berechnet sich aus:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \cong x = PX \tag{2.1}$$

Im folgenden werden wir versuchen für jede Kamara diese Parameter p_{ij} zu schätzen, um damit die sogenannten intrinsischen und extrinsischen Parameter zu berechnen.

2.1.1 Die Intrinsische und extrinsische Kameraparameter

Unter intrinsischen Parametern verstehen wir Brennweite, Verschränkung der Bildachsen (skew) und Mittelstrahl des optischen Systems. Bei einer Punktkamera sind die intrinsichen Parameter K wie in 2.2 definiert.

$$K = \begin{pmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.2}$$

Die Stauchung der Abbildung ist dabei abhängig von der Brennweite f der Kamera.

$$\tilde{x} = f\frac{X_c}{Z_c} \quad \tilde{y} = f\frac{Y_c}{Z_c} \tag{2.3}$$

f_x und f_y haben in einer Punktkamera den gleichen Wert f. Für eine reale Kamera wird entweder das Seitenverhältnis mit angegeben oder ein Wert in jeder Bildachse.

x_0 und y_0 ist ein Offset der optischen Achse auf der Bildebene. Die optische Achse verläuft vom Projektionszentrum C durch den *principal point* zu deutsch Hauptpunkt P

s ist der Skew-Faktor welcher ein Neigen der Bildebene relativ zur optischen Achse beschreibt.

$$K = \begin{pmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & s/f_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.4}$$

Die intrinsischen Parameter mit den extrinsischen

$$x = PX = \begin{bmatrix} f & 0 & 0 & P_x \\ 0 & f & 0 & P_y \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.5}$$

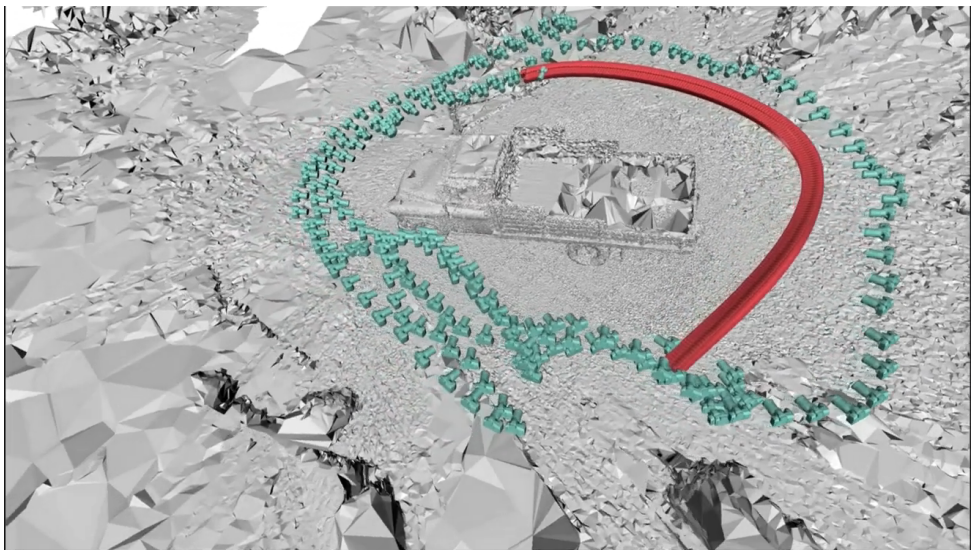


Abbildung 2.1: In Grün: Kamereapositionen im Aufgenommenen Datensatz. In Rot: Position neuer, frei synthetisierten Ansichten der Szene.

2.2 Kameralokalisierung durch Structure-from-Motion

Structure-from-Motion (SfM), zu deutsch Struktur oder Form aus Bewegung, beschreibt Verfahren bei denen, durch Veränderung der Ansicht die Ausmaße eines Objekts oder einer Szene bestimmt werden können. Dabei machen sich alle Verfahren die Eigenschaften von Strahlensatz und der Möglichkeit der relativen Positionsbestimmung durch Triangulation zu eigen.

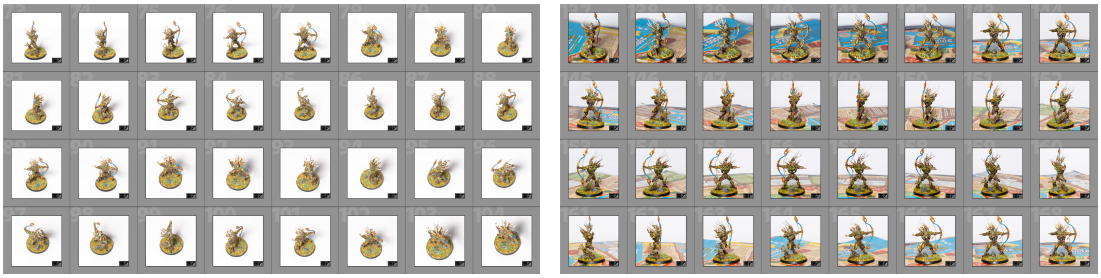
Bei SfM Verfahren wird keine physische (durch Berührung) Abtastung eines Objekts zur Bestimmung von Gestalt und Oberfläche benötigt. Eine Sammlung von zweidimensionalen Abbildungen ist ausreichend, um Rückschlüsse über die dreidimensionale Ausbreitung einer Szene zu machen.

Wenn die Aufnahmepunkte der Abbildungen relativ zu einander bekannt sind, lassen sich durch Abstände zwischen den Positionen der Kameras und erkannten Merkmalen im jeweiligen Bildraum Längen im dreidimensionalen Raum errechnen.

2.2.1 Fotografische oder synthesische Aquise der Bilder

Für die Erstellung eines Datensatz wird zuerst eine Sammlung von Bildern aus möglichst vielen Blickrichtungen erstellt.

Bei der Aufnahme muss auf eine gleichmäßige, sich nicht ändernde Beleuchtung und genügend Anhaltspunkte für die sich ändernde Perspektive zwischen den einzelnen Bildern geachtet werden. Ein einfarbiger Hintergrund bietet keine Merkmale, die Rückschlüsse auf die räumliche Positionierung der Kamera geben können.



(A) Weißer Hintergrund

(B) Farbiger Hintergrund bietet mehr Merkmale zur Erkennung.

ABBILDUNG 2.2: Zwei Datensätze auf unterschiedlichem Hintergrund aufgenommen.

Eine Ausnahme hierbei sind synthetisch, mit Hilfe eines Renderers erstellte Datensätze, da hierbei die Lokalisation der virtuellen Kamera gegeben ist, und diese direkt aus der virtuellen Szene entnommen werden kann.

Die Bilder alleine sind nicht genug, um ein *Neural Radiance Fields* (NeRF) zu lernen, da die räumliche Positionierung der Kamera relativ zur aufgenommenen Szene, aus den Bildern alleine nicht hervorgeht.

2.2.2 Bestimmung der Kameraposition durch SfM

Ziel ist es, aus dem Bildinhalt alleine auf die relative Positionierung der Kamera bei der Aufnahme der einzelnen Bilder zurück zu schließen. Das wird durch das Aufnehmen eines Datensatz mit einer mit der Hand gehaltenen Kamera ermöglicht. Somit ersetzt SfM einen kalibrierten Versuchsaufbau, bei dem die Kamerpositionen durch Vermessen des Aufbaus bestimmt wird.

SfM ist durch einfache Handhabung (Handelsübliche Fotokamera ist ausreichend) und flexible Positionierung (aus der Hüfte und vor Ort), zum Standard der Kameralokalisierung in der Photogrammetrie geworden.

2.2.3 Erkennen und Beschreiben von Merkmalen in Bildern

Eine Methode die häufig verwendet wird, um Merkmale in Bildern zu erkennen und zu Beschreiben ist SIFT [2]. Das Erkennen und Vergleichen von Merkmalen ist ein fester Bestandteil traditioneller Photogrammetrie-Verfahren wie AliceVision/Meshroom [3] und COLMAP [4, 5], beide haben SIFT implementiert.

SIFT steht für *Scale invariant feature transform*. Es ist ein Verfahren zur Beschreibung von lokalen Merkmalen in einem Bild durch so genannte *feature vectors*. Diese *feature vectors* werden verwendet, um Merkmale in verschiedenen Bildern eines Bilddatensatz wiederzufinden. Die Korrespondenz zwischen verschiedenen Bildern ist nötig, um im Verfahren der Photogrammetrie vom Bildraum auf den gemeinsamen Szenenraum schließen zu können. Dabei wird der Inhalt eines einzelnen Bildes auf wenige Punkte mit ihren Beschreibungen reduziert. Diese wenige Beschreibungen lassen sich einfacher mit anderen Bildern im Datensatz untereinander vergleichen, um gemeinsame Merkmale zu finden.

Werden mehrere solcher korrespondierenden Punkte in einem Bildpaar gefunden, kann daraus die relative Positionierung der Kameras und der Merkmale in einer gemeinsamen Szene bestimmt werden.

Das Merkmal eines Punktes wird durch die Art und Weise, wie sich der Farbwert in den umliegenden Pixeln ändert, beschrieben. Jedem *feature point* wird also ein *feature descriptor*, zugeordnet. Der *feature point* ist eine Bildkoordinate und der *feature descriptor* ein Vektor mit 128 Einträgen [6]. Dieser Vektor kodiert den Gradient vieler kleiner Flächen um den *feature point* herum.

Es ist sehr schwer, für jeden Pixel eines Bildes den korrespondierenden Pixel in einem anderen Bild wiederzufinden, wenn durch Perspektivänderung die Erscheinung eines Merkmals sich ändert. SIFT filtert lokale Punkte mit ihren umliegenden Merkmalen so heraus, dass diese auch nach affinen Transformationen des Bildraums wiedergefunden werden können. Die Beschreibung und Lokalisierung eines Merkmals ist invariant zu allen affinen Transformationen.

Abbildung 2.3: Der Buchstabe *R* kann in beiden Bildern lokalisiert werden, auch wenn diese aus unterschiedlichen Perspektiven aufgenommen wurden. Die Fähigkeit ist für Photogrammetrie essentiell, da ein Objekt von vielen verschieden Blickwinkeln aufgenommen wird [7].

Die Invarianz zu Rotation wird dadurch erreicht, dass der Gradient für den *feature vektor* in 45°Schritten um den *feature point* gedreht gebildet wird. Invarianz zu Skalierung wird durch ein Vorgehen erreicht, das an eine Bildpyramide angelehnt ist. Dabei wird bei Verjüngung der Pyramide nicht nur die Bildauflösung reduziert, sondern auch mit einem variablen Sigma das Bild weichgezeichnet. Wird die Reduzierung der Bildauflösung durch das Filtern mit einem *Gauß-Kernel* [8] vorgenommen, spricht man von einer *Gauß-Pyramide*.

Die Gauß-Bildpyramide erweitert den Bildraum um eine Dimension, in der ein *Gaussian Blur* mit variablem σ als Parameter für die *Kernelbreite*, auf den Bildraum angewandt wird. Eine Erweiterung des Bildraums abhängig des Parameter σ, wird *Skalenraum* genannt. Bei zunehmender *Skalenkoordinate* im *Skalenraum*, nimmt der Blur-Faktor des Gauß-Kernels zu. Der Gauß-Skalenraum $\mathcal{G}$ mit drei Vektoren, x, y Richtung im Bildraum und σ als Parameter des gaußförmigen Filterkernel wird wie folgt gebildet 2.6 [9, 7]:

$$\mathcal{G}(x, y, \sigma) = \left(F * H^{G,\sigma}\right)(x, y) \tag{2.6}$$

In 2.6 gilt: $H^{G,\sigma} \equiv G_\sigma(x, y)$ und $\sigma \geq 0$, F ist eine Funktion, die auf die Koordinaten (x, y, σ) im Skalenraum angewandt wird. Die Hesse-Matrix H des

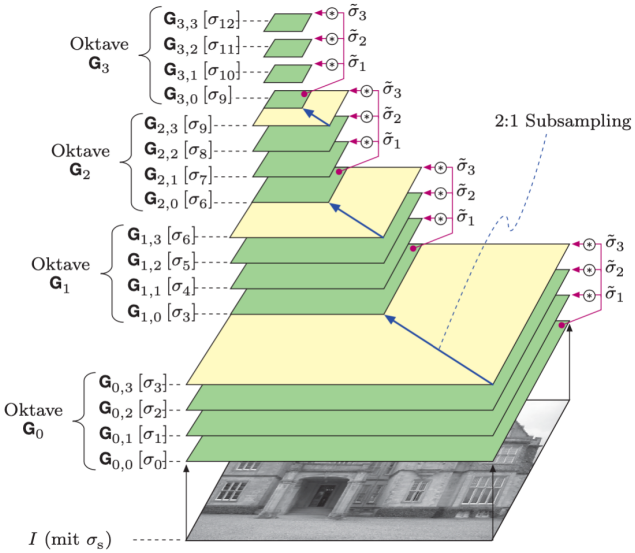


Abbildung 2.4: Ein hirarchischer Gauß-Skalenraum mit vier Oktaven $[G_0, G_1, G_2, G_3]$ die jeweils aus vier Schichten $[G_{0,0}, G_{0,1}...G_{3,3}]$ mit änderndem σbestehen [9]. Um ein Merkmal zu bilden, wird zwischen den einzelnen Schichten der Gradient $\tilde{\sigma}$ bestimmt .

Bildraums, entspricht dem Gradienten der vom Skalenenraum gebildeten Dimension.

Das ursprüngliche Bild, in dem Merkmale gefunden werden, entspricht einem Schnitt auf Höhe $(x, y, 0)$ durch den Skalenraum.

$$\mathcal{G}(x, y, 0) = \left(F * H^{G,0}\right)(x, y) = (F * \delta)(x, y) = F(x, y). \tag{2.7}$$

Weitere interessante Stellen im Skalenraum lassen sich an Stellen des Nulldurchgangs von F abhängig von σ finden. Jeder σ Wert entspricht einem Faktor mit dem das Bild weichgezeichnet wird, dadurch wird die Invarianz zur Größe eines Merkmals gewonnen. Merkmale mit niedrigen Frequenzen im Bildraum ähneln sich in Schichten des Skalenraums mit größerem σ.

2.2.4 Feature Matching

Hat man durch einen *Feature* Erkennungsalgorithmus in einem Bilddatensatz Merkmale erkannt und beschrieben, werden diese Merkmale anhand ihrer Beschreibungen verglichen und Bildpaare gebildet. Da die Daten sensorisch entstanden sind und sich einige Merkmale durch Zufall ähneln, müssen die Merkmale der Bildpaare gefiltert werden.

Random Sample Consenus (RANSAC) ist ein iteratives Verfahren, um Merkmale auszufiltern. Dafür wird aus zufälligen Samples aus einem Datensatz ein Modell des Gesamten aufgestellt und damit ein Fehlerwert für die übrigen Samples berechnet.

Ein einfaches Beispiel ist das Finden einer Linie, die eine Abhängigkeit einer verrauschten Messung abbildet. Hat man eine Menge Punkte, kann man iterativ

Abbildung 2.5: Der erste Schritt ist das Finden von Merkmalen in jedem Bild. Mit SIFT werden in jedem Bild Merkmale extrahiert und gesammelt. Je größer der Kreis, desto größer σ im Skalenraum.

zufällige Punktpaare auswählen und den Abstand zwischen Linie und übrig gebliebenen Punkte berechnen. Die akkumulierten Abstände bewerten welches Punktpaar die beste Linie als Abbildung des Datensatzes bildet.

Für das Rekonstruieren einer Szene aus einem Bilddatensatz werden Merkmale in Bildpaaren gesucht, die dieselbe Stelle der Szene abbilden. Damit gibt es eine Verbindung zwischen den Pixelpositionen der Merkmale in den Bildern und Positionen im dreidimensionalen Raum der Szene. Stellt man nun ein Kameramodell auf, welches auf den physikalischen Gegebenheiten der Ausbreitung von Licht basiert, kann man dieses Modell auf den Bilddatensatz optimieren. Die Parameter, die gesucht werden, sind die intrinsischen und extrinsischen Kameraparameter. Die extrinsischen Parameter geben Aufschluss über die Positionierung des optischen System der Kamera im Raum und die intrinsischen Parameter über die Projektion durch das optische Zentrum auf die Bildebene.

2.2.5 Epipolar Geometry

Die Epipolarebene eines z.B. mit SIFT gefundenen Merkmalpaars und dem dazugehörigen Punkt im Szenenraum, verläuft in Weltkoordinaten (einer gemeinsamen 3D-Szene) durch drei Punkte. Diese sind durch die Positionen der Kameras und die Position des zu betrachtenden Merkmals gegeben. Ziel ist es damit die Beziehung zwischen einem Punkt x in einer Szene und den gefundenen Merkmalen $(\widetilde{x}1, \widetilde{x}_2)$ in einem Bildpaar zu beschreiben.

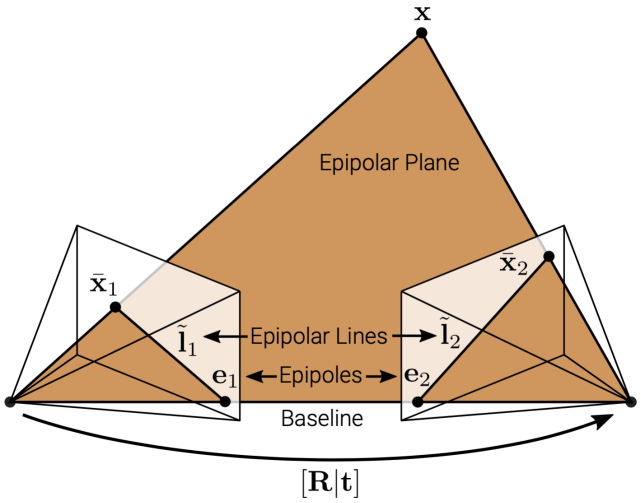


Abbildung 2.6: Epipolargeometrie beschreibt das Verhältnis zwischen Punkten im Raum und deren Abbildung in Bildebenen. Links Kamerea π_1, rechts Kamera π_2. [10]

In Abbildung 2.6 beschreibt $[R|t]$ die relative Ausrichtung zweier Kameras. Eine Punkt x in der Szene wird an Bildpunkt $\widetilde{x}_1$ von einer Kamera π_1 projiziert.

Eine zweite Kamera π_2 würde diesen Sichtstrahl nun als Gerade $\widetilde{l}_2$ durch das optische Zentrum von Kamera π_1 und dem Objektpunkt x in der Szene erkennen. Auch umgekehrt, ist $\widetilde{x}_2$ ein Featurepunkt in Kamera π_2, sieht die Kamera π_1 diesen Sichtstrahl durch π_2 und x als Gerade, welche auch $\widetilde{x}_2$ enthält.

Fallen nun die Bildpunkte $\widetilde{x}_1$ und $\widetilde{x}_2$ in der 3D-Szene zusammen auf den selben Objektpunkt $\boldsymbol{x}$, das heißt $(\widetilde{x}_0, \widetilde{x}_1)$ sind korrespondierende Featurepaare, so ist durch die beiden Kamerazentren von π_1 und π_2 zusammen mit $(\widetilde{x}_0 = \widetilde{x}_1)$ ein Dreieck definiert, die sogenannte epipolare Ebene.

Ist $\widetilde{y}$ nun ein beliebig anderer Featurepunkt, so ist (bei gleichbleibenden Aufnahmeorten) die Linie durch die beiden Kamerazentren, also die Basislinie in Abbildung 2.6, eine Fixgerade.

Das Abbild, oder der Schnitt, dieser Epipolarebene im jeweiligen Bild ist also eine Gerade als Schnitt zweier Ebenen (der Bildebene und der epipolaren Ebene zu einem korrespondierenden Featurepaar). Diese Schnittgerade in den zwei Bildern verläuft immer durch den Bildpunkt $\widetilde{x}_0$ und $\widetilde{x}_1$ des jeweilig anderen Kamerazentrums. Diese Punkte nennt man Epipole.

Ist also $\widetilde{x} = (\widetilde{x}_1, \widetilde{x}_2)^T$ ein Featurepunkt in Bild 1 und x ein korrespondierender dreidimensionaler Objektpunkt auf der "Richtung"x, in homogenen Koordinaten $(x_1, x_2, 1)^T$ so liegt dieser Bildpunkt auf der Richtung $(y_1, y_2, 1)$ in Bild 2.

