Hochschule Düsseldorf
University of Applied Sciences

Fachbereich Medien
Faculty of Media

PointNet

Funktionsweise und Anwendungsgebiete

Marco Winter
Matrikel-Nr.: 740416

Wissenschaftliche Vertiefung im Studiengang

Bachelor Medieninformatik
16. Juni 2020

Betreuer:
Prof. Dr. rer. nat. Christian Geiger
M. Sc. Marcel Tiator

Zusammenfassung

Diese Arbeit beschäftigt sich mit PointNet, einem von Qi et al. [1] entworfenen neuronalen Netzwerk zur Analyse von 3D-Punktwolken. Nach einer kurzen Einführung in die zum Verständnis nötigen Grundlagen werden der Aufbau und die Funktionsweise des Netzwerks schrittweise erläutert. Ebenso wird die praktische Umsetzung von PointNet anhand einer Beispiel-Implementation mithilfe der Deep Learning API Keras besprochen. Hierfür werden mehrere interaktive Jupyter Notebooks zur Verfügung gestellt. Anschließend werden drei Projekte vorgestellt, in denen PointNet zum Einsatz kommt. Zum Schluss wird ein Ausblick auf PointNet++ gegeben, eine Erweiterung der PointNet-Architektur [2].

Mithilfe einer detaillierten Analyse soll diese Arbeit einen Einblick in den Themenbereich der Objekterkennung und -Segmentierung im dreidimensionalen Raum geben und dabei helfen, ein tiefergehendes Verständnis für die behandelte Netzarchitektur PointNet zu erlangen.

Kapitel 1 beschreibt die Grundlagen neuronaler Netze und soll einen thematischen Einstieg für das Verständnis dieser Arbeit bieten.

Kapitel 2 erläutert den Aufbau und die theoretische Funktionsweise der von Qi et al. [1] entworfenen PointNet-Architektur.

Kapitel 3 analysiert eine Beispiel-Implementation der PointNet-Architektur. Hierfür wird ein interaktives Jupyter Notebook bereitgestellt.

Kapitel 4 zeigt Projekte, in denen PointNet erfolgreich angewendet wurde.

Kapitel 5 bietet einen kurzen Ausblick auf eine Weiterentwicklung der PointNet-Architektur und beschreibt die Unterschiede zwischen PointNet++ und PointNet.

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abb.	–	Abbildung
API	–	application programming interface (Anwendungsschnittstelle)
bspw.	–	beispielsweise
CNN	–	convolutional neural network (faltendes neuronales Netzwerk)
FC	–	fully connected (vollständig verknüpft)
i.d.R.	–	in der Regel
K.	–	Kapitel
KNN	–	künstliches neuronales Netz
MLP	–	multi layer perceptron (mehrschichtiges Perzeptron)
MNIST DB	–	Modified National Institute of Standards and Technology database
S.	–	Seite
STN	–	spatial transformer network (räumliches Transformations-Netz)
z. B.	–	zum Beispiel

Abbildungsverzeichnis

1. Einleitung: Grundlagen neuronaler Netzwerke

Künstliche neuronale Netze (KNN) sind Berechnungsmodelle, deren Struktur stark vom Aufbau und der Funktionsweise biologischer Nervennetzwerke im Gehirn inspiriert ist. Sie bestehen aus einer Vielzahl untereinander verknüpfter Recheneinheiten, welche als künstliche Neuronen bezeichnet werden und gemeinsam ein hochkomplexes Kommunikationsnetzwerk bilden [3, S. 268].

Auf Basis vorgegebener Beispieldaten sind diese Systeme ohne spezifische Programmierung in der Lage, die Ausführung komplexer Aufgaben zu erlernen. Hierbei sind verschiedenste Problemstellungen denkbar. Von der Mustererkennung in beliebigen Datenmengen über die Realisierung autonomer Fahrzeuge [4] bis hin zur Erzeugung von virtuellen Gemälden [5] - künstliche neurale Netze werden in verschiedensten Bereichen eingesetzt.

Eine Anwendung für neuronale Netze, welche im Rahmen dieser Arbeit besondere Aufmerksamkeit erfährt, ist die Klassifizierung, Segmentierung und semantische Analyse dreidimensionaler Objekte. Hierbei lernt ein Netzwerk, anhand eines entsprechenden Datensatzes – bspw. das Signal eines Laserscanners in Form einer Punktwolke – Objekte und deren einzelne Bestandteile in einem gewissen Umfeld zu identifizieren.

In diesem Kapitel werden grundlegende Konzepte und Begriffe neuronaler Netze erläutert, welche für das Verständnis des Hauptteils dieser Arbeit vonnöten sind.

1.1. Topologie neuronaler Netzwerke

Ein KNN besteht in seinem Kern aus einem Netzwerk miteinander verbundener künstliche Neuronen. Jedes dieser Neuronen verfügt in der Regel über ein oder mehrere gewichtete *Eingangssignale* (input), eine *Aktivierungsfunktion* (activation function) und eine *Ausgabe* (output). Der oder die Input-Werte werden mit ihren jeweiligen Gewichten multipliziert und anschließend als Summe an die Aktivierungsfunktion übergeben. Diese wiederum berechnet einen einzelnen Output-Wert für etwaige nachfolgende Neuronen oder die finale Ausgabe des Netzwerks [6, K. 1].

Die Neuronen eines KNN sind häufig in miteinander verknüpften Schichten angeordnet wie in Abbildung 1.1 dargestellt. Mathematisch betrachtet stellen diese Schichten disjunkte Teilmengen dar, deren Neuronen über gewichtete Kanten miteinander in Beziehung stehen [3, S. 269]. Die erste und letzte Schicht eines Netzwerks werden hierbei als Eingabe- (input layer) beziehungsweise Ausgabeschicht (output layer) bezeichnet. Die Neuronen dieser Schichten verfügen über besondere Eigenschaften:

Eingangsneuronen (input neurons) besitzen keinen gewichteten Eingang und keine Aktivierungsfunktion. Sie nehmen als Input-Schnittstelle die zu verarbeitenden Daten auf und geben diese über gewichtete Kanten an die zweite Schicht weiter.

Ausgangsneuronen (output neurons) hingegen besitzen ausschließlich eingehende Kanten sowie ggf. eine Aktivierungsfunktion und stellen die Output-Schnittstelle dar.

Die Anzahl der Neuronen in diesen Schichten hängt von der Dimensionalität der zu verarbeitenden Daten bzw. des gewünschten Outputs ab [3, S. 269]. Ein Netzwerk, das z. B. den Verkaufswert eines Hauses anhand der Parameter *Grundfläche* und *Anzahl der Zimmer* ermittelt, verfügt über zwei Eingangs- und ein Ausgangsneuron.

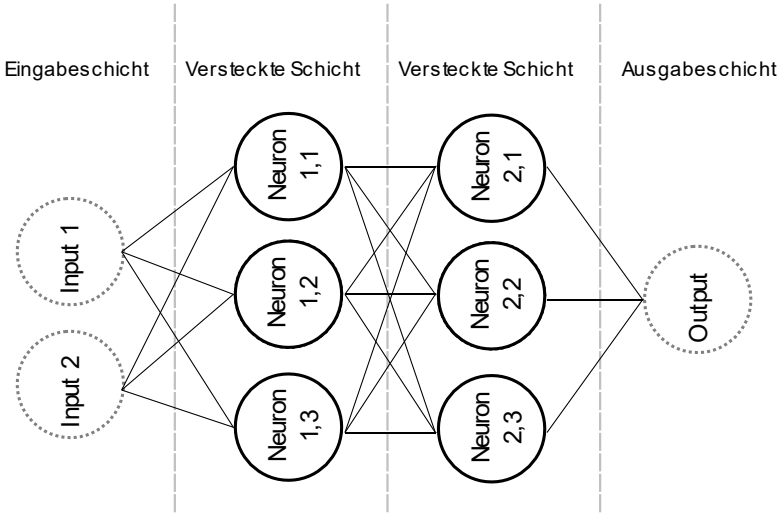


Abbildung 1.1: Aufbau eines vollständig verbundenen KNN

Zwischen der Eingangs- und Ausgangsschicht eines KNN liegen die sogenannten versteckten Schichten (hidden layer). Die Neuronen dieser Schichten verarbeiten die Daten aus der Eingangsschicht oder einer vorherigen versteckten Schicht und übergeben das Ergebnis an die Ausgabeschicht oder eine nachfolgende versteckte Schicht. Sie helfen dem Netzwerk, die eingehenden Daten zu verstehen und erzeugen Schicht für Schicht die Ausgabe [6, K. 1].

Wie die Verknüpfung zwischen den Schichten eines neuronalen Netzes im Detail realisiert wird, ist ebenso wie die Anzahl der versteckten Schichten und deren Neuronen gänzlich von der gewählten Architektur und dem Zweck des KNN abhängig. Prinzipiell kann jedoch zwischen vorwärts gerichteten (feed forward neural network) und rückgekoppelten Netzwerken (recurrent / feedback neural network) unterschieden werden.

Bei Letzterem werden Verbindungen zwischen den Ausgängen einzelner Neuronen wahlweise mit deren eigenen Eingängen (direct feedback), den Eingängen von Neuronen in derselben Schicht (lateral feedback) oder den Eingängen von Neuronen in einer vorangegangenen Schicht (indirect feedback) verknüpft. Durch die Rückkopplung wird eine Art interner „Speicher“ in die Netzwerkstruktur eingebaut, welcher unter anderem die Verarbeitung von Zeitreihendaten [6, K. 13] oder Differentialgleichungen ermöglicht [7, S. 141].

1.2. Aktivierungsfunktionen

Wie im Abschnitt 1.1 beschrieben, besitzen die Neuronen eines KNN (mit Ausnahme der Eingabeneuronen) eine Aktivierungsfunktion. Diese Funktion ermöglicht es dem Neuron, aus der Summe seiner gewichteten Eingänge einen Ausgangswert zu berechnen.

Aktivierungsfunktionen können viele verschiedene Formen haben. Die Auswahl der korrekten Funktion stellt einen wichtigen Schritt beim Entwurf einer Netzarchitektur dar und hat einen starken Einfluss auf das benötigte Format der Eingabedaten sowie auf die Performanz und Genauigkeit des Netzwerks [6, K. 1].

In diesem Abschnitt werden zwei Aktivierungsfunktionen vorgestellt, welche für das Verständnis der Erläuterungen in Kapitel 2 und 3 vonnöten sind. Weitere häufig vorkommende Aktivierungsfunktionen können in [6] und [7] nachgelesen werden.

1.2.1. Rectified Linear Unit

Eine Rectified Linear Unit, kurz ReLU genannt, ist ein Neuron, dessen Aktivierungsfunktion ein Gleichrichter (rectifier) ist. Die Funktionsgleichung eines Gleichrichters wird wie folgt beschrieben:

$$\boldsymbol{\Phi(z) = max(0, z)} \tag{1.1}$$

Eine ReLU setzt also sämtlichen negativen Input auf null und gibt positiven Input unverändert weiter. Der zugehörige Funktionsgraph wird in Abbildung 1.2 gezeigt. Obwohl es sich um eine simple Funktion handelt, liefert die ReLU aufgrund ihrer mathematischen Eigenschaften bessere Ergebnisse als andere Aktivierungsfunktionen und wird daher häufig als beste Alternative für die versteckten Schichten eines KNN empfohlen [6, K. 1].

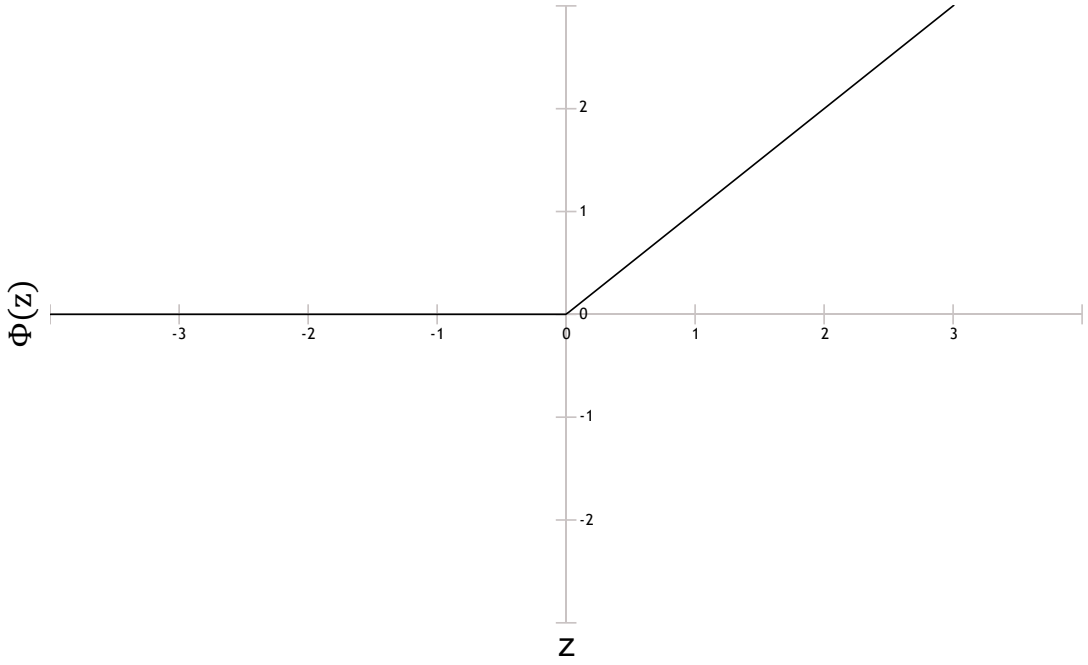


Abbildung 1.2: Graph der Gleichrichterfunktion $\Phi(z) = max(0, z)$

1.2.2. Softmax-Funktion

Bei der Softmax-Funktion handelt es sich um eine Aktivierungsfunktion, die vornehmlich in der Ausgabeschicht von klassifizierenden KNN eingesetzt wird [6, K. 1]. Hierbei wird jedem Neuron der Ausgabeschicht ein Wert zwischen null und eins zugewiesen. Dieser stellt die prozentuale Wahrscheinlichkeit dar, mit welcher ein Eingabewert der zum Ausgabeneuron gehörigen Klasse zugeordnet werden kann. Die Funktionsvorschrift der Softmax-Funktion wird wie folgt beschrieben:

$$\boldsymbol{\Phi_i} = \frac{\boldsymbol{e^{z_i}}}{\sum_{\boldsymbol{j \in group}} \boldsymbol{e^{z_j}}} \tag{1.2}$$

Hierbei bezeichnet z_i den Eingabewert eines Neurons, während die Summe die Werte aller *j* Eingabewerte für alle Neuronen der Ausgabeschicht aggregiert. Es handelt sich hierbei um eine normalisierte Exponentialfunktion und kann als Generalisierung der logistischen Sigmoid-Funktion betrachtet werden [8, S. 198]. Die Besonderheit der Softmax-Funktion ist die Tatsache, dass der Wert eines Neurons in Abhängigkeit zur Eingabe der benachbarten Neuronen gebildet wird. Dies führt dazu, dass die Summe aller Ausgabewerte einer Schicht nach Verwendung der Softmax-Funktion gleich eins ist. Als Aktivierungsfunktion der Ausgabeschicht eines KNN wird mithilfe der Softmax-Funktion diejenige Klasse gewählt, deren Neuron den höchsten Ausgabewert vorweist [6, K. 1].

1.3. Training eines KNN

Damit ein KNN korrekte Vorhersagen treffen, Daten klassifizieren oder anderweitige Probleme lösen kann, muss es trainiert werden. Beim Training eines KNN werden anhand von Beispieldatensätzen mittels eines Algorithmus die Verbindungsgewichte ggf. weitere Parameter schrittweise angepasst, bis ein bestimmtes Kriterium optimiert ist [7, S. 40]. Die Trainingsstrategie und der Algorithmus hängen hierbei von der Art der Testdaten und der Aufgabe des Netzwerks ab.

Im folgenden Abschnitt wird die Trainingsstrategie des überwachten Lernens (supervised learning) erläutert. Weitere Methoden und Algorithmen, welche im Rahmen dieser Arbeit von geringerer Bedeutung sind, können in der Literatur von Bishop [8], Bibel et al. [7] oder Heaton [6] zur weiteren Lektüre nachgeschlagen werden.

1.3.1. Supervised Learning

Das Ziel des *supervised learning* ist die Modellierung einer bedingten Wahrscheinlichkeitsverteilung $p(t \mid x)$, mit deren Hilfe ein Ergebnis *t* für jeden möglichen Eingabewert *x* möglichst vorhergesagt werden kann. Hierfür werden Trainingsdaten in Form von Wertepaaren (t, x) benötigt, wobei t der korrekte Ausgangswert ist [8, pp. 137–138].

Um das Konzept an einem Beispiel zu verdeutlichen: Gegeben sei ein KNN, das E-Mails als *Spam* oder *non-Spam* klassifizieren soll. Um dieses Netzwerk mithilfe von überwachtem Training auf seine Aufgabe vorzubereiten, lässt man es wiederholt Datensätze aus Beispiel-Mails verarbeiten. Diese sind bereits als *Spam* oder *non-Spam* markiert. Das Ergebnis ist also bei einem Trainingsdatensatz für alle darin enthaltenen Beispiele bekannt.

Zu Beginn des Trainings sind die vom KNN vorhergesagten Werte aufgrund zufälliger Initialisierung der Gewichtungen oftmals falsch. Durch die Verwendung eines passenden Algorithmus wird die Differenz zwischen Vorhersage und vorgegebener Wahrheit (dargestellt durch eine Verlustfunktion) schrittweise reduziert. Auf diese Weise wird eine Annäherung der oben genannten bedingten Wahrscheinlichkeitsverteilung ermittelt, mit der Vorhersagen für neue Werte x getroffen werden können.

1.4. Deep Learning

Deep Learning ist ein untergeordneter Themenbereich des maschinellen Lernens, welcher vor allem für moderne KNN-Architekturen von großer Bedeutung ist. Deep Learning stellt eine neue Art des Lernens dar, deren Schwerpunkt in der Kombination aufeinanderfolgender Schichten zunehmend aussagekräftiger Darstellungen liegt [9, S. 8].

Verständlicher ausgedrückt handelt es sich bei einem „tiefen“ KNN um ein Netzwerk mit zwei oder mehr versteckten Schichten. In diesen Schichten werden die Eingabedaten sukzessive abstrahiert und in eine Darstellung überführt, mit welcher sich die Aufgabe des Netzwerkes leichter lösen lässt. Mithilfe von Abbildung 1.3 lässt sich diese zunehmende Abstraktion am Beispiel eines KNN zur Erkennung handschriftlicher Zahlen optisch gut erkennen. Die Ausgaben der einzelnen Schichten werden zunehmend einfacher, bis schlussendlich anhand von 3x3-Gittern die Klassifizierung durchgeführt werden kann.

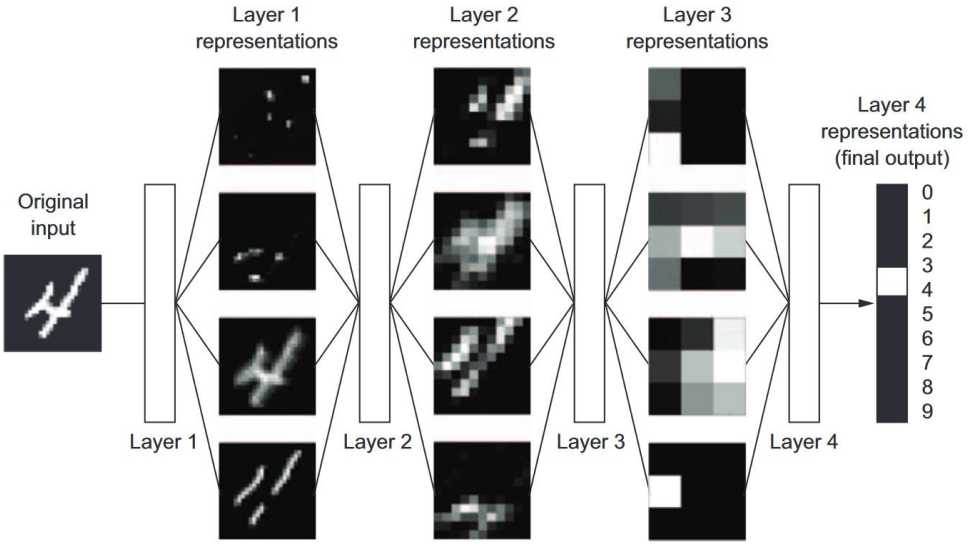


Abbildung 1.3: Beispiel eines "tiefen" neuronalen Netzwerks [9, S. 9].

Das Ziel eines „tiefen“ KNN ist also die (deterministische) Umwandlung der Eingabedaten in eine Form, die für die eigentliche Aufgabe des Netzwerks (Klassifizierung, Segmentierung etc.) am besten geeignet ist. Oft wird dieser Schritt auch Feature Building genannt, da das Netzwerk die zur Verarbeitung nötigen Eigenschaften für einen bestimmten Input selbst erzeugt.

Im folgenden Abschnitt wird eine verbreitete Architektur für „tiefe“ neuronale Netze vorgestellt, um deren Eigenschaften näher zu beleuchten und zeitgleich wichtige Grundlagen für den Hauptteil dieser Arbeit zu vermitteln.

1.4.1. Convolutional Neural Network

Convolutional Neural Networks (faltendes neuronales Netzwerk) sind eine spezialisierte Art von neuronalen Netzen zur Verarbeitung von Daten, die eine gitterartige Topologie besitzen. Beispiele sind mittels Sampling erzeugte Zeitreihendaten, die als 1-D-Gitter betrachtet werden können, sowie Bilddaten, die man sich als ein 2-D-Pixelgitter vorstellen kann. CNNs waren in den vergangenen Jahren enorm erfolgreich in der praktischen Anwendung. Der Name "Convolutional Neural Network" bedeutet, dass das Netzwerk eine mathematische Operation namens Faltung verwendet [10, S. 326].

CNNs sind Feed-Forward Netzwerke, besitzen gegenüber gewöhnlichen Netzwerken dieser Art jedoch mehrere Vorteile:

- Die von ihnen gelernten Muster sind invariant gegenüber Translationen [9, S. 123].
- CNNs können durch aufeinanderfolgende Faltungen räumliche Hierarchien lernen und somit immer komplexere Abstraktion durchführen [9, S. 123].
- Durch die Eigenschaften der Faltung (nur lokale Verbindungen, geteilte Gewichtungsparameter) sowie durch das Max-Pooling ist die Anzahl der zu trainierenden Parameter z.T. mehrere Größenordnungen kleiner als bei gewöhnlichen, voll verbundenen Netzwerken.