Epipolarlinie und die Beziehung zwischen korrespondierenden Punkten ergibt die Beziehung:

$$\begin{pmatrix} x_2 & y_2 & 1 \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = 0.$$

Die Matrix $F = \left(f_{ij}\right)_{3\times 3}$ wird Fundamentalmatrix genannt und ihre Einträge f_{ij} können durch Bestimmung von mindestens 8 korrespondierenden Featurepaaren geschätzt werden (8-Punkt-Algorithmus). Ist andererseits F geschätzt, dann können die beiden Epipole e und e' durch die Eigenschaft

$$\mathbf{F}^{\mathrm{T}}\mathbf{e} = \mathbf{0} \qquad \text{und} \qquad \mathbf{F}\mathbf{e}' = \mathbf{0}$$

berechnet werden. Schließlich können auch die relativen extrinsischen Parameter des Kamerapaares berechnet werden, also Translation und Rotation der Kamerazentren zueinander. [11] [7]

2.2.6 Bundle Adjustment

Die Bündelblockausgleichung 2.8 minimiert den Projektionsfehler der Merkmale durch Optimierung der Kameraparameter. Dabei wird mit den extrinsischen und intrinsischen Kameraparametern π_i ein Punkt $\boldsymbol{x}^w$ aus dem Weltkoordinatensystem an die Stelle $\boldsymbol{x}^s$ im Bildkoordiantensystem (screen) projiziert und verglichen wie nah dieser am ursprünglichen Merkmal der Featureerkennung liegt $(\mathbf{x}^s_{ip} - \pi_i\left(\mathbf{x}^w_p\right))$.

- $\Pi = \{\pi_i\}$ ist die Projektionsmatrix aus intrinsischen und extrinsischen Parametern
- $\mathcal{X}_w = \{\mathbf{x}^w_p\}$ mit $\mathbf{x}^w_p \in \mathbb{R}^3$ sind die Merkmale in Weltkoordinaten
- $\mathcal{X}_s = \{\mathbf{x}^s_{ip}\}$ mit $\mathbf{x}^s_{ip} \in \mathbb{R}^2$ sind die Merkmale in Bildkoordinaten

$\mathbf{K}_i$ und $[\mathbf{R}_i \mid \mathbf{t}_i]$ sind die intrinsischen und extrinsischen Parameter von Kamera π_i. Während der Bündelausgleichung optimieren wir $\{(\mathbf{K}_i, \mathbf{R}_i, \mathbf{t}_i)\}$ und $\{\mathbf{x}^w_p\}$ gemeinsam.

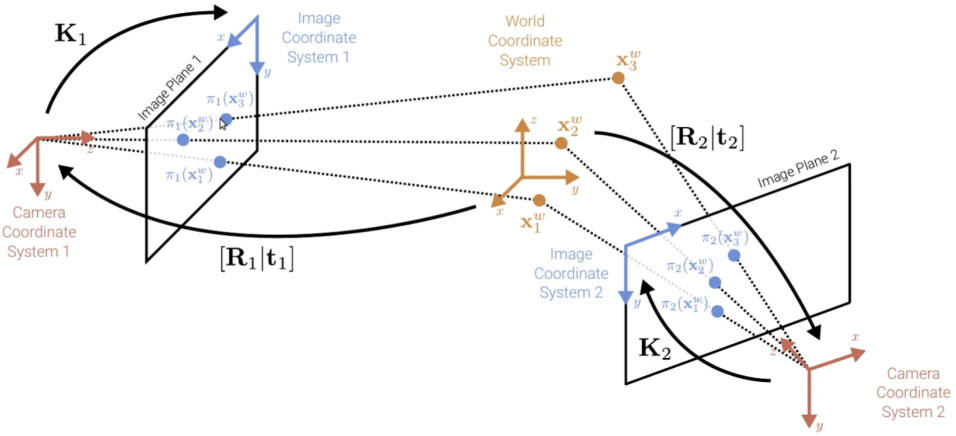


Abbildung 2.7: Grafik entnommen aus [10]

$$\Pi^*, \mathcal{X}^*_w = \underset{\Pi, \mathcal{X}_w}{\operatorname{argmin}} \sum_{i=1}^{N} \sum_{p=1}^{P} w_{ip} \left\| \mathbf{x}^s_{ip} - \pi_i\left(\mathbf{x}^w_p\right) \right\|_2^2 \tag{2.8}$$

Dabei gibt w_{ip} an, ob ein Punkt p in Bild i zu sehen ist und $\left(\mathbf{x}_p^w\right)$ ist die 3D-zu-2D Projektion des 3D Punktes $\S_p^w$ auf die 2D Bildebene der i=ten Kamera:

$$\pi_i\left(\mathbf{x}_p^w\right) = \begin{pmatrix} \tilde{x}_p^s / \tilde{w}_p^s \\ \tilde{y}_p^s / \tilde{w}_p^s \end{pmatrix} \quad \text{with} \quad \tilde{\mathbf{x}}_p^s = \mathbf{K}_i\left(\mathbf{R}_i\mathbf{x}_p^w + \mathbf{t}_i\right)$$

Falsche Zuordnungen werden durch die Anwendung der Bündelblockausgleichung aussortiert. Es gilingt nicht nur korrespondierende Feature Paare zu bestimmen, sondern auch die relative Positionen der Kameras zueinander, d.h. die sog. extrinsischen Parameter.

Das Modell für die Generierung von Kamerapositionsdaten basiert auf dem Prinzip der Epipolargeometrie und wird durch eine 3x3 Matrix, genannt Fundamentalmatrix als Beschreibung einer Kameraposition relativ zu den gesehen Merkmalen, repräsentiert.

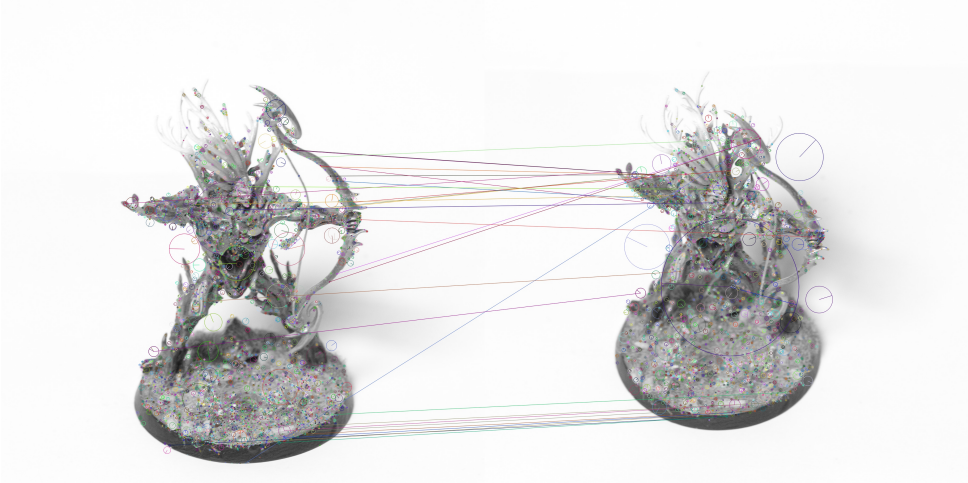


Abbildung 2.8: Zuordnung von Merkmalen aus zwei Aufnahmepositionen.

2.3 Szenenausbreitung

Ein bemerkenswertes Problem bei der Parametrisierung und Rekonstruktion einer 3D-Szene ist die Beschränkung der Ausmaße.

Eine nach innen gerichtete Szene, auch 360°-Szene genannt, ist ein Datensatz, der Bilder aller Seiten eines Objekts beinhaltet. Das setzt voraus, dass die Kameras Positionen eingenommen haben, von denen sie sich gegenseitig sehen können. Bei einem Datensatz der in einer Lightstage oder von einem Objekt auf einem Drehteller, kann man sich die Kamerapositionen auf einer imaginären (Hemi)sphere vorstellen, wie in Abbildung 2.9 dargestellt.

Ein Merkmal mit einer geringen Pixeldisparität, also im entfernten Hintergrund, führt entweder zu sehr großen oder sehr kleinen Parametern für die Position im Raum. Bei einer uniformen Abtastung führt das zu einer ungünstigen Verteilung, da potentiell viel leerer Raum betrachtet wird und die Auflösung abnimmt.

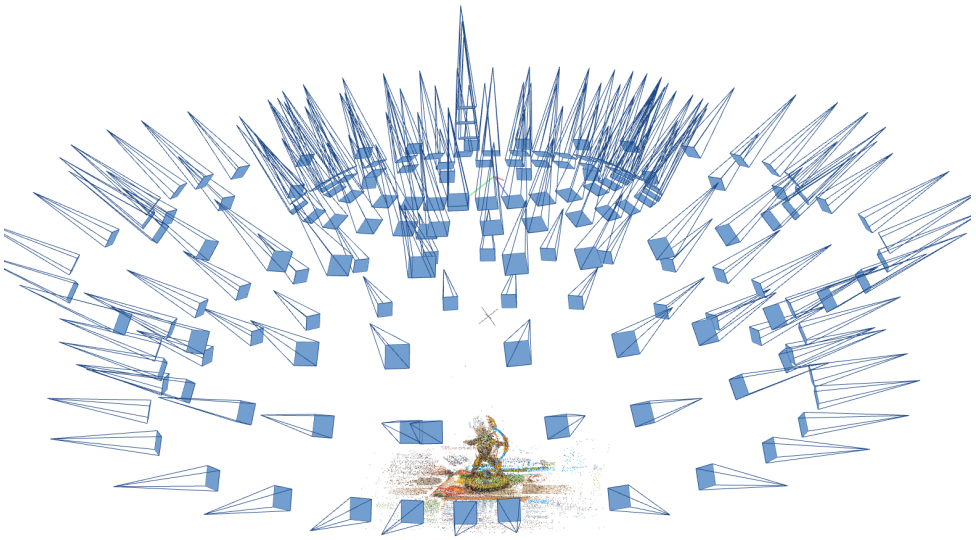


Abbildung 2.9: Der *Archer*-Datensatz ist eine nach innen gerichtet Szene. In Blau sind die Kamerapositionen visualisiert. Die Punktwolke der SfM Rekonstruktion ist farbig zu sehen.

Forward Facing (dt. vorwärts gerichtet) bedeutet, dass eine Ebene die Kameras von der Szene trennt. Das entspricht mehr eine Szene, die wie ein Foto mit 3D Effekt wirkt, aber nie die Rückseite von Objekten in der Szene sieht. Eine solche Szene kann nicht als vollständiges 3D-Objekt rekonstruiert werden. Für die Parametrisierung einer solchen Szene muss die nahe und ferne Szenengrenze angegeben werden, um Punkte nicht in eine unendlich tiefe Szene zu projizieren, was zu unendlich großen Koordinaten oder einer unendlich kleinen Disparität führt. Um unendliche große oder kleine Werte zu vermeiden, wird für die Implementierung in Computerprogrammen *Normalized Device Coordinates (NDC) Space* verwendet.

Kapitel 3

Grundlagen der Lichtfeldtheorie

3.1 Licht-, und Radianzfelder

Ein Lichtfeld (eng. Lightfield) beschreibt die Ausbreitung von Licht im Raum als Funktion der Position und der Blickrichtung. Mathematisch betrachtet ist ein Lichtfeld eine fünfdimensionale Funktion, welche für eine Position (x, y, z) und einen Raumwinkel (Θ, Φ), eine Lichtmenge beschreibt.

Michael Faraday schrieb 1846 in "Thoughts on Ray Vibrations"[12] das Licht als ein Feld, ähnlich dem magnetischem Feld verstanden werden kann. Der Begriff Licht Feld wurde von Andrey Gershun in seiner gleichnamigen Arbeit [13], über die radiometrischen Eigenschaften von Licht im dreidimensionalen Raum, geprägt.

Alle Lichtstrahlen, die von Lichtquellen oder Oberflächen emittiert oder reflektiert werden, bilden ein Lichtfeld. Eine Kamera oder ein Auge nimmt einen Schnitt durch ein Lichtfeld wahr, welches durch eine Linse fokussiert werden muss, um scharf abgebildet zu werden. Die Linse bildet alle Strahlen, die aus ähnlicher Richtung und Entfernung kommen in einer Bildebene ab. Damit erhält man einen, auf eine bestimmte Tiefe fokussierten, Schnitt durch ein Lichtfeld. Um eine Szene scharf abzubilden, ist Fokussierung nötig, da eine sichtbare Oberfläche, Lichtstrahlen auf jede Stelle im Bild werfen würde. Allerdings entsteht ein Bild erst, wenn ein Oberflächenpunkt nur von einer Stelle auf der Bildebene "gesehen"wird.

Eine Sammlung von Bildern, die von verschiedenen Positionen auf verschiedene Tiefen fokussiert aufgenommen wurden, bildet eine Sammlung von aufgenommenen Lichtstrahlen eines Lichtfeldes einer Szene.

Mathematisch betrachtet ist ein Lichtfeld eine fünfdimensionale Funktion, welche für jede Position (x,y,z) und jeden Raumwinkel (θ, ϕ), eine Lichtmenge beschreibt.

Ein Radianzfeld (*Radiance Field*) beschreibt alle Lichtstrahlen die, von Oberflächen oder Lichtquellen auf einen Punkt (x, y, z) aus Richtung (θ, ϕ)fallen.

3.2 Volumen Rendering

Als Bildsynthese oder "Rendern"bezeichnet man die Berechnung eines Bildes aus einer Szenenbeschreibung. In 2D kann das die Darstellung einer HTML Seite oder einer Vektorbilddatei sein. In 3D liegt bei klassischen Szenenbeschreibungen meist eine Datenstruktur mit geometrischen Primitiven wie Polygone vor.

Um ein Bild zu berechnen, wird die Szene von einer virtuellen Kamera in eine zweidimensionale Ansicht projiziert. Die Kameraposition ist der Ursprung von Strahlen, die sich zur Berechnung von sichtbaren Polygonen in den Bildraum ausbreiten.

Ein iteratives Verfahren testet für alle, in der Szenenbeschreibung vorkommenden Primitive, ob eine geometrische Überschneidung mit dem Strahl vorliegt. Kommt es zu mehreren Überschneidungen werden diese nach ihrem Abstand zur Kamera geordnet und der geringste Abstand zur Kamera als sichtbar gewertet.

Liegt die Bildbeschreibung als Volumen in Form eines dreidimensionalen Voxelgrid oder Skalarfeld vor, wird der Strahl durch das komplette Volumen ausgewertet.

Da Informationen über Objektoberflächen nicht direkt gespeichert werden, kann die Sichtbarkeit nicht durch Testen von Überschneidungen zwischen vorliegenden Listen an Primitiven berechnet werden. Volumenbeschreibungen geben die Möglichkeit, die Absorption des partipizierenden Mediums, durch das sich ein Lichtstrahl bewegt, zu modellieren. Ein Beispiel ist die realistische Darstellung von Rauch oder einer Wolke. Um die Sichtbarkeit zu bestimmen, wird entlang des Strahls bestimmt, wie viel Licht durch einen hohen Dichtewert absorbiert wird. Ist die Dichte an einem Abschnitt hoch, hat diese Stelle einen höheren Anteil am berechneten Pixel, vorausgesetzt die maximale Opazität zwischen Kamera und Sampleposition ist nicht erreicht.

3.3 Differential-Rendering

Beim *Differential Rendering* sind nicht Geometriedaten Grundlage des Rendering-Prozesses, sondern die aufgenommenen Sensordaten einer Szene selbst: aus Sensordaten werden 3D-Geometrie, Beleuchtung, Materialien und Bewegungen so abgeleitet, dass ein Renderer die beobachtete Szene realistisch wiedergeben kann. Klassische "Renderer"(s.o.) sind im Gegensatz dazu darauf ausgelegt, in einem Vorwärtsprozess die Bildsynthese zu lösen. Diese approximativen differenzierbare Renderer (DR), modellieren explizit eine funktionale Beziehung zwischen Änderungen der Modellparameter und den Bildbeobachtungen bzw. Sensordaten oder Bilddaten.

Die Differenzierbarkeit des Rendering-Prozess ermöglicht es, den Fehler im Vergleich zu einem Ground-Truth Bild zu berechnen und in einem Rückwärtsprozess abhängig von Änderungen der Szenenparameter zu machen.

Im Kontext von NeRF bedeutet Differntial Rendering: NeRF ist ein neuronales Netz welches selber kein Bild generiert, sondern nur die Parameter einer Szene lernt. Die gelernten Parameter bilden eine Szenebeschreibung, aus der mit Hilfe eines Renders Bilder generiert werden können. Die generierten Bilder ermöglichen eine visuelle Bewertung des gelernten Netzes. Die Bewertung des Ergebnisses (*Loss*) wird durch den Vergleich mit Ground-Truth Bildern der Szene vorgenommen. Das heißt, um den Traingsloss berechnen zu können, muss die Szenenbeschreibung erst aus einer Ansicht gerendert werden.

Um den Fehlerwert einer Lossfunktion durch einen Computational Graph zurückrechnen zu können, müssen alle Funktionen des Graphen differenzierbar

sein. Durch Einsetzen eines diffrenziebaren Renders zwischen Ergebnis eines neuronalen Netz und Lossfunktion, ist es möglich Szenenbeschreibungen End-zu-Ende aus Bilddatensätzen zu lernen.

Kapitel 4

Einführung in künstliche neuronale Netze

Neuronale Netze sind ein von der Natur inspirierter Mechanismus, der es Computern ermöglicht, ähnlich wie ein Gehirn zu lernen - der Computer kann so, wenn er die Lösung für ein Problem kennt, korrekte Lösungen für ähnliche Probleme finden und vieles mehr. Das Studium der künstlichen Neuronalen Netze ist motiviert durch die Ähnlichkeit zu erfolgreich arbeitenden biologischen Systemen, welche im Vergleich zum Gesamtsystem aus sehr einfachen, aber dafür vielen und massiv parallel arbeitenden Nervenzellen bestehen und Lernfähigkeit besitzen. Ein Neuronales Netz muss nicht explizit für seine Aufgaben programmiert werden, es kann aus Trainingsbeispielen lernen oder durch Bestärkung (Reinforcement Learning). [14]

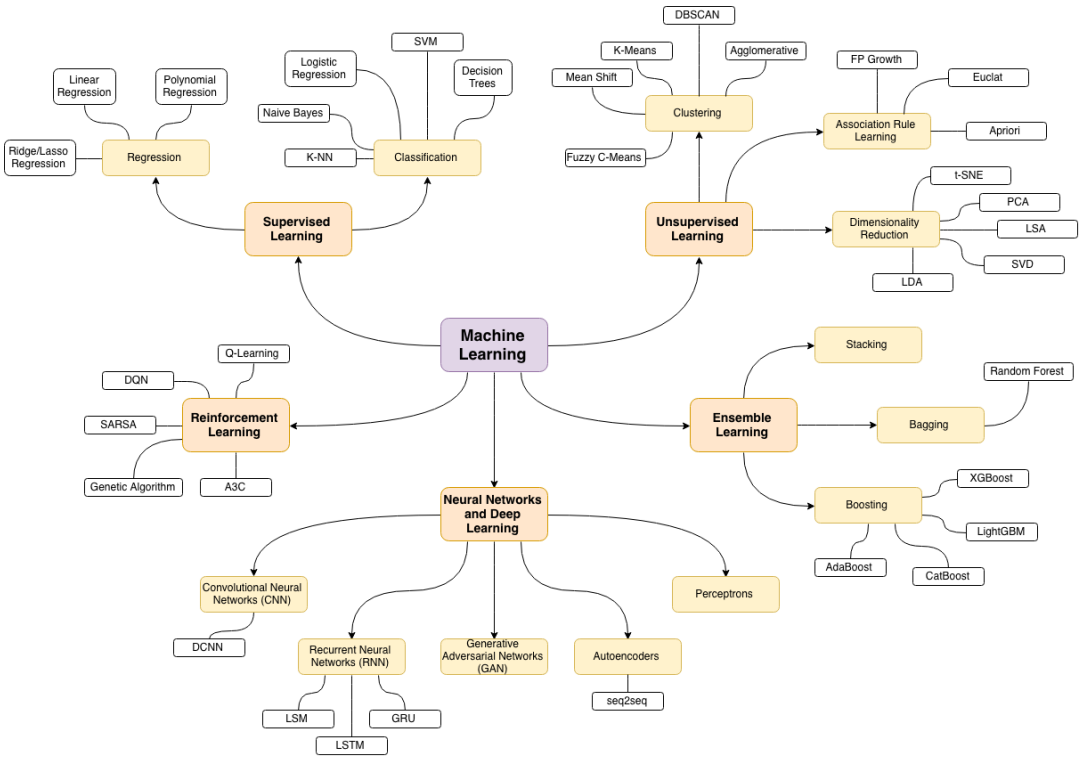


Abbildung 4.1: Einordnung verschiedener Machine Learning Gebiete. Der untere Baum umfasst Techniken wie die neuronalen Netze die im NeRF-Verfahren Anwendung finden. [15]

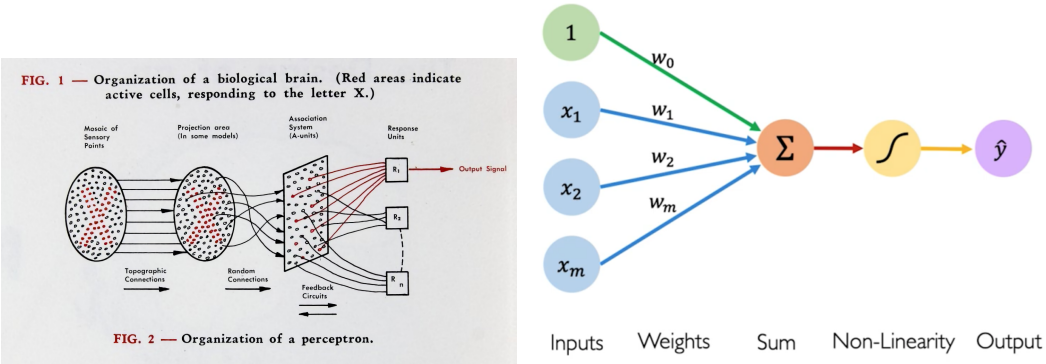
Der Vorteil von neuronalen Netzen kommt von der Fähigkeit, aus Trainingsdaten die Repräsentation des Problems zu erkennen und mit den Eingangsdaten in Beziehung zu setzen. Dabei müssen keine expliziten Regeln für die Repräsentation oder das zu lösende Problem definiert werden. Das Netz wird beim Trainieren nur anhand seines Ergebnis bewertet und schließt selbstständig daraus, wie die interne Verschaltung verändert werden muss, um aus Eingangsdaten mit Hilfe von gelernten Parametern auf ein korrektes Ergebnis zu schließen. Im mathematischen Sinn wird hierbei eine Abbildung zwischen zwei Räumen gelernt. Es ist bewiesen, dass ein neuronales Netzwerk ein als universeller Approximierungsalgorithmus jede beliebige Mapping-Funktion lernen kann [16, 17]. Das heißt nicht, dass ein neuronales Netz jede Funktion exakt lösen kann, sondern das es jede Funktion annähern kann und das Ergebnis genauer wird, wenn mehr Neuronen vorhanden sind.