Nachfolgend werden die typischen Bestandteile eines CNN kurz vorgestellt. Detailliertere Erläuterungen und Beispiele können in [10, K. 9], [9, K. 5] sowie [6, K. 10] nachgeschlagen werden.

Faltungsschicht (Convolutional Layer):

Der Hauptzweck dieser Schicht ist es, die Eingabedaten zu filtern und so lokale Merkmale zu erschließen, anhand derer die Aufgabe des Netzwerks erfüllt werden kann. Diese Merkmale können bspw. Kanten, Linien oder andere visuelle Eigenschaften eines Bildes sein. Die Anzahl der verwendeten Filter (auch Kernel genannt) korreliert hierbei direkt mit der Menge an Merkmalen [6, K. 10].

Für den Fall eines Bildes, dessen Inhalt klassifiziert werden soll, kann man sich die Filter der Faltungsebene als Pixel-Gitter vorstellen. Jeder „Pixel“ des Filters verfügt über einen trainierbaren Gewichtungsfaktor. Im Rahmen der Faltung wird das Bild schrittweise vom Filter-Gitter durchlaufen. Für jeden Schritt wird die Summe aller Produkte aus übereinanderliegenden Bildpixeln und Filterpixeln bzw. Gewichtungswerten gebildet und ausgegeben. Abbildung 1.4 stellt diesen Prozess schematisch dar. Der 2x2 Kernel durchläuft die Eingabe für alle gültigen Positionen und gibt die entsprechenden Produktsummen aus.

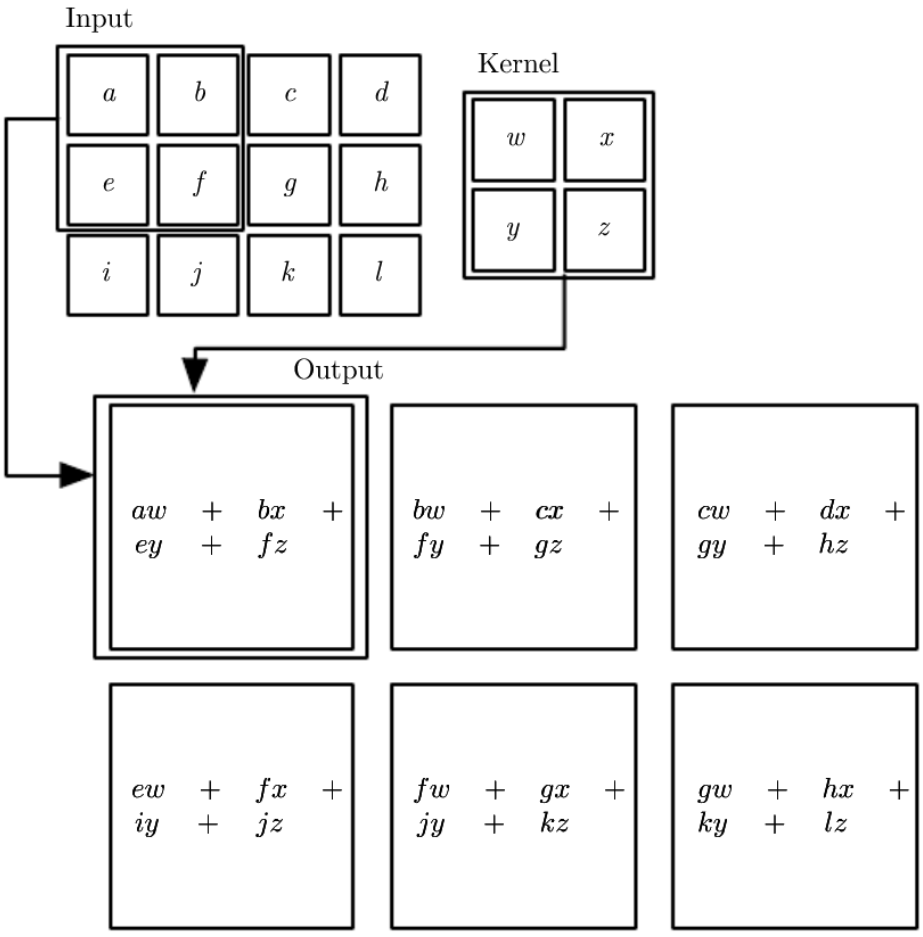


Abbildung 1.4: Schema der Faltungsoperation [10, S. 330].

Das Ergebnis der Faltungsoperation wird als Eingangssignal an die Neuronen der Faltungsschicht übergeben und mittels einer Aktivierungsfunktion, welche i.d.R. ReLU ist, verarbeitet.

Max-Pooling:

Beim Max-Pooling werden die Ausgangswerte der Faltungsschicht aggregiert. Hierbei durchläuft in Abhängigkeit mehrerer Parameter (Größe, Schrittgröße) ein Filter die Daten und übernimmt in jedem Schritt lediglich den maximalen Eingangswert in seinem Einflussbereich. Dieser Prozess wird in Abbildung 1.5 anhand eines Beispiels visualisiert. Hierbei wird das Gitter von einem 2x2-Filter mit der Schrittgröße 2 durchlaufen. Die farbig hinterlegten Quadrate stellen die einzelnen Filterschritte dar.

Zu den Zielen des Max-Pooling zählt die Reduktion der Gewichtungsparameter zur Steigerung der Trainingsgeschwindigkeit [9, S. 128] sowie die Möglichkeit, nicht-uniforme Eingabedaten wie bspw. Bilder in verschiedenen Auflösungen mit einem einzigen Netzwerk zu verarbeiten [10, S. 339].

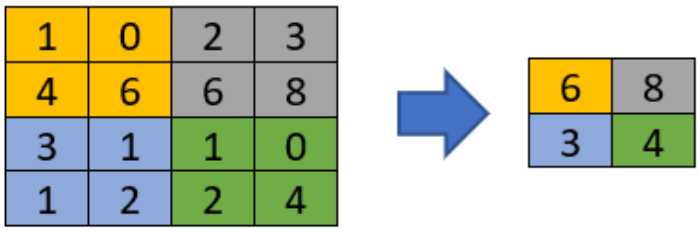


Abbildung 1.5: Beispiel für Max-Pooling

Fully Connected Layer:

Nachdem mithilfe von Faltungen und Pooling eine Darstellung der Daten erreicht ist, welche für die eigentliche Aufgabe des CNN optimal ist, folgt abschließend eine vollständig verknüpfte Schicht (Fully-connected Layer), auch FC Layer oder Dense Layer genannt. Jedes Neuron der FC Layer ist mit jedem Neuron der vorangegangenen Schicht verbunden und hat somit die typische Struktur, welche bereits aus Feed-Forward-Netzwerken bekannt ist. Als Aktivierungsfunktionen werden hier typischerweise die ReLU oder im Falle der Ausgangsschicht die Softmax-Funktion verwendet.

2. PointNet – Aufbau und Workflow

Bei PointNet handelt es sich um eine KNN-Architektur, die 2016 von Qi et al. [1] entworfen wurde. Das Ziel bzw. die Aufgaben von PointNet sind die Klassifizierung, Segmentierung und die semantische Analyse von dreidimensionalen Objekten und Szenen.

PointNet verfolgt im Gegensatz zu vergleichbaren Netztypen (siehe Kapitel 5) eine neuartige Strategie: Objekte und Szenen werden in Form von Punktwolken direkt vom Netzwerk analysiert. Diese direkte Verarbeitung hat den Vorteil, dass die Daten nicht gesondert aufbereitet und in bestimmte Formate konvertiert werden müssen. Sie ist hocheffizient und im empirischen Vergleich dem bisherigen Stand der Technik zum Teil überlegen [1, S. 1].

Allerdings birgt die Verarbeitung von Punktwolken auch einen entscheidenden Nachteil. Mathematisch betrachtet handelt es sich bei Punktwolken um ungeordnete Mengen. Die Reihenfolge, in welcher die Elemente einer Punktwolke an PointNet übergeben werden, ist also nicht fest, sondern variabel. Basierend auf dieser Eigenschaft muss das Netzwerk eine wichtige Voraussetzung erfüllen: Die Ausgabe für eine beliebige Punktwolke muss invariant gegenüber allen möglichen Permutationen der Punktwolken-Elemente sein [1, S. 1f.].

In den folgenden Abschnitten wird die in Abbildung 2.1 dargestellte PointNet-Architektur in separate Schritte aufgeteilt und in seiner Funktionsweise erläutert. Hierbei werden die Workflows zur Klassifizierung, Segmentierung und der semantischen Analyse von Szenen dargestellt.

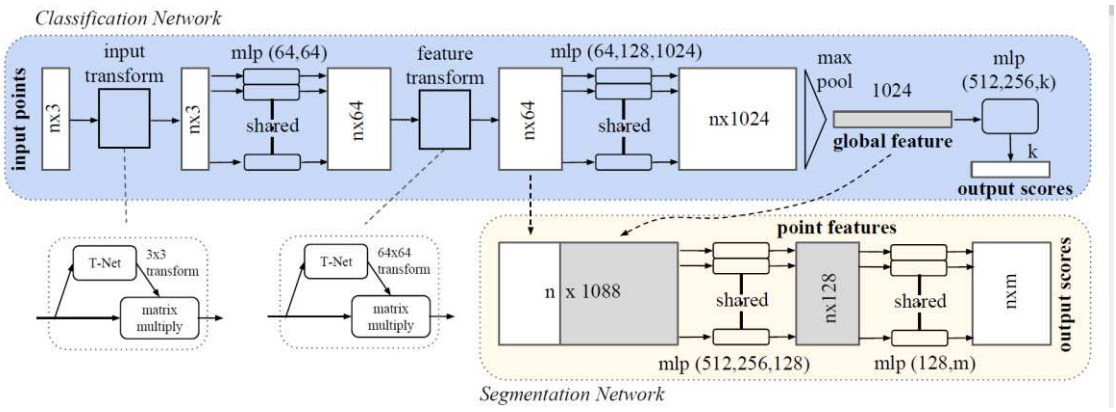


Abbildung 2.1: Aufbau der PointNet-Architektur [1, S. 3]

2.1. Workflow der PointNet-Architektur: Klassifizierung

Die PointNet-Architektur lässt sich grundlegend in zwei Bereiche unterteilen: Das Klassifizierungs- und das Segmentierungsnetzwerk. Im Klassifizierungsnetzwerk werden auf Basis der Eingabedaten für jedes Element der Punktwolke Eigenschaften (Features) ermittelt, welche lokale Informationen der Punktwolke enthalten. Dies können Kanten, Punktcluster oder Ähnliches sein. Anschließend wird mithilfe des Max-Pooling eine gegenüber dem Input invariante globale Signatur aggregiert, welche über mehrere vollständig verknüpfte Schichten (FC Layer) einer Objektklasse zugeordnet werden kann. Für die versteckten Schichten wird i.d.R. ReLU als Aktivierungsfunktion verwendet.

Damit die Werte in jeder Schicht in derselben Größenordnung liegen, findet zudem nach jeder Schicht eine Normalisierung statt. Bei der hierfür verwendeten Technik *batch normalization* [11] wird der Output einer Schicht für jedes Neuron vom Durchschnittswert des aktuell verarbeiteten Datenstapels (batch) abgezogen und durch dessen Varianz dividiert.

Für die Ausgabeschichten zur Klassifizierung und Segmentierung findet die Softmax-Funktion Anwendung [1, S. 3], um prozentuale Wahrscheinlichkeitswerte für die verfügbaren Objekt- bzw. Segmentklassen zu ermitteln.

Wie an Abbildung 2.2 zu erkennen ist, besteht das in der Abbildung markierte Klassifizierungsnetzwerk aus insgesamt sechs Teilschritten, welche nachfolgend erläutert werden.

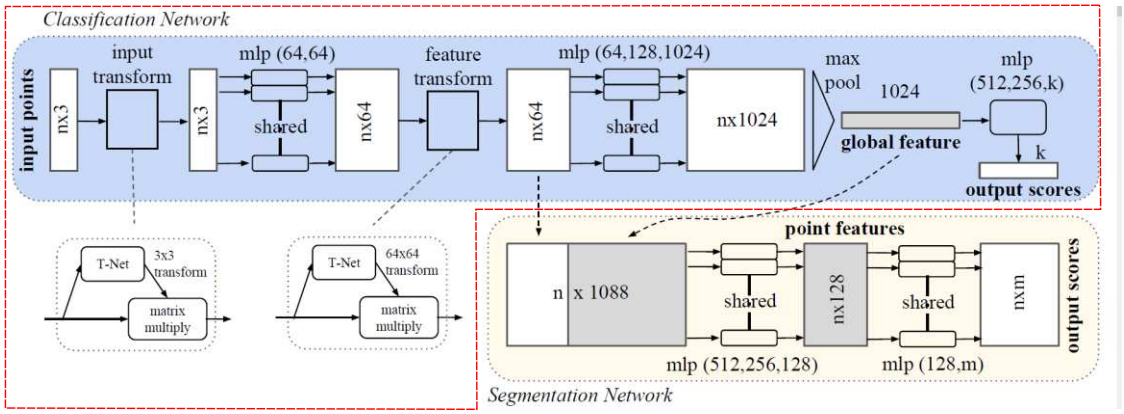


Abbildung 2.2: Klassifizierungsnetzwerk der PointNet-Architektur [1, S. 3]

2.1.1. Input-Transformation

Wie bereits einleitend erwähnt ist die Invarianz gegenüber Permutationen des Inputs eine Grundvoraussetzung für die Funktion der PointNet-Architektur. Für die korrekte semantische Beschriftung einer Punktwolke wird zusätzlich eine weitere Form der Invarianz benötigt: Die Ausgabe darf nicht von geometrischen Transformationen wie Rotationen und Translationen beeinflusst werden. PointNet löst diese Problematik mithilfe eines „Mini-Netzwerks“, welches die Bezeichnung T-Net trägt. Das T-Net folgt in seinem Aufbau der PointNet-Architektur und hat das Ziel, eine affine Transformationsmatrix zu bestimmen. Diese Matrix wird auf die Elemente der Punktwolke angewendet und führt so zu einer Normalisierung des Inputs. Die Input-Transformation, welche in Abbildung 2.3 markiert ist, stellt somit den ersten Schritt im PointNet-Workflow dar.

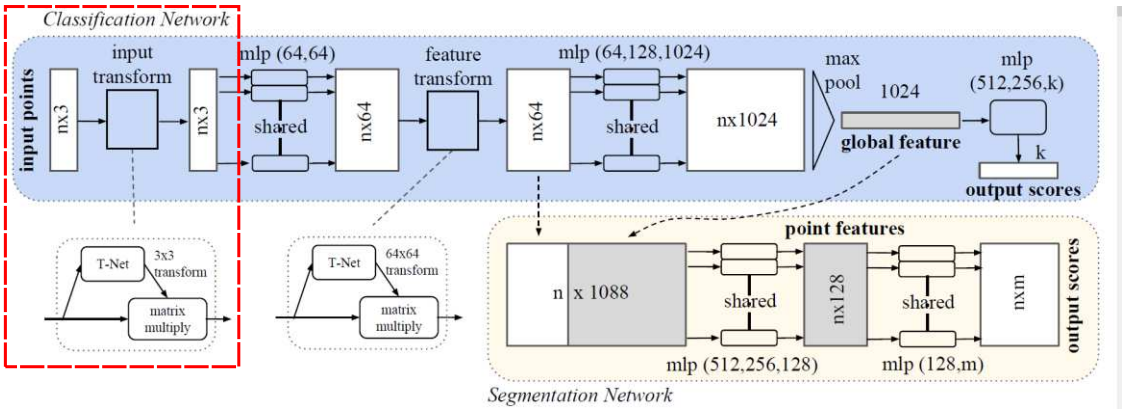


Abbildung 2.3: Erster Schritt - Input-Transformation mittels T-Net [1, S. 3]

Die Idee hinter dem T-Net basiert auf dem von Jaderberg et al. [12] vorgestellten Spatial Transformer Network (STN). Vereinfacht ausgedrückt ist es das Ziel eines STN, etwaige Verschiebungen oder Verzerrungen eines Inputs gegenüber eines definierten Basiszustands zu kompensieren, bevor eine weitere Verarbeitung stattfindet. Abbildung 2.4 zeigt das in [12, S. 2] vorgestellte Beispiel für die Funktionsweise eines STN. Hier wurden Bilder von Zahlen der MNIST Datenbank [13] zufälligen Transformationen unterzogen (a) und anschließend von einem STN verarbeitet. Die Darstellung zeigt, wie das Netzwerk zunächst die relevanten Regionen eines Bildes herausfiltert (b) und diese anschließend in einer normalisierten Pose ausgibt (c). Diese Ausgabe wird dann für die Klassifizierung genutzt (d).

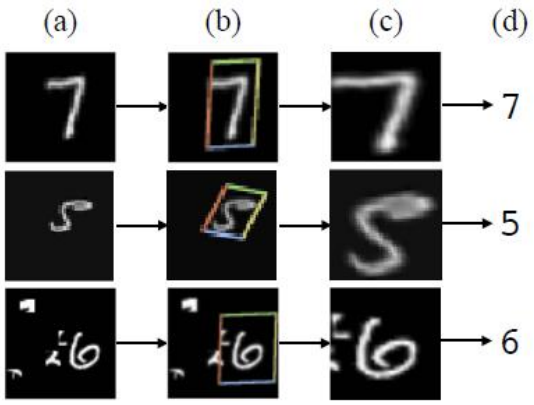


Abbildung 2.4: Beispiel für die Funktion eines STN [12, S. 2]

Auf PointNet übertragen bedeutet dies also, dass im Schritt der Input-Transformation die einzelnen Elemente der Punktwolke gewissermaßen in Position gerückt werden. Da die Punkte in Form von Vektoren an das Netzwerk übergeben werden und für die Bildung der Features voneinander unabhängig auf höherdimensionale Räume abgebildet werden (mehr dazu in Abschnitt 2.2.2), kann dies durch eine einfache Matrixmultiplikation erreicht werden [1, S. 4].

Wie in der nachfolgenden Grafik (Abb. 2.5) zu sehen ist, nutzt das T-Net einige Kernelemente der übergeordneten Architektur. Mithilfe eines mehrschichtigen Perzeptrons (multi layer perceptron), kurz MLP genannt, werden Features gebildet, welche in einer Pooling-Schicht zu einer Signatur aggregiert werden. Anhand dieser Signatur wird durch mehrere FC Layer die 3x3-Transformationsmatrix gebildet.

Da der Aufbau des T-Net im Wesentlichen dem des übergeordneten PointNet gleicht, werden die Details der einzelnen Bestandteile im nachfolgenden Abschnitt 2.2.2 näher erläutert.

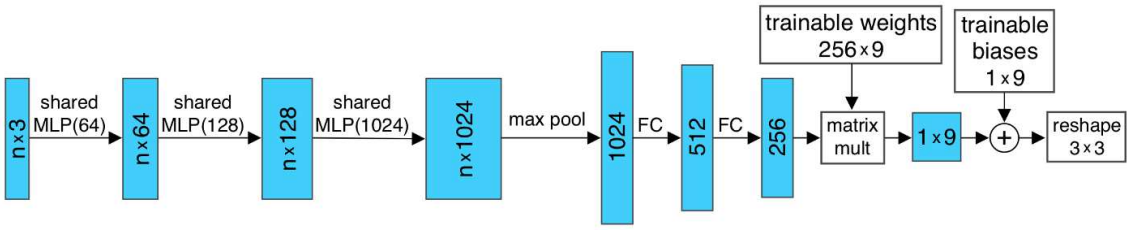


Abbildung 2.5: Aufbau T-Net [14]

2.1.2. Feature Building

Im zweiten Schritt des Klassifizierungsnetzwerks, wie in Abbildung 2.6 dargestellt, werden die normalisierten Koordinaten über einen MLP mit zwei versteckten Schichten und geteilten Gewichtungsparametern individuell auf einen höherdimensionalen Raum abgebildet. Umgangssprachlich ausgedrückt heißt dies, dass für jeden Punkt auf Basis seiner drei Raumkoordinaten insgesamt 64 Werte auf Basis der Gewichte berechnet werden.

Hierbei sind zwei Dinge von elementarer Bedeutung:

1. Die trainierbaren Gewichte der MLP-Schichten werden von allen Punkten geteilt. Jede Änderung der Gewichte im Trainingsprozess hat unmittelbare Folgen für alle Elemente der Punktwolke. Das Netzwerk muss also eine für alle Punkte passende Kombination von Gewichtungsparametern finden.
2. Alle Punkte werden individuell abgebildet. Abgesehen von den geteilten Gewichten gibt es keine Relation zwischen benachbarten Punkten. Dies bedeutet, dass die Bestimmung der Features invariant gegenüber der Reihenfolge der Punkte im Input ist.

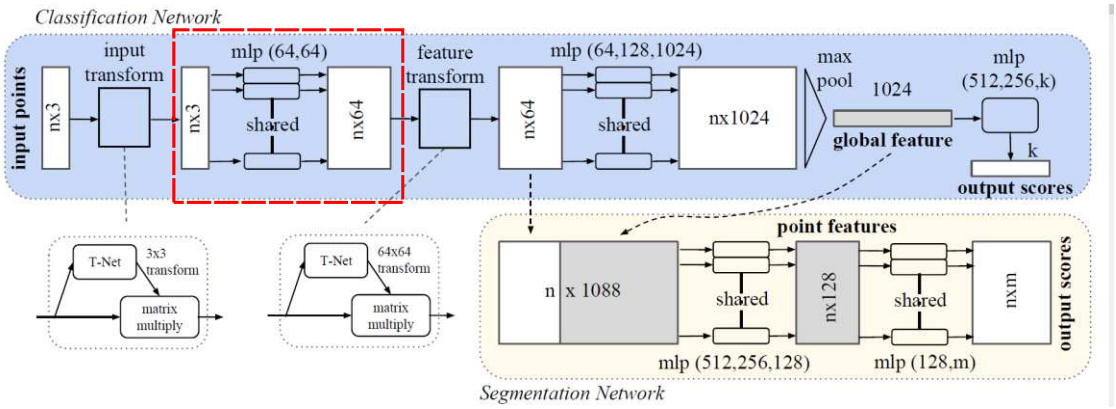


Abbildung 2.6: Zweiter Schritt - Feature Building

Die vom MLP berechneten 64 Werte sind für jeden Punkt einerseits unterschiedlich. Doch durch die Verwendung geteilter Gewichte und der daraus folgenden Verknüpfung aller Punkte enthalten diese Werte lokale Informationen über die Punktwolke. Räumlich nah beieinander liegende Punkte besitzen beispielsweise mathematisch ähnliche Feature-Vektoren.