Die Architektur ist stark an den Informationsfluss biologischer Neuronen angelehnt. Ziel war es dabei ein einfaches Modell eines biologischer Hirns aufzubauen, welches komplexe Aufgaben, wie vorausschauende Modellierung, lösen kann. Eine Illustration eines der ersten neuronalen Netze ist in Abbildung 4.2A zu sehen. Der Funktionsfluss in einem einzelnen Neuron eines neuronalen Netzes ist Diagramm 4.2B abgebildet. Dabei wird eine Menge eingehender Signale gewichtet, summiert und abhängig eines Schwellwerts eine Klassifizierung vorgenommen.

Gegeben ist eine Funktion $f(x)$ die auf eine Genauigkeit $\epsilon > 0$ berechnet werden soll. Es ist garantiert, dass durch Hinzufügen von Schichten von Neuronen, ein neuronales Netz gefunden werden kann, für dessen Ergebnis $g(x)$ aller Eingangsdaten x, $|g(x) - f(x)| < \epsilon$ gilt.

Ein typisches Beispiel für ein Problem was sich gut mit maschinellem Lernen bewältigen lässt, ist die Erkennung von handschriftlichen Ziffern. Ein Datensatz besteht hierbei aus mehreren Tausend handschriftlichen Ziffern mit korrekter Zuordnung. Jedem Bild einer Ziffer kann eine eindeutige Kategorie oder Label, $y \in (0, 1, ..., 9)$, zugewiesen werden. Dem neuronalen Netz werden nacheinander Bilder aus dem Datensatz gezeigt, so genannte Eingangsdaten $\mathbf{X}$. Für jeden dieser Datenpunkte $x \in \boldsymbol{X}$ berechnet das Netz eine Vorhersage $\hat{y} \in (0, 1, ..., 9)$. Während des Trainings wird jede Vorhersage $\hat{y}$ mit dem erwartetem Label y verglichen und dementsprechend die Parameter des Netzwerks angepasst. Um ein Auswendiglernen des Datensatzes zu vermeiden und eine gewisse Generalisierung zu erreichen, wird der Datensatz in einen Trainings-, und einen Testdatensatz geteilt. Den Testdatensatz sieht das Netz während des Trainierens nicht, sondern er wird nur für eine Bewertung herangezogen. Schneidet ein neuronales Netz mit einem Testdatensatz schlechter ab, als mit dem Trainingsdatensatz, deutet das auf so genanntes "Overfitting" hin. Das Netz hat Parameter gefunden, welche zwar die Traningsdaten gut beschreiben, aber schlecht auf unvorhergesehene Eingangsdaten generalisieren. Ziel ist es viel mehr, das Netzwerk die dem Datensatz zu Grunde liegende Struktur erkennen zu lassen und darauf basierend eine Vorhersage zu treffen. Wie gut ein Netzwerk trainiert wurde, wird daran erkannt, dass es eine hohe Generalisierung erreicht hat und auch handschriftliche Ziffern erkennt, welche nicht im Datensatz enthalten waren.

4.0.1 Perzeptron



(A) Ein Bild aus Rosenblatt's "The Design of an Intelligent Automaton", 1958. Entnommen aus [18].

(B) Flussdiagramm eines Neurons aufgeteilt in einzelne Operationen.

Abbildung 4.2

Das Prinzip eines Neurons in einem künstlichen neuronalen Netzwerk lässt sich am einfachsten am Beispiel eines neuronalen Netzes mit nur einer Schicht, genannt Perzeptron, erklären. Ein Perzeptron ist ein Einzelneuronenmodel, und damit Vorläufer des *Multilayer Perceptrons* (MLP), zu Deutsch mehrschichtiges Perzeptrons, in dem mehrere Schichten Neuronen hintereinander angeordnet sind. Neuronale Netze lernen durch die Beobachtung des Fehlers der Vorhersage im Vergleich zu einem bekannten Datensatz. Ein Perzeptron kann, durch eine iterative Fehlerrechnung zwischen Beispiel und Vorhersage lernen, einen Datensatz in zwei Kategorien einzuteilen. Im Fall eines binären Klassifizierungsproblems benötigt ein Perzeptron eine begrenzte Zahl an Iterationen, bis es einen Datensatz mit linear separierbarem Mustern richtig erkennen kann [19] [20]. Ein Perzeptron berechnet eine gewichtete Summe aller Eingangsdaten und gibt ein Signal aus, wenn ein bestimmter Schwellwert überschritten wird. Es besteht aus einem Vektor von Gewichten $\boldsymbol{w} = [w_1 ... w_m]$, eins für jeden Eingangsdatenpunkt $x = [x_1 ... x_m]$ und ein unabhängiges Gewicht genannt Bias, b. $\boldsymbol{w}$ und b sind die *Parameter* des Netzwerks. Wir bezeichnen die Parameter des Netz mit Φ, von denen $\phi_i \in \Phi$ der $i$te Parameter ist. [21]

Als Beispiel sind x_1 und x_2 Datenpunkte für die ein Perzeptron eine Vorhersage machen soll, w_1 und w_2 sind die jeweiligen zu lernenden Gewichte. Daraus ergibt sich die gewichtete Summe $\sum_{i=1}^{2} w_i x_i = w_1 x_1 + w_2 x_2$. Wenn die Summe einen bestimmten Schwellwert b erreicht, wird das Perzeptron aktiviert und gibt ein Signal weiter. Damit ergibt sich folgendes Verhalten für ein Perzeptron:

$$f_\Phi(\mathbf{x}) = \begin{cases} 1 & \text{falls } b + \sum_{i=1}^{l} x_i w_i > 0 \\ 0 & \text{andernfalls} \end{cases} \tag{4.1}$$

Die Schwellwertfunktion ist im Allgemeinen eine Aktivierungsfunktion und mit einem σ bezeichnet. Die Aktivierungsfunktion eines Neurons bestimmt wie Stark das Signal an eine folgende Ebene weitergegeben wird. Im Falle eines Perzeptrons, welches aus nur einem Neuron und dadurch aus nur einer Ebene

besteht, ist das Ergebnis der Aktivierungsfunktion auch das finale Ergebnis der Klassifizierung.

Abbildung 4.4 zeigt Datenpunkte $X \in \{\text{Lila, Grün}\}$ verteilt in zwei Dimensionen. Der linke Datensatz ist linear separierbar, wie die rote Linie zeigt. Es sollen w_1 und w_2 die Gewichte eines Perzeptrons mit zwei Eingaben sein und θ der Schwellwert der Aktivierungsfunktion. Ein Perzeptron kann eine Klassifizierung der Punkte dadurch lernen, dass $w_1x_1 + w_2x_2 > b$ in die eine und $w_1x_1 + w_2x_2 \leq b$ in die andere Kategorie fallen. Bei korrekt gewählten Gewichten, werden die zwei Regionen durch die Linie $w_1x_1 + w_2x_2 + b = 0$ getrennt.

Um eine exklusives Oder, ***XOR*** zu lernen, müssten die Ungleichheiten 4.2 eingehalten werden:

$$\begin{array}{llll} x_1 = 0 & x_2 = 0 & w_1x_1 + w_2x_2 = 0 & \Rightarrow 0 < \theta \\ x_1 = 1 & x_2 = 0 & w_1x_1 + w_2x_2 = w_1 & \Rightarrow w_1 \geq \theta \\ x_1 = 0 & x_2 = 1 & w_1x_1 + w_2x_2 = w_2 & \Rightarrow w_2 \geq \theta \\ x_1 = 1 & x_2 = 1 & w_1x_1 + w_2x_2 = w_1 + w_2 & \Rightarrow w_1 + w_2 < \theta \end{array} \tag{4.2}$$

Als visuelle Analogie dient Abbildung4.3.

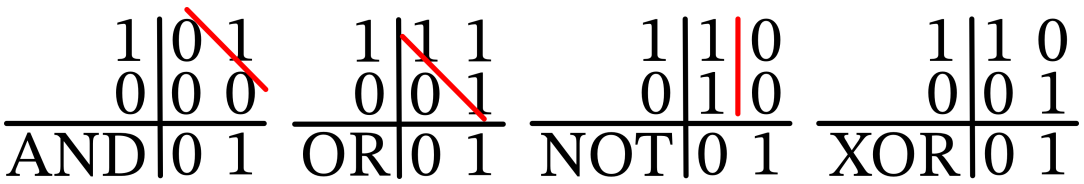


Abbildung 4.3

Logische Operationen wie AND, OR, NOT sind linear separierbar und können daher von einem einschichtigen Perzeptron gelernt werden. Es ist nicht möglich mit nur einer geraden Trennlinie die Kategorien einer XOR Verknüpfung zu teilen. XOR ist nicht linear trennbar, da mehr als eine Gerade gebraucht wird, um den Datensatz in zwei Dimensionen zu separieren.

Die erste Ungleichung beschränkt θ auf Werte größer Null und aus den folgenden zwei Ungleichungen ergeben sich w_1 und w_2 als positive Werte. Daher kann $w_1 + w_2 < 0$ in der vierten Ungleichung nicht eingehalten werden. Diese Widersprüchlichkeit impliziert dass kein einschichtiges Perzeptron in der Lage ist eine

Ein Perzeptron mit den Gewichten $\boldsymbol{w}$, dem Bias b und einer Aktivierungsfunktion

$$\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \sigma\left(\sum_{i=1}^{d} w_i x_i + b\right) \tag{4.3}$$

$$\hat{y} = \sigma\left(b + \sum_{i=1}^{l} x_i w_i\right) \tag{4.4}$$

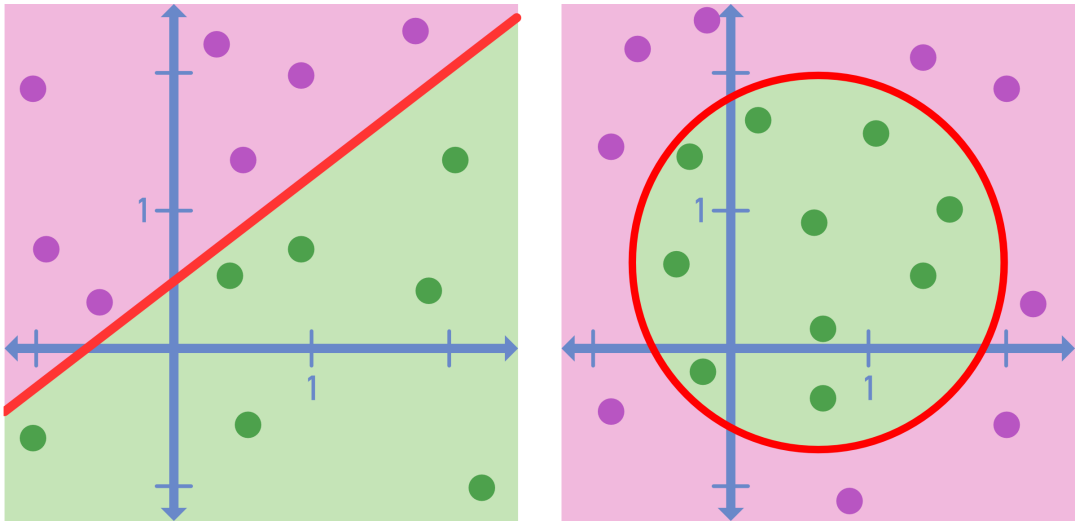


Abbildung 4.4: Visuelle Darstellung eines linear separierbarer Datensatz (links) und einem nicht linear separierbaren Datensatz (rechts) mit Trennline in Rot. Ersteres kann von einem einzelnen Neuron gelernt werden, für Zweiteres benötigt man ein neuronales Netz mit mindestens zwei Schichten oder eine andere Parametrisierung, z.B. Polarkoordinaten.

$$\hat{y} = \sigma\left(b + \boldsymbol{X}^T \boldsymbol{W}\right) \tag{4.5}$$

$$\text{wobei: } \boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ und } \boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \tag{4.6}$$

4.0.2 Lineare Regression

Ein Algorithmus für maschinelles Lernen zeichnet sich dadurch aus, dass er "Erfahrung sammeln" und sich verbessern kann. Das geschieht durch das iterative Ausführen eines Algorithmus, dessen Ergebnis bewertet werden kann. Durch das Vergleichen von iterative Ausführungen und Bewertungen, wird bestimmt ob Anpassungen der Parameter zu einem besseren Ergebnis führen. Ein konkretes Beispiel dafür ist lineare Regression, damit können die passenden Gewichte $\boldsymbol{w}$ des Perzeptrons für einen Datensatz (X, Y) bestimmt werden. Lineare Regression nimmt einen Vektor $x \in \mathbb{R}^n$ und bestimmt dafür einen Wert $y \in \mathbb{R}$. Das Ergebnis ist eine lineare Funktion, siehe 4.7, der Eingangsdaten mit den Gewichten $w \in \mathbb{R}^n$ als Parameter.

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} \tag{4.7}$$

Die Parameter bestimmen wie sich das System verhält. In diesem Fall sind w_i Koeffizienten die mit den den Eingangsdaten x_i multipliziert und am Ende zusammen gezählt werden. Man kann sich vorstellen, dass w eine Menge von Gewichten sind, die bestimmen, wie viel Einfluss ein Eingangswert auf das Ergebnis hat.

Die Aufgabe T der linearen Regression ist es, y von x durch das Modell $\hat{y} = \boldsymbol{w}^T \boldsymbol{x}$ zu bestimmen. Es wird eine Möglichkeit, zu bewerten wie korrekt die Vorhersage $\hat{y}$ ist, benötigt.

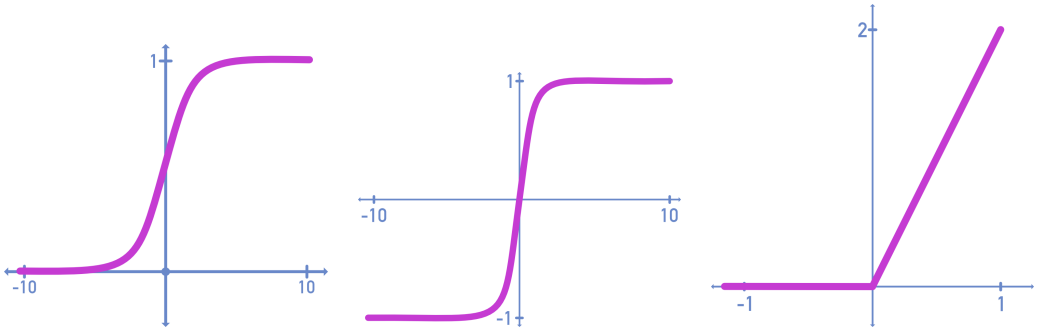
Die Richtigkeit kann bestimmt werden, indem der mittlere quadratische Fehler des Modells mit Hilfe der Testdaten berechnet wird. Wenn $\hat{y}^{\text{test}}$ die Vorhersage des Models und $\hat{y}$ das korrekte Ergebnis der Testdaten ist, ergibt sich der mittlere quadratische Fehler (***M**ean**S**quare**E**rror*) wie folgt:

$$\text{MSE}_{\text{test}} = \frac{1}{m} \sum_i \left(\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})} \right)_i^2 \tag{4.8}$$

Der Fehler nähert sich 0 an, wenn $\hat{y}^{\text{test}} = \hat{y}$ ist und vergrößert sich wenn der Euklidische Abstand zwischen Vorhersage und korrektem Ergebnis größer wird.

Um einen Algorithmus für maschinelles Lernen zu erstellen, muss dieser seine Gewichte w so anpassen können, dass der MSE_{test} unter Zuhilfenahme von Trainigsdaten $(\boldsymbol{X}^{\text{train}}, \boldsymbol{Y}^{\text{train}})$ minimiert wird.[22]

4.0.3 Aktivierungsfunktionen



(A) $g(z) = \sigma(z)$ (B) $g(z) = \tanh(z)$ (C) $g(z) = \max(0, z)$

Abbildung 4.5

Um einen Nutzen aus mehreren Lagen von Neuronen ziehen zu können, müssen diese durch nicht lineare Aktivierungsfunktionen verbunden werden.

Lange Zeit hat man dies nicht erkannt, da das oben beschriebene Perzeptron durch seine Linearität als Matritzenmultiplikation $\boldsymbol{y} = \boldsymbol{XW}$ beschrieben werden kann. Fügt man nach den Eingangsneuronen eine Schicht linearer Neuronen $\boldsymbol{U}$ ein, die mit der folgenden Ausgabeschicht $\boldsymbol{V}$ verbunden ist, ergibt sich:

$$\boldsymbol{y} = (\boldsymbol{xU})\boldsymbol{V} = \boldsymbol{x}(\boldsymbol{UV}) \tag{4.9}$$

Durch Anwendung des Assoziativgesetz der Matrizenmultiplikation wird deutlich, dass aller erhoffter Mehrwehrt der zweiten Schicht, genauso gut in einem einlagigen Netz mit $\boldsymbol{W} = \boldsymbol{UV}$ abgebildet werden kann wie 4.9 zeigt.

Einfache Abhilfe schafft die Einführung einer nichtlinearen Berechnung zwischen den Schichten. Diese nicht lineare Funktion als Verbindung zwischen den Schichten eines neuronalen Netzwerks wird Aktivierungsfunktion genannt. In den letzten Jahren ist eine häufig verwendete Aktivierungsfunktion die *rectified linear unit*, kurz **relu** genannt, zu sehen in Abbildung 4.5c. Davor war lange die Sigmoid Funktion 4.5a populär. Sie wird heute oft verwendet, um das Ergebnis zwischen Null und Eins zu begrenzen. **relu** dagegen liefert ein Ergebnis zwischen Null und Unendlich. Ein Problem einer Aktivierungsfunktion mit begrenztem Ausgabewert ist, dass *vanishing gradient* Problem, bei dem der Gradient der vorderen Schichten verschwindend gering wird [21].

4.0.4 Multilayer Perceptron

Ein *Multilayer Perceptron* (MLP), aus dem Englischen übersetzt "mehrlagiges Perzeptron", ist eine der einfachsten Architekturen eines Neuronalen Netzwerks. Es besteht mindestens aus drei Ebenen, Eingabeschicht, Ausgabeschicht und dazwischen beliebig vielen "hidden layer"(dt. versteckte Ebenen), die durch nichtlineare Aktivierungsfunktionen verbunden sind. Das Erweitern eines Perzeptrons in der Tiefe durch weitere Schichten, ermöglicht, dass eine Folgeschicht bestimmte Merkmale in der vorherigen Schicht erkennt und verstärkt.

$$\Pr(A(x)) = \sigma\left(\rho\left(\mathbf{x}\mathbf{U} + \mathbf{b}_u\right)\mathbf{V} + \mathbf{b}_v\right) \tag{4.10}$$

$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g\left(z_{k-1,j}\right) w_{j,i}^{(k)} \tag{4.11}$$

Für ein exklusives Oder ***XOR***, welches von einem einfachen Perzeptron nicht gelöst werden kann, wird ein Netzwerk mit mindestens zwei Schichten Neuronen benötigt. Die erste Schicht lernt eine nicht lineare Transformation in den sogenannten *Latent Space*, die folgende Schicht kann dann in diesem gelernten Raum die Klassifizierung vornehmen. Ein neuronales Netzwerk mit mehreren "hidden Layer"ist ein "Multi Layer Perceptron".

Ein MLP wird oft für Klassifizierung, Erkennung, Vorhersage und Annäherung von Mustern verwendet. Ein multilayer Perzeptron ist ein neuronales Netz mit Vorwärtskopplung auch *Feedforward Neural Network* genannt, welches mehrere Schichten Neuronen zwischen Ausgangs-, und Eingangsschicht hat. Vorwärtskopplung (Feedforward) bedeutet, dass die Daten in der Richtung von Eingangs zur Ausgangsschicht fließen, es kommt zu keiner zirkularen Abhängigkeit. Diese Art von neuronalen Netz kann mit Backpropagation durch mehrere Schichten nicht linear trennbare Probleme wie das XOR-Problem lösen. Backpropagation ist ein Algorithmus mit dem *Feedforward Neural Networks* durch *supervised learning* mit *gradient descent* trainiert werden [22].

Die verdeckten Neuronen in den Zwischenschichten eines Neuronalen Netzwerks wirken als Merkmalerkennung. Im Laufe des Trainings eines neuronalen Netzes entdecken sie grundlegende Muster, welche die Eingangsdaten charakterisieren. Das geschieht, indem die Eingangsdaten durch eine nicht lineare Transformation in den so genannten *Latent Space*, z.Dt. Merkmal Raum, gehoben

werden. In diesem neuen Raum ordnen sich die z.B für ein Klassifizierungsproblem interessanten Kategorien so an, dass sie einfacher separierbar sind als im orginalen Raum der Eingangsdaten [20]. Die Bildung dieses *Latent Space* durch *supervised learning* ist der Hauptunterschied zwischen Rosenblatt's einschichtigen Perzeptron in Abbildung 4.2a und einem mehrschichtigen Perzeptron.

Der Verlauf des Fehlers ändert sich während des Trainings abhängig von den gewählten Werten der Gewichte. Gewichte die einen höheren Einfluss auf ein richtiges Ergebnis haben und den Fehler minimieren, nehmen während der Trainings zu. Die Verwendung einer monoton steigenden Aktivierungsfunktion für die Verknüpfungen der Neuronen mit der folgenden Schicht, macht ein mehrschichtiges Perzeptron differenzierbar.

Ein Algorithmus der mit *supervised learning*, deutsch überwachtes Lernen, trainiert wird, setzt voraus, dass das Ergebnis des Algorithmus mit einem Beispieldatensatz aus Eingangsdaten mit bekanntem Ergebnis *ground truth* verglichen werden kann. Diese Kontrolle berechnet einen Fehlerwert (*Loss*), welcher die Schrittweite im Gradientenverfahren für die Rückwärtspropagation im Optimierungsschritt bestimmt [22].

Der Fehler zwischen der Ausgabe des Netzes und einem bekannten Zielwert, kann durch ein neuronales Netz mit stetig differenzierbarer Aktivierungsfunktion, auf Auswirkung von Veränderung der Gewichte zurückverfolgt werden.

4.0.5 Loss

Mit der Lossfunktion wird berechnet, wie korrekt die Vorhersage eines neuronalen Netzes ist. Bei einem korrektem Ergebnis nähert sich der Loss dem Wert Null.

Als Beispiel nehmen wir eine Perzeptron mit zwei Parametern. Während des Trainings sollen die Gewichte Φ optimiert also für einen Datensatz angepasst werden. Der Betrag $\delta\phi_i$, um den ein Gewicht angepasst wird, berechnet sich aus der *learning rate* ($\mathcal{L}$), dem Loss (L) und dem momentanen Wert von Φ_i.

$$\Delta\phi_i = -\mathcal{L}\frac{\partial L}{\partial \phi_i} \tag{4.12}$$

Wir denken uns eine euklidische Ebene mit den Achsen ϕ_1 und ϕ_2 wie in Abbildung 4.7. Jeder Punkt der Ebene entspricht einem Loss $\mathcal{L}$ abhängig von den gewählten Parametern Φ des Perzeptrons. Da der Loss für jeden Parameter unterschiedlich verläuft, ergibt sich eine Art Hügellandschaft ähnlich Abbildung 4.7. Ziel des Trainieres eines neuronalen Netzwerks ist es, das globale Minimum in dieser Hügellandschaft zu finden und lokale Minima zu vermeiden.

Nehmen wir an der momentane Wert von ϕ_1 ist 1.0 und ϕ_2 ist 2.2. Um diese zu optimieren schauen wir, wie sich L um die Position (1.0, 2.2) auf der Ebene ändert. Abbildung 4.6 ist ein Schnitt entlang der Ebene $\phi_2 = 2.2$ und zeigt den angenommenen Verlauf des Loss abhängig von ϕ_1 . An der Stelle $\phi_1 = 1$ hat der Loss eine Steigung von ungefähr $-\frac{1}{4}$. Wenn die *learning rate* $\mathcal{L} = 0.5$ angenommen wird, dann besagt 4.12 dass ϕ_2 um $(-0.5) * (-\frac{1}{4}) = 0.125$ erhöht wird und tatsächlich ist der Loss 0.125 Einheiten weiter rechts niedriger [21].

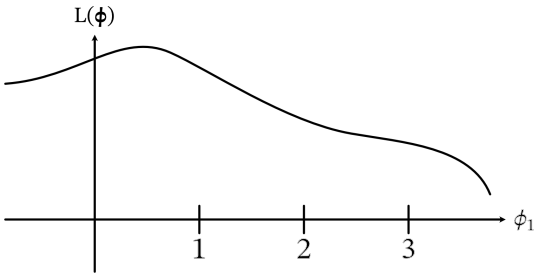


Abbildung 4.6: Loss abhängig von ϕ_1

4.0.6 Backpropagation und Gradient Descent

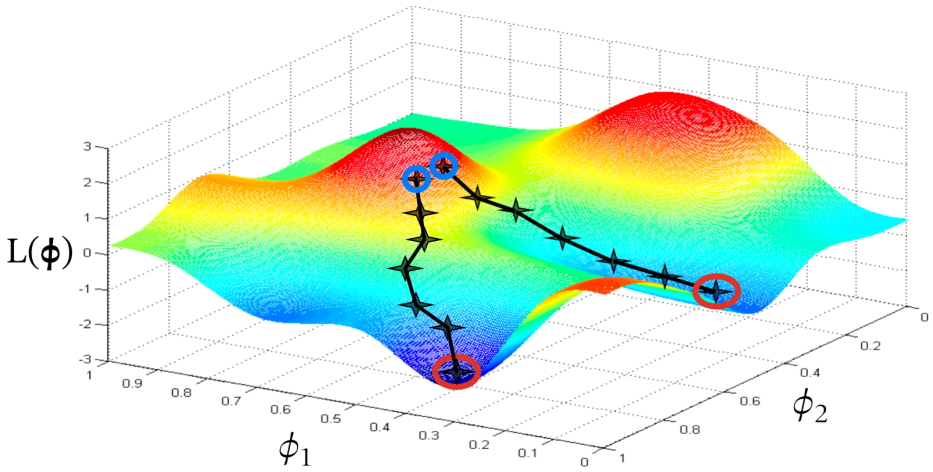


Abbildung 4.7: Plot des Loss L eines Perzeptrons abhängig von den beiden Parametern ϕ_1, ϕ_2. Abhängig vom gewählten Startpunkt kann ein Gradientenverfahren in verschiedenen lokalen Minima enden.

Ein neuronales Netz formt einen *computation graph*, siehe Abbildung 4.8, aus einer Serie von Funktionen. Das Ergebnis wird *Loss*, $L \in \mathbb{R}$, genannt. Der Einfluss aller Parameter der Eingangsdaten auf das Ergebnis L wird Gradient genannt.

Das neuronale Netz und die *Loss*-Funktion bilden eine große differenzierbare Funktion.

Backpropagatation bestimmt die Änderungsrate aller Knotenpunkte im computational graph. Dadurch lässt sich der exakte Beitrag eines Eingangswerts zum Ergebnis des neuronalen Netzes berechnen.

Die Differenzierung des neuronalen Netzes beschreibt die Gradienten aller eingehenden Parameter relativ zum Ergebnisraum des *Loss*. Ziel von Backpropagation ist es, den *Loss* und die Änderungsrate der Gewichte während des Trainings zu minimieren.

Abbildung 4.8 zeigt den *computational graph* einer Funktion f mit den Parametern [x,y,z] welche die Rechnung $(x + y)z$ ausführt.

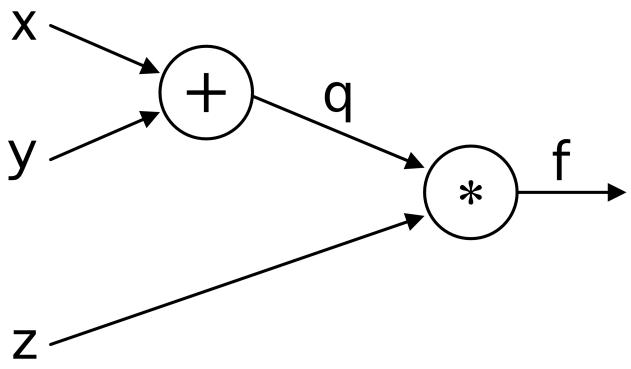


ABBILDUNG 4.8: Ein Beispiel für einen computational graph aus zwei Operationen. Die Funktion $f(x+y)z$ mit den Parametern [x,y,z] und das Ergebnis der Summe $(x+y)$ ist q.

$$\begin{aligned} f(x,y,z) &= (x+y)z \\ q &= x+y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1 \\ f &= qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q \\ &\text{Gesucht: } \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \end{aligned} \tag{4.13}$$

Das Zwischenergebnis q der Summe $x+y$ ist in gleichem Maße von beiden eingehenden Werten abhängig und trägt jeweils mit einem Faktor von 1 zum Ergebnis von f bei. Die Multiplikation von q mit z, ergibt Abgeleitet ein invertiertes Verhältnis und trägt damit im mit einem Faktor von q zum Ergebnis bei.

Die Ableitungen, also Gradienten, werden an allen Knotenpunkt des *computational graph*, abhängig von jedem eingehenden Parameter berechnet.

$$f = Wx$$
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Dabei wird Stochastic Gradien Descent verwendet, um den *Loss* des Neuronalen Netz abhängig von seinen Gewichten zu minimieren. Die Cost Function $\boldsymbol{C}$ für die Berechnung des *Loss* ist hierbei differenzierbar.

$$\textbf{Cost: } C(w_1, b_1, w2, b2, w_3, b_3)$$

$$\textbf{Cost} \longrightarrow C_0(...) = (a^L - y)^2$$
$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$
$$a^{(L)} = \sigma(z^{(L)})$$

Backpropagation hat das Ziel, während des iterativen Trainings Gewichte zu maximieren und dabei ihren Anteil am Fehler der Vorhersage zu minimieren.

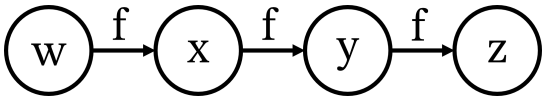


Abbildung 4.9: Ein Rechengraph der wiederholt Folgeinstruktionen aufruft, um seinen Gradienten zu berechnen.

In 4.9 ist $w \in \mathbb{R}$ der Eingabewert und $f : \mathbb{R} \longrightarrow \mathbb{R}$ eine Operation die bei jedem Schritt durch die Kette: $x = f(w), y = f(x), z = f(y)$ angewandt wird. Um die Ableitung $\frac{\partial z}{\partial w}$ zu berechnen wird die Kettenregel angewendet:

$$\frac{\partial z}{\partial w} \tag{4.14}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{4.15}$$

$$= f'(y)f'(z)f'(x)f'(w) \tag{4.16}$$

$$= f'(f(f(w)))f'(f(w))f'(w) \tag{4.17}$$

Im Rückwärtspass addieren sich Gradienten, wenn sie durch den gleichen Knotenpunkt zurück fließen.

Im Fall von Backpropagation wird dafür eine Objective Function aufgestellt, welche einen *Loss* abhängig vom Fehler des Ergebnisses berechnet und minimiert.

Verfolgt man in Abbildung 4.2b die Eingangsdaten X durch das Diagramm eines Neurons, werden diese individuell gewichtet und dann summiert.

Das Ergebnis der Addition wird danach einer nichtlinearen Aktivierungsfunktion übergeben, um das Endergebnis $\hat{y}$ zu berechnen. Auffällig dabei ist, dass jedes Neuron ein Gewicht mehr $w_n + 1$ als Eingangsdaten x_n hat. Dieser so genannte Biasterm b des linearen Teil eines Perzeptrons ist von den Eingangssignalen unabhängig und ermöglicht eine Verschiebung der Aktivierungsfunktion. Das ermöglicht die Verstärkung vieler in einem Neuron zusammen kommender Signale, auch wenn diese einen kleinen Wert haben. [23]

NeRF ist ein neuronales Netz welches selber kein Bild generiert, sondern nur die Parameter einer Szene lernt. Die gelernten Parameter bilden eine Szenebeschreibung, aus der mit Hilfe eines Renders Bilder generiert werden können. Die generierten Bilder ermöglichen eine visuelle Bewertung des gelernten Netzes. Die Bewertung des Ergebnisses (*Loss*) wird durch den Vergleich mit Ground-Truth Bildern der Szene vorgenommen. Das heißt, um den Traingsloss berechnen zu können, muss die Szenenbeschreibung erst aus einer Ansicht gerendert werden.

Um den Fehlerwert einer Lossfunktion durch einen Computational Graph zurückrechnen zu können, müssen alle Funktionen des Graphen differenzierbar sein. Durch Einsetzen eines diffrenziebaren Renders zwischen Ergebnis eines neuronalen Netz und Lossfunktion, ist es möglich Szenenbeschreibungen End-zu-Ende aus Bilddatensätzen zu lernen.

Kapitel 5

Neuronales Rendering

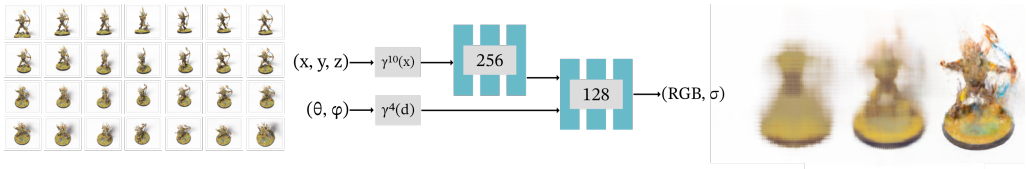


Abbildung 5.1: Neuronales Rendering erstellt aus einem Fotodatensatz mit Hilfe eines neuronalen Netz neue Ansichten.

Geoffrey Hinton Urenkel von George Bool, dem Begründer der Boolschen Algebra auf der alle konventionelle Computer basieren, schlägt vor, bei der Erschaffung künstlicher Intelligenz regelbasierte Systeme außen vor zu lassen und statt dessen, das System Muster und Verbindungen in großen Datenmengen selber erkennen zu lassen. Aus dieser Grundidee sind die künstlichen neuronalen Netze entstanden. Im speziellen tiefe Netze mit einer Vielzahl von Schichten (*Deep Learning*) haben eine Anwendung zur Lösung von Problemen der Computergrafik und Computer Vision gefunden. Beispiele sind *Convolutional Neural Networks* [24] für Objekterkennung oder *Generative Adversarial Networks* [25] zum hochskalieren von Bildern.

Als *Neural Rendering* werden Verfahren zur Generierung von Bildern und Videos bezeichnet, für die *Deep Neural Networks* also Netze aus mehreren Schichten verwendet werden. Dabei können Eigenschaften der Szene wie Kameraparameter, Pose, Geometrie, Lichtverhältnisse oder Oberflächeneigenschaften kontrolliert werden, um fotorealistische Ergebnisse zu erzielen.

Beim Trainieren eines NeRF gehen wir davon aus, dass die Regeln des Lichttransports, von einem neuronalen Netz entdeckt werden und mit den vorhandenen Trainingsdaten die höchste Übereinstimmung mit einer vorgegebenen Kontrollgruppe haben.

Bei NeRF dient die Bildersammlung einer Szene als Datensatz und in Verbindung mit mathematischen/physischen Modellen des Lichttransports als Kontrollgruppe. Pixel in den Bildern verkörpern im Datensatz die radiometrischen Messungen der in der Szene sichtbaren Oberflächen und lichtdurchlässiger Volumen. Diese Pixel wurden von Kameras aufgenommen, die an bestimmten Stellen in der Szene platziert wurden. Abhängig von der Beschreibung des Strahlengangs in einer Kamera und der Ausbreitung von Licht im Raum, wird ein mathematisches Modell aufgebaut, welches eine Szene repräsentiert und abhängig von Betrachtungspunkt und Blickrichtung ist. Das Modell kommt zu

unterschiedlichen Antworten, abhängig davon, wo der Betrachter steht und wo er hin schaut bzw. aus welcher Richtung auf die Szene geschaut wird.

Das lernende Modell entsteht in Form von verschalteten Knotenpunkte, welche unterschiedlich gewichtet Informationen austauschen. Die Gewichte der einzelnen Knotenpunkte und ihrer Verbindungen werden beim Trainieren des neuronalen Netzes gesucht. Um die gefundene Lösung zu bewerten, wird das Ergebnis des Neuronalen Netzes mit expliziten Modellierungen, durch gut verstandene Modelle des Lichttransports, verglichen. Die gleichen Modelle, die eine Bewertung des neuronalen Netze ermöglichen, simulieren in "render engines"die Ausbreitung und Interaktion von Licht in einer Szene, für die Erstellung fotorealistischer Bilder.

Als Datenstruktur bzw. als Modell, auf das wir die Architektur des neuronalen Netz ausrichten, dient ein Radianzfeld einer Szene. Durch die Unabhängigkeit von Position und Richtung, können damit Szenen beschrieben werden, die aus verschiedenen Betrachtungswinkeln unterschiedlich von einem optischen System wahrgenommen werden. Es wird also nicht nur die Objektoberfläche beschrieben, sondern auch die Unterschiede der Erscheinung aus verschiedenen Blickwinkeln. Das umfasst Reflektionen, welche für eine realistische Abbildung der Realität unabdingbar sind und geht somit über eine Beschreibung der örtlichen Ausbreitung einer Szene hinaus. Unter *Neuonalem Rendering* fassen wir Verfahren zusammen, bei denen mit der Hilfe von neuronalen Netzen Bilder synthetisiert (gerendert) werden. Damit verbindet diese Disziplin die physikalischen Grundlagen der Bildsynthese mit dem Trainieren von Neuronalen Netzen zur Erzeugung von fotorealistischen Bildern.

Ein Problem der klassischer Generierung von fotorealistischen Bilddarstellungen, ist die manuelle Arbeit die zur Erstellung detaillierter Szenenbeschreibungen benötigt wird. Die Modellierung von Geometrie, Oberflächeneigenschaften, Lichtquellen und Bewegung in hoher Qualität nehmen lange Zeit in Anspruch und setzen Expertenwissen voraus.

Der Vorteil des Einsatzes von Machine Learning Algorithmen liegt in der effizienten Lösung von differenzierbaren Teilproblemen der Bildgenerierung. Da fotorealistische Bilder den Anspruch haben, Eigenschaften der realen Welt wahrheitsgetreu nachzubilden, können neuronale Netze dafür trainiert werden, physikalische Gesetzmäßigkeiten effizient an Hand von Beispielen oder mathematischen Modellen wiederzugeben.

Neuronales Rendering kombiniert physikalisches Wissen wie z.B. die mathematische Beschreibung von Lichttransport mit gelernten Komponenten, um neue Algorithmen für die kontrollierte Generierung von Bildern zu schaffen.[26]

5.1 NeRF

Abbildung 5.2 gibt einen Überblick über die wesentlichen Bestandteile um ein NeRF zu trainieren. 5.2a zeigt die Trainingsdaten bestehend aus Bilddatensatz und fourier-transformierten extrinsischen Kameraparametern, mit denen das neuronale Netz F_θ trainiert wird. Für die Validierung wird das Netz in 5.2b aus bekannten Blickrichtungen abgefragt und mit Hilfe eines 5.2 Volumen Render

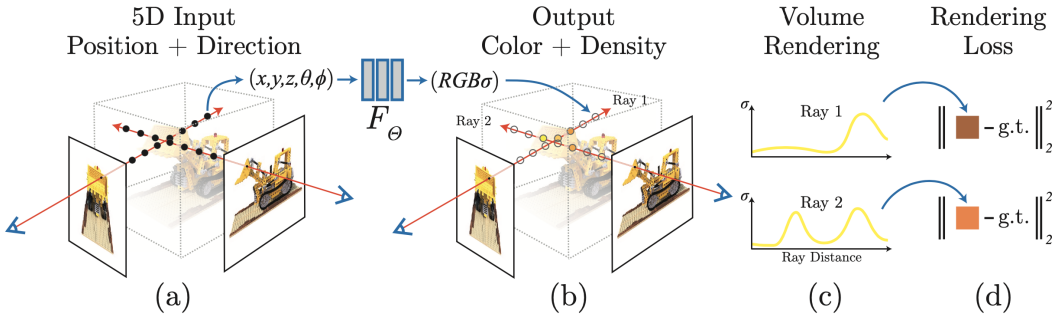


ABBILDUNG 5.2: Überblick über die neuronale Szenenbeschreibung und den Renderingprozess. Grafik entnommen aus der originalen NeRF Veröffentlichung von [27]

.