Allgemein entspricht dieser Schritt der PointNet-Architektur der in Abschnitt 1.4 beschriebenen Abstraktion und Umwandlung des Inputs in eine für die weitere Verarbeitung geeignete Form.

2.1.3. Feature Transform

Die Feature-Transformation im PointNet wird in der folgenden Grafik (Abb. 2.7) dargestellt und entspricht im Grunde dem unter 2.1.1 beschriebenen ersten Schritt des Netzwerks. Auch hier ist die individuelle Ausrichtung bzw. Normalisierung der Feature-Vektoren mithilfe einer Matrixmultiplikation das Ziel. Da die Dimensionen der Feature-Transformationsmatrix (64x64) um ein Vielfaches höher sind als bei der Input-Transformation (3x3), gestaltet sich die Optimierung wesentlich schwieriger. Um den Prozess zu stabilisieren und bessere Ergebnisse zu ermöglichen, wurde folgender Term zur Regularisierung der Verlustfunktion hinzugefügt [1, S. 4]:

$$\boldsymbol{L_{reg}} = ||\boldsymbol{I} - \boldsymbol{AA^T}||^2 \tag{2.1}$$

Hierbei entspricht $\boldsymbol{I}$ der Einheitsmatrix und $\boldsymbol{A}$ der vom T-Net bestimmten Transformationsmatrix.

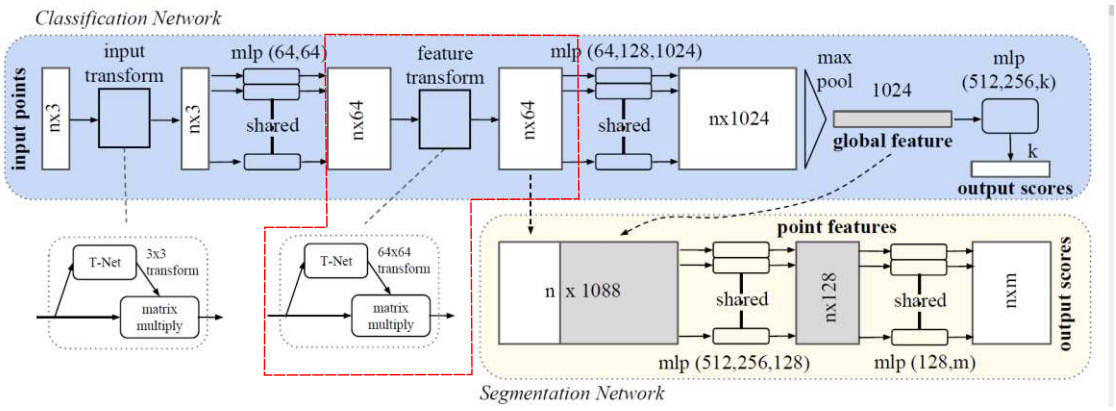


Abbildung 2.7: Dritter Schritt - Feature Transformation

2.1.4. Zwischenspeicher und erneutes Feature Building

Bevor der Input weiterverarbeitet wird, findet nach der Feature-Transformation eine Zwischenspeicherung der Daten statt. Dieser Schritt kann als Schnittstelle verstanden werden. Er ist streng genommen Teil des Segmentierungs-Netzwerks und wird in Abschnitt 2.2 erläutert. Im Anschluss an die Zwischenspeicherung werden die Feature-Vektoren der einzelnen Punkte erneut durch einen MLP mit geteilten Gewichten auf einen höherdimensionalen Raum abgebildet. Dieser MLP wird in der untenstehenden Grafik (Abb. 2.8) dargestellt und verfügt über drei Schichten mit 64, 128 und schließlich 1024 Elementen.

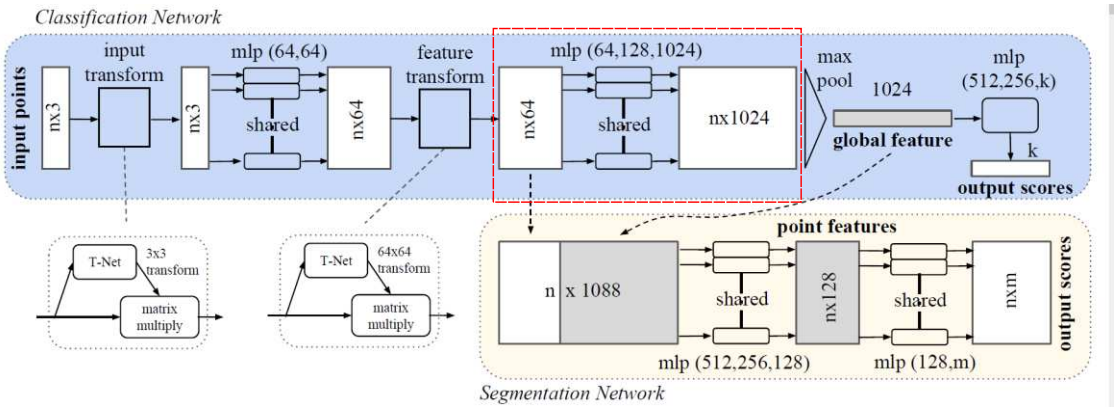


Abbildung 2.8: Vierter Schritt - erneutes Feature Building [1, S. 3]

Wie in der vorherigen Iteration des Feature Building ist auch in diesem Schritt die Erzeugung von Eigenschaftswerten zur weiteren Analyse das Ziel. Der Anstieg von 64 auf 1024 Werte pro Punkt bedeutet hierbei, dass lokale Eigenschaften und Nachbarschaftsbeziehungen wesentlich feingranularer dargestellt werden können.

2.1.5. Max-Pooling

In der Pooling-Schicht des PointNet werden die erzeugten Features zu einer globalen Signatur aggregiert. Hierbei wird die Max-Funktion genutzt, um die 1024 Feature-Werte aller Punkte Index für Index zu vergleichen und die jeweiligen größten Werte herauszufiltern. Abbildung 2.9 zeigt die Pooling-Schicht im PointNet-Workflow.

Da es sich bei der Max-Funktion um eine symmetrische Funktion handelt, ist die Reihenfolge der zu vergleichenden Werte irrelevant. In Verbindung mit den vorangegangenen Schichten bedeutet dies, dass im Rahmen des Max-Pooling für die Punktwolke eines Objektes unabhängig von der Reihenfolge ihrer Elemente und etwaiger Transformationen dieselbe globale Feature-Signatur ausgegeben wird.

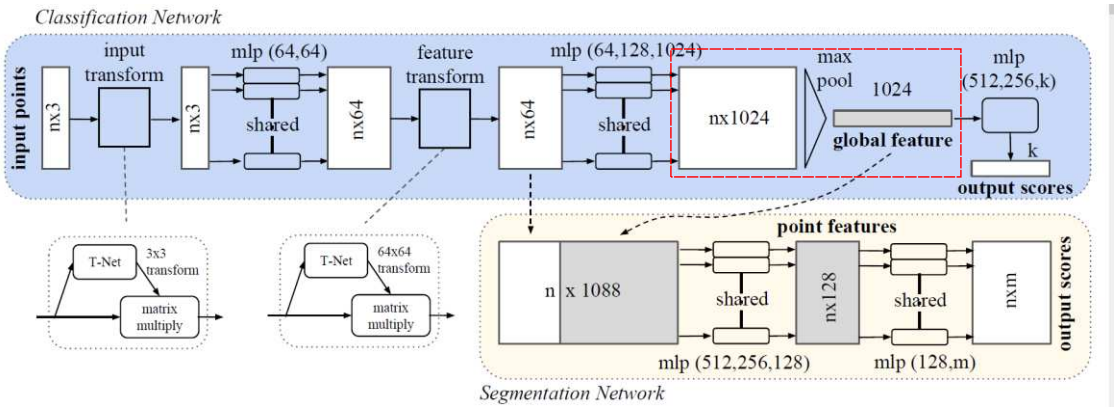


Abbildung 2.9: Fünfter Schritt - Max-Pooling

Ein weiterer interessanter Aspekt ist die Existenz von spezifischen Grenzmengen. Die erzeugte Feature-Signatur eines Objektes ist in einem gewissen Intervall der Vollständigkeit und Dichte seiner Punktwolke stets gleich. Die untere Grenze des Intervalls wird hierbei als kritische Punktmenge bezeichnet, die obere als obere Grenzform [1, S. 5].

Zum besseren Verständnis dieser Grenzen werden diese in der nachfolgenden Grafik (Abb. 2.10) anhand von vier Beispielen visualisiert. Die Farbkodierung dient der Darstellung von Tiefeninformationen.

Anhand der Grafik wird deutlich, dass PointNet aufgrund der Existenz solcher Grenzmengen einen entscheidenden Vorteil besitzt: Das Netzwerk ist robust gegenüber unvollständigen Datensätzen. Diese Robustheit ist so stark, dass Qi et al. empirisch selbst bei Datensätzen mit 60 Prozent fehlenden Daten eine Genauigkeit von 80 Prozent nachweisen konnten [1, S. 10].

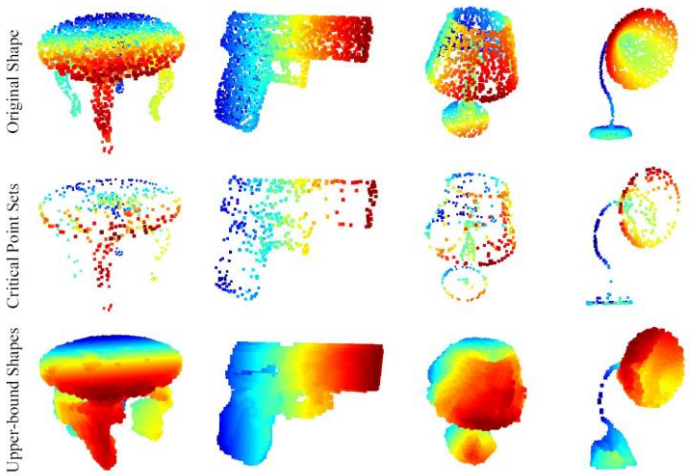


Abbildung 2.10: Kritische Punktmenge und obere Grenzform [1, S. 8]

2.1.6. Klassifizierung

Bevor in diesem Abschnitt auf den finalen Schritt des Klassifizierungsnetzwerks eingegangen wird, ist ein kurzer Rückblick auf die vorangegangenen Abschnitte sinnvoll.

Das Ziel des Netzwerks ist es, ein durch eine Punktwolke repräsentiertes Objekt invariant gegenüber Permutationen und Transformationen zu klassifizieren. Hierfür werden die 3D-Punktvektoren schrittweise räumlich normalisiert und zum Erzeugen von Features auf den $\mathbb{R}^{1024}$ abgebildet. Aus den gewonnenen Features wird mittels Max-Pooling eine aus 1024 Werten bestehende globale Signatur erzeugt. Alle Schritte bis zu diesem Punkt dienen also lediglich dazu, eine beliebige Punktwolke in eine abstrahierte und standardisierte Form zu bringen, die den Anforderungen genügt.

Der eigentliche Klassifizierungsschritt, welcher in Abbildung 2.11 markiert ist, beinhaltet lediglich einen MLP. Dieser erzeugt über zwei FC-Layer mit ReLU und die Anwendung der Softmax-Funktion in der Ausgabeschicht aus den 1024 Werten der Feature-Signatur einen k-dimensionalen Vektor. Die Variable k steht hierbei für die Anzahl der Objektklassen.

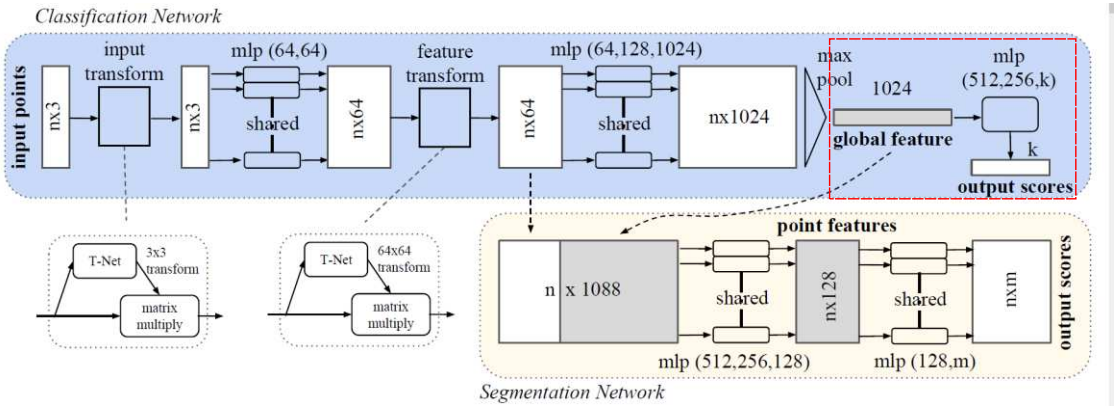


Abbildung 2.11: Sechster Schritt – Klassifizierung

2.2. Workflow der PointNet-Architektur: Segmentierung

Neben der simplen Klassifizierung von Objekten ist es für verschiedenste Aufgaben und Anwendungsbereiche von großer Bedeutung, die einzelnen Bestandteile eines Objektes erkennen zu können. Hierbei spricht man von der Segmentierung eines Objektes.

In diesem Abschnitt wird erläutert, wie PointNet durch eine Kombination lokaler und globaler Informationen die Segmentierung von Punktwolken ermöglicht. Der hierfür verwendete Teil des PointNet wird als Segmentierungsnetzwerk bezeichnet und ist in Abbildung 2.12 markiert.

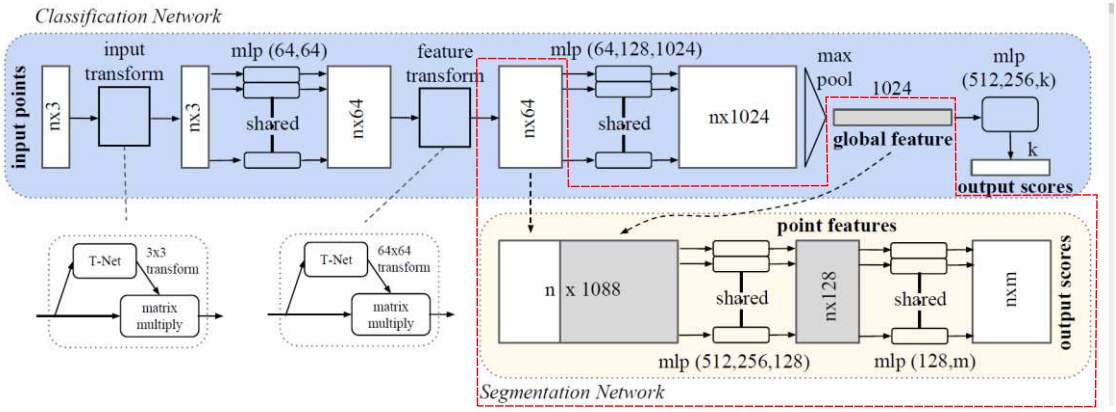


Abbildung 2.12: Segmentierungsnetzwerk

2.2.1. Feedback der globalen Signatur

Bei der Segmentierung muss ein KNN für jedes Element eines Objektes bestimmen, in welchem Teil des Objektes es sich befindet. Für PointNet heißt dies, dass jedem Element einer Punktwolke ein Wert bzw. eine Rolle für das entsprechende Segment zugeordnet werden muss. Für diese Zuordnung werden sowohl lokale als auch globale Informationen benötigt. Das Netzwerk muss die lokale Geometrie bzw. lokale Features kennen und benötigt die Daten des Gesamtobjekts als Kontext zu deren Einordnung.

PointNet löst diese Aufgabe über einen Feedback-Mechanismus. Wie in Abschnitt 2.1.4 beschrieben, werden die Feature-Vektoren im $\mathbb{R}^{64}$ für die Segmentierung zwischengespeichert. Die im späteren Verlauf des Klassifizierungsnetzwerks erzeugte globale Feature-Signatur wird an jeden einzelnen Feature-Vektor angehängt. Diese Konkatenation ergibt Vektoren mit 1088 Werten, die sowohl die zuvor gelernten lokalen Features als auch die globale Signatur des Objektes beinhalten. Somit ist die zuvor beschriebene Voraussetzung zur Segmentierung effizient und effektiv erfüllt [1, S. 4].

2.2.2. Erzeugung der Punktlabels

Die konkatenierten Feature-Vektoren werden im Segmentierungsnetzwerk über einen MLP von 1088 Werten auf m-dimensionale Output-Vektoren reduziert, wobei m den möglichen Segmentklassen entspricht. Der MLP enthält insgesamt vier Schichten, die wie beim Feature Building des Klassifizierungsnetzwerks über geteilte Gewichtungsparameter und keine Verknüpfungen zwischen benachbarten Punkten verfügen. Hervorgehoben wird dieser Teil des Workflows in Abbildung 2.13.

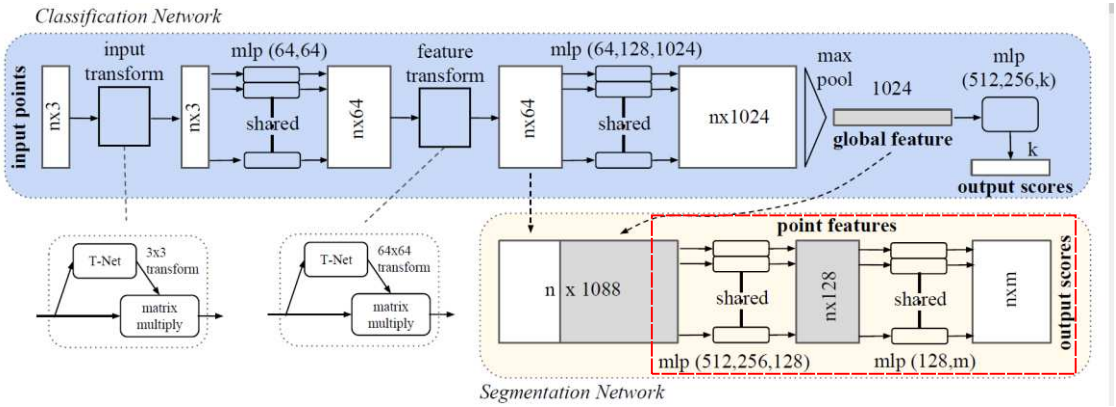


Abbildung 2.13: Erzeugung von Punktlabels

Neben der Segmentierung erlaubt dieser Teil des Netzwerks zudem die Bestimmung weiterer Werte, welche sowohl von der lokalen Geometrie als auch der globalen Semantik abhängen. So zeigen Qi et al., dass mithilfe von PointNet auch die Normalen aller Elemente einer Punktwolke akkurat bestimmt werden können [1, S. 4].

2.3. Workflow der PointNet-Architektur: Semantik

Die semantische Analyse von 3D-Szenen ist zwar nicht per se Teil von PointNet. Allerdings kann das Segmentierungsnetzwerk ohne größere Umstände erweitert und für die Segmentierung von Szenen anpassen. Hierfür werden als Punktlabels anstelle von Objektteilen ganze Objekte vergeben. Anders ausgedrückt: Anstatt eines Objektes wird beispielsweise die Punktwolke eines Zimmers als Gesamt-

heit betrachtet. Den Elementen der Punktwolke werden Objektklassen zugeordnet. Auf diese Weise ist PointNet in der Lage, die einzelnen Bestandteile einer Szene zu lokalisieren und zu klassifizieren [1, S. 6ff.].

Abbildung 2.14 zeigt die Visualisierung dreier Beispiele für die semantische Segmentierung von 3D-Szenen. Neben der Eingabe-Punktwolke werden das Ergebnis von PointNet (pred) und das vorgegebene Ergebnis (GT) im Vergleich dargestellt. Die einzelnen semantischen Klassen werden mit unterschiedlichen Farben dargestellt.

Im gegebenen Beispiel ist zu beachten, dass die Fehler der Eingabedaten (bspw. der Stuhl im zweiten Beispiel) absichtlich eingebaut wurden, um die Robustheit von PointNet gegenüber unvollständigen Punktwolken zu zeigen [1, S. 19].