Bilder generiert. Die aus den gelernten Parametern des neuronalen Netzes generierten Bilder werden dann mit dem Validierungsbildern verglichen und der Trainigsloss berechnet. Abhängig des Loss werden die Trainingsparameter angepasst und der Prozess wiederholt. Der Vorgang ist abgeschlossen, wenn ein Validierungsbild sich möglichst wenig von einem gerenderten Bild unterscheidet.

Ziel ist es die räumliche Ausbreitung und Erscheinung einer Szene aus Fotografischen Aufnahmen zu lernen und in einem neuronalen Netz zu kodieren. Die Repräsentation der Szene wird dabei in den Gewichten eines Neuronalennetzes gespeichert. Ein mehrlagiges Perzeptron (MLP) kodiert dabei auf der Basis von Raumkoordinaten eine kontinuierliches Lichtfeld der Szene. Das neuronale Netz kann nach erfolgreichem Lernen, neue Ansichten der Szene, abhängig von Position und Blickrichtung, mit Hilfe eines Volume Renders, generieren.

5.1.1 Trainingssample aus Kamerastrahlen

Als Trainingsdaten dient eine Bildersammlung einer Szene mit entsprechenden intrisischen und extrinsischen Kameraparametern. Jeder Pixel im Datensatz dient mit seiner Position und Richtung als Sample für das Training. Jedes Sample repräsentiert damit einen Sample des zu lernenden Radianzfelds der Szene. Die Farbe eines Pixels ist das Ergebnis von mehreren photometrischen Effekten entlang eines Lichtstrahls der auf den Sensor fällt. Es wird angenommen, dass ein Lichtstrahl sich gerade durch den Raum bewegt, dabei kann ein partizipierendes Medium (z.B. Nebel, Glas) den Strahl teilweise Absorbieren. Dadurch lernt NeRf nicht nur die Erscheinung von Oberflächen, sondern modelliert auch semitransparente und damit refraktive Eigenschaften von Volumen.

Im Datensatz liegen alle Bilder mit ihren intrinsischen und extirnsischen Kameraparametern vor. Für jeden Bildpixel wird aus Bildhöhe, Bildbreite und Brennweite ein Strahl im lokalen Kamerakoordinatensystem generiert und mit Hilfe der extrinsischen Kameraparameter durch eine c2w (Kamera zu Welt) Matrix in einen gemeinsamen globalen Raum R^3 transformiert. Jeder Pixel ist durch einen Strahlursprung (x, y, z), Strahlrichtung (Θ, Φ) und Farbe (RGB) parametrisiert.

Als letzter Datenaufbereitungsschritt, bevor die Daten dem neuronalen Netz übergeben werden, werden Position und Richtung der Strahlen aus R^3 in einen hochdimensionalen Raum gehoben, um das Erlernen feiner Strukturen zu verbessern [28]. Abbildung 5.4 zeigt den Unterschied beim Fitten eines Bildes.

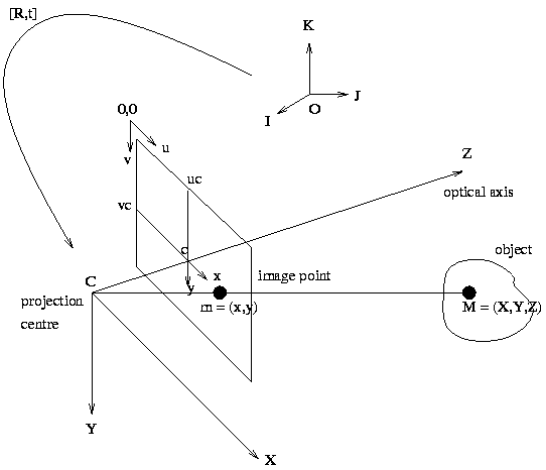


Abbildung 5.3: Bild zu Kamera zu Welt Koordinatensystem

5.1.2 Higher Embedding

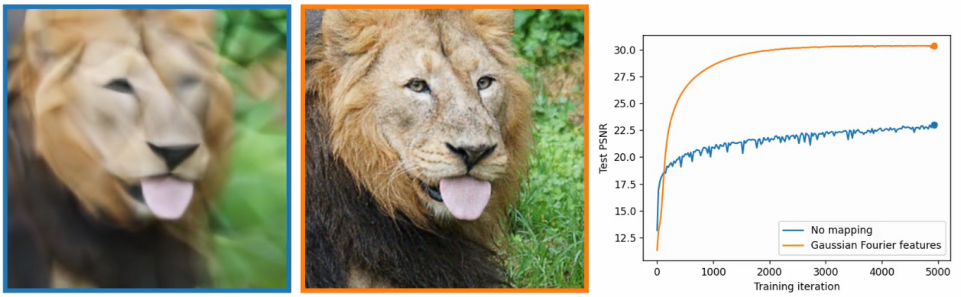


Abbildung 5.4: Niederdimensionale Problemdomäne für jede Bildkoordinate (x,y) eine Ausgabefarbe (r,g,b), kann durch ein MLP besser approximiert werden, wenn die Bildkoordianten zuvor durch Feature Mapping in einen hochdimensionalen fourier Raum abgebildet werden.

Tancik et al.[28] haben gezeigt, dass durch die Transformation der Eingabewerte in einen hochdimensionalen Raum, ein MLP Details bzw. hohe Bildfrequenzen einfacher lernen kann. Dafür werden die Eingabepunkte in den Fourier Raum transformiert, bevor sie dem MLP übergeben werden. MLPs tendieren dazu beim *Fitten* (optimieren von Parametern einer Funktion um Messwerte abzubilden) eines Signals als Tiefpass zu wirken, womit Details verloren gehen und das Ergebnis unter spektraler Verzerrung leidet. Das Lernen der hohen Frequenzen der niederdiemnsionalen Problemdomäne in einem hochdimensionalen Funktionsraum führt dabei zur Abhilfe [29].

Mildenhall et al.[27] stellen fest, dass unter direkter Verwendung der Koordinaten (x, y, z, Θ, Φ), das Rendern von hochfrequenten Strukturen leidet. Dabei

stützen sie sich auf Rahaman et al.[30] die gezeigt haben, dass neuronale Netze einen Hang zum Erlernen von niederfrequenter Informationen haben. Außerdem zeigen sie, dass eine Abbildung der Eingangsdaten in einen hochdimensionalen Raum, zu einer genaueren Anpassung an Datensätze mit hochfrequenten Details führt.

Mit dieser Erkenntnis formuliert Mildenhall et al. die Fittingfunktion F_Θ zu

$$F_\Theta = F'_\Theta \cdot \gamma$$

um.

F'_Θ bleibt ein MLP. γ ist die Abbildung von $\mathbb{R}$ in den hochdimensionalen Raum $\mathbb{R}^{2L}$.

$$\gamma(p) = \left(\sin\left(2^0\pi p\right), \cos\left(2^0\pi p\right), \cdots, \sin\left(2^{L-1}\pi p\right), \cos\left(2^{L-1}\pi p\right)\right) \quad (5.1)$$

Die Funktion $\gamma(\cdot)$ wird auf jede Koordinate einzeln angewandt und alle Koordinaten zwischen [-1, 1] normalisiert. Für die Positionskoordinaten x wurde $L = 10$ gewählt, für die Blickrichtung d, $L = 4$. Je höher L gewählt wird, desto schmaler wird die Bandbreite des interpolations Kernel [31].

$$\gamma^k : \mathbf{p} \to \left(\sin\left(2^0\mathbf{p}\right), \cos\left(2^0\mathbf{p}\right), \sin\left(2^1\mathbf{p}\right), \cos\left(2^1\mathbf{p}\right), \ldots, \sin\left(2^k\mathbf{p}\right), \cos\left(2^k\mathbf{p}\right)\right)$$

5.1.3 Netzwerk Architektur

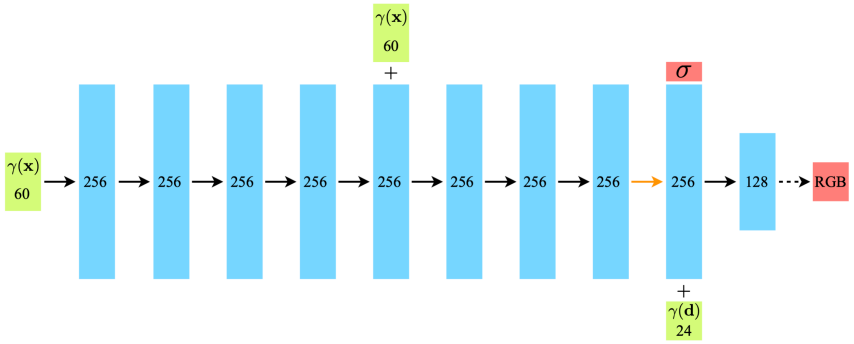


Abbildung 5.5: NeRF besteht aus einem Mehrschichtigen Perzeptron mit 8 Schichten im Hauptteil, gefolgt von zwei Schichten, welche den Finalen RGB-Wert ($\boldsymbol{c}$) bilden. Der erste Teil lernt die Ausbreitung im Raum abhängig der Positionsdaten $\boldsymbol{x}$, zum zweiten Teil wird der Richtungsvektor $\boldsymbol{d}$ gegeben, um blickrichtungsabhängigen Veränderungen zu erfassen. Zwischen erster und 5. Schicht gibt es eine sogenannte *Skip connection* welche das Erlernen feiner Strukturen verbessert. Grafik entnommen aus [27].

NeRF besteht im Prinzip aus zwei mehrlagigen Perzeptron die hintereinander Angeordnet sind. Dabei hat das vordere MLP eine Vielzahl mehr Schichten als das Hintere. Das erste hat 8 vollständig verbundene Lagen mit jeweils 256 Neuronen und bekommt nur die Kameraposition zu sehen. Die Ausgabe wird

mit den Blickrichtungen konkateniert und in ein MLP mit nur zwei Schichten und verringerter Anzahl Neuronen gegeben. Die verringerte Kapazität des zweiten MLP führt zu einer impliziten Regularisierung, welche das Netz dazu zwingt sich auf die Position der Oberfläche zu konzentrieren und weniger mit Veränderung der Oberflächeneigenschaften die Erscheinung der Szene aus Sicht der Kamera, zu ändern. Wird die Blickrichtung in einer früheren Schicht beigegeben, hat das Netz bis dahin noch nicht die richtige Dichteverteilung gelernt, und versucht Unregelmäßigkeiten der Objektoberflächen durch richtungsabhängige Oberflächeneffekte auszugleichen. Die folgende Veröffentlichung "NeRF++"5.3 geht darauf genauer ein.

Das neuronale Netz $F_\Theta : (\mathbf{x}, \mathbf{d}) \rightarrow (\mathbf{c}, \sigma)$ lernt abhängig von (x, y, z, Θ, Φ), eine implizite neuronale Repräsentation der Szene. Für ein beliebige Position (x, y, z, Θ, Φ) berechnet das MLP einen Farbwert (RGB) und eine Dichte (σ).

Wie in Abbildung 5.5 zu sehen, wird die 5. Schicht nochmals mit den Positionsdaten $\gamma(\boldsymbol{x})$ konkateniert. Das nennt sich *skip connection* und verbessert das Lernen von feinen Strukturen, indem hochfrequente Details die in den ersten Schichten verloren gehen, dem Netz in spätere Schichten nochmal hinzugefügt werden.

5.1.4 Volumen Rendering

Um aus der Szenenbeschreibung, kodiert als neuronales Netz/NeRF, ein Bild darzustellen, werden von einer Kameraposition aus, Strahlen erzeugt, die durch das gelernte Volumen fallen. (Ray Marching).

Zur Bestimmung der endgültigen Pixelfarbe, wird aus den Dichte und Farbwerten die endgültige Pixelfarbe bestimmt. Die Dichte kann als Wahrscheinlichkeit mit der ein Lichtstrahl an einer Position im Raum absorbiert wird gesehen werden. Absorbiert eine Position im Volumen den Strahl vollkommen, befindet sich diese Stelle an einer Objektoberfläche und färbt den Sichtstrahl entsprechend ein. Die Dichte des Volumens $\sigma(x)$ entlang eines Strahls $\boldsymbol{r}$, bestimmt wie stark ein Pixel eingefärbt wird. Dabei wird der Strahl von Fern zu Nah abgetastet und aufsummiert. Das ermöglicht die Erfassung von semitransparenten Objekten zwischen Kamera und strahlterminierender Objektoberfläche.

$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ ist ein in 5.1.1 generierter Sichtstrahl eines Pixels p abhängig der Kameraposition $\boldsymbol{o}$ und der Blickrichtung $\boldsymbol{d}$. Dann berechnet sich die Farbe eines generierten Pixels wie folgt [27]:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt, \text{ mit } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right) \quad (5.2)$$

Der Parameter t des Integrals tastet den Strahl innerhalb einer Region, die durch t_n und t_f begrenzt wird, ab. Die Funktion $\mathrm{T}(t)$ beschreibt, wie stark der Sichtstrahl schon zwischen t_n und der gesampelten Stelle t „eingefärbt“ wurde.

Um eine neue Ansicht der gelernten Szene zu synthetisieren, muss $C(\boldsymbol{r})$ für jeden Strahl, der durch ein Pixel der virtuellen Kamera fällt, berechnet werden.

5.1.5 Hierarchisches Volume Sampling

Da für jedes einzelne Sample entlang des Sichtstrahls das gesamte MLP evaluiert werden muss, verkürzt das Einsparen von Samples die Renderzeit. Mildenhal et al. [27] optimiert für ein effizienteres Sampeln zwei MLP gleichzeitig. Das eine *NeRF coarse* wird mit N_c gleichmäßig verteilten Positionen abgetastet, dessen Ergebnis bestimmt auf welche Stellen sich die Sample von *NeRF fine* konzentrieren.

Es werden entlang eines Sichstrahls mehrere Positionen abgetastet und durch Alpha Blending von Fern nach Nah für den finalen Farbwert kombiniert.

Der größte Teil des durchschrittenen Volumen wird dabei eine niedrige Dichte haben und damit wenig bis gar nichts zur Bildfarbe beitragen. Die zum Ergebnis beitragenden Abschnitte des Strahls haben eine Dichte $\sigma > 0$.

Um leeren Raum möglichst effizient zu erkennen und die Genauigkeit an Oberflächen zu erhöhen, werden zwei Netzwerke gelernt, ein Grobes und ein Feines. Das grobe Netz, evaluiert in regelmäßigen Abständen entlang des Lichtstrahls Farb-, und Dichtewerte und erstellt durch Normalisierung $\hat{w}_i = w_i / \sum_{j=1}^{N_c} w_j$ eine Wahrscheinlichkeitsdichtefunktion. Das feine Netz hat die gleiche Architektur wie das grobe, wird allerdings abhängig der Wahrscheinlichkeitsdichte des groben Netzes, gezielter abgetastet. Für jedes Sample des groben Netz wird der Anteil an der finalen Farbe C_c bestimmt. Daraus wird eine Wahrscheinlichkeitsdichte berechnet, die beschreibt wie hoch die Chance ist, dass bei einem Abtasten des feinen Netzes eine Position gesamplet wird, die zu einem hohen Anteil an der Pixelfarbe beiträgt.

$$\hat{C}_c(\mathbf{r}) = \sum_{i=1}^{N_c} w_i c_i, \quad w_i = T_i \left(1 - \exp\left(-\sigma_i \delta_i\right)\right) \tag{5.3}$$

Abhängig dieser Dichtefunktion tastet das feine Netzwerk mit "Abtastung durch inverse Transformation" (inverse transform sampling [32]), den entsprechenden Lichtstrahl durch das Volumen ab. Das Endergebnis setzt sich aus den Abtastungen des groben und des feinen Netzes zusammen $N_c + N_f$.

5.1.6 Loss und Validierung

Der Loss L berechnet sich aus dem dem quadrierten Fehler beider Netze, *NeRF coarse* und *NeRF fine*.

$$\mathcal{L} = \sum_{\mathbf{r} \in \mathcal{R}} \left[\left\| \hat{C}_c(\mathbf{r}) - C(\mathbf{r}) \right\|_2^2 + \left\| \hat{C}_f(\mathbf{r}) - C(\mathbf{r}) \right\|_2^2 \right] \tag{5.4}$$

Für die Rückkopplung beim trainieren eines NeRF-Netz, werden Bilder aus bekannten Blickrichtungen generiert und verglichen, wie stark diese der wahren Aufnahme entsprechen.

Das neuronale Netz lernt die Verteilung der Lichtstrahlen im Raum, da diese im Idealfall keine widersprüchliche Lösung ergibt, und damit den Verlust des neuronalen Netzwerks beim Trainieren minimiert.

$$\hat{C}_c(\mathbf{r}) = \sum_{i=1}^{N_c} w_i c_i, \quad w_i = T_i \left(1 - \exp\left(-\sigma_i \delta_i\right)\right) \tag{5.5}$$

5.1.7 Ray Marching um Geometrie zu generieren

NeRF ordnet jeder Stelle im Volumen eine Dichte und einen Farbwert zu. Ist die Dichte gering, absorbiert oder reflektiert diese Stelle weniger Licht und beeinflusst die Erscheinung in dieser Blickrichtung nicht. Hat eine Stelle eine hohe Dichte, färbt diese Stelle den Lichtstrahl ein. Das ist bei einer lichtundurchlässigen Stelle, also einer opaken Objektoberfläche im Volumen der Fall. Befindet sich ein semitransparentes Objekt oder Gas im Lichtstrahl, hat dieser eine Dichte $0 < \sigma < 1$ und weist damit zum Anteil von σ dem Lichtstrahl den entsprechenden Farbton der Stelle zu.

Um aus einem NeRF eine Geometrie der Szene zu extrahieren, wird zuerst ein Voxelvolumen erstellt. Jede Position eines Voxels wird im NeRF abgefragt und in eine Zelle eingetragen, das Gleiche wird für den Farbwert getan. Aus dem Voxelvolumen kann dann mit einem Marching Cubes Algorithmus [33] ein Mesh generiert werden, welches in jeder gängigen 3D-Software weiter verarbeitet werden kann.

5.2 Nerf in the Wild

Während NeRF mit Bildern von statischen Objekten, die unter kontrollierten Bedingungen aufgenommen wurden, gut funktioniert, ist es nicht in der Lage, viele allgegenwärtige, reale Phänomene in unkontrollierten Bildern zu modellieren. *Nerf in the Wild* macht NeRF robust gegenüber variabler Beleuchtung oder vorübergehenden Verdeckungen.

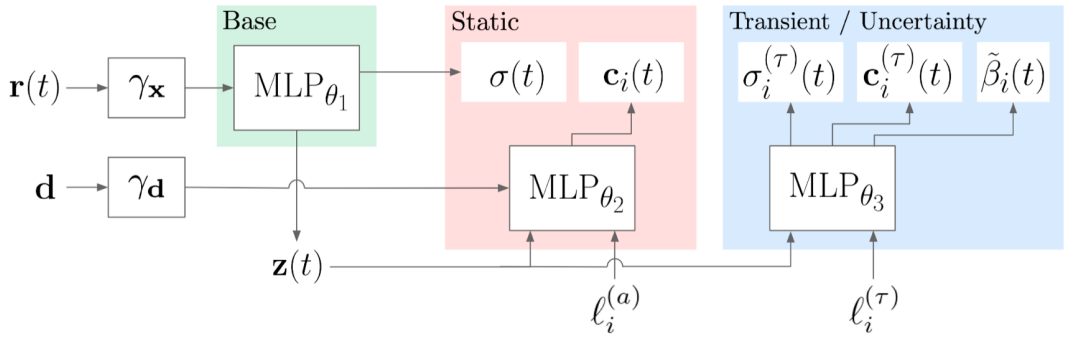
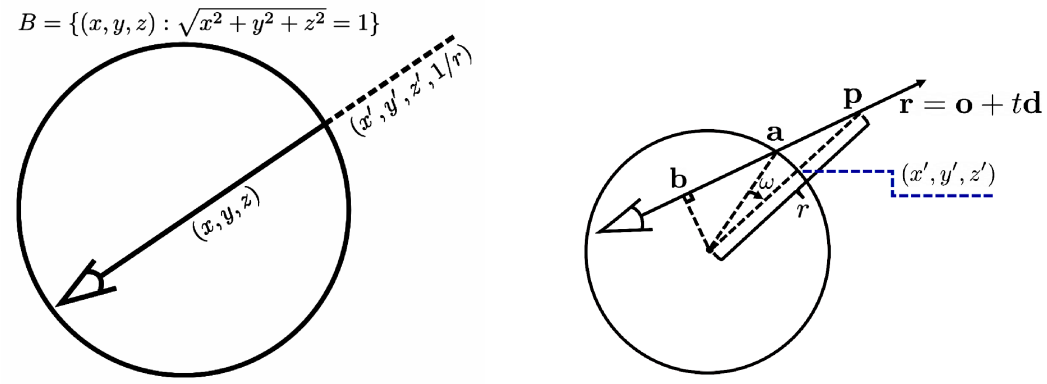


Abbildung 5.6: Architektur des NeRF-W-Modells. Bei gegebener 3D-Position $r(t)$, Blickrichtung d, Erscheinungseinbettung l_i^a und transienter Einbettung l_i^τ erzeugt NeRF-W differentielle Dichtewerte $\sigma(t)$, $\sigma_i^{(\tau)}(t)$, Farben $c_i(t)$, $c_i^{(\tau)}(t)$ und Unsicherheitsfaktor $\beta_i(t)$. Die statische Opazität $\sigma(t)$ wird generiert, bevor das Modell auf die Erscheinungseinbettung l_i^a konditioniert wird, um sicherzustellen, dass die statische Geometrie über alle Bilder hinweg geteilt wird.

"NeRF in the Wild" erweitert NeRFs Term zur Oberflächenbeschreibung um einen Unsicherheitsfaktor, der ermöglicht, sich zwischen Aufnahmen verändernde Lichtverhältnisse oder in einzelnen Bildern erscheinende Verdeckungen zu lernen.

NeRF [27] kann Lichtstrahlen die zwar vom selben Punkt in der Szene Reflektiert werden, sich aber in ihrer Erscheinung durch einfallendes Licht unterscheiden nicht modellieren. Das stellt ein großes Problem für das Lernen von Oberflächen mit starken Reflektionen wie z.B. Glanzlichtern dar. Eine zweite Erweiterung ist die Möglichkeit, temporär erscheinende Artefakte wie, Personen die durch die Aufnahme Laufen, ignorieren zu können. Beide Artefakte, richtungsabhängige Oberflächenerscheinung und temporäre Okkluder sind prominent in Datensätzen die in einer unkontrollierten Umgebung aufgenommen wurden. Beispiel dafür sind Bildersammlungen aus dem Internet von Sehenswürdigkeiten, oder Aufnahmen im Freien mit sich durch Wolkenbewegung verändernden Lichtverhältnissen.

5.3 NeRF++



(A) Nerf++ wendet außerhalb der Einheitskugel eine andere Parametriesierung als innerhalb an.

(B) Herleitung für den auf die Stelle (x', y', z') projizierten Punkt p der außerhalb der Einheitskugel mit bekanntem $\frac{1}{r}$ liegt.

Abbildung 5.7: NeRF++ stellt eine alternative Parametrisierung des gelernten Raums vor. Dabei wird eine Sphere um den Haupttteil der Szene gelegt um den Vordergrund vom Hintergrund getrennt zu lernen.

5.3.1 Verbesserte Parametrisierung der Raumkoordinaten.

Eine große Zahl der auf NeRF aufbauenden Veröffentlichungen beschäftigen sich damit das Sampling zu verbessern. NeRF++ trennt eine *inward facing/360°* Szene in zwei Bereiche, außerhalb und innerhalb einer Hemisphere liegend, ein. Dabei verwendet es ein MLP für innen und eines für außen. [34].

NeRF versucht die gesamte zu lernende Szene in einen gemeinsamen Raum zu kodieren. Dabei kommt es zu einer Abwägung zwischen dem Detailgrad des Vordergrundes und der maximalen Ausbreitung des Hintergrunds.

Bei der Aufnahme eines 360° NeRF kann angenommen werden, dass sich die zu betrachtende Szene innerhalb der konvexen Hülle der Kamerapositionen liegt, der Hintergrund außerhalb.

Ursprüngliches NeRF projiziert weit von der Kamera entfernte Punkte auf ähnliche Werte, da die Disparität gering ist. Zur Verbesserung wird bei NeRF++ Hintergrund und Vordergrund getrennt kodiert und gelernt.

Zhang et al.[34] beschreiben NeRF++ als Verfahren zur "novel view Synthesis", das sowohl begrenzte 360° Szenen beschreiben kann, wie auch vorwärts gerichtete Ansichten. Dabei wird ein MLP trainiert, das Dichte und Farbe entlang eines Lichtstrahls aus einem Bildersatz erlernt. Dabei ist die Dichte, welche Oberflächen beschreibt, von der Blickrichtung unabhängig, die Farbe der Volumen allerdings blickrichtungsabhängig.

NeRF++ behandelt zwei Probleme:

1. Mehrdeutigkeit von Erscheinungen in Radianzfeldern
2. Parametriesierung von in der Tiefe unbegrenzten 360° Szenen.

In Bezug auf die Mehrdeutigkeit von Erscheinungen stellt Zhang et al. die Frage, weshalb NeRF nicht dazu tendiert, eine "Abkürzung" zu lernen, bei der die zu optimierende Ansicht als Halluzination vor die Kamera gesetzt wird. Dabei würden von der Form abhängige Unterschiede aus verschiedenen Blickrichtungen als blickrichtungsabhängiger Effekt auf einer Kugel interpretiert. Dazu müsste das neuronale Netz die Dichte nicht abhängig der echten Form lernen, sondern kann eine Kugel um die Szene setzen, deren blickrichtungsabhängiges Aussehen die wahre Form verschleiert.

Zhang et al. begründen die Resistenz gegen fehlerhaftes erlernen der Form durch zwei Faktoren.

1. Mit wachsender Abweichung der Dichte σ von der echten Form muss die erlernte Farbe c höherfrequentere richtungsabhängige Effekte lernen, als dies bei einer echten liberischen Oberfläche nötig wäre. Beim Erlernen der echten Form wird das Lichtfeld "glatter" und es werden daher weniger hohe Frequenzen gebraucht. Die durch seine Größe limitierte Anpassungsfähigkeit des neurnalen Netzes, führt zwangsweise zum Erlernen von möglichst glatten, niederfrequenten Abbildungen des Lichtfeldes.

2. Der zweite Faktor liegt in der Architektur des Netzes, da die ersten Ebenen nur die Position im Raum sehen. Richtungsinformationen des abgetasteten Strahls werden erst in den letzten Ebenen des Netzwerks eingefügt. Der "Embedding Space" der Richtungskoordinaten hat hierbei auch weniger und niedrigere Komponenten als die Positionskoordinaten.

5.4 Deformierbare NeRF

Das Konzept Nerf lässt sich mit einer Vielzahl von zu lernenden Parametern erweitern, von Oberflächeneigenschaften bis zur Beleuchtung der Szene. Die populärste Fortführung ist hierbei die Einführung der Zeit, welche ein Deformation bzw. Bewegung in der Szene ermöglicht.

Um Robustheit gegenüber Deformationen der Szene während der Aufnahme der Trainingsdaten zu erlangen, kann neben den Kameraposen noch der Aufnahmezeitpunkt t als Bildindex in die Eingabedaten einfließen. Dieser Ansatz wird von NeRFlow [35], NSFF [36] und Video-NeRF [37] beschrieben.

Ein weiterer Ansatz ist es, ein statisches NeRF mit einem Deformations-Netz zu kombinieren. Dabei wird ein statischen Template und eine davon abhängige Deformation passend zum einzelnen Eingangsbild generiert. [38] Zu jedem Eingangsbild wird ein Latent Vektor [39] mitgelernt, der für das Deformations-Netz am besten beschreibt, wie das Eingangsbild im Vergleich zum Template liegt.

5.4.1 Elastic Regularisation

Die Einführung eines weiteren Parameters erhöht die Kapazität eines NeRF, eine Szene zu beschreiben, das heißt allerdings nicht, dass es das Trainieren erleichtert. Eine höherer Kapazität führt zu mehr Möglichkeiten sich während des Trainings in einem lokalen Minimum "zu verrennen". Das deformierende Netzwerk führt Mehrdeutigkeit ein, da eine Translation von der Kamera weg und eine Skalierung gleich wahrgenommen werden. Um diese Überanpassung (*overfitting*) zu vermeiden, wird eine Regularisierung eingeführt, die die Kapazität des Deformations-Modells reduziert.

Ein Beispiel hierfür ist der von NSFF [40] verwendete *Scene Flow*.

Der Szenenfluss ist das dreidimensionale Bewegungsfeld von Punkten in der Welt, so wie der optische Fluss das zweidimensionale Bewegungsfeld von Punkten in einem Bild ist. [41]. Das Bewegungsfeld beschreibt wie sich am Zeitpunkt i eine Position (x, y, z) hin zum nächsten Zeitschritt j ändert. Es muss eingehalten werden, dass ein Zeitpunkt i stetig aus seinen benachbarten Zuständen $j \in N(i)$ hervorgeht. Das heißt das Bewegungsfeld auf Zeitpunkt j in Richting i angewandt, muss eine Deformation von Zeitpunkt i zu j Rückgängig machen.

Deformable Neural Radiance Fields [42] Regularisierung bevorzugt eine rigide Deformation, also eine Kombination aus Translation und Rotation. Ein Auseinanderziehen oder Zusammendrücken der Szene soll vermieden werden und kann damit eine Skalierung von einer Bewegung, hin oder weg zur Kamera, unterscheiden.

5.4.2 Nerfies

Nerfies ist eine Anwedung von *Deformable Neural Radiance Fields* [42] und hat das Ziel ein dreidimensionales Selbstportrait aus einer Smartphone-Videoaufnahme zu generieren. Dafür filmt sich der Anwender selbst mit der Frontkamera eines Smartphones. Es ist nicht möglich, dass hierbei die Kopf- und Oberkörperposition still gehalten werden kann. NeRF setzt allerdings voraus, dass sich die Szenen zwischen einzelnen Aufnahmen des Datensatzes nicht ändern. NeRf in

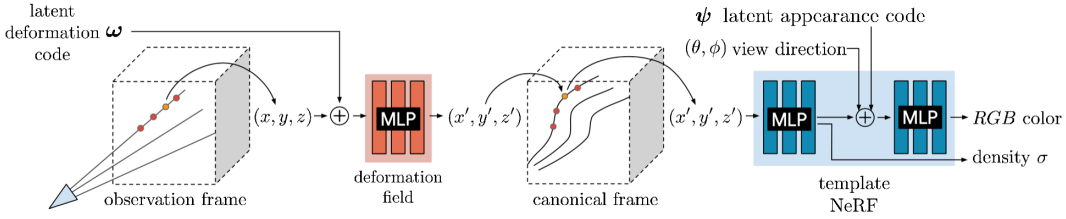


Abbildung 5.8: Das aufgenommene Bild wird mit ω dem Deformations-MLP übergeben, welches den Sichtstrahl auf die entsprechende Stelle im erlernten starren Template umlenkt. Das eigentliche NeRF wird mit den kanonischen Positionen des Templates gelernt.

the Wild [43] führt einen Transient-Term ein, der temporäre Okluder in der Szene herausfiltert. Das zwingt NeRF allerdings nur dazu, Änderungen der Szene zu ignorieren, nicht zu verstehen.

Nerfies Ziel ist es NeRF so zu erweitern, dass es über die Aufnahmen hinweg sich ändernde Körperositionen erkennt und ausgleichen kann. Das wird erreicht, indem ein weiteres Netz trainiert wird, welches eine Deformation der starren Pose der Szene lernt. Diese starre Pose entsteht implizit aus den Trainingsdaten, indem für jedes Videobild eine rigide Deformation gesucht wird, die mit möglichst wenig Unterschied zwischen Pose im Bild und Referenzpose transformieren kann.

Diese kanonische (aus der Struktur ergebende) Referenzpose aller Einzelbilder wird von einem MLP abhängig eines Deformationscode ω gelernt. Das Ergebnis ist eine Reparametrisierung x', y', z' des Sichtstrahls, mit der das NeRF-MLP trainiert wird. Der gesamte Prozess ist in Abbildung 5.8 zu sehen.

Der Trainingsprozess eines Nerfies umfasst somit nicht nur die Optimierung des NeRF-MLP, sondern auch die des vorgeschalteten Deformations-MLP.

Daraus ergibt sich folgende Formulierung (5.6). Das Ergebnis eines Nerfies berechnet sich aus Position $\boldsymbol{x}$, Blickrichtung $\boldsymbol{d}$, einem Erscheinungscode [44] und den Deformationscode. Der Erscheinungscode $\boldsymbol{\phi}$ gleicht variierende Helligkeit und Weißabgleich zwischen einzelnen Bildern aus. Der Deformationscode ω, beschreibt wie die Szene im aktuellen Bild sich vom gelernten Template unterscheidet.

$$G\left(\mathbf{x}, \mathbf{d}, \boldsymbol{\psi}_i, \boldsymbol{\omega}_i\right) = F\left(T\left(\mathbf{x}, \boldsymbol{\omega}_i\right), \mathbf{d}, \boldsymbol{\psi}_i\right) \tag{5.6}$$

Zuerst wird die Transformation $T_i : x \rightarrow x'$ der beobachteten Koordinaten in den kanonischen Template Raum vorgenommen. Das übernimmt ein MLP $T : (\boldsymbol{x}, \boldsymbol{\omega}_i)$
rightarrowx' mit Hilfe eines für jeden Frame optimierten Latentcode ω_i. Danach wird das eigentliche NeRF-MLP ausgewertet, welches Farbwert und Dichte bestimmt.

Die Idee den Deformationscode implizit aus den Trainingsdaten mit zu lernen, ist von [45] adaptiert.

5.5 NeRF–

NeRF– zeigt, dass das Vorberechnen der Kameraparameter nicht nötig ist. Intrinsische und extrinsische Kameraparameter der einzelnen Bilder werden im Trainingsvorgang des NeRF mitgelernt. Damit fällt der COLMAP Arbeitsablauf als vorbereitender Schritt weg, die Trainingsdaten bestehen nur aus einem Bilddatensatz.

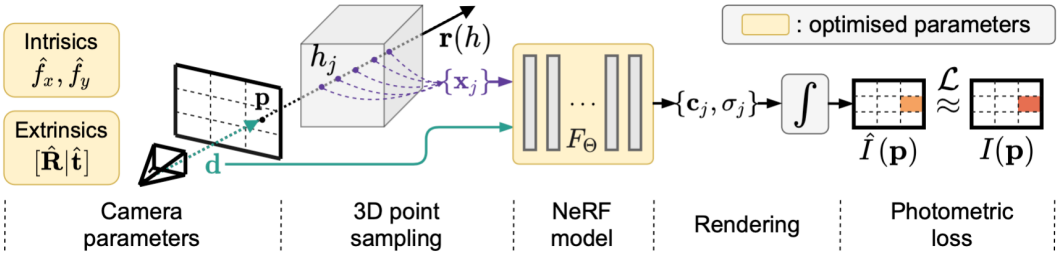


Abbildung 5.9: Der NeRF– Prozess optimiert das NeRF-MLP und die Kameraparameter der Eingabebilder durch Minimierung des fotometrischen Rekonstruktionsfehlers. Das gesamte Vorgehen wird Ende-zu-Ende trainiert, ohne eine vorherige Bestimmung der Kameraparameter. Grafik entnommen aus [46]

Einige vorherige Ansätze Kameraparameter mit neuronalen Netzen zu lernen finden sich in [47]. Ein weiterer Versuch Kameraparameter mit NeRF zu lernen ist BARF: Bundle-Adjusting Neural Radiance Fields. [48]

NeRF– optimiert drei Module gleichzeitig, neben dem NeRF-MLP, wird für jedes Bild noch die Brennweite und die Pose $(\boldsymbol{x}, \boldsymbol{d})$ gesucht. Die Ergebnisse aller drei Module werden zusammengeführt um mit einem Volumen-Render eine neue Ansicht zu generieren. Der Fehler der synthetisierten Ansicht im Vergleich zur Ground-Truth, fließt in alle drei Module zurück und optimiert diese für ein besseres Gesamtergebnis. Die einzelnen Module tauschen keine Informationen aus, können daher nicht voneinander lernen.

Kapitel 6

Experimente mit NeRF

Für NeRF [27] und einige Weiterführungen sind Beispielimplementierungen veröffentlicht.

Als Experimente wurden folgenden NeRF Implementierungen betrachtet:

- `github.com/kwea123/nerf_pl` (Pytorch) basierend auf der Implementierung der Autoren des original NeRF `github.com/bmild/nerf` (Tensorflow)
- `github.com/google/nerfies` (Jax)
- `github.com/ActiveVisionLab/nerfmm` (Pytorch)

6.1 Experimentier Umgebung

Da NeRF mit relativ großen Datenmengen und hohen Rechenressourcen arbeitet, ist das Experimentieren auf dem Heimcomputer meist nicht möglich.

Am Institut für Künstliche Intelligenz der Hochschule der Medien (HdM) gibt es speziell dafür eingerichtete Workstations, welche mit 4 Nvidia GTX 2080 mit jeweils 11GB VRAM und 120GB Arbeitsspeicher ausgerüstet sind. Wenn mehr Leistung nötig ist, bietet Google Colab [49] eine Ausweichmöglichkeit mit *data center* GPUs, allerdings ohne beständigen Speicherplatz und mit begrenzter Laufzeit.

Die Workstation der HdM ist für die Durchführung von Experimenten im Rahmen einer Vorlesung eingerichtet. Um unabhängig von Systemadministratoren zu sein, wird Docker als Containervirtualisierungsumgebung angeboten. Docker Container geben eine Unabhängigkeit vom Host System und bieten eine einfache Konfiguration von Softwareabhängigkeiten und Vorinstallationen.

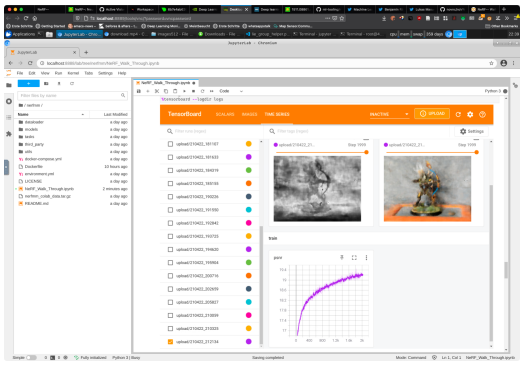


Abbildung 6.1: Jupyter Notebook und Tensorboard auf einer remote Workstation die über VNC gesteuert wird.

ml-workspace [50] ist ein Docker-Container der die remote Arbeit an Machine Learning Projekten erleichtert. Der ml-workspace kommt

vorkonfiguriert mit einer kompletten Entwicklungsumgebung rund um jupyer notebooks [51] welche große Beliebtheit bei *machine learning practitioners* gefunden haben. Git, Visual studio Code als IDE im Webbrowser und alle gängigen Programmbibliotheken für maschinelles Lernen sind vorinstalliert. noVNC [52], ein Virtual-Desktop Client der über einen Webbrowser zu erreichen ist, erleichtert das Bewerten von Bildern. Dazu kommen eine Vielzahl von Überwachungswerkzeugen wie Netdata oder Tensorboard.

Portforwarding mit SSH ermöglicht es, alle in einem Docker Container auf dem Remote Hostrechner gestarteten Server lokal im Webbrowser aufzurufen.

In der Praxis können mehrere Docker Container mit unterschiedlichen *Dependencies* auf dem Remote Hostrechner gestartet werden, die unterschiedliche Experimente ausführen und lokal in über verschiedene Jupyter Webinterfaces gesteuert werden.

Verbinden mit der Remote-Workstation über SSH und Wiederaufnahme einer bestehenden *tmux* Session:

```
    ssh -i ~/HdM_VPN/id_rsa.pub  -L localhost:8888:172.17.0.0:8888
    mk301@hal9k.mi.hdm-stuttgart.de -t tmux a
```

```
docker run -p 8080:8080 --runtime nvidia --env
    NVIDIA_VISIBLE_DEVICES="all" mltooling/ml-workspace-gpu:0.12.1
```

Ein neues Docker Image wird von *ml-workspace-gpu:latest* abgeleitet und erweitert:

```
FROM ml-workspace-gpu:latest
USER root
ENV PATH /usr/local/nvidia/bin:/usr/local/cuda/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64:$
    {LD_LIBRARY_PATH}
WORKDIR /workdir/
# Create the environment:
COPY /workdir/nerfies/requirements.txt .
RUN pip install -r requirements.txt
RUN apt-get update && apt-get install openexr libopenexr-dev colmap
     ffmpeg
RUN pip install --upgrade jax jaxlib==0.1.59+cuda102 -f https://
    storage.googleapis.com/jax-releases/jax_releases.html
RUN pip install git+https://github.com/google/nerfies.git
RUN pip install "git+https://github.com/google/nerfies.git#egg=
    pycolmap&subdirectory=third_party/pycolmap"
RUN pip install numpy==1.19.3 mediapipe tensorflow_graphics flax
    frozendict ipyplot nbdime imageio-ffmpeg
```

Docker Compose erleichtert das Starten von Containern mit einer Vielzahl an Argumenten:

```
version: '3'
services:
  nerfies:
    container_name: nerfies
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
```

```
      - "/home/mk301/:/home/mk301"
    ports:
      - "8888:8080"
    runtime: nvidia
    environment:
      - NVIDIA_VISIBLE_DEVICES=0,1
      - USER_GID="1016"
      - WORKSPACE_HOME=/home/mk301/
      - CONFIG_BACKUP_ENABLED=false
```

Registierung der Conda Environemnt als Jupyter Kernel.

```
source activate nerfies
python -m ipykernel install --user --name nerfies --display-name "
    Python (nerfies)"
```

Abbildung 6.1 zeigt die Arbeitsumgebung. Die lokale Maschine verbindet sich über VPN in die Hochschule, SSH tunnelt den richtigen Port, mit dem der Webserver des Docker Containers zu erreichen ist. Der Webbrowser der lokalen Maschine verbindet sich zum jupyter lab Server im Docker Container. Im Docker Container läuft noVNC, das einen virtuellen Desktop im lokalen Browser bereitstellt. Über den virtuellen Desktop lässt sich ein Browser im Docker dontainer zu beliebigen jupyter Instanzen verbinden ohne das Ports neu getunnelt werden müssen. Falls die Verbindung zwischen lokaler Maschine und remote im VPN abbricht, kann die Verbindung wiederhergestellt werden, ohne dass die Arbeitsumgebung neu gestartet werden muss.