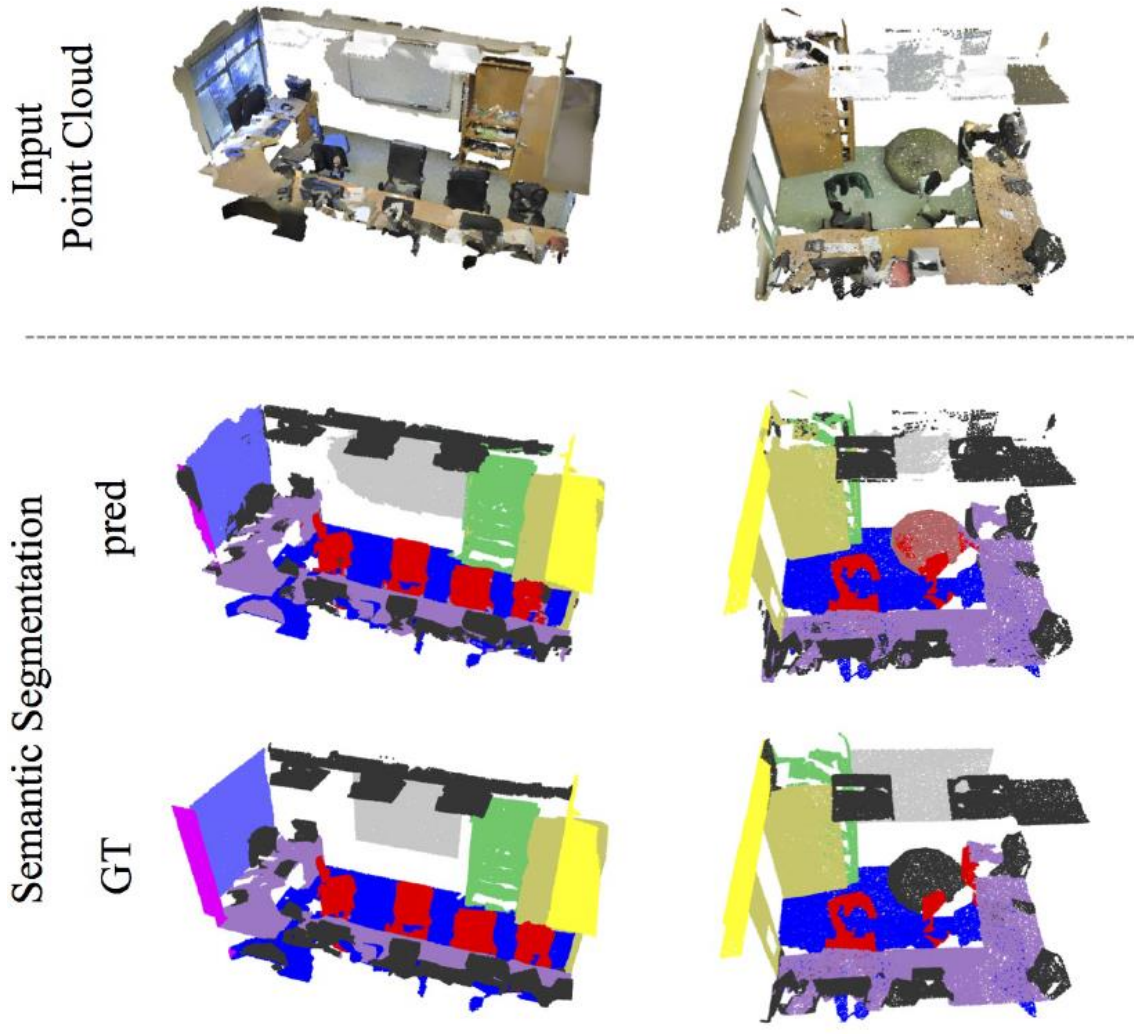


Abbildung 2.14: Beispiel für semantische Analyse von Szenen

3. PointNet in der Praxis – Implementation mit Keras

Um die theoretische Analyse des vorangegangenen Kapitels anhand eines praktischen Beispiels zu unterstützen, wurden im Rahmen dieser Arbeit mehrere Jupyter Notebooks angefertigt. Diese Notebooks beinhalten eine interaktive Beispiel-Implementation von PointNet mit der Deep-Learning-API Keras.

Die Codebasis für die Notebooks entstammt folgendem Git-Repository des Autors G. Li: https://github.com/garyli1019/pointnet-keras. Die Implementation wurde aufgrund ihrer Kompaktheit und Simplizität gewählt und für die Anfertigung der Jupyter Notebooks auf die für das Verständnis von PointNet nötigen Elemente reduziert.

Eine statische Version der Jupyter Notebooks ist im Anhang beigefügt. Die Notebooks selbst sind in einem Git-Repository unter der URL https://github.com/m-117/PointNet-ein-Implementationsbeispiel-mit-Jupyter-Notebooks.git hinterlegt und können bspw. mithilfe des Toolkits Anaconda in einer interaktiven Form geöffnet werden. Hierbei ist zu beachten, dass nach der Installation von Anaconda die Pakete für Tensorflow und Keras zur Root-Umgebung hinzugefügt und heruntergeladen werden müssen. Nähere Informationen zur Installation von Anaconda und dem Hinzufügen von Paketen sind der entsprechenden Dokumentation zu entnehmen.

4. Anwendungsgebiete der PointNet-Architektur

Nachdem PointNet in den beiden vorangegangenen Kapiteln sowohl auf theoretischer Basis als auch anhand einer simplen Implementation erläutert wurde, stellt sich die Frage nach möglichen Anwendungsgebieten der Architektur. Im Folgenden werden daher drei Projekte vorgestellt, welche auf PointNet basieren.

4.1. Erkennung von 3D-Handposen

Im Rahmen der Arbeit von Ge et al. [15] wird die Erkennung von komplexen Handposen behandelt. Basierend auf PointNet [1] und der Weiterentwicklung PointNet++[2], welche in Kapitel 5 kurz beschrieben wird, entwickeln die Autoren ein eigenes neuronales Netzwerk. Dieses Netzwerk ist in der Lage, anhand einer Punktwolken-Darstellung der Handoberfläche die Haltung und Pose der Hand zu bestimmen.

Der schematische Aufbau des von Ge et al. entworfenen Netzwerks wird in Abbildung 4.1 illustriert. In diesem wird zunächst mithilfe einer bounding box die globale Rotation der Hand kompensiert. Anschließend durchläuft die Punktwolke ein angepasstes PointNet, dessen Ergebnis die Gelenkpositionen der Hand sind. Diese werden mithilfe eines weiteren modifizierten PointNet analysiert, um die Position und Ausrichtung der Fingerspitzen möglichst genau erkennen zu können.

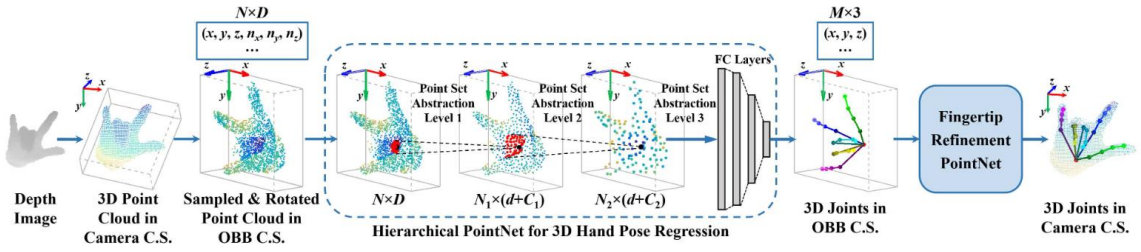


Abbildung 4.1: PointNet zur Erkennung von 3D-Handposen [15, S. 2]

In den Experimenten der Autoren zeigt sich diese Art der Erkennung von Handposen wesentlich performanter als andere Methoden [15].

4.2. Erkennung von 3D-Umgebungen zur Unterstützung von Robotik-Prothesen

Eine weitere Anwendung für die PointNet-Architektur ist die von Zhang et al. [16] beschriebene Klassifizierung von 3D-Umgebungen zur Unterstützung tragbarer Robotik-Produkte. Die Autoren entwickeln eine gerichtete Variante der PointNet-Architektur, um mithilfe eines Sensor-System dessen unmittelbare Umgebung zu analysieren.

Abbildung 4.2 veranschaulicht, wie die realen Sensordaten zunächst reduziert und anschließend an das modifizierte PointNet übergeben werden, um beispielweise die Existenz einer Treppe sowie deren Ausrichtung (aufwärts/abwärts) zu erkennen. Dies soll beispielsweise den Einsatz von Exoskelett-Prothesen in komplexen alltäglichen Situationen ermöglichen [16].

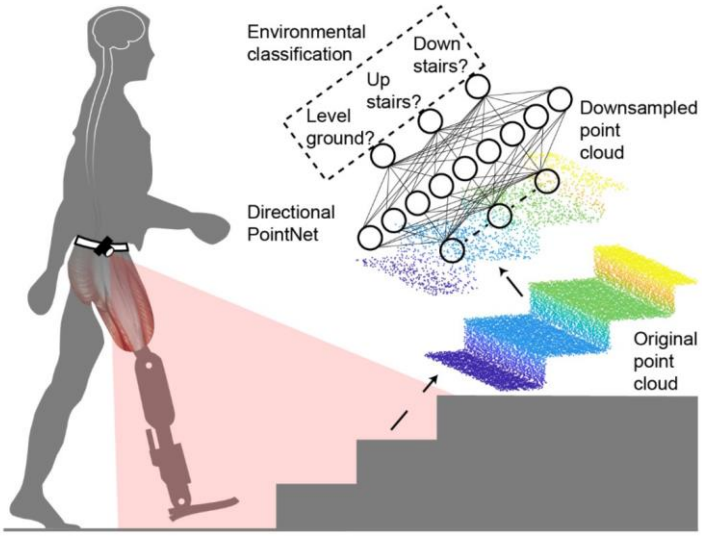


Abbildung 4.2: Gerichtetes PointNet für 3D-Umgebungsanalyse [16, S. 2]

Durch das Hinzufügen von Informationen über die Orientierung der Punktwolke zum Analyseprozess konnten die Autoren das T-Net der ursprünglichen PointNet-Architektur eliminieren und erreichen experimentell eine Genauigkeit von 99% im Test-Datensatz für alltägliche Terrains [16].

4.3. Analyse von Landschafts-Laserscans

Im dritten und letzten vorgestellten Beispiel wird die PointNet-Architektur von Lindenbergh et al. [17] genutzt, um die Daten luftgestützter Laserscans von Landschaften zu analysieren. Das Ziel der Autoren ist der Nachweis, dass neuronale Netzwerke wie bspw. PointNet dazu eingesetzt werden können, um anhand von Laserscans auf nationaler Ebene zuverlässig Entitäten wie Gebäude oder Vegetation klassifizieren zu können.

Eines der Testergebnisse, welches die Autoren im Rahmen ihrer Arbeit gewinnen konnten, wird in der nachfolgenden Grafik (Abb. 4.3) dargestellt. Im direkten Vergleich zwischen dem Output des neuronalen Netzwerks (a) und der tatsächlichen Werte (b) zeigt sich eine nicht unerhebliche Anzahl von Fehlklassifizierungen von Gebäuden (gelb) als Vegetation (grün). Nichtsdestotrotz erreichen die Autoren eine allgemeine Genauigkeit von 87%. Sie schließen daraus, dass die vorgeschlagene Anwendung ihres Netzwerks zur großflächigen Analyse von Landschafts-Scans praktikabel ist, aber weiterer Experimente und Anpassungen bedarf [17].

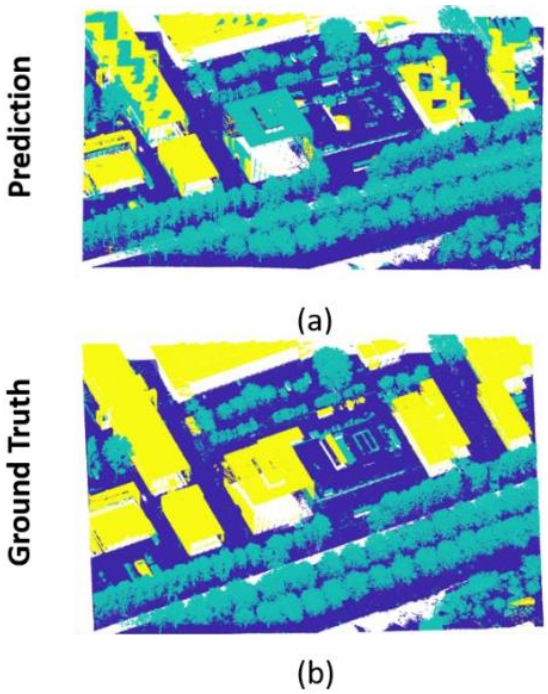


Abbildung 4.3: Beispiel für die Analyse von Landschafts-Scans [17, S. 6]

5. PointNet++ - eine Weiterentwicklung

Wie im vorangegangenen Kapitel gezeigt, bietet PointNet als eine der ersten Deep-Learning-Architekturen zur direkten Verarbeitung von Punktwolken eine Grundlage für Lösungen in einer Vielzahl von Anwendungsfällen. Hierbei wird PointNet oft modifiziert, um die gewünschte Funktionalität zu erreichen. Eine solche Modifikation bzw. Erweiterung ist auch das von Qi et al. vorgestellte PointNet++[2], welches in diesem Kapitel kurz beschrieben wird.

Aufgrund seines Aufbaus und seiner Funktionsweise erfasst PointNet keine lokalen Strukturen, welche durch den metrischen Raum, der die Punktwolke enthält, induziert werden. Dies schränkt die Fähigkeit des Netzwerks, feinkörnige Muster zu erkennen und auf komplexe Szenen verallgemeinert zu werden, stark ein. PointNet++ löst dieses Problem durch die rekursive Partitionierung des Inputs in lokale, überlappende Punktcluster, auf welche PointNet angewendet wird [2, S. 1].

Der Ablauf und die Elemente von PointNet++ sind, wie in der untenstehenden Grafik (Abb. 5.1) dargestellt, in drei Gruppen aufgeteilt. Während der Phase *sampling & grouping* werden einzelne Elemente der Punktwolke als Zentren lokaler Cluster ausgewählt und die umliegenden Punkte innerhalb eines festgelegten Radius ermittelt. Anschließend wird PointNet auf diese Cluster angewendet, um lokale Informationen zu gewinnen. Dieser Prozess wird wiederholt, sodass eine für die Klassifizierung bzw. Segmentierung genügende Darstellung erreicht wird.

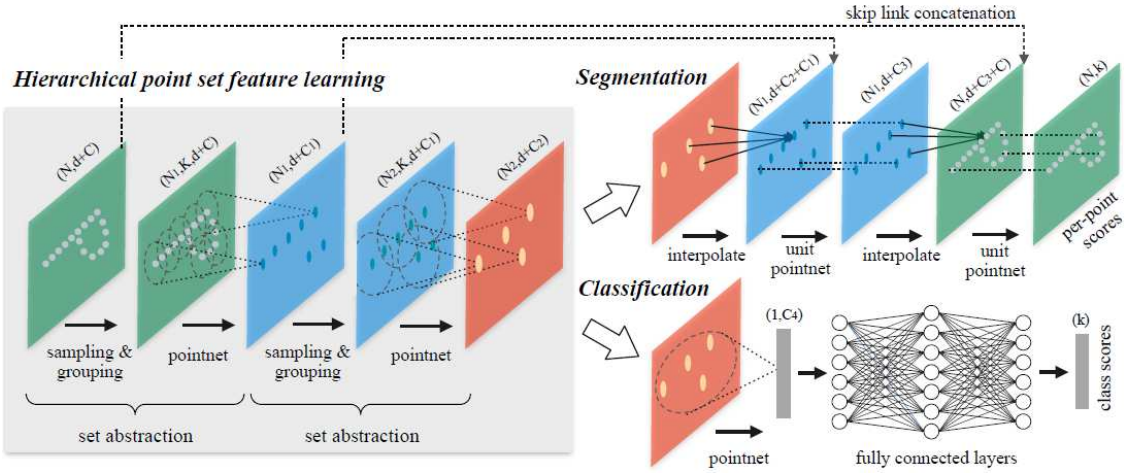


Abbildung 5.1: Aufbau PointNet++ [2, S. 3]

Im Rahmen der Klassifizierung wird PointNet auf die gewonnenen Features angewendet und die globale Signatur ermittelt. Bei der Segmentierung wird der eingangs durchgeführte hierarchische Abstraktionsprozess umgekehrt. Via Interpolation werden auf Basis der Cluster-Features die Labels der einzelnen Punkte bestimmt.

Eine wichtige Eigenschaft von PointNet++ ist die Adaptivität der PointNet-Schichten in Bezug auf die Dichteverteilung einer Punktwolke. Da die Elemente einer Punktwolke in realen Datensätzen normalerweise nicht uniform verteilt sind, gestaltet sich die Erkennung feingranularer Features schwierig. Um dieses Problem zu lösen, sind die PointNet-Schichten in PointNet++ adaptiv gestaltet. Sie lernen, Features aus Regionen unterschiedlicher Größe zu kombinieren, wenn sich die Punktdichte ändert [2, S. 4].

PointNet++ erweist sich als äußerst robust gegenüber unvollständigen Datensätzen und erzielt selbst bei Punktwolken mit weniger als 200 Elementen eine Genauigkeit von fast 90 Prozent. Darüber hinaus erzielen die Autoren in der semantischen Analyse von 3D-Szenen wesentlich bessere Ergebnisse als mit dem ursprünglichen PointNet. Abbildung 5.2 illustriert anhand des Beispiels zweier Räume, dass PointNet (a) zwar die Struktur des Raumes erkennt, aber im Vergleich zu PointNet++ (b) klare Defizite in der Klassifizierung der Objekte aufweist.

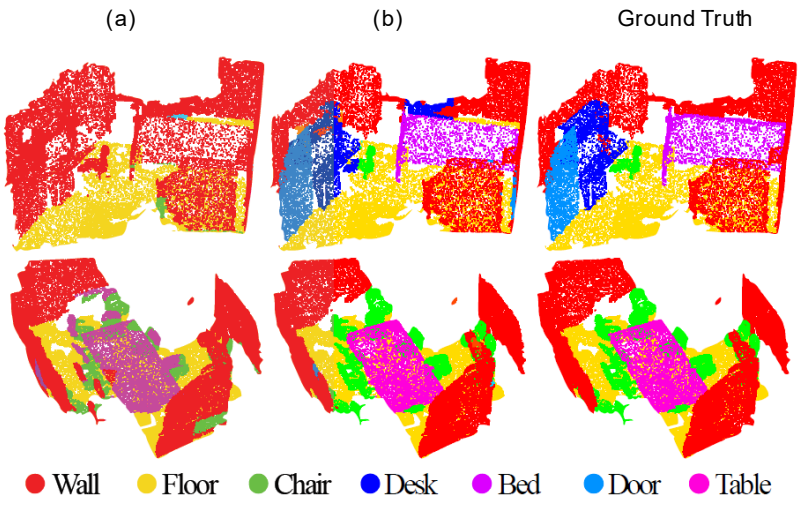


Abbildung 5.2: Vergleich PointNet / PointNet++

Da eine detaillierte Analyse der Bestandteile und Funktionsweise von PointNet++ den Rahmen dieser Arbeit sprengen würde, sei für die weitere Lektüre auf die Publikation von Qi et al. verwiesen[2].

6. Fazit

Ziel dieser Arbeit war die Analyse und Erläuterung der von Qi et al. vorgestellten PointNet-Architektur [1]. Zu Beginn wurden die für das Verständnis der späteren Kapitel notwendigen Grundlagen neuronaler Netzwerke behandelt. Anschließend wurde die PointNet-Architektur sowohl auf theoretischer Ebene als auch anhand einer praktischen Beispiel-Implementation Schritt für Schritt erläutert. Für den praktischen Teil wurden mehrere interaktive Jupyter Notebooks angefertigt, mit deren Hilfe das Verständnis für die Funktionsweise eines PointNet vertieft werden soll. Darüber hinaus wurden drei Projekte vorgestellt, in denen PointNet für die von den jeweiligen Autoren entwickelte Lösung verwendet wurde. Abschließend wurde mit PointNet++ eine Erweiterung der PointNet-Architektur vorgestellt.

PointNet bietet eine neuartige Herangehensweise für die Analyse dreidimensionaler Daten und zeigt basierend auf den empirischen Ergebnissen der Autoren enormes Potential. Die direkte Verarbeitung von Punktwolken eliminiert die Notwendigkeit etwaiger Konversionen und ermöglicht effiziente und effektive Echtzeit-Analysen. Die vorgestellten Beispiele zeigen zudem die Versatilität der PointNet-Architektur.

Da es sich bei PointNet um eine Pionierarbeit im Bereich des Deep Learning mit Punktwolken handelt [2, S. 1], bieten sich eine Vielzahl von Möglichkeiten, die Architektur zu modifizieren und zu verbessern. Ein möglicher Ansatzpunkt ist hierbei die Verwendung von Prinzipien des *reinforcement learning*, um die Effizienz und Geschwindigkeit des Trainingsprozesses zu erhöhen. Hierbei werden Informationen über die Performanz des Netzwerks während des Trainingsprozesses über einen Feedback-Mechanismus zur adaptiven Auswahl von Trainingsdaten genutzt. Sollten sich die Ergebnisse von ähnlichen Versuchen wie beispielsweise der Arbeit von Schaul et al. [18] auf PointNet übertragen lassen, könnte dies das Potential der Architektur erneut steigern.

7. Quellenverzeichnis

[1] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation,” *CoRR*, vol. abs/1612.00593, 2016, [Online]. Available: http://arxiv.org/abs/1612.00593.

[2] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, “PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. v Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 5099–5108.

[3] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[4] D. A. Pomerleau, “Efficient Training of Artificial Neural Networks for Autonomous Navigation,” *Neural Computation*, vol. 3, no. 1, pp. 88–97, 1991, doi: 10.1162/neco.1991.3.1.88.

[5] L. A. Gatys, A. S. Ecker, and M. Bethge, “A Neural Algorithm of Artistic Style,” *CoRR*, vol. abs/1508.06576, 2015, [Online]. Available: http://arxiv.org/abs/1508.06576.

[6] J. Heaton, *AIFH, Volume 3: Deep Learning and Neural Networks*. 2015.

[7] W. Bibel, R. Kruse, and B. Nebel, “Computational Intelligence: Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze,” 2015. [Online]. Available: http://www.springer.com/series/12572.

[8] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[9] F. Chollet, *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[11] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *CoRR*, vol. abs/1502.03167, 2015, [Online]. Available: http://arxiv.org/abs/1502.03167.

[12] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, “Spatial Transformer Networks,” *CoRR*, vol. abs/1506.02025, 2015, [Online]. Available: http://arxiv.org/abs/1506.02025.

[13] Y. LeCun, C. Cortes, and C. J. C. Burges, “The MNIST database of handwritten digits,” 1998, Accessed: Jun. 08, 2020. [Online]. Available: http://yann.lecun.com/exdb/mnist.

[14] L. Gonzales, “An In-Depth Look at PointNet,” Apr. 2019, Accessed: Jun. 02, 2020. [Online]. Available: https://medium.com/@luis_gonzales/an-in-depth-look-at-pointnet-111d7efdaa1a.

[15] L. Ge, Y. Cai, J. Weng, and J. Yuan, “Hand pointnet: 3d hand pose estimation using point sets,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8417–8426.

[16] K. Zhang, J. Wang, and C. Fu, “Directional PointNet: 3D Environmental Classification for Wearable Robotics,” *CoRR*, vol. abs/1903.06846, 2019, [Online]. Available: http://arxiv.org/abs/1903.06846.

[17] M. Soilán, R. Lindenbergh, B. Riveiro, and A. Sánchez-Rodríguez, “Pointnet for the automatic classification of aerial point clouds,” 2019.

[18] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” *arXiv preprint arXiv:1511.05952*, 2015.

Anhang

Einleitung

June 14, 2020

PointNet: Ein interaktives Implementationsbeispiel mit Jupyter Notebooks

Dieses Juypter-Notebook sowie die zugehörigen Notebooks wurden im Rahmen einer wissenschaftlichen Vertiefungsarbeit entworfen. Das Ziel dieser Arbeit ist es, einen Einstieg in die Funktionalität künstlicher neuronaler Netzwerke zu geben und die PointNet-Architektur von Qi et al. [1] detailliert zu erläutern.

Um PointNet nicht nur auf theoretischer Ebene zu erklären, wird im Rahmen dieser Notebooks eine simple Implementation mit der Deep Learning API Keras Schritt für Schritt analysiert. Der Nutzer kann bei seiner Lektüre die Parameter des Netzwerks beliebig editieren und die Effekte beim Training live mitverfolgen.

1 Installationshinweise

Da es sich bei dem Training neuronaler Netzwerke um einen zeit- und ressourcenintensive Prozess handelt, wird für die interaktive Lektüre eine lokale Installation von Python und den nötigen Paketen benötigt.

Hierfür empfiehlt sich das Toolkit **Anaconda**.

Nach Installation von Anaconda werden wahlweise über den Anaconda-Navigator oder die Konsole die Pakete für Tensorflow und Keras für die root-Umgebung heruntergeladen.

Mithilfe der Anaconda-App "Jupyter Notebook" startet ein lokaler Python-Server, welcher die interaktive Lektüre im Webbrowser ermöglicht.

1.1 Inhalt

[PointNet - Klassifizerungsnetzwerk]

Analyse des PointNet-Klassifizierungsnetzwerks anhand einer Beispielimplementation. Neben den Bestandteilen des Netzwerks werden auch die Art und Form des Inputs erläutert. Das Netzwerk kann mit individuell editierbaren Parametern trainiert und anschließend gespeichert werden.

[PointNet - Segmentierungsnetzwerk]

Analyse des PointNet-Segmentierungsnetzwerks anhand einer Beispielimplementation. Neben den Bestandteilen des Netzwerks werden auch die Art und Form des Inputs erläutert. Das Netzwerk kann mit individuell editierbaren Parametern trainiert und anschließend gespeichert werden.

Github-Repository

https://github.com/m-117/PointNet-ein-Implementationsbeispiel-mit-Jupyter-Notebooks

Quellen:

[1] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation,” CoRR, vol. abs/1612.00593, 2016, [Online]. Available: http://arxiv.org/abs/1612.00593.

PointNet-Klassifizierungsnetzwerk

June 18, 2020

1 PointNet-Klassifizierungsnetzwerk

Im Rahmen dieses Notebooks wird anhand einer beispielhaften Implementation die praktische Anwendung der PointNet-Architektur zur Klassifizierung dreidimensionaler Objekte in Form von Punktwolken gezeigt. Der verwendete Code entstammt einer Implementation von G. Li, zu finden im folgenden Git-Repository: https://github.com/garyli1019/pointnet-keras.

Die folgende Grafik von Qi et al. [1] zeigt die Architektur von PointNet. Der markierte Bereich stellt hierbei das Klassifikationsnetzwerk dar. Diese Darstellung wird im weiteren Verlauf des Notebooks wiederholt genutzt, um das Durchlaufen der Pipeline zu visualisieren.

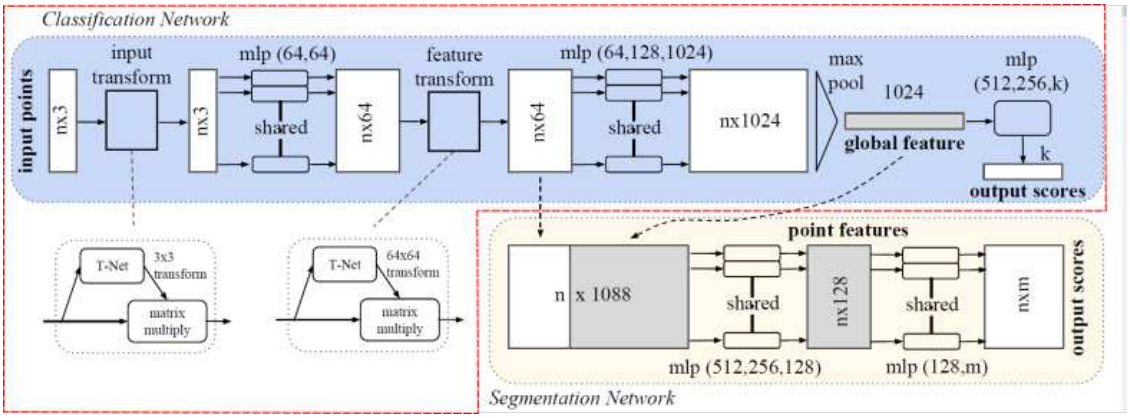


Abbildung 1: PointNet-Klassifizierungsnetzwerk [1]

1.1 Anwendungshinweis

Da es sich bei dem Code in diesem Notebook um ein zusammenhängendes Script handelt, empfiehlt sich bei Änderungen stets das Zurücksetzen des Kernels. Hierfür wird unter dem Reiter "Kernel" die Option "Restart & Run all" ausgewählt.

1.2 Imports

Für die hier analyiserte Keras-Implementation der PointNet-Architektur werden drei Pakete benötigt: * NumPy für die Verwaltung von Datenstrukturen sowie der Handhabung von Vek-

toren und Matrizen * H5Py als Schnittstelle für das HDF5-Datenformat * Tensorflow für die Verwendung der Keras-API, welche Teil des Tensorflow-Cores ist

Darüber hinaus werden die nötigen Keras-Pakete für die Schichttypen und die Erzeugung des Modells importiert.

[1]:
```
import warnings
warnings.simplefilter(action='ignore')
import numpy as np
import os
import tensorflow as tf
from keras import optimizers
from keras.layers import Input
from keras.models import Model
from keras.layers import Dense, Flatten, Reshape, Dropout
from keras.layers import Convolution1D, MaxPooling1D, BatchNormalization
from keras.layers import Lambda
from keras.utils import np_utils
import h5py

tf.logging.set_verbosity(tf.logging.ERROR)
```

```
Using TensorFlow backend.
```

1.3 Hilfsfunktionen

Nachfolgend werden Hilfsfunktionen definiert. Die Funktion mat_mul wird benötigt, um die Matritzenmultiplikation innerhalb einer Keras-Schicht anwenden zu können. Load_h5 dient dem Laden einer HPF5-Datei und gibt deren Inhalt und Labels zurück.

[2]:
```
def mat_mul(A, B):
    return tf.matmul(A, B)
```

[3]:
```
def load_h5(h5_filename):
    f = h5py.File(h5_filename)
    data = f['data'][:]
    label = f['label'][:]
    return (data, label)
```

Die Funktionen rotate_point_cloud sowie jitter_point_cloud dienen dazu, die Trainingsdaten wiederholt nutzbar zu machen. Durch zufällige Rotationen und Mikro-Translationen der Objekte bzw. Punkte erzeugen die Funktionen aus einem Datensatz eine beliebige Menge unterschiedlicher Trainingsdaten.

rotate_point_cloud erzeugt zunächst eine Nullmatrix mit den Dimensionen des Inputs. Anschließend wird für jedes Objekt eine zufällige Rotationsmatrix erzeugt. Die Produkte der Rotationsmatrix und der Objektpunkte werden in die Nullmatrix eingefügt, welche nach Durchlaufen aller Objekte zurückgegeben wird. Die Rotationsachse ist die Y-Achse.

jitter_point_cloud erzeugt ebenfalls eine Kopie des Inputs. Diese wird jedoch nicht als Nullmatrix initialisiert, sondern mit kleinen Zufallszahlen gefüllt. Anschließend erfolgt eine elementweise Addition mit der Input-Matrix.

[4]:
```
def rotate_point_cloud(batch_data):
    """ Randomly rotate the point clouds to augument the dataset
        rotation is per shape based along up direction
        Input:
          BxNx3 array, original batch of point clouds
        Return:
          BxNx3 array, rotated batch of point clouds
    """
    rotated_data = np.zeros(batch_data.shape, dtype=np.float32)
    for k in range(batch_data.shape[0]):
        rotation_angle = np.random.uniform() * 2 * np.pi
        cosval = np.cos(rotation_angle)
        sinval = np.sin(rotation_angle)
        rotation_matrix = np.array([[cosval, 0, sinval],
                                    [0, 1, 0],
                                    [-sinval, 0, cosval]])
        shape_pc = batch_data[k, ...]
        rotated_data[k, ...] = np.matmul(shape_pc.reshape((-1, 3)),␣
↪rotation_matrix)
    return rotated_data
```

[5]:
```
def jitter_point_cloud(batch_data, sigma=0.01, clip=0.05):
    """ Randomly jitter points. jittering is per point.
        Input:
          BxNx3 array, original batch of point clouds
        Return:
          BxNx3 array, jittered batch of point clouds
    """
    B, N, C = batch_data.shape
    assert(clip > 0)
    jittered_data = np.clip(sigma * np.random.randn(B, N, C), -1 * clip, clip)
    jittered_data += batch_data
    return jittered_data
```

Die folgenden globalen Variablen bestimmen wichtige Randbedingungen der verwendeten Datensätze sowie den verwendenten Optimierungsalgorithmus und sollten daher nicht verändert werden.

[6]:
```
# number of points in each sample
num_points = 2048

# number of categories
k = 40
```

```
# define optimizer
adam = optimizers.Adam(lr=0.001, decay=0.7)
```

1.4 PointNet als Keras-Modell

In der Deep-Learning-API Keras werden künstliche neuronale Netze in Form von Modellen implementiert. Beginnend mit der Definition eines Inputs werden werden Schichten und Funktionen aneinandergereiht. Abgeschlossen wird das Modell durch die Angabe eines Outputs, welcher über die Schichten mit dem Input verbunden ist.

Die Verknüfpung erfolgt bei vorwärtsgerichteten KNN durch die Angabe der vorherigen Schicht als Parameter. Ein Beispiel:

```
input = Input(shape=(data))
op1 = Schichtfunktion(Parameter)(input)
op2 = Schichtfunktion(Parameter)(op2)
...
output = Schichtfunktion(Parameter)(opx)
```

Im Folgenden wird das Modell für PointNet aufgebaut. Hierbei findet im ersten Schritt die in der Abbildung markierte Input-Transformation statt.

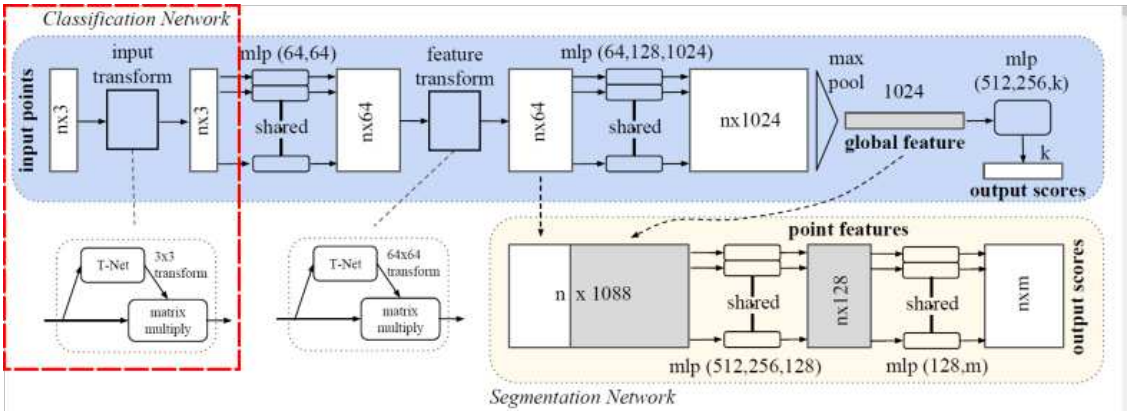


Abbildung 2: Input-Transformation [1]

Mithilfe eines eigenständigen Mini-Netzwerks bestimmt PointNet eine Transformationsmatrix, welche via Matritzenmultiplikation auf den Input angewendet wird. Hierdurch soll die räumliche Ausrichtung auf eine Basis-Pose durchgeführt werden. Der Aufbau des T-Net sieht wie folgt aus:

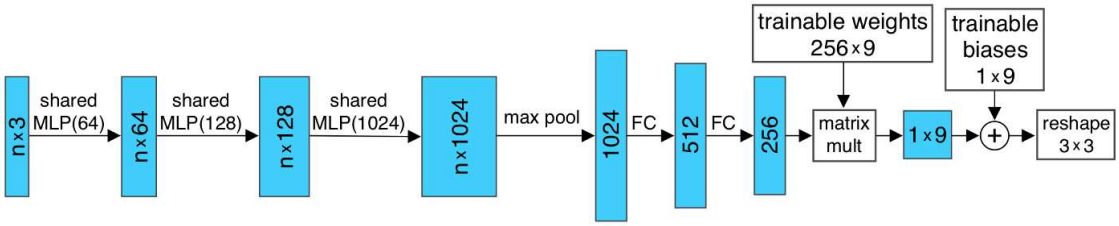


Abbildung 3: Aufbau T-Net [3]

Im nachfolgenden Code-Block wird wie zuvor beschrieben zunächst der Input definiert. Die MLP-

Schichten werden als 1D-Faltungen mit 64, 128 bzw. 1024 Filtern sowie einer Kernelgröße von 1 realisiert. Als Aktivierungfunktion wird ReLU verwendet.

Die Punkte werden demnach zwar mit geteilten Gewichten, aber dennoch individuell auf höherdimensionale Räume abgebildet. Nach jeder Schicht wird eine batch normalization durchgeführt, bei der die Output-Werte stapelweise von deren Durchschnitt subtrahiert und durch die Varianz geteilt werden.

[7]:
```
# ---------------------------------- Pointnet Architecture
# input_Transformation_net
input_points = Input(shape=(num_points, 3))
x = Convolution1D(64, 1, activation='relu',
                  input_shape=(num_points, 3))(input_points)
x = BatchNormalization()(x)
x = Convolution1D(128, 1, activation='relu')(x)
x = BatchNormalization()(x)
x = Convolution1D(1024, 1, activation='relu')(x)
x = BatchNormalization()(x)
```

Im Anschluss an die letzte Faultung werden die Features mittels Max-Pooling aggregiert und über mehrere FC-Layer zunächst auf 256 Werte reduziert. Diese sind in Form mehrerer Dense Layer implementiert.

Die letzte Dense Layer bestimmt hieraus mit 256x9 Gewichten die 9 Elemente der Transformationsmatrix `input_T`. Der Output wird als Einheitsmatrix initialisiert und im finalen Schrit in die gewünschte 3x3-Form gebracht.

Auch hier wird mit Ausnahme des letzten Schrittes nach jeder Schicht eine batch normalization durchgeführt.

[8]:
```
x = MaxPooling1D(pool_size=num_points)(x)
x = Dense(512, activation='relu')(x)
x = BatchNormalization()(x)
x = Dense(256, activation='relu')(x)
x = BatchNormalization()(x)
x = Dense(9, weights=[np.zeros([256, 9]), np.array([1, 0, 0, 0, 1, 0, 0, 0, 1]).
 ↪astype(np.float32)])(x)
input_T = Reshape((3, 3))(x)
```

Nachdem die Transformationsmatrix bestimmt wurde, kann diese über eine Lambda-Schicht mit dem Input multipliziert werden. Lambda-Schichten ermöglichen es, beliebige Funktionen in das Modell eines neuronalen Netzwerks einzubauen.

Das Feature-Building wird gemäß der Architekturvorgaben - markiert in der nachfolgenden Abbildung - mittels zwei 1D-Faltungen mit 64 Filtern, einer Kernelgröße von 1 und der Aktivierungsfunktion ReLU durchgeführt.

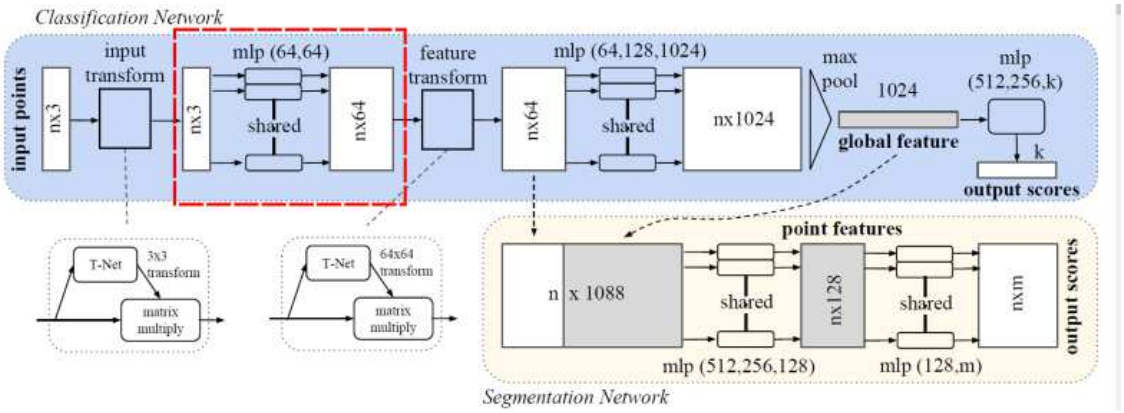


Abbildung 4: Feature Building 1 [1]

[9]:
```
# forward net
g = Lambda(mat_mul, arguments={'B': input_T})(input_points)
g = Convolution1D(64, 1, input_shape=(num_points, 3), activation='relu')(g)
g = BatchNormalization()(g)
g = Convolution1D(64, 1, input_shape=(num_points, 3), activation='relu')(g)
g = BatchNormalization()(g)
```

Der dritte Schritt im PointNet-Klassifizierungsnetzwerk ist die in der folgenden Abbildung markierte Feature-Transformation.

Wie im ersten Schritt wird über ein unabhängiges T-Net eine Transformationsmatrix bestimmt und mit dem aktuellen Input multipliziert. Im Aufbau unterscheidet sich das T-Net zur Feature-Transformation nur in den Werten der Parameter einzelner Schichten. So besitzt die Transformationsmatrix bspw. eine Größe von 64x64 Werten.

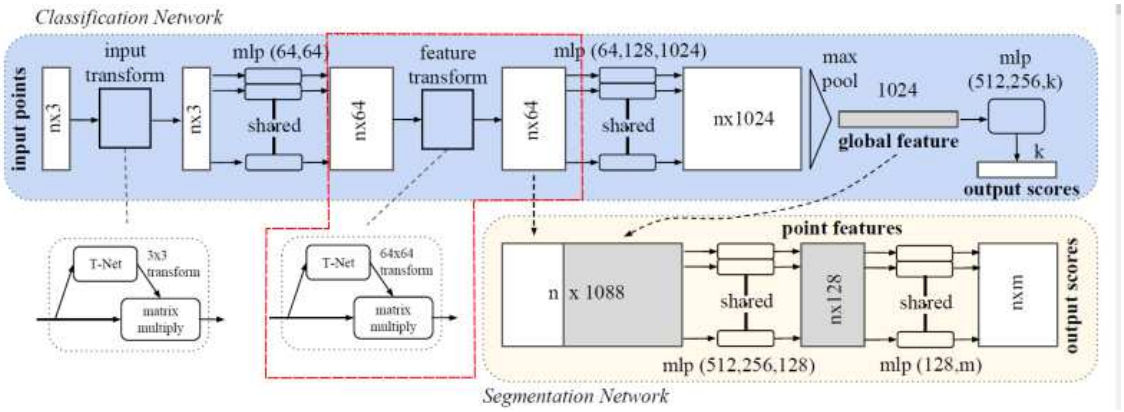


Abbildung 5: Feature-Transformation [1]

[10]:
```
# feature transform net
f = Convolution1D(64, 1, activation='relu')(g)
f = BatchNormalization()(f)
f = Convolution1D(128, 1, activation='relu')(f)
f = BatchNormalization()(f)
f = Convolution1D(1024, 1, activation='relu')(f)
```

```
f = BatchNormalization()(f)
f = MaxPooling1D(pool_size=num_points)(f)
f = Dense(512, activation='relu')(f)
f = BatchNormalization()(f)
f = Dense(256, activation='relu')(f)
f = BatchNormalization()(f)
f = Dense(64 * 64, weights=[np.zeros([256, 64 * 64]), np.eye(64).flatten().
 ↪astype(np.float32)])(f)
feature_T = Reshape((64, 64))(f)
```

Mit der Feature-Transformationsmatrix kann nun durch eine Matritzenmultiplikation der Input für die zweite Iteration des Feature Building angepasst werden. Auch hier ist das Ziel eine Ausrichtung der Daten zur Erzeugung einer einheitlichen Form, welche invariant gegenüber Transformationen des ursprünglichen Inputs ist.

Der transformierte Input wird mittels dreier 1D-Faltungsoperationen von 64 zunächst auf 128 und schließlich auf 1024 Werte abgebildet. Wie in den vorangegangenen Schritten werden hierfür Kernel der Größe 1 und ReLU als Aktivierungsfunktion genutzt. Zwischen den Schichten erfolgt batch normalization.

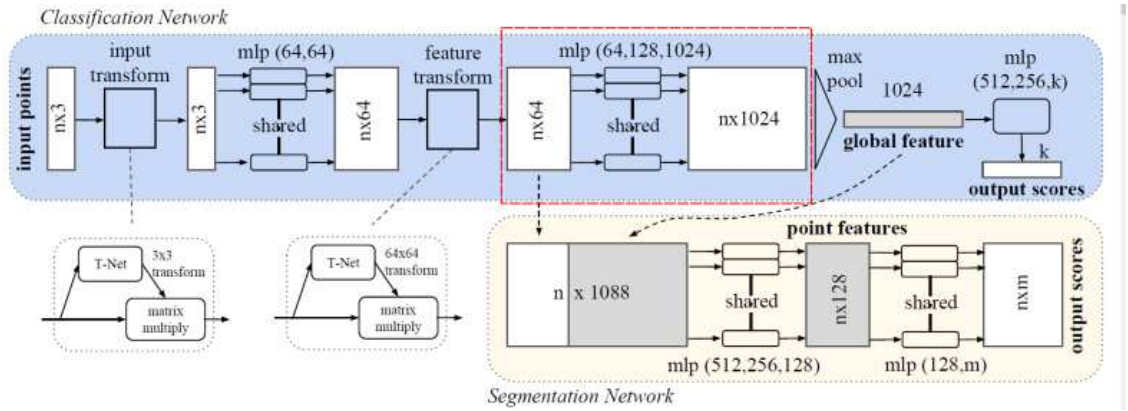


Abbildung 6: Feature Building 2 [1]

[11]:
```
# forward net
g = Lambda(mat_mul, arguments={'B': feature_T})(g)
g = Convolution1D(64, 1, activation='relu')(g)
g = BatchNormalization()(g)
g = Convolution1D(128, 1, activation='relu')(g)
g = BatchNormalization()(g)
g = Convolution1D(1024, 1, activation='relu')(g)
g = BatchNormalization()(g)
```

Für den vorletzten Schritt des Klassifizierungsnetzwerks ist lediglich eine einzelne Zeile vonnöten.