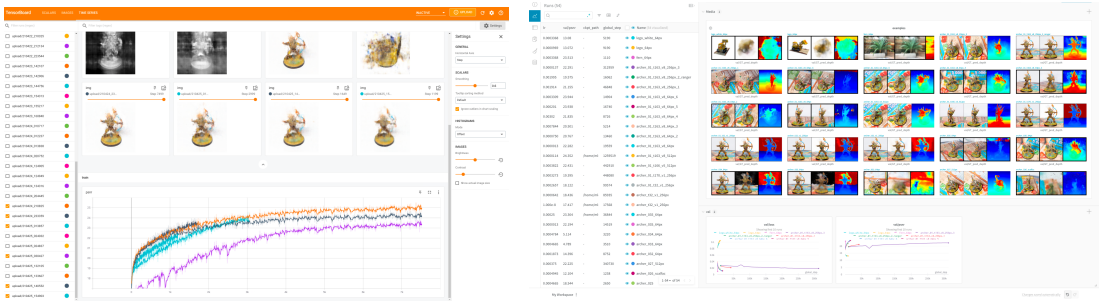
6.2 Tensorboard und Wandb

Für das Überwachen des Trainingsvorgangs wurde *Tensorboard* [53] und *Weights and Biases* [54] (wandb) verwendet. Tensorboard und wandb bieten den Werkzeugkasten um den Trainingsvorgang zu beobachten und zu bewerten. Es werden Graphen von gesammelten Metriken wie Loss oder Genauigkeit generiert. Bilder und 3D Daten können angezeigt werden. Tabellen vereinfachten das Sortieren und Filtern großer Mengen von Trainingsläufen.

Beides sind Client-Server-Anwedungen, wobei Tensorbaord lokal ausgeführt wird und wandb ein Webservice ist, der für akademische Zwecke kostenfrei ist. Tensorboard ist quelloffen und ist im *ml-workspace*, siehe 6.1, vorkonfiguriert. Alternative Logging Libarires die Tensorboard als Frontend verwenden sind unter anderem TestTubeLogger, MLFlow, Comet, Netptun.

In allen Fällen wird in der NeRF Python Anwendung die dazugehörige Library importiert und ein entsprechendes Logger Objekt erstellt welches dem pytroch Trainer-Objekt mit übergeben wird.

```
from pytorch_lightning import loggers as pl_loggers
tb_logger = pl_loggers.TensorBoardLogger('logs/')
wandb_logger = pl_loggers.WandbLogger(save_dir='logs/')
trainer = Trainer(logger=[tb_logger, comet_logger])
```



(A) Tensorboard Dashboard (B) wandb.ai Dashboard.

ABBILDUNG 6.2

6.3 Bilddatensätze

Die Datensätze lassen sich in drei Kategorien unterteilen.

- Vorwärts gewandte Szene (*forward facing scene*)
- Inwärts gewandte oder 360°Szene (*inward facing scene*)
- Selfie Video

ABBILDUNG 6.3: Archer Datensatz mit weißem Hintergrund, konnte mit keiner Implementierung zufriedenstellend trainiert werden. Weißer Hintergrund kann zu lokalen Minima führen die von den wenig farbigen Pixeln der Figur nicht vermieden werden können.

6.4 Durchführung

Jedes NeRF Experiment besteht aus folgenden Schritten:

1. Bilddatensatz aufnehmen oder rendern
2. Mit COLAMP Posen Kameraparameter bestimmen
3. Falls nötig Daten in Trainings/Test/Validierungsset einteilen
4. Trainieren
5. Werte zu Wandb/Tensorboard loggen

Als erstes ist aufgefallen, dass die veröffentlichten Implementierungen am besten als experimentelle Forschungssoftware gesehen werden sollten. Um die darin erforschten Techniken in einer Produktion anzuwenden ist viel Arbeit nötig. Vergleiche zwischen verschiedenen Projekten sind schwierig da nichts vereinheitlicht ist, und beschränken sich auf Bildähnlichkeitsmetriken. Vor allem Laufzeitvergleiche sind durch die Verwendung unterschiedlicher *Machine Learning Frameworks* auf verschiedenen Rechnersystemen mit unterschiedlichen Beschleunigungskarten (consumer GPU vs. datacenter ML solutions) nicht möglich.

Das führt dazu, dass die Implementierungen jeweils als eigenes Experiment mit spezifischen Abhängigkeiten gesehen werden müssen. Im speziellen die Parallelisierung von machine learning Anwendungen vergrößert Softwareprobleme.

6.4.1 Loss und PSNR

Abbildung 6.4 zeigt eine Auswahl der NeRF Training Runs. Ziel des Trainings ist, den Loss zu minimieren. Bei erfolgreichem Training sollte die Kurve am Anfang stark abfallen und sich einem Minimum annähern. Beginnt die Kurve erneut zu steigen, oder fällt im Vergleich zu anderen erfolgreichen Runs nicht ab, lässt das auf einen gescheiterten Versuch, das neuronale Netz zu trainieren schließen.

Abbildung 6.4: Tensorboard Dashboard

PSNR steht für *Peak Signal to Noise Ratio*, eine Messung wie genau ein Eingangsbild von einem NeRF wiedergegeben wird.

Abbilding 6.5 zeigt den Verlauf der PSNR von einigen Trainingsläufen.

Abbildung 6.5: Tensorboard Dashboard

Abbildung 6.6: Links ein Foto, in der Mitte ein interferiertes NeRF, rechts eine Falschfarbendarstellung der Tiefenkarte.

6.5 NeRF

Die ersten Experiemente wurden mit `github.com/kwea123/nerf_pl` [55] Codebase durchgeführt und werden im folgenden *nerf_pl* genannt. Für *nerf_pl* wurde die originale NeRF Implementation [27] von Tensorflow zu Pytorch portiert.

Erster Schritt ist es, für einen Bilddatensatz die extrinsischen und intrinsischen Kameraparameter zu bestimmen. Dafür wird aus dem *Local Light Field Fusion* (LLFF) Repo das `github.com/Fyusion/LLFF/blob/master/imgs2poses.py` Skript verwendet, welches mit Hilfe von COLMAP [4] Kameraposen und Brennweite berechnet.

COLMAP speichert nach Durchlaufen des SfM Vorgangs drei Datein ab, an denen wir interessiert sind:

1. ***cameras*** beinhaltet intrinsische Kameraparameter.

```
# Camera list with one line of data per camera:
#   CAMERA_ID, MODEL, WIDTH, HEIGHT, PARAMS[]
#   Number of cameras: 1
1 SIMPLE_RADIAL 2048 2048 12318.3 1024 1024 0.0284226

```

2. ***images*** beinhaltet alle extrinsischen Parameter und Feature Positionen im Bildraum.

```
# Image list with two lines of data per image:
#   IMAGE_ID, QW, QX, QY, QZ, TX, TY, TZ, CAMERA_ID, NAME
#   POINTS2D[] as (X, Y, POINT3D_ID)
# Number of images: 171, mean observations per image:
   10288.064327485381
```

```
171 0.798836 0.235407 0.152813 0.532065 0.258237 0.24798
    1.3036 1 archer2_4833.jpg
1245.12 1056.18. 157545 1245.12 1056.18 -1 936.954 ....  1
    archer2_4833.jpg
....

```

3. ***points3D*** beinhaltet alle Feature Positionen im Szenenraum mit Pixelfarbe. Sie werden zum Trainieren eines NeRF nicht benötigt. Eine Visualisierung der Punkte dient als Kontrolle des SfM Vorgangs.

```
# 3D point list with one line of data per point:
#   POINT3D_ID, X, Y, Z, R, G, B, ERROR, TRACK[] as (IMAGE_ID,
     POINT2D_IDX)
# Number of points: 218689, mean track length:
    8.0445701429884444
391664 -0.24613807009608912 1.6044902734577424
    2.2500856827071338 162 123 91 0.37121827189611223 168 13918
     165 15354
....

```

Bevor mit dem Trainieren begonnen werden kann, müssen alle Daten geladen und (Hyper)-Prameter konfiguriert werden. Um die Bilddaten und Kameraparameter zu importieren, wird für eine aus COLMAP kommende Szene das `nerf_pl/datasets/llff.py` Skript verwendet. Eine Alternative ist, Bilder und Kameradaten, die aus einer Blender Szene exportiert wurden, mit `nerf_pl/datasets/blender.py` zu laden. Beide Importmodule konvertieren die Koordinatensysteme in die von NeRF erwartet Achsenreihenfolge. Das *LLFF* Modul hat zwei Parameter, *spheric_poses* und *val_num*. *spheric_poses* ist eine Flag die zwischen einer nach innen gerichteten Szene und einer vorwärts gerichteten Szene, wie in Abschnitt 2.3 beschrieben, unterscheidet. *val_num* bestimmt wieviele Bilder für den Validierungsvorgang beiseite gelegt werden.

Nach dem Trainieren werden die finalen Gewichte abgespeichert.

Die Funktionalität der *nerf_pl* Implementierung ist dadurch aufgefallen, dass sie nicht nur neue Ansichten generiert, sondern auch das Volumen abtasteten kann, um ein 3d-Objekt mit Objektfarbe exportieren zu können. Dafür wird das NeRF mit den Koordinaten eines Voxelgitters evaluiert. Aus den zurückgegebenen Dichtewerten wird mit einem Marching Cubes Algorithmus [33] ein Polygon-Mesh erstellt. Ein so erstelltes Modell ist in Abbildung 6.8 zu sehen.

Abbildung 6.7 zeigt eine Kamerafahrt um die Szene, die aus dem MLP mit einem Volumen-Renderer berechnet wurde.

Ein Report bei wandb lässt sich unter `https://wandb.ai/sungam/nerf/reports/first-try-with-geometry-logged--VmlldzozNTAwODg` finden.

6.6 Nerfies

Nerfies ist eine Demonstration für *Deformable Neural Radiance Fields* [42]. Dabei wird ein Porträt mit dreidimensionaler Erscheinung generiert, die durch das Rendern mit einer bewegten Kamera deutlich wird.

Abbildung 6.7: Eine Kamerafahrt um eine Modellfigur, gerendert aus einem NeRF.

Eine Implementierung für Forschungszwecke wurde von Google auf Github unter github.com/google/nerfies veröffentlicht. Nerfies ist in jax [56] implementiert und basiert auf der von Google veröffentlichten JaxNerf [57] Implementierung. Der komplette Workflow, von Video Aufbereitung über NeRF Training hin zu Validierung ist in drei Google Colab Notebooks gegliedert.

Da Google Colab, alle 24h neu konfiguriert werden muss, ist es Ziel ein Dockerfile zu schreiben, der Nerfies auf einem eigenen Server zum Einsatz bringen kann.

Basierend auf dem *ml-tooling* Docker-Image, werden vorausgesetzte Abhängigkeiten im Dockerfile gesammelt und vorinstalliert, docker-compose regelt die Einbindung der GPUs und Portfreigabe für den Fernzugriff. Nach dem Start des Docker Containers kann Preprocessing, Training und Rendering ohne weitere Konfigurationen gestartet werden.

Während des Trainierens werden die Werte der *elastic regularization* ausgegeben:

Zur Bewertung wird das Trainieren gestoppt und die optimierten Gewichte der neuronalen Netze abgespeichert. Die daraus generierten NeRF Module werden dann mit einer frei generierten Kamerafahrt abgefragt. Man spricht von

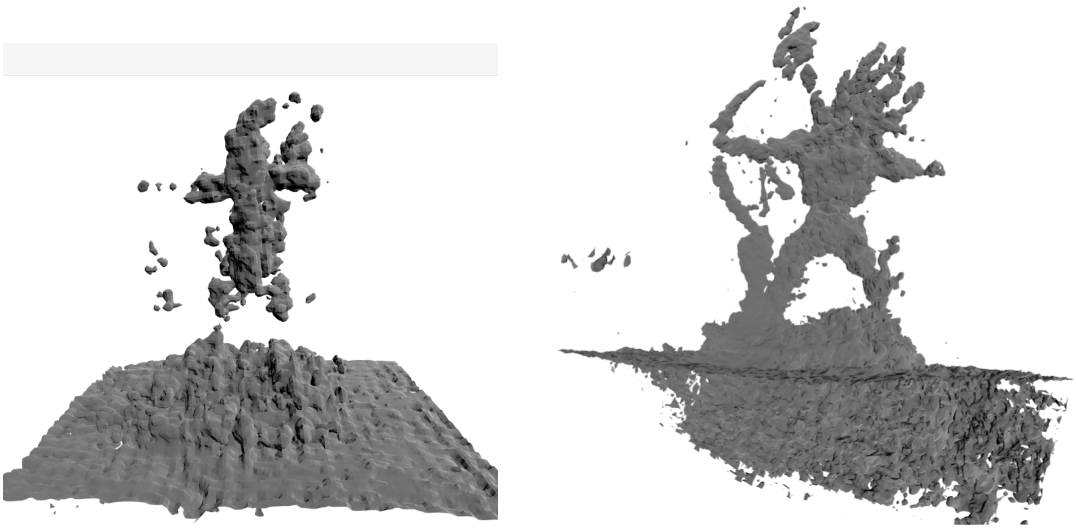


Abbildung 6.8: Extrahiert man aus NeRF ein Voxelvolumen, erhält man eine sehr verrauschte Objektoberfläche.

Abbildung 6.9: Synthetisierte Kamerafahrt zur Validierung eines Nerfies.

"Interferieren". Zur Interferenzzeit wird das NeRf-Modul mit einer Kamerapose $(x, y, z, \theta\phi)$ aus der Samples der Sichtstrahlen generiert werden, abgefragt. Damit lassen sich beliebige Kamerafahrten generieren. Um eine Zentrierung des aufgenommenen Gesichts zu erreichen, bestimmt Nerfies für jedes Bild mit google/mediapipe [58] Gesichtsmerkmale und Trianguliert diese in einer gemeinsamen 3D-Szene. Realtiv zu der aus der Bildsequenz bestimmten Gesichtsposition generiert Nerfies eine Kamerafahrt in Form einer waagerechen "8", was den parallax Effekt verstärkt. Für jedes Einzelbild der Kameraanimation wird das Nerf-Modul mit Kamerapose interferiert und am Ende zu einem Video zusammengefügt. Für eine bessere Beurteilung werden neben den RGB-Bildern noch Tiefekarten generiert. Dabei fällt das von [40] und [42] beschriebene Artefakt, dass Bewegungen die kolinear zur Kamerarichtung liegen zu lokalen Minima neigen, ins Auge. Hier ist das durch eine lange Nase in einer interferierten Kamerafahrt in Abbildung 6.9 zu erkennen.

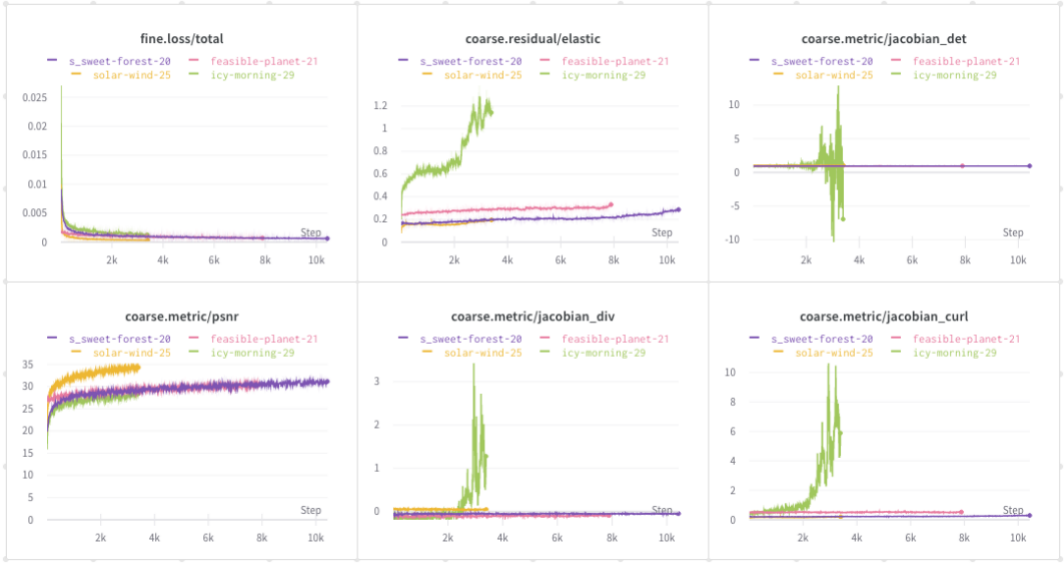


Abbildung 6.10: Starke Ausreißer der elastischen Regularisierung deuten auf einen degradierten Zustand hin.

6.7 Nerf–

Der erste Versuch NeRF– mit dem Archer Datensatz zu trainieren ist daran gescheitert, dass die veröffentlichte Implementierung nur mit forwärtsgerichteten Szenen trainiert werden kann.

Abbildung 6.11: *Front facing* Archer Datensatz mit NeRF– trainiert. RGB-Render hat viele Artefakte, da durch den einfarbigen Hintergrund die Kameraparameter nicht robust genug bestimmt werden können.

Nachdem das `github.com/ActiveVisionLab/nerfmm` Repository geklont ist, kann auf ml-workspace aufgebaut werden:

```
docker run -p 8080:8080 --gpus all mltooling/ml-workspace-gpu
# Innerhalb der Dockerumgebung
conda env create -f environment.yml
# Registierung der Conda Environemnt als Jupyter Kernel entweder mit:
conda install nb_conda -y
# oder mit:
```

```
source activate nerfmm
python -m ipykernel install --user --name nerfmm --display-name "
    Python (nerfmm)"
```

In der NeRF– Codebase *ActiveVisionLab/nerfmm*, liegt ein jupyter-Notebook bei. Nach dem Trainieren wird eine Kamerafahrt generiert, welche die räumliche Ausbreitung besser zu erkennen gibt. Da der interessante Teil die Kameralokalisierung ist, wird eine Animation generiert, welche die relativen Positionierungen der einzelnen Bilder über den Trainignszeitraum darstellt. Ein Standbild ist in Abbildung 6.12 zu sehen.

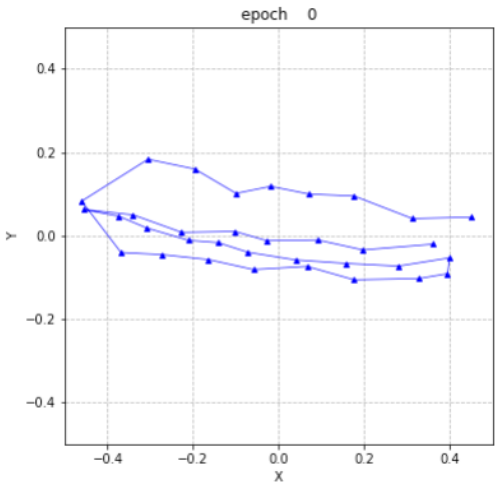


Abbildung 6.12: Eine orthografische Projektion der optimierten Kamerapositionen.

Das NeRF– jupyter Notebook visualisiert nach dem Trainingsvorgang die gefundenen Kamerapositionen über die Trainingszeit in einer Animation. Darin ist zu erkennen, dass die Vorderseite nicht von der Rückseite unterschieden werden kann. Es entsteht eine falsche symmetrische Annahme des "Archer"-Objekts, wenn alle Kameras in eine gemeinsame Richtung schauend optimiert werden. Da *Normalized Device Coordinates* für die Parametrisierung der Lichtstrahlen verwendet wird, ist das zu erwarten.