Mithilfe der MaxPooling1D-Funktion wird aus den Features eine globale Signatur erzeugt. Die Größe des Pools entspricht hierbei der Anzahl der Punkte n. Dies führt dazu, dass für jeden der 1024 Feature-Werte das Maximum aus allen Punkten herausgefiltert wird.

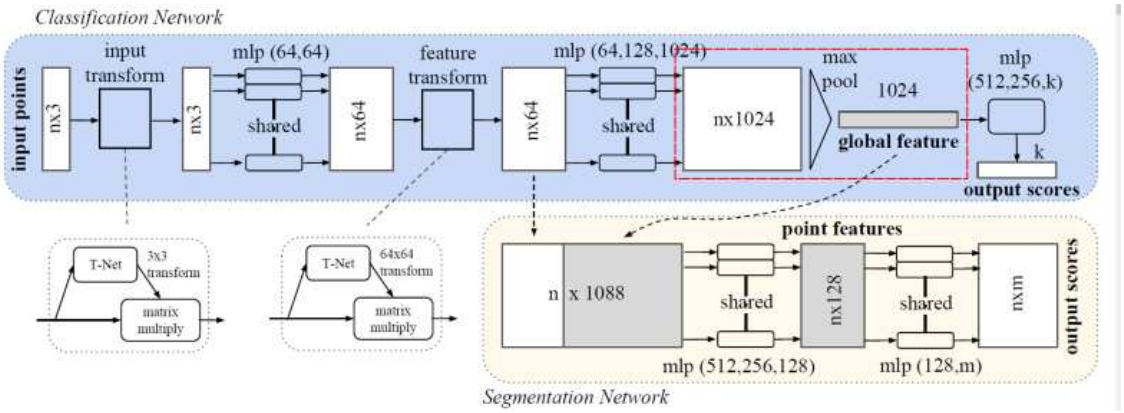


Abbildung 7: Max-Pooling [1]

[12]:
```
# global_feature
global_feature = MaxPooling1D(pool_size=num_points)(g)
```

Mit dem globalen Feature-Vektor kann nun die eigentliche Klassifizierung erfolgen. Alle vorangegangenen Schritte dienten dazu, den Input in eine klassifizierbare Form zu bringen, welche invariant gegenüber Permuationen und Transformationen ist.

Für die Klassifzierung werden zwei Dense Layer mit ReLU genutzt, um die globalen Features auf 256 Werte reduzieren. Nach jedem Schritt findet eine batch normalization. Darüber hinaus werden pro Dense Layer 30% der Punkte zufällig via Dropout entfernt.

Die dritte Dense Layer nutzt die Softmax-Aktivierungsfunktion und reduziert den Feature-Vektor auf einen k-dimensionalen Vektor, wobei k die Anzahl der Kategorien ist. Das Ergebnis wird als `prediction` gespeichert.

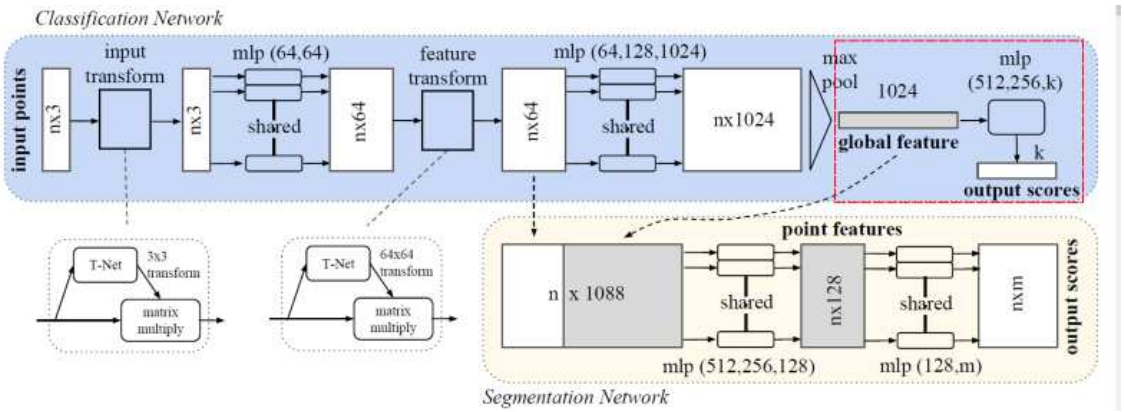


Abbildung 8: Klassifizierung [1]

[13]:
```
# point_net_cls
#TODO: Dropout auf 0.3
c = Dense(512, activation='relu')(global_feature)
c = BatchNormalization()(c)
c = Dropout(rate=0.3)(c)
c = Dense(256, activation='relu')(c)
```

```
c = BatchNormalization()(c)
c = Dropout(rate=0.3)(c)
c = Dense(k, activation='softmax')(c)
prediction = Flatten()(c)
# ---------------------------------------------------end of pointnet
```

Nachdem Input und Output definiert und über mehrere Schichten miteinander verknüpft wurden, kann das Modell definiert werden.

Die folgende Code-Zelle gibt eine Zusammenfassung über die Schichten und die Anzahl der Parameter im Netzwerk aus.

Da es sich bei diesem Jupyter Notebook um ein interaktives Werk handelt, sei der Leser herzlich eingeladen, die Parameter der Schichten zu editieren und deren Auswirkungen auf die Größe des Netzwerks und den Trainingsprozess zu beobachten. Hierfür eignen sich vor neben den Größenparametern der Schichten auch die Aktivierungsfunktionen und die Dropout-Rate.

[14]:
```
# print the model summary
model = Model(inputs=input_points, outputs=prediction)
print(model.summary())
```

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2048, 3)           0
_________________________________________________________________
lambda_1 (Lambda)            (None, 2048, 3)           0
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 2048, 64)          256
_________________________________________________________________
batch_normalization_6 (Batch (None, 2048, 64)          256
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 2048, 64)          4160
_________________________________________________________________
batch_normalization_7 (Batch (None, 2048, 64)          256
_________________________________________________________________
lambda_2 (Lambda)            (None, 2048, 64)          0
_________________________________________________________________
conv1d_9 (Conv1D)            (None, 2048, 64)          4160
_________________________________________________________________
batch_normalization_13 (Batc (None, 2048, 64)          256
_________________________________________________________________
conv1d_10 (Conv1D)           (None, 2048, 128)         8320
_________________________________________________________________
batch_normalization_14 (Batc (None, 2048, 128)         512
_________________________________________________________________
conv1d_11 (Conv1D)           (None, 2048, 1024)        132096
_________________________________________________________________
```

```
batch_normalization_15 (Batc (None, 2048, 1024)        4096
_________________________________________________________________
max_pooling1d_3 (MaxPooling1 (None, 1, 1024)           0
_________________________________________________________________
dense_7 (Dense)              (None, 1, 512)            524800
_________________________________________________________________
batch_normalization_16 (Batc (None, 1, 512)            2048
_________________________________________________________________
dropout_1 (Dropout)          (None, 1, 512)            0
_________________________________________________________________
dense_8 (Dense)              (None, 1, 256)            131328
_________________________________________________________________
batch_normalization_17 (Batc (None, 1, 256)            1024
_________________________________________________________________
dropout_2 (Dropout)          (None, 1, 256)            0
_________________________________________________________________
dense_9 (Dense)              (None, 1, 40)             10280
_________________________________________________________________
flatten_1 (Flatten)          (None, 40)                0
=================================================================
Total params: 823,848
Trainable params: 819,624
Non-trainable params: 4,224
_________________________________________________________________
None
```

Das finale Modell wird nun von Keras kompiliert. Es werden der Standard-Optimizer `Adam` sowie die Verlustfunktion `categorical_crossentropy` verwendet. Gemessen wird die Genauigkeit des Netzwerks.

[15]:
```
# compile classification model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

1.5 Vorbereitung der Daten

Die nächsten beiden Codezellen beinhalten den Ladeprozess für die Trainings- und Testdaten. Die entsprechenden HPF5-Files in den Ordnern "Prepdata" und "Prepdata_test" werden eingelesen und in einer for-Schleife konkateniert. Quelle der Daten ist das ModelNet von Wu et al. [2].

Die Reshape-Funktion bringt die Arrays am Ende in die Form `[Objekt, Punkt, Koordinaten]` und erzeugt einen Vektor für die Labels.

[16]:
```
# load train points and labels
path = os.getcwd()
train_path = os.path.join(path, "Prepdata")
filenames = [d for d in os.listdir(train_path)]
```

```
train_points = None
train_labels = None
for d in filenames:
    cur_points, cur_labels = load_h5(os.path.join(train_path, d))
    cur_points = cur_points.reshape(1, -1, 3)
    cur_labels = cur_labels.reshape(1, -1)
    if train_labels is None or train_points is None:
        train_labels = cur_labels
        train_points = cur_points
    else:
        train_labels = np.hstack((train_labels, cur_labels))
        train_points = np.hstack((train_points, cur_points))
train_points_r = train_points.reshape(-1, num_points, 3)
train_labels_r = train_labels.reshape(-1, 1)
```

[17]:
```
# load test points and labels
test_path = os.path.join(path, "Prepdata_test")
filenames = [d for d in os.listdir(test_path)]

test_points = None
test_labels = None
for d in filenames:
    cur_points, cur_labels = load_h5(os.path.join(test_path, d))
    cur_points = cur_points.reshape(1, -1, 3)
    cur_labels = cur_labels.reshape(1, -1)
    if test_labels is None or test_points is None:
        test_labels = cur_labels
        test_points = cur_points
    else:
        test_labels = np.hstack((test_labels, cur_labels))
        test_points = np.hstack((test_points, cur_points))
test_points_r = test_points.reshape(-1, num_points, 3)
test_labels_r = test_labels.reshape(-1, 1)
```

Bevor die Daten verwendet werden können, müssen die Labels der einzelnen Objekte mithilfe der Funktion `to_categorical()` in eine binäre Klassenmatrix überführt werden. Diese ist notwendig, um die Verlustfunktion `categorical_crossentropy` für den Trainingsprozess zu verwenden.

[18]:
```
# label to categorical
Y_train = np_utils.to_categorical(train_labels_r, k)
Y_test = np_utils.to_categorical(test_labels_r, k)
```

1.6 Trainingsprozess

Mit den vorbereiteten Daten kann nun das breits kompilierte Modell trainiert werden. Hierfür wird eine for-Schleife mit 50 Iterationen genutzt. In jedem Durchlauf wird der Trainingsdatensatz mithilfe der eingangs definierten Hilfsfunktionen minimal transformiert und über die Funktion

`model.fit()` für das Training des Modells verwendet.

Die `batch-size` legt fest, in welchen Chargen die Daten vom Netzwerk verarbeitet bzw. wieviele Elemente pro Update der Gradienten genutzt werden. Darüber hinaus sind bei der `batch normalization` der Bezeichung entsprechend die Elemente einer Charge betroffen.

Der Parameter `epochs` legt die Anzahl der Iterationen fest. Da dies in der übergeordneten Schleife festgelegt ist, um die Trainingsdaten in jeder Iteration zu verändern, liegt der Wert bei 1.

`shuffle` führt dazu, dass die Trainingsdaten vor jeder Epoche in eine zufällige Reihenfolge gebracht werden.

`verbose` legt die Visualisierung fest. Der Wert 1 steht hierbei für die Darstellung eines Fortschrittbalkens.

Alle fünf Iterationen findet zudem eine Evaluation des Modells mit den Testdaten statt.

[19]:
```
# Fit model on training data
for i in range(1,50):
    #model.fit(train_points_r, Y_train, batch_size=32, epochs=1, shuffle=True,␣
 ↪verbose=1)
    # rotate and jitter the points
    train_points_rotate = rotate_point_cloud(train_points_r)
    train_points_jitter = jitter_point_cloud(train_points_rotate)
    model.fit(train_points_jitter, Y_train, batch_size=32, epochs=1,␣
 ↪shuffle=True, verbose=1)
    s = "Current epoch is:" + str(i)
    print(s)
    if i % 5 == 0:
        score = model.evaluate(test_points_r, Y_test, verbose=1)
        print('Test loss: ', score[0])
        print('Test accuracy: ', score[1])
```

```
Epoch 1/1
9840/9840 [==============================] - 135s 14ms/step - loss: 1.8717 -
accuracy: 0.4860
Current epoch is:1
Epoch 1/1
9840/9840 [==============================] - 131s 13ms/step - loss: 1.1586 -
accuracy: 0.6588
Current epoch is:2
Epoch 1/1
9840/9840 [==============================] - 132s 13ms/step - loss: 1.0067 -
accuracy: 0.6953
Current epoch is:3
Epoch 1/1
9840/9840 [==============================] - 131s 13ms/step - loss: 0.9302 -
accuracy: 0.7194
Current epoch is:4
Epoch 1/1
```

```
9840/9840 [==============================] - 131s 13ms/step - loss: 0.8651 -
accuracy: 0.7427
Current epoch is:5
2468/2468 [==============================] - 16s 6ms/step
Test loss:  1.8702477720993265
Test accuracy:  0.49756887555122375
Epoch 1/1
9840/9840 [==============================] - 131s 13ms/step - loss: 0.8173 -
accuracy: 0.7547
Current epoch is:6
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.8008 -
accuracy: 0.7596
Current epoch is:7
Epoch 1/1
9840/9840 [==============================] - 134s 14ms/step - loss: 0.7798 -
accuracy: 0.7616
Current epoch is:8
Epoch 1/1
9840/9840 [==============================] - 131s 13ms/step - loss: 0.7417 -
accuracy: 0.7690
Current epoch is:9
Epoch 1/1
9840/9840 [==============================] - 131s 13ms/step - loss: 0.7414 -
accuracy: 0.7713
Current epoch is:10
2468/2468 [==============================] - 15s 6ms/step
Test loss:  0.9934237548172764
Test accuracy:  0.7050243020057678
Epoch 1/1
9840/9840 [==============================] - 133s 13ms/step - loss: 0.7172 -
accuracy: 0.7771
Current epoch is:11
Epoch 1/1
9840/9840 [==============================] - 133s 13ms/step - loss: 0.6994 -
accuracy: 0.7821
Current epoch is:12
Epoch 1/1
9840/9840 [==============================] - 135s 14ms/step - loss: 0.6764 -
accuracy: 0.7905
Current epoch is:13
Epoch 1/1
9840/9840 [==============================] - 135s 14ms/step - loss: 0.6698 -
accuracy: 0.7908
Current epoch is:14
Epoch 1/1
9840/9840 [==============================] - 133s 13ms/step - loss: 0.6840 -
accuracy: 0.7868
```

```
Current epoch is:15
2468/2468 [==============================] - 15s 6ms/step
Test loss:  0.8143643490499191
Test accuracy:  0.7576985359191895
Epoch 1/1
9840/9840 [==============================] - 133s 13ms/step - loss: 0.6580 -
accuracy: 0.7962
Current epoch is:16
Epoch 1/1
9840/9840 [==============================] - 132s 13ms/step - loss: 0.6478 -
accuracy: 0.7973
Current epoch is:17
Epoch 1/1
9840/9840 [==============================] - 131s 13ms/step - loss: 0.6324 -
accuracy: 0.7997
Current epoch is:18
Epoch 1/1
9840/9840 [==============================] - 133s 13ms/step - loss: 0.6298 -
accuracy: 0.8027
Current epoch is:19
Epoch 1/1
9840/9840 [==============================] - 135s 14ms/step - loss: 0.6384 -
accuracy: 0.8008
Current epoch is:20
2468/2468 [==============================] - 16s 6ms/step
Test loss:  0.7762728981685793
Test accuracy:  0.7564829587936401
Epoch 1/1
9840/9840 [==============================] - 135s 14ms/step - loss: 0.6166 -
accuracy: 0.8027
Current epoch is:21
Epoch 1/1
9840/9840 [==============================] - 132s 13ms/step - loss: 0.6151 -
accuracy: 0.8046
Current epoch is:22
Epoch 1/1
9840/9840 [==============================] - 134s 14ms/step - loss: 0.5949 -
accuracy: 0.8118
Current epoch is:23
Epoch 1/1
9840/9840 [==============================] - 132s 13ms/step - loss: 0.6023 -
accuracy: 0.8078
Current epoch is:24
Epoch 1/1
9840/9840 [==============================] - 130s 13ms/step - loss: 0.5996 -
accuracy: 0.8091
Current epoch is:25
2468/2468 [==============================] - 15s 6ms/step
```

```
Test loss:  0.908817028496988
Test accuracy:  0.752836287021637
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.5964 -
accuracy: 0.8117
Current epoch is:26
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.5942 -
accuracy: 0.8134
Current epoch is:27
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.5785 -
accuracy: 0.8150
Current epoch is:28
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5738 -
accuracy: 0.8174
Current epoch is:29
Epoch 1/1
9840/9840 [==============================] - 131s 13ms/step - loss: 0.5677 -
accuracy: 0.8223
Current epoch is:30
2468/2468 [==============================] - 15s 6ms/step
Test loss:  1.1533760912414501
Test accuracy:  0.6632901430130005
Epoch 1/1
9840/9840 [==============================] - 132s 13ms/step - loss: 0.5534 -
accuracy: 0.8246
Current epoch is:31
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5471 -
accuracy: 0.8272
Current epoch is:32
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5399 -
accuracy: 0.8251
Current epoch is:33
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5383 -
accuracy: 0.8269
Current epoch is:34
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5323 -
accuracy: 0.8243
Current epoch is:35
2468/2468 [==============================] - 16s 6ms/step
Test loss:  0.7561112018226611
Test accuracy:  0.7718800902366638
```

```
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5479 -
accuracy: 0.8255
Current epoch is:36
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5261 -
accuracy: 0.8274
Current epoch is:37
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.5119 -
accuracy: 0.8342
Current epoch is:38
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5366 -
accuracy: 0.8249
Current epoch is:39
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5204 -
accuracy: 0.8361
Current epoch is:40
2468/2468 [==============================] - 15s 6ms/step
Test loss:  0.8826023187590959
Test accuracy:  0.7297406792640686
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5275 -
accuracy: 0.8266
Current epoch is:41
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5129 -
accuracy: 0.8349
Current epoch is:42
Epoch 1/1
9840/9840 [==============================] - 129s 13ms/step - loss: 0.5166 -
accuracy: 0.8352
Current epoch is:43
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.4954 -
accuracy: 0.8379
Current epoch is:44
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.4927 -
accuracy: 0.8427
Current epoch is:45
2468/2468 [==============================] - 15s 6ms/step
Test loss:  0.8047663408121761
Test accuracy:  0.765802264213562
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.5115 -
```

```
accuracy: 0.8347
Current epoch is:46
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.5058 -
accuracy: 0.8356
Current epoch is:47
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.4867 -
accuracy: 0.8415
Current epoch is:48
Epoch 1/1
9840/9840 [==============================] - 128s 13ms/step - loss: 0.4836 -
accuracy: 0.8449
Current epoch is:49
```

Nach abgeschlossenem Trainingsprozess findet eine letzte Evaluation des Netzwerks statt. Die Ergebnisse der Verlustfunktion und Genauigkeit werden ausgegeben.

[20]:
```
# score the model
score = model.evaluate(test_points_r, Y_test, verbose=1)
print('Test loss: ', score[0])
print('Test accuracy: ', score[1])
```

```
2468/2468 [==============================] - 15s 6ms/step
Test loss:  0.6230143568504185
Test accuracy:  0.8176661133766174
```

Quellen:

[1] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," CoRR, vol. abs/1612.00593, 2016, [Online]. Available: http://arxiv.org/abs/1612.00593.

[2] Z. Wu et al., "3d shapenets: A deep representation for volumetric shapes," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2015, pp. 1912–1920. [Online]. Available: http://modelnet.cs.princeton.edu/

[3] L. Gonzales, "An In-Depth Look at PointNet," Apr. 2019, Accessed: Jun. 02, 2020. [Online]. Available: https://medium.com/@luis_gonzales/an-in-depth-look-at-pointnet-111d7efdaa1a.

PointNet-Segmentierungsnetzwerk

June 15, 2020

[Inhalt]

1 PointNet-Segmentierungsnetzwerk

Aufbauend auf dem vorherigen Notebook zum PointNet-Klassifizierungsnetzwerk widmet sich dieses Dokument einem praktischen Beispiel für die Segmentierung von 3D-Punktwolken mithilfe von PointNet. Der verwendete Code entstammt einer Implementation von G. Li, zu finden im folgenden Git-Repository: https://github.com/garyli1019/pointnet-keras.

Segmentierung ist eine besondere Form der Klassifizierung, bei der die Elemente eines Objektes spezifischen Teilstücken oder -Bereichen des Gesamtobjekts zugeordnet werden. Ein klassisches Beispiel für Segmente in einem Objekt sind die Beine, Sitzfläche und die Lehne eines Stuhls. Das segmentierte Objekt könnte etwa so aussehen wie in der folgenden Grafik:

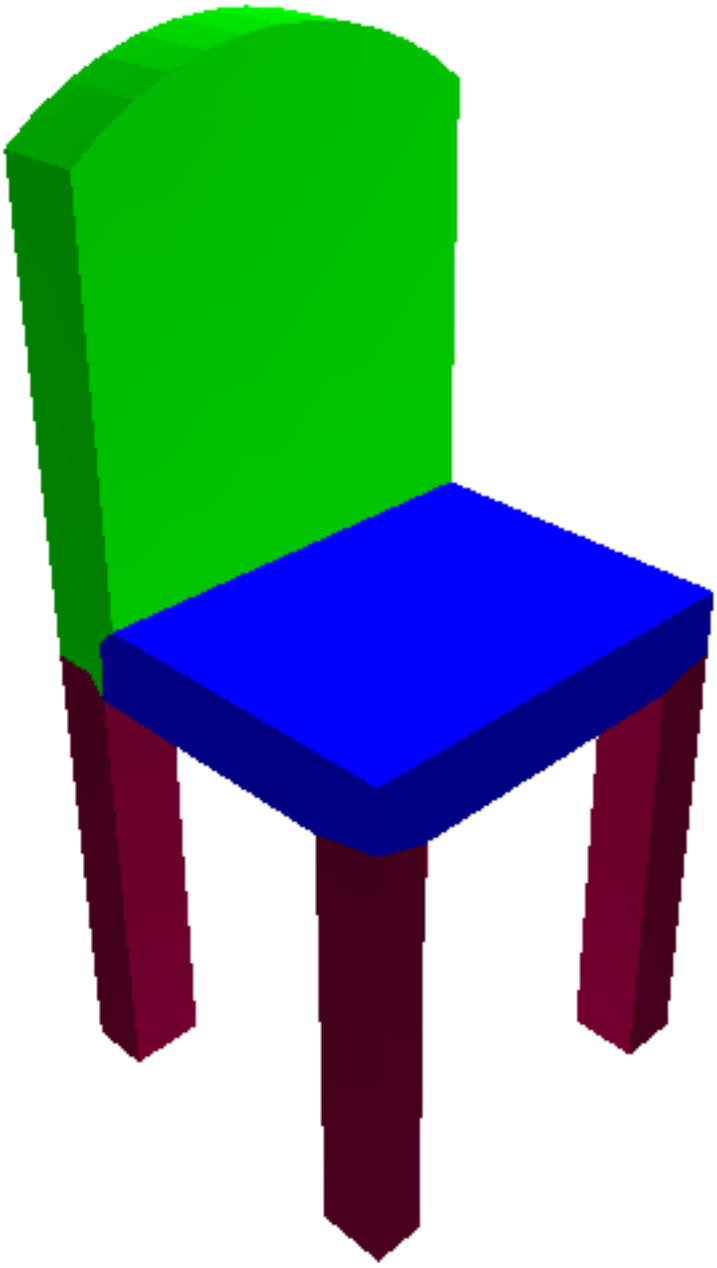


Abbildung 1: Segmente eines Stuhls [1]

Die folgende Grafik von Qi et al. [1] zeigt die Architektur von PointNet. Der markierte Bereich stellt hierbei das eigentliche Segmentierungsnetzwerk dar. Wie unschwer zu erkennen ist, wird die Architektur vom Input bis zur Aggregation der globalen Feature-Signatur gleichermaßen vom Klassifikations- und vom Segmentierungsnetzwerk genutzt.

Diese Darstellung wird wie auch im vorherigen Notebook im weiteren Verlauf wiederholt genutzt, um das Durchlaufen der Pipeline zu visualisieren.

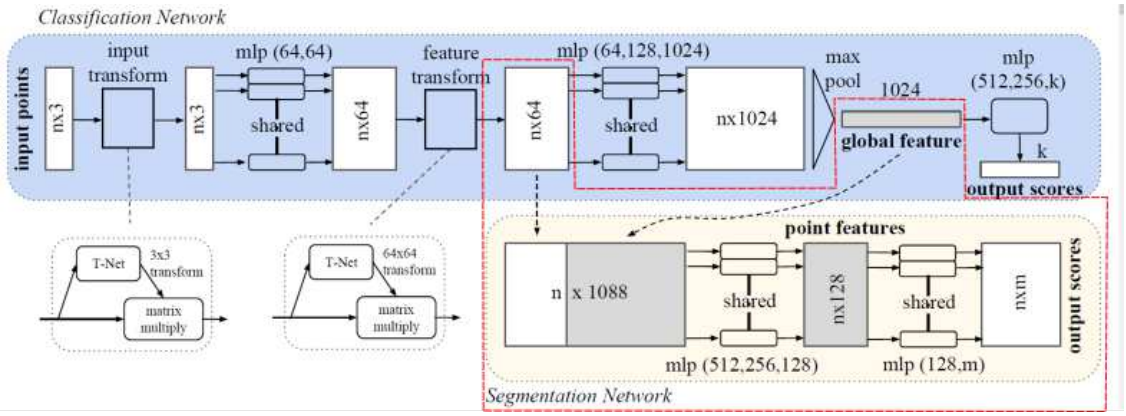


Abbildung 2: PointNet-Segmentierungsnetzwerk [2]

1.1 Anwendungshinweis

Da es sich bei dem Code in diesem Notebook um ein zusammenhängendes Script handelt, empfiehlt sich bei Änderungen stets das Zurücksetzen des Kernels. Hierfür wird unter dem Reiter "Kernel" die Option "Restart & Run all" ausgewählt.

1.2 Imports

Für die hier analyiserte Keras-Implementation der PointNet-Architektur werden drei Pakete benötigt: * NumPy für die Verwaltung von Datenstrukturen sowie der Handhabung von Vektoren und Matrizen * H5Py als Schnittstelle für das HDF5-Datenformat * Tensorflow für die Verwendung der Keras-API, welche Teil des Tensorflow-Cores ist

Darüber hinaus werden die nötigen Keras-Pakete für die Schichttypen und die Erzeugung des Modells importiert.

Für die Darstellung einer segmentierten Punktwolke werden pyplot und Axes3D genutzt.

[1]:
```
import warnings
warnings.simplefilter(action='ignore')

import numpy as np
import os
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import tensorflow as tf
from keras import optimizers
from keras.layers import Input
from keras.models import Model
from keras.layers import Dense, Reshape
from keras.layers import Convolution1D, MaxPooling1D, BatchNormalization
from keras.layers import Lambda, concatenate
#from keras.utils import np_utils
import h5py
```

```
tf.logging.set_verbosity(tf.logging.ERROR)
```

```
Using TensorFlow backend.
```

1.3 Hilfsfunktionen

Nachfolgend werden Hilfsfunktionen definiert. Wie beim Klassifizierungsnetzwerk werden die Funktionen `mat_mul` und `load_5` für die Matrizenmultiplikation und das Laden von HPF5-Files benötigt.

Die Funktion `exp_dim` wird im Modell im Rahmen einer Lambda Layer genutzt, um den globalen Feature-Vektor für die nachfolgende Konkatenation zu erweitern.

[2]:
```
def mat_mul(A, B):
    return tf.matmul(A, B)
```

[3]:
```
def exp_dim(global_feature, num_points):
    return tf.tile(global_feature, [1, num_points, 1])
```

[4]:
```
def load_h5(h5_filename):
    f = h5py.File(h5_filename)
    data = f['data'][:]
    label = f['label'][:]
    return (data, label)
```

Die Funktionen rotate_point_cloud sowie jitter_point_cloud dienen dazu, die Trainingsdaten wiederholt nutzbar zu machen. Durch zufällige Rotationen und Mikro-Translationen der Objekte bzw. Punkte erzeugen die Funktionen aus einem Datensatz eine beliebige Menge unterschiedlicher Trainingsdaten.

rotate_point_cloud erzeugt zunächst eine Nullmatrix mit den Dimensionen des Inputs. Anschließend wird für jedes Objekt eine zufällige Rotationsmatrix erzeugt. Die Produkte der Rotationsmatrix und der Objektpunkte werden in die Nullmatrix eingefügt, welche nach Durchlaufen aller Objekte zurückgegeben wird. Die Rotationsachse ist die Y-Achse.

jitter_point_cloud erzeugt ebenfalls eine Kopie des Inputs. Diese wird jedoch nicht als Nullmatrix initialisiert, sondern mit kleinen Zufallszahlen gefüllt. Anschließend erfolgt eine elementweise Addition mit der Input-Matrix.

[5]:
```
def rotate_point_cloud(batch_data):
    """ Randomly rotate the point clouds to augument the dataset
        rotation is per shape based along up direction
        Input:
          BxNx3 array, original batch of point clouds
        Return:
          BxNx3 array, rotated batch of point clouds
    """
    rotated_data = np.zeros(batch_data.shape, dtype=np.float32)
    for k in range(batch_data.shape[0]):
```

```
        rotation_angle = np.random.uniform() * 2 * np.pi
        cosval = np.cos(rotation_angle)
        sinval = np.sin(rotation_angle)
        rotation_matrix = np.array([[cosval, 0, sinval],
                                    [0, 1, 0],
                                    [-sinval, 0, cosval]])
        shape_pc = batch_data[k, ...]
        rotated_data[k, ...] = np.matmul(shape_pc.reshape((-1, 3)),␣
 ↪rotation_matrix)
    return rotated_data
```

[6]:
```
def jitter_point_cloud(batch_data, sigma=0.01, clip=0.05):
    """ Randomly jitter points. jittering is per point.
        Input:
          BxNx3 array, original batch of point clouds
        Return:
          BxNx3 array, jittered batch of point clouds
    """
    B, N, C = batch_data.shape
    assert(clip > 0)
    jittered_data = np.clip(sigma * np.random.randn(B, N, C), -1 * clip, clip)
    jittered_data += batch_data
    return jittered_data
```

Um das Netzwerk erfolgreich trainieren zu können, muss der Input auf einen Objekttypen wie bspw. Stühle festgelegt werden. Hierfür dient die Variable `s`. Alle verfügbaren Typen sind im Dictionary `sList` in der darauffolgenden Codelzelle zu finden.

[7]:
```
# chosen shape
s = 'Guitar'
```

Die folgenden globalen Variablen bestimmen wichtige Randbedingungen der verwendeten Datensätze sowie den verwendenten Optimierungsalgorithmus und sollten daher nicht verändert werden.

[8]:
```
# number of points in each sample
num_points = 1024

# shapes and categories
sList = {'Airplane': 4, 'Chair': 4, 'Guitar': 3, 'Knife': 2, 'Lamp': 3, 'Table':␣
 ↪2}

# number of categories
k = sList[s]
# define optimizer
adam = optimizers.Adam(lr=0.001, decay=0.7)
```

1.4 Keras-Modell

Wie auch die PointNet-Variante zur Klassifizierung wird das Segmentierungsnetzwerk als Keras-Modell implementiert. Der Aufbau ist vom Input bis zur Aggregation der globalen Feature-Signatur identisch, daher wird im Weiteren nur kurz darauf eingegangen.

Das Hauptaugenmerk liegt auf den letzten Teil des Netzwerks, welcher für die eigentliche Segmentierung genutzt wird.

Zu Beginn des Netzwerks findet die bereits bekannte Input-Transformation statt. Hierbei wird eine Transformationsmatrix bestimmt, mit welcher die Elemente einer eingegebenen Punktwolke räumlich ausgerichtet werden.

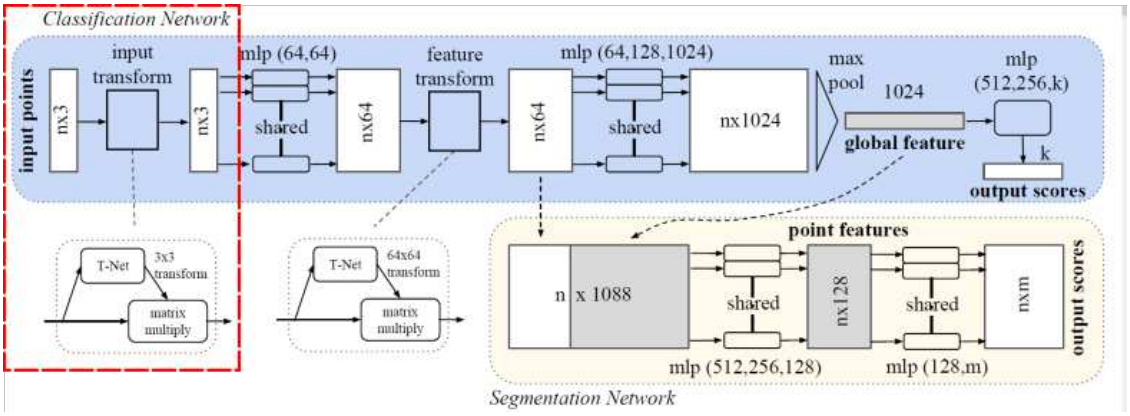


Abbildung 3: Input-Transformation [2]

[9]:
```
# ---------------------------------- Pointnet Architecture
# input_Transformation_net
input_points = Input(shape=(num_points, 3))
x = Convolution1D(64, 1, activation='relu',
                  input_shape=(num_points, 3))(input_points)
x = BatchNormalization()(x)
x = Convolution1D(128, 1, activation='relu')(x)
x = BatchNormalization()(x)
x = Convolution1D(1024, 1, activation='relu')(x)
x = BatchNormalization()(x)
x = MaxPooling1D(pool_size=num_points)(x)
x = Dense(512, activation='relu')(x)
x = BatchNormalization()(x)
x = Dense(256, activation='relu')(x)
x = BatchNormalization()(x)
x = Dense(9, weights=[np.zeros([256, 9]), np.array([1, 0, 0, 0, 1, 0, 0, 0, 1]).
 ↪astype(np.float32)])(x)
input_T = Reshape((3, 3))(x)
```

Im zweiten Schritt findet das bereits bekannte Feature-Building statt. Zur Erinnerung: Hierbei werden die Raumkoordinaten eines jeden Punktes auf einen höherdimensionalen Raum abgebildet, um vom Netzwerk analysierbare Features zu gewinnen. Diese Form der Abstraktion ermöglicht es, beliebige Punktwolken mit PointNet zu verarbeiten, ohne dass diese vorher in eine

spezifische Form konvertiert werden müssen.

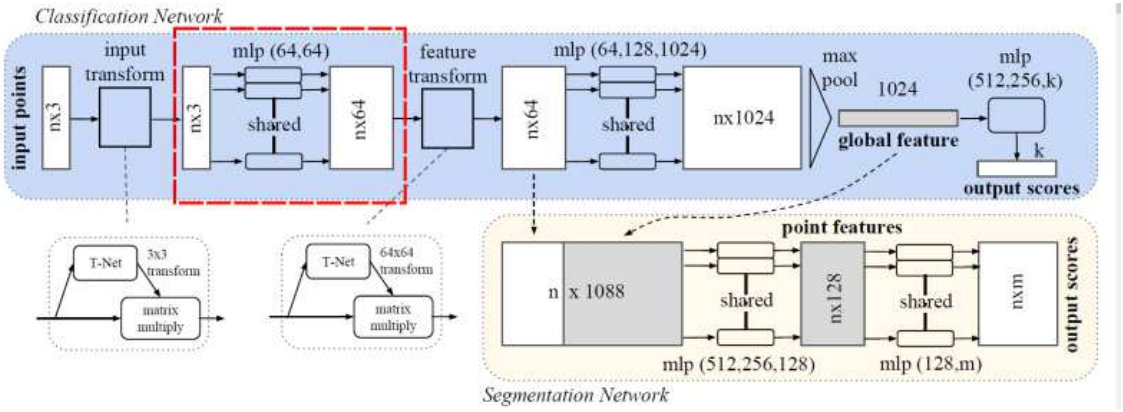


Abbildung 4: Feature Building 1 [2]

[10]:
```
# forward net
g = Lambda(mat_mul, arguments={'B': input_T})(input_points)
g = Convolution1D(64, 1, input_shape=(num_points, 3), activation='relu')(g)
g = BatchNormalization()(g)
g = Convolution1D(64, 1, input_shape=(num_points, 3), activation='relu')(g)
g = BatchNormalization()(g)
```

Auch die Feature-Transformation findet in bereits bekannter Weise statt. Die Ausrichtung der Feature-Vektoren im $\mathbb{R}^{64}$ dient ebenso wie die räumliche Ausrichtung im dreidimensionalen Raum dazu, etwaige Transformationen des Inputs wie bspw. Rotationen zu kompensieren.

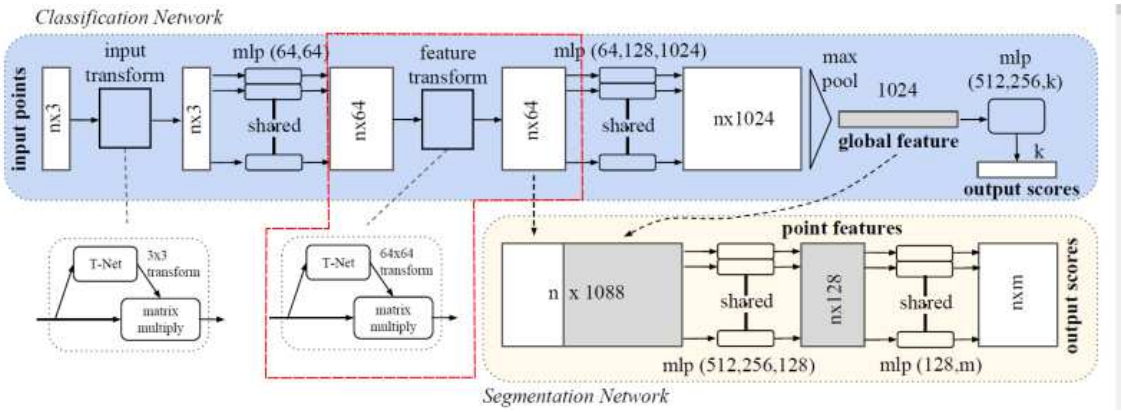


Abbildung 5: Feature-Transformation [2]

[11]:
```
# feature transformation net
f = Convolution1D(64, 1, activation='relu')(g)
f = BatchNormalization()(f)
f = Convolution1D(128, 1, activation='relu')(f)
f = BatchNormalization()(f)
f = Convolution1D(1024, 1, activation='relu')(f)
f = BatchNormalization()(f)
f = MaxPooling1D(pool_size=num_points)(f)
```

```
f = Dense(512, activation='relu')(f)
f = BatchNormalization()(f)
f = Dense(256, activation='relu')(f)
f = BatchNormalization()(f)
f = Dense(64 * 64, weights=[np.zeros([256, 64 * 64]), np.eye(64).flatten().
 ↪astype(np.float32)])(f)
feature_T = Reshape((64, 64))(f)
```

Im nachfolgenden Schritt findet sich der erste Unterschied zwischen Klassifizierung und Segmentierung. Nach Anwendung der Feature-Transformationsmatrix durch eine Lambda Layer wird der Zwischenstand in der Variablen `seg_part1` zwischengespeichert. Anschließend werden die ausgerichteten Feature-Vektoren über Faltungsschichten auf den $\mathbb{R}^{1024}$ abgebildet.

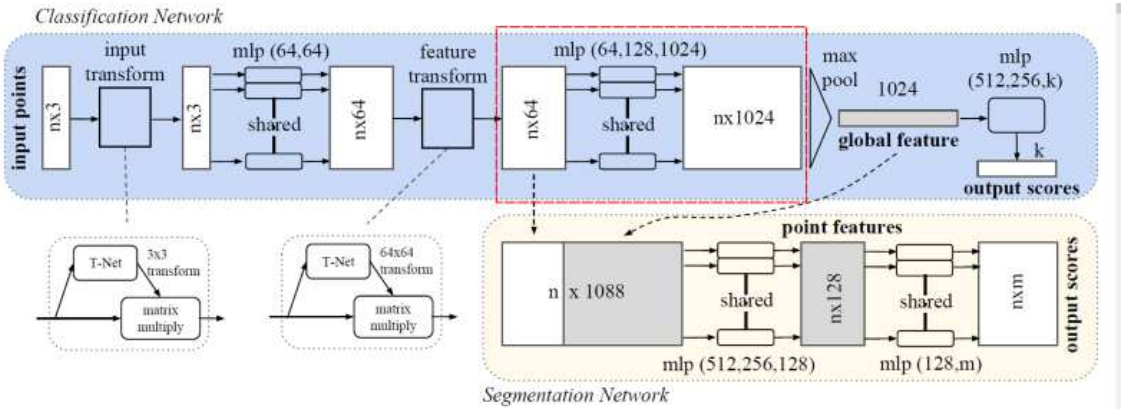


Abbildung 6: Feature Building 2 [2]

[12]:
```
# forward net
g = Lambda(mat_mul, arguments={'B': feature_T})(g)
seg_part1 = g
g = Convolution1D(64, 1, activation='relu')(g)
g = BatchNormalization()(g)
g = Convolution1D(128, 1, activation='relu')(g)
g = BatchNormalization()(g)
g = Convolution1D(1024, 1, activation='relu')(g)
g = BatchNormalization()(g)
```

Der durch Max-Pooling aggregierte Feature-Vektor wird über eine Lambda Layer und der eingangs definierten Hilfsfunktion `exp_dim` erweitert. Hierbei werden die 1024 Werte des Vektors für jedes Element einer Punktwolke dupliziert. Im gegebenen Beispiel wird also der Feature-Vektor von 1x1024 auf 1024x1024 erweitert, wobei jede Zeile eine Kopie der globalen Signatur enthält.

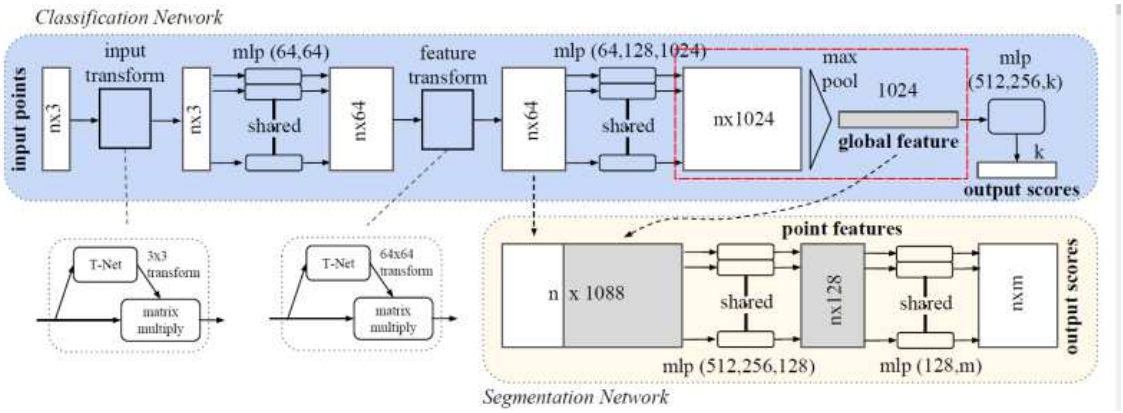


Abbildung 7: Max-Pooling [2]

```
[13]: # global_feature
      global_feature = MaxPooling1D(pool_size=num_points)(g)
      global_feature = Lambda(exp_dim, arguments={'num_points':␣
       ↪num_points})(global_feature)
```

Im Anschluss an die Aggregation der globalen Feature-Signatur beginnt das eigentliche Segmentierungsnetzwerk. Im ersten Schritt, welcher in Abbbildung 8 markiert ist, findet eine Konkatenation der zuvor zwischengespeicherten lokalen Features und der globalen Features statt. Durch die nach dem Max-Pooling durchgeführte Erweiterung des globalen Feature-Vektors kann dies ohne Umstände mit der Funktion `concatenate` erreicht werden.