Eine Idee ist, alle Ansichten der Rückseite zu löschen, von den 56 Aufnahmen des Archer Datensatz, die ungefähr von einer Hemisphere um das Objekt herum aufgenommen wurden, blieben 22 übrig, die das Objekt von vorne gesehen haben. Das Ergebnis ist eine Art invertierte Szene, die an eine optische Täuschung (Hohlkehlen Porträt, schaut immer in Richtung des Betrachters) erinnert. Eine mögliche Erklärung ist, dass die Optimierung des NeRF-MLP zu schnell fortschreitet und ein lokales Maximum findet bevor die Brennweite und Posen genau genug bestimmt wurden. Man kann sich vorstellen, dass eine falsch bestimmte Brennweite als Art Tiefpass wirkt und somit falsche Posen begünstigt, vor allem bei einem Datensatz mit gleichfarbigem Hintergrund und einem fast symmetrischen Objekt im Zentrum.

Eine Überlegung ist, ob die Tendenz zu lokalen Maxima des NeRF-MLP während des Trainings herausgezögert werden kann, und damit die Optimierung von Brennweite und Posen genauer wird. Nerf– hat für die Optimierung

von MLP, Brennweite und Posen drei verschiedene Werte für die Schrittgröße des jeweiligen Gradientenverfahrens, die relativ zueinander angepasst werden können.

Die Parameter eines Trainingsvorgangs die nicht aus dem Datensatz hervorgehen, nennen sich Hyperparameter. Oftmals möchte man eine Reihe solcher Werte ausprobieren um die Auswirkungen auf den Trainingsverlauf besser zu verstehen. Das nennt sich Hyperparametersweep.

Wandb [54] bietet die Funktionalität, viele Trainingsläufe mit unterschiedlichen Parametern zu starten und gemeinsam auszuwerten. Dafür wird ein yaml-File geschrieben, der die einzelnen Parameternamen und die abzusuchenden Wertebereiche konfiguriert.

```yaml
method: bayes
metric:
  goal: maximize
  name: train/psnr
parameters:
  focal_lr:
    distribution: uniform
    max: 0.01
    min: 0.00025
  nerf_lr:
    distribution: uniform
    max: 0.01
    min: 0.00025
  pose_lr:
    distribution: uniform
    max: 0.01
    min: 0.00025
program: train.py
```

Abhängig der Ergebnisse vorherigen Trainingsläufe werden die Parameter angepasst um gegenseitige Einflussnahme der Parameter zu erkennen. Abbildung 6.13 zeigt einen in wandb generierten Report eines Hyperparametersweeps.

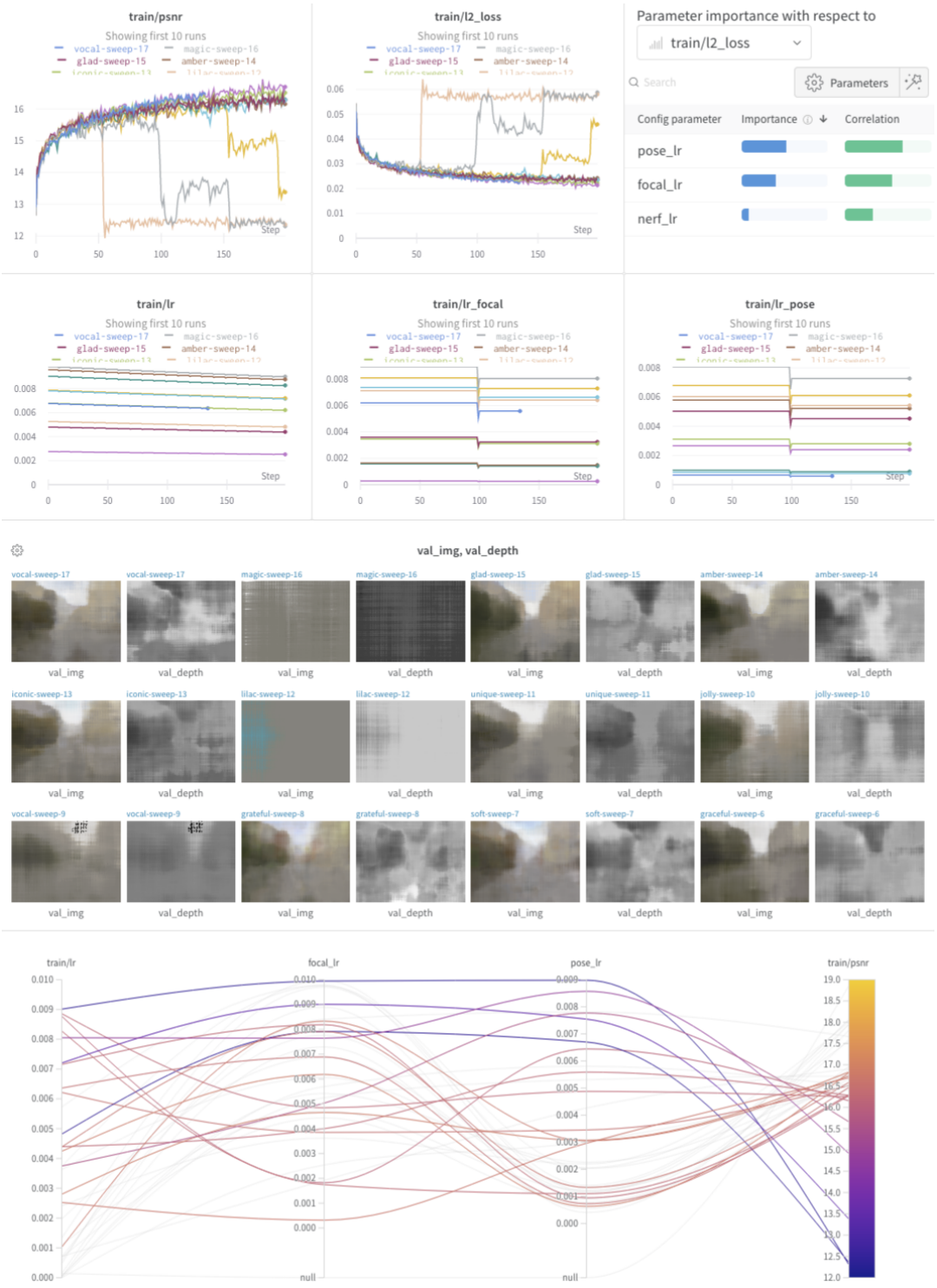


Abbildung 6.13: Ergebnis eines Hyperparametersweeps mit verschiedenen Schrittgrößen (*learning rate*). Rechts oben ist eine Bewertung der Einflüsse der untersuchten Parameter zu sehen.

Kapitel 7

Weiterführung

Eine Sammlung von möglichen Verbesserungen und Anwendungsszenarien.

7.1 Mixed Reality

NeRF [27] zeigt, dass die Genauigkeit der Szene hoch genug ist, um die gelernte Szene als Geometrie zu exportieren und mit virtuellen Objekten zu augmentieren. `github.com/kwea123/nerf_Unity/` [59] ist eine Implementierung die eine NeRF-Szene in Unity visualisiert.

"GRAF: Generative Radiance Fields for 3D-Aware Image Synthesis"[60] zeigt die Möglichkeit ein Compositing von mehrern Radianzfeldern nebeneinander vorzunehmen.

Eine gemeinsame Renderpipeline mit polygonalen Objekten oder Signed Distance Fields, die nicht erfordert, in einem Zwischenschritt aus dem neuronalen Netz eine Szenengeometrie zu generieren ist nicht bekannt.

7.2 Stereokonvertierung

Szenario: Man gebe eine Filmsequenz mit einem Mindestmaß an bewegter Kamera als Trainingsdaten vor. NeRF generiert das Bild für das fehlende Auge. Dafür wird eine verschobene und verdrehte zweite Ansicht der Szene gerendert. Zu Problemen kommt es, wenn die vorhandene Aufnahme nicht genug perspektivische Veränderung beinhaltet, um die Generierung von neuen Perspektiven zu ermöglichen, ohne Teile der Szene zu halluzinieren.

NeRf sollte mindestens in der Lage sein, eine Stützgeometrie erstellen zu können. Dafür muss eine einheitliche Tiefe zwischen den Bildern gegeben sein. Um die Anwendung robuster zu machen, muss die Anwendung in der Lage sein, in den Trainigsdaten nicht vorkommende Stellen der Szene zu halluzinieren oder *Inpainting* anwenden zu können.

Literatur

[1] B. G. Breitmeyer, „Parallel Processing in Human Vision: History, Review, and Critique“, in *Advances in Psychology*, J. R. Brannan, Hrsg., Bd. 86, North-Holland, 1. Jan. 1992. DOI: 10.1016/S0166-4115(08)61349-7.

[2] D. G. Lowe, „Distinctive Image Features from Scale-Invariant Keypoints“, *International Journal of Computer Vision*, Jg. 60, Nr. 2, Nov. 2004, ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94.

[3] (). „AliceVision | Photogrammetric Computer Vision Framework“, Adresse: https://alicevision.org/.

[4] J. L. Schonberger und J.-M. Frahm, „Structure-from-Motion Revisited“, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE, Juni 2016, ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.445.

[5] J. L. Schönberger, T. Price, T. Sattler, J.-M. Frahm und M. Pollefeys, „A Vote-and-Verify Strategy for Fast Spatial Verification in Image Retrieval“, in *Computer Vision – ACCV 2016*, Ser. Lecture Notes in Computer Science, S.-H. Lai, V. Lepetit, K. Nishino und Y. Sato, Hrsg., Bd. 10111, Cham: Springer International Publishing, 2017, ISBN: 978-3-319-54180-8. DOI: 10.1007/978-3-319-54181-5_21.

[6] (). „OpenCV: Introduction to SIFT (Scale-Invariant Feature Transform)“, Adresse: https://docs.opencv.org/master/da/df5/tutorial_py_sift_intro.html.

[7] R. J. Radke, *Computer Vision for Visual Effects*. Cambridge: Cambridge University Press, 2013, 398 S., ISBN: 978-0-521-76687-6.

[8] T. Lindeberg, „Scale-Space for Discrete Signals“, *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, Jg. 12, 1. Apr. 1990. DOI: 10.1109/34.49051.

[9] W. Burger und M. J. Burge, *Digitale Bildverarbeitung*, Ser. X.Media.Press. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, ISBN: 978-3-642-04603-2. DOI: 10.1007/978-3-642-04604-9.

[10] A. Geiger, „Structure from Motion“, **presentedat**Computer Vision Lecture (Universität Tübingen), 2021, Adresse: https://uni-tuebingen.de/fakultaeten/mathematisch-naturwissenschaftliche-fakultaet/fachbereiche/informatik/lehrstuehle/autonomous-vision/lectures/computer-vision/.

[11] R. Szeliski, „Structure from Motion“, in *Computer Vision: Algorithms and Applications*, Ser. Texts in Computer Science, R. Szeliski, Hrsg., London: Springer, 2011, ISBN: 978-1-84882-935-0. DOI: 10.1007/978-1-84882-935-0_7.

[12] M. Faraday, „LIV. Thoughts on Ray-Vibrations“, *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Jg. 28, Nr. 188, 1846.

[13] A. Gershun, „The Light Field“, *Journal of Mathematics and Physics*, Jg. 18, Nr. 1-4, Apr. 1939, ISSN: 00971421. DOI: 10.1002/sapm193918151.

[14] David Kriesel, *Ein Kleiner Überblick Über Neuronale Netze*. 2007. Adresse: http://www.dkriesel.com.

[15] O. Trekhleb. (22. März 2021). „Trekhleb/Homemade-Machine-Learning“, Adresse: https://github.com/trekhleb/homemade-machine-learning.

[16] A. Barron, „Barron, A.E.: Universal Approximation Bounds for Superpositions of a Sigmoidal Function. IEEE Trans. on Information Theory 39, 930-945“, *Information Theory, IEEE Transactions on*, Jg. 39, 1. Juni 1993. DOI: 10.1109/18.256500.

[17] A. Kratsios und L. Papon. (27. Jan. 2021). „Quantitative Rates and Fundamental Obstructions to Non-Euclidean Universal Approximation with Deep Narrow Feed-Forward Networks“, Adresse: http://arxiv.org/abs/2101.05390.

[18] F. Rosenblatt und A. I. Kitov, „Frank Rosenblatt Publications“, Adresse: http://newcatalog.library.cornell.edu/catalog/5162789.

[19] R. Rojas, *Neural Networks: A Systematic Introduction*. Berlin ; New York: Springer-Verlag, 1996, 502 S., ISBN: 978-3-540-60505-8.

[20] S. Haykin und S. S. Haykin, *Neural Networks and Learning Machines*, 3rd ed. New York: Prentice Hall, 2009, 906 S., ISBN: 978-0-13-147139-9.

[21] E. Charniak, *Introduction to Deep Learning*. Cambridge, Massachusetts: The MIT Press, 2018, 174 S., ISBN: 978-0-262-03951-2.

[22] I. Goodfellow, Y. Bengio und A. Courville, *Deep Learning*, Ser. Adaptive Computation and Machine Learning. Cambridge, Massachusetts: The MIT Press, 2016, 775 S., ISBN: 978-0-262-03561-3.

[23] S. Wang, T. Zhou und J. A. Bilmes, „Bias Also Matters: Bias Attribution for Deep Neural Network Explanation“,

[24] R. Girshick, J. Donahue, T. Darrell und J. Malik, „Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation“, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Juni 2014.

[25] C. Dong, C. C. Loy, K. He und X. Tang. (31. Juli 2015). „Image Super-Resolution Using Deep Convolutional Networks“, Adresse: http://arxiv.org/abs/1501.00092.

[26] A. Tewari, O. Fried, J. Thies, V. Sitzmann, S. Lombardi, K. Sunkavalli, R. Martin-Brualla, T. Simon, J. Saragih, M. Nießner, R. Pandey, S. Fanello, G. Wetzstein, J.-Y. Zhu, C. Theobalt, M. Agrawala, E. Shechtman, D. B. Goldman und M. Zollhöfer. (8. Apr. 2020). „State of the Art on Neural Rendering“, Adresse: http://arxiv.org/abs/2004.03805.

[27] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi und R. Ng. (3. Aug. 2020). „NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis“, Adresse: http://arxiv.org/abs/2003.08934.

[28] M. Tancik, P. P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron und R. Ng. (18. Juni 2020). „Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains“, Adresse: http://arxiv.org/abs/2006.10739.

[29] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell und G. Wetzstein. (17. Juni 2020). „Implicit Neural Representations with Periodic Activation Functions“, Adresse: http://arxiv.org/abs/2006.09661.

[30] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. A. Hamprecht, Y. Bengio und A. Courville. (31. Mai 2019). „On the Spectral Bias of Neural Networks“, Adresse: http://arxiv.org/abs/1806.08734.

[31] R. Szeliski, „Image Processing“, in *Computer Vision: Algorithms and Applications*, Ser. Texts in Computer Science, R. Szeliski, Hrsg., London: Springer, 2011, ISBN: 978-1-84882-935-0. DOI: 10.1007/978-1-84882-935-0_3.

[32] L. Devroye, *Non-Uniform Random Variate Generation*. New York: Springer-Verlag, 1986, ISBN: 978-1-4613-8645-2. DOI: 10.1007/978-1-4613-8643-8.

[33] W. E. Lorensen und H. E. Cline, „Marching Cubes: A High Resolution 3D Surface Construction Algorithm“, *ACM SIGGRAPH Computer Graphics*, Jg. 21, Nr. 4, 1. Aug. 1987, ISSN: 0097-8930. DOI: 10.1145/37402.37422.

[34] K. Zhang, G. Riegler, N. Snavely und V. Koltun. (14. Okt. 2020). „NeRF++: Analyzing and Improving Neural Radiance Fields“, Adresse: http://arxiv.org/abs/2010.07492.

[35] Y. Du, Y. Zhang, H.-X. Yu, J. B. Tenenbaum und J. Wu. (17. Dez. 2020). „Neural Radiance Flow for 4D View Synthesis and Video Processing“. Version 1, Adresse: http://arxiv.org/abs/2012.09790.

[36] Z. Li, S. Niklaus, N. Snavely und O. Wang. (25. Nov. 2020). „Neural Scene Flow Fields for Space-Time View Synthesis of Dynamic Scenes“. Version 1, Adresse: http://arxiv.org/abs/2011.13084.

[37] W. Xian, J.-B. Huang, J. Kopf und C. Kim. (25. Nov. 2020). „Space-Time Neural Irradiance Fields for Free-Viewpoint Video“, Adresse: http://arxiv.org/abs/2011.12950.

[38] R. A. Newcombe, D. Fox und S. M. Seitz, „DynamicFusion: Reconstruction and Tracking of Non-Rigid Scenes in Real-Time“, in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA: IEEE, Juni 2015, ISBN: 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298631.

[39] S. Tan und M. L. Mayrovouniotis, „Reducing Data Dimensionality through Optimizing Neural Network Inputs“, *AIChE Journal*, Jg. 41, Nr. 6, 1995, ISSN: 1547-5905. DOI: 10.1002/aic.690410612.

[40] Z. Li, S. Niklaus, N. Snavely und O. Wang. (20. Apr. 2021). „Neural Scene Flow Fields for Space-Time View Synthesis of Dynamic Scenes“, Adresse: http://arxiv.org/abs/2011.13084.

[41] S. Vedula, S. Baker, P. Rander, R. Collins und T. Kanade, „Three-Dimensional Scene Flow“,

[42] K. Park, U. Sinha, J. T. Barron, S. Bouaziz, D. B. Goldman, S. M. Seitz und R. Martin-Brualla. (25. Nov. 2020). „Deformable Neural Radiance Fields“, Adresse: http://arxiv.org/abs/2011.12948.

[43] R. Martin-Brualla, N. Radwan, M. S. M. Sajjadi, J. T. Barron, A. Dosovitskiy und D. Duckworth. (13. Aug. 2020). „NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections“, Adresse: http://arxiv.org/abs/2008.02268.

[44] S. Bi, Z. Xu, P. Srinivasan, B. Mildenhall, K. Sunkavalli, M. Hašan, Y. Hold-Geoffroy, D. Kriegman und R. Ramamoorthi. (16. Aug. 2020). „Neural Reflectance Fields for Appearance Acquisition“, Adresse: http://arxiv.org/abs/2008.03824.

[45] P. Bojanowski, A. Joulin, D. Lopez-Paz und A. Szlam. (20. Mai 2019). „Optimizing the Latent Space of Generative Networks“, Adresse: http://arxiv.org/abs/1707.05776.

[46] Z. Wang, S. Wu, W. Xie, M. Chen und V. A. Prisacariu. (16. Feb. 2021). „NeRF–: Neural Radiance Fields Without Known Camera Parameters“, Adresse: http://arxiv.org/abs/2102.07064.

[47] Y. Shavit und R. Ferens. (16. Juli 2019). „Introduction to Camera Pose Estimation with Deep Learning“, Adresse: http://arxiv.org/abs/1907.05272.

[48] C.-H. Lin, W.-C. Ma, A. Torralba und S. Lucey. (13. Apr. 2021). „BARF: Bundle-Adjusting Neural Radiance Fields“, Adresse: http://arxiv.org/abs/2104.06405.

[49] (). „Google Colaboratory“, Adresse: https://colab.research.google.com.

[50] B. Räthlein und L. Masuch, *Ml-Tooling/Ml-Workspace*, Machine Learning Tooling, 22. Apr. 2021. Adresse: https://github.com/ml-tooling/ml-workspace.

[51] (). „Project Jupyter | Home“, Adresse: https://jupyter.org/.

[52] *noVNC*, noVNC, 22. Apr. 2021. Adresse: https://github.com/novnc/noVNC.

[53] *Tensorflow/Tensorboard*, tensorflow, 28. Apr. 2021. Adresse: https://github.com/tensorflow/tensorboard.

[54] L. Biewald, „Experiment Tracking with Weights and Biases“, 2020. Adresse: https://www.wandb.com/.

[55] C. Quei-An, „Nerf_pl: A Pytorch-Lightning Implementation of NeRF“, 2020. Adresse: https://github.com/kwea123/nerf_pl/.

[56] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne und Q. Zhang, *JAX: Composable Transformations of Python+NumPy Programs*, Version 0.2.5, 2018. Adresse: http://github.com/google/jax.

[57] B. Deng, J. T. Barron und P. P. Srinivasan, *JaxNeRF: An Efficient JAX Implementation of NeRF*, Version 0.0, 2020. Adresse: https://github.com/google-research/google-research/tree/master/jaxnerf.

[58] *Google/Mediapipe*, Google, 7. Mai 2021. Adresse: https://github.com/google/mediapipe.

[59] AI , *Kwea123/Nerf_Unity*, 24. März 2021. Adresse: https://github.com/kwea123/nerf_Unity.

[60] K. Schwarz, Y. Liao, M. Niemeyer und A. Geiger. (5. Juli 2020). „GRAF: Generative Radiance Fields for 3D-Aware Image Synthesis“, Adresse: http://arxiv.org/abs/2007.02442.