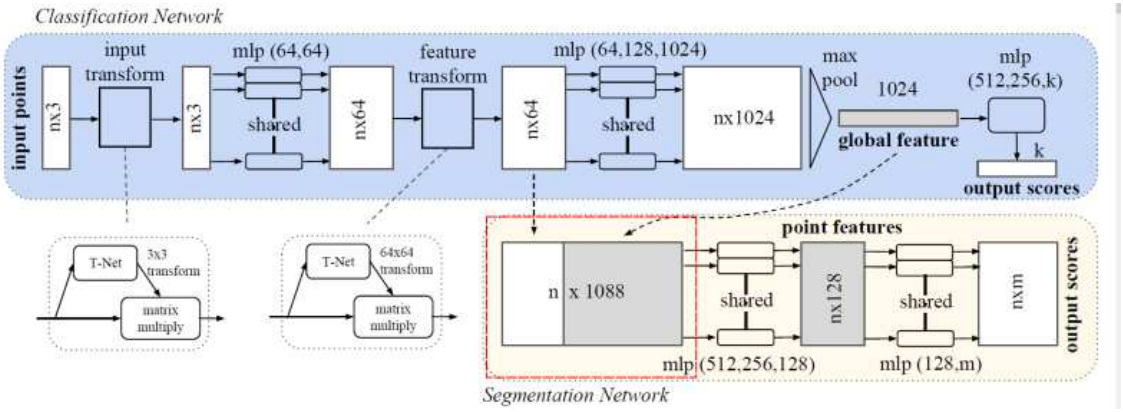


Abbildung 8: Konkatenation lokaler und globaler Features [2]

```
[14]: # point_net_seg
      c = concatenate([seg_part1, global_feature])
```

Mit dem konkatenierten Feature-Vektor wird jedes Element einer Punktwolke durch 1088 Werte beschrieben. Die Kombination von Informationen über lokale Geometrie und globale Semantik ermöglicht dem Netzwerk, mithilfe eines MLP eine akkurate Vorhersage über die Zugehörigkeit der einzelnen Punkte zu den verfügbaren Objektteilen zu treffen. Dieser Schritt wird in Abbildung 9 markiert dargestellt. Der MLP wird mit insgesamt fünf Faltungsschichten mit Filtern der Größe 1 realisiert. Zwischen den Schichten findet wie gewohnt `batch normalization` statt. Die ersten vier Schichten nutzen ReLU als Aktivierungsfunktion, während die letzte mittels Softmax die finalen

Segmentierungswerte für jeden Punkt als Output ermittelt.

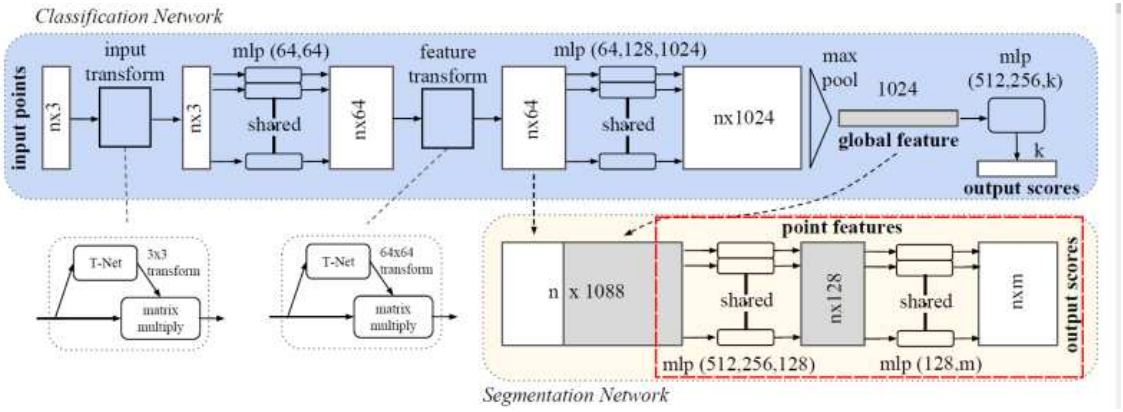


Abbildung 9: Segmentierung [2]

[15]:
```
c = Convolution1D(512, 1, activation='relu')(c)
c = BatchNormalization()(c)
c = Convolution1D(256, 1, activation='relu')(c)
c = BatchNormalization()(c)
c = Convolution1D(128, 1, activation='relu')(c)
c = BatchNormalization()(c)
c = Convolution1D(128, 1, activation='relu')(c)
c = BatchNormalization()(c)
prediction = Convolution1D(k, 1, activation='softmax')(c)
# ----------------------------------------------------end of pointnet
```

Wie im vorherigen Notebook wird nach der Vervollständigung des Netzwerks mittels verknüpftem Input und Output das entsprechende Keras-Modell gebildet und dessen Struktur tabellarisch ausgegeben. Anschließend wird das Modell kompiliert, wobei auch in diesem Beispiel der Optimizer `Adam` und die Verlustfunktion `categorical_crossentropy`genutzt werden.

[16]:
```
# define model
model = Model(inputs=input_points, outputs=prediction)
print(model.summary())
```

```
Model: "model_1"
_________________________________________________________________________________
__________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================
==================
input_1 (InputLayer)         (None, 1024, 3)           0
_________________________________________________________________________________
__________________
lambda_1 (Lambda)            (None, 1024, 3)           0           input_1[0][0]
_________________________________________________________________________________
__________________
```

```
conv1d_4 (Conv1D)               (None, 1024, 64)     256         lambda_1[0][0]
__________________________________________________________________________________
__________________
batch_normalization_6 (BatchNor (None, 1024, 64)     256         conv1d_4[0][0]
__________________________________________________________________________________
__________________
conv1d_5 (Conv1D)               (None, 1024, 64)     4160
batch_normalization_6[0][0]
__________________________________________________________________________________
__________________
batch_normalization_7 (BatchNor (None, 1024, 64)     256         conv1d_5[0][0]
__________________________________________________________________________________
__________________
lambda_2 (Lambda)               (None, 1024, 64)     0
batch_normalization_7[0][0]
__________________________________________________________________________________
__________________
conv1d_9 (Conv1D)               (None, 1024, 64)     4160        lambda_2[0][0]
__________________________________________________________________________________
__________________
batch_normalization_13 (BatchNo (None, 1024, 64)     256         conv1d_9[0][0]
__________________________________________________________________________________
__________________
conv1d_10 (Conv1D)              (None, 1024, 128)    8320
batch_normalization_13[0][0]
__________________________________________________________________________________
__________________
batch_normalization_14 (BatchNo (None, 1024, 128)    512         conv1d_10[0][0]
__________________________________________________________________________________
__________________
conv1d_11 (Conv1D)              (None, 1024, 1024)   132096
batch_normalization_14[0][0]
__________________________________________________________________________________
__________________
batch_normalization_15 (BatchNo (None, 1024, 1024)   4096        conv1d_11[0][0]
__________________________________________________________________________________
__________________
max_pooling1d_3 (MaxPooling1D)  (None, 1, 1024)      0
batch_normalization_15[0][0]
__________________________________________________________________________________
__________________
lambda_3 (Lambda)               (None, 1024, 1024)   0
max_pooling1d_3[0][0]
__________________________________________________________________________________
__________________
concatenate_1 (Concatenate)     (None, 1024, 1088)   0           lambda_2[0][0]
                                                                 lambda_3[0][0]
__________________________________________________________________________________
```

```
------------------
conv1d_12 (Conv1D)              (None, 1024, 512)    557568
concatenate_1[0][0]
__________________________________________________________________________________
------------------
batch_normalization_16 (BatchNo (None, 1024, 512)    2048        conv1d_12[0][0]
__________________________________________________________________________________
------------------
conv1d_13 (Conv1D)              (None, 1024, 256)    131328
batch_normalization_16[0][0]
__________________________________________________________________________________
------------------
batch_normalization_17 (BatchNo (None, 1024, 256)    1024        conv1d_13[0][0]
__________________________________________________________________________________
------------------
conv1d_14 (Conv1D)              (None, 1024, 128)    32896
batch_normalization_17[0][0]
__________________________________________________________________________________
------------------
batch_normalization_18 (BatchNo (None, 1024, 128)    512         conv1d_14[0][0]
__________________________________________________________________________________
------------------
conv1d_15 (Conv1D)              (None, 1024, 128)    16512
batch_normalization_18[0][0]
__________________________________________________________________________________
------------------
batch_normalization_19 (BatchNo (None, 1024, 128)    512         conv1d_15[0][0]
__________________________________________________________________________________
------------------
conv1d_16 (Conv1D)              (None, 1024, 3)      387
batch_normalization_19[0][0]
==================================================================================
==================
Total params: 897,155
Trainable params: 892,419
Non-trainable params: 4,736
__________________________________________________________________________________
------------------
None
```

[17]:
```
# compile classification model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

1.5 Vorbereitung der Daten

Der Ladeprozess der Trainings- und Testdaten gestaltet sich ähnlich wie im Klassifikationsbeispiel. Die in der globalen Variable `s` festgelegte Objektklasse wird genutzt, um die korrekten HPF5-Files aus den Ordnern "Seg_Prep" bzw. "Seg_Prep_test" auszuwählen. Diese werden anschließend über die Hilfsfunktion `load_h5` eingelesen. Die Daten über Punkte und Labels werden in entsprechend benannte Variablen zwischengespeichert und via `reshape` in die Formen `[Objekt, Punkt, Koordinaten]` bzw. `[Objekt, Punkt, Klasse]` gebracht.

[18]:
```
# load TRAIN points and labels
path = os.getcwd()
train_path = os.path.join(path, "Seg_Prep")
filename = s + ".h5"

train_points = None
train_labels = None
cur_points, cur_labels = load_h5(os.path.join(train_path, filename))

train_labels = cur_labels
train_points = cur_points

train_points_r = train_points.reshape(-1, num_points, 3)
train_labels_r = train_labels.reshape(-1, num_points, k)
```

[19]:
```
# load TEST points and labels
test_path = os.path.join(path, "Seg_Prep_test")
filename = s + ".h5"

test_points = None
test_labels = None
cur_points, cur_labels = load_h5(os.path.join(test_path, filename))

test_labels = cur_labels
test_points = cur_points

test_points_r = test_points.reshape(-1, num_points, 3)
test_labels_r = test_labels.reshape(-1, num_points, k)
```

1.6 Trainingsprozess

Das Training des Segmentierungsnetzwerks erfolgt in derselben Weise wie im vorherigen Notebook. Über eine for-Schleife wird eine beliebige Anzahl Iterationen realisiert, in denen die mittels der Funktionen `train_points_rotate` und `train_points_jitter` minimal unterschiedliche Versionen der Trainingsdaten über `model.fit` an das Netzwerk übergeben werden.

Die `batch-size` legt fest, in welchen Chargen die Daten vom Netzwerk verarbeitet bzw. wieviele Elemente pro Update der Gradienten genutzt werden. Darüber hinaus sind bei der `batch normalization` der Bezeichung entsprechend die Elemente einer Charge betroffen.

Der Parameter `epochs` legt die Anzahl der Iterationen fest. Da dies in der übergeordneten Schleife festgelegt ist, um die Trainingsdaten in jeder Iteration zu verändern, liegt der Wert bei 1.

`shuffle` führt dazu, dass die Trainingsdaten vor jeder Epoche in eine zufällige Reihenfolge gebracht werden.

`verbose` legt die Visualisierung fest. Der Wert 1 steht hierbei für die Darstellung eines Fortschrittbalkens.

Wie im vorherigen Notebook findet in jeder fünften Iteration eine Evaluation des Netzwerks mit den Testdaten statt.

[20]:
```
# train model
for i in range(1, 51):
    # rotate and jitter point cloud every epoch
    train_points_rotate = rotate_point_cloud(train_points_r)
    train_points_jitter = jitter_point_cloud(train_points_rotate)
    model.fit(train_points_jitter, train_labels_r, batch_size=32, epochs=1,␣
 ↪shuffle=True, verbose=1)
    e = "Current epoch is:" + str(i)
    print(e)
    # evaluate model
    if i % 5 == 0:
        score = model.evaluate(test_points_r, test_labels_r, verbose=1)
        print('Test loss: ', score[0])
        print('Test accuracy: ', score[1])
```

```
Epoch 1/1
631/631 [==============================] - 6s 10ms/step - loss: 0.4772 -
accuracy: 0.8190
Current epoch is:1
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1835 -
accuracy: 0.9354
Current epoch is:2
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1441 -
accuracy: 0.9476
Current epoch is:3
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1345 -
accuracy: 0.9502
Current epoch is:4
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1322 -
accuracy: 0.9511
Current epoch is:5
158/158 [==============================] - 1s 4ms/step
Test loss:  2.8031916950322406
```

```
Test accuracy:  0.0874270647764206
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1300 -
accuracy: 0.9517
Current epoch is:6
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1295 -
accuracy: 0.9509
Current epoch is:7
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1244 -
accuracy: 0.9522
Current epoch is:8
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1261 -
accuracy: 0.9512
Current epoch is:9
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1234 -
accuracy: 0.9520
Current epoch is:10
158/158 [==============================] - 0s 2ms/step
Test loss:  5.0158094394056105
Test accuracy:  0.0874270647764206
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1187 -
accuracy: 0.9549
Current epoch is:11
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1155 -
accuracy: 0.9552
Current epoch is:12
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1169 -
accuracy: 0.9544
Current epoch is:13
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1154 -
accuracy: 0.9551
Current epoch is:14
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1162 -
accuracy: 0.9548
Current epoch is:15
158/158 [==============================] - 0s 2ms/step
Test loss:  7.235829504230354
Test accuracy:  0.0874270647764206
Epoch 1/1
```

```
631/631 [==============================] - 4s 6ms/step - loss: 0.1146 -
accuracy: 0.9553
Current epoch is:16
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1117 -
accuracy: 0.9562
Current epoch is:17
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1101 -
accuracy: 0.9569
Current epoch is:18
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1077 -
accuracy: 0.9576
Current epoch is:19
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1091 -
accuracy: 0.9571
Current epoch is:20
158/158 [==============================] - 0s 2ms/step
Test loss:  1.4206434337398675
Test accuracy:  0.742502748966217
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1078 -
accuracy: 0.9577
Current epoch is:21
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1092 -
accuracy: 0.9570
Current epoch is:22
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1083 -
accuracy: 0.9574
Current epoch is:23
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1073 -
accuracy: 0.9575
Current epoch is:24
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1063 -
accuracy: 0.9576
Current epoch is:25
158/158 [==============================] - 0s 2ms/step
Test loss:  1.870902877819689
Test accuracy:  0.7081932425498962
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1050 -
accuracy: 0.9583
```

```
Current epoch is:26
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1051 -
accuracy: 0.9583
Current epoch is:27
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1070 -
accuracy: 0.9570
Current epoch is:28
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1050 -
accuracy: 0.9585
Current epoch is:29
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1014 -
accuracy: 0.9601
Current epoch is:30
158/158 [==============================] - 0s 2ms/step
Test loss:  0.8824337901948374
Test accuracy:  0.7382874488830566
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1012 -
accuracy: 0.9599
Current epoch is:31
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1012 -
accuracy: 0.9601
Current epoch is:32
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1023 -
accuracy: 0.9591
Current epoch is:33
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1041 -
accuracy: 0.9579
Current epoch is:34
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1041 -
accuracy: 0.9577
Current epoch is:35
158/158 [==============================] - 0s 2ms/step
Test loss:  0.24156014470359946
Test accuracy:  0.9123071432113647
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0984 -
accuracy: 0.9608
Current epoch is:36
Epoch 1/1
```

```
631/631 [==============================] - 4s 6ms/step - loss: 0.0986 -
accuracy: 0.9607
Current epoch is:37
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0999 -
accuracy: 0.9602
Current epoch is:38
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0949 -
accuracy: 0.9625
Current epoch is:39
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0984 -
accuracy: 0.9605
Current epoch is:40
158/158 [==============================] - 0s 2ms/step
Test loss:  0.1557120973173576
Test accuracy:  0.944508969783783
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.1005 -
accuracy: 0.9592
Current epoch is:41
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0975 -
accuracy: 0.9612
Current epoch is:42
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0952 -
accuracy: 0.9621
Current epoch is:43
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0925 -
accuracy: 0.9630
Current epoch is:44
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0938 -
accuracy: 0.9629
Current epoch is:45
158/158 [==============================] - 0s 2ms/step
Test loss:  0.13900183546769468
Test accuracy:  0.9471296668052673
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0928 -
accuracy: 0.9629
Current epoch is:46
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0952 -
accuracy: 0.9619
```

```
Current epoch is:47
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0943 -
accuracy: 0.9625
Current epoch is:48
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0940 -
accuracy: 0.9625
Current epoch is:49
Epoch 1/1
631/631 [==============================] - 4s 6ms/step - loss: 0.0938 -
accuracy: 0.9622
Current epoch is:50
158/158 [==============================] - 0s 2ms/step
Test loss:  0.13251101866930345
Test accuracy:  0.9516910314559937
```

1.7 Visualisierung

Um die Effektivität des Netzwerks visuell nachvollziehen zu können, wird in den nachfolgenden Code-Zellen ein 3D-Plot eines beliebigen Datensatzes erzeugt.

An dieser Stelle ein wichtiger Hinweis: Um verschiedene Objekte zu visualisieren, muss das Netzwerk nicht jedes Mal von Neuem trainiert werden. Es ist ausreichend, lediglich die nachfolgende Codezelle via "Run"-Befehl auszuführen.

Hierfür wird zunächst ein `figure()`-Objekt des Pakets `matplotlib` erzeugt und mit einem 3D-Subplot gefüllt. Die Achsenbegrenzungen und die Größe der Darstellung können nach Belieben angepasst werden.

Die Arrays `color` und `m` legen die Farbe und Form der einzelnen Segmente in der Darstellung fest und können ebenfalls nach Belieben angepasst werden. Mögliche Werte können in der Dokumentation von `matplotlib` nachgelesen werden.

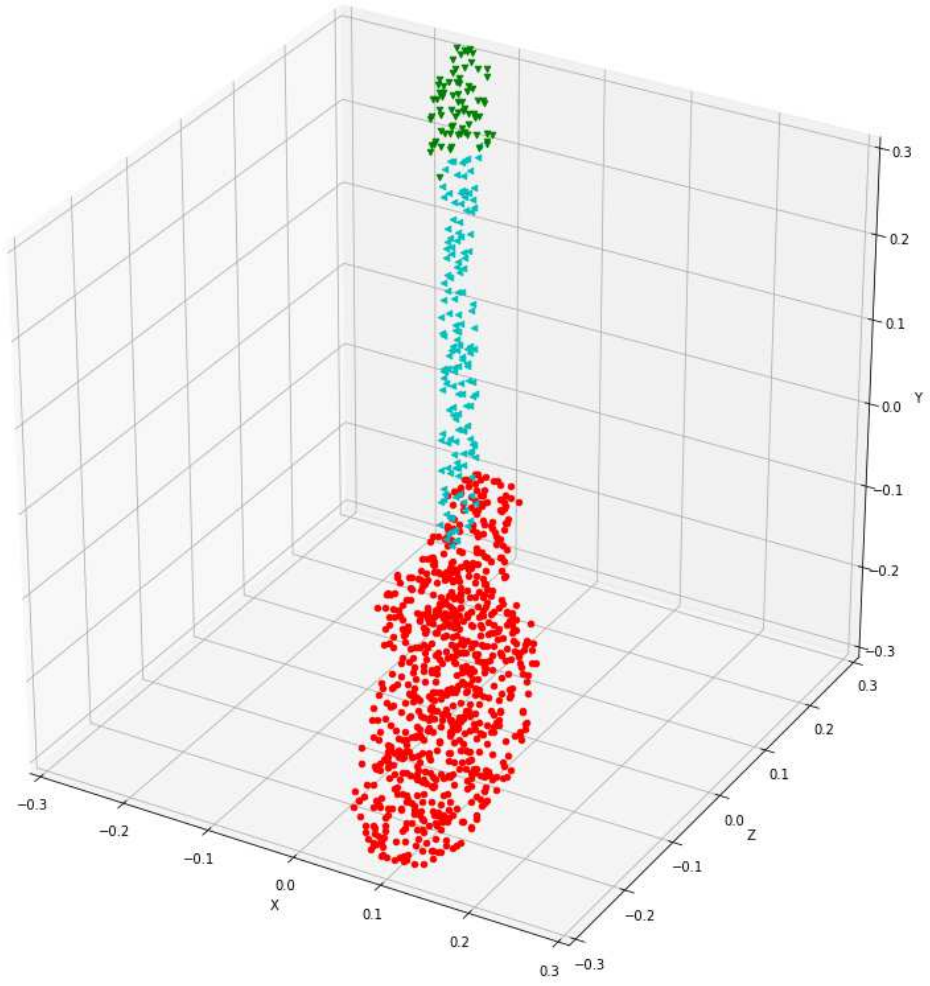
Über die Variable `d_num` kann auf die einzelnen Objekte des Testdatensatzes zugegriffen werden. Die entsprechenden Punkte werden extrahiert und an das Netzwerk übergeben. Der Output des Netzwerks wird in der Variable `pred` gespeichert und in eine Liste konvertiert.

Die `squeeze`-Funktion eliminiert atomare Dimensionen, um das Durchlaufen des Datensatzes zu vereinfachen.

Nachdem die Daten ausgewählt und in eine passende Form gebracht wurden, werden die einzelnen Punkte über eine for-Schleife als Scatter-Plot visualisiert.

[24]:
```python
# initialize plot
fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(111, projection='3d')

# set marker style and color
color = ['r', 'g', 'c', 'y', 'm']
m= ['o', 'v', '<', '>', 's']
```

```
# select test data to visualize
d_num = 1

v_points = test_points_r[d_num:d_num+1,:,:]
pred = model.predict(v_points)
pred = np.squeeze(pred)
v_points = np.squeeze(v_points)
pred = pred.tolist()

# add data to plot
for i in range(v_points.shape[0]):
    xs = v_points[i,0]
    ys = v_points[i,1]
    zs = v_points[i,2]
    ind = pred[i].index(max(pred[i]))
    ax.scatter(xs, zs, ys, c=color[ind], marker=m[ind])

# set axis labels and limits
ax.set_xlim3d(-0.3,0.3)
ax.set_ylim3d(-0.3,0.3)
ax.set_zlim3d(-0.3,0.3)

ax.set_xlabel('X')
ax.set_ylabel('Z')
ax.set_zlabel('Y')

plt.show()
```



Quellen:

[1] L. Yi et al., "A Scalable Active Framework for Region Annotation in 3D Shape Collections," SIGGRAPH Asia, 2016. [Online]. Available: http://web.stanford.edu/~ericyi/project_page/part_annotation/index.html

[2] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," CoRR, vol. abs/1612.00593, 2016, [Online]. Available: http://arxiv.org/abs/1612.00593.